MAD1 and MAD2: the complete guide

Every lesson, the viva kit, the cheat sheets and the glossary on one page. Use your browser's Print command to save it as a PDF, or download it: PDF · Markdown.

MAD1: Modern Application Development I

1. Week 1: How the web works: clients, servers, URLs and HTTP
2. Week 2: HTML and CSS: building and styling pages
3. Week 3: Views: Jinja2 templates, Bootstrap and forms
4. Week 4: Models: databases, SQL, ER diagrams and SQLAlchemy
5. Week 5: Controllers: Flask routes and business logic
6. Week 6: APIs and REST: OpenAPI and Flask-RESTful
7. Week 7: Backend systems: indexing, search, caching and logging
8. Week 8: The frontend: JavaScript, the DOM and fetch
9. Week 9: Security: authentication, sessions, tokens and attacks
10. Week 10: Testing web applications with pytest
11. Week 11: HTML5, accessibility, encoding and beyond HTML
12. Week 12: Deployment: Git, WSGI, servers and the cloud

MAD2: Modern Application Development II

1. Week 1: JavaScript basics: values, functions and the DOM
2. Week 2: Advanced JavaScript: closures, promises and async/await
3. Week 3: The web frontend: SPAs, browser storage and tooling
4. Week 4: Vue basics: instances, directives and reactivity
5. Week 5: Vue with APIs: fetch, axios and lifecycle
6. Week 6: Advanced Vue: components, props, events and Vue Router
7. Week 7: State management with Vuex
8. Week 8: Token authentication, API design, CORS and cookies
9. Week 9: Asynchronous jobs: Redis and Celery
10. Week 10: Messaging: webhooks, server-sent events and email
11. Week 11: Performance: measuring and caching
12. Week 12: The project: putting it all together

MAD1 + MAD2: the full-stack map

Project viva guide

MAD1 viva question bank

MAD2 viva question bank

MAD1 cheat sheet

MAD2 cheat sheet

Glossary

MAD1 week 1: How the web works: clients, servers, URLs and HTTP

> **Official syllabus line:** Basic terminologies of Web

The one-minute story

THINK OF IT LIKE THIS

The web is a restaurant. You (the **client**, in practice your browser) sit at a table and hand over an order slip: "one plate of `/menu/pizza`, please". The kitchen (the **server**) reads the slip, cooks, and sends the plate back with a note on top: "served" (200), "we do not have that" (404), or "kitchen on fire" (500).

The roads between the table and the kitchen are the **internet**. The restaurant itself, with its menu pages and plates, is the **web**. The kitchen has an **address** (IP address), a **phone book entry** that turns the restaurant name into that address (DNS), and several **doors** for different kinds of visitors (ports).

The waiter has a bad memory: every time you speak, he treats you like a stranger. That is why **HTTP is stateless** and why websites hand you a little **cookie** token to prove who you are.

By the end of this week you can:

- **explain** what happens between typing a URL and seeing a page;
- **split** any URL into scheme, host, port, path, query string and fragment;
- **read** an HTTP request and response and name the method, headers, body and status code;
- **choose** the right HTTP method and status code for a situation;
- **calculate** requests per second, bandwidth (Mbps) and RAM needs, which the papers love;
- **predict** what a tiny Flask route returns for a given URL (200, 404, 405 or 400).

Words you need

WORDS YOU NEED

- **Internet** - the global network of connected computers (the roads).
- **Web (WWW)** - the collection of pages and apps you reach over the internet using HTTP (the shops on the roads).
- **Client** - the program that asks for something (browser, mobile app, `curl`).
- **Server** - the program (and the machine) that waits for requests and answers them.
- **IP address** - the number address of a machine, like `142.250.77.46` (IPv4) or `::1` (IPv6). `127.0.0.1` means "this very computer".
- **DNS** - the Domain Name System, the phone book that turns `example.com` into an IP address.
- **Port** - a numbered door on a machine. Web uses 80 (http) and 443 (https). Flask's dev server uses 5000.
- **URL** - the full address of a resource on the web.
- **HTTP** - the set of rules (protocol) clients and servers use to talk. HTTPS is HTTP with encryption.
- **Request / response** - one message from the client, one message back from the server.
- **Status code** - a three-digit number in the response saying how it went.
- **Stateless** - the server keeps no memory between two requests.
- **Cookie** - a small piece of text the server asks the browser to store and send back.
- **Static / dynamic page** - a ready-made file versus a page built by code on each request.
- **Bandwidth / latency** - how much data per second a link carries / how long one trip takes.

Idea 1: Internet, web, clients and servers

The **internet** is the physical and logical network (cables, routers, IP). The **web** is one service that runs on it. Email and file transfer also use the internet but are not "the web".

A **client** starts the conversation and a **server** answers. These are roles, not machines: your laptop can run a Flask server and a browser at the same time, and then talk to itself.

What happens when you type `http://example.com/hello` and press Enter?

1. The browser asks **DNS** for the IP address of `example.com`.
2. It opens a connection to that IP address on the right **port** (80 for http, 443 for https).
3. It sends an **HTTP request**: "GET /hello".
4. The **server** runs some code (or reads a file) and sends an **HTTP response**: status, headers, body.
5. The browser reads the HTML body, finds more files it needs (CSS, images, JavaScript), requests those too, and **renders** the page.

THE IDEA

One request, one response. HTTP is request-response: the client always speaks first. The server cannot push a page to the browser unasked.

Web server versus web application. A *web server* (Apache, Nginx) accepts connections and serves files. A *web application* (your Flask code) decides what to send. Flask ships with a small development server so you can test locally; real sites put it behind a stronger one.

Static versus dynamic.

	STATIC PAGE	DYNAMIC PAGE
Content	A fixed file on disk	Built by code on every request
Same for all users?	Yes	Can differ (your name, today's date, database rows)
Example	`about.html`	Your marks page
Needs a program?	No, only a file server	Yes (Flask, a database)

You can serve static files with one command in a folder that has `index.html`:

```
python -m http.server 8000
```

Now open `http://localhost:8000/` in the browser. Python sends `index.html` for `/`, and any other file by its path. Press Ctrl+C to stop. The server prints one line per request, for example `"GET /index.html HTTP/1.1" 200 -`.

YOUR TURN: 1. ROLES

You run `python app.py` (a Flask app on port 5000) and then open `http://127.0.0.1:5000` in Chrome on the same laptop. Who is the client, who is the server, and which IP address is used?

▸ **Need a hint?**

▸ **Show the full solution**

Idea 2: Anatomy of a URL

Take this URL:

```
https://shop.example.com:8443/books/list?genre=sci-fi&page=2#reviews
```

PART	VALUE	MEANING
Scheme	`https`	The protocol to use
Host	`shop.example.com`	Which machine (a domain name or an IP)

PART	VALUE	MEANING
Port	`8443`	Which door. Omitted means the default (80 for http, 443 for https)
Path	`/books/list`	Which resource on that server
Query string	`genre=sci-fi&page=2`	Extra `name=value` pairs, joined by `&`, after `?`
Fragment	`reviews`	A spot inside the page, handled by the browser only

> **THE IDEA**
>
> **The fragment (after `#`) is never sent to the server.** The browser uses it to scroll to an element with that `id`. Everything else reaches the server.

Python can split a URL for you, which is a safe way to check your answer:

```
from urllib.parse import urlparse, parse_qs

u = urlparse("https://shop.example.com:8443/books/list?genre=sci-fi&page=2#reviews")
print(u.scheme, u.hostname, u.port)
print(u.path)
print(parse_qs(u.query))
print(u.fragment)
```

Output:

```
https shop.example.com 8443
/books/list
{'genre': ['sci-fi'], 'page': ['2']}
reviews
```

Notice that every query value is a **string** (`'2'`, not `2`), and a key can repeat (`?tag=a&tag=b`), so `parse_qs` gives lists.

Path parameter versus query parameter. These are two different ways to send a value, and the exam tests the difference constantly. In Flask:

```
from flask import Flask, request
app = Flask(__name__)

@app.route("/greet/<name>")          # value lives in the PATH
```

```
def greet(name):
    return f"Hi {name}"

@app.route("/search")                # value lives in the QUERY STRING
def search():
    return "q=" + request.args.get("q", "nothing")
```

I ran this app. The results:

URL	RESULT
`/greet/Ria`	`Hi Ria` (200)
`/greet/`	**Not Found** (404): the path variable is missing
`/greet?name=Ria`	**Not Found** (404): wrong place, it is not a query route
`/search`	`q=nothing`
`/search?q=flask`	`q=flask`
`/search?q`	`q=` (the key is present with an empty value, so the default is NOT used)

WATCH OUT - REAL EXAM TRAP

- `<name>` in the route is a **path** value; `request.args` reads the **query string**. They never replace each other.
- `/greet/` with nothing after the slash does not match `/greet/<name>`. Result: 404, not "Hi None".
- `request.args.get("q", "default")` returns the default only when the key is **missing**; for `?q` the key exists with value `""`.
- `<int:number>` accepts only whole numbers. `/validate/abc` for `<int:number>` is a 404, not a 400.

Idea 3: HTTP request and response

An HTTP message is plain text. A **request** has a start line, headers, an empty line, and an optional body:

```
POST /login HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Cookie: session=abc123

uname=ria&pwd=secret
```

A **response** looks similar, with a **status line** first:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Set-Cookie: theme=dark; Path=/
Content-Length: 31

<h1>Welcome, ria</h1>
```

PIECE	REQUEST	RESPONSE
First line	method, path, version	version, status code, reason
Headers	`Host`, `Content-Type`, `Cookie`, `Accept`, `User-Agent`	`Content-Type`, `Set-Cookie`, `Location`, `Content-Length`
Body	form data or JSON (POST/PUT/PATCH)	the HTML, JSON or file

`Content-Type` is the label on the parcel. `text/html` means "show it as a page"; `application/json` means "this is data". Flask's `jsonify(...)` returns status 200 and the mimetype `application/json`, which is a favourite exam fact.

Try it with `curl`, a command-line client (`-i` shows the headers, `-X` picks the method, `-H` adds a header, `-d` sends a body):

```
curl -i http://127.0.0.1:5000/api/courses
curl -X POST http://127.0.0.1:5000/api/course -H "Content-Type: application/json" -d
'{"id": "CS1001"}'
```

HTTP methods

METHOD	MEANING	HAS A BODY?	SAFE (DOES NOT CHANGE DATA)?	TYPICAL USE
GET	Read a resource	No	Yes	Open a page, fetch a list
POST	Create something / submit a form	Yes	No	Sign up, add a row
PUT	Replace a whole resource	Yes	No	Overwrite a record
PATCH	Change part of a resource	Yes	No	Update one field

METHOD	MEANING	HAS A BODY?	SAFE (DOES NOT CHANGE DATA)?	TYPICAL USE
DELETE	Remove a resource	Usually no	No	Delete a record
HEAD	Like GET but **headers only**	No	Yes	Check if a file exists

WATCH OUT - REAL EXAM TRAP

- GET is **not** the same as HEAD. GET returns headers and body; HEAD returns the headers only.
- A browser address bar always sends **GET**. You cannot send POST by typing a URL.
- Putting secrets in a GET query string is bad: URLs are stored in history and logs.
- GET responses may be **cached** by the browser; POST responses are not cached by default. "Browsers cache all GET and POST requests" is false.

Status code families

FAMILY	MEANING	CODES YOU MUST KNOW
1xx	Informational	100 Continue
2xx	**Success**	200 OK, 201 Created, 204 No Content
3xx	**Redirection**	301 Moved Permanently, 302 Found (temporary), 304 Not Modified
4xx	**Client error** (your request is wrong)	400 Bad Request, 401 Unauthorized (not logged in), 403 Forbidden (logged in but not allowed), 404 Not Found, 405 Method Not Allowed
5xx	**Server error** (the server's code failed)	500 Internal Server Error, 503 Service Unavailable

I checked these in Flask: `redirect(url_for("home"))` gives **302** with a `Location: /` header; `abort(403)` gives **403**; a missing route gives **404**; calling a POST-only route with GET gives **405**; returning `"saved", 201` gives **201**.

SOLVED EXAMPLE: 1. PREDICT THE STATUS AND THE PAGE

Question. Flask app, running on `http://127.0.0.1:5000` :

```
from flask import Flask, abort

app = Flask(__name__)
users = ["admin", "user_1"]

@app.route("/home/<username>", methods=["POST"])
def home(username):
    if username in users:
        return f"Hello {username}"
    abort(407)

@app.errorhandler(404)
def nf(e):
    return "The user you are looking for is invalid."

@app.errorhandler(405)
def bad_method(e):
    return "Please check the web request."
```

What does the browser show for `http://127.0.0.1:5000/home/user_4` ?

Solution.

1. The URL matches the pattern `/home/<username>` , so it is not a 404.
2. The address bar sends **GET**, but the route allows only **POST**. Flask answers 405 *before* the function runs.
3. The 405 error handler returns its text.

Answer: **Please check the web request.** (If you had guessed "invalid user" you ignored the method; the function body never ran, so `abort(407)` is never reached.)

`abort` and `errorhandler`. `abort(400, "message")` stops the function and sends that error. `@app.errorhandler(404)` replaces the default error page for that code. When the argument is missing, Flask raises 400 for `request.args["uid"]` (dictionary-style access) but returns `None` for `request.args.get("uid")` .

SOLVED EXAMPLE: 2. A LOGIN CHECK THAT FAILS FOR THE SECOND USER

Question. What does `/signin?uid=professor&pwd=54321` print?

```
from flask import Flask, request
valid = [{"uid": "student", "pwd": "12345"},
         {"uid": "professor", "pwd": "54321"}]
app = Flask(__name__)

@app.route("/signin")
def login():
    args = request.args
    for vu in valid:
        if vu["uid"] == args["uid"] and vu["pwd"] == args["pwd"]:
            return "You are authorized!!"
        else:
            return "You are unauthorized!!"
```

Solution. `return` ends the function at the first loop pass. The first user is `student`, which does not match `professor`, so the `else` runs and returns **You are unauthorized!!** The second user is never checked. (I ran this: it prints exactly that.) Without the query string at all, `args["uid"]` fails and the browser shows 400 Bad Request.

IN YOUR PROJECT

Your project must use the right status codes: 200 for a normal page, 404 when an id does not exist (use `abort(404)`), 400 or 403 for bad or forbidden actions. In the viva, expect "what does your app return for an unknown id?" and "why POST for this form and not GET?".

Idea 4: Stateless HTTP and cookies

Each request is independent: the server does not remember that you logged in one second ago. To fake memory, the server sends a **cookie** in a `Set-Cookie` response header. The browser stores it and sends it back in a `Cookie` header on every later request to that site. The cookie usually holds a short **session id**, and the real data stays on the server.

```
from flask import Flask, make_response, request
app = Flask(__name__)

@app.route("/set")
def set_it():
    r = make_response("ok")
    r.set_cookie("theme", "dark")
```

```
    return r

@app.route("/get")
def get_it():
    return request.cookies.get("theme", "none")
```

The first route sends `Set-Cookie: theme=dark; Path=/`. After the browser has that cookie, `/get` shows `dark`; in a fresh browser it shows `none`.

> **THE IDEA**
>
> **Stateless** means the *protocol* has no memory. Cookies (and server-side sessions) are the add-on that gives websites memory. Never store a password in a cookie.

Idea 5: How a browser renders and what developer tools show

After the HTML arrives the browser **parses** it into a tree called the **DOM** (Document Object Model), applies the CSS to each node, works out the layout and paints the pixels. Each new file it discovers (image, stylesheet, script) is another request.

Press **F12** (or right-click, Inspect) to open the developer tools:

- **Elements tab**: the live DOM and the CSS applied to the selected element. You can edit text and styles to experiment; the change lives only in your browser and vanishes on reload.
- **Network tab**: one row per request, with method, URL, **status code**, type, size and time. Click a row to see request and response headers. This is where you check that your form sent POST and what status came back.
- **Console tab**: JavaScript messages and errors.

View Source (Ctrl+U) shows the HTML the server sent; Elements shows the page after the browser and JavaScript have worked on it.

Server-side versus client-side rendering. With Jinja2 (week 3) the server builds the full HTML and the browser just shows it. That gives a fast first view and works well with search engines. In client-side rendering the server sends data (JSON) and JavaScript builds the page in the browser (that is the MAD2 way, with Vue). Jinja2 server-side rendering does **not** remove the need for later fetches when the user interacts; it just makes the first load simple.

Idea 6: Bandwidth, latency and counting requests

Papers set small arithmetic problems. Memorise these facts and units.

THE IDEA

- **Bandwidth** = bits per second (Mbps = million bits/s, Gbps = 1000 Mbps). **1 byte = 8 bits.**
- **Latency** = time for one round trip (milliseconds), mostly set by distance and the number of steps.
- **Transfer time** = size in bits divided by bandwidth in bits per second.
- **Throughput needed** = requests per second times size of one response.
- **Capacity** = (requests one worker handles per second) times (number of workers). Requests per second = 1 divided by the time per request in seconds.

SOLVED EXAMPLE: 3. THROUGHPUT

Question. An app serves 10,000 requests per second, each response is 50 KB. How much network speed is needed? A 10 Gbps link: enough?

Solution. Total data per second = 10,000 x 50 KB = 500,000 KB = 500 MB per second. In bits: 500 MB x 8 = 4,000 Mb = **4 Gbps**. A 10 Gbps link is enough. Because the link and the CPU are not the problem, this single server only needs a modest upgrade: the exam answer is **scale up** (a bigger machine), not scale out (more machines).

SOLVED EXAMPLE: 4. REQUESTS PER SECOND AND DOWNLOAD TIME

Question. (a) One worker takes 20 ms per request. How many requests per second can 4 workers handle? (b) A 25 MB file is downloaded on a 100 Mbps line. How long does it take?

Solution. (a) One worker: 1 / 0.020 = 50 requests per second. Four workers: **200** per second. (b) 25 MB = 25 x 8 = 200 Mb. Time = 200 / 100 = **2 seconds** (plus latency).

Memory and concurrency. If each connected viewer needs 3 GB and at most 1,900 viewers are online at the busiest minute, the server must hold 3 GB x 1,900 = 5,700 GB = **5.7 TB** (the real paper reads the peak from a graph and asks exactly this; use the **highest** point, not the average, because everyone may be there at once). Watch the unit: 1,900 x 3 GB is in GB, so convert before choosing GB or TB.

Scale up versus scale out. *Scale up* (vertical) means a stronger machine. *Scale out* (horizontal) means more machines behind a **load balancer**. Scale out when one machine's CPU, memory or network is the bottleneck; scale up when one machine has room and a bigger one is simpler. Caching (keeping a copy of a response near the client) reduces load and latency; it never "degrades service quality".

Cloud service models (a small recurring question): **IaaS** gives you raw machines and networks; **PaaS** gives a ready platform (language runtime, web server, database) so you only bring your code; **SaaS** gives a finished application (Gmail). A developer who wants a ready environment without managing servers wants **PaaS**.

Idea 7: MVC in one page

Most web apps split their code in three, so each part has one job:

PART	JOB	IN THIS COURSE
Model	The data and rules (tables, queries)	SQLAlchemy classes (later weeks)
View	What the user sees	HTML templates with Jinja2 (week 3)
Controller	Receives the request, uses the model, picks the view	Flask route functions

A request flows: browser, controller (route), model (data), back to the controller, view (template), response. Keep SQL out of templates and HTML out of models.

YOUR TURN: 2. SPLIT THE URL

For `http://localhost:5000/user/42/orders?sort=date&page=3#top` : name the scheme, host, port, path, query string and fragment. Which of these does the server never see?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 3. PICK THE STATUS CODE

Which status code fits: (a) a new student record was created, (b) the student id in the URL does not exist, (c) the user is logged in but is not an admin, (d) the server crashed because of a bug, (e) the page has moved for good to a new URL?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 4. BANDWIDTH

A page needs 600 requests per second and each response is 250 KB. What link speed in Mbps is the minimum?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 5. WHICH FLASK URL?

`@app.route("/course/<name>")` with `request.args.get("id")` inside. Which URL works: `/course/MAD1?id=CS2003`, `/course?name=MAD1&id=CS2003`, `/MAD1?id=CS2003`?

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

The graded questions are mostly **Quiz 1 and Quiz 2**, and for this first week the keyword match (about 235 of 692 Quiz 1, about 252 of 624 Quiz 2, about 477 of 1600 End Term questions) is broad, because almost every Flask question touches URLs, requests and status codes. What the real papers actually ask:

- **Predict what the browser shows** for a URL against a small Flask app: path variable versus query string, a missing value (404), `abort(400)` for a bad value, a custom `errorhandler`. These repeat across several years.
- **Method and status code facts**: "GET is the same as HEAD" (false), which method can be cached, 404 versus 405 versus 400, the mimetype and status of `jsonify`.
- **Back-of-envelope calculations**: requests per second times response size gives Gbps (the 10,000 x 50 KB = 4 Gbps one); viewers x memory gives peak RAM (the 3 GB video question); scale up versus scale out.
- **Cloud and rendering concepts**: PaaS versus IaaS versus SaaS; why server-side rendering gives a faster first load.
- **curl commands** for a JSON API: `-X POST`, `-H 'Content-Type: application/json'`, `-d '{ ... }'`. Without the JSON header, the request is not read as JSON.

WATCH OUT - REAL EXAM TRAP

- A browser address bar is always **GET**. A POST-only route visited in the browser gives **405**, not the function's output.
- `<int:x>` routes: a non-number is **404**; a number is accepted.
- `/route/` with a missing path variable is **404**; `?uname` with no value gives an empty string, not the default.
- `return` inside a loop ends the function at the first pass.
- 401 means not logged in, 403 means not allowed; 4xx is the client's fault, 5xx the server's.
- Bits versus bytes: network speed is in **bits**, file sizes are in **bytes**. Multiply by 8.

Quick quiz

QUICK QUIZ

A user types `http://127.0.0.1:5000/home/user_1` in the browser, where `/home/<username>` allows only POST. What status does the server return?

- 200
- 404
- 405
- 500

Check

Which part of a URL is NOT sent to the server?

- the path
- the query string
- the fragment
- the port

Check

For `@app.route("/greet/<name>")`, what does `http://127.0.0.1:5000/greet/` return?

- Hi None
- Hi (blank name)
- 400 Bad Request
- 404 Not Found

Check

Select all TRUE statements. (select all that apply)

- HTTP is stateless; cookies are used to remember a user.
- A 5xx status code means the server had a problem.

GET and HEAD return exactly the same response.

POST requests are cached by browsers by default.

Check

A service handles 10,000 requests per second, each response 50 KB. The network speed needed is about...

0.5 Gbps

40 Gbps

4 Gbps

400 Mbps

Check

A developer wants a ready runtime, web server and database, and only wants to deploy code. Which cloud model?

SaaS

PaaS

IaaS

DNS

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- **What happens when you type a URL in a browser?**
- **What is the difference between GET and POST, and which does your form use?**
- **Why do websites need cookies if HTTP already works?**
- **Explain 401, 403 and 404.**
- **Which of your routes return which status codes?**
- **What is MVC and where is it in your project?**
- **How would you check what your browser actually sent?**

Recap

REMEMBER THIS

- The **internet** is the network; the **web** is HTTP pages on it. Clients ask, servers answer, one request gets one response.
- A URL has scheme, host, port, path, query string and fragment. The fragment stays in the browser.
- A request has method, headers and maybe a body; a response has a status line, headers and a body.
- Methods: GET read, POST create, PUT replace, PATCH change part, DELETE remove, HEAD headers only.
- Status families: 2xx fine, 3xx redirect, 4xx your fault, 5xx server's fault.
- HTTP is **stateless**; **cookies** carry a session id to give memory.
- Bandwidth is in bits (x 8 from bytes). Capacity is workers times requests per second. Use the **peak** for memory.
- MVC: model = data, view = template, controller = route function.

CODE AND FACTS TO KEEP

- Requests per second = 1 / (seconds per request); with `n` workers multiply by `n`
- Bits = bytes x 8; Gbps = 1000 Mbps; throughput = (requests per second) x (response size)
- Download time = size in megabits / speed in Mbps
- Defaults: http port 80, https 443, Flask 5000, `python -m http.server` 8000
- Flask: `request.args.get("k")` (query), `<int:id>` (path), `abort(404)`, `jsonify(x)` gives 200 and `application/json`
- Status: 200 OK, 201 Created, 301/302 redirect, 400 Bad Request, 401 not logged in, 403 forbidden, 404 not found, 405 wrong method, 500 server error
- curl: `curl -i URL`, `curl -X POST URL -H "Content-Type: application/json" -d '{"a": 1}'`

MAD1 week 2: HTML and CSS: building and styling pages

> **Official syllabus line:** Webpages written in HTML and CSS

The one-minute story

THINK OF IT LIKE THIS

A web page is a house. HTML is the bricks and rooms: it says "this is a heading, this is a list, this is a door (link), this is a picture". **CSS** is the paint, furniture and layout: "make the walls blue, put the sofa on the left". Later, **JavaScript** is the electricity: lights that switch on when you press a button.

Keep the jobs apart. HTML says **what** something is. CSS says **how** it looks. If you paint with bricks (styling inside the HTML) you will suffer when you want to repaint the whole house.

Bootstrap is a ready-made furnished interior: you pick "blue button" and "three equal columns" from a catalogue instead of painting every wall yourself.

By the end of this week you can:

- **write** a complete HTML page with headings, lists, links, images, tables and forms;
- **tell** block elements from inline elements and use semantic tags;
- **write** CSS with the right selector and **predict** which rule wins (specificity, order);
- **calculate** the size of a box (content, padding, border, margin);
- **build** a simple layout with `display: flex`;
- **use** Bootstrap through its CDN link, with grid and utility classes.

Words you need

WORDS YOU NEED

- **HTML** - HyperText Markup Language: tags that describe the structure of a page.
- **Element** - an opening tag, content and a closing tag, such as `<p>Hello</p>`.
- **Attribute** - extra information inside the opening tag, such as `href="a.html"`.
- **CSS** - Cascading Style Sheets: rules that style elements.
- **Selector** - the part of a CSS rule that picks which elements to style.
- **Declaration** - one `property: value;` pair.
- **Specificity** - the score that decides which of two clashing rules wins.
- **Box model** - every element is a box: content, padding, border, margin.
- **Block / inline** - elements that start on a new line and take the full width / elements that sit inside a line of text.
- **Semantic tag** - a tag whose name says its meaning (`nav`, `footer`) rather than just its looks.
- **Pseudo-class / pseudo-element** - a selector for a state (`:hover`) / for a part of an element (`::before`).
- **Flexbox** - a CSS layout mode that arranges children in a row or column.
- **Bootstrap** - a CSS (and JS) library of ready-made classes and components.
- **CDN** - a public server that hosts libraries so you can link them by URL.

Idea 1: Creating and serving an HTML document

An HTML file is plain text. Save this as `index.html` in a new folder:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>My first page</title>
</head>
<body>
  <h1>Hello, web</h1>
  <p>This is my first page.</p>
</body>
</html>
```

Open a terminal in that folder and serve it:

```
python -m http.server 8000
```

Visit `http://localhost:8000/`. The server sends `index.html` for `/`. You can also just double-click the file, but then the address starts with `file://` and no server is involved.

PART	WHAT IT DOES
`<!DOCTYPE html>`	Tells the browser "this is modern HTML"
`<html lang="en">`	The root of everything; `lang` helps screen readers
`<head>`	Information about the page (not shown): title, character set, links to CSS
`<title>`	The text on the browser tab
`<body>`	Everything you see on the page

THE IDEA

The three languages of the front end. HTML defines the **content and structure**, CSS defines the **styling**, JavaScript adds **behaviour**. JavaScript is an add-on for extra features, not a required part of every page. A page may load a script from a file like this:

`<script src="app.js"></script>`

Scripts are linked with `src`, stylesheets with `<link rel="stylesheet" href="…">`, images with `<img src="…">`, and page links with `<a href="…">`.

Elements can be **nested** (a list inside a body) but must close in reverse order: `<p><b>Hi</b></p>`, never `<p><b>Hi</p></b>`. Some tags have no content and no closing tag (**void** elements): `<br>`, `<hr>`, `<img>`, `<input>`, `<meta>`, `<link>`.

Idea 2: Text, lists, links, images and tables

```
<h1>Main title</h1>            <!— h1 to h6, h1 is biggest —>
<h2>Section</h2>
<p>A paragraph with <b>bold</b>, <i>italic</i> and <br> a line break.</p>
<hr>                           <!— a horizontal line —>

<ul>                           <!— unordered: bullets, order does not matter —>
  <li>Rice</li>
  <li>Dal</li>
</ul>
<ol>                           <!— ordered: numbers, order matters —>
  <li>Wash</li>
  <li>Boil</li>
</ol>

<a href="https://example.com">An external link</a>
<a href="page2.html">A link to another file</a>
<a href="#reviews">Jump to the element with id reviews</a>
```

```
<img src="cat.png" alt="A sleeping cat" width="200">
```

- **Lists.** `ul` has bullets, `ol` has numbers, and both hold `li` items. A recipe needs `ul` for ingredients (order irrelevant) and `ol` for the steps (order matters). This exact choice appeared in a past paper.
- **Links.** The `href` value can be a full URL, a relative file path, or `#id` to jump inside the page (that is the fragment from week 1).
- **Images.** `src` is the file, `alt` is the text for screen readers and for when the image fails to load.

Tables

```
<table border="1">
  <thead>
    <tr><th>Course</th><th>Credits</th></tr>
  </thead>
  <tbody>
    <tr><td>MAD1</td><td>4</td></tr>
    <tr><td>DBMS</td><td>4</td></tr>
  </tbody>
</table>
```

`tr` is a row, `th` a header cell, `td` a data cell. A cell can span several columns with `colspan="2"` or rows with `rowspan="2"`.

Block and inline elements

	BLOCK	INLINE
Starts a new line?	Yes	No
Width	Full line by default	Only as wide as its content
Can you set width/height?	Yes	No
Examples	`div`, `p`, `h1` - `h6`, `ul`, `li`, `table`, `form`	`span`, `a`, `b`, `i`, `img`, `input`, `button`

Use `div` (block) and `span` (inline) as neutral containers that carry no meaning of their own.

Semantic tags

`header`, `nav`, `main`, `section`, `article`, `aside`, `footer`. They look like plain boxes but say what the content is, which helps search engines, screen readers, and your own reading.

```
<body>
  <header><h1>Study Hub</h1></header>
  <nav><a href="/">Home</a> <a href="/about">About</a></nav>
  <main>
    <article><h2>Post</h2><p>Text … </p></article>
  </main>
  <footer>Copyright 2026</footer>
</body>
```

YOUR TURN: 1. BUILD A PAGE

Write a body that shows a heading "Menu", an unordered list of two dishes, and a link "Home" that goes to `index.html`.

▸ **Need a hint?**

▸ **Show the full solution**

Idea 3: Forms and input types

A **form** is how a browser sends data to the server. The `action` is the URL to send to, and `method` is GET or POST (week 1).

```
<form action="/signup" method="post">
  <label for="u">Name</label>
  <input type="text" id="u" name="uname" required>

  <input type="email" name="mail" placeholder="you@example.com">
  <input type="password" name="pwd" minlength="6">
  <input type="number" name="age" min="5" max="99">
  <input type="date" name="dob">

  <input type="radio" name="level" value="fd"> Foundation
  <input type="radio" name="level" value="dip"> Diploma
  <input type="checkbox" name="agree" value="yes"> I agree

  <select name="city">
    <option value="che">Chennai</option>
    <option value="blr">Bengaluru</option>
  </select>

  <textarea name="note" rows="3"></textarea>
```

```
  <button type="submit">Send</button>
</form>
```

INPUT TYPE	GIVES YOU
`text`, `password`	One line of text (password shows dots)
`email`, `url`, `number`, `date`	The same, plus built-in checks and a suitable keyboard
`radio`	Pick **one** in a group (same `name`)
`checkbox`	Tick any number
`submit` / `button type="submit"`	Sends the form
`hidden`	A value sent without being shown
`file`	Upload a file (needs `enctype="multipart/form-data"`)

THE IDEA

The `name` attribute is the key the server receives. Browsers send `name=value` pairs. An input with no `name` is **not sent at all**. The `id` is for CSS, JavaScript and `<label for="…">`; it is never what the server sees. In week 3 you read the values with `request.form["uname"]` (POST body) or `request.args["uname"]` (GET query string).

HTML5 validation. Attributes such as `required`, `type="email"`, `min`, `max`, `minlength` and `pattern` make the browser check the data **before** sending. This is **client-side validation**. It improves the experience but a user (or a script) can bypass it, so the server must validate again (week 3).

`:valid` / `:invalid` in CSS. A field with `required` and no text is invalid, so a rule like `input:valid { border: 2px solid red; }` does **not** colour an empty required field, and does colour a correctly filled one. Past papers have asked this.

WATCH OUT - REAL EXAM TRAP

- No `name` means the value is never sent.
- Radio buttons are grouped by the **same** `name`; different names make independent groups.
- A form with `method="get"` puts values in the URL; the default method is GET.
- Client-side validation can be bypassed, so it never replaces server-side checks.

Idea 4: Adding CSS: inline, internal, external

A CSS **rule** has a selector and declarations:

```
p {
  color: darkblue;
  font-size: 18px;
}
```

Three ways to attach CSS:

```
<!— 1. Inline: on one element —>
<p style="color: red;">Red text</p>

<!— 2. Internal: a style tag in head —>
<style>
  p { color: blue; }
</style>

<!— 3. External: a separate file, linked in head (best for real projects) —>
<link rel="stylesheet" href="style.css">
```

External CSS is shared by many pages and is cached by the browser. In Flask the file lives in the `static` folder and is linked with `url_for('static', filename='style.css')` (week 3).

The cascade: who wins?

When several rules target one element, the browser decides in this order:

1. `!important` beats everything else (use rarely).
2. Higher **specificity** wins.
3. If specificity ties, the rule that comes **later** wins.
4. Some properties (colour, font) are **inherited** from the parent when no rule applies.

Specificity score, compare the columns from left to right:

SELECTOR KIND	WEIGHT	EXAMPLE
Inline style	1000	`style="…"`
id	100	`#main`
class, attribute, pseudo-class	10	`.note`, `[type="text"]`, `:hover`
type (tag), pseudo-element	1	`p`, `::before`

SOLVED EXAMPLE: 1. WHICH COLOUR?

Question. What colour is the text?

```
<style>
  p            { color: blue; }
  .note        { color: green; }
  #first       { color: red; }
  p.note       { color: orange; }
</style>
<p id="first" class="note">Hello</p>
<p class="note">World</p>
```

Solution. First paragraph: scores are 1, 10, 100, 11. The id rule is highest: **red**. Second paragraph has no id: `p.note` (11) beats `.note` (10) and `p` (1): **orange**. If we added `style="color: black"` to the first paragraph, it would become **black** (inline = 1000), unless a rule used `!important`.

Idea 5: Selectors, all kinds

```
h1 { }                    /* type: every h1 */
.note { }                 /* class: class="note" */
#main { }                 /* id: id="main" (unique) */
* { }                     /* universal: every element */
h1, h2 { }                /* group: either */

div p { }                 /* descendant: any p anywhere inside a div */
div > p { }               /* child: p directly inside a div */
h1 + p { }                /* adjacent sibling: the p right after an h1 */
h1 ~ p { }                /* general sibling: all later p siblings of an h1 */

a:hover { }               /* pseudo-class: mouse is over */
li:nth-child(2) { }       /* li that is the 2nd child of its parent */
li:nth-child(odd) { }     /* 1st, 3rd, 5th ... (even: 2nd, 4th ...) */
li:first-child { }
input:focus { }
input:valid { }

p::before { content: "Note: "; }   /* pseudo-element: injected text before */
p::first-line { }

input[type="text"] { }    /* attribute equals */
[class~="out"] { }        /* attribute contains the WORD "out" in a space-separated
list */
a[href^="https"] { }      /* attribute starts with */
```

```
a[href$=".pdf"] { }       /* attribute ends with */
a[href*="example"] { }    /* attribute contains the text anywhere */
```

- **One colon** `:` is a pseudo-**class** (a state), **two colons** `::` is a pseudo-**element** (a part).
- `div p` goes at any depth; `div > p` only one level down.
- `[class~=out]` matches an element whose `class` list contains the word `out`. `[id~=out]` does the same for id, so it matches `id="out"`.

SOLVED EXAMPLE: 2. READ THE SELECTOR

Question. Which items turn red?

```
<style>
  ul > li:nth-child(even) { color: red; }
  li ~ li:last-child      { font-weight: bold; }
</style>
<ul>
  <li>A</li><li>B</li><li>C</li><li>D</li><li>E</li>
</ul>
```

Solution. `nth-child(even)` picks the 2nd and 4th: **B and D are red**. The second rule picks the last `li` that has an `li` sibling before it: **E is bold**. A, B, C and D are not bold.

SOLVED EXAMPLE: 3. ATTRIBUTE SELECTORS IN A PAST-PAPER STYLE

Question. With these two rules, what border does each paragraph get?

```
<style>
  [id~=out]    { border: 10px solid red; }
  [class~=out] { border: 5px solid yellow; }
</style>
<p id="out">One</p>
<p class="out">Two</p>
<p class="outer">Three</p>
```

Solution. One has the id, so it gets the **red** 10px border. Two has the class word `out`: **yellow** 5px border. Three has class `outer`, a different word, so it gets **no border**. The paragraph with the class does not get the id rule, and the headings get nothing.

YOUR TURN: 2. SPECIFICITY RACE

`div p {color: red}`, `.box p {color: blue}`, `p {color: green}`. HTML: `<div class="box"><p>Hi</p></div>`. What colour?

▸ **Need a hint?**

▸ **Show the full solution**

Idea 6: Box model, units and colours

Every element is a box made of four layers, from the inside out: **content**, **padding** (inside space), **border**, **margin** (outside space).

```
.card {
  width: 200px;
  padding: 20px;
  border: 5px solid black;
  margin: 10px;
}
```

With the default `box-sizing: content-box`, `width` is the **content** width:

- Visible box width = 200 + 20 + 20 (padding) + 5 + 5 (border) = **250px**.
- Space taken in the row = 250 + 10 + 10 (margin) = **270px**.

With `box-sizing: border-box` the 200px already includes padding and border: visible width **200px**; content shrinks to 150px. Bootstrap sets `border-box` everywhere.

Shorthand: `margin: 10px 20px;` means top/bottom 10, left/right 20. Four values go clockwise: top, right, bottom, left.

UNIT	MEANING
`px`	A fixed pixel
`%`	Percent of the parent
`em`	Times the current element's font size
`rem`	Times the root (`html`) font size, usually 16px
`vw`, `vh`	1 percent of the window width / height

Colours: names (`red`), hex (`#ff0000`, `#f00`), `rgb(255, 0, 0)`, `rgba(255, 0, 0, 0.5)` (the last number is opacity) and `hsl(0, 100%, 50%)`.

Display and position

- `display: block | inline | inline-block | none | flex`. `none` removes the element entirely. (`visibility: hidden` hides it but keeps the space.)
- `position`: `static` (default, normal flow), `relative` (shift from its normal place), `absolute` (placed against the nearest positioned ancestor), `fixed` (stays on the screen when scrolling), `sticky` (sticks after scrolling to a point). Offsets use `top`, `left`, `right`, `bottom`.

Flexbox basics

```
<style>
  .row { display: flex; justify-content: space-between; align-items: center; gap: 10px;
}
  .row div { flex: 1; }
</style>
<div class="row">
  <div>One</div><div>Two</div><div>Three</div>
</div>
```

Setting `display: flex` on the **parent** makes the children flex items laid out in a row.

PROPERTY (ON THE PARENT)	EFFECT
`flex-direction` (`row` or `column`)	Main axis direction (row is the default)
`justify-content`	Spacing along the **main** axis: `flex-start`, `center`, `space-between`, `space-around`
`align-items`	Alignment along the **cross** axis: `stretch`, `center`, `flex-start`
`flex-wrap: wrap`	Allow items to go to the next line
`gap`	Space between items

On a child, `flex: 1` makes children share the free space equally. Above, three equal columns.

YOUR TURN: 3. BOX SIZE

`div {width: 300px; padding: 10px; border: 2px solid; margin: 15px; box-sizing: border-box;}`. What is the visible width, and what space does it take in a row?

▸ **Need a hint?**

▸ **Show the full solution**

Idea 7: Bootstrap

Bootstrap gives ready classes. No install is needed: link it from a CDN in the `head` (and, only if you use components such as dropdowns or modals, add the script before the closing `body`).

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Bootstrap demo</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body>
  <div class="container">
    <h1 class="text-center mt-4">Courses</h1>
    <div class="row">
      <div class="col-12 col-md-6">
        <div class="alert alert-info">Left half on medium screens and up</div>
      </div>
      <div class="col-12 col-md-6">
        <button class="btn btn-primary">Enrol</button>
      </div>
    </div>
  </div>
  <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js">
</script>
</body>
</html>
```

- **Containers.** `container` is centred with a maximum width; `container-fluid` spans the whole width.
- **Grid.** A `row` holds **12 columns** per line. `col-md-6` means "take 6 of 12 columns (half) from the medium breakpoint upward; below that stack full width". Breakpoints: `sm` 576px, `md` 768px, `lg` 992px, `xl` 1200px. `col-4 col-4 col-4` makes three equal columns.
- **Utilities.** Small single-purpose classes: spacing `m-3`, `mt-2`, `px-4`, `p-0` (scale 0 to 5, where `3` is 1rem), text `text-center`, `fw-bold`, colour `text-danger`, `bg-light`, display `d-flex`, `d-none`, `justify-content-between`, `border`, `rounded`.
- **Components.** `btn btn-primary`, `card`, `navbar`, `table table-striped`, `form-control`, `alert alert-success`, `list-group`.

WATCH OUT - REAL EXAM TRAP

- Missing the `viewport` meta tag makes the responsive grid behave badly on phones.
- Columns in a row should add up to 12; extras wrap to the next line.
- `col-md-6` does nothing below 768px: items stack. Large-to-small is **mobile first**.
- Forgetting the CDN link means the classes do nothing and the page looks plain.

IN YOUR PROJECT

Use one `static/style.css` for your own styles and Bootstrap from a CDN for layout. In the viva be ready to show one `container` / `row` / `col` layout, explain `col-md-6`, and change a button colour or a column width live. Use a semantic structure (`nav`, `main`, `footer`) and `alt` text on images.

Exam radar

EXAM RADAR

The graded questions that touch this week's keywords are about **215 of 692 in Quiz 1, 128 of 624 in Quiz 2 and 252 of 1600 in the End Term** (a loose keyword match, since forms and templates are counted too). The questions that really test HTML and CSS are fewer and are very predictable:

- **Selectors and specificity:** predict which paragraph gets which border or colour (attribute selectors such as `[class~=out]`, `:nth-child`, descendant versus child).
- **Which tag?** `ul` versus `ol` for a recipe; the right way to link a script (`script` with `src`) or a stylesheet.
- **Forms and validation:** what `required` and `:valid` do; whether a form has client-side validation, server-side validation or both; the `name` attribute.
- **Statements:** "HTML defines content, CSS defines styling" (true); "JavaScript is always the core part of the frontend" (false).
- **Small JavaScript:** an `onclick` button or `JSON.parse` on a string and showing properties through `innerHTML`. Know only the basics: `getElementById`, `innerHTML`, `alert`.
- Box model and Bootstrap grid questions appear less often but are easy marks.

WATCH OUT - REAL EXAM TRAP

- The **class** word match `[class~=out]` is not the same as `[class*=out]` (`outer` matches `*=` but not `~=`).
- More specific beats later: an id rule above beats a class rule below.
- `div > p` is direct children only.
- `visibility: hidden` keeps the space; `display: none` does not.
- `ol` is numbered and ordered, `ul` has bullets.

Quick quiz

QUICK QUIZ

A recipe page needs ingredients (order does not matter) and cooking steps (order matters). Which tags?

- `ol` for ingredients, `ul` for steps
- `ol` for both
- `ul` for both
- `ul` for ingredients, `ol` for steps

Check

A page has `p {color: blue}` in a style tag, `.note {color: green}`, and `<p class="note" style="color: black">`. The text colour is...

- blue
- green
- black
- red

Check

Which selector picks only `li` elements that are direct children of a `ul`?

- `ul li`
- `ul > li`
- `ul ~ li`
- `ul + li`

Check

Select all TRUE statements. (select all that apply)

- HTML describes content and structure; CSS describes styling.
- An input without a `name` is not submitted with its form.
- JavaScript is always required for every web page.
- `:hover` is a pseudo-element.

Check

`width: 200px; padding: 20px; border: 5px solid; margin: 10px;` with `content-box`. What is the visible width of the box?

- 200px
- 220px
- 250px
- 270px

Check

In Bootstrap, `<div class="col-md-6">` means…

- half the width at all screen sizes
- half the row from the md breakpoint upward, full width below
- 6 pixels wide
- hide on medium screens

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- **What is the difference between HTML, CSS and JavaScript?**
- **How does the browser decide which CSS rule wins?**
- **What is the box model?**
- **Why do you use semantic tags and `alt` text?**
- **Explain client-side and server-side validation in your form.**
- **How did you do your layout?**
- **Where do your CSS files live in your Flask project?**

Recap

REMEMBER THIS

- HTML skeleton: `doctype`, `html`, `head` (title, meta, links), `body`. Serve with `python -m http.server`.
- Block elements start a new line; inline elements sit in a line. Use semantic tags.
- Forms send `name=value` pairs; `name` matters, `id` does not. HTML5 validation is client-side only.
- CSS comes inline, internal or external; the winner is decided by `!important`, specificity (id 100, class 10, tag 1, inline 1000), then order.
- Selectors: type, class, id, descendant, `>`, `+`, `~`, `:hover`, `:nth-child`, `::before`, `[attr]` forms.
- Box width = content + padding + border (+ margin for space). Flexbox: `display: flex` on the parent.
- Bootstrap: CDN link, `container`, `row`, 12-column `col-md-*`, utility classes, components.

CODE AND FACTS TO KEEP

- Skeleton: `<!DOCTYPE html><html><head><title></title></head><body></body></html>`
- Links: `<link rel="stylesheet" href="s.css">`, `<script src="a.js"></script>`, `<a href="">`, `<img src="" alt="">`
- Specificity: inline 1000, `#id` 100, `.class [attr] :pseudo-class` 10, `tag ::pseudo-element` 1
- Visible width (content-box) = width + 2 padding + 2 border; space = + 2 margin
- Flex: `display: flex; justify-content: ...; align-items: ...; gap: ...`
- Bootstrap 5 CSS: `https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css`
- Breakpoints: sm 576, md 768, lg 992, xl 1200; grid = 12 columns

MAD1 week 3: Views: Jinja2 templates, Bootstrap and forms

> **Official syllabus line:** Presentation layer - View

The one-minute story

THINK OF IT LIKE THIS

A template is a form letter. "Dear NAME, your marks in COURSE are MARKS." You do not write a new letter for every student. You keep one letter with blanks, and a clerk fills in the blanks from a list. The **template** is the letter, the **data** is the list, and **rendering** is the clerk filling the blanks.

A website does the same. The **controller** (a Flask route) collects the data. The **view** (a Jinja2 template) is an HTML page with blanks and a few instructions like "repeat this row for every student". Rendering produces plain HTML that the browser shows. When the visitor fills in a **form**, the browser posts the answers back to the controller.

By the end of this week you can:

- **write** a string with variables using f-strings and `string.Template`, and say how Jinja2 differs;
- **predict the output** of a Jinja2 template with variables, filters, loops, conditions and `set`;
- **build** a base layout with `extends`, `block`, `super()`, `include` and a macro;
- **render** a template from Flask with `render_template` and link CSS with `url_for('static', ...)`;
- **read** form data with `request.form` and `request.args` and validate it on the server;
- **explain** why Jinja2 escapes HTML and what `|safe` does.

Words you need

WORDS YOU NEED

- **Template** - a text or HTML file with placeholders and small instructions.
- **Rendering** - filling a template with data to produce the final text.
- **Jinja2** - the template engine Flask uses.
- **Expression** - `{{ ... }}` : print a value.
- **Statement** - `{% ... %}` : control flow such as `for` and `if` , prints nothing itself.
- **Filter** - a small function applied with a pipe, like `name|upper` .
- **Inheritance** - a child template reuses a base layout and fills its `block` slots.
- **Macro** - a template function you define once and call many times.
- **Autoescape** - Jinja2 turns `<` into `<` in values so user text cannot inject HTML.
- **Static file** - CSS, JavaScript or an image served as it is from the `static` folder.
- **request.form / request.args** - data from the POST body / from the URL query string.
- **Client-side / server-side validation** - checking input in the browser / in your Flask code.
- **PyHTML** - a Python library that builds HTML from nested function calls (it also appears in past papers).

Idea 1: Strings with blanks, the stepping stone

Before templates, Python itself can fill blanks. Compare three ways:

```
name, price = "Ria", 49.5

print(f"{name:>6}|{price:.2f}")             # f-string
print("{} owes {:.1f}".format(name, price)) # str.format
print("%s owes %d" % (name, 3))             # old % style

from string import Template
t = Template("The $color car is in $place.")
print(t.substitute(color="red", place="Goa"))
print(t.safe_substitute(color="red"))
try:
    t.substitute(color="red")
except KeyError as e:
    print("KeyError", e)
```

Output:

```
   Ria|49.50
Ria owes 49.5
Ria owes 3
The red car is in Goa.
```

```
The red car is in $place.
KeyError 'place'
```

- **f-string**: `f"…"` with `{expression}` and a format spec after a colon: `.2f` two decimals, `>6` right-align in width 6, `,` thousands separator.
- `string.Template`: placeholders are written `$name`. `substitute` raises **KeyError** if a value is missing; `safe_substitute` leaves the placeholder as it is.
- The `Template` from `string` and the `Template` from `jinja2` are **different tools** with different syntax. `string.Template` only substitutes simple `$names`: no loops, no conditions, no filters.

SOLVED EXAMPLE: 1. TWO TEMPLATING ENGINES IN A ROW

Question. What does this print?

```
from string import Template as T1
from jinja2 import Template as T2

temp = "A student needs to complete $c1 level to get to the {{c2}} level."
out = T1(temp).substitute({'c1': 'foundation', 'c2': 'diploma'})
print(out)
print(T2(out).render({'c2': 'diploma'}))
```

Solution. `T1` replaces only `$c1`. It does not understand `{{c2}}`, and the extra key `c2` in the dictionary is simply ignored. So the first line is `A student needs to complete foundation level to get to the {{c2}} level.` That string is then fed to Jinja2, which fills `{{c2}}`, so the second line is `A student needs to complete foundation level to get to the diploma level.`

If a `$` placeholder is left unfilled, `substitute` raises `KeyError: 'place'`, while `safe_substitute` keeps `$place`. I ran both.

WATCH OUT - REAL EXAM TRAP

- `substitute` fails (KeyError) on a missing key; `safe_substitute` does not.
- Extra keys are ignored by both.
- Two engines in a row: each only replaces its own syntax, and the other's syntax passes through untouched.
- In Jinja2 a missing variable prints as an empty string (not an error).

PyHTML is a third way: build HTML with Python calls, for example `html(head(title("Quiz 2")), body(h2("Heading"), p("Text")))` and `.render()` gives the nested HTML in the same order as the calls. Jinja2 is the one used in this course for real pages.

Idea 2: Jinja2 basics: variables, expressions, filters

Jinja2 works without Flask too, which is a good way to test a template:

```
from jinja2 import Template

t = Template("Hi {{ name }}, 2 x 3 = {{ 2 * 3 }}, you have {{ items|length }} items. {{
missing }}|")
print(t.render(name="Ria", items=[1, 2, 3]))
```

Output: `Hi Ria, 2 x 3 = 6, you have 3 items. |`

The missing variable printed nothing.

> THE IDEA
>
> **Three kinds of markers.**
>
> - `{{ expression }}` prints a value.
> - `{% statement %}` is logic (`for` , `if` , `set` , `block` ...) and prints nothing itself.
> - `{# comment #}` is removed from the output.

Passing data. `render(name="Ria")` and `render({'name': 'Ria'})` are the same. Values may be strings, numbers, lists, dictionaries or objects. Inside the template, `d.k` and `d['k']` both read a dictionary key, `l[0]` reads a list item, and **methods can be called**: if `loan1.get_loan_details()` returns `(101, 45900, 5)` then `{{ l1.get_loan_details() }}` prints `(101, 45900, 5)` .

Expressions work like Python: `{{ 7 // 2 }}` is `3` , `{{ 7 / 2 }}` is `3.5` , `{{ 'a' ~ 1 }}` is `a1` (`~` joins text), `{{ 'ab' * 2 }}` is `abab` , `{{ 'yes' if n > 1 else 'no' }}` .

Filters go after a pipe and can be chained:

- `{{ name|upper }}` gives `RIA` , `{{ 'ria dev'|title }}` gives `Ria Dev`
- `{{ name|default('guest') }}` gives `guest` when `name` is not defined
- `{{ items|length }}` is the size (the paper's right answer; `len(items)` and `items|len` are wrong)
- `{{ items|join(' / ') }}` , `{{ items|first }}` , `{{ items|last }}` , `{{ items|sort }}`
- `{{ price|round(1) }}` : `3.14159` becomes `3.1`
- `{{ items|sum }}` , `{{ items|max }}` , `{{ s|trim }}` , `{{ s|replace('a','z') }}`

Chain: `{{ ['b','a']|sort|reverse|list }}` gives `['b', 'a']` .

Autoescaping and `|safe`

If a user types `<script>` as their name, printing it raw would run their script on every visitor's page (an attack called XSS). So **Flask turns autoescape on for `.html` templates**: `<` becomes `<` , the browser shows the characters instead of treating them as a tag.

```
from flask import render_template_string
render_template_string("{{ s }} | {{ s|safe }}", s="<b>hi</b>")
```

Result: `<b>hi</b> | <b>hi</b>`. The first shows the literal text `<b>hi</b>` in the browser; the second makes real bold text.

THE IDEA

`|safe` **means "I trust this HTML, do not escape it".** Never use it on text a user typed. Note: a plain `jinja2.Template( ... )` used outside Flask does **not** escape by default (autoescape is off unless you ask), which is why `{{ s }}` printed `<b>hi</b>` unchanged in my console test.

SOLVED EXAMPLE: 2. A MISSING VARIABLE AND A METHOD CALL

Question. Predict the output.

```
from jinja2 import Template

template = '''
Welcome to {{Degree}} DEGREE
First level to clear is {{level}}
'''
print(Template(template).render(Degree="IITM BS"))
```

Solution. Only `Degree` is passed. `{{level}}` is undefined and prints nothing:

```
(blank line)
Welcome to IITM BS DEGREE
First level to clear is
```

Names are case sensitive: `{{ Level }}` would also be blank even if `level` had a value.

Idea 3: Control flow: `for`, `if`, `set`, whitespace

```
{% for c in courses %}
  <li>{{ loop.index }}. {{ c }}{% if loop.last %} (last){% endif %}</li>
{% else %}
  <li>No courses</li>
{% endfor %}
```

With `courses = ["MAD1", "MAD2", "DBMS"]` the items are `1. MAD1`, `2. MAD2`, `3. DBMS (last)`. The `else` of a `for` runs only when the list is **empty**.

LOOP VARIABLE	MEANING
`loop.index`	1, 2, 3 ... (starts at 1)
`loop.index0`	0, 1, 2 ...
`loop.first`, `loop.last`	True on the first / last pass
`loop.length`	Total passes
`loop.revindex`	Passes left, counting down to 1

Loop over a dictionary with `{% for k, v in d.items() %}{{ k }}={{ v }};{% endfor %}`, which prints `a=1;b=2;` for `d = {"a": 1, "b": 2}`. Use `range`: `{% for i in range(3) %}` runs for 0, 1, 2.

Conditions: `{% if n > 10 %}big{% elif n > 5 %}mid{% else %}small{% endif %}`; with `n = 7` this prints `mid`. You can use `and`, `or`, `not`, `in`, and tests such as `is defined`, `is even`.

`{% set %}` and its scope. `set` makes a variable. A `set` inside a `for` loop is **local to that loop** and is gone afterwards.

```
from jinja2 import Template
t = Template("""
{% for i in range(3) %}
  {% set x = i %}
{% endfor %}
[{{ x }}]
""")
print(t.render())
```

The output contains `[]`: the `x` outside the loop is undefined, so it prints **nothing** (not 0, not 2). A past paper asked exactly this twice, once with `range(3)` and once with `[1, 2, 3]` in a Flask template (answer: "Output is:" and nothing after it). To add up values across a loop, use `namespace`:

```
{% set ns = namespace(t=0) %}{% for i in [1,2,3] %}{% set ns.t = ns.t + i %}{% endfor %}
{{ ns.t }}
```

This prints `6`. A `set` outside any loop works normally: `{% set x = 5 %}[{{ x }}]` prints `[5]`.

Whitespace. Jinja2 keeps the newlines around your tags. A minus sign next to the tag brackets trims the whitespace on that side:

```
A = "<ul>\n{% for x in c %}\n<li>{{ x }}</li>\n{% endfor %}\n</ul>"
B = "<ul>\n{% for x in c -%}\n<li>{{ x }}</li>\n{%- endfor %}\n</ul>"
# c = ["a", "b"]
# A renders to '<ul>\n\n<li>a</li>\n\n<li>b</li>\n\n</ul>'
# B renders to '<ul>\n<li>a</li><li>b</li>\n</ul>'
```

The extra blank lines in `A` are harmless in HTML (the browser ignores them), so you will rarely need `-`. A past paper question puts `groupby` in a template:

```
temp = """
{% for vid, members in interfaces|groupby(attribute='lan') %}
 Interfaces in lan {{ vid }}: {{ members|map(attribute='name') | join(', ') }}
{% endfor %}
"""
```

With interfaces `Ethernet1 (50)`, `Ethernet2 (40)`, `Ethernet3 (50)`, `Ethernet4 (60)` the output is three lines: `lan 40: Ethernet2`, `lan 50: Ethernet1, Ethernet3`, `lan 60: Ethernet4`. `groupby` **sorts by the key** and groups equal keys together; `map(attribute='name')` takes each item's name.

YOUR TURN: 1. PREDICT THE LOOP

`{% for c in ['a','b','c'] %}{{ loop.index }}{{ c }}{% if not loop.last %}-{% endif %}{% endfor %}` prints what?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 2. SCOPE OF SET

What does `{% set t = 0 %}{% for i in [1,2,3] %}{% set t = t + i %}{% endfor %}{{ t }}` print?

▸ **Need a hint?**

▸ **Show the full solution**

Idea 4: Templates in Flask

```
myapp/
  app.py
  templates/
```

```
    base.html
    index.html
    form.html
  static/
    style.css
```

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html", courses=["MAD1", "MAD2"], user="Ria")
```

- `render_template` looks in a folder named **`templates`** next to `app.py`, and extra arguments become variables. If you create the app with `Flask(__name__, template_folder='my_templates')` Flask looks **only** in `my_templates`, so a file in `templates/` fails with `TemplateNotFound`. (This was the 5-mark question in a recent paper: routes using paths inside `my_templates` work, those asking for `index.html` or `page.html` that live elsewhere raise TemplateNotFound.) Subfolders are named in the call: `render_template('public/home.html')`.
- **Static files** (CSS, images, JS) go in `static/`. Link them with `url_for`:

```
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
<img src="{{ url_for('static', filename='logo.png') }}" alt="logo">
```

which renders as `/static/style.css`.

`url_for(function_name, **values)` builds a URL from the **name of the view function**, so changing the route later does not break your links:

```
@app.route("/home")
def index(): ...

@app.route("/student/<student_name>/<int:student_id>")
def profile(student_name, student_id): ...

with app.test_request_context():
    print(url_for("index"))                                      # /home
    print(url_for("profile", student_name="John", student_id=101))# /student/John/101
    print(url_for("index", page=2))                              # /home?page=2
```

Values that are part of the route fill the path; **extra values become the query string**. The argument is the function name `profile`, not the path. `redirect(url_for("index"))` sends the browser a **302** and a `Location` header, and the browser then makes a new GET (as in the 2022 paper's `index2` route).

Idea 5: Reuse: inheritance, `include`, macros

Inheritance. Put the common page frame in one base file and fill the gaps in each page.

```
<!— templates/base.html —>
<!doctype html>
<html>
<head>
  <title>{% block title %}Study Hub{% endblock %}</title>
  <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
</head>
<body>
  {% block content %}{% endblock %}
  {% block foot %}(c) Hub{% endblock %}
</body>
</html>
```

```
<!— templates/page.html —>
{% extends "base.html" %}
{% block title %}Home – {{ super() }}{% endblock %}
{% block content %}<h1>Hello</h1>{% endblock %}
```

Rendering `page.html`: the title becomes `Home – Study Hub` (`super()` pulls in the parent's block text), the content is `<h1>Hello</h1>`, and `foot` is not overridden so the parent's `(c) Hub` appears. I rendered a small version of this.

> **THE IDEA**
>
> **Rules of `extends`.** `{% extends "base.html" %}` must be the **first** tag in the child. In a child, only things inside `{% block name %}` … `{% endblock %}` are shown; anything outside blocks is ignored. A block is replaced by the child's version; `{{ super() }}` keeps the parent's text too. The right syntax is `{% extends "file" %}` and `{% block name %}` (a block needs a name).

Include inserts another file at that spot, as is: `{% include "navbar.html" %}`. It sees the same variables. Use it for small repeated pieces (navbar, footer).

Macros are functions in a template:

```
{% macro box(text, kind='info') %}
  <div class="alert alert-{{ kind }}">{{ text }}</div>
{% endmacro %}
```

```
{{ box('Saved') }}
{{ box('Failed', 'danger') }}
```

Output: `<div class="alert alert-info">Saved</div>` and `<div class="alert alert-danger">Failed</div>` (default argument used for the first). In the 2022 paper a macro with a `for` loop printed the courses **as page text, one after another**, `App1 App2 python Data science Database machine learning`, not as a Python list with brackets and quotes. A macro must be **called** (`{{ display(list) }}`) or nothing shows.

YOUR TURN: 3. INHERITANCE

`base.html` has `{% block content %}{% endblock %}` . A child contains `{% extends "base.html" %}{% block content %}<p>Hi</p>{% endblock %}` plus an extra line `<h1>Outside</h1>` before the block. What shows?

▸ **Need a hint?**

▸ **Show the full solution**

Idea 6: Forms: GET and POST into Flask

Template `form.html` :

```
{% extends "base.html" %}
{% block content %}
<form action="{{ url_for('signup') }}" method="post">
  <input type="text" name="uname" required>
  <input type="number" name="age" min="5">
  <button type="submit">Send</button>
</form>
{% if error %}<p class="text-danger">{{ error }}</p>{% endif %}
{% endblock %}
```

Route:

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route("/signup", methods=["GET", "POST"])
def signup():
    if request.method == "POST":
        name = request.form.get("uname", "").strip()
```

```
        age = request.form.get("age", "")
        if not name:
            return render_template("form.html", error="Name is required"), 400
        if not age.isdigit():
            return render_template("form.html", error="Age must be a number"), 400
        return f"Welcome {name}, age {int(age)}"
    return render_template("form.html")
```

Test results (I posted these with Flask's test client): `uname=Ria, age=20` returns `Welcome Ria, age 20`; an empty name returns the form again with the error and status **400**; `age=x` returns the second error. A plain GET to `/signup` shows the form.

	`REQUEST.FORM`	`REQUEST.ARGS`	`REQUEST.METHOD`
Holds	POST body fields	URL query string	`"GET"` or `"POST"`
Used with	`method="post"`	`method="get"` or a link with `?k=v`	choosing what to do

- A route accepts only **GET** unless you list others in `methods=[...]`. Forgetting `"POST"` makes the form submit fail with **405**.
- `request.form["k"]` raises **400 Bad Request** if the key is missing; `request.form.get("k")` returns `None`.
- A form with `method="get"` puts the fields in the query string, so you read them with `request.args`. Reading `request.form` for a GET form gives nothing.
- `request.args.getlist("tag")` returns all values for a repeated key: `?tag=a&tag=b` gives `['a', 'b']`.
- **Good practice:** after a successful POST, do `return redirect(url_for(...))` so a page refresh does not submit the form again.

Client-side versus server-side validation

- **Client-side**: HTML5 attributes (`required`, `type="email"`, `min`, `pattern`) in the browser. Fast and friendly, but anyone can bypass it (edit the HTML, use `curl`).
- **Server-side**: your Python checks. **Required** for security and correctness. The form above has both: the browser blocks an empty name, and `signup` re-checks.

Flask-WTF (with WTForms) is an extension that defines forms as Python classes and adds validators and CSRF protection; know its name and purpose, but plain `request.form` is what the course shows first.

WATCH OUT - REAL EXAM TRAP

- `request.form` is for **POST** form data; `request.args` is for **URL** parameters. Match them to the form's `method`.
- Missing `methods=["GET", "POST"]` gives 405 when the form posts.
- HTML validation alone is **not** safe; always validate on the server.
- For `request.args.get(...)` a missing key is `None`; for `request.args["..."]` it is a 400 error. The paper's login check `args["uid"]` fails with 400 when `uid` is absent.

Idea 7: Bootstrap inside templates

Put the CDN `link` once in `base.html`, and every page that extends it gets the styles. A loop turns data into a styled table:

```
<table class="table table-striped">
  <thead><tr><th>#</th><th>Course</th></tr></thead>
  <tbody>
  {% for c in courses %}
    <tr><td>{{ loop.index }}</td><td>{{ c }}</td></tr>
  {% endfor %}
  </tbody>
</table>
```

With two courses this produces two `tr` rows, numbered 1 and 2, inside a Bootstrap-striped table. Other useful pieces: a form field is `<input class="form-control" ...>` in a `<div class="mb-3">`, a submit button is `class="btn btn-primary"`, messages are `alert alert-danger`, and a top bar is a `navbar` (put it in its own file and `include` it).

Server-side rendering (what you do this week) sends finished HTML: fast first load and easy for search engines. In client-side rendering (MAD2) the server sends JSON and JavaScript builds the page. Neither removes the need for requests after the page loads.

SOLVED EXAMPLE: 3. TRACE A REQUEST

Question. The user opens `/hello/Ria`. Describe all steps and give the HTML, given:

```
@app.route("/hello/<name>")
def hello(name):
    return render_template("hello.html", n=name, tags=["a", "b"])
```

```
<p>Hi {{ n|upper }}!</p>
<ul>{% for t in tags %}<li>{{ t }}</li>{% endfor %}</ul>
```

Solution. Flask matches `/hello/<name>` and calls `hello("Ria")`. `render_template` finds `templates/hello.html` and renders it with `n="Ria"` and `tags`. The browser receives status 200 and the body `<p>Hi RIA!</p>` and `<ul><li>a</li><li>b</li></ul>`. The browser then parses and displays the HTML. That is the whole controller to view path.

IN YOUR PROJECT

Make one `base.html` with your Bootstrap link, navbar and `{% block content %}`, and let every page extend it. Never build HTML strings in Python; always use `render_template`. In the viva you may be asked to add a field to a form: be ready to change the HTML input, the route (`request.form`), validation, and the template that shows it.

YOUR TURN: 4. WHICH REQUEST READS WHAT?

A form has `method="get"` and an input `name="q"`. In the route, which expression reads the value: `request.form['q']` or `request.args['q']`?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 5. ESCAPE OR NOT

`render_template_string("{{ s }}", s="<i>x</i>")` in Flask: what does the page source contain? And with `{{ s|safe }}`?

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

This week mixes templates with Flask routing. The keyword counts are about **140 of 692 in Quiz 1, 45 of 624 in Quiz 2 and 252 of 1600 in the End Term** (loose, as the match also catches forms and routes). In the real questions:

- **"Predict the output of this Jinja2 / `string.Template` code"** appears again and again in Quiz 1: `substitute` versus `safe_substitute` (KeyError), `{{ }}` versus `$`, a missing variable printing blank, a method call in `{{ }}`, `groupby`, passing a dictionary to `render`.
- **`{% set %}` inside a loop** (about two to four questions in recent Quiz 2 papers): the value is not visible after the loop.
- **Template inheritance syntax** (`extends`, a named `block`), and **macros** (a loop printing text).
- **Syntax questions:** `{{ courses|length }}` for the count; matching PyHTML, Jinja2, template rendering and string template.
- **Flask view questions:** `url_for` output, `redirect`, `template_folder` and `TemplateNotFound`, `abort(404)` versus `render_template( ... ), 404`, `first_or_404()`.
- **Statement questions:** why server-side Jinja2 gives a faster first load and good SEO; front-end and back-end validation of a form.

WATCH OUT - REAL EXAM TRAP

- `{{ x }}` after a loop that did `set x` inside prints nothing.
- Autoescape is on in Flask `.html` templates, off in bare `jinja2.Template`.
- `url_for` takes the **function name**, not the URL; extra values go to the query string.
- Returning `render_template("404.html")` alone gives status **200**; to send a 404 you need `, 404` or `abort(404)`.
- In the child template, text outside blocks is ignored; `extends` goes first.
- Case matters in variable names: `{{ Level }}` is not `{{ level }}`.

Quick quiz

QUICK QUIZ

Which code throws a KeyError? `t = Template("The $color car is at the $building.")` (string.Template)

```
t.substitute({'color': 'red'})
```

```
t.safe_substitute({'color': 'green'})
```

```
t.substitute({'color': 'blue', 'building': 'office', 'place': 'Chennai'})
```

`t.safe_substitute({})`

Check

What is printed? `Template("{% for i in range(3) %}{% set x = i %}{% endfor %}[{{ x }}]").render()`

`[0]`

`[2]`

`[]`

an error

Check

Which is the correct way to show how many courses are in a list in Jinja2?

`{{ len(courses) }}`

`{{ courses|len }}`

`{{ courses|length }}`

`{{ courses.size }}`

Check

Select all TRUE statements. (select all that apply)

`{% extends "base.html" %}` must come first in the child template.

`{{ super() }}` inside a block includes the parent's block content.

Text outside blocks in a child template is shown.

`request.form` reads the URL query string.

Check

A Flask app is created with `Flask(__name__, template_folder='my_templates')`. A route renders `'index.html'`, which exists only in `templates/`. What happens?

- It renders normally
- Flask falls back to `templates/`
- TemplateNotFound error (a 500 page)
- 404 Not Found

Check

`render_template_string("{{ s }}", s="<b>x</b>")` in Flask returns…

- `<b>x</b>` as real bold text
- `<b>x</b>` which the browser shows as the characters
- an error
- an empty string

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- **What is a template engine and why use one?**
- **Explain `extends`, `block` and `include` in your project.**
- **What is autoescaping and why does it matter?**
- **How does form data reach your Flask code?**
- **Why validate on the server when the HTML already has `required`?**
- **What does `url_for('static', filename='style.css')` do?**
- **What is the difference between a macro and an include?**

Recap

REMEMBER THIS

- `f"..."`, `format` and `string.Template` fill blanks in Python; `substitute` raises KeyError for a missing key, `safe_substitute` does not.
- Jinja2: `{{ }}` prints, `{% %}` is logic, `{# #}` is a comment. Filters: `length`, `upper`, `default`, `join`, `round`.
- A missing variable is blank; `set` in a loop does not leak out; `for` has `loop.index`, `loop.first`, `loop.last` and an `else` for empty lists.
- Flask: `templates/` folder, `static/` folder, `render_template`, `url_for` (function names), `redirect`.
- Reuse with `extends` + `block` + `super()`, `include`, and macros.
- Forms: `methods=["GET", "POST"]`, `request.form` for POST, `request.args` for GET. Validate on the server; HTML validation is only a convenience. Flask-WTF is the extension that formalises this.
- Autoescape protects against injected HTML; `|safe` switches it off for trusted text.

CODE AND FACTS TO KEEP

- Render: `render_template("page.html", name=value)`
- Static link: `href="{{ url_for('static', filename='style.css') }}"`
- Loop: `{% for x in xs %}` … `{{ loop.index }}` … `{% else %}empty{% endfor %}`
- If: `{% if a %}` .. `{% elif b %}` .. `{% else %}` .. `{% endif %}`
- Inheritance: `{% extends "base.html" %}` and `{% block content %}` … `{{ super() }}{% endblock %}`
- Macro: `{% macro name(arg, opt='x') %}` … `{% endmacro %}` then `{{ name(1) }}`
- Whitespace trim: `{%-` … `-%}` ; comment: `{#` … `#}`
- Form data: `request.form.get("k")` , `request.args.get("k")` , `request.method`
- Count: `{{ items|length }}` ; default: `{{ x|default('n/a') }}`
- `string.Template` : `$name` , `substitute()` (KeyError) / `safe_substitute()`

MAD1 week 4: Models: databases, SQL, ER diagrams and SQLAlchemy

> **Official syllabus line:** Models - Introduction to databases

The one-minute story

THINK OF IT LIKE THIS

A database is a well-run school office. Student files are not thrown in one big heap on the floor. There is one register for students, one for courses and one for who joined which course. Each student has a roll number, so two students with the same name never get mixed up. A clerk (the **DBMS**) hands out files, stops two people from writing in the same register at once, and refuses a form with the roll number missing.

SQL is the language you use to talk to the clerk ("give me all students in MAD1, sorted by name"). **SQLAlchemy is a translator**: you speak Python objects (`Student(name="Asha")`) and it speaks SQL to the clerk for you.

This week is the **Model** in MVC: the data your app remembers.

By the end of this week you can:

- **draw** a small ER diagram and **turn** it into tables with primary and foreign keys;
- **write** SQL with `WHERE` , `GROUP BY` , `HAVING` , `ORDER BY` , `LIMIT` and joins, and **predict the output** of a query on a small table;
- **explain** NULL, constraints, normalisation, transactions and ACID;
- **write** SQLAlchemy and Flask-SQLAlchemy models and **predict** what `query` , `filter_by` , `commit` and `rollback` do;

- **fix** classic ORM mistakes (forgetting `commit`, using `and` in a filter, positional constructor arguments).

Words you need

WORDS YOU NEED

- **DBMS** - Database Management System: the software that stores, protects and queries data (SQLite, MySQL, PostgreSQL).
- **RDBMS** - a DBMS that stores data in related tables (the relational model).
- **Table, row, column** - a table is one kind of thing; a row (record, tuple) is one item; a column (field, attribute) is one property.
- **Primary key** - the column (or columns) that identifies a row uniquely and is never NULL.
- **Foreign key** - a column that holds the primary key of a row in another table; it creates the link.
- **Candidate key** - any column set that could serve as the primary key; the one you choose is the primary key. A **composite key** is a key made of two or more columns.
- **Entity, attribute, relationship** - the three parts of an ER diagram: a thing, its properties, and how things connect.
- **Cardinality** - how many on each side of a relationship: one-to-one, one-to-many, many-to-many.
- **Constraint** - a rule the database enforces: `NOT NULL`, `UNIQUE`, `DEFAULT`, `CHECK`, primary or foreign key.
- **Transaction** - a group of steps that either all happen or none happen.
- **ORM** - Object Relational Mapper: maps classes to tables, objects to rows, attributes to columns.
- **Session** - the ORM's "shopping basket" of pending changes until you `commit`.

Idea 1: Why a database, and the relational model

Why not just keep data in a text file or a Python list? Because files give you no help with **searching** fast, **sharing** between many users, **avoiding duplicates**, **protecting** against half-finished writes, and **access control**. A DBMS gives all of these. (A question in the papers asks which database fits a university with enrolments, prerequisites, grades, fee payments and refunds: the answer is an **RDBMS**, because the data is structured, linked, and needs transactions.)

In the **relational model** all data lives in **tables**.

ID (PK)	NAME	EMAIL
1	Asha	a@x.in
2	Ravi	r@x.in

Each **row** is one student, each **column** has one type, and the order of rows has no meaning.

THE IDEA

Keys.

- **Primary key (PK)**: unique and never NULL. Example: `student.id`.
- **Candidate keys**: all the columns that could be the PK (`id`, and also `email` if every email is unique).
- **Composite key**: a PK made of two columns, such as `(student_id, course_code)` in an enrolment table.
- **Foreign key (FK)**: `enrolment.student_id` refers to `student.id`. The FK may repeat many times (one student, many enrolments) but must match an existing PK value (or be NULL).

A few more words the exam uses: **redundancy** (the same fact stored in many places) wastes space and causes mistakes, while **replication** (deliberate copies on different servers) protects against loss and helps reads. One real question gave "redundancy improves performance and reduces server load" and "replication ensures data is not lost if a copy is destroyed" as a pair, and the right call was that **both statements were false as written**, so read such statements slowly. An **index** is an extra lookup structure (like the index at the back of a book) whose job is to **speed up searching**; it does not encrypt or replicate anything.

WATCH OUT - REAL EXAM TRAP

- A foreign key is **not** unique by itself. The primary key is.
- NULL is not a value. A primary key can never be NULL.
- "Replication" and "redundancy" are not the same word. Replication is deliberate.

Idea 2: ER diagrams and turning them into tables

An **ER (entity-relationship) diagram** is a plan drawn before any table is made. **Entities** are rectangles (Student, Course), **attributes** are ovals or a list inside the box, and **relationships** are lines (often a diamond) between entities, labelled with a verb ("enrols in").

How to read it. Read the line from each side: "One channel *has many* videos. One video belongs to *one* channel." That is **one-to-many**. Check both directions before you answer.

CARDINALITY	EXAMPLE	HOW TO BUILD TABLES
One-to-one	Person and passport	Put an FK (marked unique) on either side, or merge the tables
One-to-many	Channel and videos	Put the FK on the **many** side (`video.channel_id`)
Many-to-many	Students and courses	A new **link (association) table** holding two FKs

SOLVED EXAMPLE: 1. FROM ER TO TABLES

Question. A student can join many courses and a course has many students. Each enrolment also stores marks. Design the tables.

Solution. Many-to-many needs a third table. The link table's primary key is the **pair** of FKs (a composite key), and extra facts about the relationship (marks) go there.

```
CREATE TABLE student(
  id    INTEGER PRIMARY KEY,
  name  TEXT NOT NULL,
  email TEXT UNIQUE
);
CREATE TABLE course(
  code    TEXT PRIMARY KEY,
  title   TEXT NOT NULL,
  credits INTEGER DEFAULT 4
);
CREATE TABLE enrolment(
  student_id  INTEGER REFERENCES student(id),
  course_code TEXT    REFERENCES course(code),
  marks       INTEGER CHECK(marks BETWEEN 0 AND 100),
  PRIMARY KEY (student_id, course_code)
);
```

The composite key also stops the same student joining the same course twice.

How an examiner reads a model to find the relationship: if `Enrollment` has `course_id = db.Column(db.Integer, db.ForeignKey("course.id"))` and `Course` does not have a column pointing back, the FK is on the Enrollment side, so one course has **many** enrolments and each enrolment has one course: **one-to-many**. No link table, so not many-to-many.

IN YOUR PROJECT

Draw your project's ER diagram first (even on paper) and keep it in your report. The viva question is always "why is this table separate?" and "where is the foreign key and why on that side?". Name tables in a consistent style (`student` , not `Students1`).

Idea 3: SQL, the language of tables

Use SQLite (a database in a single file, built into Python). Statements end with `;`. Keywords are case-insensitive, text values are in single quotes.

```
INSERT INTO student(name, email) VALUES ('Asha','a@x.in'), ('Ravi','r@x.in'),
('Meena','m@x.in');
INSERT INTO course(code, title) VALUES ('MAD1','App Dev 1'), ('MAD2','App Dev 2'),
('DBMS','Databases');
INSERT INTO enrolment VALUES (1,'MAD1',80), (1,'DBMS',60), (2,'MAD1',70);

SELECT name FROM student ORDER BY name DESC LIMIT 2;       -- Ravi, Meena
UPDATE enrolment SET marks = marks + 5 WHERE course_code = 'MAD1';
DELETE FROM enrolment WHERE marks < 70;
```

Run on those tables, `SELECT name FROM student ORDER BY name DESC LIMIT 2` gives `Ravi` then `Meena`. After the update the marks are 85, 60, 75; after the delete (marks under 70) only the rows with 85 and 75 remain, so `SELECT COUNT(*) FROM enrolment` is **2**.

THE IDEA

The order SQL thinks in (very useful for predicting output):

1. `FROM` (pick the table, do the joins)
2. `WHERE` (drop rows, **before** grouping, cannot use aggregates)
3. `GROUP BY` (make groups)
4. `HAVING` (drop groups, **after** grouping, can use aggregates)
5. `SELECT` (choose columns)
6. `ORDER BY`, then `LIMIT`

Aggregate functions collapse many rows to one value: `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`.

SOLVED EXAMPLE: 2. WHERE THEN GROUP BY THEN HAVING

Question. Table `production(Manufacturer, Product, Quantity, Cost)`:

MANUFACTURER	PRODUCT	QUANTITY	COST
Industry A	Biscuits	3	100
Industry B	Shoes	2	500
Industry A	Chocolate	10	200
Industry C	Pen	20	50
Industry B	Shirts	5	700
Industry D	Beverages	30	150
Industry C	Notebook	40	250

What does this print?

```
SELECT Manufacturer, COUNT(*) FROM production
WHERE Quantity > 10
GROUP BY Manufacturer
HAVING COUNT(*) = 2;
```

Solution. Step 1: `WHERE Quantity > 10` keeps Pen (20), Beverages (30), Notebook (40). Note that Chocolate (exactly 10) is **not** kept. Step 2: group them: Industry C has 2 rows (Pen, Notebook), Industry D has 1. Step 3: `HAVING COUNT(*) = 2` keeps only C. Output: one row, `Industry C | 2`.

Run without the `WHERE`, the same query would give A, B and C (each has two rows), so the filter order matters.

SOLVED EXAMPLE: 3. GROUP BY WITH ORDER BY AND AN ALIAS

Question. `table_A(ID, First_Name, Last_Name, Course)` has 9 students: Java 4, Python 3, C++ 2. What does this print?

```
SELECT Course, count(ID) AS No_Of_Students
FROM table_A GROUP BY Course
ORDER BY No_Of_Students ASC;
```

Solution. One output row per course, sorted by the count, smallest first: `C++ 2`, `Python 3`, `Java 4`. An alias given with `AS` can be used in `ORDER BY`. The wrong options in the real paper listed nine un-grouped rows or sorted largest first.

A SQLite quirk the papers use. With a table `flowers(ID, Name, color)` holding Jasmine (white), Rose, Lily, Lotus (all pink) and Red rose (red), the query `SELECT count(ID), Name, color FROM flowers GROUP BY color HAVING count(color) < 2` returns `1 | Jasmine | white` and `1 | Red rose | red` (the pink group has 3 rows, so it is dropped). `Name` is not an aggregate nor in `GROUP BY`: other databases refuse this, SQLite quietly picks a value from the group (here the only one).

Joins

A **join** glues rows from two tables where a condition holds (normally FK equals PK).

```
SELECT s.name, e.course_code, e.marks
FROM student s JOIN enrolment e ON e.student_id = s.id
ORDER BY s.name, e.course_code;
```

Result: `Asha DBMS 60`, `Asha MAD1 80`, `Ravi MAD1 70`. **Meena has no enrolment, so she disappears** from an inner `JOIN`. A `LEFT JOIN` keeps every student: `SELECT s.name, e.course_code FROM student s LEFT JOIN enrolment e ON e.student_id = s.id` gives Asha twice, Ravi, and `Meena | NULL`. Forgetting the `ON` condition produces a **cross join**: 3 students times 3 courses is 9 rows.

NULL behaviour

NULL means "unknown", so almost every comparison with it is not true.

```
SELECT NULL = NULL;           -- NULL (not 1)
SELECT NULL IS NULL;          -- 1
SELECT 1 + NULL;              -- NULL
```

For a column `x` holding 1, NULL, 3: `COUNT(*)` is 3, `COUNT(x)` is **2** (NULLs are skipped), `AVG(x)` is 2.0, `SUM(x)` is 4, `WHERE x <> 1` returns only the 3 (the NULL row is **not** returned), and you must write `WHERE x IS NULL` to find it.

Constraints

```
CREATE TABLE t(
  id   INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  age  INTEGER CHECK(age ≥ 0) DEFAULT 18
);
INSERT INTO t(name) VALUES ('Amit');   -- row (1, 'Amit', 18): DEFAULT filled age
INSERT INTO t(name, age) VALUES ('x', -1);  -- fails: CHECK constraint failed: age≥0
INSERT INTO t(age) VALUES (5);              -- fails: NOT NULL constraint failed: t.name
```

`UNIQUE` rejects a duplicate value (several NULLs are still allowed in SQLite). **SQLite does not enforce foreign keys unless you run `PRAGMA foreign_keys = ON;` for each connection.** That surprises people.

YOUR TURN: 1. PREDICT THE SQL OUTPUT

Using the `student`, `course`, `enrolment` tables above (enrolments: (1,MAD1,80), (1,DBMS,60), (2,MAD1,70)), what does `SELECT course_code, AVG(marks), COUNT(*) FROM enrolment GROUP BY course_code HAVING COUNT(*) > 1;` print?

▸ Need a hint?

▸ Show the full solution

YOUR TURN: 2. WRITE THE QUERY

Write SQL to list the names of students who are enrolled in at least one course, without repeating a name.

▸ Need a hint?

▸ Show the full solution

Idea 4: Normalisation and transactions (ACID)

Normalisation is the discipline of splitting tables to remove redundancy so each fact is stored **once**.

FORM	RULE IN PLAIN WORDS
1NF	Every cell holds a **single value** (no "MAD1, MAD2" in one cell); each row unique
2NF	In a table with a composite key, every other column depends on the **whole** key, not part of it

FORM	RULE IN PLAIN WORDS
3NF	Columns depend on the key, **not on other non-key columns** (store `city` with the pin code table, not in every student row)

Example of a violation: an `orders` table with `customer_name` and `customer_phone` copied on every order. Change the phone once and some rows keep the old one (an **update anomaly**). Fix: a `customer` table and an FK.

A **transaction** is a group of statements treated as one unit. The classic: transferring 500 rupees means "subtract from A" and "add to B". If the power fails in between, money vanishes. The database promises **ACID**:

> **THE IDEA**
>
> - **A**tomicity: all of the transaction happens or none of it.
> - **C**onsistency: the data moves from one valid state to another (constraints still hold).
> - **I**solation: parallel transactions do not see each other's half-done work.
> - **D**urability: once committed, the data survives a crash.
>
> `COMMIT` makes the changes permanent; `ROLLBACK` undoes everything since the last commit.

Python's built-in `sqlite3` module shows it clearly:

```
import sqlite3
con = sqlite3.connect("shop.db")      # creates the file if missing
cur = con.cursor()
cur.execute("CREATE TABLE IF NOT EXISTS acct(name TEXT PRIMARY KEY, bal INTEGER)")
cur.execute("INSERT OR REPLACE INTO acct VALUES (?, ?)", ("A", 1000))
con.commit()                          # saved
cur.execute("UPDATE acct SET bal = bal - 500 WHERE name = 'A'")
con.rollback()                        # undone
print(cur.execute("SELECT bal FROM acct WHERE name='A'").fetchall())   # [(1000,)]
con.close()
```

Always pass values with `?` placeholders, never by pasting strings (that invites **SQL injection**).

Idea 5: SQLAlchemy and the ORM

Writing raw SQL strings inside Python gets messy. An **ORM** maps **class to table, object to row, attribute to column**. SQLAlchemy has these parts:

- **Engine**: knows how to reach the database (`create_engine("sqlite:///shop.db")`).
- **Connection / Session**: the Session holds your pending changes and talks to the engine. You `add` objects, then `commit` once.

- **Declarative base**: the parent class of all your models.

```
from sqlalchemy import create_engine, Column, Integer, String, ForeignKey
from sqlalchemy.orm import declarative_base, sessionmaker, relationship

Base = declarative_base()

class Dept(Base):
    __tablename__ = "dept"
    id    = Column(Integer, primary_key=True)
    name  = Column(String, nullable=False)
    staff = relationship("Emp", back_populates="dept")

class Emp(Base):
    __tablename__ = "emp"
    id      = Column(Integer, primary_key=True)
    name    = Column(String)
    dept_id = Column(Integer, ForeignKey("dept.id"))
    dept    = relationship("Dept", back_populates="staff")

engine = create_engine("sqlite:///:memory:")
Base.metadata.create_all(engine)          # CREATE TABLE for every model
s = sessionmaker(bind=engine)()           # a Session

d = Dept(name="CS")
d.staff.append(Emp(name="Ria"))           # link through the relationship
s.add(d)
s.commit()

e = s.query(Emp).first()
print(e.name, e.dept.name, d.staff[0].name)   # Ria CS Ria
```

What happened: `ForeignKey("dept.id")` is the **database-level** link (a column). `relationship(...)` is the **Python-level** convenience: `d.staff` is a list of `Emp`, and `e.dept` is one `Dept`. `back_populates` says the two relationship attributes are two ends of the same link (use the same names on both sides). The older shortcut `backref="dept"` on one side creates the other side automatically. The commit wrote both rows and filled `dept_id` for you.

A model declaration must match the table: a question gave `CREATE TABLE "Items" (ID INTEGER, name TEXT, cost INTEGER, PRIMARY KEY(ID AUTOINCREMENT))` and the correct class has `__tablename__ = "Items"`, `ID = Column(Integer, primary_key=True, autoincrement=True)`, `name = Column(String)`, `cost = Column(Integer)`. Same class name, table name and column names as the SQL.

Commit and rollback. `s.add(Emp(name="Tom")); s.rollback()` throws the pending Tom away, so `s.query(Emp).count()` is still 1.

Idea 6: Flask-SQLAlchemy and CRUD

Flask-SQLAlchemy wires SQLAlchemy into Flask so you write less setup. The names are different, so learn them as a set.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import func

app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///shop.sqlite3"
db = SQLAlchemy(app)                      # or db = SQLAlchemy() then db.init_app(app)

class Author(db.Model):
    id    = db.Column(db.Integer, primary_key=True)
    name  = db.Column(db.String(80), nullable=False)
    books = db.relationship("Book", backref="author", lazy=True)

class Book(db.Model):
    id        = db.Column(db.Integer, primary_key=True)
    title     = db.Column(db.String(100), nullable=False, unique=True)
    price     = db.Column(db.Integer, default=100)
    author_id = db.Column(db.Integer, db.ForeignKey("author.id"))
    def __repr__(self): return f"<Book {self.title}>"

with app.app_context():                   # needed outside a request
    db.create_all()                       # creates missing tables only
```

Notes: the table name is the **lower-case class name** (`Author` becomes `author`), which is why `ForeignKey("author.id")` uses lower case. In Flask-SQLAlchemy 3 the models and `db.session` need an **application context** (inside a view you already have one; in a script use `with app.app_context():`). `create_all()` does not alter a table that already exists: after changing a model, delete the `.sqlite3` file (or use migrations).

THE IDEA

CRUD with the ORM (assume the context is active):

ACTION	CODE
Create	`b = Book(title="SQL", author=a)` then `db.session.add(b)` then `db.session.commit()`
Read all	`Book.query.all()` (a list of objects)
Read one	`Book.query.filter_by(title="SQL").first()` (object or `None`), or `db.session.get(Book, 1)`
Filter	`Book.query.filter(Book.price > 150).all()`, `filter_by(price=100)`
Sort / limit	`Book.query.order_by(Book.price.desc()).limit(1).all()`
Count	`Book.query.filter_by(price=100).count()`
Update	change an attribute (`b.price += 50`), then `db.session.commit()`
Delete	`db.session.delete(b)` then `db.session.commit()`

SOLVED EXAMPLE: 4. RUN A WHOLE CRUD SESSION

Question. What does each print? (Fresh database, the models above.)

```
with app.app_context():
    db.create_all()
    a = Author(name="Asha")
    b1 = Book(title="DB 101", price=300, author=a)
    b2 = Book(title="SQL", author=a)
    db.session.add_all([a, b1, b2]); db.session.commit()
    print(a.books)                          # [<Book DB 101>, <Book SQL>]
    print(b1.author.name, b2.price)         # Asha 100
    print(Book.query.filter(Book.price > 150).all())      # [<Book DB 101>]
    print(db.session.query(func.max(Book.price)).scalar())   # 300
    b = Book.query.filter_by(title="SQL").first()
    b.price += 50; db.session.commit()
    print(b.price)                          # 150
    db.session.delete(b); db.session.commit()
    print(Book.query.all())                 # [<Book DB 101>]
```

Solution. `b2` has no price, so the `default=100` is used when the row is saved: the price is 100. `func.max( … )` with `.scalar()` returns the single number 300; **without** `.scalar()` you only print the SQL text `SELECT max(book.price) …` , not the answer. That exact trap was in a real question: the right choice was `print(db.session.query(func.max(ExamScores.score)).scalar())` , and `db.session.query.max( … )` is invalid.

Constructing objects. SQLAlchemy's default constructor takes **keyword arguments only**. `Book("a", 1, 1)` raises `TypeError: __init__() takes 1 positional argument but 4 were given` . Use `Book(title="a", price=1)` . The right order is create, `add` , then `commit` ; `commit` before `add` saves nothing.

Violating a constraint. Adding a second book titled "SQL" and committing raises `IntegrityError` (the `unique=True`). The session is then in a failed state, so always call `db.session.rollback()` in the `except` block.

SOLVED EXAMPLE: 5. WHAT DOES THIS UPDATE PRINT?

Question. Table `student` with first names John (roll M01, last Doe), John (M02, Luther), Harry (M03, Doe). Code:

```
user1 = Student.query.filter_by(first_name="John").first()
user1.first_name = "Harry"
user1.last_name = "Luther"
db.session.commit()
for s in Student.query.all():
    print(s.first_name, s.last_name)
```

Solution. `.first()` returns only the **first** matching row (M01), not every John. It becomes Harry Luther. The other two rows are untouched. Output: `Harry Luther`, `John Luther`, `Harry Doe`. This matches the real-paper answer.

YOUR TURN: 3. UPDATE MANY ROWS

Table `product(id, name, type, form, quantity)`. Write code to change every edible liquid to type `perishable` and add 450 to its quantity.

▸ Need a hint?

▸ Show the full solution

YOUR TURN: 4. DAYS BETWEEN TWO DATES

A `GlassRepair` model has date columns `broken_at` and `fixed_at`. Write a list of `(id, days)` pairs.

▸ Need a hint?

▸ Show the full solution

Combining conditions

```
Order.query.filter(Order.status == "PAID").filter(Order.amount > 500).all()   # correct
Order.query.filter(Order.status == "PAID", Order.amount > 500).all()          #
correct, same
from sqlalchemy import and_, or_
Order.query.filter(and_(Order.status == "PAID", Order.amount > 500)).all()    # correct
```

```
Order.query.filter(Order.status == "PAID" and Order.amount > 500)            #
TypeError!
```

The last one uses Python's `and` on two SQL expressions, which fails with `TypeError: Boolean value of this clause is not defined`. Use `,` / chained `.filter` / `and_` / `&` (with brackets). `filter_by` takes keyword equality only (`filter_by(status="PAID")`); `filter` takes expressions (`Order.amount > 500`).

WATCH OUT - REAL EXAM TRAP

- `first()` gives **one** object; `all()` gives a **list**; `get` or `first` can give `None` .
- No `commit()` means no permanent change (and another connection will not see it).
- `print(first_name)` inside the loop instead of `emp1.first_name` is a `NameError` : the real paper's answer was "None of these" for exactly this reason. Read the print line.
- `Model.query` is an attribute, not a function: `Book.query( ... )` is wrong.
- `and` / `or` / `not` do not work on SQLAlchemy expressions.
- Table name is lower case: `ForeignKey("author.id")` , not `"Author.id"` .

IN YOUR PROJECT

In the viva you will be asked "show me your model file and explain each relationship". Know which side has the `ForeignKey` , what `backref` or `back_populates` gives you, and what happens to children when a parent is deleted. Keep `db.create_all()` in one obvious place and mention that you do not use migrations.

Exam radar

EXAM RADAR

The evidence file counts graded questions whose text matches week 4 topics (keyword match, so these are approximate): about **52 of 692** in Quiz 1, about **121 of 624** in Quiz 2 and about **209 of 1600** in End Term. This is one of the heaviest topics of Quiz 2, with about 108 distinct questions in our sample.

What the sample asks, most common first:

- **Predict the output of a SQL query** on a small table (`GROUP BY` , `HAVING` , `COUNT` , `ORDER BY` , aliases), about 4 or 5 of the sampled questions.
- **Predict what a Flask-SQLAlchemy snippet prints** after `filter_by( ... ).first()` and `commit()` (changing one row, many rows), also about 4 or 5.
- **Pick the correct model class or query**: `Column( ... )` , `nullable=False` , `primary_key=True` , `func.max( ... ).scalar()` , `filter` with two conditions, subtracting dates.
- **Name the relationship** from an ER picture or from `ForeignKey` lines (one-to-many came up repeatedly).
- **Theory**: which DBMS fits a requirement (an RDBMS for enrolments and fee payments), what an index does, what redundancy and replication mean.
- The correct insert sequence is `create_all()` , build the object **with keyword arguments**, `db.session.add(obj)` , then `db.session.commit()` .

WATCH OUT - REAL EXAM TRAP

- A real wrong option repeated "an unrelated row is also changed"; the right answer touched only the **first** match.
- Wrong options often swap the order `commit` then `add` , or use positional arguments in the constructor.
- `COUNT(*)` after `HAVING` removes whole groups; `WHERE` removes rows **before** the groups form.
- `item.quantity = 450` versus `+= 450` .

Quick quiz

QUICK QUIZ

The table `t` has x values 1, NULL, 3. What is `SELECT COUNT(x) FROM t` ?

- 3
- 2
- 1

NULL

Check

In SQL, which clause may use an aggregate such as `COUNT(*)` to filter groups?

WHERE

ORDER BY only

HAVING

LIMIT

Check

Which of these correctly prints the highest score using Flask-SQLAlchemy?

`print(db.session.query.max(ExamScores.score))`

`print(db.session.query(func.max(ExamScores.score)).scalar())`

`print(db.session.query(func.max(ExamScores.score)))`

`print(ExamScores.query.max())`

Check

`Enrollment` has `course_id = db.Column(db.Integer, db.ForeignKey("course.id"))` and `Course` has no such column. The relationship between Course and Enrollment is...

many-to-many

one-to-many

one-to-one

unrelated

Check

Select all statements that are TRUE. (select all that apply)

A composite primary key is made of two or more columns.

`db.session.rollback()` discards uncommitted changes.

A foreign key must be unique in its own table.

`Model.query.first()` returns a list.

Check

What happens with `Book.query.filter(Book.price > 50 and Book.title == "SQL")`?

It returns books satisfying both conditions

It returns all books

It raises a TypeError

It returns an empty list

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- **Why did you use a database instead of a file?**
- **Explain the relationships in your ER diagram.**
- **What is the difference between a primary key, a foreign key and a candidate key?**
- **What is an ORM and why use SQLAlchemy?**
- **What is the difference between `add`, `commit` and `rollback`?**
- **What does `db.create_all()` do and what does it not do?**
- **What is ACID?**
- **Show me how you would change a student's email live.**

Recap

REMEMBER THIS

- A **DBMS** stores structured data safely; the **relational model** uses tables with **PK**, **FK**, candidate and composite keys.
- **ER diagram**: entities, attributes, relationships. One-to-many puts the FK on the many side; many-to-many needs a **link table**.
- SQL order of thought: `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `SELECT`, `ORDER BY`, `LIMIT`. `WHERE` filters rows, `HAVING` filters groups.
- **NULL** is not equal to anything (use `IS NULL`); `COUNT(col)` skips NULLs.
- **Normalisation** stores each fact once; **ACID** describes safe transactions.
- **SQLAlchemy**: engine, session, declarative models, `relationship` plus `ForeignKey`. **Flask-SQLAlchemy**: `db.Model`, `db.session`, `Model.query`, `db.create_all()`, `SQLALCHEMY_DATABASE_URI`.
- CRUD: `add` / `commit`, `query.all/first/filter_by`, edit an attribute then `commit`, `delete` then `commit`. Use keyword arguments in constructors.

CODE AND FACTS TO KEEP

- `CREATE TABLE t(id INTEGER PRIMARY KEY, name TEXT NOT NULL UNIQUE, n INTEGER DEFAULT 0 CHECK(n≥0));`
- `SELECT cols FROM t JOIN u ON t.id=u.t_id WHERE ... GROUP BY ... HAVING ... ORDER BY ... LIMIT n;`
- Aggregates: `COUNT SUM AVG MIN MAX` ; NULL test: `IS NULL`
- SQLite file URI: `sqlite:///name.sqlite3` ; memory: `sqlite:///:memory:`
- `db = SQLAlchemy(app)` ; models extend `db.Model` ; `db.Column(db.String(80), nullable=False, unique=True)`
- `db.relationship("Child", backref="parent")` plus `db.ForeignKey("parent.id")` on the child
- `db.session.add(obj)` , `db.session.commit()` , `db.session.rollback()` , `db.session.delete(obj)`
- `Model.query.all()` / `.first()` / `.filter_by(a=1)` / `.filter(Model.a > 1)` / `.order_by(...)` / `.count()`
- `db.session.query(func.max(M.col)).scalar()`
- `with app.app_context(): db.create_all()`

MAD1 week 5: Controllers: Flask routes and business logic

> **Official syllabus line:** Controllers - Business logic

The one-minute story

THINK OF IT LIKE THIS

A controller is the manager of a restaurant. The customer (browser) sends an order ("a table for 2, please"). The manager (**controller**) reads the order, checks the rules ("is this table free? has this person paid?"), asks the kitchen (**model**, the database) for ingredients, and then sends the finished plate through the waiter (**view**, the template). If the order is nonsense, the manager politely says no and shows a message.

In Flask, the manager is a **view function** with a `@app.route` label on top. The label says "I handle orders for this address". This week is all about those managers: how they are named, how they read what the browser sent, how they answer, and where the **business rules** live.

By the end of this week you can:

- **write** routes with variable parts, allowed methods and `url_for` ;
- **predict the output** of small Flask apps (trailing slashes, wrong methods, `abort` , error handlers, redirects);

- **read** form data, query strings and JSON from the `request` object;
- **use** `flash`, `session`, redirect-after-POST and error handlers;
- **organise** a bigger app with blueprints and an application factory, and **explain** MVC;
- **put** validation, duplicate checks, permission checks and delete rules in the controller.

Words you need

WORDS YOU NEED

- **Route** - a URL pattern tied to a Python function (`@app.route("/hello")`).
- **View function / controller** - the function that runs when its route is requested and returns the response.
- **Endpoint** - the name of a view function (its function name by default); `url_for` uses it.
- **Request object** - Flask's object holding everything the browser sent (`request.args` , `request.form` , `request.json` ...).
- **Query string** - the part after `?` in a URL (`/search?q=milk&page=2`).
- **HTTP method** - GET (read), POST (send data), PUT, DELETE...
- **Redirect** - a reply that tells the browser "go to this other URL" (status 302).
- **Flash message** - a one-time message stored for the next page.
- **Session** - data remembered between requests, kept in a **signed cookie**.
- **Blueprint** - a group of routes that can be plugged into the app.
- **Business logic** - the rules of your app: validation, who is allowed, what is not allowed twice.
- **Virtual environment** - a private folder of packages for one project.

Idea 1: A Flask app and its folders

Install and run in a **virtual environment**, so this project's packages do not clash with others:

```
python -m venv venv
venv\Scripts\activate          # on Windows; on Mac/Linux: source venv/bin/activate
pip install flask flask-sqlalchemy
pip freeze > requirements.txt  # anyone can rebuild with: pip install -r
requirements.txt
```

A typical project:

```
my_app/
  app.py            controller code (routes) and app setup
  models.py         model classes (db.Model)
  templates/        Jinja HTML files (render_template looks here)
  static/           css, js, images (served at /static/ ... )
  requirements.txt
  venv/             never commit this
```

The smallest app:

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def index():
    return "Hello"

if __name__ == "__main__":
    app.run(debug=True)       # http://127.0.0.1:5000/
```

`Flask(__name__)` tells Flask where this file lives, so it can find `templates/` and `static/` next to it. The default folder names are `templates` and `static`, and you can change them: `Flask(__name__, template_folder="my_templates")`.

SOLVED EXAMPLE: 1. WHICH TEMPLATE IS FOUND?

Question. Folders: `views/index.html`, `templates/page.html`, `my_templates/public/home.html`, `my_templates/private/dashboard.html`. The app is `Flask(__name__, template_folder="my_templates")`. Which of `render_template("public/home.html")`, `"private/dashboard.html"`, `"index.html"`, `"page.html"` work?

Solution. Flask now searches **only** `my_templates/` (it does not also search `templates/`). So the first two work, and `index.html` and `page.html` raise `TemplateNotFound`. This was a real 5-mark question.

Decorators in one minute

`@app.route("/")` is a **decorator**: a function that takes your function and returns a wrapped version. Stacked decorators apply **from the bottom up**.

```
def logger(func):
    def inner():
        print("start")
        r1 = func(); r2 = func()
        return r1, r2
    return inner

@logger
def get_data():
    return "Data"
```

```
print(get_data())
```

Output: `start`, then `('Data', 'Data')`. The name `get_data` now refers to `inner`, which prints first and runs the original twice. With two decorators (`@modify` above `@update`), `update` wraps the function first and `modify` wraps the result. A real paper asked for `modify` (reverses a list) over `update` (halves each value) over a function that squares `[2,4,1,7,5,9]`: squares `[4,16,1,49,25,81]`, then halve `[2,8,0.5,24.5,12.5,40.5]`, then reverse: `[40.5, 12.5, 24.5, 0.5, 8.0, 2.0]`.

Idea 2: Routing and URL rules

`@app.route(rule, methods=[...])` binds a URL to a function. Anything in angle brackets is a **variable part** captured and passed as an argument.

```
@app.route("/greet/<name>")          # name is a string without slashes
def greet(name): return f"Hi {name}"

@app.route("/n/<int:n>")             # converter: only digits, passed as int
def plus_one(n): return str(n + 1)
```

URL	RESULT
`/greet/Ann`	`Hi Ann`
`/greet/`	404 Not Found (the variable cannot be empty)
`/greet`	404 Not Found
`/n/5`	`6`
`/n/x`	404 (not an int)
`/greet/Ann?x=1`	`Hi Ann` (the query string is not part of the route)

Converters: `string` (default), `int`, `float`, `path` (allows slashes), `uuid`.

> **THE IDEA**
>
> **Reading a route against a URL** (this is the exam skill):
>
> - Match the **path** only. Part after `?` is ignored by routing.
> - Each `<var>` must match exactly one non-empty path piece. `/user/<id>` matches `/user/2001` and also `/user/id=2001` (then `id` is the text `id=2001`), but it does **not** match `/2001` or `/id=2001`. With `<int:id>`, the text `id=2001` would no longer match.
> - A query string never fills a `<var>`: for `/<uname>` and the URL `/102?uname=MAD1`, `uname` is `102`, not `MAD1`.

SOLVED EXAMPLE: 2. PATH VARIABLE VERSUS QUERY STRING

Question. The route is `@app.route("/<uname>")` and the function does `if uname[0].isdigit(): abort(400, "Bad Request")` else returns `Good Username`. What does the browser show for `/MAD1?uname=102` and for `/102?uname=MAD1`?

Solution. `uname` always comes from the **path**. `/MAD1?uname=102`: `uname = "MAD1"`, first character not a digit, so `Good Username`. `/102?uname=MAD1`: `uname = "102"`, starts with a digit, so `abort(400)` and the page shows **Bad Request**.

Trailing slash rules

```
@app.route("/home")
def home1(): return "This is homepage 1"

@app.route("/home/")
def home2(): return "This is homepage 2"
```

A rule **without** a trailing slash matches only `/home`. A rule **with** the slash matches `/home/`, and if a user types `/home` for a rule that ends in a slash, Flask redirects them to `/home/`. With both rules defined as above, `/home` gives homepage 1 and `/home/` gives homepage 2, so the real answer for `/home/` was "This is homepage 2".

Methods

A route answers **GET** only unless you list other methods.

```
@app.route("/only", methods=["POST"])
def only(): return "posted"
```

A browser address bar sends GET, so `/only` gives **405 Method Not Allowed**. If the app has `@app.errorhandler(405)` returning "Please check the web request", that message is what you see. In one real question the route was `/home/<username>` with `methods=['POST']`, the URL typed in the browser was `/home/user_4` (a GET), and the code inside (including `abort(404)`) **never ran**: the 405 handler's text appeared. Check the **method** before you trace the body.

`url_for` builds URLs from endpoint names

```
with app.test_request_context():
    print(url_for("greet", name="J"))        # /greet/J
    print(url_for("plus_one", n=3, x=1))     # /n/3?x=1   (extra values become the
query string)
    print(url_for("static", filename="a.css"))   # /static/a.css
```

Use `url_for` instead of typing paths, so renaming a URL does not break your links. In templates: `{{ url_for('greet', name='J') }}`. Missing a required variable raises BuildError. In a real paper `url_for('profile', student_name='John', student_id=101)` for the route `/student/<student_name>/<student_id>` printed `/student/John/101`, **not** a query string.

Idea 3: The request object and responses

Import it: `from flask import request`.

WHAT	CODE	EXAMPLE
Method	`request.method`	`"GET"` or `"POST"`
Query string	`request.args.get("page", 1, type=int)`	`/q?page=3` gives `3`
Form fields	`request.form.get("name")`	from `<form method="post">`
Uploaded files	`request.files["photo"]`	needs `enctype="multipart/form-data"`
JSON body	`request.json` or `request.get_json()`	needs `Content-Type: application/json`
Headers, cookies	`request.headers`, `request.cookies`	

`request.args["x"]` raises a **400 Bad Request** if `x` is missing, while `request.args.get("x")` returns `None`. A POST whose body is JSON but with no JSON content type makes `request.json` fail with **415 Unsupported Media Type** (Flask 3).

What a view can **return**:

RETURN VALUE	WHAT THE BROWSER GETS
a string	status 200, `text/html`
`"made", 201`	body plus a status code
a `dict`	JSON, `application/json` (Flask converts it)
`jsonify( ... )`	JSON, `application/json`; also works for lists
`render_template("x.html", a=1)`	HTML from a template
`redirect(url_for("index"))`	status 302 and a `Location` header
`abort(404)`	stops now and shows the 404 error page

SOLVED EXAMPLE: 3. READING THE BROWSER'S DATA

Question. What does this print for the URL `/signin?uid=professor&pwd=54321` ?

```
valid_users = [{"uid": "student", "pwd": "12345"}, {"uid": "professor", "pwd":
"54321"}]

@app.route("/signin")
def login():
    args = request.args
    for vu in valid_users:
        if vu["uid"] == args["uid"] and vu["pwd"] == args["pwd"]:
            return "You are authorized!!"
        else:
            return "You are unauthorized!!"
```

Solution. The `else` belongs to the `if` , **inside** the loop. The loop checks only the first user (`student`), finds a mismatch and returns `unauthorized` straight away; `professor` is never examined. Output: **You are unauthorized!!** (Correct code would put `return "unauthorized"` after the loop.) I ran this to confirm. If `uid` was missing from the URL, `args["uid"]` would give 400 Bad Request.

Dict versus `jsonify` . For the route that returns `{"Restaurant 1": "Hotel Alpine", ...}` the content type is `application/json` . `jsonify({"The list of gadgets": Gadgets})` gives an object with **one key** whose value is the **whole list** of dicts (not a merged dictionary). For `jsonify(courses)` where `courses` is a list of two dicts, status is **200** and mimetype `application/json` .

Idea 4: Redirects, errors, flash messages and sessions

Redirect. `redirect(url_for("index"))` sends 302 and the browser requests the new URL itself. In a real question, `/first_index/XYZ` redirected to `index1` 's URL (`/XYZ`) and `/first_index/ABC` redirected to `/second_index` , which rendered a page with a "Go back" link.

Errors. `abort(404)` raises an HTTP error. Customise the page with a handler:

```
@app.errorhandler(404)
def not_found(e):
    return "Course not available!", 404
```

If you do not give the status after the text, the page still has status 200: fine for the browser output, wrong for tests. Common codes: 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 405 Method Not Allowed, 500 Server Error. In recent Flask, a handler for an unknown code such as 407 will not even register, so such questions always hinge on which error happens **first** (here the 405).

SOLVED EXAMPLE: 4. WHICH HANDLER RUNS?

Question. Two routes use the `path` converter, which allows slashes in the variable: `/course/<path:subject>` shows "Welcome to Python course!" if "Python" is in `subject`, else calls `abort(404)`; `/learn/<path:subject>` redirects to the course route if "Python" is in `subject`, else `abort(404)`. A 404 handler returns "Course not available!". What shows for `/course/ML/Python?topic=NLP`, `/learn/web/Python?level=advanced` and `/course/Java/React?type=Framework`?

Solution. The query string is ignored by routing. First URL: `subject = "ML/Python"`, which contains "Python": **Welcome to Python course!**. Second: `subject = "web/Python"`, so `/learn` redirects to `/course/web/Python` and the browser shows **Welcome to Python course!** (it is *not* "Course not available"). Third: `subject = "Java/React"`, no "Python", so `abort(404)` and the handler shows **Course not available!**. I ran this to confirm. With a plain `<subject>` (no `path`), a URL with two pieces like `/one/ML/Python` would match nothing and give 404 directly. Always check which converter the rule uses.

Flash messages show a one-time note on the next page.

```
from flask import flash, redirect, url_for, request

@app.route("/add", methods=["GET", "POST"])
def add():
    if request.method == "POST":
        name = request.form.get("name", "").strip()
        if not name:
            flash("Name required")
            return redirect(url_for("add"))
        flash("Added")
        return redirect(url_for("index"))
    return render_template("add.html")
```

In the template: `{% for m in get_flashed_messages() %}<p>{{ m }}</p>{% endfor %}`. `Flash needs a secret_key`, because messages ride in the session.

Sessions remember a user between requests. The data is stored in a **cookie in the browser, signed with `secret_key`**: the user can read it (it is only base64) but cannot change it without Flask noticing.

```
app.secret_key = "change-me"      # or app.config["SECRET_KEY"]

session["uid"] = 7                # log in
uid = session.get("uid")          # read; None if absent
session.pop("uid", None)          # log out
```

With no secret key, using `session` raises `RuntimeError: The session is unavailable because no secret key was set`. Do not store passwords or big objects in the session, and never commit a real secret key to a public repository.

YOUR TURN: 1. PREDICT THE STATUS

For the app with `@app.route("/home", methods=["GET"])` returning a string and no other routes, what status do `/home` and `/home/` give?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 2. SPOT THE ERROR

A student writes `@app.route("/user/<id>")` and tests `http://127.0.0.1:5000/2001`. Which URLs from `/user/2001`, `/user/id=2001`, `/id=2001`, `/2001` give Not Found?

▸ **Need a hint?**

▸ **Show the full solution**

Idea 5: Post/Redirect/Get, validation and business logic

Business logic is the part that says what your app **allows**. It belongs in the controller (or a service function it calls), never only in the HTML.

THE IDEA

The standard recipe for a form route (Post/Redirect/Get):

1. GET: show the empty form.
2. POST: read `request.form`, **validate**.
3. If bad: `flash` a message and show the form again (or redirect to it).
4. If good: change the database, `commit`, then **redirect** (do not render directly), so a browser refresh does not re-submit the form.

There are two layers of validation. **Frontend validation** happens in the browser (`required`, `type="email"`, `min`, `max`, `pattern` in the HTML) and is easy to bypass. **Backend validation** happens in the Python view and is the one you must trust. A real MSQ listed a form that had `required` attributes **and** a Python check `if name == "": return redirect('/add')` as having **both**, and a form whose HTML is built by the Python function as **dynamically generated**. One more subtlety from that paper: a CSS rule `input:valid {border: 2px solid red}` colours the border when the input is valid, not when it is blank.

SOLVED EXAMPLE: 5. A COMPLETE CONTROLLER WITH ALL THE USUAL CHECKS

Question. Write a to-do app with users and tasks where: you must be logged in, a title cannot be blank or duplicated, and you can delete only your own tasks.

Solution. Models (week 4 style):

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), unique=True, nullable=False)
    tasks = db.relationship("Task", backref="owner", cascade="all, delete-orphan")

class Task(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
    user_id = db.Column(db.Integer, db.ForeignKey("user.id"), nullable=False)
```

Controllers:

```
@app.route("/tasks/add", methods=["GET", "POST"])
def add_task():
    uid = session.get("uid")
    if uid is None:
        abort(401)                                  # not logged in
    if request.method == "POST":
        title = request.form.get("title", "").strip()
        if not title:
            flash("Title is required")              # validation
            return redirect(url_for("add_task"))
        if Task.query.filter_by(user_id=uid, title=title).first():
            flash("Duplicate task")                 # duplicate check
            return redirect(url_for("add_task"))
        db.session.add(Task(title=title, user_id=uid))
        db.session.commit()
        flash("Task added")
        return redirect(url_for("list_tasks"))      # PRG
    return render_template("add.html")

@app.route("/tasks/<int:task_id>/delete", methods=["POST"])
def delete_task(task_id):
    t = db.session.get(Task, task_id)
    if t is None:
        abort(404)
    if t.user_id ≠ session.get("uid"):
        abort(403)                                  # authorisation
```

```
    db.session.delete(t)
    db.session.commit()
    return redirect(url_for("list_tasks"))
```

I ran a version of this with Flask's test client. Results: before login `/tasks/` redirects to the login route; a `POST` adding "Buy milk" redirects (302) to the list, which then shows the task and the flash "Task added"; adding "Buy milk" again or an empty title redirects back to the form with a flash message; a POST to delete task 1 redirects (302) and removes it; deleting a task that does not exist is 404; and a `GET` to the delete URL is **405**, because the route allows POST only. Deleting should be POST (or DELETE), never a plain link, because links can be triggered by browsers and crawlers.

Cascade and delete rules. If a user owns tasks and you delete the user, the tasks would be left pointing at nothing. Decide a rule: `cascade="all, delete-orphan"` deletes the children too; or refuse deletion while children exist (check and `flash("Remove tasks first")`). The viva loves this question.

WATCH OUT - REAL EXAM TRAP

- A POST-only route typed in the address bar gives **405**, whatever the code inside says.
- After a successful POST, **redirect**; do not `render_template` (the refresh re-posts).
- A `flash` with no `secret_key` fails; a `session` with no `secret_key` fails.
- Checking permissions only by hiding a button is not security; the controller must check too.
- `request.form["x"]` for a missing field gives 400; use `.get`.

IN YOUR PROJECT

Put at least these in your controller and be ready to show them: a login check on every private route, a validation message for blank and duplicate input, a 404 page for a missing record, and a delete that refuses or cascades. In the viva, the examiner will submit a blank form or delete something twice to see what happens.

Idea 6: Blueprints, configuration and the MVC mapping

MVC. Model = `db.Model` classes and the database. **View** = Jinja templates (what the user sees). **Controller** = the route functions: they read the request, apply business rules, call models and pick a view. A good app keeps SQL out of templates and HTML out of models.

Blueprints split a big app into parts (auth, tasks, admin). Each blueprint has its own routes, and the app registers it with a URL prefix.

```
from flask import Blueprint
tasks_bp = Blueprint("tasks", __name__, url_prefix="/tasks")

@tasks_bp.route("/")
```

```
def list_tasks():  ...

app.register_blueprint(tasks_bp)
```

The endpoint name now has the blueprint's name in front: `url_for("tasks.list_tasks")` gives `/tasks/`. Inside the same blueprint you can write `url_for(".list_tasks")`.

Configuration is a dictionary on the app:

```
app.config["SECRET_KEY"] = "change-me"
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///app.sqlite3"
app.config.update(DEBUG=True)          # many keys at once
app.config.from_object("config.Dev")  # or from a class or file
```

Application factory. Instead of a global `app`, write a function that **builds** the app. This lets tests create a fresh app each time and avoids import loops.

```
db = SQLAlchemy()                      # created without an app

def create_app(config=None):
    app = Flask(__name__)
    app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///:memory:"
    app.config["SECRET_KEY"] = "change-me"
    app.config.update(config or {})
    db.init_app(app)                   # connect db to this app
    app.register_blueprint(tasks_bp)
    with app.app_context():
        db.create_all()
    return app

if __name__ == "__main__":
    create_app().run(debug=True)
```

Whether your course project needs a factory depends on its size: a single `app.py` is acceptable. Know the idea.

YOUR TURN: 3. NAME THE ENDPOINT

A blueprint is created with `Blueprint("shop", __name__, url_prefix="/shop")` and has a route `@bp.route("/items")` on a function `items`. What does `url_for("shop.items")` return, and what URL does the browser use?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 4. WHERE DOES IT BELONG?

Sort these into model, view, controller: (a) check a username is not taken, (b) the HTML table of tasks, (c) `class Task(db.Model)`, (d) deciding to redirect to login.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 5. ORDER OF DECORATORS

In `@a` above `@b` above `def f()`, which wrapper runs first when `f()` is called?

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

The evidence counts questions whose text matches week 5 topics (keyword match, approximate): about **109 of 692** in Quiz 1, about **119 of 624** in Quiz 2 and about **232 of 1600** in End Term; 119 distinct questions in the sample. Most of these are **predict what the browser shows** questions.

Typical questions in the sample:

- A small app with several routes, then "what will the browser render for this URL?" (trailing slash, missing variable gives Not Found, query string versus path variable).
- A POST-only route requested from the browser (405 and its handler text).
- `url_for` output, including extra keyword arguments becoming `?x=1`.
- `abort(400/404)` plus an `errorhandler`.
- Which URL must be used for `/text/size/<size>/<color>` (the pieces go in route order).
- Frontend versus backend validation, dynamic form generation.
- Decorator output order, `render_template` with a changed `template_folder`, `redirect(url_for( ... ))` chains.

WATCH OUT - REAL EXAM TRAP

- `/home` and `/home/` are different routes when both are defined.
- A route variable never matches an empty piece or a `/` (except `path`).
- URL pieces must follow the route order: `/text/size/20/red`, not `/text/size/red/20`.
- A query string is not a path variable.
- A view function with the wrong method list never runs its body.
- The `else` inside a `for` loop's `if` returns early.

Quick quiz

QUICK QUIZ

A route is `@app.route("/user/<int:id>")`. Which URL gives 404?

- `/user/2001`
- `/user/abc`
- `/user/7?x=1`
- `/user/0`

Check

A route allows only POST. What does the browser show when you open its URL?

- [ ] the function's output
- [ ] 405 Method Not Allowed (or the 405 handler's text)
- [ ] 404 Not Found
- [ ] 500 Internal Server Error

Check

Which statements about `session` are TRUE? (select all that apply)

- [ ] It needs `secret_key`.
- [ ] The data is stored in a signed cookie in the browser.
- [ ] It is stored in the database by default.
- [ ] The user can edit it freely and Flask will accept it.

Check

`url_for("plus_one", n=3, x=1)` **for the route** `/n/<int:n>` **returns...**

- [ ] `/n/3/1`
- [ ] `/n/1/3`
- [ ] `/n/3?x=1`
- [ ] an error

Check

After a successful form POST you should...

- [ ] render the page directly
- [ ] redirect to another page (Post/Redirect/Get)
- [ ] call `abort(200)`
- [ ] clear the cookies

Check

`@app.route("/n/<n>")` returns the string `n`. What does `/n/5` return?

- 5 as an int
- the text `5`
- 404
- an error

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- Explain how a request travels through your Flask app.
- What is the difference between `request.args` and `request.form`?
- Why do you redirect after a POST?
- How does `session` work and why do you need `secret_key`?
- What is a blueprint and why did you use one?
- Where is your business logic and give an example.
- Why `url_for` instead of typing the path?
- What does a virtual environment and `requirements.txt` give you?

Recap

REMEMBER THIS

- A Flask app: `app.py`, `templates/`, `static/`, models; use a **virtual environment** and `requirements.txt`.
- **Routing**: `@app.route(rule, methods=[...])`; `<int:id>` variables; only the path is matched; trailing slash rules; default method is GET.
- **request**: `args`, `form`, `files`, `json`, `method`. **Responses**: string, tuple with status, dict or `jsonify`, `render_template`, `redirect`, `abort`.
- **Errors**: `abort(code)` plus `@app.errorhandler(code)`. A wrong method gives 405; a missing variable piece gives 404.
- **flash** and **session** need `secret_key`; session is a signed cookie.
- **Post/Redirect/Get**; validation, duplicates, authorisation and delete rules are business logic in the controller.
- **Blueprints** group routes (`url_for("bp.func")`); an **application factory** builds the app; `app.config` holds settings.
- **MVC**: model = data, view = template, controller = route.

CODE AND FACTS TO KEEP

- `@app.route("/x/<int:id>", methods=["GET", "POST"])`
- `request.args.get("q")`, `request.form.get("f")`, `request.json`, `request.method`
- `return "text", 201`; `return {"a": 1}`; `return jsonify(list)`
- `redirect(url_for("endpoint", arg=1))`; `abort(404)`
- `@app.errorhandler(404)`; `flash("msg")`; `get_flashed_messages()`
- `app.secret_key = "..."`; `session["k"] = v`; `session.get("k")`; `session.pop("k", None)`
- `Blueprint("name", __name__, url_prefix="/p")`; `app.register_blueprint(bp)`
- `db.init_app(app)`; `def create_app(): ...; return app`
- `python -m venv venv`; `pip install -r requirements.txt`; `pip freeze > requirements.txt`
- Status codes: 200 OK, 302 redirect, 400 bad request, 401 unauthorised, 403 forbidden, 404 not found, 405 wrong method, 415 wrong media type, 500 server error

MAD1 week 6: APIs and REST: OpenAPI and Flask-RESTful

> **Official syllabus line:** APIs and REST APIs

The one-minute story

THINK OF IT LIKE THIS

An API is a restaurant menu plus a waiter. You do not walk into the kitchen. You read the menu (the **documentation**), tell the waiter "one dosa, please" (an **HTTP request**), and the waiter brings the dish (the **response**) without showing you how it was cooked. A **REST API** is a menu with strict, simple manners: every dish has its own name (**URL**), every action has a standard word (**GET** to look, **POST** to order, **PUT** to change the order, **DELETE** to cancel), and the waiter treats every call as new: he does not remember your last visit, so you carry what you need each time (**stateless**).

A normal web page route returns HTML for a person. An API route returns **JSON** for another program: a mobile app, a Vue page, or a test script.

By the end of this week you can:

- **explain** what an API is and name the REST constraints;
- **design** URLs and choose the right verb and status code for each action;
- **write** a Flask-RESTful API with `Resource`, `reqparse`, `marshal_with` and `abort`;
- **predict the output** of Flask-RESTful code and of `curl` commands (including which call gives an error);
- **document** an API in OpenAPI (Swagger) YAML;
- **test** an API with curl, Postman or Python `requests`.

Words you need

WORDS YOU NEED

- **API** - Application Programming Interface: a defined way for one program to talk to another.
- **REST** - a style for web APIs built on resources, URLs and standard HTTP verbs.
- **Resource** - a thing the API exposes (a student, a book), named by a URL.
- **Endpoint** - one URL plus one method (for example `GET /api/books/3`).
- **JSON** - JavaScript Object Notation: the text format for API data (objects, lists, strings, numbers, true, false, null).
- **Stateless** - the server keeps no memory of earlier requests; each request carries all it needs.
- **Idempotent** - doing it once or many times leaves the server in the same state.
- **Status code** - the three-digit result number (200, 404...).
- **OpenAPI** - a standard format (in YAML or JSON) to describe an API; Swagger is the tooling around it.
- **Flask-RESTful** - a Flask extension where each resource is a class with methods named after HTTP verbs.
- **reqparse** - Flask-RESTful's helper to read and validate request data.
- **marshal / marshal_with** - shape the response using a fields dictionary.
- **curl** - a command-line program that sends HTTP requests.
- **Pagination** - returning a long list in pages.

Idea 1: What an API is and what makes it REST

The server exposes data, and **any client** can use it: a browser page, an Android app, or a Python script. They agree on **URLs**, **methods** and **JSON**.

> **THE IDEA**
>
> **REST constraints** (know each in one line):
>
> 1. **Client-server**: UI and data storage are separate, so each can change on its own.
> 2. **Stateless**: every request has everything the server needs (no server-side memory between calls).
> 3. **Cacheable**: responses say whether they can be reused, so repeated reads are cheap.
> 4. **Uniform interface**: resources identified by URLs, standard verbs (GET, POST, PUT, DELETE), representations such as JSON.
> 5. **Layered system**: the client cannot tell if there is a proxy or load balancer between it and the server.
>
> (A sixth, "code on demand", is optional.)

API route versus web page route. A web route returns HTML (`render_template`) and is meant for a person; an API route returns JSON and a status code, and its errors are JSON too (not a styled error page). In one real question the route `/` returned a Python dict and the answer was that its **content type is `application/json`**; and the route returning `jsonify({"The list of gadgets": Gadgets})` was a valid API access point: both `/` and `/home` can be considered API endpoints.

JSON in 60 seconds

```
{
  "name": "Asha",
  "age": 27,
  "married": false,
  "phones": ["212-555-1234", 2125551234],
  "address": {"city": "Chennai", "pin": null}
}
```

Rules: **double quotes only** on keys and strings, no trailing comma, no comments, values are string, number, true, false, null, object or array. Python mapping: object is `dict`, array is `list`, `true` is `True`, `null` is `None`. In JavaScript, `JSON.parse('{"name":"icecream","flavor":"vanilla","type":"cup"}')` gives an object, and `obj.name + ", " + obj.flavor + ", " + obj.type` prints `icecream, vanilla, cup`: access is by **key name**, not by position.

Idea 2: Designing the API: URLs, verbs, status codes

Name resources with nouns, in plural, and put the action in the verb, not in the URL.

ACTION	METHOD AND URL	SUCCESS STATUS
list all books	`GET /api/books`	200
read one	`GET /api/books/3`	200 (404 if absent)
create	`POST /api/books` with a JSON body	**201 Created**
replace/update	`PUT /api/books/3` with a JSON body	200 (or 204)
delete	`DELETE /api/books/3`	200 or **204 No Content**

Not RESTful: `GET /api/deleteBook?id=3` or `/getAllBooks`. (A real exam question used URLs like `/delete/val1` and a `PUT` request: because a route only answered PUT, a plain GET gave "none of these" = 405.)

The CRUD mapping: **C**reate = POST, **R**ead = GET, **U**pdate = PUT (or PATCH for a partial update), **D**elete = DELETE.

Safe and idempotent

METHOD	SAFE (NO CHANGE)	IDEMPOTENT
GET	yes	yes
PUT	no	**yes** (setting the same value twice gives the same state)
DELETE	no	**yes** (the second delete changes nothing more; it may return 404)
POST	no	**no** (two POSTs create two records)

Status codes for APIs

CODE	MEANING	TYPICAL USE
200	OK	successful GET or PUT
201	Created	successful POST that made a record
204	No Content	success with no body (delete)
400	Bad Request	malformed or missing input
401	Unauthorized	not logged in or bad credentials
403	Forbidden	logged in but not allowed
404	Not Found	no such resource or URL
405	Method Not Allowed	the URL exists but not for this verb

CODE	MEANING	TYPICAL USE
409	Conflict	duplicate, such as an existing username
422	Unprocessable	input is well-formed but fails validation
500	Server Error	a bug in your code

SOLVED EXAMPLE: 1. CHOOSE THE VERB AND THE CODE

Question. Match each action: (a) a new student is added; (b) a student's grade is changed; (c) a student that does not exist is requested; (d) someone adds a student whose roll number already exists; (e) a student is removed.

Solution. (a) `POST /students`, 201. (b) `PUT /students/4`, 200. (c) `GET /students/99`, 404. (d) `POST /students`, **409 Conflict** (some courses use 400 or 422; say which you picked and why). (e) `DELETE /students/4`, 204 (or 200 with a message).

WATCH OUT - REAL EXAM TRAP

- Do not return 200 with `{"error": ...}` for a failure. Use the proper code so clients can react.
- 401 is "who are you?", 403 is "I know you, but no".
- A 404 for a missing resource and a 405 for a wrong verb are different problems.
- POST is not idempotent; PUT and DELETE are.

Idea 3: Flask-RESTful: Api, Resource and methods

Install: `pip install flask-restful`. Each **Resource** is a class; its methods are named `get`, `post`, `put`, `delete` and are called by the verb. You register it with `add_resource(Class, url ...)`.

```
from flask import Flask, request
from flask_restful import Api, Resource

app = Flask(__name__)
api = Api(app)

students = {1: {"name": "Arjun", "grade": "A"}}

class Student(Resource):
    def get(self, student_id):
        if student_id in students:
            return students[student_id], 200
        return {"message": "Student not found"}, 404
```

```
    def put(self, student_id):
        data = request.json
        if student_id in students:
            students[student_id]["grade"] = data.get("grade", students[student_id]
["grade"])
            return {"message": "Student record updated", "student":
students[student_id]}, 200
        return {"message": "Student not found"}, 404

api.add_resource(Student, "/student/<int:student_id>")
```

I ran this with the test client. A `GET /student/1` returns status 200 and `{"name": "Arjun", "grade": "A"}`. `GET /student/10` returns 404 and `{"message": "Student not found"}`. `PUT /student/1` with JSON `{"grade": "A+"}` returns 200 and the updated student (grade `A+`). `PUT /student/4` returns 404 because 4 is not in the dictionary: it does not create a record. A URL like `/student/abc` does not match `<int: ... >` and gives the plain 404 page.

What Flask-RESTful does for you: a returned `dict` or list becomes JSON; a returned tuple `(body, status)` sets the status; the response mimetype is `application/json`. Returning `None` gives JSON `null` with status 200.

THE IDEA

Reading `add_resource` (very common question).

- The URL variables must match the **method's parameters by name**. `def get(self, tno)` with `"/api/ver1/<int:tno>"` works; `"/api/ver1/<tno>"` gives a string `tno`, so `train["train_no"] == tno` (an int) would never be equal.
- One class can have **many URLs**: `api.add_resource(UserResource, "/users", "/users/<int:user_id>")`. The same method is used for each; a method that has the wrong parameters for the URL fails with a **500** error.
- A verb with **no method** in the class gives **405 Method Not Allowed**.
- Only the `< ... >` parts bind variables. `/api/<brand>/<model>` matches `/api/maruti/swift`, but **not** `/api/brand=Ford,model=Fiesta` (one piece, one variable missing) and not `/api/Honda&City`.

SOLVED EXAMPLE: 2. WHICH CURL COMMAND FAILS?

Question.

```
class UserResource(Resource):
    def get(self):
        return {"status": "GET request"}, 200
    def post(self, user_id):
        return {"status": "POST request", "user_id": user_id}, 201

api.add_resource(UserResource, "/users", "/users/<int:user_id>")
```

Which of these give an error: `GET /users`, `GET /users/5001`, `POST /users`, `POST /users/5001`?

Solution. I ran it. `GET /users` gives 200. `POST /users/5001` gives 201 and `{"status": "POST request", "user_id": 5001}`. **`GET /users/5001` fails**: Flask passes `user_id=5001` to `get(self)`, which takes no such argument: `TypeError ... unexpected keyword argument`, shown as 500. **`POST /users` fails** the same way: `post(self, user_id)` is called with nothing: `TypeError ... missing 1 required positional argument`. The two failing commands are `GET /users/5001` and `POST /users`.

SOLVED EXAMPLE: 3. METHODS AND ROUTES TOGETHER

Question.

```
class MyApi(Resource):
    def get(self):  return {"greet": "Hello from GET Api!"}
    def put(self):  return {"greet": "Hello from PUT Api!"}

api.add_resource(MyApi, "/api/get", "/api/put", "/api/post")
```

Which of `curl ... /api/get -X get`, `-X put` on `/api/put`, `-X post` on `/api/post`, `-X put` on `/api/get`, `-X get` on `/api/put` and `-X get` on `/api/post` throw an error?

Solution. The **URL name means nothing**: all three URLs go to the same class, whose methods are `get` and `put`. Only the verb matters. So everything works except **POST** (no `post` method), which gives 405: only `-X post` on `/api/post` fails. A lower-case `-X get` also works, because Flask-RESTful lower-cases the method name.

Idea 4: Input and output: reqparse, marshal_with, abort

reqparse declares which inputs you expect, converts their type and rejects bad input with a 400.

```
from flask_restful import reqparse

parser = reqparse.RequestParser()
parser.add_argument("n", type=int, required=True, help="n must be an int")

class MyResponse(Resource):
    def post(self):
        args = parser.parse_args()
        return args                      # a dict-like, so it becomes JSON

api.add_resource(MyResponse, "/")
```

```
curl -X POST http://127.0.0.1:5000 -H "Content-Type: application/json" -d "{\"n\":3}"
```

The reply is `{"n": 3}` (status 200). With `{"n": "x"}`, or with no `n` at all, the reply is **400** and `{"message": {"n": "n must be an int"}}`. If you send JSON with `-d` but **leave out the `Content-Type` header**, Flask 3 refuses it with a **415** error: so for JSON, curl needs `-H "Content-Type: application/json"` (as the real question's right answers had). Plain `-d "status=active&role=admin"` sends **form-encoded** data (not JSON); with `-X PUT` it is a PUT in form format, which was the right answer to a real question.

By default reqparse looks in the JSON body and the form. For values from the query string, say so: `parser.add_argument("page", type=int, default=1, location="args")`. (I found that on Flask 3 a GET with no JSON header and reqparse's default location raises 415, so **always set `location="args"` for GET parameters**.)

marshal_with filters and converts your output to the shape in a `fields` dictionary.

```
from flask_restful import fields, marshal_with

out_fields = {"Id": fields.Integer, "name": fields.String}

class MyApi(Resource):
    @marshal_with(out_fields)
    def post(self):
        info = parser.parse_args()     # parser has Id, name, email
        return info
```

`curl -X POST ... -d '{"Id":3, "name":"user_1", "email":"user_1@gmail.com"}'` returns `{"Id": 3, "name": "user_1"}`. The `email` is **dropped** because it is not in the fields dictionary, and `"Id": 3` is a number (`fields.Integer`), not the string `"3"`. A real question had the same code and its right option was `{"Id": 3, "name": "user_1"}`. The `get` method in that class returned a plain dict with the keys `Identity`, `Name`, `E-mail`: no marshal there, so keys are what you wrote.

`marshal_with(fields, envelope="book")` wraps the output as `{"book": {...}}`. Fields: `fields.Integer`, `fields.String`, `fields.Float`, `fields.Boolean`, `fields.List(...)`, `fields.Nested(...)`.

abort with a message. Flask-RESTful has its own `abort`.

```
from flask_restful import abort
abort(404, message="Book 99 not found")     # JSON: {"message": "Book 99 not found"},
status 404
```

SOLVED EXAMPLE: 4. A REAL CRUD API IN ONE CLASS

Question. Write the full API for books: list with paging, read one, create with validation.

Solution. Tested code:

```
from flask import Flask
from flask_restful import Api, Resource, reqparse, fields, marshal_with, marshal,
abort

app = Flask(__name__)
api = Api(app)

ITEMS = [{"id": i, "title": f"Book {i}", "price": i * 10, "secret": "x"} for i in
range(1, 6)]
book_fields = {"id": fields.Integer, "title": fields.String, "price":
fields.Float}

list_parser = reqparse.RequestParser()
list_parser.add_argument("page", type=int, default=1, location="args")
list_parser.add_argument("per_page", type=int, default=2, location="args")
list_parser.add_argument("q", location="args")

new_parser = reqparse.RequestParser()
new_parser.add_argument("title", required=True, help="title is required")
new_parser.add_argument("price", type=int, default=0)

class Books(Resource):
    def get(self):
        a = list_parser.parse_args()
        rows = [b for b in ITEMS if not a["q"] or a["q"] in b["title"]]
        start = (a["page"] - 1) * a["per_page"]
        return {"page": a["page"], "total": len(rows),
                "items": [marshal(b, book_fields) for b in rows[start:start +
a["per_page"]]]}

    @marshal_with(book_fields, envelope="book")
    def post(self):
        a = new_parser.parse_args()
        b = {"id": len(ITEMS) + 1, "title": a["title"], "price": a["price"]}
        ITEMS.append(b)
        return b, 201

class Book(Resource):
    @marshal_with(book_fields)
    def get(self, bid):
```

```
        for b in ITEMS:
            if b["id"] == bid:
                return b
        abort(404, message=f"Book {bid} not found")

api.add_resource(Books, "/api/v1/books")
api.add_resource(Book, "/api/v1/books/<int:bid>")
```

Results from a test client:

- `GET /api/v1/books` gives 200 and `{"page": 1, "total": 5, "items": [{"id": 1, "title": "Book 1", "price": 10.0}, {"id": 2, ...}]}`. The `secret` field never appears: `marshal` dropped it. `price` is `10.0` because `fields.Float` is used.
- `GET /api/v1/books?page=3&per_page=2` gives the last page with one item (id 5).
- `GET /api/v1/books?q=Book 4` gives `total` 1.
- `POST /api/v1/books` with `{"title": "New", "price": 55}` gives 201 and `{"book": {"id": 6, "title": "New", "price": 55.0}}`; without `title` it gives 400 and `{"message": {"title": "title is required"}}`.
- `GET /api/v1/books/99` gives 404 and `{"message": "Book 99 not found"}`.

The `/api/v1/` in the path is **versioning**: when the design must change in a way that breaks clients, you publish `/api/v2/` and let the old one live on.

YOUR TURN: 1. PREDICT THE OUTPUT

```
salaries = {1: {"name": "Ravi", "basic": 14500}, 2: {"name": "David", "basic": 9500}}

class Salaries(Resource):
    def do_something(self):
        for s in salaries.keys():
            salaries[s]["hra"] = salaries[s]["basic"] * 0.10
    def get(self):
        self.do_something()
        return salaries

api.add_resource(Salaries, "/api/salaries")
```

What does `GET /api/salaries` return?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 2. URL MATCHING

Two resources: `api.add_resource(Default, "/")` (its `get(self)` returns `{"output": "Default computed value : 0"}`) and `api.add_resource(Power, "/<int:x>/<int:y>")` (its `get(self, x, y)` returns `{"output": "Computed x^y = " + str(x**y)}`). What do `GET /`, `GET /6/2`, `GET /x=6/y=2` and `GET /6` give?

▸ **Need a hint?**

▸ **Show the full solution**

Idea 5: Calling and testing an API

curl options to know:

OPTION	MEANING
`-X POST`	choose the HTTP method (GET is the default)
`-H "Content-Type: application/json"`	add a header
`-d '{"name":"Asha"}'`	request body (implies POST if `-X` is absent)
`-v`	verbose: shows request and response **headers** (for debugging)
`-i`	print the response headers followed by the body

```
curl http://127.0.0.1:5000/api/v1/books/1
curl -X POST http://127.0.0.1:5000/api/v1/books -H "Content-Type: application/json" -d "{\"title\":\"New\",\"price\":55}"
curl -v http://127.0.0.1:5000/api/v1/books
curl -X DELETE http://127.0.0.1:5000/api/v1/books/1
```

On Windows `cmd`, JSON in `-d` must use double quotes with `\"` escapes (as in line 2); in Git Bash or Linux you can use single quotes. A real question asked which option shows detailed request/response information for debugging: **`curl -v`** (`-i` just adds the response headers to the output).

From Python with the **requests** library:

```
import requests
r = requests.post("http://127.0.0.1:5000/api/v1/books", json={"title": "New", "price": 55})
print(r.status_code)      # 201
```

```
print(r.json())           # {'book': {...}}
r = requests.get("http://127.0.0.1:5000/api/v1/books", params={"page": 2})
```

`json=` sets the body and the `Content-Type` header for you. **Postman** is a point-and-click tool for the same: choose a method, enter the URL, put JSON in the Body tab (raw, JSON), and read status and body.

YOUR TURN: 3. WRITE THE CURL

Write a command that updates student 1's grade to `B` using the `Student` resource above.

▸ **Need a hint?**

▸ **Show the full solution**

IN YOUR PROJECT

Expose at least one JSON API for your project's main table (CRUD for a resource), with proper status codes and JSON errors. Keep the web page routes (HTML) separate from the API routes (`/api/...`). In the viva, you may be asked to call it with curl, so keep two or three working commands ready, and the OpenAPI file next to it.

Idea 6: Documenting with OpenAPI (Swagger)

Documentation says what endpoints exist, their parameters, body and responses. In the exam, "what describes available endpoints, request parameters and response formats?" has the answer **API documentation / API reference**; and "**OpenAPI Specification is a way to describe interfaces for RESTful APIs**" (it is not a programming language, performance tester or auth library).

An OpenAPI file is YAML (or JSON) with these top-level keys:

KEY	PURPOSE
`openapi`	the version of the spec (for example `3.0.3`)
`info`	title, version, description
`servers`	base URLs
`paths`	each URL, then each method under it
`components/schemas`	reusable data shapes, referenced with `$ref`

```
openapi: 3.0.3
info:
  title: Books API
```

```
  version: "1.0"
paths:
  /api/v1/books:
    get:
      summary: List books
      parameters:
        - name: page
          in: query
          schema: {type: integer, default: 1}
      responses:
        "200":
          description: A page of books
          content:
            application/json:
              schema:
                type: array
                items: {$ref: "#/components/schemas/Book"}
    post:
      summary: Create a book
      requestBody:
        required: true
        content:
          application/json:
            schema: {$ref: "#/components/schemas/NewBook"}
      responses:
        "201": {description: Created}
        "400": {description: Missing or invalid title}
  /api/v1/books/{bid}:
    get:
      summary: Get one book
      parameters:
        - name: bid
          in: path
          required: true
          schema: {type: integer}
      responses:
        "200": {description: The book}
        "404": {description: No such book}
components:
  schemas:
    Book:
      type: object
      properties:
        id: {type: integer}
        title: {type: string}
        price: {type: number}
```

```
  NewBook:
    type: object
    required: [title]
    properties:
      title: {type: string}
      price: {type: integer}
```

Points to notice: path variables are written `{bid}` (curly braces) here, but `<int:bid>` in Flask; parameters have `in:` (`query`, `path`, `header`); response codes are strings; the YAML is indentation-sensitive (use spaces, never tabs). **Swagger UI** or the Swagger Editor reads this file and gives you a page where you can read and try the API.

Idea 7: Pagination, filtering, versioning and error responses

- **Pagination**: `GET /books?page=2&per_page=10`. The server returns one slice and often the total, so clients do not download everything.
- **Filtering and sorting**: `GET /books?author=Asha&sort=price`. Filters go in the **query string**, resource identity goes in the **path**.
- **Versioning**: put the version in the path (`/api/v1/`) or a header, so old clients keep working.
- **Error responses**: always JSON with the right status, such as `{"message": "Book 99 not found"}` (404). Flask-RESTful does this for you when you use its `abort`.
- **Cacheable GET**: GET responses can be cached; POST responses normally not.

YOUR TURN: 4. TRUE OR FALSE

(a) PUT is idempotent. (b) A REST server stores each client's state between requests. (c) `abort(404, message="...")` from `flask_restful` returns JSON. (d) `-d '{...}'` alone is enough for a JSON POST in Flask 3.

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

The evidence counts questions whose text matches week 6 topics (keyword match, approximate): about **24 of 692** in Quiz 1, about **82 of 624** in Quiz 2 and about **108 of 1600** in End Term, with 56 distinct questions in our sample. Many Quiz 2 items are "read this Flask-RESTful class and a URL or curl command, then say the output".

What the sample asks:

- **Which curl command or URL works / throws an error**, depending on methods defined, `add_resource` URLs and URL variables (about 6 questions in the sample).
- **Predict the JSON** a resource returns (including `marshal_with` dropping fields and converting types, and the `reqparse` result `{"n": 3}`).
- **Status codes and mimetype** of a `jsonify` response (200 and `application/json`).
- **Match** GET, PUT, POST with their URLs; read a PUT that updates a database row and print the final state.
- **Theory**: what OpenAPI is, what documentation contains, valid JSON, `curl -v`, form-encoded versus JSON.

WATCH OUT - REAL EXAM TRAP

- A URL variable like `<int:x>` means the URL has just the value, `/6/2`; `x=6/y=2` in the path is **Not Found**.
- Only the **method** decides which Python method runs; the URL text (such as `/api/post`) means nothing.
- `curl` with `-d '{ ... }'` but no `Content-Type: application/json` header is not a JSON request.
- `marshal_with` removes keys that are not in the fields dictionary.
- JSON keys are always strings, so integer dictionary keys print with quotes.
- A method whose parameters do not match the URL variables is a runtime (500) error, not a 404.

Quick quiz

QUICK QUIZ

Which HTTP method is NOT idempotent?

- GET
- PUT
- DELETE

POST

Check

A resource class defines only `get` and `put`. A client sends `POST` to its URL. The response is...

404 Not Found

405 Method Not Allowed

200 with an empty body

201 Created

Check

Which status code fits "a new record was created"?

200

201

204

301

Check

A resource uses `@marshal_with({"Id": fields.Integer, "name": fields.String})` and returns `{"Id": "3", "name": "u", "email": "e@x.in"}`. The JSON is...

`{"Id": "3", "name": "u", "email": "e@x.in"}`

`{"Id": 3, "name": "u"}`

`{"name": "u", "email": "e@x.in"}`

an error

Check

Select all statements that are TRUE about REST. (select all that apply)

It is stateless.

Resources are identified by URLs.

The server must remember the last request of every client.

It can only return HTML.

Check

In an OpenAPI file, which top-level section lists the URLs and their methods?

info

paths

components only

servers

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- **What is a REST API?**
- **Why is REST called stateless?**
- **Which status codes does your API return and when?**
- **Difference between `reqparse` and `marshal_with`?**
- **How is an API route different from a normal Flask route?**
- **How did you document and test your API?**
- **Why POST for create and PUT for update?**
- **What is versioning and why use it?**

Recap

REMEMBER THIS

- An **API** is a contract between programs; **REST** is resources plus URLs plus standard verbs, with client-server, stateless, cacheable, uniform interface and layered constraints.
- CRUD maps to POST, GET, PUT, DELETE. GET is safe and idempotent; PUT and DELETE are idempotent; POST is not.
- Status codes: 200, 201, 204, 400, 401, 403, 404, 405, 409, 422, 500.
- **Flask-RESTful**: `Api(app)`, `Resource` subclasses with `get/post/put/delete`, `add_resource(Class, url, ...)`; method arguments must match URL variables.
- `reqparse` validates input (set `location="args"` for query values); `marshal_with` shapes output; `abort(code, message=...)` sends a JSON error.
- JSON needs double quotes; curl needs `-H "Content-Type: application/json"` for a JSON body; `-v` shows headers.
- **OpenAPI** YAML: `openapi`, `info`, `paths` (`parameters`, `requestBody`, `responses`), `components/schemas`.
- Pagination and filtering use the query string; version the API in the path.

CODE AND FACTS TO KEEP

- `api = Api(app)` ; `api.add_resource(Cls, "/x", "/x/<int:id>")`
- `class Cls(Resource): def get(self, id): return { ... }, 200`
- `parser = reqparse.RequestParser(); parser.add_argument("n", type=int, required=True, location="json")` ; `args = parser.parse_args()`
- `@marshal_with({"id": fields.Integer, "name": fields.String})`
- `abort(404, message="...")` from `flask_restful`
- `curl -X POST URL -H "Content-Type: application/json" -d "{\"a\":1}"` ; `curl -v URL`
- `requests.post(URL, json={...})` , `.status_code` , `.json()`
- 201 created, 204 no content, 400 bad input, 401 not logged in, 403 not allowed, 404 missing, 405 wrong verb, 409 conflict
- OpenAPI: `openapi: 3.0.3` , `info` , `paths: /x: get: parameters/responses` , `components: schemas`

MAD1 week 7: Backend systems: indexing, search, caching and logging

> **Official syllabus line:** Backend Systems

The one-minute story

THINK OF IT LIKE THIS

A busy restaurant kitchen. Your Flask app is the kitchen. When it works you want to know *what happened* (the **logbook**: "7:02 table 4 ordered, 7:05 oven failed"). When something burns you need to *stop time and look inside* (**debugging**). The pantry is huge, so finding the rice quickly needs a **label system** (an **index**), or a cook will search every shelf. A guest who wants "anything with mango" needs a **search catalogue** (**full-text search**). The same popular dish ordered 500 times an hour should not be cooked from scratch each time: keep a few ready on the counter (a **cache**). And when the queue at the door gets long, you either buy a bigger stove (**scale up**) or open more kitchens behind a **manager at the door** (**scale out** with a **load balancer**).

Backend systems are the habits that keep a working app **observable, fast and able to grow**.

By the end of this week you can:

- **predict the output** of Python `logging` programs for any level and format;
- **debug** a Flask app with debug mode, `pdb` , the Flask shell and tracebacks;
- **create an index**, read `EXPLAIN QUERY PLAN` , and say when an index helps and when it hurts;
- **build a full-text search** with SQLite FTS5 and keep it in sync with triggers;

- **explain caching** (client vs server, TTL, keys, invalidation, `Cache-Control`, `ETag`) and use Flask-Caching;
- **calculate** latency, throughput, bandwidth, data volume and requests per second;
- explain **scaling, load balancers, stateless servers, batch jobs, N+1 queries and pagination**.

Words you need

WORDS YOU NEED

- **Log** - a time-ordered record of events written by the program while it runs.
- **Log level** - the importance of a message: DEBUG, INFO, WARNING, ERROR, CRITICAL.
- **Handler** - decides *where* a log record goes (console, file, email).
- **Formatter** - decides *how* a log record looks (time, level, message).
- **Traceback** - the list of function calls that led to an error, printed when it crashes.
- **Index** - an extra sorted structure that lets the database find rows without scanning the whole table.
- **Full-text search (FTS)** - searching *words inside* long text, not exact column values.
- **Cache** - a small fast store of copies of results so they need not be recomputed or refetched.
- **TTL** (time to live) - how long a cached item may be used before it expires.
- **Latency** - the *delay* before something starts or arrives (milliseconds).
- **Throughput** - how much work or data is completed per unit time (requests/s, MB/s).
- **Bandwidth** - the maximum data rate of a link or memory (bits or bytes per second).
- **Scale up / scale out** - make one machine bigger / add more machines.
- **Stateless server** - a server that keeps no per-user memory between requests, so any copy can answer any request.
- **N+1 problem** - one query for a list plus one extra query for every item in it.
- **Pagination** - returning big results one page at a time.

Idea 1: Logging (the app's diary)

`print()` is fine on day one. A real app needs **levels**, **timestamps**, **files**, and the ability to switch noise on and off without editing every line. That is the `logging` module.

THE IDEA

Five levels, in increasing severity (the number is what Python stores):

LEVEL	NUMBER	TYPICAL USE
DEBUG	10	detailed values for developers
INFO	20	normal events ("user logged in")
WARNING	30	something odd but the app continues
ERROR	40	an operation failed
CRITICAL	50	the app may not continue

Rule: a message is shown only if its level is greater than or equal to the configured level. The default level is **WARNING**, so DEBUG and INFO are silent unless you lower the level.

```
import logging
logging.basicConfig(level=logging.INFO,
                    format='%(levelname)s:%(name)s:%(message)s')
log = logging.getLogger("shop")
log.debug("hidden")             # 10 < 20, suppressed
log.info("hello %s", "ravi")    # shown
log.warning("low stock")
log.error("fail")
log.critical("down")
```

Output (checked by running it):

```
INFO:shop:hello ravi
WARNING:shop:low stock
ERROR:shop:fail
CRITICAL:shop:down
```

Handlers, formatters, loggers. A **logger** (named, e.g. `"shop"`) creates *records*. A **handler** sends them somewhere (`StreamHandler` = console, `FileHandler("app.log")`). A **formatter** turns the record into text using fields like `%(asctime)s`, `%(levelname)s`, `%(message)s`, `%(name)s`. Each handler can have its own level, so you can print everything to the screen but save only errors to a file:

```
fh = logging.FileHandler("app.log")
fh.setLevel(logging.ERROR)
fh.setFormatter(logging.Formatter("%(asctime)s %(levelname)s %(message)s"))
log.addHandler(fh)
```

```
log.warning("seen on console only")
log.error("seen on console and in app.log")
```

`basicConfig()` is a one-line setup of the **root** logger. Other details that are asked:

- `logging.exception("msg")` is `error` plus the **full traceback**; use it inside `except`.
- Use lazy formatting: `log.info("user %s", name)`, not an f-string, so the string is built only if the level is on.
- `basicConfig` does nothing if the root logger already has handlers; call it once, early.
- **Flask's logger** is `app.logger` (named after your app). It logs unhandled exceptions as ERROR with a traceback and returns **500** to the browser. In debug mode Flask also shows werkzeug's request lines.

```
@app.route("/")
def home():
    app.logger.info("visited home")
    return "ok"
```

SOLVED EXAMPLE: 1. PREDICT THE OUTPUT (A REAL PAPER STYLE)

Question. `log.py` :

```
import logging, sys
logging.basicConfig(level=logging.INFO,
                    format='%(levelname)s - %(message)s')
num1, num2 = int(sys.argv[1]), int(sys.argv[2])
logging.info("This operation checks factors")
if num1 % num2 == 0:
    logging.debug(f"{num2} is a factor of {num1}")
else:
    logging.warning(f"{num2} is not the factor of {num1}")
```

What is printed for `python log.py 25 5` ?

Solution. Level is INFO (20). The first call logs INFO: shown. `25 % 5 == 0`, so the `if` branch runs and logs DEBUG (10), which is below INFO and **suppressed**. Output: `INFO - This operation checks factors`.

Same style with `level=logging.WARNING` and a `critical` call: `CRITICAL` (50) is always shown, in the format `CRITICAL - text`.

SOLVED EXAMPLE: 2. LOGGING.EXCEPTION

Question. Level is `WARNING`, format `%(asctime)s - %(levelname)s - %(message)s`. The code calls `logging.info("Value added")` when the value is fine, but raises `ValueError` for negative numbers and handles it with `logging.exception("Exception occurred: %s", str(ve))`. What happens for `python log.py -34` and for `python log.py 34`?

Solution. For `-34`: the `ValueError` is caught, `logging.exception` logs at **ERROR** (40 >= 30) so a line such as `2026-10-06 11:51:07,727 - ERROR - Exception occurred: Invalid value: Please enter a positive value.` appears, followed by the traceback. For `34`: only `logging.info` runs (20 < 30), so **nothing at all** is printed. (We ran both.)

WATCH OUT - REAL EXAM TRAP

- `sys.argv` splits on **spaces**: `python code.py mad 1` gives `argv[1] == 'mad'`, not `'mad 1'`. A real paper used this to make the `else` branch run.
- A lower level number means *more* output. `level=DEBUG` shows everything; `level=ERROR` shows only ERROR and CRITICAL.
- Without `basicConfig`, the default format is `LEVEL:logger:message`, and the default level is WARNING.
- The timestamp in `asctime` has milliseconds after a comma: `2024-07-29 10:15:32,125`.

Idea 2: Debugging

Debug mode. `app.run(debug=True)` (or `flask --app app run --debug`) does two things: it **auto-reloads** when you save a file, and it shows an interactive **error page** with the traceback in the browser. **Never use it in production**, because it lets anyone who sees the page run code on your server.

Reading a traceback. Read it **from the bottom**: the last line is the exception type and message; the line above shows the exact line of *your* code that failed; the lines above show who called whom.

```
Traceback (most recent call last):
  File "app.py", line 11, in b
    return 1/0
ZeroDivisionError: division by zero
```

`pdb`, the Python debugger. Put `breakpoint()` (or `import pdb; pdb.set_trace()`) where you want to stop. Execution pauses with a `(Pdb)` prompt:

COMMAND	MEANING
`n` (next)	run the next line, do not enter functions
`s` (step)	step *into* a function call

COMMAND	MEANING
`c` (continue)	run until the next breakpoint
`p expr`	print an expression, e.g. `p user.name`
`l` (list)	show the code around the current line
`q`	quit

Flask shell. `flask --app app shell` opens a Python prompt with the **app context** already pushed, so `db`, models and queries work right away. It is the fastest way to test a query or insert test data.

```
flask --app app shell
>>> User.query.count()
```

> **TIP**
> Debugging recipe: reproduce it, read the last line of the traceback, add a log line or `breakpoint()` just before the failure, check the *values*, fix, and re-run. Do not edit five things at once.

> **IN YOUR PROJECT**
> Add a `logging.basicConfig` at the top of your project, log every login, every failed login and every database error at the right level, and keep one `app.log` file. In the viva you may be asked "how would you know if a user complained about a failure yesterday?" The answer is your log file.

Idea 3: Database indexing

Without help, `SELECT * FROM student WHERE city='c7'` makes SQLite read **every row** (a **full table scan**, cost proportional to *n*). An **index** is a separate sorted structure, like the index at the back of a book: it maps a column value to the rows that hold it.

B-tree intuition. Real indexes are **B-trees**: a short, wide tree whose nodes hold many sorted keys. To find a value you go from the root down a few levels, choosing the right branch at each node. For a million rows a binary search needs about 20 comparisons; a B-tree with 100 keys per node needs only about **3 levels**. Cost grows with the **logarithm** of the table size, not the size itself.

```
import sqlite3
c = sqlite3.connect(":memory:")
c.execute("create table student(id integer primary key, name text, city text, marks
int)")
c.executemany("insert into student(name,city,marks) values(?,?,?)",
              [("s%d" % i, "c%d" % (i % 50), i % 100) for i in range(10000)])
```

```
q = "select * from student where city='c7'"
print(c.execute("explain query plan " + q).fetchall())
c.execute("create index idx_city on student(city)")
print(c.execute("explain query plan " + q).fetchall())
```

Real output (the middle numbers differ by version):

```
[(2, 0, 216, 'SCAN student')]
[(3, 0, 62, 'SEARCH student USING INDEX idx_city (city=?)')]
```

`SCAN` = reads all rows. `SEARCH ... USING INDEX` = uses the index. `EXPLAIN QUERY PLAN` shows what SQLite *plans* to do without running the query.

THE IDEA

Index facts

- Syntax: `CREATE INDEX idx_city ON student(city);` and `CREATE UNIQUE INDEX ...` (also enforces uniqueness).
- The **primary key** is already indexed (`SEARCH student USING INTEGER PRIMARY KEY (rowid=?)`).
- An index can cover **several columns**: `CREATE INDEX idx_cm ON student(city, marks)` helps `WHERE city=? AND marks>?` .
- An index can be on a column with **duplicate values** (city repeats 200 times above and still helps).
- **Helps:** `WHERE` , `JOIN ... ON` , `ORDER BY` on the indexed column; and *selective* conditions (few rows match).
- **Cost:** extra disk space, and **every INSERT, UPDATE and DELETE must also update the index**, so writes get slower. Index the columns you search on, not everything.
- **Does not help:** tiny tables; conditions like `name LIKE '%5'` (a leading wildcard cannot use a sorted order; we ran it and it still said SCAN); columns where most rows match.

SOLVED EXAMPLE: 3. READING A QUERY PLAN

Question. After `create index idx_city on student(city)` , what does SQLite do for `select * from student where city='c7' and marks>50` ? And after also `create index idx_cm on student(city,marks)` ?

Solution (run on the database above). First: `SEARCH student USING INDEX idx_city (city=?)` : it uses the city index then checks marks on those rows. Second: `SEARCH student USING INDEX idx_cm (city=? AND marks>?)` : both conditions are answered from the index. The two-column index is better for this query.

In Flask-SQLAlchemy: `city = db.Column(db.String, index=True)` creates the index with the table. Foreign key columns used in joins are good candidates.

WATCH OUT - REAL EXAM TRAP

Two statements from a real End Term were both **false**: "an index can only be created on one column" and "indexes cannot be created on columns having duplicate values". Also remember the one-line purpose: an index **speeds up data retrieval**; it does not encrypt, replicate, or describe the schema.

Idea 4: Full-text search with SQLite FTS5

`WHERE body LIKE '%flask%'` scans everything and cannot rank results. **FTS5** is an SQLite extension that builds an **inverted index** (word -> list of rows containing it). You create a **virtual table**: it looks like a table but is stored in special search structures.

```
c.execute("create virtual table notes using fts5(title, body)")
c.execute("insert into notes values('Flask intro','Flask is a micro framework'),"
          "('Jinja','Templates for Flask apps'),('SQL','Indexes make queries fast')")
print(c.execute("select title from notes where notes match 'flask'").fetchall())
print(c.execute("select title from notes where notes match 'micro OR
index*'").fetchall())
```

```
[('Flask intro',), ('Jinja',)]
[('Flask intro',), ('SQL',)]
```

- `MATCH` searches words; `'micro OR index*'` uses `OR` and the prefix star `index*` (matches *indexes*). Also `AND`, `NOT` and quoted phrases `'"micro framework"'`.
- Matching is **by whole words** (the default tokenizer does no stemming): `cache` does not match `caching`; use `cache*`.
- Order by relevance with `ORDER BY rank`; highlight with `snippet( … )` or `highlight( … )`.

Keeping it in sync. If your real data lives in a normal table `post`, make the FTS table an **external-content** table and add **triggers** (automatic actions that run after insert, update, delete) so the search index never goes stale:

```
create table post(id integer primary key, title text, body text);
create virtual table post_fts using fts5(title, body, content='post',
content_rowid='id');
create trigger post_ai after insert on post begin
  insert into post_fts(rowid,title,body) values (new.id,new.title,new.body);
end;
create trigger post_ad after delete on post begin
  insert into post_fts(post_fts,rowid,title,body)
values('delete',old.id,old.title,old.body);
end;
```

```
create trigger post_au after update on post begin
  insert into post_fts(post_fts,rowid,title,body)
values('delete',old.id,old.title,old.body);
  insert into post_fts(rowid,title,body) values (new.id,new.title,new.body);
end;
```

Search and join back to the real rows: `select p.id,p.title from post_fts f join post p on p.id=f.rowid where post_fts match 'cache*'`. We tested all three triggers: after an update the new words match, after a delete the row disappears from results.

> **IN YOUR PROJECT**
> A search box over your project's items (courses, trackers, posts) is a good "extra feature". Say in the viva: "plain `LIKE` scans every row; FTS5 uses an inverted index and can rank, and triggers keep it in sync."

Idea 5: Caching

A **cache** stores the result of expensive work (a database query, a rendered page, an API call) so the next identical request is answered instantly. **Why:** lower latency for the user, less load on the server and database, lower cost.

	CLIENT-SIDE CACHE	SERVER-SIDE CACHE
Lives in	browser (or a proxy/CDN in between)	server memory, Redis, Memcached
Saves	network trip and server work	database and computation work
Controlled by	HTTP headers	your code (Flask-Caching)

Server-side with Flask-Caching. (Install `flask-caching`.)

```
from flask_caching import Cache
cache = Cache(app, config={"CACHE_TYPE": "SimpleCache", "CACHE_DEFAULT_TIMEOUT": 60})

@app.route("/slow")
@cache.cached(timeout=30)          # whole response cached for 30 s
def slow():
    ...

@cache.memoize(timeout=30)         # cache by function arguments
def square(x): ...
```

We ran this: two calls of `/slow` ran the function **once** (the second came from the cache). `square(4)` twice ran once, but `square(5)` would compute again, because the **cache key** includes the arguments. A

cache key is the label under which the result is stored: for `@cache.cached` it is the request path, for `memoize` it is the function name plus arguments.

> THE IDEA
>
> **Three problems every cache has**
>
> 1. **Expiry (TTL):** after `timeout` seconds the item is dropped and recomputed, so data can be stale for up to the TTL.
> 2. **Invalidation:** when the real data changes (a new post is saved) you must **delete the affected cache keys** (`cache.delete(key)` , `cache.delete_memoized(fn, args)` , `cache.clear()`), or users keep seeing old data. "There are only two hard things in computing: cache invalidation..."
> 3. **Cache hit / miss:** a *hit* is served from the cache; a *miss* goes to the real source and then fills the cache.
>
> Cache things that are **read often and change rarely**. Never cache per-user private pages under a shared key.

Client-side caching with HTTP headers.

- `Cache-Control: public, max-age=60` - any cache may reuse this response for 60 s without asking. `private` = only the user's browser. `no-store` = never keep it.
- **ETag** - a fingerprint of the response ("v1"). Next time the browser sends `If-None-Match: "v1"`. If unchanged the server replies **304 Not Modified** with *no body*: the browser reuses its copy and saves bandwidth. If changed: 200 with the new body.

```
@app.route("/data")
def data():
    resp = make_response('{"v": 1}')
    resp.set_etag("v1")
    resp.headers["Cache-Control"] = "public, max-age=60"
    return resp.make_conditional(request)
```

Tested: first request 200 with `ETag: "v1"` ; second request with `If-None-Match: "v1"` gave **304** and an empty body.

REST APIs and caching. Only **safe, idempotent reads (GET)** are meant to be cached. POST, PUT, DELETE change data and are not cached by default. A paper asked which statements about REST caching were *false*; the false ones were: "caching degrades service and increases server load", "browsers treat all GET and POST requests as cacheable by default", and "caching optimises the network by *increasing* latency". Caching **reduces** latency and load; only GET is cacheable by default.

WATCH OUT - REAL EXAM TRAP

- Caching helps *reads*. After a write you must invalidate, otherwise users see **stale** data.
- A 304 response has no body; the data comes from the browser's own copy.
- A bigger TTL means fewer misses but staler data.

Idea 6: Performance vocabulary and arithmetic

Latency is "how long until the first byte" (delay). **Throughput** is "how much per second" (rate). **Bandwidth** is the maximum rate a link or memory can carry. A highway analogy: latency is the travel time of one car; bandwidth is the number of lanes.

Memory hierarchy (fast and small at the top, slow and large at the bottom):

LEVEL	LATENCY	CAPACITY PER COST	THROUGHPUT
Registers	lowest	smallest	highest
SRAM (CPU cache)	very low	small	very high
DRAM (main memory)	low	medium	high
SSD	higher	large	medium
HDD	**highest**	**largest**	**lowest**

So "highest latency" = HDD, "lowest density" = registers, "lowest throughput" = HDD. For small, hot data wanting very low latency (such as frequently used records), the answer in the papers is **SRAM**-like cache memory.

The arithmetic. Convert units first: 1 byte = 8 bits; use the question's own definition (usually 1 KB = 1000 B, 1 MB = 1000 KB).

SOLVED EXAMPLE: 4. DRAM BANDWIDTH

Question. A DRAM module has bus width 64 bits, clock 400 MHz, **DDR** (two transfers per clock). Maximum bandwidth in dual-channel mode?

Solution. Bus = 64 bits = 8 bytes. Transfers per second = 400 million x 2 = 800 million. One channel = 8 x 800 million = 6.4 GB/s. **Dual channel = 2 x 6.4 = 12.8 GB/s.** (At 800 MHz dual channel: 25.6. At 1 GHz single channel DDR: 8 bytes x 2 billion = 16 GB/s. These three appeared in real papers.)

SOLVED EXAMPLE: 5. LATENCY AND REQUESTS PER SECOND

Question. Speed of signal in cable is 2 x 10^8 m/s. Distance to the data centre is 2000 km. Find the one-way and round-trip latency, and the maximum requests per second one client can finish one after another.

Solution. One way: 2,000,000 m / 2x10^8 = 0.01 s = **10 ms**. Round trip = **20 ms**. If each request waits for its reply, requests per second = 1000 / 20 = **50**. (For 1000 km: 5 ms one way.) A bigger response adds transfer time = size / bandwidth.

SOLVED EXAMPLE: 6. DATA CONSUMED

Question. A phone moves away at 60 km/h from a tower with range 40 km, using all of 120 Mbps the whole time. How many GB are used?

Solution. Time = 40/60 h = 40 min = 2400 s. Data = 120 Mbit/s x 2400 s = 288,000 Mbit = 36,000 MB (divide by 8) = **36 GB**. For a graph of bandwidth against time, split into steps and add up: 8 Mbps for 4 h = 8 x 14,400 s = 115,200 Mbit = 14.4 GB, and so on.

Other quick formulas: **disk worst-case rotational latency = one full rotation = 60000 / RPM milliseconds** (6000 RPM gives 10 ms, 5000 RPM gives 12 ms; the *average* is half). **HDD throughput = data per revolution x revolutions per second**: to match an SSD of 180 MB/s with 4.8 Mbit (0.6 MB) per revolution you need 300 rev/s = **18000 RPM**.

Idea 7: Scaling, stateless servers, background jobs

- **Scale up (vertical):** a bigger machine (more RAM, CPU). Simple, but has a ceiling and one point of failure.
- **Scale out (horizontal):** more machines behind a **load balancer**, which spreads requests (round robin, least connections). Practically unlimited, and one machine failing does not stop the site.
- **Stateless servers** make scale-out easy: if user data lives in the **database or a shared store (Redis)** and not in one server's memory, any server can answer any request. This is why a login can be carried by a token or a shared session store instead of a local variable.
- **Redundancy and replication:** keeping copies (extra servers, database replicas) for **availability and fault tolerance**. A paper had two statements, "redundancy improves performance and reduces load" and "replication ensures data is not lost", and marked **both false**. The official wording favours: redundancy is for reliability, not speed, and "ensures" is too strong. Check your course notes for the exact phrasing.
- **ACID vs BASE:** ACID (atomic, consistent, isolated, durable) suits **financial transactions**; BASE (basically available, soft state, eventually consistent) suits **social media** feeds. In the paper: scale up = increasing RAM, scale out = adding physical servers.

Session memory is also a scaling calculation: 1.5 million sessions x 4 KB = 6,000,000,000 bytes = **6 GB** of RAM (a paper asked this for a Redis session store).

Background (batch) jobs. Work that is slow and does not need an instant answer (emailing reports, resizing images, nightly exports) should not block the web request. The request **puts a job on a queue**

and returns at once; a separate **worker** takes jobs from the queue. **Cron** runs a task on a schedule ("every night at 2"). The tool for this in Flask is **Celery**, which you meet in MAD2; here you only need the idea.

Idea 8: N+1 queries and pagination

N+1 problem. One query fetches N authors, then touching `author.books` fires **one more query per author**: N+1 queries. We counted queries with 5 authors and 3 books each:

```
for a in Author.query.all():          # 1 query
    len(a.books)                      # +5 lazy queries  → 6 total
for a in Author.query.options(selectinload(Author.books)).all():   # 2 queries
    len(a.books)
for a in Author.query.options(joinedload(Author.books)).all():     # 1 query (JOIN)
    len(a.books)
```

Measured: **6, 2 and 1** queries. Fix: **eager loading** (`selectinload` , `joinedload`) or a single JOIN. Add an index on foreign keys.

Pagination. Never return a million rows. Return a page at a time with `LIMIT` and `OFFSET` (or `db.paginate`):

```
p = db.paginate(db.select(Book).order_by(Book.id), page=2, per_page=4)
# p.items = [b11, b12, b20, b21]; p.page=2, p.pages=4, p.total=15, p.has_next,
p.has_prev
```

Always add `ORDER BY` so pages are stable. API shape: `GET /books?page=2&per_page=4` .

Your turn

YOUR TURN: 1. LOGGING OUTPUT

`logging.basicConfig(level=logging.WARNING, format='%(levelname)s - %(message)s')` . Then `logging.debug("a")` , `logging.warning("b")` , `logging.critical("c")` , `logging.info("d")` . What prints?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 2. INDEX OR NOT

Table `orders` has 2 million rows. For each query say whether an index on the column helps: (a) `WHERE customer_id = 17`, (b) `WHERE note LIKE '%gift%'`, (c) a table of 20 rows, (d) a column `status` where 99 percent of rows are 'done'.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 3. BANDWIDTH

Dual-channel DDR, 64-bit bus, 200 MHz clock. Maximum bandwidth in GB/s?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 4. DISK AND NETWORK

(a) A disk turns at 7500 RPM. Worst-case rotational latency? (b) Distance 3000 km, speed in cable 2 x 10^8 m/s: round-trip latency?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 5. CACHE OR DATABASE

Your home page shows "top 10 courses", recomputed from 1 million rows every time, but changes only when an admin edits courses. Design the caching.

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

Counts are **about** (keyword-matched, so rough). Of roughly 692 Quiz 1 questions about 27 touch this week; of 624 Quiz 2 about 55; of 1600 End Term about 119. A sample of about 62 distinct questions shows these repeating:

- **Predict the output of a `logging` program** (the most reliable one: about 6 such questions across quizzes and End Terms, with `INFO`, `WARNING`, `logging.exception`, and `argv` values).
- **Memory and bandwidth arithmetic**: DRAM bandwidth (64-bit, DDR, dual channel: 12.8, 25.6, 16 GB/s variants), disk RPM latency (6000 RPM and 5000 RPM), SSD vs HDD throughput, data in a network graph, mobile data in GB.
- **Latency and requests per second** (speed of light in cable, 1 KB responses, 20 ms round trip gives 50 per second).
- **Indexes**: purpose of an index; the two false index statements.
- **Caching in REST APIs** (find the false statements) and scale up vs scale out, ACID vs BASE, redundancy and replication.
- Memory hierarchy match-the-column (latency, density, throughput).

WATCH OUT - REAL EXAM TRAP

- The debug line vanishes when the level is higher; read the level first, then each call.
- Do not forget `DDR` doubles transfers and `dual channel` doubles again.
- Round trip is twice the one-way delay; requests/second = 1000 / (RTT in ms) for one at a time.
- "Index" is not encryption, replication or metadata; it is speed.
- Caching is for reads; false options say it *raises* load or latency.

Quick quiz

QUICK QUIZ

`basicConfig(level=logging.INFO, format='%(levelname)s - %(message)s')`. The code calls `logging.info("start")` then, for `python log.py 25 5`, `logging.debug("factor")`. The terminal shows...

- `INFO - start` and `DEBUG - factor`
- `INFO - start` only
- `DEBUG - factor` only
- nothing

Check

Level is WARNING. The program calls `logging.warning("a")`, `logging.info("b")`, `logging.error("c")`. Which lines print?

- a, b, c
- only c
- a and c
- b and c

Check

DRAM, 64-bit bus, 800 MHz, DDR, dual channel. Maximum bandwidth?

- 12.8 GB/s
- 25.6 GB/s
- 51.2 GB/s
- 102.4 GB/s

Check

Select all FALSE statements about indexes. (select all that apply)

- An index can be created on only one column.
- Indexes cannot be created on columns with duplicate values.
- An index speeds up data retrieval.
- Indexes slow down inserts and updates.

Check

A browser sends `If-None-Match: "v1"` and the resource is unchanged. The server replies...

- 200 with the full body

304 with no body

404

201

Check

A disk spins at 6000 RPM. Worst-case wait before data transfer starts?

5 ms

10 ms

20 ms

60 ms

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- **Why use `logging` instead of `print`?**
- **What is an index and what does it cost?**
- **How would you check that a query uses your index?**
- **Explain caching and the main risk.**
- **What is the difference between latency and throughput?**
- **Why do you want stateless servers when you scale out?**
- **What is the N+1 problem and how do you fix it?**
- **Why paginate an API?**

Recap

REMEMBER THIS

- **Logging:** levels 10/20/30/40/50, default WARNING; a message prints only if its level is at or above the configured one; `logging.exception` adds a traceback.
- **Debugging:** debug mode (reload plus error page, never in production), read the traceback from the bottom, `breakpoint()` and `pdb`, `flask shell`.
- **Indexes** speed up reads (B-tree, about logarithmic) and slow writes; check with `EXPLAIN QUERY PLAN`; multi-column and duplicates are fine.
- **FTS5**: virtual table, `MATCH`, whole-word matching, triggers keep it in sync.
- **Caching:** key, TTL, invalidation; client side with `Cache-Control` and `ETag` (304); server side with Flask-Caching; only GET is cacheable by default.
- **Numbers:** bandwidth = bytes x transfers x channels; latency = distance / speed; RPM latency = 60000 / RPM ms; data = rate x time.
- **Scale out** with load balancer and stateless servers; push slow work to a queue; avoid N+1; paginate.

CODE AND FACTS TO KEEP

- Levels: DEBUG 10, INFO 20, WARNING 30 (default), ERROR 40, CRITICAL 50
- `logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')`
- `breakpoint()`; pdb: `n`, `s`, `c`, `p x`, `l`, `q`; `flask --app app shell`
- `CREATE INDEX idx ON t(col);` `EXPLAIN QUERY PLAN SELECT ...;` SCAN vs SEARCH USING INDEX
- `CREATE VIRTUAL TABLE t USING fts5(a, b);` `WHERE t MATCH 'word OR pre*'`
- `@cache.cached(timeout=60)`, `@cache.memoize()`, `cache.delete(key)`, `cache.clear()`
- `Cache-Control: public, max-age=60`, `ETag` and `If-None-Match` give 304
- DDR bandwidth = (bus bytes) x (2 x clock) x channels; worst disk latency = 60000 / RPM ms
- Round trip = 2 x distance / speed; requests per second = 1000 / RTT ms
- Data = rate (Mbit/s) x seconds / 8 (MB); 1.5 million x 4 KB = 6 GB

MAD1 week 8: The frontend: JavaScript, the DOM and fetch

> **Official syllabus line:** Application Frontend

The one-minute story

THINK OF IT LIKE THIS

A restaurant with a waiter who thinks. Until now your Flask app was a kitchen that sent out a *whole new menu* (a full HTML page) every time anything changed. JavaScript puts a **clever waiter at the table** (inside the browser). When you tick "no onions", the waiter crosses the word out on your own card (**the DOM**) without walking back to the kitchen. When something really needs the kitchen (save my order), the waiter phones it with a short note in a standard form (**fetch + JSON**), listens to the reply, and updates one line on your card.

HTML is the content, CSS is the look, JavaScript is the behaviour. The kitchen (Flask) stays in charge of **truth and security**; the waiter only makes the experience smooth.

By the end of this week you can:

- **add** JavaScript to a page correctly (`script`, `defer`, external files);
- **write** variables, functions, arrays and objects in JavaScript and predict their results;
- **select and change** the DOM (`getElementById`, `querySelector`, `textContent`, `innerHTML`, `createElement`, `classList`, `style`);
- **handle events** and read forms with JavaScript;
- **call a Flask JSON API** with `fetch` (GET and POST), handle status codes and errors;

- **explain** client vs server rendering, localStorage vs cookies, CORS, XSS risk of `innerHTML`, and what Vue is for.

Words you need

WORDS YOU NEED

- **JavaScript (JS)** - the programming language that runs *inside the browser* and can change the page.
- **DOM** (Document Object Model) - the browser's live tree of objects built from your HTML; JS reads and changes the DOM.
- **Event** - something that happens: a click, a key press, a form submit, the page finishing loading.
- **Event listener** - a function that runs when a given event happens on an element.
- **JSON** - a text format for data: objects, arrays, strings, numbers, true/false/null.
- **AJAX** - updating part of a page by talking to the server in the background, without a full reload.
- **Fetch API** - the modern browser function `fetch(url, options)` that makes HTTP requests and returns a Promise.
- **Promise / async / await** - a way to write code that waits for a slow thing (network) without freezing the page.
- **Client-side rendering** - the browser builds the HTML from data using JS.
- **Server-side rendering** - the server builds the HTML (Jinja) and sends it ready.
- **localStorage** - small key-value storage kept in the browser, never sent to the server automatically.
- **CORS** - the browser rule that blocks a page from reading responses from a *different origin* unless that server allows it.
- **XSS** (cross-site scripting) - an attacker's script running inside your page.
- **Framework** - a library that structures a big frontend (Vue, React, Angular).

Idea 1: JavaScript in HTML, and a JS refresher

Put JS in a `script` tag, inline or in an external file. Where the tag sits and `defer` matter because **JS that touches an element must run after the element exists**.

```
<!DOCTYPE html>
<html>
<head>
  <title>Hello JS</title>
  <script src="app.js" defer></script>   <!— external, runs after the HTML is parsed —
>
</head>
<body>
  <h1 id="title">Hi</h1>
  <script>
```

```
    console.log("inline script, h1 exists:", document.getElementById("title") !== null);
  </script>
</body>
</html>
```

ATTRIBUTE	BEHAVIOUR
none	download and run immediately, **blocks** the parsing
`defer`	download in parallel, run **after** the whole HTML is parsed, in order
`async`	download in parallel, run **as soon as ready**, order not guaranteed

Classic rule: a script in `head` without `defer` runs *before* the body exists, so `document.getElementById("title")` is `null`. Fix: put the script at the end of `body`, or use `defer`, or wait for the `DOMContentLoaded` event. In Flask, JS files go in `static/` and are linked with `url_for('static', filename='app.js')`.

`console.log()` prints to the browser's **console** (developer tools, key F12). It is *not* `alert()` (a popup) nor `document.write()` (writes into the page).

A refresher on the language

```
let n = 5;            // can be reassigned
const name = "Asha";  // cannot be reassigned
var old = 1;          // old style; avoid

function square(x) { return x * x; }
const cube = x => x * x * x;          // arrow function

const marks = [40, 90, 65];           // array
marks.push(70);                       // add at end
console.log(marks.length, marks[0]);  // 4 40
console.log(marks.map(m => m + 5));        // [45, 95, 70, 75]
console.log(marks.filter(m => m >= 65));   // [90, 65, 70]

const user = { name: "Ravi", city: "Pune" };   // object
console.log(user.name, user["city"]);          // Ravi Pune
const { name: nm } = user;                     // destructuring
console.log(`Hello ${user.name}`);             // template string
```

Surprises that exams love (all checked in Node):

```
console.log("5" + 3);     // "53"  (+ with a string joins)
console.log("5" - 3);     // 2     (- converts to numbers)
```

```
console.log(5 == "5");    // true  (loose equality converts)
console.log(5 === "5");   // false (strict: also compares type)
console.log(typeof null); // "object" (an old quirk)
console.log([10, 9, 1].sort());  // [1, 10, 9]  (sorts as text!)
console.log(0.1 + 0.2);   // 0.30000000000000004
```

`let` is block-scoped; `var` is not: three callbacks made in a `var` loop all see the final `i` (`[3,3,3]`), but with `let` they see `[0,1,2]`. **Always use `===`, and `let` / `const`.**

Idea 2: The DOM

The browser turns your HTML into a **tree of objects**. JS changes the tree; the screen updates. **Changing the DOM does not change your HTML file** and the DOM is **not identical** to the source: it is built *before* your JS runs and JS can change it afterwards. (Both were in a real paper. JS is the usual way to use the DOM, but it is a language-neutral interface: "DOM can only be manipulated with JS" is false, and "its structure is rigid" is false.)

THE IDEA

Pick an element, then change it.

TASK	CODE
by id	`document.getElementById("msg")`
first match of a CSS selector	`document.querySelector("#msg")`, `".card"`, `"ul li"`
all matches	`document.querySelectorAll("li")`
all by tag	`document.getElementsByTagName("li")` (a **live** collection)
change text safely	`el.textContent = "Hi"` (or `el.innerText`)
change HTML (risky)	`el.innerHTML = "<b>Hi</b>"`
form value	`input.value`
style	`el.style.fontWeight = "bold"` (camelCase)
classes	`el.classList.add("hot")`, `remove`, `toggle`, `contains`
attribute	`el.setAttribute("href", "/x")`
create / insert / remove	`document.createElement("li")`, `parent.appendChild(li)`, `li.remove()`

`querySelector` returns the **first** match, or `null` if none. `querySelectorAll` returns all matches.

SOLVED EXAMPLE: 1. MAKING TEXT BOLD (A REAL QUESTION, THREE TIMES)

Question. A page has `<input id="txt1" value="Welcome … ">` and a button calling `doSomething()` . Which code makes the text in the box bold?

```
function doSomething(){
  const txt1 = document.getElementById("txt1");    // the element
  txt1.style.fontWeight = "bold";                   // style is an OBJECT;
fontWeight is camelCase
}
```

Solution. Correct: `document.getElementById("txt1").style.fontWeight = "bold"` (with or without a temporary variable). **Wrong:** `.style = "bold"` (we tried it: it sets an invalid style string and *nothing* becomes bold), and `getElementById("txt1").value` followed by `.style` … because `.value` is a **string**, and a string has no `style` (it throws `TypeError`). CSS `font-weight` becomes JS `fontWeight` ; `background-color` becomes `backgroundColor` .

SOLVED EXAMPLE: 2. ADDING TO A LIST

Question. The page has `<input id="txt_course">` , a button "Add" and an empty `<ul id="courses">` . Complete `add_item()` so each typed course appears as a list item.

```
function add_item(){
  const course = document.getElementById("txt_course").value;
  document.getElementById("txt_course").value = "";
  const courses = document.getElementById("courses");      // the UL, not the
input
  courses.innerHTML = courses.innerHTML + "<li>" + course + "</li>";
}
```

Solution. Take the *input's value*, then append `"<li> … </li>"` to the **ul** (`getElementById("courses")`). Using the input element as `courses` is the trap; assigning with `=` instead of `+=` would *replace* the earlier items. After adding "Java" and "PDSA" the ul holds `<li>Java</li><li>PDSA</li>` . A cleaner way:

```
const li = document.createElement("li");
li.textContent = course;
courses.appendChild(li);
```

SOLVED EXAMPLE: 3. CREATEELEMENT BUILDS A ROW

Question. What does this do when the button is clicked?

```
function perform(){
  let row = document.createElement("tr");
  let td1 = document.createElement("td"); td1.innerText = "name_1";
  let td2 = document.createElement("td"); td2.innerText = "city_1";
  row.appendChild(td1); row.appendChild(td2);
  document.getElementById("rows").appendChild(row);
}
```

Solution. It adds one new **row** (`tr`) to the element with id `rows` (the table body), with `name_1` in the **first** cell and `city_1` in the **second**. Order of `appendChild` is the order of the columns. Each click adds another row.

SOLVED EXAMPLE: 4. A LIVE COLLECTION

Question. `<ul id="lst"><li>App Dev-1</li></ul>` . A script does `let items = document.getElementsByTagName("li");` then `document.getElementById("lst").innerHTML += "<li>App Dev-2</li><li>App dev projects</li>";` then `console.log(items.length)` . Output?

Solution. 3. `getElementsByTagName` returns a **live** collection: it updates when the DOM changes, so it now sees all three `li` . (`querySelectorAll` returns a *static* list; take its length before and after if you want to see the difference.) We ran this in a DOM simulator and got 3.

CSS reminders the papers use with the DOM

Many rendering questions mix JS with CSS `display` . Quick table:

DISPLAY	BEHAVIOUR
`block` (div, p, h1)	starts on a new line, takes the full width
`inline` (span, a)	flows within the text, width and height ignored
`inline-block`	flows in a line **and** accepts width/height; elements sit **side by side** until the line is full
`flex` (on the parent)	children sit in a **row** by default, even if they are divs

So a `div` made `inline-block` appears on the same line as the next, and a flex container shows `1 2 3` as three boxes **in a row**. An `id` rule (`#id1`) beats a class rule (`.class1`), which beats a tag rule (`span`). Also: `ol` is numbered, `ul` is bulleted, and a nested list has its own numbering.

WATCH OUT - REAL EXAM TRAP

- `.style = "bold"` and `.value.style` do nothing useful; use `.style.fontWeight`.
- `innerHTML +=` re-parses everything and drops listeners on the old children.
- `getElementById` returns **one** element (`null` if absent); `getElementsByTagName` returns a collection, so you index it (`[0]`).
- Two ids in an HTML page must not clash, and the id in JS has **no** `#`, while `querySelector` needs `#`.

Idea 3: Events, forms and validation

An **event listener** runs your function when something happens. Three ways, all legal (a real question's answer was "All of these"):

```
<button id="myBtn">Click Me</button>
<button onclick="alert('Clicked!')">Click Me 2</button>
<script>
  document.getElementById("myBtn").onclick = function(){ alert("Clicked!"); };
  document.getElementById("myBtn").addEventListener("click", function(){
alert("Clicked!"); });
</script>
```

`addEventListener` is preferred because you can attach **many** listeners to one event and remove them. The listener gets an **event object** (`e.target`, `e.preventDefault()`).

```
document.getElementById("btn").addEventListener("click", () => {
  document.querySelector("#msg").textContent = "Welcome to JavaScript!";
});
```

Both `getElementById` and `querySelector("#msg")` work, and both `textContent` and `innerHTML` change the visible text. But `msg.text = "..."` does nothing visible (there is no `.text` property on a div), and it relies on the id as a global name.

Forms. Read `input.value`; stop the normal submit with `e.preventDefault()` when JS will handle it.

```
<form id="f">
  <input id="title" required minlength="3">
  <button>Add</button>
</form>
<script>
  document.getElementById("f").addEventListener("submit", e => {
    e.preventDefault();                          // do not reload the page
    const t = document.getElementById("title").value.trim();
```

```
    console.log("will send:", t);
  });
</script>
```

Validation: frontend vs backend. HTML5 attributes (`required`, `minlength`, `type="email"`, `pattern`) and a JS function such as `validateForm()` give **instant feedback** in the browser (this is *client-side validation*; the browser's prompt on an empty `required` field reads "Please fill out this field." and the JS function does not run on the server). But **anyone can bypass the browser** (curl, dev tools), so the **backend must validate again**; frontend validation is for the user's convenience, never for security. The CSS pseudo classes `:valid` and `:invalid` can colour a field as the user types (an input with a `minlength` turns from red to green once the minimum length is reached and stays green while typing is still allowed).

A form with **no** `method` sends **GET**: the data goes in the URL. `method="post"` puts it in the body.

Idea 4: JSON and the Fetch API (AJAX)

JSON looks like a JS object but is **text**: keys in double quotes, no trailing commas, no comments, values only string, number, boolean, null, array, object.

```
const txt = '{"name":"Balu","age":30,"city":"New Delhi"}';
const obj = JSON.parse(txt);          // text → object
console.log(obj.age * 3);             // 90
console.log(JSON.stringify(obj));     // object → text:
{"name":"Balu","age":30,"city":"New Delhi"}
```

A real paper put this in `innerHTML` as `"Hi\n" + obj.name + ",\t" + obj.city + "," + obj.age*3`. The page shows **Hi Balu, New Delhi,90**: the string really contains a newline and a tab, but HTML **collapses whitespace** to one space, and `age*3` is a *number* 90 (not "303"). Notes: `JSON.stringify` drops `undefined` values (`{a:1,c:undefined}` becomes `{"a":1}`); `JSON.parse` on bad JSON such as `{'a':1}` throws `SyntaxError`.

fetch with a Flask API

Our Flask API (we ran it):

```
@app.get("/api/items")
def lst(): return jsonify(items)

@app.post("/api/items")
def add():
    data = request.get_json(silent=True)
    if not data or "title" not in data:
        return jsonify(error="title required"), 400
    it = {"id": len(items) + 1, "title": data["title"]}
```

```
    items.append(it)
    return jsonify(it), 201

@app.get("/api/items/<int:i>")
def one(i): ...                          # returns {"error": "not found"}, 404 if missing
```

The browser side, with **async/await** (the modern style):

```
async function loadItems() {
  const res = await fetch("/api/items");          // GET by default
  if (!res.ok) { throw new Error("HTTP " + res.status); }
  const items = await res.json();                  // parse the body
  const ul = document.getElementById("list");
  ul.innerHTML = "";
  for (const it of items) {
    const li = document.createElement("li");
    li.textContent = it.title;                     // textContent: safe
    ul.appendChild(li);
  }
}

async function addItem(title) {
  const res = await fetch("/api/items", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ title: title })
  });
  const data = await res.json();
  if (res.status === 201) { loadItems(); } else { alert(data.error); }
}
```

Same idea with **.then**: `fetch(url).then(r => r.json()).then(data => ...).catch(err => ...)`.

Results from our test run:

CALL	`RES.STATUS`	`RES.OK`
GET `/api/items`	200	true
POST with JSON header and body	201	true
GET `/api/items/99`	404	**false**, but fetch did **not** throw
POST without `Content-Type: application/json`	400	false (Flask's `get_json(silent=True)` gave nothing)

CALL	`RES.STATUS`	`RES.OK`
server not running	none	fetch **throws** `TypeError` (network error)

> **THE IDEA**
>
> **The two kinds of failure.** `fetch` **rejects** (goes to `catch`) only on a **network failure**. A 404 or 500 is a *successful* HTTP exchange, so you must check `res.ok` (true for 200-299) or `res.status` yourself. For POST JSON you need **both** the `Content-Type` header and `JSON.stringify` on the body. `await res.json()` is also asynchronous.

AJAX is this idea: a page that updates part of itself with background requests, without reloading. **Debugging:** in the browser's **dev tools** use the **Console** (errors and `console.log`), **Network** (see each request, its status, headers and response body) and **Elements** (the live DOM). `console.log(res.status)` is your friend.

> **IN YOUR PROJECT**
>
> Add one `fetch` call to your project: for example a "mark as done" button that does `PUT /api/tasks/3` and updates the row without reload. In the viva be ready to say what happens when the API returns 401, 404 or 500 and show where you check `res.ok`.

Idea 5: Where is the page built? Rendering styles and frameworks

	SERVER-SIDE RENDERING (SSR, JINJA)	CLIENT-SIDE RENDERING (CSR, JS + API)
Server sends	finished HTML	an almost empty page, JS, and later JSON
First paint	fast, works without JS	slower first load, then very smooth
Next page	full reload	only the changed part
SEO, simple sites	easy	harder
Fits	MAD1 style apps	MAD2 style apps with a JSON API

Progressive enhancement: build a page that works with plain HTML and forms (SSR), then **add JS on top** for comfort. If JS fails, the site still works. This is a good, safe default design.

Frameworks (Vue, React, Angular) give structure for large UIs: **components**, **reactive data** (change the data and the screen updates automatically), and routing. They are *frontend* frameworks. **Node.js is not a frontend framework**: it runs JS on the server. **WebAssembly** is a binary format, not a *web component*; the real web-component standards are **Custom Elements, Shadow DOM and HTML Templates**. You will use **Vue in MAD2**; for MAD1 you only need "why use one": less manual DOM code. JS is *not always* the core of a frontend; HTML and CSS are the core and JS adds extra functionality.

Idea 6: Storing things in the browser, CORS and XSS

	LOCALSTORAGE	COOKIES
Size	about 5 MB	about 4 KB
Sent to server?	**no**, JS only	**yes**, automatically with every request to that site
Expiry	never, until cleared	set by `Expires` / `Max-Age`, or when the session ends
Typical use	UI preferences, a draft	session id, "remember me"

```
localStorage.setItem("theme", "dark");
console.log(localStorage.getItem("theme"));   // "dark"
localStorage.removeItem("theme");
localStorage.setItem("user", JSON.stringify({name: "Asha"}));   // store objects as text
```

localStorage holds **strings only**, hence `JSON.stringify`. Anything JS can read, an attacker script can read too, so do **not** keep secrets there.

CORS in one paragraph. The browser enforces the **same-origin policy**: a page from `http://a.com` may not *read* a response from `http://b.com` (a different scheme, host or port) unless server b answers with `Access-Control-Allow-Origin: http://a.com`. In Flask-CORS you write `CORS(app)`. If your page and API are served by the same Flask app, there is no CORS issue. CORS is enforced **by the browser**, not by the server, so curl is not affected.

XSS and `innerHTML`. If you put **user-supplied text** into `innerHTML`, the browser parses it as HTML, so a comment containing an `img` tag with an `onerror` handler would **run an attacker's script** in your visitors' browsers. Checked in a DOM simulator:

```
el.textContent = '<img src=x onerror="alert(1)">';  // shows the text literally:
&lt;img ... &gt;
el.innerHTML   = '<img src=x onerror="alert(1)">';  // creates a real img element
(dangerous)
```

Rule: **`textContent` for plain text; `innerHTML` only for HTML you wrote yourself.** More about XSS and its server-side twin in week 9.

Your turn

YOUR TURN: 1. PREDICT THE JS

What do these print? `console.log("2" + 2)`, `console.log("2" * "3")`, `console.log([3, 20, 100].sort())`, `console.log(typeof [])`.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 2. CHANGE THE DOM

`<p id="info" class="note">Hello</p>`. Write JS that (a) changes the text to `Welcome`, (b) adds the class `hot`, (c) makes the colour red.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 3. SPOT THE BUG

`<ul id="todo"></ul>`. The code `document.getElementById("todo").innerHTML = "<li>" + text + "</li>"` is called for every new task, but only the last task is visible. Why, and fix it.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 4. FETCH AND STATUS CODES

Write a function that POSTs `{"title": t}` to `/api/items` and shows `error` from the response if the status is not 201.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 5. SAFE OR NOT

A comment box shows what users type with `div.innerHTML = comment`. Why is this dangerous and what is the fix?

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

About 16 of ~692 Quiz 1 questions, about 5 of ~624 Quiz 2 questions and about 47 of ~1600 End Term questions match this week's keywords (rough keyword counts). The real questions are *far more about the DOM and small JS snippets than about fetch*. The sample of about 26 questions repeats:

- **Make text bold with JS**: asked in three different End Terms (answer: `getElementById("txt1").style.fontWeight = "bold"`), also the variant with `.value` (wrong).
- **Add to a list or table dynamically**: `innerHTML` on the correct element (the `ul`), and `createElement("tr")` with two `td` cells.
- **Event listeners**: `onclick` vs `addEventListener`, "All of these".
- **True/false about the DOM**: it is created before the JS runs; JS can change it; changing it does not change the HTML file; HTML and DOM are not identical.
- `console.log` vs `alert` vs `document.write`; `JSON.parse` with arithmetic (`age*3` = 90).
- **Frontend frameworks** (Node JS is *not* one), web components, frontend vs backend validation, HTML vs CSS vs JS roles.
- **Rendering HTML+CSS** (`inline-block`, flex row, specificity).

WATCH OUT - REAL EXAM TRAP

- `style = "bold"`, `.value.style`, appending to the wrong element, `=` instead of `+=`.
- `getElementsByTagName` is live; `querySelectorAll` is not.
- Frontend validation is not security; the backend must repeat the checks.
- HTML collapses whitespace: `\n` and `\t` show as one space in a page.
- `fetch` does not throw on 404; check `res.ok`.

Quick quiz

QUICK QUIZ

Which code bolds the text of the element with id `txt1`?

- `document.getElementById("txt1").style = "bold";`
- `document.getElementById("txt1").style.fontWeight = "bold";`
- `document.getElementById("txt1").value.style.fontWeight = "bold";`
- `document.getElementsByName("txt1").style.fontWeight = "bold";`

Check

Select all TRUE statements about the DOM. (select all that apply)

- JS can interact with DOM elements.
- The DOM is created before the JS runs.
- Changing the DOM changes the HTML source file.
- HTML and the DOM are identical.

Check

`const obj = JSON.parse('{"name":"Balu","age":30}'); console.log(obj.age * 3);` **prints...**

- 303
- NaN
- 90
- 30

Check

Which is NOT a frontend framework?

- Vue JS
- React JS
- Node JS

Angular JS

Check

A `fetch` call to your API returns status 404. What happens?

the Promise rejects and `catch` runs

the Promise resolves; `res.ok` is false

the page reloads

JavaScript stops running

Check

Select all TRUE statements about validation. (select all that apply)

Frontend validation gives immediate feedback.

Backend validation is mandatory for security.

Frontend validation is enough for security.

Backend validation is unnecessary if the form has `required`.

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- **What is the DOM?**
- **Explain the difference between `textContent` and `innerHTML`, and which is safer.**
- **How does your page talk to your Flask backend without reloading?**
- **What is the difference between server-side and client-side rendering?**
- **Why validate on the server when you already validate in JavaScript?**
- **When would you pick localStorage and when cookies?**
- **What is CORS?**
- **How do you debug JavaScript?**

Recap

REMEMBER THIS

- Load JS with `script`; use `defer` or put it at the end of body so elements exist; `console.log` writes to the console.
- JS basics: `let` / `const`, arrow functions, arrays (`map`, `filter`), objects; use `===`.
- **DOM**: select (`getElementById`, `querySelector`), change (`textContent`, `style.fontWeight`, `classList`), build (`createElement`, `appendChild`).
- **Events**: `addEventListener("click", fn)`; forms use `e.preventDefault()`; validate on the client for UX and on the server for safety.
- **JSON**: `JSON.parse` and `JSON.stringify`; **fetch** needs `Content-Type` and a stringified body for POST; check `res.ok`.
- **Render** on the server (Jinja) or the client (JS + API); progressive enhancement; Vue is for large UIs.
- localStorage (browser only) vs cookies (sent every request); CORS is a browser rule; `innerHTML` with user text is XSS.

CODE AND FACTS TO KEEP

- `<script src="app.js" defer></script>`
- `document.getElementById("id")` , `document.querySelector("#id .cls")` , `querySelectorAll("li")`
- `el.textContent` , `el.innerHTML` , `el.style.fontWeight = "bold"` , `el.classList.add("x")`
- `const li = document.createElement("li"); ul.appendChild(li)`
- `el.addEventListener("click", e => { e.preventDefault(); })`
- `JSON.parse(text)` , `JSON.stringify(obj)`
- `const res = await fetch(url, {method:"POST", headers:{"Content-Type":"application/json"}, body: JSON.stringify(data)})` , `res.ok` , `res.status` , `await res.json()`
- `localStorage.setItem(k, v)` , `getItem(k)` , `removeItem(k)`
- `display` : block, inline, inline-block (side by side), flex (row)

MAD1 week 9: Security: authentication, sessions, tokens and attacks

> **Official syllabus line:** Application Security

The one-minute story

THINK OF IT LIKE THIS

A building with guards. A visitor arrives. At the gate the guard asks "who are you?" and checks an ID card (**authentication**). Inside, a second question is asked at every door: "are you *allowed* in this room?" (**authorisation**). You do not keep a copy of the visitor's secret PIN: you keep only a scrambled fingerprint of it, so a thief who steals your register cannot read PINs (**password hashing**). After the check, the guard gives a numbered **wristband** (**session cookie**) so he need not re-check at every door. Meanwhile attackers try tricks: slipping a forged note to a clerk (**CSRF**), leaving a booby-trapped sticker on the notice board that harms whoever reads it (**XSS**), or whispering clever words into a form so the database obeys *them* (**SQL injection**). Security is mostly a list of habits, each one closing one of these doors.

By the end of this week you can:

- **explain** authentication vs authorisation, and RBAC vs ABAC;
- **store and check passwords** safely with `generate_password_hash` and `check_password_hash` ;
- **explain and build** session-based login with cookies, Flask `session` and Flask-Login (`login_user` , `@login_required` , roles);
- **name and prevent** CSRF, XSS and SQL injection;

- **explain** HTTPS/TLS, secrets in environment variables, input validation, rate limiting, token authentication and least privilege;
- **predict the output** of Flask login/session snippets and use a security checklist for your project.

Words you need

WORDS YOU NEED

- **Authentication (authn)** - proving *who you are* (username and password, token, fingerprint).
- **Authorisation (authz)** - deciding *what you may do* once known (admin or normal user).
- **Hash** - a one-way scramble of data into a fixed-size fingerprint; it cannot be reversed.
- **Salt** - random extra data mixed in before hashing so equal passwords get different hashes.
- **Session** - the server's memory of a logged-in user between requests.
- **Cookie** - a small named value the server asks the browser to store and send back with every request.
- **RBAC** - role-based access control: rights come from the user's role.
- **ABAC** - attribute-based access control: rights come from attributes (department, location, number of posts).
- **CSRF** - cross-site request forgery: a hostile page makes your browser send a request you did not intend.
- **XSS** - cross-site scripting: attacker's script runs in your page.
- **SQL injection** - attacker's text becomes part of your SQL command.
- **HTTPS / TLS** - HTTP carried inside an encrypted, authenticated channel.
- **Token** - a signed string the client sends (for example in a header) instead of a cookie session.
- **Least privilege** - give each user and program only the access it needs.

Idea 1: Threats and the two big questions

The **OWASP Top 10** is the well-known list of the most common web application weaknesses. You do not need to memorise it; know the family: **broken access control**, **cryptographic failures** (plain-text passwords, no HTTPS), **injection** (SQL, XSS), **insecure design**, **security misconfiguration** (debug mode on, default passwords), **vulnerable components**, **authentication failures**, **logging failures**. Attacks come from the **browser side** (the user, or a script in their page) and from the **network** (someone reading the wire).

THE IDEA

Authentication = who are you? Authorisation = what may you do? Authentication comes first. Wrong credentials: **401 Unauthorized** (really "unauthenticated"). Known user, not allowed: **403 Forbidden**. Some apps return **404** for private things to hide that they exist. Mnemonic: *401 means log in, 403 means you are logged in but not allowed.*

Access control models.

MODEL	DECISION BASED ON	EXAMPLE
RBAC	the user's **role**	"admin can delete courses", "student can only view"
ABAC	**attributes** of user, resource, context	a forum unlocks image posting after 25 posts; a cloud service allows files by department, job title or location
Owner-based (discretionary)	the owner decides	file owner shares a document

Real paper answers: "a forum unlocks images once a user makes more than 25 posts" is **ABAC**; "access by job role in an organisation" is **RBAC**; "grants access by department, job title or current location" is **ABAC**.

HTTP authentication schemes (a real match-the-column):

MECHANISM	ONE-LINE FACT
HTTPS	an **encrypted channel**
Cookies	must be **sent with each request** (the browser does it automatically)
Digest authentication	uses a **hash function** on the password, so the password is not sent in clear
HTTP Basic authentication	`username:password` sent **base64-encoded**, which is encoding, not encryption, so it is only safe over HTTPS

Do not mix: base64 can be decoded by anyone; a hash cannot be reversed; encryption can be reversed *with a key*.

> **WATCH OUT - REAL EXAM TRAP**
>
> A real question gave a login form with a wrong-password path that shows nothing. The heuristic broken is **visibility of system status** (the user should always be told what is happening). Also note: a form with **no** `method` uses **GET**, so a username and password would appear in the URL and the server logs. Login forms must use `method="post"`.

Idea 2: Passwords: hash, salt, never store plain text

If you store passwords as text, anyone who reads your database (a leak, a backup, a curious admin) learns every password, and people reuse passwords. So store only a **hash**.

Why not encryption? Encryption can be reversed by whoever holds the key, and the key sits on the same server. A hash cannot be reversed: you hash what the user types at login and compare the two hashes.

Why a salt? The same password always gives the same hash, so an attacker can precompute a huge table of hashes of common passwords (a **rainbow table**) and look yours up. A **salt** is random data added before hashing, stored with the hash, so two users with the same password get different hashes and

precomputed tables are useless. Good password hashes (bcrypt, scrypt, PBKDF2, Argon2) are also **deliberately slow**, to make guessing expensive. A plain fast hash such as SHA-256 on its own is a poor password hash.

```
from werkzeug.security import generate_password_hash, check_password_hash
h1 = generate_password_hash("secret")
h2 = generate_password_hash("secret")
print(h1 == h2)                          # False (different random salts)
print(check_password_hash(h1, "secret")) # True
print(check_password_hash(h1, "Secret")) # False
print(h1[:12], len(h1))                  # scrypt:32768:8:1$ ...   (length 162 on our
machine)
```

We ran it. The stored string contains the **method, its settings, the salt and the hash**, so `check_password_hash` needs only the stored string and the typed password. Older Werkzeug versions start with `pbkdf2:sha256:...`; both are fine. `bcrypt` (a separate library) works the same way: `bcrypt.hashpw(pw, bcrypt.gensalt())` and `bcrypt.checkpw(pw, hash)`.

SOLVED EXAMPLE: 1. SAFE REGISTRATION AND LOGIN

```
# register
user = User(name=name, pw=generate_password_hash(password))   # store the HASH
db.session.add(user); db.session.commit()

# login
user = User.query.filter_by(name=name).first()
if user and check_password_hash(user.pw, password):
    login_user(user)
else:
    return "bad credentials", 401        # same message for 'no such user' and
'wrong password'
```

Never compare with `user.pw == password`. Use the same error message for unknown user and wrong password, so attackers cannot learn which usernames exist.

WATCH OUT - REAL EXAM TRAP

Real questions store passwords as plain columns (`password = db.Column(db.String, nullable=False)`) just to test SQLAlchemy. That is fine for an *exercise*; in your **project** you store the hash. Also watch the classic logic bug below.

```
for vu in valid_users:
    if vu["uid"] == args["uid"] and vu["pwd"] == args["pwd"]:
        return "You are authorized!!"
    else:
        return "You are unauthorized!!"      # else is INSIDE the loop
```

For `uid=professor&pwd=54321` the first record is `student` , the test fails, the `else` returns at once, and the second record is never checked: the output is **You are unauthorized!!** (We ran this.) A real paper asked exactly this, and the credentials in the URL are also a security fault (query strings are logged).

Idea 3: Cookies and sessions

HTTP is **stateless**: each request is separate. To "remember" a user, the server gives the browser a **session id** in a **cookie**; the browser sends it back every time.

```
Server response:  Set-Cookie: session=abc123; HttpOnly; Secure; SameSite=Lax
Next request:     Cookie: session=abc123
```

Both statements in a real question were true: cookies let a web app **recognise users**, and they can hold login data, language, location and similar information.

COOKIE FLAG	MEANING	PROTECTS AGAINST
HttpOnly	JavaScript cannot read it (`document.cookie`)	XSS stealing the session id
Secure	sent only over HTTPS	sniffing on the network
SameSite=Lax/Strict	not sent on most cross-site requests	CSRF
`Max-Age` / `Expires`	lifetime	long-lived stolen cookies

Flask's `session` . Flask keeps the session **in the cookie itself**, as a **signed** (not encrypted) value. The signature uses `SECRET_KEY` , so the user cannot *change* it without detection, but the user *can* **read** it. We decoded one: the cookie held `{"role":"general","user":"a"}` in readable base64. So: never store secrets in `session` , and keep the `SECRET_KEY` secret. Setting `SESSION_COOKIE_HTTPONLY` , `SESSION_COOKIE_SECURE` , `SESSION_COOKIE_SAMESITE` configures the flags.

SOLVED EXAMPLE: 2. PREDICT THE STATUS CODES

Question. (Real End Term, rebuilt and run.)

```
app.config['SECRET_KEY'] = "yekterces"

@app.route('/login')
def log_in():
    user = request.args['user']
    role = request.args['role'] if 'role' in request.args else 'general'
    session['user'], session['role'] = user, role
    return "Logged in successfully!"

@app.route('/home')
def land():
    if 'user' in session:
        if session['role'] == 'admin':
            return f"Welcome {session['user']}"
        return abort(401)
    return abort(404)

@app.route('/logout')
def log_out():
    session.pop('user', None); session.pop('role', None)
    return "Logged out sucessfully!"
```

Visit in order: `/home`, `/login?user=a&role=general`, `/home`, `/login?user=a&role=admin`, `/home`, `/logout`, `/home`.

Solution. 404, 200, 401, 200, 200, 200, 404. No session yet: 404. Logged in as a general user: 401. Logged in as admin: 200 and `Welcome a`. After logout the keys are gone: 404 again. The browser (client) keeps the cookie between visits; that is why the same URL changes its answer.

Session memory arithmetic. A server-side session store with 1.5 million active sessions of 4 KB each needs 1.5 million x 4 KB = 6 GB of RAM (this exact calculation appeared in a paper; see week 7). It also shows why server sessions need a shared store (Redis) once you scale out.

Idea 4: Flask-Login and roles

Flask-Login manages "who is logged in" for you. Four ingredients:

```
from flask_login import LoginManager, UserMixin, login_user, logout_user,
login_required, current_user

login_manager = LoginManager(app)
```

```
login_manager.login_view = "login"           # where to send anonymous users

class User(UserMixin, db.Model):             # UserMixin adds is_authenticated etc.
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String, unique=True)
    pw = db.Column(db.String)
    role = db.Column(db.String, default="user")

@login_manager.user_loader
def load_user(user_id):                      # called on each request with the id from
the session
    return db.session.get(User, int(user_id))
```

```
@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        u = User.query.filter_by(name=request.form["name"]).first()
        if u and check_password_hash(u.pw, request.form["pw"]):
            login_user(u)                    # the USER OBJECT, not user.id, not
user.name
            return redirect("/dash")
        return "bad credentials", 401
    return render_template("login.html")

@app.route("/dash")
@login_required                              # anonymous users are redirected to the
login page
def dash():
    return "Welcome " + current_user.name

@app.route("/logout")
def logout():
    logout_user(); return "bye"
```

We ran all of this with the test client. Observed: anonymous `/dash` gives a **302 redirect to `/login?next=%2Fdash`** (not 404, not 403); a wrong password gives 401; a right one gives a 302 to `/dash` and a `session=…` cookie; then `/dash` returns `Welcome b`. `current_user` is the logged-in `User` object (an anonymous placeholder if nobody is logged in).

Two real questions here: the correct registration call is **`login_user(user)`** (the object), and a real snippet that called `login_user(u1)` for `Rose` then `logout_user()` printed, in order, `Rose logged in`, `current user is Rose`, `logged out` when visited as `/`, `/home`, `/logout`.

Roles (RBAC). With Flask-Security you attach roles to users and write `@roles_required('admin')` (all roles needed) or `@roles_accepted('admin', 'editor')` (any one). With plain Flask-Login you can write

the decorator yourself:

```
def roles_required(*roles):
    def deco(fn):
        @wraps(fn)
        @login_required
        def inner(*a, **k):
            if current_user.role not in roles:
                return "forbidden", 403          # authenticated but not authorised
            return fn(*a, **k)
        return inner
    return deco

@app.route("/admin")
@roles_required("admin")
def admin(): return "admin area"
```

Tested: user `b` (role user) gets **403** on `/admin`; user `a` (role admin) gets `admin area`. A user can have many roles; model it with a many-to-many table (roles in a real paper are a `Role` table with a relationship back to `User`).

> **IN YOUR PROJECT**
> Protect every route that changes data. In the viva be ready to show `@login_required` on the dashboard, a role check on the admin pages, and the line where you hash the password. Also explain what the *user_loader* does: "Flask-Login stores only the id in the session; this function turns the id back into the user object on each request."

Idea 5: CSRF, XSS and SQL injection

CSRF (cross-site request forgery)

You are logged in to your bank. In another tab you open a hostile page that contains a hidden form posting to `bank.com/transfer`. **Your browser attaches your cookie automatically**, so the bank thinks *you* sent it. Defences:

- **CSRF token**: a random secret value put in each form (`<input type="hidden" name="csrf" ...>`) and kept in the session. The server accepts a POST only if the submitted token equals the session's. The hostile page cannot read your page, so it cannot know the token.
- **SameSite cookies**, and never change data with GET.
- Flask-WTF does this for you (`CSRFProtect(app)` and `{{ form.csrf_token }}`). A manual version we tested:

```
def token():
    if "csrf" not in session: session["csrf"] = secrets.token_hex(16)
```

```
    return session["csrf"]

@app.post("/pay")
def pay():
    if request.form.get("csrf") ≠ session.get("csrf"): abort(403)
    return "paid"
```

Result: no token gives **403**, a wrong token gives **403**, the right token gives `paid`.

XSS (cross-site scripting)

If user-written text is placed into a page **as HTML**, an attacker's `script` runs in the victim's browser (it can steal data, act as the user). Defence: **escape output**. **Jinja autoescape** (on for `.html` templates) turns `<` into `<`:

```
render_template_string("<p>{{ x }}</p>", x="<script>alert(1)</script>")
#  <p>&lt;script&gt;alert(1)&lt;/script&gt;</p>        (safe: shown as text)
render_template_string("<p>{{ x|safe }}</p>", x="<script>alert(1)</script>")
#  <p><script>alert(1)</script></p>                    (dangerous: runs)
```

Never use `|safe` or `Markup()` on user input. On the JS side use `textContent`, not `innerHTML` (week 8). `HttpOnly` cookies and a `Content-Security-Policy` header limit the damage.

SQL injection

If you **build SQL by gluing strings**, the user's input can change the *meaning* of the query. We ran it:

```
name = "asha"; pw = "' OR '1'='1"
q = f"select * from u where name='{name}' and pw='{pw}'"
# select * from u where name='asha' and pw='' OR '1'='1'      → returns the row: login
bypassed!
```

Another payload, `asha' --`, turns the rest of the line into a comment. **Fix: parameterised queries**, where the database receives the SQL and the values *separately*, so values are always data, never code:

```
cur.execute("select * from u where name=? and pw=?", (name, pw))   # returned [] for
the same attack
```

The **ORM is safe by default**: `User.query.filter_by(name=n)` and `User.name == n` always bind parameters (our attack string returned no user). It becomes unsafe again if you build raw SQL with f-strings in `db.session.execute(text(f"…"))`; use `text("… :n")` with parameters.

WATCH OUT - REAL EXAM TRAP

- Escaping fixes **XSS**, parameters fix **SQL injection**, tokens fix **CSRF**. Do not mix the three.
- `|safe` on user input disables Jinja's protection.
- A GET route that deletes or pays is a CSRF gift.
- Client-side checks (JS, `required`) are never a defence against injection; validate and escape on the server.

Idea 6: Transport, secrets, validation, limits, tokens

HTTPS and TLS. HTTPS is HTTP inside **TLS**, which gives **encryption** (nobody on the way can read), **integrity** (nobody can alter) and **authentication of the server** (a certificate from a trusted authority proves the site is the real `bank.com`). It protects passwords, cookies and form data on the wire. It does *not* fix XSS or SQL injection. Use the `Secure` cookie flag with it.

Secrets and environment variables. `SECRET_KEY`, database passwords and API keys must **not** be written in the code or pushed to Git. Read them from the environment:

```
import os
app.config["SECRET_KEY"] = os.environ["SECRET_KEY"]   # fails loudly if missing
```

Keep a local `.env` (listed in `.gitignore`) and a `.env.example` without real values. Anyone who learns the `SECRET_KEY` can forge Flask sessions.

Input validation. Treat all input as hostile: check **type, length, range and format** on the **server** (and on the client for comfort). Prefer an *allow list* (only letters and digits) to a block list. A real End Term: frontend validation improves user experience, backend validation is mandatory for security because the frontend can be bypassed, and the backend should check the same constraints.

Rate limiting. Limit how many requests an address or user may make in a time window, to slow **brute force** password guessing and abuse (Flask-Limiter: `@limiter.limit("5 per minute")`). Our small hand-made version let 3 requests through and refused the 4th and 5th: `[True, True, True, False, False]`. Return **429 Too Many Requests**.

Token-based authentication (introduction). Instead of a server-side session and a cookie, the server gives a signed **token** (for example a **JWT**, or a random string). The client sends it in every request: `Authorization: Bearer <token>`. The server only verifies the signature, so it keeps **no session state** (stateless, good for scaling) and the same token works for a mobile app or JavaScript calling your REST API. Tokens need an **expiry** and **HTTPS**; if stored in `localStorage` they can be stolen by XSS. You will use token auth with the REST API in MAD2.

Principle of least privilege. Give every person and program the **smallest access** that lets it do its job: the web app's database user should not be able to drop tables; a normal user gets the `user` role, not `admin`; an API key can be read-only. If something is compromised, the damage is small.

Idea 7: A secure project checklist

AREA	DO THIS
Passwords	`generate_password_hash`, never plain text, one error message for login failure
Sessions	long random `SECRET_KEY` from the environment; `HttpOnly`, `Secure`, `SameSite`; logout clears the session
Access	`@login_required` plus a role check on every sensitive route; check ownership (a user may edit only *their* rows)
Forms	CSRF token on every POST; POST (never GET) to change data
Output	keep Jinja autoescape on; no safe filter on user text; `textContent` in JS
Database	ORM or `?` parameters, never f-string SQL; least-privilege DB user
Transport	HTTPS in production; debug mode **off** (the debugger page can run code)
Input	validate type, length and format on the server; limit upload size and type
Abuse	rate-limit login; log failures (week 7 logging) without logging passwords
Secrets	environment variables, `.env` in `.gitignore`

Your turn

YOUR TURN: 1. HASH FACTS

For each: true or false. (a) A hash can be reversed with the right key. (b) Two users with the same password get the same stored string when salted. (c) `check_password_hash` needs only the stored string and the typed password.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 2. WHICH CONTROL?

Name the access-control type: (a) only users with role `teacher` may open `/grades`; (b) a file service allows access only when `user.country == file.country` and time is working hours; (c) a user who has made 25 comments may upload images.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 3. STATUS CODES

A route has `@login_required`. What is returned to (a) an anonymous user, (b) a logged-in non-admin user on a route that checks `current_user.role == 'admin'` and returns `abort(403)`, (c) a request with a wrong CSRF token that the code aborts with 403?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 4. FIND THE VULNERABILITY

Name the attack and the fix for each. (a) `cur.execute(f"select * from users where id = {uid}")` (b) `{{ comment|safe }}` for user comments (c) a "delete my account" link that is a plain GET `/delete`.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 5. MEMORY ARITHMETIC

A store keeps 2 million sessions of 3 KB each in memory (1 KB = 1000 B). How much RAM at minimum?

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

About 18 of ~692 Quiz 1, about 36 of ~624 Quiz 2 and about 50 of ~1600 End Term questions match this week's keywords (rough counts, and many of those matches are really neighbouring topics such as SQLAlchemy models, forms and Flask routes). In the sample of about 27 questions the genuinely *security* ones are fewer:

- **Flask-Login** (`login_user(user)` , `@login_required` redirecting to the login page, `user_loader` and `UserMixin` , `current_user`) and **Flask sessions with status codes** (404/401/200 sequences).
- **Access control types**: ABAC vs RBAC (twice) and the match-the-column on HTTPS, cookies, digest and basic authentication.
- **Cookie statements** (recognise users, can hold login data).
- **Frontend (HTML5/JS) vs backend validation**; the form `method` default (GET).
- **Login logic with credentials in a URL** (the for/else trap) and **session memory arithmetic** (1.5 million sessions of 4 KB).
- Many login-themed questions are really **SQLAlchemy ORM** tests (insert a user, `filter_by( … ).first()` , relationships between `User` and `Role`), taught in earlier weeks.

I did not find questions on CSRF, rate limiting or JWT in this sample, so learn them for the viva and as concept; they are more likely in MAD2 and in project discussions.

WATCH OUT - REAL EXAM TRAP

- `login_user(user)` , not `user.id` , not `user.name` .
- 404 vs 401 vs 403 in session-based routes: read which `if` the code reaches.
- A `for … else` or an early `return` inside a loop can reject valid users.
- Base64 is encoding, not encryption; a hash is not encryption; a salt is not secret.
- Frontend checks never replace backend checks.

Quick quiz

QUICK QUIZ

Which `flask_login` call registers the user as logged in for the session?

`login_user(user.id)`

`login_user(user)`

`login_user(user.name)`

`login_user(user.name and user.password)`

Check

An anonymous visitor opens a route decorated with `@login_required`. What happens by default?

- 404
- the page shows "Welcome None"
- a redirect to the login page
- 403 error page

Check

A forum unlocks image posting once a user has more than 25 posts. This is...

- role-based access control
- attribute-based access control
- both
- neither

Check

Select all TRUE statements about passwords. (select all that apply)

- Store a salted hash, not the password.
- The salt makes equal passwords produce different hashes.
- Base64 encoding is a safe way to store passwords.
- Hashing is reversible if you know the key.

Check

Match: HTTPS, Cookies, Digest authentication, HTTP Basic authentication.

- HTTPS: base64; Cookies: hash; Digest: encrypted channel; Basic: sent each time

HTTPS: encrypted channel; Cookies: sent with each request; Digest: hash function; Basic: base64 encoding

HTTPS: hash function; Cookies: encrypted channel; Digest: base64; Basic: sent each time

HTTPS: sent each time; Cookies: base64; Digest: encrypted channel; Basic: hash

Check

Which defence stops SQL injection?

CSRF tokens

HttpOnly cookies

parameterised queries or the ORM

Jinja autoescape

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- How do you store passwords in your project and why?
- Explain authentication and authorisation in your app.
- How does your login stay active between requests?
- What is CSRF and how do you stop it?
- What is XSS and how does Flask protect you?
- How does SQL injection work and how did you avoid it?
- Why use HTTPS and environment variables?
- What is the principle of least privilege?

Recap

REMEMBER THIS

- **Authentication** = who; **authorisation** = what. 401 means log in, 403 means not allowed. RBAC by role, ABAC by attributes.
- **Passwords:** store a salted slow hash (`generate_password_hash` / `check_password_hash`); never plain text; hash is not encryption; salt defeats rainbow tables.
- **Sessions and cookies:** cookie carries the session; flags `HttpOnly` , `Secure` , `SameSite` ; Flask session is signed, not encrypted; keep `SECRET_KEY` secret.
- **Flask-Login:** `login_user(user)` , `logout_user()` , `@login_required` (redirects), `user_loader` , `current_user` ; roles with `@roles_required` .
- **Attacks and cures:** CSRF to token and SameSite; XSS to escaping (autoescape, no `|safe` , `textContent`); SQL injection to parameters or the ORM.
- **Hardening:** HTTPS/TLS, secrets in environment variables, server-side validation, rate limiting, least privilege, debug off in production.
- **Tokens** (`Authorization: Bearer ...`) give stateless authentication for APIs; used more in MAD2.

CODE AND FACTS TO KEEP

- `generate_password_hash(pw)` and `check_password_hash(stored, pw)` from `werkzeug.security`
- `login_user(user)`, `logout_user()`, `@login_required`, `current_user`, `@login_manager.user_loader`
- `@roles_required('admin')` (all), `@roles_accepted('a','b')` (any)
- `session['k'] = v`, `session.pop('k', None)`, `app.config['SECRET_KEY'] = os.environ['SECRET_KEY']`
- `SESSION_COOKIE_HTTPONLY = True`, `SESSION_COOKIE_SECURE = True`, `SESSION_COOKIE_SAMESITE = "Lax"`
- Cookie header: `Set-Cookie: session=...; HttpOnly; Secure; SameSite=Lax`
- Parameterised SQL: `cur.execute("select * from u where name=?", (n,))`
- Jinja: `{{ x }}` escapes; `{{ x|safe }}` does not
- Status codes: 401 unauthenticated, 403 forbidden, 404 not found, 429 too many requests, 302 redirect
- Basic auth = base64 (not encryption); Digest = hash; HTTPS = encrypted channel

MAD1 week 10: Testing web applications with pytest

> **Official syllabus line:** Testing of Web Applications

The one-minute story

THINK OF IT LIKE THIS

Testing is the quality check at a biscuit factory. Before a packet leaves the factory, somebody checks it. One person checks that a single biscuit is not broken (that is a **unit test**). Another checks that biscuits, wrapper and box fit together (an **integration test**). A third opens a sealed box exactly as a customer would and tastes it (an **end-to-end test**). If you checked by hand every time you changed the recipe, you would get tired and skip some checks. So you write the checks as **small programs** that run in seconds, and you run them after every change.

pytest is the machine that runs all your check-programs and prints "passed" or "failed". The **Flask test client** is a pretend browser that lives inside your Python program, so you can click links and submit forms without opening a real browser or starting a real server.

By the end of this week you can:

- explain why we test and name the kinds of tests (unit, integration, system, regression, acceptance);
- write pytest tests with `assert`, fixtures, `parametrize` and `conftest.py`;

- test Flask pages and JSON APIs with `app.test_client()` against a separate in-memory SQLite database;
- predict the exact summary line of `pytest -k ...` and `pytest -m ...` runs;
- explain mocking, test doubles, statement vs branch coverage, and continuous testing.

Words you need

WORDS YOU NEED

- **Test** - a small program that runs part of your code and checks the result.
- **Unit test** - tests one small unit (a function or method) alone.
- **Integration test** - tests several units working together (a route plus the database).
- **System test (end-to-end)** - tests the whole running application, server included, from the outside.
- **Regression test** - a test that is re-run after every change to make sure old features did not break.
- **User acceptance test (UAT)** - real users try the software before release (alpha and beta testing).
- **White box testing** - tests designed by looking at the code inside. **Black box**: only inputs and outputs are looked at.
- **TDD** - test-driven development: write the failing test first, then the code.
- **Fixture** - a function that prepares something a test needs (an app, a client, test data).
- **Mock** - a fake object that replaces a real one (a web service, a clock) so the test is fast and predictable.
- **Test double** - the general name for any stand-in: dummy, stub, fake, mock, spy.
- **Coverage** - the percentage of your code that the tests actually executed.
- **Marker** - a label such as `@pytest.mark.slow` that lets you pick which tests to run.

Idea 1: Why test, and the kinds of tests

Without tests you find bugs when a user complains. With tests you find them in seconds, on your own laptop. Tests also let you **change code without fear**: if all tests still pass, you probably broke nothing.

KIND	WHAT IT CHECKS	SPEED	EXAMPLE
Unit	one function or method, alone	very fast	`square(5) == 25`
Integration	several parts together	fast	route + database model
System / end-to-end	whole app in a full environment, including the server	slow	open the real site and log in
System test automation	a script that drives a real browser like a user (Selenium)	slow	script clicks the Login button

KIND	WHAT IT CHECKS	SPEED	EXAMPLE
Regression	re-run old tests after each change	depends	"adding search did not break login"
User acceptance (beta)	real users verify before release	slowest	beta testers use the app

Testing pyramid. Write **many** unit tests (cheap, fast), **fewer** integration tests, and **very few** end-to-end tests (slow, fragile). The picture is a pyramid with unit tests as the wide base.

White box vs black box. If you design tests by reading the code (every `if` branch must run), it is white box testing. If you only know "this input should give that output", it is black box testing.

Test-driven development (TDD) is a habit with three steps, repeated in short loops:

1. **Red**: write a test for a feature that does not exist yet. Run it. It fails.
2. **Green**: write the smallest code that makes the test pass.
3. **Refactor**: clean the code while keeping all tests green.

Continuous testing. Tests are run automatically on every push (see **CI** in week 12). **Continuous integration** means changes from many people are merged into one code base often and tested each time. **Continuous delivery** means that after the tests pass, a release package is prepared automatically. **Continuous deployment** goes one step further: every passing change is put into production automatically.

> **THE IDEA**
>
> **Arrange - Act - Assert** is the shape of almost every test.
>
> 1. **Arrange**: set up the data (a client, a database with one user).
> 2. **Act**: do the one thing under test (call the function, send the request).
> 3. **Assert**: check the result (`assert status_code == 200`).
>
> Keep one idea per test. A test name should say what is checked: `test_add_requires_title` .

> **WATCH OUT - REAL EXAM TRAP**
>
> - "System testing automation" (scripting a real browser) is **not** the same as a unit test. In the match-the-column questions: unit = smallest logical units; integration = several modules together; system = whole app with server; automation = simulated user browser interaction; acceptance = real users before release; white box = internal functioning; regression = runs after every change.
> - Beta testing = user acceptance testing. Integration testing is **one step before** system testing.

Idea 2: pytest basics

Install: `pip install pytest`. pytest is the tool; the tests are plain functions with plain `assert`.

```
# file: test_cases.py
def square(n):
    return n * n

def test1():
    assert square(5) == 25

def test2():
    assert square(square(2)) == 16

def test3():
    assert square(square(2)) == 8
```

Run `pytest -v test_cases.py`. Real output (trimmed):

```
test_cases.py::test1 PASSED                                              [ 33%]
test_cases.py::test2 PASSED                                              [ 66%]
test_cases.py::test3 FAILED                                              [100%]
FAILED test_cases.py::test3 - assert 16 == 8
========================= 1 failed, 2 passed in 0.09s =========================
```

`square(square(2))` is `square(4)` which is 16, so test3 fails and pytest shows both sides of the broken comparison: `assert 16 == 8`.

THE IDEA

Test discovery rules (pytest finds tests by name; nothing else is run):

- files named `test_*.py` or `*_test.py`;
- inside them, functions named `test_*`;
- classes named `Test*` (with no `__init__`), whose methods named `test_*` are tests.

A function called `case_test2` or `sample_test3` is **silently ignored**: it is not collected, not run, not counted. A test passes if it finishes without an exception; a failed `assert` raises `AssertionError` and the test is marked failed.

What the summary line means. Common words: `passed`, `failed`, `deselected` (collected but skipped because of `-k` or `-m`), `error` (a problem outside the test body, such as a missing fixture), `warnings`.

```
# test_class_demo.py
class TestGroup:
    def do_something(self, x, y):      # helper, not a test (no test_ prefix)
```

```
        return x % y

    def test_one(self):
        assert self.do_something(10, 5) == 0, "Is 5 factor of 10?"

    def test_two(self):
        assert self.do_something(32, 5) == 0, "Is 5 factor of 32?"
```

`pytest -v` prints `TestGroup::test_one PASSED` and `TestGroup::test_two FAILED`; the failure line carries your message after `AssertionError:`. Only the **failed** test's message is shown; the passing test's message never appears. Summary: `1 failed, 1 passed`.

Useful pytest pieces

```
import pytest

def test_float():
    assert 0.1 + 0.2 == pytest.approx(0.3)      # never compare floats with ==

def test_error():
    with pytest.raises(ZeroDivisionError):      # passes only if the error is raised
        1 / 0

def test_tmp(tmp_path):                         # built-in fixture: a fresh temp folder
    f = tmp_path / "notes.txt"
    f.write_text("hi")
    assert f.read_text() == "hi"
```

`tmp_path` gives every test its own empty temporary folder that is cleaned up later, so tests that write files never disturb each other or your real files. pytest also has `monkeypatch` (temporarily replace a variable or function) and `capsys` (capture `print` output).

SOLVED EXAMPLE: 1. PREDICT THE PYTEST SUMMARY

```
def dec(x):
    return x - 1

def test_answer():
    assert dec(3) == 3
```

Question. What does `pytest` report?

Solution. `dec(3)` is 2, so `assert 2 == 3` fails. Output: `FAILED test1.py::test_answer - assert 2 == 3` and `1 failed`. It is "failed", not "passed", and the count is 1, not 2.

Idea 3: Fixtures, conftest.py, parametrize, markers, -k and -m

A **fixture** prepares something and hands it to the test. A test asks for it by writing the fixture's name as a **parameter**.

```
import pytest

@pytest.fixture
def students():
    return {"names": ["ravi", "raj"], "ages": [12, 9]}

def test_name(students):
    assert "ravi" in students["names"]

def test_age(students):
    assert students["ages"][0] > 9
```

Each test gets a **fresh** copy (the default scope is per test), so one test cannot damage another's data. Fixtures that need cleanup use `yield`: the code before `yield` is the setup, the value after `yield` is given to the test, and the code after `yield` runs as teardown.

```
@pytest.fixture
def numbers():
    print("setup")
    yield [1, 2, 3]
    print("teardown")     # runs after the test finishes
```

> **THE IDEA**
>
> **conftest.py** is a special file. Fixtures written in it are available to **every test file in that folder and below**, with no import. Put shared fixtures (`app` , `client`) there. A fixture defined inside one test file is visible only to that file.

Parametrize runs one test with many inputs:

```
@pytest.mark.parametrize("a,b,expected", [(2, 3, 5), (0, 0, 0), (-1, 1, 0)])
def test_add_numbers(a, b, expected):
    assert a + b == expected
```

This is **three** tests, shown as `test_add_numbers[2-3-5]` , `test_add_numbers[0-0-0]` , `test_add_numbers[-1-1-0]` . Real run: all pass.

Markers label tests: `@pytest.mark.slow` . Select with `-m` . Register your markers in `pytest.ini` to avoid warnings:

```
[pytest]
markers =
    slow: tests that take long
    unit: unit tests
```

Selecting tests (this is what the papers ask the most):

COMMAND	SELECTS
`pytest -k email`	tests whose **name** (function, class or file, any part, case-insensitive) contains `email`
`pytest -m slow`	tests carrying the marker `slow`
`pytest -m "not slow"`	everything without the marker
`pytest -m "slow and integration"`	tests carrying both markers
`pytest test_a.py::test_x`	one specific test
`pytest -v`	one line per test, with PASSED / FAILED
`pytest -x`	stop after the first failure

SOLVED EXAMPLE: 2. -M AND -K BY PREDICTION

```
import pytest

def square(x):
    sum = 0
    for counter in range(x):
        sum += x
    return sum        # this really computes x times x

@pytest.mark.marker1
def testcase_1(): assert square(10) == 100
@pytest.mark.marker2
def testcase_2(): assert square(4) == 4
@pytest.mark.marker3
def testcase_3(): assert square(5) == 25
```

Question. What is the summary of `pytest -m marker2` ? Of `pytest -m marker1` ?

Solution. `square(x)` adds `x` a total of `x` times, so `square(4)` is 16, and `assert 16 == 4` fails. With `-m marker2` only that one test runs: **1 failed, 2 deselected**. With `-m marker1` : `square(10)` is 100, so **1 passed, 2 deselected** (real run confirmed). The unselected tests are never run, so their outcome does not matter, but they are counted as deselected.

SOLVED EXAMPLE: 3. -K WITH CLASSES AND A FUNCTION THAT IS NOT COLLECTED

```
def func(x, y):
    return x**2 + y**2

class Test_class0:
    def test_case1(self): assert func(1, 2) == 5      # 1+4 = 5, pass
    def case_test2(self): assert func(2, 3) == 13     # not collected
    def test_case3(self): assert func(6, 2) == 38     # 36+4 = 40, fail

class Test_class1:
    def test_case1(self): assert func(5, 2) == 29     # 25+4 = 29, pass
    def case_test2(self): assert func(1, 1) == 2      # not collected
```

Question. Output of `pytest test_file.py -k Test_class` ? Of `-k Test_class0` ?

Solution. Only three methods are collected (`test_case1` , `test_case3` , and `Class1.test_case1`); the two `case_test2` methods do not start with `test` . `-k Test_class` matches both class names, so nothing is deselected: **1 failed, 2 passed**. With `-k Test_class0` only the first class is picked: **1 failed, 1 passed, 1 deselected**. Real run confirmed both.

WATCH OUT - REAL EXAM TRAP

- `-k` is a **substring match on names**; `-m` matches **markers**. Mixing them up is the most common error. (A detail: `-k` also matches marker names, so `-k marker` in the example above selects all three tests.)
- Tests that do not start with `test` are not "deselected" and not "failed": they simply do not exist for pytest. Do not count them.
- Count `deselected` only for tests that were collected and then filtered out.
- A fixture that lives in `conftest.py` is shared by several files. If one test file uses a fixture that is not defined anywhere, that test shows `error` , not `failed` .

Idea 4: Testing Flask with the test client

`app.test_client()` returns an object that behaves like a browser but calls your app **directly in memory**. No server, no port, very fast.

Here is the small app we test (an application factory, so a test can build its own app):

```
# app.py
from flask import Flask, jsonify, request, redirect, url_for
from flask_sqlalchemy import SQLAlchemy
```

```
db = SQLAlchemy()

class Todo(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(80), nullable=False)

def create_app(config=None):
    app = Flask(__name__)
    app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///todo.db"
    if config:
        app.config.update(config)           # a test can override the database
    db.init_app(app)
    with app.app_context():
        db.create_all()

    @app.route("/")
    def home():
        return "Todo app"

    @app.route("/add", methods=["POST"])
    def add():
        title = request.form.get("title", "").strip()
        if not title:
            return "Title required", 400
        db.session.add(Todo(title=title))
        db.session.commit()
        return redirect(url_for("home"))

    @app.route("/api/todos", methods=["GET", "POST"])
    def todos():
        if request.method == "POST":
            data = request.get_json()
            if not data or "title" not in data:
                return jsonify(error="title missing"), 400
            t = Todo(title=data["title"])
            db.session.add(t)
            db.session.commit()
            return jsonify(id=t.id, title=t.title), 201
        return jsonify([{"id": t.id, "title": t.title} for t in Todo.query.all()])

    return app
```

The key trick: a separate in-memory database for tests. If tests used `todo.db`, they would fill your real data with junk. The URI `sqlite:///:memory:` creates a database that lives only in RAM and vanishes

when the app object goes away. Every test gets a fresh, empty database. (Flask-SQLAlchemy 3 keeps one shared connection for in-memory SQLite, so the tables you create stay visible.)

```
# conftest.py
import pytest
from app import create_app, db

@pytest.fixture
def app():
    app = create_app({"TESTING": True, "SQLALCHEMY_DATABASE_URI": "sqlite:///:memory:"})
    yield app
    with app.app_context():
        db.drop_all()

@pytest.fixture
def client(app):
    return app.test_client()
```

`TESTING=True` makes Flask let exceptions propagate to the test instead of hiding them behind a 500 page. A fixture can use another fixture (`client` asks for `app`).

```
# test_app.py
def test_home(client):
    resp = client.get("/")
    assert resp.status_code == 200
    assert b"Todo" in resp.data           # resp.data is BYTES

def test_add_requires_title(client):
    resp = client.post("/add", data={"title": "  "})
    assert resp.status_code == 400

def test_add_redirects(client):
    resp = client.post("/add", data={"title": "milk"})
    assert resp.status_code == 302
    assert resp.headers["Location"] == "/"

def test_add_follow(client):
    resp = client.post("/add", data={"title": "milk"}, follow_redirects=True)
    assert resp.status_code == 200
    assert resp.data == b"Todo app"
```

All of these passed in a real run. Notes:

- `client.get(url)` and `client.post(url, data={ … })` return a **response**. `data={ … }` sends a **form** (like an HTML form post).

- `resp.status_code` is the HTTP status (200, 302, 400, 404, 405...). `resp.data` is the **body as bytes** (`b"..."`); `resp.get_data(as_text=True)` gives text. `resp.headers` holds headers.
- A redirect is **not followed by default**: you see the 302 and the `Location` header. With `follow_redirects=True` the client follows it and you see the final page.
- Posting to a route that allows only GET gives **405**; an unknown URL gives **404**.

Testing JSON API endpoints

```
def test_api_create_and_list(client):
    r = client.post("/api/todos", json={"title": "milk"})   # json= sends a JSON body
    assert r.status_code == 201
    assert r.get_json() == {"id": 1, "title": "milk"}
    r = client.get("/api/todos")
    assert r.get_json() == [{"id": 1, "title": "milk"}]

def test_api_bad(client):
    r = client.post("/api/todos", json={})
    assert r.status_code == 400
    assert r.get_json()["error"] == "title missing"

def test_isolated(client):
    assert client.get("/api/todos").get_json() == []         # fresh database again
```

`json=` sets the `Content-Type: application/json` header and encodes the body; `r.get_json()` decodes the reply. The id is 1 in the first test, and `test_isolated` sees an empty list, proving each test has its own database. Whole file: `10 passed`.

WATCH OUT - REAL EXAM TRAP

- **Test client vs the `requests` library.** `requests.get("http://127.0.0.1:5000/...")` talks to a **real running server**. If the server is not started in another terminal, the test fails with a connection error. The `test_client()` needs no server. A paper question runs `app.py` and the test in two terminals: only then does `requests` pass.
- `resp.data` is bytes: `resp.data == "Hello"` is False; use `b"Hello"`.
- Without `follow_redirects=True` the status after a redirect is 302, not 200.
- Forgetting the in-memory database means tests write into the real `.db` file.

Idea 5: Mocking and test doubles

A good unit test must be **fast, repeatable and independent of the outside world**. Calling a real weather service, sending a real e-mail or reading the real clock breaks all three. So we replace those parts with fakes. All the fakes are called **test doubles**:

DOUBLE	WHAT IT DOES
Dummy	fills a parameter, never really used
Stub	returns fixed answers
Fake	a simple working version (the in-memory SQLite database is a fake database)
Mock	records how it was called, so you can assert "it was called once"
Spy	wraps the real thing and records the calls

```
# weather.py
import requests

def temperature(city):
    r = requests.get("https://api.example.com/temp", params={"city": city}, timeout=5)
    r.raise_for_status()
    return r.json()["celsius"]

def describe(city):
    t = temperature(city)
    return "hot" if t ≥ 30 else "ok"
```

```
# test_weather.py
from unittest.mock import patch, MagicMock
import weather

@patch("weather.requests.get")                    # replace requests.get inside
weather.py
def test_describe_hot(mock_get):
    mock_get.return_value = MagicMock(status_code=200)
    mock_get.return_value.json.return_value = {"celsius": 35}
    assert weather.describe("Chennai") == "hot"
    mock_get.assert_called_once()

def test_describe_ok(monkeypatch):
    monkeypatch.setattr(weather, "temperature", lambda city: 20)
    assert weather.describe("Ooty") == "ok"
```

Both pass in a real run, with no internet used. The golden rule: **patch where the name is used** (`weather.requests.get`), not where it was defined. Mocks are also the way to force rare cases (a timeout, a 500 from a service).

SOLVED EXAMPLE: 4. WHICH TESTS GO WHERE

Question. Classify: (a) `assert is_prime(7)` ; (b) POST a form to `/add` and check the row in the in-memory database; (c) a Selenium script logs in on the running site and clicks "Logout"; (d) friends try your beta; (e) re-run everything after adding search.

Solution. (a) unit; (b) integration (route + database); (c) system test automation (UI test); (d) user acceptance (beta); (e) regression.

Idea 6: Code coverage, UI tests, and the habit of running tests

Code coverage answers "how much of my code did the tests execute?". With the `pytest-cov` plugin:

```
pip install pytest-cov
pytest --cov=app --cov-branch --cov-report=term-missing
```

Real output on our app with the 10 tests above:

```
Name     Stmts   Miss Branch BrPart  Cover   Missing
-----------------------------------------------------
app.py      38      0      8      1    98%   14->16
-----------------------------------------------------
TOTAL       38      0      8      1    98%
```

All 38 statements ran (Miss 0), but one of 8 branches was only taken one way: line 14 is `if config:`, and our tests always pass a config, so the "no config" path was never taken (`14->16` means "from line 14 jump to line 16" was never seen). That is why statement coverage is 100 percent yet total coverage shows 98 percent once branches count.

THE IDEA

Coverage kinds, using `if x > 0 and y > 0: z = x` :

- **Function coverage**: every function is called at least once.
- **Statement (line) coverage**: every statement runs. `foo(1,1)` is enough.
- **Branch coverage**: every `if` goes both **true and false**. `foo(1,1)` and `foo(1,0)` give it.
- **Condition coverage**: every **individual condition** (`x > 0` , `y > 0`) takes both values. `foo(0,1)` and `foo(1,0)` give each condition a False and a True.

100 percent statement coverage does **not** imply 100 percent branch coverage (an `if` with no `else` can run its body but never test the false side), and it does not imply condition coverage. High coverage does **not** mean no bugs; it only means the lines were run, not that the asserts were good.

UI testing with Selenium (concept). Selenium lets a script start a real browser, open your site, fill the form, click buttons and read the page. It tests what a user sees (HTML, CSS, JavaScript together). It is **slow and fragile** (a changed button label breaks it), so you write few of them. In the syllabus it is "system testing automation".

```
# concept only; needs a browser driver installed
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Chrome()
driver.get("http://127.0.0.1:5000/login")
driver.find_element(By.NAME, "email").send_keys("a@b.com")
driver.find_element(By.ID, "login-btn").click()
assert "Dashboard" in driver.title
driver.quit()
```

IN YOUR PROJECT

For your project viva. Add a `tests/` folder with a `conftest.py` (app + client fixtures on an in-memory database) and at least five tests: one page loads (200), a bad form gives 400, login works, an API returns JSON, and a database row is created. Run `pytest --cov` and be ready to show it live. The examiner will ask: "How do you know your app works after a change?" and "Why do your tests not touch your real database?"

Your turn

YOUR TURN: 1. TEST DISCOVERY

A file `test_x.py` has `def test_a()`, `def check_b()`, `def testc()`, and `class Helper` with a method `test_d`. Which tests does pytest run?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 2. PREDICT -K

```
def test_user_add(): assert True
def test_user_del(): assert False
def test_item_add(): assert True
```

Output summary for `pytest -k user`? For `pytest -k add`?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 3. CONFTEST AND FIXTURES

`conftest.py` has a fixture `divisible_by_39` returning 39. File A has `test_div3` (`x % 3 == 0`) and `test_div6` (`x % 6 == 0`) using it. File B has `test_div13` (`x % 13 == 0`) using it. Summary of `pytest`?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 4. WRITE A CLIENT TEST

Write a test that checks `GET /nothing` returns 404 for the todo app, using the `client` fixture.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 5. LATE BINDING (A TRICKY PAPER QUESTION)

```
powers = []
for i in range(1, 5):
    def powers_of_x(x):
        return x ** i
    powers.append(powers_of_x)
powers_of_3 = [f(3) for f in powers]
```

Tests assert `9 in powers_of_3`, `27 in powers_of_3`, `64 in powers_of_3`. Summary?

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

The keyword match for this week finds about **45 distinct graded questions**, almost all in **End Term** (about 177 keyword hits in End Term papers against about 3 in Quiz 2 and none in Quiz 1, so this is purely an End Term topic and the count is coarse because the keyword also catches unrelated questions). The real recurring questions are:

- **Predict the pytest summary** for a small test file: `1 failed, 2 passed`, `3 failed`, with `-v` lines like `test3 FAILED`. About a dozen real questions look like this.
- **`-k` and `-m` selection** (about 7 questions): which command gives `1 passed, 2 deselected`? Output of `pytest -k Test_class`. Fixture plus `-k email` gives `1 passed, 1 deselected`. Marker questions with `and` and `not`.
- **Fixtures and conftest.py** shared by several files (one question asks for the summary `1 failed, 2 passed`).
- **Match the type of testing** with its description (unit, integration, system, system testing automation, user acceptance, regression, white box). About 5 real questions, the same table repeated.
- **Coverage match**: function / statement / branch / condition coverage for a small C function, and the true/false statement that 100 percent statement coverage does not imply branch coverage.
- **Continuous integration and continuous delivery** statements (both correct).
- **Test client vs requests**: a live-server test passes only if the app runs in another terminal.

WATCH OUT - REAL EXAM TRAP

- Names matter: `case_test2` and `sample_test3` are not collected. Do not count them in the summary.
- `-k` for names, `-m` for markers; `deselected` appears only when a filter is used.
- With `-m "not slow"`, count carefully which tests lack the marker.
- An assert message (the text after the comma) is shown only for the tests that **fail**. A passing test prints nothing.
- Closures in loops (late binding) make every function use the last `i`.
- Statement coverage 100 percent is **not** branch coverage 100 percent.

Quick quiz

QUICK QUIZ

A file contains `def test_a(): assert 1 == 1`, `def check_b(): assert 1 == 2` and `def test_c(): assert 2 == 2`. What is the summary of running `pytest`?

- 1 failed, 2 passed
- 3 passed
- 2 passed
- 2 passed, 1 deselected

Check

A fixture that should be available to every test file in a folder is best placed in...

- every test file, copied
- `__init__.py`
- `conftest.py`
- `requirements.txt`

Check

The test `client.post("/add", data={"title": "milk"})` returns a redirect. Without `follow_redirects=True`, `resp.status_code` is...

200

302

404

201

Check

Match each test type with its focus: unit, integration, system test automation, user acceptance.

unit: server and environment; integration: one function; automation: real users; acceptance: browser script

unit: smallest function; integration: several modules together; automation: simulated browser interaction; acceptance: real users before release

unit: real users; integration: browser script; automation: single function; acceptance: several modules

all four are the same thing

Check

Select all TRUE statements about Flask tests with pytest. (select all that apply)

Test functions must start with `test_` to be collected.

Fixtures help give each test an isolated setup.

Routes cannot be tested with `client.get()`.

You must start the Flask server before using `app.test_client()`.

Check

Which is TRUE?

100 percent statement coverage always means 100 percent branch coverage

100 percent statement coverage does not imply 100 percent branch coverage

Code coverage proves the program has no bugs

Branch coverage is weaker than statement coverage

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- Why did you write tests for your project, and what kinds?
- How do your tests avoid damaging your real data?
- What is a fixture, and what is conftest.py?
- What is the difference between the Flask test client and Selenium?
- What is mocking and when did you use it?
- Explain statement coverage and branch coverage.
- What is TDD?

Recap

REMEMBER THIS

- Test types: **unit** (one function), **integration** (parts together), **system / end-to-end** (whole app), **system test automation** (script a browser, Selenium), **regression** (after every change), **acceptance / beta** (real users). Pyramid: many unit, few end-to-end.
- pytest collects `test_*.py` files, `test_*` functions, `Test*` classes. Names that do not start with `test` are ignored.
- **Fixtures** prepare data; `conftest.py` shares them; `yield` gives teardown; `parametrize` repeats a test with many inputs.
- `-k` selects by name substring, `-m` by marker; filtered-out tests are `deselected`.
- Flask: `app.test_client()`, `client.get/post`, `status_code`, `data` (bytes), `get_json()`, `follow_redirects=True`; use a separate in-memory SQLite database.
- **Mocks** and other **test doubles** replace slow or unpredictable parts. Patch where the name is used.
- **Coverage**: function, statement, branch, condition; 100 percent statement is not 100 percent branch.
- Arrange - Act - Assert; TDD is red, green, refactor; CI runs tests on every push.

CODE AND FACTS TO KEEP

- `pip install pytest pytest-cov`; run: `pytest`, `pytest -v`, `pytest -x`, `pytest file.py::test_name`
- Selection: `-k name_part`, `-m marker`, `-m "not slow"`, `-m "a and b"`
- Fixture: `@pytest.fixture` (use `yield` for teardown); parametrize: `@pytest.mark.parametrize("a,b", [(1,2),(3,4)])`
- Checks: `assert x == y`, `pytest.approx(0.3)`, `with pytest.raises(Err):`, `tmp_path`
- Flask: `app.test_client()`; `client.get("/")`; `client.post("/x", data={...})`; `client.post("/api", json={...})`
- Response: `.status_code`, `.data` (bytes), `.get_json()`, `.headers["Location"]`
- Test database: `{"TESTING": True, "SQLALCHEMY_DATABASE_URI": "sqlite:///:memory:"}`
- Mock: `from unittest.mock import patch, MagicMock`; `@patch("module.name")`
- Coverage: `pytest --cov=app --cov-branch --cov-report=term-missing`
- Summary words: passed, failed, deselected, error, warnings

MAD1 week 11: HTML5, accessibility, encoding and beyond HTML

> **Official syllabus line:** HTML Evolution and Beyond HTML

The one-minute story

THINK OF IT LIKE THIS

HTML is a building, and the web has been renovating it for 30 years. The first version was a plain hall with signboards. Then people added tiny shops (forms), then decorations (CSS and scripts), and the rules got messy because every browser built differently. HTML5 is the big, tidy renovation: **named rooms** (header, nav, main, footer) so that visitors and machines know where things are; **built-in appliances** (audio, video, canvas) so you do not need outside plug-ins; and **cupboards inside the visitor's own browser** (web storage).

Two more things make a building good for everyone: **ramps and lifts** (accessibility, so people who cannot see, hear or use a mouse can still enter), and a **common language on the signboards** (character encoding, so "₹" on your page does not turn into garbage on someone else's screen).

By the end of this week you can:

- say how HTML 4, XHTML and HTML5 differ and use the semantic elements;
- name the four accessibility principles and the guideline under each, and spot which principle an example follows;
- **calculate** the size of a text in bits for a fixed-length encoding, and explain ASCII, Unicode, UTF-8 and UTF-16;
- compare cookies, localStorage, sessionStorage and IndexedDB;
- explain, at concept level, WebSockets, service workers and PWAs, web components, and the formats JSON, XML and Markdown.

Words you need

WORDS YOU NEED

- **SGML** - an old, very general markup language; HTML 4 was defined with it, XML is a simplified form of it.
- **XHTML** - HTML rewritten with the strict rules of XML (every tag closed, lower case).
- **Semantic element** - a tag that says what the content **means** (`<nav>` , `<article>`), not only how it looks.
- **Accessibility (a11y)** - making a site usable by people with disabilities.
- **WCAG** - Web Content Accessibility Guidelines, the standard with four principles (POUR).
- **ARIA** - attributes like `role` and `aria-label` that describe elements to screen readers.
- **Character set / encoding** - a table of characters and the rule for turning them into bytes.
- **Code point** - the number of a character in Unicode, written `U+20B9` .
- **Mojibake** - garbled text caused by reading bytes with the wrong encoding.
- **Web storage** - key-value storage in the browser (`localStorage` , `sessionStorage`).
- **Service worker** - a script that runs in the background of the browser, between your page and the network.
- **PWA** - progressive web app: a website that behaves like an installable app.
- **Viewport** - the visible area of the page on the user's screen.

Idea 1: A short history and what HTML5 added

VERSION	YEAR (APPROX.)	WHAT TO REMEMBER
HTML 2.0, 3.2	1995 to 1997	early, many browser-specific tags
HTML 4.01	1999	defined using **SGML**; separates structure (HTML) and style (CSS)
XHTML 1.0	2000	HTML 4 rewritten as **XML**; strict: tags must be closed and nested, lower case
HTML5	finished 2014	**not** SGML-based; has its **own parsing rules**; forgiving; adds multimedia, canvas, semantic tags, new inputs; **backwards compatible**

So XHTML is based on XML, which is itself based on SGML. HTML5 is a living standard kept by WHATWG and W3C. Two facts that exams love: "HTML gets separated from SGML in HTML4" is **false** (that happened with HTML5), and "HTML5 is not backwards compatible" is **false** (old pages still work).

Other facts: HTML tag names and attribute names are **case-insensitive** (`<DIV>` and `<div>` are the same). The HTML5 doctype is just `<!DOCTYPE html>`. For large sites, **external CSS** is preferred over internal or inline CSS (one cached file, shared by all pages).

> **THE IDEA**
>
> HTML5 added four big families of things:
>
> 1. **Semantic structure**: `<header>` `<nav>` `<main>` `<section>` `<article>` `<aside>` `<footer>`.
> 2. **Richer forms**: new input types and attributes with built-in checking.
> 3. **Multimedia and graphics**: `<audio>`, `<video>`, `<canvas>`, inline `<svg>`.
> 4. **APIs**: web storage, WebSockets, geolocation, service workers, drag and drop.

Idea 2: Semantic elements, new inputs, media

A page made of only `<div>` tags is called "div soup": a screen reader cannot tell the menu from the footer. Semantic tags fix this.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Recipe blog</title>
</head>
```

```
<body>
  <header><h1>Recipe blog</h1></header>
  <nav>
    <a href="/">Home</a>
    <a href="/about">About</a>
  </nav>
  <main>
    <article>
      <h2>Masala dosa</h2>
      <section>
        <h3>Ingredients</h3>
        <ul><li>rice</li><li>urad dal</li></ul>   <!— order does not matter —>
      </section>
      <section>
        <h3>Steps</h3>
        <ol><li>Soak</li><li>Grind</li><li>Cook</li></ol>   <!— order matters —>
      </section>
    </article>
    <aside>Related: sambar recipe</aside>
  </main>
  <footer>Copyright 2026</footer>
</body>
</html>
```

TAG	MEANING
`<header>`	intro or top of the page / of a section
`<nav>`	main navigation links
`<main>`	the main content; **only one** per page
`<section>`	a themed group of content, usually with a heading
`<article>`	self-contained item that makes sense on its own (a post, a news story)
`<aside>`	side content, related but not central
`<footer>`	bottom of the page / section

Use `<ul>` for an **unordered** list (ingredients) and `<ol>` for an **ordered** list (steps); a paper question asks exactly this. Semantic tags have **no visual effect** by default; they help screen readers, search engines and you.

New input types and form attributes give free validation and the right mobile keyboard:

```
<form>
  <label for="em">Email</label>
  <input type="email" id="em" name="em" required placeholder="you@example.com">

  <label for="dob">Date of birth</label>
  <input type="date" id="dob" name="dob">

  <label for="age">Age</label>
  <input type="number" id="age" name="age" min="18" max="60">

  <input type="range" name="vol" min="0" max="10">
  <input type="color" name="c">
  <input type="url" name="site">
  <input type="tel" name="phone" pattern="[0-9]{10}">
  <input type="search" name="q" autofocus>
  <button>Send</button>
</form>
```

Other new types: `time`, `datetime-local`, `month`, `week`. Attributes: `required`, `placeholder`, `pattern` (a regular expression), `min` / `max` / `step`, `autofocus`, `autocomplete`, `disabled`, `readonly`, and `<datalist>` for suggestions. Remember: this browser-side checking is only for convenience. **The server must still validate** (the user can bypass the browser).

Multimedia and graphics

```
<video controls width="320" poster="cover.jpg">
  <source src="clip.mp4" type="video/mp4">
  <source src="clip.webm" type="video/webm">
  <track kind="captions" src="clip.vtt" srclang="en" label="English">
  Your browser does not support video.
</video>

<audio controls src="song.mp3"></audio>

<canvas id="c" width="200" height="100"></canvas>
<script>
  const ctx = document.getElementById("c").getContext("2d");
  ctx.fillStyle = "teal";
  ctx.fillRect(10, 10, 80, 50);      // x, y, width, height: a pixel drawing
</script>

<svg width="100" height="100"><circle cx="50" cy="50" r="40" fill="orange"/></svg>
```

- `<audio>` and `<video>` play without Flash or other plug-ins; `controls` shows the player buttons; several `<source>` lines let the browser pick a format it knows; `<track>` adds captions or subtitles.
- `<canvas>` is a pixel grid you paint with JavaScript (games, charts). Pixels do not remember what shapes they came from.
- **SVG** is vector graphics written as XML; each shape stays an element in the page that can be styled, scripted and scaled without blur.

WATCH OUT - REAL EXAM TRAP

- `<main>` appears once; `<article>` is **not** the same as `<section>` (an article could be read alone, a section is a part of something).
- Canvas = pixels (JavaScript); SVG = shapes (markup). Do not swap them.
- `required` and `type="email"` stop most bad input in the browser, but never replace server-side checks.

Idea 3: Accessibility

Many people live with some disability: blind, low vision, colour blind, deaf, with motor problems, or with dyslexia. Accessibility also helps everyone (captions in a noisy bus, keyboard use when the touchpad is broken). The standard is **WCAG**, built on **four principles**, remembered as **POUR**:

PRINCIPLE	THE QUESTION	GUIDELINES UNDER IT	EXAMPLES
Perceivable	Can users **sense** the content?	text alternatives for non-text content; captions and alternatives for audio and video; content that can be shown in different ways; **easier to see and hear** (colour contrast, resizable text)	`alt` text on images, captions, good contrast
Operable	Can users **use** the controls?	everything works from the **keyboard**; **enough time** to read and use content; **no seizure-causing** flashing; navigable (skip links, clear titles, focus order); **input modalities** beyond the keyboard (touch, voice)	tab order, visible focus, no content flashing more than 3 times a second
Understandable	Can users **understand** the content and the interface?	readable text (set the page language); **predictable** behaviour; **input assistance**: labels, instructions, error messages that help people avoid and correct mistakes	`<label>` plus a hint text, clear error text
Robust	Does it work with different browsers and assistive technology?	valid, compatible markup; name, role and value exposed to tools	semantic HTML, ARIA used correctly

```
<!— Perceivable: text alternative for an image —>
<img src="chart.png" alt="Bar chart: sales rose from 10 to 40 in 2025">

<!— Understandable: label and instruction attached to the input —>
<label for="word">Search for a word:</label>
<input type="text" id="word" aria-describedby="hint">
<p id="hint">Enter a word to find relevant results.</p>
<button>Start Search</button>

<!— Operable: a real button works with Tab, Enter and Space. A div does not. —>
<button onclick="save()">Save</button>
<div onclick="save()">Save</div>      <!— bad: not focusable, no role —>

<!— Robust: ARIA when no native tag fits —>
<div role="alert" aria-live="polite">Saved successfully</div>
<button aria-label="Close dialog">X</button>
```

ARIA (Accessible Rich Internet Applications) adds `role="…"` (what it is), `aria-label="…"` (a name for screen readers), `aria-live` (announce changes), `aria-expanded`, `aria-hidden`. The first rule of ARIA: **use a native HTML element if one exists** (`<button>`, `<nav>`); use ARIA only to fill gaps. Also: `alt=""` (empty) for purely decorative images; never use colour alone to show meaning; contrast ratio at least 4.5 to 1 for normal text (WCAG AA); every form control gets a `<label>`; the page must be fully usable with the keyboard (Tab, Shift+Tab, Enter, Esc).

SOLVED EXAMPLE: 1. WHICH PRINCIPLE IS IT?

Question. Match each example to a principle: (a) an `alt` attribute on an image; (b) a form with a label and an instruction under the input; (c) the website works with keyboard only and also with voice control; (d) "Give users enough time to read and use content"; (e) valid markup that works with screen readers.

Solution. (a) **Perceivable** (text alternative). (b) **Understandable** (labels and input assistance; this is the real paper example with "Search for a word"). (c) **Operable**. (d) **Operable**, so it is the odd one out in "which is NOT a perceivable principle". (e) **Robust**.

WATCH OUT - REAL EXAM TRAP

- "Which is **not** a perceivable principle?" The answer is usually the sentence about **time** (enough time, no seizures), because that belongs to **Operable**. Text alternatives, captions and "easier to see and hear" are Perceivable.
- Seizures and physical reactions: Operable, not Perceivable.
- "Helps developers avoid and correct mistakes": Understandable (input assistance).
- "Works with different input modalities beyond the keyboard": Operable.

Idea 4: Character encoding

A computer stores only numbers. To store text, we agree on a **character set** (a table: number for each character) and an **encoding** (how the number becomes bytes).

- **ASCII**: 7 bits per character, 128 characters (English letters, digits, punctuation, control codes). Usually stored in one byte with the top bit 0. `A` is 65, `a` is 97, `0` is 48.
- **Unicode**: one table for all writing systems (over a million possible **code points**, written `U+0041` for `A`, `U+20B9` for the rupee sign, `U+1F600` for a smiley). Unicode is the table; UTF-8, UTF-16 and UTF-32 are three ways to **encode** it.
- **UTF-32 (UCS-4)**: fixed 4 bytes (32 bits) for every character. Simple, but wasteful.
- **UCS-2**: fixed 2 bytes (16 bits), only the first 65,536 code points.
- **UTF-16**: 2 bytes for the common ones, **4 bytes** (a surrogate pair) for the rest; variable length.
- **UTF-8**: **variable length, 1 to 4 bytes**. Can encode **every** Unicode character. The first 128 code points are encoded **exactly like ASCII** (1 byte), so every ASCII file is already valid UTF-8. UTF-8 is the encoding of the web.

CODE POINT RANGE	UTF-8 BYTES	EXAMPLE	REAL UTF-8 BYTES
U+0000 to U+007F	1	`A` U+0041	`41`
U+0080 to U+07FF	2	`é` U+00E9	`C3 A9`
U+0800 to U+FFFF	3	`₹` U+20B9	`E2 82 B9`
U+10000 to U+10FFFF	4	smiley U+1F600	`F0 9F 98 80`

(All checked with Python.) For the text made of h, é, l, l, o, a space, the rupee sign, a space and the smiley U+1F600, we get 9 **characters** but 15 **bytes** in UTF-8, 20 bytes in UTF-16 and 36 bytes in UTF-32. Lesson: **number of characters is not number of bytes**.

```
s = "héllo"
print(len(s))                       # 5 characters
print(len(s.encode("utf-8")))       # 6 bytes (é takes 2)
print(len(s.encode("utf-32-le")))   # 20 bytes (5 x 4)
```

Declaring the encoding. Put `<meta charset="utf-8">` as one of the first things in `<head>` (within the first 1024 bytes), and make the server send `Content-Type: text/html; charset=utf-8`. If the browser guesses wrongly, you get **mojibake**. For example, the UTF-8 bytes `C3 A9` of `é` read as Latin-1 become the two characters `Ã©`, so "café" shows as "cafÃ©". Another bug: saving the file in one encoding and declaring another. Fix: save as UTF-8 and declare UTF-8 everywhere (also the database and the response header).

The exam sums on encoding

Fixed-length "minimum encoding". If a text uses a character set with N distinct characters, each character needs at least the ceiling of log2 N bits. Total bits = (number of characters in the text) times (bits

per character).

SOLVED EXAMPLE: 2. MINIMUM ENCODING OF A SENTENCE

Question. The sentence "the five boxing wizards jump quickly" is saved with the minimum fixed-length encoding of its own character set. How many bits?

Solution. The sentence has 36 characters (letters and spaces). Distinct characters: the 26 letters of the alphabet appear (it is a pangram) plus the space: 27. So the ceiling of log2 27 is 5 bits (4 bits give only 16 codes, 5 bits give 32). Size = 36 x 5 = **180** bits. (The "sphinx of black quartz judge my vow" version has 35 characters, 27 distinct, so 175 bits; "pack my box with five dozen liquor jugs" has 39 characters, 27 distinct, so 195 bits. Checked: these are the three real options marked correct.)

SOLVED EXAMPLE: 3. UPGRADING A CHARACTER SET

Question. A document of 2000 characters uses only 8 characters. Five digits are added to the set. By how many bits does the document grow?

Solution. Before: 8 characters need 3 bits (since 8 = 2 to the 3). After: 13 characters need the ceiling of log2 13 is 4 bits. Growth per character is 1 bit, so 2000 x 1 = **2000** bits. (Same pattern: 10 characters to 17 characters means 4 to 5 bits, and 1000 characters grow by 1000 bits.)

SOLVED EXAMPLE: 4. CONVERTING BETWEEN ENCODINGS AND SIZES

Question (a). A text of only alphanumeric characters takes 35,552 bits in UCS-4 (32 bits per character). How many bits in 7-bit ASCII?

Solution. Characters = 35552 / 32 = 1111. In ASCII: 1111 x 7 = **7777** bits. (Same method: 136000 / 32 = 4250 characters, x 7 = 29750 bits; in UCS-2 320000 bits means 20000 characters, x 7 = 140000 bits.)

Question (b). A fixed-length document of a 13-character set has size 3 kilobytes (1 KB = 1000 bytes). How many characters?

Solution. 13 characters need 4 bits. 3 KB = 3000 x 8 = 24000 bits. Characters = 24000 / 4 = **6000**. (With a set of 12 distinct characters the answer is also 4 bits; 2 KB then gives 16000 / 4 = 4000.)

WATCH OUT - REAL EXAM TRAP

- Distinct characters, not the length of the text, decide the bits per character. Count the **space** as a character.
- Use the **ceiling** of log2 (27 distinct needs 5 bits, never 4.75).
- Read the **unit** carefully (bits or bytes, KB = 1000 or 1024 as the question says).
- UTF-8 is **variable**, UTF-32 is **fixed** (4 bytes). "Both UTF-8 and UTF-32 are variable length" is false.
- UTF-8 is **not limited** to 65,536 characters; it can encode all of Unicode.
- ASCII text is identical in UTF-8, but a non-ASCII character in a file declared as ASCII breaks.

Idea 5: Storing data in the browser

STORAGE	SIZE (ABOUT)	LIFETIME	SENT TO SERVER?	DATA TYPE	USE FOR
Cookie	about 4 KB each	until expiry (or tab close if no expiry)	**yes, with every request**	text	session id, login
localStorage	about 5 MB per site	stays until cleared by code or user	no	strings	theme, saved preferences
sessionStorage	about 5 MB per site	one **tab**; gone when the tab closes	no	strings	form draft in one tab
IndexedDB	large (hundreds of MB, browser decides)	stays	no	structured objects, files	offline data for PWAs

```
localStorage.setItem("theme", "dark");
console.log(localStorage.getItem("theme"));    // "dark"
localStorage.setItem("count", 5);
console.log(typeof localStorage.getItem("count"));   // "string"  (everything is text)
localStorage.setItem("user", JSON.stringify({name: "Asha"}));
const u = JSON.parse(localStorage.getItem("user"));   // back to an object
console.log(u.name);                                  // "Asha"
localStorage.removeItem("theme");
sessionStorage.setItem("draft", "hello");             // lost when the tab is closed
```

Rules: storage is **per origin** (scheme + host + port); pages on other sites cannot read it. The API is **synchronous** and stores only **strings** (so use `JSON.stringify` and `JSON.parse`). **Never store passwords or secrets** there: any script on your page can read it (an XSS attack). IndexedDB is asynchronous and holds objects, indexes and files.

Idea 6: Beyond request and response

WebSockets. Normal HTTP: the browser asks, the server answers, the connection ends. A **WebSocket** keeps **one connection open in both directions**, so either side can send a message at any time (chat, live scores, multiplayer games). It starts as an HTTP request that is "upgraded" (URL starts with `ws://` or `wss://`).

Server-sent events (SSE). The browser opens a connection and the server **pushes** text events down it **one way only** (server to browser): a news ticker, progress bar, notifications. Simpler than WebSockets; the browser reconnects automatically (`EventSource` in JavaScript).

Polling (older idea): the browser asks again and again, for example every 5 seconds. Simple, but wasteful.

Progressive Web Apps (PWA) and service workers. A **service worker** is a JavaScript file that the browser runs in the background, separately from the page. It sits between the page and the network and can **cache files and answer requests when offline**, and receive push notifications. A PWA is an ordinary website plus: **HTTPS**, a **web app manifest** (a JSON file with name, icons, colours so it can be "installed" on the home screen), and a **service worker** (offline and speed).

```
// register the worker from your page
if ("serviceWorker" in navigator) {
  navigator.serviceWorker.register("/sw.js");
}
// sw.js: cache a file at install time, answer from cache first
self.addEventListener("install", e ⇒
  e.waitUntil(caches.open("v1").then(c ⇒ c.addAll(["/", "/style.css"]))));
self.addEventListener("fetch", e ⇒
  e.respondWith(caches.match(e.request).then(r ⇒ r || fetch(e.request))));
```

Web components let you make your **own HTML tags** that work in any page. They are three technologies: **Custom Elements** (define `<my-card>`), **Shadow DOM** (private DOM and CSS that does not leak out) and **HTML Templates** (`<template>`, inert markup you clone). **WebAssembly** is something else (fast compiled code in the browser) and is **not** a web component; neither is SVG.

```
<hello-user name="Asha"></hello-user>
<script>
  class HelloUser extends HTMLElement {
    connectedCallback() {
      const root = this.attachShadow({mode: "open"});   // Shadow DOM
      root.innerHTML = "<p>Hello, " + this.getAttribute("name") + "!</p>";
    }
  }
  customElements.define("hello-user", HelloUser);       // names must contain a hyphen
</script>
```

This shows "Hello, Asha!" Custom element names **must contain a hyphen**.

Idea 7: Data formats and responsive design

FORMAT	LOOKS LIKE	NOTES
JSON	`{"name": "Asha", "marks": [90, 85]}`	keys in double quotes; types: string, number, true/false, null, list, object; the default for web APIs
XML	`<student id="7"> <name>Asha</name> </student>`	tags and attributes; based on SGML; both human and machine readable
Markdown	`# Title`, `**bold**`, `- item`	plain text that converts to HTML; for documents and README files
CSV	`name,marks` lines	tables; simple

Checked with Python: `json.loads` of `{"name": "Asha", "marks": [90, 85], "passed": true, "address": null}` gives a dictionary whose `marks[1]` is 85, `passed` is the Python `True` and `address` is `None`. XML parsing with `xml.etree.ElementTree` gives tag `student`, attribute `id` equal to `'7'`, and element text always as strings.

Responsive design means one page that fits phone, tablet and desktop. Three tools:

1. The **viewport meta tag**: `<meta name="viewport" content="width=device-width, initial-scale=1">`. Without it, a phone pretends to be about 980 pixels wide and shrinks the page.
2. **Flexible layout**: percentages, `flex`, `grid`, `max-width`.
3. **Media queries**: `@media (max-width: 600px) { .menu { display: none; } }`.

> **IN YOUR PROJECT**
>
> **For your project viva.** Open your own pages and check: `<meta charset="utf-8">` and the viewport tag are present; the page uses `<header>`, `<nav>`, `<main>`, `<footer>`; every image has `alt`; every input has a `<label>`; text contrast is good; you can reach every button with only the Tab key. The examiner may ask: "Why UTF-8?", "What is the viewport tag for?" and "How is your site accessible?"

Your turn

YOUR TURN: 1. ENCODING SIZE

A message of 500 characters uses only the letters a to z, a space and a full stop. How many bits are needed with the minimum fixed encoding?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 2. UCS-4 TO ASCII

A document takes 152,000 bits in UCS-4. How many bits in 7-bit ASCII?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 3. UTF-8 BYTES

How many bytes does the string `"a₹é"` need in UTF-8?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 4. PICK THE PRINCIPLE

For each, say Perceivable, Operable, Understandable or Robust: (a) captions on a video; (b) a keyboard-reachable menu; (c) an error message that says what to fix; (d) markup that works across browsers and screen readers.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 5. WHICH STORAGE?

Choose cookie, localStorage, sessionStorage or IndexedDB: (a) remember the dark theme for next month; (b) keep a half-written form while the tab is open; (c) store thousands of records for offline use; (d) the browser must send a login token with every request.

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

The keyword match for this week gives about **24 distinct graded questions** (about 36 Quiz 1, 12 Quiz 2 and 35 End Term keyword hits; the count is coarse, and a few hits are stray list-tag questions). This week has few matched questions overall, but they repeat very reliably:

- **Encoding size calculations** are the single most frequent type (about 12 of the 24): minimum encoding of a pangram ("pack my box...", "sphinx of black quartz...", "the five boxing wizards..."), characters in a given file size, size growth after adding characters, and "UCS-4 / UCS-2 bits to ASCII 7-bit bits".
- **Accessibility principles** (about 6 questions): "which is NOT a perceivable principle", which principle an HTML snippet follows (alt text is Perceivable, label plus hint is Understandable), "input modalities beyond the keyboard" is Operable.
- **UTF-8 / UTF-32 statements** (true/false: variable vs fixed length; case-insensitive tags; external CSS preferred).
- **Web components** (2 questions: "which helps create custom elements", "which is not a web component": WebAssembly).
- **HTML history** (HTML4 vs HTML5, XHTML based on XML, SGML).
- No past questions were found on cookies vs localStorage, WebSockets, PWAs or Markdown in this match, but they are part of the week and likely for viva or newer papers.

WATCH OUT - REAL EXAM TRAP

- Pangram sums: count **distinct** characters including the space; use the ceiling of the log.
- "Not Perceivable" is the sentence about time or seizures: that is Operable.
- UTF-8 is variable (1 to 4 bytes) and fully Unicode-capable; UTF-32 is fixed at 4 bytes.
- HTML5 is **not** SGML-based and **is** backwards compatible; XHTML **is** XML-based.
- Web Assembly is not a web component.

Quick quiz

QUICK QUIZ

Which of the following is NOT a correct perceivable accessibility principle?

- Provide text alternatives for non-text content.
- Provide captions and other alternatives for multimedia.
- Make it easier for users to see and hear content.
- Give users enough time to read and use content.

Check

A sentence has 40 characters and uses 20 distinct characters. With minimum fixed-length encoding, its size is...

- 100 bits
- 160 bits
- 200 bits
- 320 bits

Check

A document takes 64,000 bits in UCS-4. Its size in 7-bit ASCII will be...

- 8,000 bits
- 14,000 bits
- 28,000 bits
- 64,000 bits

Check

Select all TRUE statements. (select all that apply)

- UTF-8 is a variable-length encoding that can represent all Unicode characters.
- UTF-32 is a fixed-length encoding where each character takes 4 bytes.

- Both UTF-8 and UTF-32 are variable length encodings.
- UTF-8 can only encode 65,536 characters.

Check

Which of these is NOT a web component technology?

- Custom Elements
- Shadow DOM
- HTML Templates
- WebAssembly

Check

Data in `sessionStorage` is...

- sent with every HTTP request
- kept for many years
- available to one browser tab and gone when it closes
- shared with all websites

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- Why did you use semantic tags such as nav and main?
- What are the four principles of web accessibility?
- Why do you write meta charset utf-8?
- What is the difference between a character, a code point and a byte?
- Compare cookies and localStorage.
- What is a WebSocket and how is it different from normal HTTP?
- What makes a website a PWA?
- Why do you need the viewport meta tag?

Recap

REMEMBER THIS

- **HTML 4** (SGML) then **XHTML** (strict XML) then **HTML5** (own parser, forgiving, backwards compatible, new features).
- **Semantic tags**: header, nav, main, section, article, aside, footer. New inputs (email, date, number, range, color) with `required`, `pattern`, `min`, `max` give built-in checks, but the server must validate.
- `<audio>`, `<video>`, `<canvas>` (pixels, JavaScript) and `<svg>` (vector markup) need no plug-ins.
- **POUR**: Perceivable (alt, captions, contrast), Operable (keyboard, enough time, no seizures, input modalities), Understandable (labels, predictable, input help), Robust (valid markup, ARIA).
- **Encoding**: ASCII 7 bits; Unicode is the table; UTF-8 variable 1 to 4 bytes (ASCII compatible); UTF-16 2 or 4; UTF-32 fixed 4; declare `<meta charset="utf-8">`. Minimum fixed bits per character is the ceiling of log2 of distinct characters.
- **Storage**: cookie (sent to server), localStorage (persistent), sessionStorage (one tab), IndexedDB (large structured).
- **WebSocket** two-way; **SSE** one-way push; **service worker** gives offline and caching in a PWA; **web components** = custom elements + shadow DOM + templates.
- JSON, XML and Markdown are text formats; the **viewport meta tag** plus media queries give responsive design.

CODE AND FACTS TO KEEP

- Bits per character (fixed, minimum) = ceiling of log2 of the number of distinct characters; total = characters x bits
- UCS-4 bits / 32 x 7 = ASCII 7-bit bits; UCS-2 bits / 16 x 7 = ASCII bits
- UTF-8: U+0000-007F 1 byte; to U+07FF 2; to U+FFFF 3; to U+10FFFF 4. UTF-32 always 4 bytes
- `<meta charset="utf-8">` and `<meta name="viewport" content="width=device-width, initial-scale=1">`
- `localStorage.setItem(k, v)`, `getItem(k)`, `removeItem(k)`; use `JSON.stringify` / `JSON.parse`
- POUR = Perceivable, Operable, Understandable, Robust; ARIA: `role`, `aria-label`, `aria-live`
- `ws://` / `wss://` WebSocket; `new EventSource(url)` for SSE; `navigator.serviceWorker.register("/sw.js")`
- Custom element: `customElements.define("my-tag", Class)` (name has a hyphen)

MAD1 week 12: Deployment: Git, WSGI, servers and the cloud

> **Official syllabus line:** Application Deployment

The one-minute story

THINK OF IT LIKE THIS

Deployment is opening your shop to the public. Until now your app was a dish cooked in your own kitchen (`flask run` on your laptop) and tasted only by you. Deploying means moving it into a proper restaurant: a **real kitchen** that can serve many customers at once (a production server like gunicorn), a **front desk** that greets customers and sends them to the right counter (nginx), a **lock on the door** (HTTPS), a **recipe book** so anyone can cook the same dish (requirements, Dockerfile), a **log book** (logging), and a **safe** for the money (backups).

And **Git** is the notebook where every change to the recipe is written down, with the date and the name of the cook, so you can go back to any earlier version and so that ten cooks can work on one recipe without erasing each other.

By the end of this week you can:

- use the Git workflow (init, add, commit, branch, merge, push, pull, clone) and predict what each command does;
- rebuild an environment from `requirements.txt` and keep secrets in environment variables;
- explain why `flask run` is not for production and how WSGI, gunicorn and nginx fit together;
- read a simple Dockerfile and a CI file, and tell PaaS, IaaS and SaaS apart;
- change a database model safely with Flask-Migrate, and prepare a deployment checklist and a README for your project.

Words you need

WORDS YOU NEED

- **Version control** - a system that records every change to your files so you can go back and collaborate.
- **Repository (repo)** - a folder tracked by Git, with its history in a hidden `.git` folder.
- **Commit** - one saved snapshot of the project with a message.
- **Branch** - a movable pointer to a commit; a separate line of work.
- **Remote** - a copy of the repo on a server (GitHub); the usual name is `origin` .
- **Virtual environment** - an isolated folder of Python packages for one project.
- **WSGI** - the standard interface between a Python web app and a web server.
- **gunicorn** - a production WSGI server for Python apps.
- **Reverse proxy** - a server (nginx) that receives requests and forwards them to your app.
- **Container** - a lightweight, isolated box that runs one program with all it needs.
- **Docker image** - the read-only recipe/template from which containers are started.
- **CI/CD** - continuous integration and continuous delivery or deployment: automatic test and release on every push.
- **Migration** - a versioned script that changes the database structure.

Idea 1: Git and GitHub

Git is a **distributed** version control system: every developer has a full copy of the whole history on their own machine, so Git works **offline** for everything except talking to a remote. **GitHub** (and GitLab, Bitbucket) is a **website that hosts Git repositories** and adds pull requests, issues and so on. **Git and GitHub are not the same thing.**

Three places a file can be:

1. **Working directory**: your files on disk.
2. **Staging area (index)**: the list of changes chosen for the next commit (`git add`).
3. **Repository**: the saved commits (`git commit`).

A file can be **in both the working directory and the staging area at the same time** (that is just "staged"). `git add` moves a change into the **staging area**; it does **not** send anything to GitHub.

```
git config --global user.name  "Asha"                # one-time identity
git config --global user.email "asha@example.com"    # exact command to change your e-
mail
git init                    # turn this folder into a repo (starts on branch main or
master)
git status                  # what changed, what is staged
git add app.py              # stage one file   (git add .  stages everything)
git commit -m "Add login"   # save the staged snapshot
git log --oneline           # history, one line per commit
git diff                    # unstaged changes (git diff --staged: staged ones)
```

Real run in a fresh folder with `app.py` , `.gitignore` and `.env` : `git status -s` first shows `?? .env` , `?? .gitignore` , `?? app.py` (untracked). After `git add app.py` it shows `A app.py` (staged). After `git commit -m first` , `git log --oneline` shows one line: `<id> first` .

THE IDEA

The daily loop: `git status` then `git add` then `git commit -m "message"` then `git pull` then `git push` .

- **init** creates a repo; **clone** copies an existing remote repo to your machine (and sets `origin`).
- **add** stages; **commit** saves; **push** uploads your commits to the remote; **pull** downloads the remote's new commits and merges them into your current branch (`fetch` + `merge`).
- `git remote add origin <url>` links your repo to GitHub; `git push -u origin main` pushes and remembers the link.

Branches and merging

A **branch** is only a **pointer to a commit**; creating one is nearly free. You work on a **feature branch**, then **merge** it into `main` .

```
git branch                  # list branches (* marks the current one)
git branch feature1         # CREATE a branch (does not switch)
git checkout feature1       # switch to it   (newer: git switch feature1)
git checkout -b feature2    # create AND switch in one step (newer: git switch -c
feature2)
# ... edit, add, commit on the branch ...
git checkout main           # go back to main
git merge feature1          # merge feature1 INTO the branch you are standing on (main)
git branch -d feature1      # delete the merged branch
```

Real run: after `git switch -c feature1`, one commit there, and back on `main`, `git branch` printed `feature1` and `* main`; `git merge feature1` printed `Fast-forward` (main had no new commits of its own, so Git just moved the pointer). The log then had 2 commits.

`git merge X` merges X into the *current* branch, not into main and not into "master". So to merge `feature1` and `feature2` into `main`:

```
git checkout main
git merge feature1
git merge feature2
```

(Choosing the commands where you first `git checkout feature1` and then merge `feature2` is wrong: you would merge into `feature1`.)

Merge conflicts. If both branches changed the same lines, Git stops and marks the file:

```
<<<<<<< HEAD
B
=======
A
>>>>>>> a
```

(Real output from a test where `main` wrote `B` and branch `a` wrote `A` in `x.txt`: Git said `CONFLICT (add/add)` and `Automatic merge failed`.) You edit the file to keep what you want, remove the markers, then `git add x.txt` and `git commit`.

Push rejected. If a teammate pushed first and you did not pull, your `git push` is **rejected** (`! [rejected] main → main (fetch first)`; real run). You must `git pull` (merge their work), fix conflicts if any, then push. The rule is "pull before you push".

SOLVED EXAMPLE: 1. PREDICT THE GIT WORKFLOW

Question. You work on a team project and run, in order: `git status`, `git add .`, `git commit -m "Fixed header bug"`, `git push`, `git pull`. A teammate pushed a change just before your push. What is wrong?

Solution. The push is refused because your local branch is behind the remote. The correct habit is `git pull` **before** `git push` (the real exam answer: "the pull should have been run before the push to avoid possible conflicts"). `git status` does not show remote conflicts, and push never merges automatically.

Other commands the papers ask about

COMMAND	MEANING
`git rm --cached file`	remove from the **staging area / tracking** but keep the file on disk
`git rm file`	remove from tracking **and** delete the file
`git del`	does **not exist** (a favourite wrong option)
`git restore file`	throw away unstaged edits in a file
`git stash`	put uncommitted work aside for a moment
`git branch name`	create a branch
`git checkout -b name`	create **and** switch
`git config --global user.email "..."`	set the e-mail used in commits

The first branch of a new repo is called `main` (new default) or `master` (older Git); both answers are accepted in the papers.

.gitignore is a text file listing files Git must not track: secrets (`.env`), virtual environments (`venv/`), caches (`__pycache__/`), local databases (`*.db`), uploads.

```
venv/
__pycache__/
*.pyc
.env
*.db
instance/
```

A **pull request (PR)** is a GitHub request to merge your branch into another (usually `main`): teammates review the code, CI runs the tests, and then someone clicks Merge. Never commit passwords or API keys; if you did, change them, because Git history remembers.

WATCH OUT - REAL EXAM TRAP

- `git add` is **not** "send to GitHub" (that is `git push`). Staging area is local.
- Git is **distributed**, not centralised; it works offline. Git is not GitHub.
- `git merge X` merges X into the branch you are **on**. Check the current branch first (`*` in `git branch`).
- `git rm --cached` keeps the file on disk. `git rm` deletes it.
- Branches are pointers to commits, not copies of the whole project.
- `git checkout -b name` both creates and switches.

Idea 2: Virtual environments, requirements and configuration

Different projects need different package versions. A **virtual environment (venv)** gives each project its own packages.

```
python -m venv venv                 # create (folder named venv)
source venv/bin/activate            # Linux/macOS      (Windows: venv\Scripts\activate)
pip install flask flask-sqlalchemy gunicorn
pip freeze > requirements.txt       # write exact installed versions to a file
pip install -r requirements.txt     # on another machine: rebuild the same set
deactivate
```

`pip freeze` lists the installed packages as `name==version`, which is what goes into `requirements.txt` (a question asks which command lists the packages installed from requirements.txt in the venv: `pip freeze`). Pin versions (`Flask==3.0.3`) so the server runs what you tested. Do **not** commit the `venv/` folder; commit `requirements.txt`.

Configuration and secrets in environment variables. The same code runs on your laptop and on the server, but the **settings differ** (secret key, database URL, debug flag). Do not hard-code them; read them from the environment.

```
import os
from flask import Flask

app = Flask(__name__)
app.config["SECRET_KEY"] = os.environ.get("SECRET_KEY", "dev-only")
app.config["SQLALCHEMY_DATABASE_URI"] = os.environ.get("DATABASE_URL",
"sqlite:///app.db")
app.config["DEBUG"] = os.environ.get("FLASK_DEBUG") == "1"
print(app.config["SECRET_KEY"], app.config["DEBUG"])
```

Real runs: `python cfg.py` prints `dev-only False`; `SECRET_KEY=abc FLASK_DEBUG=1 python cfg.py` prints `abc True`. On the server you set the real values in the environment (or a `.env` file that is **in `.gitignore`**, loaded with python-dotenv). One code base, different config per environment: **development**, **testing**, **production**.

Idea 3: Why not `flask run` in production? WSGI, gunicorn, nginx

`flask run` (or `app.run()`) starts Flask's built-in **development server**. It is great for coding, but it is **not** meant for real traffic: it is designed for one developer, handles few requests at a time, is not hardened for security, and with debug mode on can expose an interactive debugger that lets anyone run code. **Never run `debug=True` in production.**

WSGI (Web Server Gateway Interface) is the agreed interface: a web server calls a Python **callable** with the request information and gets back a response. A Flask `app` object **is** such a callable. Here is the whole idea in a few lines (real run: the client got `b'Hello from /hi'`):

```
def application(environ, start_response):
    body = ("Hello from " + environ["PATH_INFO"]).encode()
    start_response("200 OK", [("Content-Type", "text/plain"),
                              ("Content-Length", str(len(body)))])
    return [body]
```

Because Flask, Django and others all speak WSGI, any WSGI server can run any of them. **gunicorn** (Linux and macOS) and **uWSGI** are production WSGI servers: they start several **worker processes**, restart crashed workers and handle many connections.

```
pip install gunicorn
gunicorn -w 4 -b 127.0.0.1:8000 app:app     # 4 workers; module "app", variable "app"
```

`app:app` means "in the file `app.py`, use the object called `app`". If you use an application factory: `gunicorn "app:create_app()"`.

Reverse proxy. In front of gunicorn we put a fast web server, **nginx** (or Apache), which: terminates **HTTPS**, serves **static files** directly (CSS, JS, images) without bothering Python, forwards other requests to gunicorn, and can balance load between several app servers.

```
server {
    listen 80;
    server_name example.com;

    location /static/ {
        alias /srv/myapp/static/;          # nginx serves these itself
    }
    location / {
        proxy_pass http://127.0.0.1:8000;  # everything else goes to gunicorn
```

```
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $remote_addr;
    }
}
```

The path of one request: **browser, nginx (port 80/443), gunicorn (port 8000), Flask app, database**, and back.

Nginx vs Apache (a real MSQ): both are **open source**. Apache is **versatile** and has many modules and dynamic per-directory configuration. Nginx is generally **faster for static files and high traffic with low memory use** (it is event-driven). So "Apache is better for a large volume of concurrent connections than Nginx" is the false statement.

HTTPS. HTTP sends text anyone on the network can read. **HTTPS = HTTP over TLS**: the traffic is encrypted and the server proves its identity with a **certificate** issued by a certificate authority. **Let's Encrypt** gives free certificates (the `certbot` tool installs and renews them). Browsers show a padlock; cookies marked `Secure` are sent only over HTTPS. Many features (service workers, geolocation) require HTTPS.

> **THE IDEA**
>
> **Production request path.** Internet then **nginx** (HTTPS, static files, proxy) then **gunicorn** (WSGI server with workers) then **Flask** (your code) then **database**. The dev server replaces the middle two layers with one weak one.

Idea 4: Docker, cloud and CI/CD

Why Docker? "It works on my machine" fails when the server has a different Python or missing libraries. A **Docker image** packages your app with its exact Python, libraries and settings. A **container** is a running instance of an image: isolated, lightweight (it shares the host's kernel, unlike a full virtual machine), and starts in seconds.

```
# Dockerfile
FROM python:3.12-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["gunicorn", "-w", "4", "-b", "0.0.0.0:8000", "app:app"]
```

Line by line: start from an official Python image; work in `/app`; copy `requirements.txt` first and install (Docker caches this layer so rebuilds are fast); copy the rest of the code; document port 8000; the default command is gunicorn bound to `0.0.0.0` (all interfaces, needed inside a container).

```
docker build -t myapp .                    # build an image named myapp
docker run -p 8000:8000 myapp              # run a container; host port 8000 to
container port 8000
docker ps                                  # list running containers
```

docker-compose describes **several containers** that run together in one YAML file, for example the web app, a database and nginx:

```
services:
  web:
    build: .
    ports: ["8000:8000"]
    environment:
      DATABASE_URL: postgresql://user:pass@db/app
    depends_on: [db]
  db:
    image: postgres:16
    volumes: ["dbdata:/var/lib/postgresql/data"]
volumes:
  dbdata:
```

`docker compose up` starts everything; a **volume** keeps database files when the container is deleted (container storage is temporary).

Cloud service models

MODEL	YOU MANAGE	THE PROVIDER MANAGES	EXAMPLE
IaaS (infrastructure)	OS, runtime, app, data	servers, storage, network	a rented virtual machine (AWS EC2)
PaaS (platform)	your app and data only	OS, runtime, scaling	Render, Heroku, PythonAnywhere: "give me the code"
SaaS (software)	nothing; you just use it	everything	Gmail, Google Docs

Cloud means renting computing by the hour instead of buying machines; benefits: **scalability**, pay-as-you-use, reliability. For a student project, PaaS is the easiest.

CI/CD. Continuous integration (CI): every push is automatically built and tested, and the result reported, so errors are found early. **Continuous delivery:** after the tests pass, a release package is produced automatically and is ready to deploy. **Continuous deployment:** every passing change is automatically deployed into production. A small GitHub Actions file:

```
# .github/workflows/test.yml
name: tests
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.12"
      - run: pip install -r requirements.txt
      - run: pytest
```

On every push GitHub starts a fresh machine, installs the requirements and runs your pytest tests from week 10; a red cross on the commit means a test failed.

Idea 5: Migrations, logging, backups

Database migrations. Earlier you used `db.create_all()`. It creates **missing tables** but never changes an existing one. If you add a column to a model, the old database file does not get it. **Migrations** are numbered scripts that evolve the schema step by step, keeping the data. The tool is **Alembic**; **Flask-Migrate** wraps it for Flask.

```
# app.py
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///app.db"
db = SQLAlchemy(app)
migrate = Migrate(app, db)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
```

```
flask db init                       # once: creates the migrations/ folder
flask db migrate -m "create user"   # compare models with the database, write a script
flask db upgrade                    # APPLY the script to the database
# later you add:   email = db.Column(db.String(120))   to User
flask db migrate -m "add email"     # new script: Detected added column 'user.email'
```

```
flask db upgrade                    # apply it
flask db downgrade                  # undo the latest one
flask db current                    # which revision is the database at
```

All of this was run for real. The second migration's script contains `upgrade()` with `add_column('email', ...)` and `downgrade()` with `drop_column('email')`. After the second upgrade `flask db current` printed `38103fadeb9c (head)`; after `downgrade` it showed the first revision again. Remember: **migrate = generate script, upgrade = apply it** (a very common confusion). Always **read** the generated script (autogenerate can miss renames, and SQLite cannot drop or alter columns directly, so Alembic uses its batch mode). Commit `migrations/` to Git; on the server run `flask db upgrade` at each deployment.

Logging. In production nobody sits at the terminal, so write **logs** to a file or to the platform's log viewer. Use the `logging` module rather than `print`:

```
import logging
logging.basicConfig(level=logging.INFO, format="%(asctime)s %(levelname)s %(message)s")
log = logging.getLogger(__name__)

log.info("user %s logged in", "asha")
log.warning("disk almost full")
log.error("payment failed")
```

Levels in order: **DEBUG, INFO, WARNING, ERROR, CRITICAL**; setting `INFO` hides DEBUG. Never log passwords or tokens. **Monitoring** means watching health: is the site up (uptime check), how slow are the responses, how many errors; tools send an alert when something breaks (for example Sentry for errors, UptimeRobot for uptime). Gunicorn also writes an access log and an error log.

Backups. Servers die, disks fail, people run a wrong `DELETE`. Keep **regular, automatic, off-site copies** of the database and of uploaded files, and **test a restore** at least once, because a backup you cannot restore is not a backup. For SQLite you can copy the file when the app is idle, or use `sqlite3 app.db ".backup backup.db"`.

SOLVED EXAMPLE: 2. ADD A COLUMN THE RIGHT WAY

Question. Your live app has a `User` table with data. You add `phone = db.Column(db.String(15))` to the model. What do you do, in order?

Solution. (1) Stop thinking about `create_all()`: it will not alter the existing table. (2) `flask db migrate -m "add phone"` and open the new script to check that it only adds the column. (3) `flask db upgrade` on your laptop, run the tests. (4) Commit the script. (5) On the server: pull, `pip install -r requirements.txt`, `flask db upgrade`, restart gunicorn. (6) If something goes wrong: `flask db downgrade`, restore from backup.

SOLVED EXAMPLE: 3. READ A DEPLOYMENT

Question. What happens when the command `gunicorn -w 2 -b 127.0.0.1:8000 app:app` is running behind the nginx config of Idea 3, and a browser requests `/static/logo.png` and then `/login`?

Solution. `/static/logo.png` matches `location /static/`, so **nginx** returns the file from disk and gunicorn never sees it. `/login` matches `location /`, so nginx forwards it to `127.0.0.1:8000`; one of the **two** gunicorn workers runs the Flask view and sends the response back via nginx to the browser.

IN YOUR PROJECT

For your project viva: deployment checklist and README. Before you submit: (1) `requirements.txt` is complete and pinned; (2) `.gitignore` excludes `venv/`, `.env`, `*.db`, `__pycache__/`; (3) no secret or `DEBUG=True` in committed code, config read from environment variables; (4) tests pass (`pytest`) and migrations are committed; (5) the app runs with gunicorn, not `flask run`; (6) static files are served by nginx or the platform; (7) HTTPS is on; (8) logging is configured; (9) there is a database backup plan; (10) the repository has a clean commit history and a README.

README essentials: project name and one-line purpose; features; technology used (Flask, SQLAlchemy, ...); how to set up (clone, create venv, `pip install -r requirements.txt`, set environment variables, `flask db upgrade`); how to run and how to run the tests; default login for the demo; folder structure; screenshots; author and date. An examiner often tests whether a stranger could run the project in 5 minutes using only your README.

Your turn

YOUR TURN: 1. MERGE THE BRANCHES

`git branch` shows `feature1`, `* feature2`, `main`. Write the commands to put both features into `main`.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 2. GIT TRUE OR FALSE

(a) `git add` moves the local repo to the remote. (b) `git init` turns a folder into a repo. (c) Files in the staging area can be committed. (d) Git can be used only online. (e) Branches are pointers to commits.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 3. STOP TRACKING A FILE

You committed `secrets.env` by mistake but want to keep it on your disk. What do you do?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 4. PREDICT THE CONFIG

```
import os
print(os.environ.get("MODE", "dev"))
```

What does it print when run as `MODE=prod python x.py`, and when run as plain `python x.py`?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 5. IAAS, PAAS OR SAAS

(a) You rent a virtual machine and install Python yourself. (b) You push code to Render and it runs it. (c) You use Google Docs.

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

The keyword match for this week finds about **18 distinct graded questions**, nearly all in **End Term** papers (about 45 End Term keyword hits, about 1 in Quiz 2 and none in Quiz 1). That is a small sample and a coarse count (one hit was a stray HTML list question), so do not read too much into it. What it shows:

- **Git commands and workflow** (about 12 of the 18): merge two branches into main (the same question appears in three papers), `git rm --cached`, `git checkout -b`, `git branch`, `git config --global user.email`, "git is centralised" (false), "Git and GitHub are the same" (false), the default branch name (main or master), "the pull should come before the push".
- **Continuous integration / delivery / deployment** statements (2 questions; both statements correct).
- **Nginx vs Apache** (1 MSQ).
- **pip freeze** to list the packages in the virtual environment (1 question).
- No past question was matched on Docker, gunicorn, WSGI, migrations, HTTPS or cloud models in this sample, so treat those as viva and project topics; they are still in the week's scope.

WATCH OUT - REAL EXAM TRAP

- The order for merging into main: `checkout main` first, then `merge` each feature.
- `git del` is not a command. `git rm --cached` removes from staging only.
- Git is distributed; GitHub is a hosting service.
- `git checkout -b name` = create and switch.
- CI = automatically build and test on each change; continuous deployment = automatic release to production.
- Apache is versatile with modules; nginx is the fast, low-memory one for static files and many connections.

Quick quiz

QUICK QUIZ

You are on `main` and want the work of `feature1` and `feature2` in `main`. Which sequence is correct?

- `git merge main` then `git merge feature1` then `git merge feature2`
- `git checkout feature1` then `git merge feature2` then `git merge main`
- `git checkout main` then `git merge feature1` then `git merge feature2`
- `git push feature1` then `git push feature2`

Check

Which command removes a file from the staging area but keeps it in the directory?

- `git rm file`
- `git del file`
- `git rm --cached file`
- `git del --cached file`

Check

Select all TRUE statements about Git. (select all that apply)

- Files in the staging area can be committed.
- `git init` converts a local folder into a Git repository.
- `git add` moves the local repository to the remote repository.
- Git can be used only online.

Check

Why should you not deploy a Flask app with `app.run(debug=True)`?

- The development server is not built for real traffic and debug mode can expose an interactive debugger.
- Flask cannot run on Linux.
- It makes the database slower.
- HTTPS cannot be used with Flask.

Check

In a Flask-Migrate workflow, which command actually changes the database structure?

- `flask db init`

`flask db migrate`

`flask db upgrade`

`flask db current`

Check

Which statements are correct? (select all that apply)

Apache is versatile and supports many modules.

Nginx is generally faster for static files with low memory use.

Both Apache and Nginx are open-source web servers.

Apache is always better than Nginx for a very large number of concurrent connections.

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- How did you use Git in your project?
- What is the difference between Git and GitHub?
- What happens if two people change the same line and merge?
- Why is the Flask development server not used in production, and what do you use?
- What is WSGI?
- What is Docker and why use it?
- How do you change the database structure of a live app?
- How do you keep secrets such as the secret key out of your code?

Recap

REMEMBER THIS

- **Git** is distributed version control: working directory, staging area, repository. Loop: status, add, commit, pull, push. **GitHub** is only a host.
- **Branch** = pointer to a commit. `git checkout -b` creates and switches; `git merge X` merges X into the **current** branch; conflicts are fixed by hand then committed. Pull before push.
- **venv** isolates packages; `pip freeze > requirements.txt` and `pip install -r requirements.txt` recreate them. Secrets and settings come from **environment variables**, never from Git.
- `flask run` is a dev server only. Production: **nginx** (HTTPS, static files, proxy) in front of **gunicorn** (WSGI, workers) running Flask.
- **Docker**: Dockerfile builds an image; a container runs it; docker-compose runs several together. **IaaS / PaaS / SaaS** differ in how much you manage.
- **CI** tests every push; continuous delivery prepares the release; continuous deployment ships it.
- **Migrations**: `flask db init`, `migrate` (write script), `upgrade` (apply), `downgrade` (undo).
- Logs, monitoring and tested backups; deployment checklist and README for the project.

CODE AND FACTS TO KEEP

- Git setup: `git config --global user.name "N"`, `git config --global user.email "e"`
- Git basics: `git init`, `git status`, `git add .`, `git commit -m "msg"`, `git log --oneline`, `git diff`
- Branches: `git branch`, `git branch x`, `git checkout x` or `git switch x`, `git checkout -b x`, `git merge x`, `git branch -d x`
- Remote: `git clone URL`, `git remote add origin URL`, `git push -u origin main`, `git pull`
- Untrack but keep: `git rm --cached file`; ignore: `.gitignore`
- Env: `python -m venv venv`, `pip freeze > requirements.txt`, `pip install -r requirements.txt`
- Config: `os.environ.get("NAME", "default")`
- Serve: `gunicorn -w 4 -b 127.0.0.1:8000 app:app`; nginx `proxy_pass http://127.0.0.1:8000;`
- Docker: `docker build -t myapp .`, `docker run -p 8000:8000 myapp`, `docker compose up`
- Migrate: `flask db init`, `flask db migrate -m "msg"`, `flask db upgrade`, `flask db downgrade`
- Logging levels: DEBUG, INFO, WARNING, ERROR, CRITICAL

MAD2 week 1: JavaScript basics: values, functions and the DOM

> **Official syllabus line:** Basics of JavaScript

The one-minute story

THINK OF IT LIKE THIS

JavaScript is the waiter in a restaurant. HTML is the menu (what exists), CSS is the decoration (how it looks), and JavaScript is the waiter who actually *does* things: takes your order when you click a button, brings a new dish without rebuilding the restaurant, and remembers what each table asked for.

The waiter is fast but has odd habits. Ask him to add the text "5" and the number 3 and he does not complain, he hands you "53". Ask him for a name he was never given and he says `undefined` instead of crashing. Most of this week is learning those habits (types, coercion, hoisting, `this`) so you can **predict** exactly what he will do. The exam is full of "predict the output" questions, and they are all about these habits.

By the end of this week you can:

- explain where JavaScript runs and run it in the browser console and in Node;
- predict the output of code using `var`, `let`, `const`, hoisting, `==` and `===`;
- write functions (declaration, expression, arrow) and use `map`, `filter`, `reduce`;
- use objects, destructuring, the spread operator and JSON;
- read and write a small DOM program with events.

Words you need

WORDS YOU NEED

- **JavaScript (JS)** - the programming language of the web; runs in every browser and also outside it (Node).
- **Node.js** - a program that runs JavaScript on your computer or server, without a browser.
- **Console** - the place where `console.log` prints (browser dev tools, or the terminal in Node).
- **Scope** - the region of code where a name is visible.
- **Hoisting** - declarations are registered before the code runs, so some names exist "early".
- **Temporal dead zone (TDZ)** - the stretch between the start of a block and the line of a `let` / `const`, where using the name is an error.
- **Type coercion** - JavaScript silently converting a value to another type (`"5" * 2` becomes 10).
- **Truthy / falsy** - how a value behaves inside an `if` test.
- **Arrow function** - short function syntax `(x) ⇒ x * 2`, with its own rules for `this`.
- **Callback** - a function you hand to another function to be called later.
- **Destructuring** - unpacking array items or object properties into variables.
- **JSON** - a text format for data that looks like a JS object; `JSON.stringify` and `JSON.parse` convert.
- **DOM** - the page as a tree of objects that JavaScript can read and change.

Idea 1: What JavaScript is and where it runs

JavaScript was created for the browser, but the language itself is just a language. Two places run it:

WHERE	WHAT YOU GET	TYPICAL USE
Browser	the language plus `document`, `window`, `fetch`, `localStorage`, events	making pages interactive; your Vue app
Node.js	the language plus files, network servers, `require` / `import`	tools, build steps (`npm`), servers

Both use an **engine** (Chrome and Node use V8) that reads your code and runs it. JavaScript is **single-threaded**: one thing at a time (week 2 shows how it still waits for the network without freezing).

Put JavaScript in a page with a `script` tag. It runs when the browser reaches it.

```
<!DOCTYPE html>
<html>
  <body>
    <h1 id="title">Hello</h1>
    <script src="app.js"></script>   <!— put at the END of body, so the h1 exists —>
  </body>
</html>
```

```
// app.js
console.log("page loaded");              // goes to the browser console (F12)
document.getElementById("title").textContent = "Hi from JS";
```

`console.log` is your best friend: it prints any values, separated by spaces. Use `console.log(a, b)` to see two things at once. A `script` tag with `type="module"` (week 2) is deferred automatically; a normal one blocks the page while it loads, which is why it goes at the end of `body` (or gets the `defer` attribute).

YOUR TURN: 1. WHERE DOES IT RUN?

(a) Can `document.getElementById` run in Node? (b) Can `console.log` run in the browser? (c) Which runs a script file named `tool.js` from the terminal?

▸ **Need a hint?**

▸ **Show the full solution**

Idea 2: Variables, scope and hoisting

Three ways to declare a variable:

	`VAR`	`LET`	`CONST`
Scope	**function** (ignores blocks)	**block** `{ }`	**block** `{ }`
Re-assign	yes	yes	**no**
Re-declare in same scope	yes	error	error
Hoisted	yes, starts as `undefined`	yes, but in the **TDZ**	yes, but in the **TDZ**

THE IDEA

Use `const` by default, `let` when the value must change, and avoid `var`. `const` stops **re-assignment** of the name; it does **not** freeze the object inside (you can still `push` into a `const` array).

```
{ var v = 1; let l = 2; }
console.log(v);           // 1       var leaks out of the block
console.log(typeof l);    // undefined  l is not visible here

console.log(a);           // undefined  (var is hoisted, value not yet set)
var a = 5;

try { console.log(b); } catch (e) { console.log(e.name); }   // ReferenceError (TDZ)
let b = 5;

const list = [1, 2];
list.push(3);             // allowed
console.log(list);        // [ 1, 2, 3 ]
try { list = []; } catch (e) { console.log(e.name); }        // TypeError
```

Hoisting in one sentence: before running, JavaScript scans for declarations. `var x` is created and set to `undefined`; a **function declaration** is created with its whole body; `let` / `const` are created but locked until their line (TDZ).

```
console.log(add(2, 3));      // 5   declaration hoisted with its body
function add(a, b) { return a + b; }

try { sub(5, 2); } catch (e) { console.log(e.name + ": " + e.message); }
var sub = function (a, b) { return a - b; };   // only `var sub` is hoisted (as
undefined)
// prints: TypeError: sub is not a function
```

The most asked scope puzzle is `var` in a loop with a delayed callback:

```
for (var i = 0; i < 3; i++) setTimeout(() => console.log("var", i), 0);
for (let j = 0; j < 3; j++) setTimeout(() => console.log("let", j), 0);
// var 3, var 3, var 3, let 0, let 1, let 2
```

There is **one** `i` shared by all three callbacks, and it is 3 when the loop ends. `let` makes a **fresh** `j` **for each round**.

WATCH OUT - REAL EXAM TRAP

- `typeof` of an undeclared name is `"undefined"`, but `typeof` of a `let` in its TDZ **throws**.
- `var` inside `if` / `for` is visible outside it. `let` is not.
- `const obj = {}; obj.x = 1;` is fine. `obj = {}` is the error.
- Assigning to a name that was never declared (`total = 5`) quietly makes a global variable in sloppy mode. Do not.

Idea 3: Types, coercion and equality

There are **seven primitive types** and one object type:

TYPE	EXAMPLE	TYPEOF
number	`42`, `3.14`, `NaN`, `Infinity`	`"number"`
string	`"hi"`, `'hi'`	`"string"`
boolean	`true`, `false`	`"boolean"`
undefined	a variable never given a value	`"undefined"`
null	"deliberately empty"	`"object"` (an old bug)
symbol	`Symbol("id")` (a unique key)	`"symbol"`
bigint	`123456789012345678 90n`	`"bigint"`
object	`{}`, `[]`, functions, dates	`"object"` (functions: `"function"`)

There is **one** number type (no int/float split), and `0.1 + 0.2` is `0.30000000000000004`.

```
console.log(typeof 5, typeof "5", typeof null, typeof undefined, typeof [], typeof NaN,
typeof (() ⇒ 1));
// number string object undefined object number function
console.log(Array.isArray([]));   // true  (the right way to test for an array)
```

Coercion: `+` joins, everything else calculates

```
console.log("5" + 3);      // 53     one side is a string, so + joins
console.log("5" - 3);      // 2      - always means numbers
console.log("5" * "2");    // 10
console.log(true + 1);     // 2      true becomes 1
console.log(null + 1);     // 1      null becomes 0
console.log(undefined + 1);// NaN    undefined becomes NaN
```

```
console.log("abc" * 2);    // NaN    "not a number"
console.log(1 + 2 + "3");  // 33     left to right: 3, then "33"
console.log("1" + 2 + 3);  // 123
console.log(parseInt("42px"), Number("42px"), Number(""), parseInt(undefined));
// 42 NaN 0 NaN
```

`==` **converts types,** `===` **does not.** Always use `===` unless you know why.

```
console.log(0 == "", "0" == false, null == undefined);   // true true true
console.log(0 === "", null === undefined);               // false false
console.log(NaN == NaN, NaN === NaN, Number.isNaN(NaN)); // false false true
console.log([1] == 1, [] == false);                      // true true
```

Truthy and falsy

There are exactly **eight falsy values**: `false`, `0`, `-0`, `0n`, `""`, `null`, `undefined`, `NaN`. Everything else is truthy, including `"0"`, `"false"`, `[]` and `{}`.

```
console.log(Boolean([]), Boolean(""), Boolean("0"), !!{}, !!NaN);
// true false true true false
```

WATCH OUT - REAL EXAM TRAP

- `typeof null` is `"object"`; `typeof NaN` is `"number"`; `typeof []` is `"object"`.
- `"5" + 3` is `"53"` but `"5" - 3` is `2`.
- `NaN` is not equal to itself. Test with `Number.isNaN(x)`.
- `[]` and `{}` are **truthy**; `"0"` is truthy; `0` is falsy.
- `parseInt("12abc")` is 12 (reads digits until it fails); `Number("12abc")` is `NaN`.

YOUR TURN: 2. PREDICT THE OUTPUT

```
console.log(1 + "2" + 3, 1 + 2 + "3", "10" / 2, "a" - 1, [] + [], typeof typeof
1);
```

▸ Need a hint?

▸ Show the full solution

Idea 4: Operators, template literals, control flow and loops

- Arithmetic: `+ - * / % **` (`2 ** 3` is 8). Increments: `i++` , `i += 2` .
- Compare: `< > ≤ ≥ === !==` .
- Logic: `&&` , `||` , `!` . **They return one of the operands, not just true/false.** `||` returns the first truthy operand (or the last), `&&` returns the first falsy operand (or the last). The exam loves this.
- `??` (nullish coalescing) falls back only for `null` or `undefined` : `0 ?? 5` is `0` , but `0 || 5` is `5` .
- Ternary: `cond ? a : b` .

```
console.log(2 && 4, 0 && 2, 2 || 0, 0 || "", "" || "dflt");   // 4 0 2 "" dflt
console.log(0 || 5, 0 ?? 5, null ?? 5, undefined ?? "x");      // 5 0 5 x
const age = 17;
console.log(age ≥ 18 ? "adult" : "minor");                    // minor
```

Template literals use backticks, allow `${expression}` and span lines:

```
const name = "Asha", n = 3;
console.log(`Hello ${name}, you have ${n * 2} items.`);   // Hello Asha, you have 6
items.
```

Control flow and loops

```
const marks = 72;
if (marks ≥ 90) console.log("A");
else if (marks ≥ 60) console.log("B");      // B
else console.log("C");

switch (marks > 50) {            // switch uses ===, and needs break
  case true: console.log("pass"); break;
  default: console.log("fail");
}

for (let i = 0; i < 3; i++) console.log(i);                  // 0 1 2
let k = 0; while (k < 2) { k++; }                            // k is 2
for (const x of ["a", "b"]) console.log(x);                  // a b      values
for (const key in { p: 1, q: 2 }) console.log(key);         // p q      keys (strings)
for (const idx in ["a", "b"]) console.log(idx, typeof idx);  // 0 string / 1 string
```

THE IDEA

`for ... of` walks the **values** of something iterable (array, string). `for ... in` walks the **keys** (property names) of an object, always as **strings**. On an array `for … in` gives indexes as strings, so prefer `for … of` for arrays. `break` leaves the loop, `continue` skips to the next round.

YOUR TURN: 3. FIZZBUZZ-LITE

Print the numbers 1 to 15, but write "Fizz" for multiples of 3 and "Buzz" for multiples of 5 (and "FizzBuzz" for both).

▸ **Need a hint?**

▸ **Show the full solution**

Idea 5: Functions

Functions are **values** (first-class): you can store them in variables, pass them to other functions and return them.

```
function square(x) { return x * x; }          // declaration: hoisted
const cube = function (x) { return x ** 3; };   // expression: not usable before this
line
const double = (x) => x * 2;                    // arrow, one expression: implicit
return
const make = (a, b) => ({ sum: a + b });        // returning an object literal needs ( )
const greet = (who = "friend") => `Hi ${who}`;  // default parameter

console.log(square(3), cube(2), double(4), make(1, 2), greet(), greet("Ravi"));
// 9 8 8 { sum: 3 } Hi friend Hi Ravi
console.log((function () {})());                // undefined  no return means undefined
```

```
// first-class: pass a function, return a function
const apply = (f, v) => f(v);
console.log(apply(square, 5));       // 25
const adder = (n) => (x) => x + n;
console.log(adder(10)(5));           // 15
const nums = (...all) => all.length; // rest parameters collect the extras
console.log(nums(1, 2, 3));          // 3
```

`this` : a first look

Inside a normal function, `this` means **"the object before the dot at the call site"**. An arrow function has no `this` of its own: it uses the `this` of the code around it. Week 2 gives the full rules (`call` , `bind` , `apply`).

```
const user = { nick: "Asha", hi() { return "Hi " + this.nick; } };
console.log(user.hi());      // Hi Asha      object before the dot is user
const detached = user.hi;
```

```
try { console.log(detached()); } catch (e) { console.log("lost this"); }
// Hi undefined (sloppy script) or "lost this" (strict mode/module): the dot is gone
```

WATCH OUT - REAL EXAM TRAP

- A function declaration works before its line; a function expression (even `const f = () => {}`) does not.
- `x => { value }` with braces needs an explicit `return`; `x => value` returns it implicitly.
- `(x) => ({ a: x })`: parentheses so the braces are an object, not a function body.
- Extracting a method (`const f = obj.method`) loses `this`.

YOUR TURN: 4. REWRITE

Rewrite as an arrow function: `function area(w, h = 1) { return w * h; }`. What do `area(3)` and `area(3, 4)` give?

▸ **Need a hint?**

▸ **Show the full solution**

Idea 6: Arrays and their methods

```
const a = [10, 20, 30];
console.log(a.push(40), a);     // 4 [ 10, 20, 30, 40 ]   push returns the NEW LENGTH
console.log(a.pop(), a);        // 40 [ 10, 20, 30 ]      pop returns the removed item
console.log(a.length, a[0], a[5]);   // 3 10 undefined
console.log(a.slice(1, 3), a);       // [ 20, 30 ] [ 10, 20, 30 ]   slice COPIES,
original unchanged
console.log(a.splice(1, 1), a);      // [ 20 ] [ 10, 30 ]           splice CHANGES the
original
const b = [ ... a, 99];              // spread: copy into a new array
console.log(b, [ ... "hi"]);         // [ 10, 30, 99 ] [ 'h', 'i' ]
```

The "big four" take a **callback** and do not change the original array:

METHOD	RETURNS	CALLBACK DOES
`map(f)`	new array, same length	transform each item
`filter(f)`	new array, shorter	keep items where `f` is truthy
`reduce(f, start)`	**one value**	combine: `f(accumulator, item)`

METHOD	RETURNS	CALLBACK DOES
`forEach(f)`	**`undefined`**	just do something per item
`find(f)`	first matching item, or `undefined`	

```
const n = [1, 2, 3, 4, 5];
console.log(n.map(x ⇒ x * 2));                  // [ 2, 4, 6, 8, 10 ]
console.log(n.filter(x ⇒ x % 2 ⩵⩵ 1));          // [ 1, 3, 5 ]
console.log(n.reduce((acc, x) ⇒ acc + x, 0));   // 15
console.log(n.find(x ⇒ x > 3), n.find(x ⇒ x > 9));   // 4 undefined
console.log(n.indexOf(3), n.includes(7));       // 2 false
console.log(n.forEach(x ⇒ x));                  // undefined
console.log([10, 9, 1].sort());                 // [ 1, 10, 9 ]  default sort compares
AS TEXT
console.log([10, 9, 1].sort((p, q) ⇒ p - q));   // [ 1, 9, 10 ]
console.log(["1", "2", "3"].map(parseInt));     // [ 1, NaN, NaN ]  map also passes
the index!
```

SOLVED EXAMPLE: 1. A REAL EXAM CHAIN

```
const data = [
  { name: "Alpha", score: 40 }, { name: "Beta", score: 30 },
  { name: "Charlie", score: 50 }, { name: "Delta" },
];
const total = data
  .map(p ⇒ parseInt(p.score) || 0)
  .filter(score ⇒ score ⩾ 40)
  .reduce((acc, score) ⇒ acc + score, 0);
console.log("Total Score:", total);
```

Step by step: `map` turns each score into a number: `40, 30, 50`, and for Delta `parseInt(undefined)` is `NaN`, which is falsy, so `NaN || 0` is `0`: `[40, 30, 50, 0]`. `filter` keeps `⩾ 40`: `[40, 50]`. `reduce` adds from 0: `90`. Output: **`Total Score: 90`**.

WATCH OUT - REAL EXAM TRAP

- `push` returns the length, `pop` returns the item, `forEach` returns `undefined`, `splice` mutates, `slice` does not.
- Arrays are **objects**: `b = a` copies the **reference**; `b = [...a]` copies the items (a shallow copy).
- `["1","2","3"].map(parseInt)` is `[1, NaN, NaN]`. `sort()` without a function sorts as strings.

YOUR TURN: 5. CHAIN IT

`[3, 8, 1, 9]` : keep numbers above 2, double them, then add them up.

▸ **Need a hint?**

▸ **Show the full solution**

Idea 7: Objects, destructuring and JSON

An object is a bag of **key: value** pairs. A function stored as a value is a **method**.

```
const person = { name: "Asha", age: 20, skills: ["js", "py"], greet() { return "Hi " +
this.name; } };
console.log(person.name, person["age"], person.greet());   // Asha 20 Hi Asha
person.city = "Chennai"; delete person.age;                // add, delete
console.log("city" in person, person.age, person.zip);      // true undefined undefined
console.log(Object.keys({ a: 1, b: 2 }), Object.entries({ a: 1 }));
// [ 'a', 'b' ] [ [ 'a', 1 ] ]
```

Destructuring and **spread** are everywhere in Vue code:

```
const [first, , third = 0] = [1, 2];          // positions; third gets its default
const { name, skills: [s1] } = person;        // names must match; nested allowed
console.log(first, third, name, s1);          // 1 0 Asha js

const base = { x: 1, y: 2 };
const copy = { ...base, y: 20, z: 3 };        // copy, later keys win
console.log(copy, base);                      // { x: 1, y: 20, z: 3 } { x: 1, y: 2 }
const extra = { ...(false && { secret: 1 }) };    // spreading false adds nothing
console.log(extra);                           // {}
```

SOLVED EXAMPLE: 2. CONDITIONAL SPREAD (A PAST-PAPER PATTERN)

```
function isValidAge(age) { return !(age > 18); }
const validAgeParams = { voter_id: 123, constituency: "New Delhi" };
let some_value;                          // undefined
console.log(isValidAge(some_value) ? { name: "ABC",  ... (!isValidAge(19) &&
validAgeParams) }
                                    : { name: "XYZ" });
```

`undefined > 18` is `false`, so `isValidAge(undefined)` is `true`: take the **first** branch. `isValidAge(19)` is `!(true)` = `false`, so `!isValidAge(19)` is `true`, and `true &&` `validAgeParams` is `validAgeParams`, which is spread in. Output: `{ name: 'ABC', voter_id: 123, constituency: 'New Delhi' }`.

JSON

JSON is text. Keys and strings need **double quotes**; no functions, no comments, no trailing commas.

```
const text = JSON.stringify({ a: 1, b: undefined, c: () ⇒ 1, d: [1, undefined], e:
"hi" });
console.log(text);                          // {"a":1,"d":[1,null],"e":"hi"}
const back = JSON.parse('{"id":7,"tags":["x","y"]}');
console.log(back.id, back.tags[1]);         // 7 y
```

`stringify` **drops** `undefined` and function values in objects (and turns them into `null` inside arrays). Browser storage and `fetch` bodies are text, so you will convert with these two calls all the time.

YOUR TURN: 6. DESTRUCTURE

`const {a, b: {c}} = {a: 1, b: {c: 2, d: 3}};` What are `a` and `c`? Is `b` defined?

▸ **Need a hint?**

▸ **Show the full solution**

Idea 8: The DOM and events

The browser turns your HTML into a tree of objects called the **DOM**. `document` is the root. JavaScript finds elements, reads or changes them, and reacts to **events** (click, input, submit, keydown).

```
// browser only
const out = document.querySelector("#out");          // first match of a CSS selector
```

```
const all = document.querySelectorAll(".item");      // list of every match
out.textContent = "changed";                          // set text (safe)
out.style.color = "red";
out.classList.add("big");
document.querySelector("button").addEventListener("click", () ⇒ { out.textContent =
"clicked"; });
```

`input.value` is **always a string**, even for `type="number"`, so convert before calculating. That is the classic calculator bug.

SOLVED EXAMPLE: 3. A TINY CALCULATOR

```
<input id="a" type="number"> <select id="op"><option>+</option><option>−</option>
<option>*</option><option>/</option></select> <input id="b" type="number">
<button id="go">=</button> <span id="out"></span>
<script src="calc.js"></script>
```

```
// calc.js
function calculate(a, op, b) {
  switch (op) {
    case "+": return a + b;
    case "−": return a − b;
    case "*": return a * b;
    case "/": return b ≡≡ 0 ? "cannot divide by 0" : a / b;
    default: return "?";
  }
}
document.querySelector("#go").addEventListener("click", () ⇒ {
  const a = Number(document.querySelector("#a").value);   // value is a string:
convert!
  const b = Number(document.querySelector("#b").value);
  const op = document.querySelector("#op").value;
  document.querySelector("#out").textContent = calculate(a, op, b);
});
```

The pure function `calculate` can be tested in Node: `calculate(6, "*", 7)` is `42`. Without `Number(…)` the plus would join text: `"2" + "3"` is `"23"`. Frameworks like Vue (later weeks) remove this manual `querySelector` plumbing by tying page and data together.

IN YOUR PROJECT

Examiners ask "what is the difference between `let` and `var`" and "why `===`" even in a Vue project viva. Keep `const` / `let` only in your code and use `===` everywhere, so you can say it is a deliberate habit.

Exam radar

EXAM RADAR

The keyword scan for this week matches about **423 distinct graded questions** (about 570 of 848 in Quiz 1, 320 of 608 in Quiz 2 and 623 of 1471 in the End Term, by keyword, so approximate). Honest caution: "JavaScript" words appear in nearly every code question, and a large share of them are Vue questions whose answer is still decided by plain JavaScript rules. Pure-JavaScript "predict the output" questions mostly come in Quiz 2 and the End Term and test:

- **Array chains** with `map` / `filter` / `reduce` and `parseInt( ... ) || 0` (the score-total question above).
- **Operators returning operands** (`2 && 4` , `2 || 0` , `0 && 2`).
- **Object spread with `&&`** to add keys conditionally.
- **`this` with `call` and arrow functions** (week 2) and `var` becoming a global.
- **Storage values being strings** (`"10" + 5`), covered in week 3.

Typical questions: "what is logged?", "which statement about `let` / `const` is false?", "what does `x.sort()` return for numbers?".

WATCH OUT - REAL EXAM TRAP

- Strings: `+` joins and `*` calculates. A value read from `input` or storage is a string.
- `&&` / `||` return an operand; `??` ignores `0` and `""` .
- `parseInt("")` , `parseInt(undefined)` are `NaN` , and `NaN || 0` is `0` .
- `forEach` returns `undefined` ; `map` returns an array.

Quick quiz

QUICK QUIZ

What does this print? `console.log("5" + 3, "5" - 3)`

- 8 2
- 53 53

53 2

8 8

Check

What is the output? `console.log(2 && 4, 0 || 7, 0 ?? 7)`

2 7 7

4 7 0

true true true

4 0 7

Check

What is logged? `for (var i = 0; i < 3; i++) setTimeout(() => console.log(i), 0)`

0 1 2

3 3 3

2 2 2

Nothing

Check

Select all TRUE statements. (select all that apply)

`typeof null` is "object".

`const arr = []; arr.push(1)` is allowed.

`NaN === NaN` is true.

`let` variables are function scoped.

Check

`[10, 9, 1].sort()` returns...

[1, 9, 10]

[1, 10, 9]

[10, 9, 1]

an error

Check

Which statement about `slice` and `splice` is correct?

`slice` returns a copy and leaves the array unchanged; `splice` changes the array.

Both change the array.

`slice` changes the array; `splice` does not.

Both return a new array and leave the original unchanged.

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- ▸ **What is the difference between `var`, `let` and `const`?**
- ▸ **What is hoisting?**
- ▸ **Why prefer `===` over `==`?**
- ▸ **What is a callback and where did you use one?**
- ▸ **What is the difference between `map`, `filter` and `reduce`?**
- ▸ **What is JSON and how do you use it in JavaScript?**
- ▸ **What is the DOM?**

Recap

REMEMBER THIS

- JS runs in the browser (with `document`, `fetch`, storage) and in Node (with files and servers); it is single-threaded.
- `const` by default, `let` if it changes, no `var`. `var` ignores blocks; `let` / `const` have a TDZ; function declarations hoist whole.
- `+` joins when a string is involved; other operators convert to numbers. Use `===`. Eight falsy values; `[]` and `{}` are truthy.
- `&&` / `||` return operands; `??` only skips `null` / `undefined`.
- Functions are values. Arrow functions are short and take `this` from outside.
- `map` / `filter` / `reduce` return new values; `push` / `pop` / `splice` change the array; `forEach` returns `undefined`.
- Spread `...` copies; destructuring unpacks; JSON is text.
- `input.value` is a string; convert before calculating.

CODE AND FACTS TO KEEP

- `typeof null` -> "object"; `typeof NaN` -> "number"; `Array.isArray(x)`
- Falsy: `false 0 -0 0n "" null undefined NaN`
- `"5" + 3` -> "53"; `"5" - 3` -> 2; `1 + 2 + "3"` -> "33"
- `for (const v of arr)` values; `for (const k in obj)` keys (strings)
- `arr.reduce((acc, x) ⇒ acc + x, 0)` ; `arr.map(f)` ; `arr.filter(f)`
- `const { a, b: { c } } = obj;` `const [x, y] = arr;` `{ ...obj, k: v }` `[...arr, v]`
- `JSON.stringify(obj)` / `JSON.parse(text)`
- `document.querySelector("#id")` , `el.addEventListener("click", fn)` , `el.textContent`

MAD2 week 2: Advanced JavaScript: closures, promises and async/await

> **Official syllabus line:** Advanced JavaScript

The one-minute story

THINK OF IT LIKE THIS

Ordering coffee at a busy cafe. You tell the cashier your order and get a **token** with a number. You do not stand frozen at the counter; you sit down, read your phone, and when the barista calls your number you collect the coffee. JavaScript works like this. It has only **one cashier** (it does one thing at a time), so for slow jobs such as a network request it hands a note to the browser, carries on, and later is *called back* when the answer is ready.

A **Promise** is the token: "your result will come, or an error will". `async/await` is a way to write that waiting code so it reads top to bottom. A **closure** is a function that carries a small backpack with the variables it was born next to. And the cashier has a strict queue rule (the **event loop**) that decides who gets served first: this week you learn the rule well enough to predict the printing order in exam questions.

By the end of this week you can:

- explain closures and the scope chain, and predict `this` for method, arrow, `call` and `bind` ;
- write classes with `extends` and `super` , and explain prototypes;
- use `try/catch/finally` , `throw` , `?.` and `??` ;
- predict the printing order of code mixing `setTimeout` , promises and `await` ;
- write `fetch` code with `async/await` , `response.ok` and a JSON POST.

Words you need

WORDS YOU NEED

- **Closure** - a function together with the variables from the place where it was created.
- **Scope chain** - the look-up path: own scope, then the enclosing scope, and so on up to global.
- **Higher-order function** - a function that takes a function or returns a function (`map` , `filter`).
- **Prototype** - the object an object falls back to when a property is not found on itself.
- **Class** - a template for making objects; it is neat syntax over prototypes.
- **Module** - a JS file with its own scope that shares things with `export` and `import` .
- **Call stack** - the pile of functions currently running.
- **Task queue** - where finished timers and events wait (`setTimeout` callbacks).
- **Microtask queue** - a faster queue for promise callbacks; it is emptied **before** the next task.
- **Event loop** - the loop that moves waiting work onto the call stack when the stack is empty.
- **Promise** - an object for a future result: pending, fulfilled or rejected.
- **async/await** - syntax that lets you wait for a promise inside a function without `then` chains.
- **Fetch API** - the built-in way to make HTTP requests; it returns a promise.
- **Optional chaining** `?.` - read a property but stop with `undefined` if something on the way is `null` / `undefined` .

Idea 1: Scope chain and closures

When JavaScript meets a name, it looks in the current scope, then the one **outside** it, and so on to the global scope. This is the **scope chain**, and it is fixed by where the code is **written** (lexical scope), not where it is called.

A **closure** is what you get when an inner function remembers the variables of its outer function even after the outer function has finished.

```
function makeCounter() {
  let count = 0;                 // private: nobody outside can touch it
  return function () { count++; return count; };
}
const c1 = makeCounter(), c2 = makeCounter();
console.log(c1(), c1(), c1());   // 1 2 3
console.log(c2());               // 1    c2 has its own separate backpack
console.log(typeof count);       // undefined   (not visible outside)
```

`makeCounter` has finished, yet `count` lives on because the returned function still uses it. Each call to `makeCounter` creates a **new** `count` .

Closures also explain the loop puzzle from week 1:

```
const fs = [];
for (var i = 0; i < 3; i++) fs.push(() ⇒ i);     // all three share ONE i
```

```
console.log(fs.map(f => f()));                    // [ 3, 3, 3 ]
const gs = [];
for (let j = 0; j < 3; j++) gs.push(() => j);     // a new j per round
console.log(gs.map(g => g()));                    // [ 0, 1, 2 ]
```

THE IDEA

Callback: a function passed to another to be called later. **Higher-order function**: a function that takes or returns functions. `arr.map(fn)`, `setTimeout(fn, ms)` and `addEventListener("click", fn)` all take callbacks, and the callbacks usually use closures to reach the variables around them.

YOUR TURN: 1. CLOSURE

```
function bank() { let bal = 100; return { add: n => (bal += n), get: () => bal };
}
const a = bank(); a.add(50); const b = bank();
console.log(a.get(), b.get());
```

▸ Need a hint?

▸ Show the full solution

Idea 2: `this` rules

`this` is decided **at call time** (except in arrow functions). Learn the five cases:

HOW THE FUNCTION IS CALLED	`THIS` IS
`obj.method()`	`obj` (the object before the dot)
plain `f()`	`undefined` in strict mode and modules; the global object (`window`) in sloppy browser scripts
`f.call(x, a, b)` / `f.apply(x, [a, b])`	`x` (runs now)
`const g = f.bind(x)`	`x` forever (returns a **new** function)
arrow function	the `this` of the code **around** where it was written; `call` / `bind` cannot change it
`new F()` or `class`	the new object

```
"use strict";
const user = {
  name: "Asha",
  hi() { return "Hi " + this.name; },
  later() { return [1].map(() => this.name); },        // arrow keeps this = user
};
console.log(user.hi(), user.later());                  // Hi Asha [ 'Asha' ]
const lost = user.hi;
try { lost(); } catch (e) { console.log("TypeError: lost this"); }
const bound = user.hi.bind({ name: "Ravi" });
console.log(bound(), user.hi.call({ name: "Meena" }));      // Hi Ravi Hi Meena
function add(a, b) { return this.base + a + b; }
console.log(add.call({ base: 100 }, 1, 2), add.apply({ base: 100 }, [1, 2]));  // 103
103
```

SOLVED EXAMPLE: 1. A REAL `THIS` QUESTION

```
var first = 30;
const obj1 = { first: 50, second: () => { console.log("Value:", this.first); } };
const obj2 = { first: 80, second: function () { console.log("Value:",
this.first); this.second(); } };
obj2.second.call(obj1);
```

Run in a browser console or REPL (not as a Node module file). **Step 1:** `call(obj1)` runs `obj2.second` with `this = obj1`, so it prints `Value: 50`. **Step 2:** inside, `this.second()` is `obj1.second`, which is an **arrow function**. Its `this` is the outside `this`, the global object, where `var first = 30` created a global property `first`. It prints `Value: 30`. Answer: **50 then 30**. (Inside a Node file `var first` stays module-private and `this` is `{}`, so the second line would print `undefined`: that is why the question says REPL.)

WATCH OUT - REAL EXAM TRAP

- An arrow function **ignores** `call`, `apply` and `bind` for `this`.
- Extracting a method (`const f = obj.method; f()`) loses `this`.
- A top-level `var` becomes a property of the global object in a browser script or REPL; `let` / `const` do not.
- Arrow functions as object methods usually give the **wrong** `this`; use shorthand methods `hi() {}`.

Idea 3: Prototypes and classes

Every object has a hidden link (its **prototype**). If a property is not found on the object, JavaScript follows the link, and so on up the chain; if it reaches the end it gives `undefined`.

```
const Obj1 = { model: "AIR101", brand: "AIRBUS", type: "Passenger" };
const Obj2 = { __proto__: Obj1, wheels: "36", engines: 4 };
console.log(Obj2.model);               // AIR101      found on the prototype
console.log(Object.keys(Obj2).length); // 2           own keys only: wheels, engines
console.log(Obj1.engines);             // undefined   the link only goes UP, never down
console.log(Obj2.hasOwnProperty("model"), "model" in Obj2);   // false true
```

A **class** is tidy syntax for the same machinery:

```
class Animal {
  constructor(name) { this.name = name; }
  speak() { return `${this.name} makes a sound`; }
  static kind() { return "animal"; }
}
class Dog extends Animal {
  constructor(name, breed) { super(name); this.breed = breed; }   // super() FIRST
  speak() { return super.speak() + " (woof)"; }
}
const d = new Dog("Tommy", "lab");
console.log(d.speak());                       // Tommy makes a sound (woof)
console.log(d instanceof Dog, d instanceof Animal, Animal.kind());   // true true animal
console.log(Object.keys(d), typeof Animal);  // [ 'name', 'breed' ] function
```

- `constructor` runs on `new`; **`super(...)` must be called before `this`** in a child constructor.
- Methods live on the prototype, so `Object.keys(d)` shows only the data properties.
- `static` methods belong to the class, not to objects.

YOUR TURN: 2. PROTOTYPE LOOKUP

```
const a = { x: 1 }; const b = { __proto__: a, y: 2 }; const c = { __proto__: b,
z: 3 };
console.log(c.x, c.y, c.w, Object.keys(c));
```

▸ **Need a hint?**

▸ **Show the full solution**

Idea 4: Modules, errors and safe access

ES modules split code into files. Each file has its own scope; you share with `export` and take with `import`.

```
// browser: math.js
export const PI = 3.14;
export function area(r) { return PI * r * r; }
export default function hello() { return "hi"; }

// app.js
import hello, { area, PI as pi } from "./math.js";   // default without braces, named
with
```

In HTML: `<script type="module" src="app.js"></script>`. Modules are in strict mode, run after the page is parsed, and need the file path with extension (`./router.js`). Browsers refuse module files opened from `file://` , so use a small server (for example `npx http-server`).

Errors

```
function risky(n) {
  try {
    if (n < 0) throw new Error("negative");
    return "ok";
  } catch (e) {
    return "caught " + e.message;
  } finally {
    console.log("finally runs");      // runs on success, on error and even after return
  }
}
console.log(risky(1));                 // finally runs / ok
console.log(risky(-1));                // finally runs / caught negative
try { null.x; } catch { console.log("TypeError caught, no variable needed"); }
try { undefinedName; } catch (e) { console.log(e.name); }      // ReferenceError
```

`throw` can throw anything, but throw an `Error` . A `try/catch` only catches errors that happen **while it runs**: it cannot catch an error thrown later inside a `setTimeout` callback (use `.catch` or `await` for promises).

Optional chaining and nullish coalescing

```
const u = { profile: { city: "Pune" }, list: [] };
console.log(u.profile?.city, u.address?.city, u.address?.city ?? "unknown");  // Pune
undefined unknown
```

```
console.log(u.list?.[0], u.run?.());                                    //
undefined undefined
```

`a?.b` gives `undefined` instead of an error when `a` is `null` or `undefined`. `x ?? y` uses `y` only for `null` or `undefined` (not for `0` or `""`).

Idea 5: The event loop

Single thread, one **call stack**. Waiting work goes elsewhere (the browser timers, the network), and when ready its callback is queued:

1. **Call stack**: your running code. Always finishes completely first.
2. **Microtask queue**: promise callbacks (`then`, `catch`, `finally`, code after `await`) and `queueMicrotask`.
3. **Task queue**: timers (`setTimeout`), events, network callbacks.

> **THE IDEA**
>
> **The rule:** run the stack to empty. Then run **all** microtasks (including new ones they add). Then take **one** task from the task queue. Repeat. So **promise callbacks run before any `setTimeout`, even `setTimeout(fn, 0)`**.

```
console.log("1 start");
setTimeout(() ⇒ console.log("2 timeout"), 0);
Promise.resolve().then(() ⇒ console.log("3 promise"));
queueMicrotask(() ⇒ console.log("4 microtask"));
console.log("5 end");
// 1 start, 5 end, 3 promise, 4 microtask, 2 timeout
```

Synchronous lines `1` and `5` first; then microtasks in the order they were queued (`3`, `4`); then the timer task (`2`). A `setTimeout(fn, 0)` also does not mean "now": it means "after the stack and microtasks are done".

```
async function f() { console.log("a"); await null; console.log("b"); }
f(); console.log("c");          // a c b
```

An `async` function runs **synchronously until its first `await`**; the rest is a microtask afterwards.

YOUR TURN: 3. ORDER

```
console.log("A");
setTimeout(() ⇒ console.log("B"), 0);
Promise.resolve().then(() ⇒ console.log("C"));
(async () ⇒ { console.log("D"); await 0; console.log("E"); })();
console.log("F");
```

▸ **Need a hint?**

▸ **Show the full solution**

Idea 6: Promises

A promise is in one of **three states**: **pending**, **fulfilled** (has a value) or **rejected** (has a reason). Once settled it never changes.

```
const p = new Promise((resolve, reject) ⇒ {     // first param RESOLVES, second REJECTS
  setTimeout(() ⇒ resolve("done"), 100);
});
p.then(v ⇒ { console.log(v); return v + "!"; })        // then returns a NEW promise
 .then(v ⇒ console.log(v))                             // gets "done!"
 .finally(() ⇒ console.log("cleanup"));                // runs either way; passes
result along
```

Prints `done`, `done!`, `cleanup`.

Rules of a chain (these are tested again and again):

- Whatever a `then` **returns** becomes the next input. Return a **promise** and the chain **waits** for it.
- A **throw** (or a rejection) skips all following `then`s until a `catch`.
- A `catch` that returns normally **recovers**: the chain continues as fulfilled.
- `finally` always runs, but it need not be last, and a `catch` can be attached straight to a promise.

```
Promise.resolve(1)
  .then(v ⇒ { console.log("a", v); throw new Error("boom"); })
  .then(() ⇒ console.log("skipped"))
  .catch(e ⇒ { console.log("caught", e.message); return "recovered"; })
  .then(v ⇒ console.log("then", v));
// a 1, caught boom, then recovered
```

SOLVED EXAMPLE: 2. TIMING OF A CHAIN

```
const p1 = new Promise(res ⇒ setTimeout(() ⇒ res("First"), 1000));
p1.then(r ⇒ { console.log(r); return new Promise(res ⇒ setTimeout(() ⇒
res("Second"), 3000)); })
  .then(r ⇒ { console.log(r); return new Promise(res ⇒ setTimeout(() ⇒
res("Third"), 3000)); })
  .then(r ⇒ console.log(r));
```

Each `then` returns a new timed promise, so they run **one after another**: First at 1 second, Second at 4, Third at 7. Total: **7 seconds** (1 + 3 + 3), not 3. If the third promise called `reject`, the last `then` would be skipped and a `catch` would print the reason instead.

Combining promises

```
const wait = (ms, v, ok = true) ⇒ new Promise((res, rej) ⇒ setTimeout(() ⇒ (ok ?
res(v) : rej(v)), ms));
Promise.all([wait(30, "a"), wait(10, "b")]).then(console.log);       // [ 'a', 'b' ] in
input order
Promise.all([wait(30, "a"), wait(10, "x", false)]).catch(console.log); // x   (fails as
soon as one fails)
Promise.race([wait(30, "slow"), wait(10, "fast")]).then(console.log); // fast  (first
to settle)
Promise.allSettled([wait(5, 1), wait(5, 2, false)]).then(r ⇒ console.log(r.map(x ⇒
x.status)));
// [ 'fulfilled', 'rejected' ]
```

WATCH OUT - REAL EXAM TRAP

- A promise has **three** states, not two. "Only fulfilled and rejected" is the false statement.
- `new Promise((reject, resolve) ⇒ ...)` : parameter **names do not matter, order does**. The first is always resolve. With swapped names, calling `resolve("x")` actually rejects and runs the `catch`. (A real question: `data.find(...)` found the element, the code called the function named `resolve`, and the output was "Test Passed !! Element Found" from the `catch`.)
- `promise.then(cb)` does not wait: code after it runs first.
- `Promise.all` keeps input order, not finishing order.

Idea 7: async/await

`async` before a function makes it **always return a promise**. `await` pauses **that function** (not the whole program) until a promise settles, then gives you its value or throws its error. Use `try/catch` for errors.

```
const wait = (ms, v) => new Promise(res => setTimeout(() => res(v), ms));
async function test() {
  const x1 = await wait(1000, 1);
  console.log(2);                    // (2, 1 second)
  const x2 = await wait(3000, 3);
  console.log(x1);                   // (1, 4 seconds)
  console.log(x2);                   // (3, 4 seconds)
}
test();
console.log(4);                      // (4, 0 seconds)
```

Output with times: `(4, 0) (2, 1) (1, 4) (3, 4)`. The `4` comes first because `test()` hands back control at the first `await`; the two waits are **sequential**, so 1 + 3 = 4 seconds. To run waits **in parallel**, start both first: `const [a, b] = await Promise.all([wait(1000, 1), wait(3000, 3)])` takes only 3 seconds.

```
async function demo() {
  try {
    console.log(await Promise.resolve("Start"));
    await new Promise((res, rej) => setTimeout(() => rej("Error"), 50));
    console.log("never");
  } catch (e) { console.log("Caught:", e); }
  return "End";
}
demo().then(r => console.log(r));        // Start, Caught: Error, End
```

YOUR TURN: 4. WHAT IS A?

```
async function t() { const a = await new Promise(r => r(2 && 4)); console.log(a);
}
t();
```

▸ **Need a hint?**

▸ **Show the full solution**

Idea 8: The Fetch API, JSON and generators

`fetch(url, options)` returns a **promise of a Response**. `response.json()` returns **another promise** of the parsed body.

```
async function getJSON(url) {
  const res = await fetch(url);                    // rejects ONLY on network failure
  if (!res.ok) throw new Error("HTTP " + res.status);   // 404/500 still resolve; check
ok (200–299)
  return res.json();                               // rejects if the body is not valid
JSON
}
async function postJSON(url, obj, token) {
  const res = await fetch(url, {
    method: "POST",
    headers: { "Content-Type": "application/json", Authorization: "Bearer " + token },
    body: JSON.stringify(obj),                     // body must be text
  });
  if (!res.ok) throw new Error(res.statusText || "HTTP " + res.status);
  return res.json();
}
```

THE IDEA

Fetch rules. (1) A 404 or 500 is **not** an error for `fetch`; check `response.ok` or `response.status`. (2) The body is read once, with `.json()`, `.text()`. (3) A POST needs `method`, a `Content-Type` header and a `JSON.stringify`-ed `body`. (4) If the server returns HTML but you call `.json()`, you get a parse error (a rejected promise).

SOLVED EXAMPLE: 3. EXAM: WHICH MESSAGE IS LOGGED?

```
async function FetchFunct(url) {
  const response = await fetch(url).catch(() => { throw new Error("Network
Error"); });
  if (response.ok) {
    return await response.json().catch(() => { throw new Error("Unexpected
Error"); });
  } else throw new Error(response.statusText);
}
FetchFunct("dummyUrl").then(console.log).catch(err => console.log(err.message));
```

If the URL answers **200 with an HTML page**: the network works (no "Network Error"), `ok` is true, but `.json()` cannot parse HTML, so the inner `catch` throws **"Unexpected Error"**. A 404 would give the status text ("Not Found"); a dead server would give "Network Error".

Generators and iterators (briefly). A `function*` can pause with `yield`. Calling it returns an iterator whose `next()` gives `{ value, done }`. `for … of` and spread use this protocol.

```
function* count() { yield 1; yield 2; return 3; }
const it = count();
console.log(it.next(), it.next(), it.next(), it.next());
// { value: 1, done: false } { value: 2, done: false } { value: 3, done: true } {
value: undefined, done: true }
console.log([ … count()]);       // [ 1, 2 ]   spread ignores the return value
```

IN YOUR PROJECT

In your project, every `fetch` should have: `await`, a `response.ok` check, a `try/catch`, and a `loading` flag that disables the button while waiting (a real question asked exactly this: clicking a button quickly started overlapping requests, and the fix was to disable the button while loading). The examiner may ask "what happens if the server is down, and what if it returns 404?".

Exam radar

EXAM RADAR

The keyword scan matches about **160 distinct graded questions** for this week (about 138 of 848 in Quiz 1, 180 of 608 in Quiz 2, 263 of 1471 in the End Term; keyword based, so approximate). Many are Vue questions, but the JavaScript underneath is exactly this lesson. The questions that appear again and again are:

- **Promise chains**: what is printed, in what order, or how many seconds (the 1 + 3 + 3 = 7 second question; a `throw` in a `then` skipping to `catch`; a rejected third promise).
- **async/await ordering**: "Statement 3" prints before "Statement 1" because of the first `await`; try/catch inside async functions.
- `this` with `call` and arrow functions.
- **Prototype** lookup (`__proto__`, own keys count).
- **Fetch error handling** (which message when the body is HTML).
- True/false: a promise has three states; `finally` always runs; a `catch` need not follow a `then`.

WATCH OUT - REAL EXAM TRAP

- Sync code first, then microtasks (promises, after `await`), then timers.
- An async function runs synchronously until its first `await`.
- After a `throw`, only a `catch` (or a second argument to `then`) runs; the other `then`s are skipped.
- `fetch` does not reject on HTTP errors.
- Each `then` returning a new timed promise means the delays **add up**.

Quick quiz

QUICK QUIZ

What is printed? `console.log("A"); setTimeout(() => console.log("B"), 0); Promise.resolve().then(() => console.log("C")); console.log("D");`

- A B C D
- A D B C
- A D C B
- A C D B

Check

A chain of three `then`s returns promises that resolve after 1, 3 and 3 seconds. The minimum time for all to finish is...

- 3 seconds
- 1 second
- 7 seconds
- 0 seconds

Check

Which statement is FALSE about promises?

- A promise has the states pending, fulfilled and rejected.

- Promises can be chained.
- A promise has only two states, fulfilled and rejected.
- The constructor takes an executor function with resolve and reject.

Check

Select all TRUE statements. (select all that apply)

- A `finally` callback runs whether the promise was fulfilled or rejected.
- A `catch` can be attached directly to a promise without a `then` before it.
- A `finally` must always be the last call in a chain.
- `fetch` rejects its promise on a 404 response.

Check

`const f = obj.method; f()` loses `this` because...

- methods cannot be copied
- `this` is decided by how the function is called, and there is no object before the dot
- `const` freezes `this`
- arrow functions always lose `this`

Check

What does `class B extends A { constructor() { this.x = 1; } }` do when `new B()` runs?

- Creates the object with x = 1
- Returns undefined
- Throws a ReferenceError because super() was not called first
- Calls A's constructor automatically

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- What is a closure and where have you used one?
- Explain the event loop in simple words.
- What is the difference between a promise and async/await?
- How do you handle errors in `fetch`?
- What decides the value of `this`?
- What is the difference between a class and a prototype?
- What do `Promise.all` and `Promise.race` do?

Recap

REMEMBER THIS

- A closure keeps the variables where a function was born. Callbacks and higher-order functions rely on it.
- `this` : dot call gives the object, plain call gives `undefined` (strict) or global, `call` / `apply` / `bind` set it, arrows borrow it.
- Objects look up missing properties along the prototype chain; `class` and `extends` build that chain; `super()` comes first.
- Modules use `export` and `import` (`type="module"`); `try/catch/finally` , `throw` ; `?.` and `??` make access safe.
- Event loop: stack, then **all** microtasks (promises), then one task (timer).
- Promises: pending, fulfilled, rejected; `then` returns a new promise; `throw` skips to `catch` ; `finally` always runs.
- `async` functions return promises; `await` pauses only that function; sequential waits add up.
- `fetch` : check `response.ok` ; `res.json()` is a promise; a POST needs method, header and a stringified body.

CODE AND FACTS TO KEEP

- `const make = () => { let c = 0; return () => ++c; };`
- `f.call(obj, a)` , `f.apply(obj, [a])` , `const g = f.bind(obj)`
- `class B extends A { constructor(x) { super(x); } }`
- `export const x = 1; export default fn; import fn, { x } from "./m.js";`
- `try { } catch (e) { } finally { }` ; `a?.b` , `a ?? b`
- Order: sync, microtasks (`then` , after `await`), tasks (`setTimeout`)
- `Promise.all([..])` , `Promise.race([..])` , `Promise.allSettled([..])`
- `const res = await fetch(url, { method: "POST", headers: { "Content-Type": "application/json" }, body: JSON.stringify(obj) }); if (!res.ok) throw new Error(res.status); const data = await res.json();`

MAD2 week 3: The web frontend: SPAs, browser storage and tooling

> **Official syllabus line:** Introduction to Web Frontend

The one-minute story

THINK OF IT LIKE THIS

A paper form versus a whiteboard. An old-style website is a stack of paper forms: to change anything you hand the form back to the clerk (the server), who prints a whole new page and gives it to you. A **single-page application (SPA)** is a whiteboard on your desk: the clerk gives you the board once, and from then on you rub out and rewrite only the part that changed. You only call the clerk when you need fresh facts (an API call, with `fetch` from week 2).

But a whiteboard needs care. If your notes are only on the board (in the page's memory) and someone refreshes the page, they vanish. So you decide **where each fact lives**: on the clerk's shelf (server database), in your pocket (browser storage), on the whiteboard (memory), or written on the address (the URL). And keeping the board in sync with your notes by hand gets painful, which is why **frameworks** such as Vue exist. Lastly there is the toolbox (npm, bundlers, dev server) that builds and runs all this.

By the end of this week you can:

- compare multi-page and single-page apps and explain client-side routing;
- choose between cookies, `localStorage`, `sessionStorage` and IndexedDB and predict storage code;
- explain what a service worker and a PWA are;
- explain why frameworks and the virtual DOM exist, and what `package.json`, `node_modules`, Vite or webpack and Babel do;
- say where the state of a web application lives, and why HTTP being stateless matters.

Words you need

WORDS YOU NEED

- **MPA (multi-page app)** - every click asks the server for a new full HTML page.
- **SPA (single-page app)** - one HTML page is loaded once; JavaScript redraws parts and fetches data.
- **Client-side routing** - changing the URL and the view in the browser without asking the server for a new page.
- **Origin** - scheme + host + port, such as `http://localhost:8080`. Storage is separated per origin.
- **Cookie** - a tiny named value the server (or script) stores in the browser; it travels with every request to that site.
- **Web Storage** - `localStorage` and `sessionStorage`: key-value text in the browser.
- **IndexedDB** - a bigger, asynchronous database in the browser for objects and files.
- **Service worker** - a background script that sits between page and network and can serve cached files offline.
- **PWA** - a web app made installable and offline-capable with a manifest and a service worker.
- **Virtual DOM** - a light copy of the page in memory used to work out the smallest real change.
- **npm** - Node's package manager; reads `package.json` and fills `node_modules`.
- **Bundler** - a tool (webpack, Vite) that combines your many files into a few optimized ones.
- **Transpiling** - converting new JavaScript into older JavaScript old browsers understand (Babel).
- **Hot reload** - the dev server updates the page when you save, without a full refresh.
- **Stateless** - the server does not remember you between requests by itself.

Idea 1: Multi-page apps, single-page apps and routing

	MPA (TRADITIONAL)	SPA
Click on a link	full page request, white flash	JavaScript swaps the view, no reload
What the server sends	finished HTML for each page	one HTML shell + JS once, then **JSON** from APIs
Speed after first load	slower (whole page again)	fast, smooth, app-like
First load	quick (just HTML)	slower (download the JS bundle first)
SEO / search engines	easy: the HTML already has the content	harder: content is built by JS (fixed by server-side rendering)
Server load	higher (renders every page)	lower (only data)
Example	a classic shopping site, a Flask app with Jinja templates	Gmail-style apps, your Vue project

THE IDEA

Client-side routing: the SPA watches the URL and shows the matching component, without a request for a new page. Vue Router 3 has two modes:

- **hash mode** (default): `http://localhost:8080/#/records/2`. The part after `#` never goes to the server, so the server needs no special setup.
- **history mode**: `http://localhost:8080/records/2`, clean URLs, but the server must send the same `index.html` for every path, or a refresh gives a 404.

This is why exam questions give both `.../records/1` and `.../#/records/1` and ask which one renders: in **hash mode** the router only reads what is after `#`. In **history mode** `/#/inventory/102` does not match the route `/inventory/:id`, because the path is then just `/`.

A SPA still talks to a backend: your Flask API returns JSON and the Vue code draws it (week 2's `fetch`).

WATCH OUT - REAL EXAM TRAP

- SPAs do **not** load the whole page on every click; that is the MPA behaviour.
- "The idea of an SPA is to improve SEO" is **false**. SEO is normally the SPA's weak spot.
- gmail.com is SPA-like; amazon.in is a classic multi-page site. "Both are SPAs" is false.
- Papers disagree on one claim: "client-side routing gives better SEO by default" is marked correct in one End Term and wrong in another. Safe, agreed benefits: faster navigation, no full reload, less load on the backend.

YOUR TURN: 1. SPA OR MPA

Which of these describe an SPA? (a) The server returns finished HTML for every click. (b) After the first load only JSON travels. (c) The URL can change without a page reload. (d) It works best for search-engine visibility without extra work.

▸ **Need a hint?**

▸ **Show the full solution**

Idea 2: Browser storage

A web page can keep data in the browser in four main ways. Know their differences cold; the papers ask matching and "which statement is true" questions.

	COOKIE	LOCALSTORAGE	SESSIONSTORAGE	INDEXEDDB
Size (about)	about 4 KB each	about 5 MB per origin	about 5 MB per origin	very large (a share of the disk)

	COOKIE	LOCALSTORAGE	SESSIONSTORAGE	INDEXEDDB
Lifetime	until its expiry date (or end of session if none)	**forever** until cleared by code or user	until the **tab** is closed (survives reload)	until cleared
Sent to the server	**yes, on every request** to that site	no	no	no
Shared across tabs	yes (same site)	**yes** (same origin)	**no**, each tab has its own	yes (same origin)
Who can read	the server, and JS (unless `HttpOnly`)	JS of the same origin	JS of that tab, same origin	JS of the same origin
Data type	small text	**text only** (strings)	**text only** (strings)	objects, blobs, files, audio, video
API	`document.cookie`, `Set-Cookie` header	`getItem`, `setItem`, `removeItem`, `clear`	the same four	asynchronous, with requests or promises

All storage is **per origin** and **per browser**: nothing is shared across devices or across different websites. Storage is not secure: any script on the page can read it, so never keep passwords there (tokens are stored there in practice, with that risk).

```
// browser
localStorage.setItem("theme", "dark");           // value becomes the string "dark"
localStorage.getItem("theme");                   // "dark"
localStorage.getItem("missing");                 // null   (getItem returns null if
absent)
localStorage.missing;                            // undefined   (property style returns
undefined)
localStorage.setItem("n", 5);                    // stored as "5"
typeof localStorage.getItem("n");                // "string"
localStorage.setItem("user", JSON.stringify({ id: 1 }));
JSON.parse(localStorage.getItem("user")).id;     // 1    objects need JSON both ways
localStorage.removeItem("theme"); localStorage.clear();
```

THE IDEA

Everything in Web Storage is a string. `setItem("n", 5)` stores `"5"`; `getItem("n") + 1` is `"51"`; `getItem("n") * 2` is `10`. Objects need `JSON.stringify` to store and `JSON.parse` to read.

SOLVED EXAMPLE: 1. PREDICT THE STORAGE OUTPUT

```
// browser
localStorage.setItem('counter', '0');
sessionStorage.setItem('total', '5');
for (let i = 0; i < 3; i++) {
  let counter = localStorage.getItem('counter');
  let total = sessionStorage.getItem('total');
  counter += 2;
  total *= 2;
  localStorage.setItem('counter', counter);
  sessionStorage.setItem('total', total);
}
sessionStorage.clear();
console.log(localStorage.getItem('counter'));
console.log(sessionStorage.getItem('total'));
```

`counter` is the string `"0"`, so `counter += 2` **joins**: `"02"`, then `"022"`, then `"0222"`. `total` is `"5"`, and `*=` converts to a number: 10, 20, 40. Then `sessionStorage.clear()` removes everything, so the last line is **`null`** (not `undefined`: `getItem` returns `null`). Printed: `0222` and `null`.

A sibling question starts with `score = '10'` and `multiplier = '3'`, runs the loop twice with `score + 5` and `multiplier * 2`: `"105"`, `"1055"` and `6`, `12`. The printed `1055` and `12` appeared as "None of these" among the options.

SOLVED EXAMPLE: 2. "REFRESH THE PAGE N TIMES"

A very common long question: a Vue-style component whose `mounted()` reads storage, then writes it back changed. The skill is to **trace the storage after each refresh**.

```
// browser: runs on every page load
let name, age;
try {
  const parts = localStorage.getItem("name").split(" ");   // fails with
TypeError on null
  name = parts[0];
  age = parts[1];
  localStorage.setItem("name", parts[0] + " " + name);     // writes back
} catch {
  name = "Default"; age = "Default";
}
```

Suppose the user types `IIT Madras` and a button saves it to `localStorage.name`.

- **First ever load**, storage empty: `getItem` returns `null`, `.split` throws, the `catch` sets `Default, Default`.
- **Refresh 1**: reads `"IIT Madras"`. `name = "IIT"`, `age = "Madras"`. Shows **IIT, Madras**. Writes back `"IIT" + " " + "IIT"` = `"IIT IIT"`.
- **Refresh 2**: reads `"IIT IIT"`. Shows **IIT, IIT**. Writes `"IIT IIT"` again. It stays that way forever.

Always ask: what was in storage **before** this load, what did the code show, and what did it write **after**?

WATCH OUT - REAL EXAM TRAP

- `localStorage.getItem(missing)` is `null`; `localStorage.missing` is `undefined`.
- Values come back as strings: `"3" + 2` is `"32"`.
- `sessionStorage` is **per tab**: a fresh new tab does not see it; `localStorage` is shared by all tabs of the origin and survives closing the browser.
- A reload does **not** clear `sessionStorage`; closing the tab does.
- Storage does not travel between devices and is **not** sent to the server (unlike cookies).

YOUR TURN: 2. MATCH THE STORAGE

Which storage fits: (a) the user's dark-mode choice, kept after the browser restarts; (b) a half-filled form that should vanish when the tab closes; (c) a login cookie the server must receive on every request; (d) 200 MB of cached video for offline use.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 3. PREDICT

```
// browser
sessionStorage.setItem("username", "course_user");
const a = sessionStorage.getItem("username");
sessionStorage.removeItem("username");
const b = sessionStorage.getItem("username");
sessionStorage.clear();
const c = sessionStorage.username;
```

What are `a`, `b`, `c`?

▸ **Need a hint?**

▸ **Show the full solution**

Idea 3: Statelessness and where state lives

HTTP is stateless: each request stands alone, and the server does not automatically remember the previous one. Yet apps remember you are logged in and what is in your cart. The state is kept somewhere and **explicitly sent along** with requests.

WHERE	EXAMPLES	SURVIVES RELOAD?	SHARED WITH OTHERS?
Server (database, cache)	users, products, orders	yes	yes, the source of truth
Client memory (JS variables, Vue `data`)	which tab is open, loading flag, text typed so far	**no**	no
Browser storage	theme, token, saved draft	yes	no (this browser only)

WHERE	EXAMPLES	SURVIVES RELOAD?	SHARED WITH OTHERS?
URL (path, `?query`, `#`)	current page, search text, page number	yes, and bookmarkable	yes, by sharing the link
Cookie / token	session id, login token	yes	sent with each request

- **UI state (ephemeral state)**: short-lived interface facts such as a loading spinner, which tab is selected, a dropdown open or closed. It lives in the component and can be lost.
- **Application state**: real data (users, products), usually owned by the server, and cached in the frontend (week 4's Vuex).

THE IDEA

Rule of thumb. Truth lives on the server. Convenience copies live in the client. What should survive a reload, or be shared by link, goes in storage or in the URL. Because HTTP is stateless, the client or server holds the state and **sends it in each request** (cookie, token, query string).

YOUR TURN: 4. WHERE DOES IT LIVE?

Where would you keep: (a) the list of all students; (b) whether a modal is open; (c) the page number of a table you want to bookmark; (d) the login token?

▸ Need a hint?

▸ Show the full solution

Idea 4: Why frameworks exist

With plain JavaScript you change the page by hand:

```
// browser: the manual way
let count = 0;
document.querySelector("#plus").addEventListener("click", () => {
  count++;
  document.querySelector("#out").textContent = count;      // you must remember to
update the page
});
```

With ten pieces of data shown in five places, you will forget one, and the page and the data drift apart. A framework makes the page a **function of the state**: change the data and the framework updates the page.

```
<div id="app">{{ count }} <button @click="count++">+</button></div>
<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
<script>
  new Vue({ el: "#app", data: { count: 0 } });    // Vue 2: click changes data, page follows
</script>
```

- **Virtual DOM.** Touching the real DOM is slow. The framework keeps a light JavaScript copy of the page. When data changes it builds a **new** virtual tree, **compares** it to the old one (the "diff") and patches only the real nodes that differ. So Vue "detects the change and updates efficiently"; it does not reload the page or rebuild everything.
- **Components.** The page is built from reusable pieces (a navbar, a card), each with its own template, data and methods. Advantages: **reusability, maintainability, easier UI management**.
- **Reactivity (Vue 2).** Vue wraps your `data` properties so it knows when they change. A limit: a property added later, or `arr[0] = x`, is not noticed; use `this.$set( ... )` or array methods like `push`. (Vue 3 uses Proxies and has no such limit.)

	VUE	REACT	ANGULAR
Kind	progressive framework	UI library	full framework
Made by	Evan You and community	Meta	Google
Templates	HTML-like templates, optional single-file components	JSX (HTML inside JS)	HTML templates + TypeScript
Learning curve	gentle; can be added with one script tag	moderate	steep, many built-in parts

The course uses **Vue 2** with Vue Router 3 and Vuex 3. In newer Vue (3) the same ideas apply, with Pinia replacing Vuex.

Idea 5: Tooling, npm, bundlers and dev servers

Two ways to start with Vue:

	CDN SCRIPT TAG	VUE CLI / BUILD TOOLS
Setup	one `script` tag, no install	`npm` + a generated project
Templates	compiled in the browser	compiled at build time (faster)
`.vue` single-file components	no	yes
Modern syntax, minify, tree-shaking	no	yes (via Babel and the bundler)
Best for	demos, small pages, learning	real projects, teams

npm and `package.json`. `npm` downloads packages. The project's `package.json` lists them. `npm install` reads it and fills the `node_modules` folder (huge; never commit it, it is rebuilt from `package.json` and `package-lock.json`).

```
{
  "name": "my-vue-app",
  "version": "0.1.0",
  "scripts": {
    "serve": "vue-cli-service serve",
    "build": "vue-cli-service build",
    "lint": "vue-cli-service lint"
  },
  "dependencies": { "vue": "^2.6.14" },
  "devDependencies": { "@vue/cli-service": "^5.0.0", "babel-eslint": "^10.1.0" }
}
```

- `dependencies` are needed by the running app; `devDependencies` only while developing and building.
- `^2.6.14` means "2.6.14 or any newer 2.x" (same major version); `~2.6.14` allows only newer patches.
- `scripts` are run with `npm run name`. `npm run serve` starts the **dev server** (usually `localhost:8080`) with **hot reload**. `npm run build` makes the optimized production files in the `dist` folder and does **not** start a server.
- `npm install vue@2` adds a dependency; `npm install -D pkg` adds a dev dependency; `npx` runs a tool without installing it globally.

SOLVED EXAMPLE: 3. WHICH COMMAND?

A developer creates a Vue 2 project and runs `npm run build` hoping to see the app in the browser. What happens?

`build` compiles and bundles the app for production into `dist/`. It starts nothing. She should run `npm run serve` for the local dev server with hot reload. The same two scripts exist in every Vue CLI project because `package.json` defines them.

What the build tools do.

- **Bundler** (**webpack** inside Vue CLI; **Vite** in the newer Vue starter): follows your `import`s, joins many files into a few, **minifies** them, removes unused code (tree-shaking) and handles `.vue` files, CSS and images. Webpack bundles first and is configuration-heavy; Vite serves your modules directly during development, so it starts and updates very fast.
- **Transpiling** (**Babel**): rewrites new syntax (arrow functions, `?.`, classes) into older JavaScript so old browsers can run it.
- **Dev server + hot reload (HMR):** save a file and the changed part is swapped into the open page, often keeping your current state.

WATCH OUT - REAL EXAM TRAP

- `npm run build` does not start the app; `npm run serve` does (in Vue CLI projects).
- `node_modules` is not committed and not edited by hand.
- Vue **can** be installed through npm (a False-statement question said "it cannot be").
- A CDN script is fine for a demo; a real project uses the build tools.

Idea 6: Service workers, PWAs, responsive design and accessibility

Web worker: runs JavaScript on a **separate thread** so heavy work does not freeze the page. It cannot touch the DOM; it talks to the page with `postMessage`.

Service worker: a special worker that sits **between your page and the network**. It can intercept requests, answer from a **cache**, work offline, and receive push notifications. It runs only on **HTTPS** (and `localhost` for development).

```
// browser
if ("serviceWorker" in navigator) {
  navigator.serviceWorker.register("/sw.js");        // sw.js decides what to cache and
serve
}
```

PWA (Progressive Web App): an ordinary web app plus

- a **web app manifest** (a JSON file: name, icons, start URL, display mode), which lets it be **installed on the home screen** and look like a native app;
- a **service worker**, giving **offline** support and fast loads;
- served over **HTTPS**.

It needs no app store and no native code, and it can work with a poor or no connection.

Responsive design: one page that adapts to any screen size. Tools: the viewport tag, CSS media queries, flexbox and grid, relative units.

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

```
.card { width: 30%; }
@media (max-width: 600px) { .card { width: 100%; } }   /* on a phone, cards go full
width */
```

Accessibility reminder: use real elements (`button`, `label`, headings), give images `alt` text, make everything usable with a keyboard, keep good colour contrast, and let screen readers understand the page (ARIA only when plain HTML is not enough).

YOUR TURN: 5. TRUE OR FALSE

(a) A PWA must be downloaded from an app store. (b) A service worker lets a PWA show cached pages offline. (c) PWAs can be added to the home screen. (d) A web worker can change the DOM directly.

▸ **Need a hint?**

▸ **Show the full solution**

Idea 7: Debugging the frontend with dev tools

Press **F12**. The tabs you use:

TAB	USE
Elements	see and edit the live DOM and CSS
Console	read errors and `console.log`, run JavaScript
Network	see every request: URL, method, **status code**, request and response, timing (check your `fetch` here)
Application	inspect and edit **cookies, localStorage, sessionStorage, IndexedDB**, service workers and the manifest
Sources	set **breakpoints** and step through code
Device toolbar	test a phone-sized screen (responsive design)

The **Vue devtools** extension adds a tab showing components, their `data`, and Vuex state. Debug order: Console for the error message, Network for what the server answered (status, response body), Application for what is stored.

IN YOUR PROJECT

Be ready to **demo** this in the viva: open the Network tab and show your API call and its status code; open Application and show the token in storage; change a store value and show the page update. Also be ready for "why is your project an SPA, and what happens on refresh?".

Exam radar

EXAM RADAR

The keyword scan finds about **32 distinct questions** for this week (about 7 in Quiz 1, 57 in Quiz 2 and 39 in the End Term by keyword, so approximate). Most are storage questions; they are also the easiest to get right:

- **Predict the storage output** (string joining with `+`, `null` after `clear`), and "refresh the page N times" traces.
- **True/false on Web Storage** (`localStorage` never expires; `sessionStorage` is per tab, not shared across tabs).
- **SPA/PWA statements** (client-side routing, service workers, home screen, offline).
- **Client-side storage advantages** (preferences, offline documents, IndexedDB for audio/video; not "data stored across devices").
- **Tooling**: `npm run build` versus `npm run serve`; "Vue cannot be used through npm" (false); component advantages; virtual DOM.
- **State**: UI (ephemeral) state; HTTP is stateless so state is exchanged in requests.

WATCH OUT - REAL EXAM TRAP

- A new tab: `localStorage` yes, `sessionStorage` no.
- `getItem` of a missing key is `null`.
- Stored numbers come back as strings.
- Client-side storage never syncs across devices.

Quick quiz

QUICK QUIZ

Tab 1 runs `localStorage.setItem("theme","dark"); sessionStorage.setItem("cart","3-items");`. A fresh Tab 2 on the same site then reads both. What do they return?

- both values
- both null
- "dark" for theme, null for cart
- null for theme, "3-items" for cart

Check

`localStorage.setItem('c','0'); let c = localStorage.getItem('c'); c += 2; console.log(c)` prints...

2

02

0

NaN

Check

Which are advantages of client-side storage? (Select all) (select all that apply)

Storing personalised site preferences

IndexedDB can hold large files such as audio and video

Saving generated documents for offline use

The data is shared across all your devices

Check

Select all TRUE statements about SPAs and PWAs. (select all that apply)

SPAs often use client-side routing to update content without full reloads.

PWAs use service workers for offline use.

PWAs always need an internet connection.

SPAs load a whole new page from the server on each interaction.

Check

In a Vue CLI project, which command makes the production files in `dist` without starting a server?

`npm run serve`

`npm run build`

`npm start dev`

`npm install`

Check

In Vue 2, what happens when you change a data property?

- The page reloads
- Nothing until you refresh
- The virtual DOM finds the difference and updates only what changed
- The whole page is rebuilt from scratch

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- What is a single-page application and why did you choose one?
- What is client-side routing?
- Where is the login token stored in your project, and what are the risks?
- What is the difference between `localStorage` and `sessionStorage`?
- What does the virtual DOM do?
- What are `package.json` and `node_modules`?
- What is a PWA?

Recap

REMEMBER THIS

- MPA: new full page per click. SPA: one shell, JavaScript redraws, JSON from APIs. SPA is fast and light on the server, weaker for SEO.
- Client-side routing: hash mode (`/#/path`) needs no server setup; history mode needs a fallback to `index.html` .
- Cookies (4 KB, sent to server), `localStorage` (persistent, shared by tabs), `sessionStorage` (per tab), IndexedDB (large, async, files).
- Web Storage holds **strings**: `"0" + 2` is `"02"` ; missing key gives `null` .
- HTTP is stateless; state lives on the server, in memory, in storage or in the URL, and is sent explicitly.
- Frameworks keep UI in sync with state; the virtual DOM patches only differences; components give reuse.
- `package.json` + `npm install` fill `node_modules` ; `npm run serve` for development, `npm run build` for `dist/` ; bundlers (webpack, Vite) and Babel prepare the code.
- PWA = manifest + service worker + HTTPS. Responsive = viewport + media queries. Use dev tools (Network, Application).

CODE AND FACTS TO KEEP

- `localStorage.setItem(k, v)` , `getItem(k)` (null if missing), `removeItem(k)` , `clear()` ; `sessionStorage` has the same API
- `JSON.stringify(obj)` before storing, `JSON.parse(str)` after reading
- Sizes: cookie about 4 KB; web storage about 5 MB per origin
- `npm install` , `npm run serve` (dev, hot reload), `npm run build` (to `dist/`)
- `^2.6.14` = same major, newer minor and patch allowed
- `<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>` ; `new Vue({ el: "#app", data: {} })`
- `navigator.serviceWorker.register("/sw.js")` ; manifest = name, icons, start_url, display
- `<meta name="viewport" content="width=device-width, initial-scale=1">`

MAD2 week 4: Vue basics: instances, directives and reactivity

> **Official syllabus line:** Introduction to VueJS

The one-minute story

THINK OF IT LIKE THIS

Vue is a smart notice board. Imagine a notice board in your hostel. You write the notice once as a *template*: "Dinner today: _ at _ o'clock". The blanks are connected by invisible threads to a small notebook (your **data**). The moment someone changes a number in the notebook, the thread pulls and the board updates itself. You never walk to the board with a marker.

Plain JavaScript makes you do the walking: find the element, change its text, remember to change it again. Vue lets you **declare** what the page should look like for the current data, and it keeps the page in sync. That is called **declarative rendering** and **reactivity**.

A Vue instance also has a **life**: it is born, put on the wall, updated many times, and taken down. Vue lets you hook into each of those moments (the **lifecycle hooks**). Exam questions love asking "in which order do things happen".

By the end of this week you can:

- add Vue 2 to a page with one script tag and create an instance with `el`, `data`, `methods`, `computed` and `watch`;
- use `{{ }}` and the directives `v-bind`, `v-on`, `v-model`, `v-if`, `v-show` and `v-for` correctly;
- bind classes and styles in the object, array and ternary forms;
- choose between a computed property, a method and a watcher, and explain caching;
- predict what the template renders and what the console prints, including lifecycle hook order;
- explain reactivity caveats (adding properties, array index assignment) and fix them with `Vue.set`.

Words you need

WORDS YOU NEED

- **Vue** - a JavaScript framework for building user interfaces. This course uses Vue 2 (options API).
- **Instance** - the object you make with `new Vue({ … })`. It owns data, methods and one piece of the page.
- **Template** - the HTML with Vue extras (`{{ }}` and directives) that describes the page.
- **Reactive** - when data changes, the page changes automatically.
- **Interpolation** - printing a value into the page with `{{ expression }}`.
- **Directive** - a special attribute starting with `v-` that tells Vue to do something to an element.
- **Computed property** - a value worked out from other data, cached until those data change.
- **Watcher** - a function that runs whenever a chosen piece of data changes (for side effects).
- **Lifecycle hook** - a function Vue calls at a fixed moment: `created`, `mounted`, `updated`, `beforeDestroy` and so on.
- **Virtual DOM** - Vue's light copy of the page; it compares old and new copies and touches the real page only where needed.
- **Ref** - a name (`ref="box"`) that gives you direct access to a real element or child through `this.$refs`.
- **nextTick** - run code after Vue has finished updating the page.
- **CDN** - a public server that hosts library files, so one script tag is enough to load them.

Idea 1: What Vue is and how to add it

Vue is a **progressive** framework: you can use it on one corner of a normal HTML page, or build a whole application with it. In MAD2 you first use it the simple way, with a **CDN** script tag, and later (week 6) with the Vue CLI and `.vue` files.

```
<!DOCTYPE html>
<html>
  <body>
    <div id="app">
      <p>{{ message }}</p>
      <button @click="count++">Clicked {{ count }} times</button>
    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
    <script>
      new Vue({
        el: "#app",
        data: { message: "Hello Vue", count: 0 },
      });
    </script>
  </body>
</html>
```

How it works:

1. The browser loads Vue, which defines the global `Vue`.
2. `new Vue({ ... })` creates an instance and **mounts** it on the element matching `el` (`#app`).
3. Vue reads the HTML inside `#app` as the **template**, finds `{{ message }}` and `@click`, and renders.
4. When you click, `count++` changes data, and Vue updates only that text.

THE IDEA

The Vue instance options you must know

OPTION	WHAT IT HOLDS
`el`	Which existing element to take over, e.g. `"#app"` (alternative: `.$mount("#app")`)
`data`	The reactive state (an object on the root instance; a **function returning an object** in components)
`methods`	Functions you can call from the template or from other code
`computed`	Values derived from data, cached
`watch`	Functions that run when a data property changes
lifecycle hooks	`created`, `mounted`, ...

Inside any of these, `this` is the instance, so data is `this.message`, a method is `this.greet()`.

WATCH OUT - REAL EXAM TRAP

- Vue only controls the HTML **inside** `el`. A `{{ message }}` written outside `#app` is printed literally.
- Use arrow functions for callbacks **inside** methods, but do not write a method itself as an arrow function: `greet: () ⇒ this.message` has the wrong `this`. Write `greet() { ... }`.
- `el` can be omitted if you call `.$mount("#app")` later; the instance is then not on the page until you mount it.

IN YOUR PROJECT

Examiners ask "why did you pick Vue and not plain JavaScript?". Give a concrete reason from your project: "the dashboard list changes after every API call, and Vue re-renders it from the data, so I do not write `document.getElementById` code".

Idea 2: Interpolation and declarative rendering

`{{ expression }}` prints the value of any single **JavaScript expression**: a property, arithmetic, a ternary, a method call. It is **text** only, and it works only in the template (HTML text and attribute values never get `{{ }}`).

```
<div id="app">
  <p>{{ name }}</p>
  <p>{{ price * qty }}</p>
  <p>{{ qty > 5 ? "bulk" : "single" }}</p>
  <p>{{ name.toUpperCase() }}</p>
  <p>{{ items }}</p>
  <p v-text="name"></p>
  <p v-html="raw"></p>
</div>
<script>
  new Vue({
    el: "#app",
    data: { name: "pen", price: 10, qty: 3, items: [1, 2], raw: "<b>bold</b>" },
  });
</script>
```

Renders: `pen`, `30`, `single`, `PEN`, the JSON text of the array (arrays and objects are printed as JSON), `pen`, and **bold text** (`v-html` inserts real HTML).

> THE IDEA
>
> **Rules for `{{ }}`**
>
> - Only **expressions** (one value). A statement such as `{{ let x = 1 }}` or `{{ if (a) { } }}` is an error.
> - `undefined` or `null` prints as an empty string. `{{ user.name }}` when `user` is `null` throws an error, because you cannot read a property of `null` (more in Idea 6).
> - It escapes HTML, so it is safe. `v-html` does **not** escape; never use it on user-typed text (cross-site scripting).
> - `v-once` renders once and never updates.
> - Do not use `{{ }}` inside an attribute. Write `:title="name"`, not `title="{{ name }}"`.

YOUR TURN: 1. WHAT RENDERS

Data is `{ a: 4, b: "4", list: ["x", "y"], o: { k: 1 } }`. Write what each prints: `{{ a + b }}`, `{{ a * b }}`, `{{ list.length }}`, `{{ o.k + 1 }}`, `{{ list[1] }}`, `{{ o.z }}`.

▸ Need a hint?

▸ Show the full solution

Idea 3: Directives

A **directive** is an attribute starting with `v-`. Its value is a JavaScript expression (except `v-for` and `v-on`, which have their own forms).

v-bind (shorthand `:`) and v-on (shorthand `@`)

```
<img v-bind:src="photo" :alt="caption" :disabled="busy">
<button v-on:click="save" @click.prevent="save">Save</button>
```

- `:attr="expr"` sets an HTML attribute from data. If the expression is `false`, `null` or `undefined`, the attribute is **removed** (so `:disabled="false"` enables the button).
- `@event="handler"` listens to a DOM event. The handler can be a method name (`@click="save"`), a call with arguments (`@click="remove(item.id)"`) or a short statement (`@click="count++"`). Inside an inline handler, `$event` is the event.

Event modifiers (chain them after a dot):

MODIFIER	EFFECT
`.prevent`	calls `event.preventDefault()` (stops a form reloading the page, a link navigating)
`.stop`	calls `event.stopPropagation()` (the click does not bubble to the parent)
`.once`	the handler runs at most once
`.self`	only if the event target is the element itself
`.enter`, `.esc`, `.space`, `.tab`, `.up` …	**key modifiers**: only for that key, e.g. `@keyup.enter="send"`

v-model: two-way binding on form inputs

```
<input v-model="name">
<input v-model.trim="name">      <!— strips spaces at both ends —>
<input v-model.number="age">     <!— converts the text to a number —>
<input v-model.lazy="name">      <!— updates on change (leaving the box), not on every
key —>
<input type="checkbox" v-model="agree">             <!— true / false —>
<input type="checkbox" v-model="chosen" value="A">  <!— chosen is an array —>
<input type="radio" v-model="size" value="L">
<select v-model="city"><option>Chennai</option><option>Delhi</option></select>
<textarea v-model="notes"></textarea>
```

`v-model` is shorthand for two things: `:value="name"` (data to input) and `@input="name = $event.target.value"` (input to data). Typing updates `name` and changing `name` in code updates the box. I checked `v-model.number` and `.trim` in Vue 2: typing `42` stores the number `42`, typing `hi` stores `"hi"`.

WATCH OUT - REAL EXAM TRAP

- `v-model.number` is needed because input values are always **strings**. Without it, `age + 1` with age "20" gives `"201"`.
- `.prevent` on a form: `<form @submit.prevent="save">`. Without it the page reloads and your Vue state is lost.
- `.stop` stops **bubbling**; `.prevent` stops the **default action**. They are different.
- A key modifier is lower case and goes with a key event: `@keyup.enter`, not `@click.enter`.

Idea 4: Conditionals and lists

v-if, v-else-if, v-else and v-show

```
<p v-if="score ≥ 80">Distinction</p>
<p v-else-if="score ≥ 40">Pass</p>
<p v-else>Fail</p>
<p v-show="showHelp">Help text</p>
```

With `score = 55` Vue renders only `Pass`. The `v-else` and `v-else-if` element must come **immediately after** a `v-if` element.

THE IDEA

v-if versus v-show

	`V-IF`	`V-SHOW`
False case	element is **not in the DOM**	element is in the DOM with `display: none`
Cost	higher toggle cost (creates and destroys)	higher first-render cost, cheap toggle
Supports else	yes	no
Works on `template`	yes	no
Use when	condition rarely changes, or element must not exist (loading states)	you toggle often

A real exam option says " `v-show` and `v-if` cannot work together". That is false: you can use both (for example a loading message with `v-if` and a panel with `v-show`).

v-for: lists

```
<ul>
  <li v-for="(item, index) in items" :key="item.id">{{ index }}: {{ item.name }}</li>
</ul>
<span v-for="(value, key, i) in person">{{ i }}{{ key }}={{ value }}; </span>
<b v-for="n in 3">{{ n }}</b>
```

- Over an **array**: `(item, index) in items` .
- Over an **object**: `(value, key, index) in obj` (value first, then key).
- Over a **range**: `n in 3` gives `1 2 3` (starts at 1, not 0).
- `:key` must be a **unique, stable** id (not the index when the list is reordered or items are deleted). It lets Vue move the right element instead of re-using the wrong one.
- Never put `v-if` and `v-for` on the same element; wrap with `template v-for` or filter with a computed property.

Running this in Vue 2: `o = {a:1,b:2}` with `(v,k,i) in o` prints `0a=1;1b=2;` , and `n in 3` prints `123` .

SOLVED EXAMPLE: 1. A SMALL TO-DO LIST

```
<div id="app">
  <input v-model.trim="text" @keyup.enter="add" placeholder="new task">
  <button @click="add" :disabled="!text">Add</button>
  <ul>
    <li v-for="(t, i) in tasks" :key="t.id">
      <span :class="{ done: t.done }" @click="t.done = !t.done">{{ t.title }}
</span>
      <button @click="tasks.splice(i, 1)">x</button>
    </li>
  </ul>
  <p v-if="tasks.length === 0">Nothing to do.</p>
  <p v-else>{{ remaining }} left</p>
</div>
<script>
  new Vue({
    el: "#app",
    data: { text: "", nextId: 1, tasks: [] },
    computed: {
      remaining() { return this.tasks.filter(t => !t.done).length; },
    },
    methods: {
      add() {
        if (!this.text) return;
        this.tasks.push({ id: this.nextId++, title: this.text, done: false });
        this.text = "";
      },
    },
  });
</script>
```

Walk through: pressing Enter in the box or clicking Add runs `add`; `push` is a reactive array method, so the list re-renders; `text = ""` clears the box through `v-model`; `:disabled="!text"` is true while the box is empty; `remaining` recomputes when any `done` flag changes.

Idea 5: Class and style binding

`:class` and `:style` accept more than a string. Given the data:

```
data: {
  classObj: { is_active: true, completed: false },
  myClass: "is_active",
```

```
  status: true,
}
```

```
<div :class="classObj"></div>                          <!— class="is_active" —>
<div :class="[myClass, { completed: status }]"></div> <!— class="is_active completed" -
—>
<div class="base" :class="status ? 'on' : 'off'"></div> <!— class="base on" —>
<div :style="{ color: 'red', fontSize: '14px' }"></div>
```

I ran these in Vue 2 and they render exactly the comments shown.

THE IDEA

The three forms of `:class`

1. **Object**: `{ className: booleanExpression }` ; the class is present when the value is truthy.
2. **Array**: `[a, b]` ; every item (a string from data) is added. Objects can be mixed inside the array.
3. **Ternary** inside the array or alone: `status ? 'on' : 'off'` .

A static `class="base"` and a `:class` **merge**; the static one is kept. For `:style` use an object with camelCase (`fontSize`) or quoted kebab-case (`'font-size'`) keys.

WATCH OUT - REAL EXAM TRAP

A real question gives `classObj` and an array form and asks which statements are true. Read each carefully: an object entry adds the class only when its value is **true**; with `completed: false` the class `completed` is **not** present. Two different-looking bindings can produce the **same** rendered HTML.

Idea 6: Computed, methods and watch

Three ways to run logic. They look similar but are used for different jobs.

```
new Vue({
  el: "#app",
  data: { counter: 0, first: "Ada", last: "Lovelace" },
  computed: {
    fullName() { return this.first + " " + this.last; },     // cached
    double() { return this.counter * 2; },
  },
  methods: {
    stamp() { return Date.now(); },                           // runs every call
  },
```

```
  watch: {
    counter(newVal, oldVal) {                                  // side effect
      console.log("counter changed from " + oldVal + " to " + newVal);
    },
  },
});
```

	COMPUTED	METHOD	WATCH
Used as	a property: `{{ fullName }}`	a call: `{{ stamp() }}`	not called; runs by itself
Returns a value	yes	yes	no (do something)
Cached	**yes**, until a dependency changes	no, runs on every call/re-render	not applicable
Typical use	derived values, filtered lists	event handlers, actions	async calls, logging, reacting to a route or one property

Facts proved by running Vue 2:

- Reading a computed property three times with unchanged data runs its function **once**.
- Changing data twice in a row (`n = 2; n = 3`) calls the watcher **once**, with old value 1 and new value 3, because updates are batched.
- A computed property is **read-only** unless you give it a `get` and a `set`.
- A watcher can be written `{ handler, deep: true, immediate: true }` : `deep` watches nested changes, `immediate` also runs it once at start. To watch a nested key use a string: `'o.a'() { … }` (and `'$route.path'()` for the router).

SOLVED EXAMPLE: 2. PREDICT THE CONSOLE

```
new Vue({
  el: "#app",
  data: { counter: 0 },
  computed: { computedValue() { return this.counter * 2; } },
  watch: {
    counter(newVal, oldVal) { console.log(`Counter changed from ${oldVal} to
${newVal}`); },
  },
  mounted() { setInterval(() => { this.counter++; }, 1000); },
});
```

The watcher watches `counter` (not the computed value). After 1 second it prints `Counter changed from 0 to 1`, then `from 1 to 2`, then `from 2 to 3`. The page shows `Counter: n` and `Computed Value: 2n`. (A real paper offers "from 0 to 2" and "from 1 to 3 ... 6 ... 12" as wrong options: those would print the doubled value, which the watcher never sees.)

WATCH OUT - REAL EXAM TRAP

- Use **computed** when the result is a value for the template. Use **watch** when you must *do something* (call an API, save to storage) after a change. Using a watcher to calculate a display value is the classic wrong answer to "which is cached".
- A computed property must **return** something and must not change data.
- The question "use cases of watchers" has these true: react when a property changes; perform side effects. False: caching values (that is computed) and "creating reactive data" (that is `data`).

YOUR TURN: 2. PICK THE TOOL

For each job say computed, method or watch: (a) show `"first last"` from two fields; (b) button click that saves the form; (c) call the server whenever the search box text changes; (d) a filtered list shown in the page.

▸ **Need a hint?**

▸ **Show the full solution**

Idea 7: Reactivity and its caveats

When you create an instance, Vue 2 walks through `data` and converts every property into a **getter and setter** (using `Object.defineProperty`). The getter records "this render used me"; the setter tells Vue "re-

render". That only works for properties that **exist when the instance is created**.

Caveats (Vue 2 only) that I confirmed by running the code:

```
data: { arr: [1, 2, 3], o: { a: 1 } }

this.arr[0] = 99;          // data changes to [99,2,3] but the page does NOT update
this.arr.length = 1;       // data changes but the page does NOT update
this.o.b = 2;              // new property: o has b, but the page does NOT update

Vue.set(this.o, "b", 2);   // reactive new property (same as this.$set)
this.$set(this.arr, 0, 99);        // reactive index assignment
this.arr.splice(0, 1, 99);         // reactive too
this.o = { ...this.o, b: 2 };      // replace the whole object: reactive
```

- Array methods that Vue wraps and which **do** update the page: `push`, `pop`, `shift`, `unshift`, `splice`, `sort`, `reverse`.
- Declare every property you will use in `data` up front, even as `null` or `""`.
- Vue updates the page **asynchronously** in batches: many changes in one tick lead to one re-render. So right after `this.m = "b"` the element still shows the old text; use `this.$nextTick(() => { ... })` to run after the update.

> **THE IDEA**
>
> **Remember:** (1) properties must be declared in `data` to be reactive; (2) add new ones with `Vue.set` or by replacing the object; (3) change arrays with array methods or `Vue.set`, not `arr[i] = x`; (4) DOM updates are batched, so read the DOM inside `nextTick`.

```
this.message = "changed";
console.log(this.$refs.p.textContent);        // still the OLD text
this.$nextTick(() => console.log(this.$refs.p.textContent));  // the NEW text
```

Template refs. Put `ref="box"` on an element, and `this.$refs.box` is the real DOM element (after mount). It is the Vue way to call `focus()` on an input: `mounted() { this.$refs.box.focus(); }`. Refs exist only after mounting, and they are not reactive.

In newer Vue 3: reactivity uses `Proxy`, so adding properties and `arr[0] = x` **do** work, and `Vue.set` is gone. The instance is made with `Vue.createApp({ ... }).mount("#app")`, there is the Composition API (`ref`, `reactive`), and `beforeDestroy` / `destroyed` are renamed `beforeUnmount` / `unmounted`. This course tests Vue 2.

Idea 8: The lifecycle

Every instance goes through the same stages. Vue calls your hook at each.

```
new Vue()
  |
beforeCreate   (data and methods do NOT exist yet)
  |  (Vue sets up data, computed, methods, watchers)
created        (data and methods work; no DOM yet)
  |  (find the template, compile it)
beforeMount    (just before the first render)
  |  (render, create elements, insert in page)
mounted        (this.$el is in the page; refs work)
  |
  ├── data changes ──>  beforeUpdate  ->  (DOM re-rendered)  ->  updated   (repeats)
  |
  ├── vm.$destroy() / component removed
beforeDestroy  (still fully working: clean up here)
  |
destroyed      (watchers and listeners are removed)
```

HOOK	TYPICAL USE
`beforeCreate`	rarely used; `this.message` is `undefined`
`created`	start API calls to load data; set initial data
`beforeMount`	rarely used
`mounted`	work with the DOM or refs, start timers, third-party widgets
`beforeUpdate` / `updated`	react to a re-render (avoid changing data here, it can loop)
`beforeDestroy`	**clean up**: `clearInterval`, remove listeners
`destroyed`	final tidy-up (everything is already torn down)

SOLVED EXAMPLE: 3. HOOKS THAT CHANGE DATA (A REAL PAPER QUESTION)

```
new Vue({
  el: "#app",
  template: `<div>Status: {{message}} ETA: {{ETA}}</div>`,
  data: { message: "", ETA: 10 },
  beforeCreate() { this.message += "Creating,"; this.ETA -= 1; },
  created()      { this.message += "Created,";  this.ETA -= 1; },
  beforeMount()  { this.message += "Mounting,"; this.ETA -= 1; },
  mounted()      { this.message += "Mounted";   this.ETA -= 1; },
});
```

Output (checked in Vue 2): `Status: Created,Mounting,Mounted ETA: 7`.

Why? In `beforeCreate` the data does not exist yet, so the two lines touch nothing useful (they set plain properties that Vue then overwrites when it initialises `data`). So `message` starts as `""` and `ETA` as `10`. Then `created` adds `Created,` and makes ETA 9, `beforeMount` gives `Mounting,` and 8, and `mounted` gives `Mounted` and 7. The page shows the **final** values because the update after `mounted` is rendered in the next tick.

SOLVED EXAMPLE: 4. CREATED VERSUS MOUNTED, WITH A DELAYED UPDATE

```
new Vue({
  el: "#app",
  data() { return { message: "Initial" }; },
  created() { console.log("Created Hook"); this.message = "Created Hook Called";
this.fetchData(); },
  mounted() { console.log("Mounted Hook"); },
  methods: {
    async fetchData() {
      const res = await new Promise(resolve => setTimeout(() => resolve("Data
Loaded"), 1000));
      this.message = res;
      console.log("Data updated after fetch");
    },
  },
});
```

Console: `Created Hook`, `Mounted Hook`, **(one second later)** `Data updated after fetch`. **Page after 1 second:** `Data Loaded`. `created` runs, starts the async work and returns at once (the `await` only pauses `fetchData`), then Vue mounts and prints `Mounted Hook`. The timer finishes later. I verified this order by running it.

SOLVED EXAMPLE: 5. WHEN DOES BEFOREUPDATE RUN?

```
<div id="app">
  <p>{{ message }}</p>
  <button @click="changeMessage">Change Message</button>
  <button @click="changeCount">Change Count</button>
</div>
```

With `data: { message: "Hello, Vue!", count: 0 }`, `changeCount() { this.count++; }` and `beforeUpdate() { console.log("beforeUpdate called"); }`: clicking **Change Count prints nothing**, because `count` is not used in the template, so nothing needs re-rendering. Clicking **Change Message** prints `beforeUpdate called`. An update hook fires only when data **used by the render** changes. (I tested both.)

SOLVED EXAMPLE: 6. BEFORECREATE WITH DATA

```
new Vue({
  el: "#app",
  data() { return { message: "Hello" }; },
  beforeCreate() { console.log("Before Create: " + this.message); this.message =
"Before Create"; },
  mounted() { this.message = "Mounted"; console.log("Mounted: " + this.message);
},
  template: `<div>{{ message }}</div>`,
});
```

Console: `Before Create: undefined` then `Mounted: Mounted`. The first value is `undefined` because data is not set up yet; the assignment is then overwritten when Vue initialises `data`.

WATCH OUT - REAL EXAM TRAP

- `beforeCreate`: data and methods are **not** available. `created`: they are, but `this.$el` and the DOM are not.
- `mounted` does not guarantee that **child components** or fetched data are ready.
- `beforeDestroy` is where cleanup goes: a timer started in `mounted` must be cleared in `beforeDestroy`, otherwise it keeps running. `destroyed` is after teardown, **not** the place to declare data.
- Parent and child hook order (week 6): parent `created` first, but the **child's `mounted` runs before the parent's `mounted`**.

YOUR TURN: 3. ORDER THE HOOKS

A component has `created`, `mounted`, `updated` and `beforeDestroy`, each logging its name. It loads, the user clicks a button that changes a displayed value, then the component is removed. Write the console output.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 4. REACTIVITY FIX

`data: { cart: {} }` and a click does `this.cart[item] = 1` but the page does not change. Give two fixes.

▸ **Need a hint?**

▸ **Show the full solution**

Idea 9: Templates in HTML and in strings

Two ways to give an instance a template:

```
<!— 1. In the page: Vue reads the HTML inside #app (in-DOM template) —>
<div id="app"><p>{{ message }}</p></div>
<script>new Vue({ el: "#app", data: { message: "Hi" } });</script>

<!— 2. As a string: the template option replaces what is inside #app —>
<div id="app"></div>
<script>
  new Vue({ el: "#app", data: { message: "Hi" }, template: "<p>{{ message }}</p>" });
</script>
```

The `template` option needs the **full build** of Vue (the CDN default, which includes the compiler). The `.vue` files of week 6 are compiled in advance, so they need no compiler at run time. One extra rule in in-DOM templates: HTML is case-insensitive, so write `<my-comp>` (kebab-case) and `player-name="…"`, not `<MyComp>` or `playerName`.

Exam radar

EXAM RADAR

What the real papers test. The evidence file counts questions that match this week's keywords: about **258 of 848 Quiz 1**, **181 of 608 Quiz 2** and **300 of 1471 End Term** questions matched. That is a loose keyword match (Vue words also catch router, component and API questions), but about 180 distinct questions were found, and the Vue ones are the largest block in the MAD2 papers. Recurring shapes:

- **Lifecycle prediction (very common).** Hooks that modify `message` and `ETA` (answer: `Created, Mounting, Mounted` and ETA 7), `beforeCreate` printing `undefined`, `created` and `mounted` with an async fetch, "will `beforeUpdate` run?" when the changed value is not in the template.
- **Watch versus computed.** A watcher on `counter` logging old and new values; "which statements about watchers are true".
- **Class binding.** Object, array and mixed forms; which statements are true.
- **Template rendering with components and slots** (see week 6): global versus local registration, default props, what happens to content placed inside a component that has no slot.
- **v-if and v-show** with a loading message that disappears after a click or a timer.
- **True/false statements** on `.vue` files (they contain template, script and style, and must be compiled by Vue CLI or Vite; they cannot be loaded straight in a script tag).

WATCH OUT - REAL EXAM TRAP

- Predicting the console: write the hook order first, then apply each statement. Do not forget that `beforeCreate` sees no data.
- "Will it re-render?" Only data the **template reads** triggers `beforeUpdate` and `updated`.
- `{{ }}` renders `undefined` as empty, but `{{ obj.a.b }}` with `obj.a` undefined is an **error**.
- Watchers see raw data; they do not see computed values changing.
- Two bindings can look different but give the same HTML: always evaluate each to its final class string.

Quick quiz

QUICK QUIZ

A Vue 2 instance has `data: { message: "", ETA: 10 }`. Each of `beforeCreate`, `created`, `beforeMount` and `mounted` appends text to `message` (`"Creating,"`, `"Created,"`, `"Mounting,"`, `"Mounted"`) and subtracts 1 from `ETA`. The template shows both. What is rendered?

Status: Creating,Created,Mounting,Mounted ETA: 6

Status: Created,Mounting,Mounted ETA: 7

Status: Mounting,Mounted ETA: 8

Status: Mounted ETA: 9

Check

What does `console.log(this.message)` print inside `beforeCreate`, when `message` is declared in `data`?

the initial value

undefined

null

an error is thrown

Check

A button runs `this.count++`, `count` is declared in `data` but is not used in the template, and the instance has a `beforeUpdate` hook that logs. What happens on click?

beforeUpdate is logged

nothing is logged

an error is thrown

the whole page reloads

Check

Select all TRUE statements about Vue 2. (select all that apply)

A watcher is used to run side effects when a property changes.

A computed property is recomputed on every access.

`v-show` keeps the element in the DOM and toggles `display`.

`this.arr[0] = 5` is a reactive update.

Check

With `data: { classObj: { is_active: true, completed: false }, status: true }`, what class does `<div :class="[classObj.is_active ? 'is_active' : '', { completed: status }]">` render?

- is_active only
- completed only
- is_active completed
- no class

Check

You must call the server every time a search box value changes. The best Vue feature is...

- computed property
- v-show
- a watcher on the search text
- a static method called in the template

Check

Which hook is the right place to stop a `setInterval` started in `mounted`?

- created
- mounted
- beforeDestroy
- beforeMount

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- What is Vue and what does reactive mean?
- Explain the difference between computed, methods and watch.
- v-if or v-show: which did you use and why?
- Why do we need a key in v-for?
- Walk me through the Vue lifecycle. Where do you call your API?
- I add a new field to an object in your data and the page does not update. Why?
- How does v-model work?
- What is nextTick for?

Recap

REMEMBER THIS

- Vue renders a template from data. Add it with a CDN script and `new Vue({ el, data, methods, computed, watch })`.
- `{{ }}` prints one expression as text; `v-bind` (`:`) sets attributes; `v-on` (`@`) handles events with `.prevent`, `.stop`, `.once` and key modifiers; `v-model` is two-way with `.trim`, `.number`, `.lazy`.
- `v-if` adds or removes the element; `v-show` only hides it. `v-for` needs a unique `:key`; over an object the order is `(value, key, index)`.
- `:class` and `:style` take objects, arrays and ternaries.
- Computed is cached and returns a value; methods run each time; watch performs side effects.
- Vue 2 reactivity: declare data up front, use `Vue.set` for new properties and index changes, use array methods. Updates are batched, so use `nextTick`.
- Lifecycle: beforeCreate, created, beforeMount, mounted, beforeUpdate, updated, beforeDestroy, destroyed. `created` has data but no DOM; `mounted` has the DOM; `beforeDestroy` is for cleanup.

CODE AND FACTS TO KEEP

- CDN: `<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>`
- Create: `new Vue({ el: "#app", data: { ... }, methods: { ... }, computed: { ... }, watch: { ... } })`
- `:attr="x"` is `v-bind:attr`; `@click="f"` is `v-on:click`; `@submit.prevent`, `@click.stop`, `@click.once`, `@keyup.enter`
- `v-model.trim`, `v-model.number`, `v-model.lazy`
- `v-for="(item, i) in list"`, `(value, key, i) in obj`, `n in 5`; always `:key`
- `:class="{ active: isOn }"`, `:class="[a, { b: flag }]"`, `:style="{ color: c }"`
- Caveat fixes: `Vue.set(obj, key, val)`, `this.$set(arr, i, val)`, `arr.splice(i, 1, val)`
- `this.$nextTick(fn)`, `this.$refs.name`
- Hook order: beforeCreate, created, beforeMount, mounted, (beforeUpdate, updated)*, beforeDestroy, destroyed
- Vue 3: `createApp`, Proxy reactivity, `beforeUnmount` and `unmounted`

MAD2 week 5: Vue with APIs: fetch, axios and lifecycle

> **Official syllabus line:** Vue with APIs

The one-minute story

THINK OF IT LIKE THIS

A Vue page talking to Flask is a customer ordering at a restaurant. The Vue page is the customer at the table. The Flask API is the kitchen. The customer cannot walk into the kitchen: he writes an order slip (a **request**: what he wants, in what form), hands it to the waiter (**fetch** or **axios**), and waits. Meanwhile he does not freeze: he sips water (the page keeps working and shows "Loading...").

The kitchen sends back a dish and a note: "200 here it is" or "404 we do not have that" or "500 the chef set the stove on fire". A careful customer reads the note before he eats. Finally there is the building's **rule about outsiders**: a kitchen may refuse to serve people who came from another building. That is **CORS**, and the kitchen's manager (Flask-CORS) must put your building on the allowed list.

By the end of this week you can:

- call a Flask API from Vue with `fetch` and with `axios`, inside `created` or `mounted`, and with `async` / `await`;
- show a loading state and an error state, and handle a failed request correctly;

- build a full CRUD page (list, create, edit, delete) against a Flask REST backend;
- send JSON with the right header and read status codes;
- explain what CORS is, why the browser blocks a request, and enable it with Flask-CORS or avoid it with a dev proxy;
- predict the console output of `async` / `await` code with success, HTTP error and network error.

Words you need

WORDS YOU NEED

- **API** - the URLs a server offers, which a program (our Vue page) can call instead of a person.
- **Request / response** - what the client sends, and what the server sends back (a status code, headers and a body).
- **fetch** - the browser's built-in function to make HTTP requests; it returns a **Promise**.
- **Promise** - a value that will arrive later; it ends as *resolved* (success) or *rejected* (failure).
- **async / await** - syntax to write promise code like normal top-to-bottom code. `await` pauses only that function.
- **axios** - a popular library (not built in) that wraps requests and turns JSON automatically.
- **Status code** - the number that says how it went: 200 OK, 201 Created, 204 No Content, 400 Bad Request, 401 Unauthorized, 404 Not Found, 500 Server Error.
- **Origin** - scheme + host + port, e.g. `http://localhost:8080` .
- **CORS** - Cross-Origin Resource Sharing: the way a server tells the browser which other origins may read its responses.
- **Preflight** - an automatic `OPTIONS` request the browser sends before a "non-simple" request (JSON body, custom header, PUT/DELETE).
- **Proxy** - the dev server forwards `/api/...` requests to Flask, so the browser thinks everything is one origin.
- **Dev tools Network tab** - the browser panel that lists every request with status, headers, payload and timing.
- **CRUD** - Create, Read, Update, Delete.

Idea 1: Calling an API with fetch

`fetch(url, options)` returns a Promise of a **Response** object. The response is **not** your data: you must ask for the body, which is another promise.

```
fetch("http://127.0.0.1:5000/api/tasks")
  .then((res) ⇒ res.json())          // body → JavaScript value (also a promise)
  .then((data) ⇒ console.log(data))  // use the data
  .catch((err) ⇒ console.error(err)); // network failure or a thrown error

console.log("End of script");
```

Output order: `End of script` first, then the data. `fetch` never blocks: JavaScript carries on with the next line and runs the `.then` code later, when the response arrives. (A real paper asks exactly this: the answer is "End of script, then the JSON".)

> THE IDEA
>
> **What `fetch` does and does not do**
>
> - It **rejects** (goes to `.catch`) only when the request could not be completed at all: server down, no internet, DNS failure, **or a CORS block**. The error is a `TypeError`.
> - It does **not** reject for HTTP errors. A `404` or `500` is a *successful* round trip, so `.then` runs. You must check `res.ok` (true for status 200 to 299) or `res.status`.
> - `res.json()` rejects with a `SyntaxError` if the body is not valid JSON (for example HTML).
> - A `res` object has: `ok`, `status`, `statusText`, `headers`, and body readers `json()`, `text()`, `blob()`.

The same with async and await

```
async function getTasks() {
  const res = await fetch("http://127.0.0.1:5000/api/tasks");
  if (!res.ok) throw new Error("HTTP " + res.status);
  return await res.json();
}

getTasks().then((d) ⇒ console.log(d)).catch((e) ⇒ console.log(e.message));
```

- `await` can only be used inside an `async` function. It gives the **resolved value**, not the promise.
- An `async` function **always returns a promise**. A `throw` inside it makes that promise reject.
- If you forget `await`, `res` is a Promise: `res.ok` is `undefined` and `res.json` is not a function on the promise. A real question presents code like this and asks "what is the error".

SOLVED EXAMPLE: 1. PREDICT THE CONSOLE (FOUR CASES)

```
async function getResponse(url) {
  try {
    const response = await fetch(url);
    try {
      const data = await response.json();
      if (response.ok) { return data; } else { return response.status; }
    } catch {
      return "Network Error";
    }
  } catch {
    return "Response is not json";
  }
}
getResponse("url").then((data) ⇒ console.log(data));
```

The labels are mixed up on purpose; read the structure, not the words. The **outer** `try` wraps only the `fetch` call; the **inner** `try` wraps `response.json()` . I ran this against a small test server:

WHAT THE URL DOES	WHERE IT FAILS	PRINTED
network error (server unreachable)	`fetch` rejects, outer `catch`	`Response is not json`
500 with a JSON body	json works, `ok` is false	`500`
404 with a JSON body	same	`404`
200 with an HTML body	`json()` rejects, inner `catch`	`Network Error`

Second variant (also a real paper): the same try inside try, but an `if (!res.ok) throw new Error( … )` sits inside the outer `try` and the outer `catch` always throws `new Error("Data is not JSON serializable")` . With a URL that returns **valid HTML**, `res.ok` is true, `json()` fails, the inner catch throws `Error("Error")` , and the **outer catch** catches that and throws "Data is not JSON serializable". With a 404 you get the same message, because the thrown HTTP error is also caught by the outer catch. The lesson: trace which `try` each statement is inside.

SOLVED EXAMPLE: 2. THE CLASSIC NETWORK-VS-HTTP ERROR CHECK

```
async function fetchData() {
  try {
    const response = await fetch("https://api.example.com/data");
    if (!response.ok) throw new Error("HTTP error");
    return await response.json();
  } catch (error) {
    if (error.name === "TypeError") return { status: "network error" };
    return { status: "http error" };
  }
}
```

If the server is **unreachable**, `fetch` rejects with a `TypeError`, so the function returns `{ status: "network error" }`. If the server answers 500, the thrown `Error` has name `Error`, so you get `{ status: "http error" }`.

WATCH OUT - REAL EXAM TRAP

- `fetch` does not reject on 404 or 500. Always check `res.ok`.
- `res.json()` is a promise: write `await res.json()` or another `.then`.
- `console.log(fetch(url))` prints `Promise {<pending>}`, not data.
- Code after a `fetch( … )` call runs **before** the response arrives (the "End of script comes first" question).

Idea 2: Calling the API from a Vue component

Where to start the request? In `created` (data and methods exist, request starts early) or `mounted` (the page exists). Both are fine for plain API calls; the course examples use either. Then **store the answer in `data`** and let Vue re-render.

```
<div id="app">
  <p v-if="loading">Loading … </p>
  <p v-else-if="error" class="err">{{ error }}</p>
  <ul v-else>
    <li v-for="t in tasks" :key="t.id">{{ t.title }}</li>
  </ul>
</div>
<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
<script>
  new Vue({
    el: "#app",
```

```
    data: { tasks: [], loading: false, error: "" },
    created() { this.loadTasks(); },
    methods: {
      async loadTasks() {
        this.loading = true;
        this.error = "";
        try {
          const res = await fetch("http://127.0.0.1:5000/api/tasks");
          if (!res.ok) throw new Error("HTTP " + res.status);
          this.tasks = await res.json();
        } catch (e) {
          this.error = "Could not load tasks: " + e.message;
        } finally {
          this.loading = false;
        }
      },
    },
  });
</script>
```

Order of events (verified by running a similar component): the page renders at once with `tasks: []` and `loading: false`; `created` calls `loadTasks`, which sets `loading = true` and **returns at the first `await`**; Vue mounts (so `mounted` runs **before** the data arrives); when the response comes, `tasks` is filled, `finally` sets `loading = false`, and Vue re-renders the list. `finally` runs on both success and failure, so the "Loading..." message never gets stuck.

THE IDEA

The three states of every request: loading, error, data. Your data should hold `loading`, `error` and the result. Show them with `v-if / v-else-if / v-else`. Start with an **empty but valid** value (`[]` for lists, `{}` or `null` for one record).

WATCH OUT - REAL EXAM TRAP

Initial value of a single record. With `data() { return { user: {} } }` and the template `Hi {{ user.name }}`, the page shows `Hi` (empty) until the data arrive, with no error. With `user: null` the same template **throws** "Cannot read properties of null" on first render (I ran both). Fix: use `{}`, or guard with `v-if="user"`.

A real question shows exactly this: `user: {}`, `created()` calling an async `fetchUserData` that reads `this.$route.query.userId` and fetches `/api/users/123`. Answer: the component displays the name correctly once the response arrives. The first render has no name, which is fine.

Also: a slow API call does **not** block the page. Everything else (a sidebar in a parent component, for example) renders immediately; only the part that needs the data fills in later.

The same with axios

axios is an installed package (`npm install axios`) or a CDN script. Compared with fetch:

```
import axios from "axios";   // with the CDN script it is the global `axios`

const res = await axios.get("http://127.0.0.1:5000/api/tasks");
console.log(res.data);                    // already parsed JSON, no .json() step

const created = await axios.post("http://127.0.0.1:5000/api/tasks", { title: "Buy milk"
});
// the object is sent as JSON with Content-Type: application/json automatically
console.log(created.status, created.data);

try {
  await axios.get("http://127.0.0.1:5000/api/tasks/999");
} catch (err) {
  if (err.response) console.log(err.response.status);   // server answered 4xx/5xx
  else console.log("network or CORS problem", err.message);
}
```

	`FETCH`	`AXIOS`
Installation	built into browsers	`npm install axios`
JSON body	`JSON.stringify` + header yourself	pass an object, it does both
Read JSON	`await res.json()`	`res.data`
HTTP 404 / 500	**resolves**; check `res.ok`	**rejects**; use `err.response.status`
Network error	rejects (`TypeError`)	rejects; `err.response` is undefined

Idea 3: Sending data: JSON, headers and status codes

A `POST` that sends JSON needs three things: the method, the header and the stringified body.

```
const data = { name: "Aditya", score: 95 };

fetch("/api/submit", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify(data),
});
```

- `Content-Type: application/json` tells Flask that the body is JSON, so `request.get_json()` works. Without it `fetch` labels a string body as `text/plain`, and recent Flask rejects it (415 Unsupported Media Type).
- A real question asks which header is necessary: `'Content-Type': 'application/json'` (not `multipart/form-data`, not `application/text`, and `Accept: */*` is not what the server needs).
- `Accept` tells the server which type you want back; `Authorization` carries a token (see week 7). A `fetch` can add any custom header, send images (as `FormData` or `Blob`) and use any method.
- For a file upload with `FormData`, do **not** set `Content-Type` yourself; the browser adds the boundary.

Status codes you should react to:

CODE	MEANING	WHAT THE VUE CODE DOES
200	OK	show data
201	Created	add the new item to the list
204	No Content	success with empty body (DELETE): **do not call `res.json()`**
400	Bad Request	show the validation message
401	Unauthorized (not logged in / bad token)	send the user to the login page
403	Forbidden (logged in, not allowed)	show "not allowed"
404	Not Found	show "not found"
409	Conflict (duplicate)	show the message
500	Server error	show a general error

A real question: a `fetch("http://127.0.0.1:5000/protected", { method: "GET" })` with no token to a protected route. Answer: the request fails with **401 Unauthorized** (`fetch` itself still resolves; the response has `status` 401).

Idea 4: A full CRUD page against Flask

Backend (Flask 3, plain routes, in-memory list so it is easy to read). I tested every route with Flask's test client: create returns 201, a missing title returns 400, an unknown id returns 404 and delete returns 204.

```
from flask import Flask, jsonify, request
from flask_cors import CORS

app = Flask(__name__)
```

```
CORS(app, resources={r"/api/*": {"origins": "http://localhost:8080"}})

tasks = [{"id": 1, "title": "Learn Vue", "done": False}]
next_id = 2


@app.get("/api/tasks")
def list_tasks():
    return jsonify(tasks)


@app.post("/api/tasks")
def create_task():
    global next_id
    data = request.get_json(silent=True)
    if not data or not data.get("title"):
        return jsonify({"error": "title is required"}), 400
    task = {"id": next_id, "title": data["title"], "done": False}
    next_id += 1
    tasks.append(task)
    return jsonify(task), 201


@app.put("/api/tasks/<int:task_id>")
def update_task(task_id):
    task = next((t for t in tasks if t["id"] == task_id), None)
    if task is None:
        return jsonify({"error": "not found"}), 404
    data = request.get_json(silent=True) or {}
    task["title"] = data.get("title", task["title"])
    task["done"] = data.get("done", task["done"])
    return jsonify(task)


@app.delete("/api/tasks/<int:task_id>")
def delete_task(task_id):
    global tasks
    before = len(tasks)
    tasks = [t for t in tasks if t["id"] != task_id]
    if len(tasks) == before:
        return jsonify({"error": "not found"}), 404
    return "", 204
```

Frontend (Vue 2 from a CDN). One helper `request` handles the checks for every call, so each method stays short. I ran this code against a fake server in a browser-like environment and checked list, add,

empty-title error, toggle, delete and network failure.

```
<div id="app">
  <p v-if="loading">Loading … </p>
  <p v-if="error" class="err">{{ error }}</p>
  <form @submit.prevent="addTask">
    <input v-model.trim="title" placeholder="New task">
    <button :disabled="!title">Add</button>
  </form>
  <ul>
    <li v-for="t in tasks" :key="t.id">
      <span :class="{ done: t.done }" @click="toggle(t)">{{ t.title }}</span>
      <button @click="removeTask(t)">Delete</button>
    </li>
  </ul>
</div>
```

```
const API = "http://127.0.0.1:5000/api/tasks";
new Vue({
  el: "#app",
  data: { tasks: [], title: "", loading: false, error: "" },
  created() { this.loadTasks(); },
  methods: {
    async request(url, options) {
      this.error = "";
      const res = await fetch(url, options);
      if (!res.ok) {
        let msg = "HTTP " + res.status;
        try { msg = (await res.json()).error || msg; } catch (e) {}
        throw new Error(msg);
      }
      return res.status === 204 ? null : res.json();
    },
    async loadTasks() {
      this.loading = true;
      try { this.tasks = await this.request(API); }
      catch (e) { this.error = e.message; }
      finally { this.loading = false; }
    },
    async addTask() {                                  // Create
      try {
        const task = await this.request(API, {
          method: "POST",
          headers: { "Content-Type": "application/json" },
```

```
          body: JSON.stringify({ title: this.title }),
        });
        this.tasks.push(task);
        this.title = "";
      } catch (e) { this.error = e.message; }
    },
    async toggle(task) {                                  // Update
      try {
        const updated = await this.request(API + "/" + task.id, {
          method: "PUT",
          headers: { "Content-Type": "application/json" },
          body: JSON.stringify({ done: !task.done }),
        });
        task.done = updated.done;
      } catch (e) { this.error = e.message; }
    },
    async removeTask(task) {                              // Delete
      try {
        await this.request(API + "/" + task.id, { method: "DELETE" });
        this.tasks = this.tasks.filter((t) => t.id !== task.id);
      } catch (e) { this.error = e.message; }
    },
  },
});
```

How the parts fit:

1. **Read:** `created` runs `loadTasks`, filling `tasks`.
2. **Create:** the form's `@submit.prevent` calls `addTask`; the body is JSON with a header; Flask answers `201` and the new task including its id; we `push` it (a reactive method).
3. **Update:** `PUT` sends only the changed field. We copy the server's answer back (`task.done` already exists, so this is reactive).
4. **Delete:** `204` has no body, so the helper returns `null` instead of calling `res.json()`; we remove it with `filter`.
5. Errors from any call end in `this.error`, which the template shows.

YOUR TURN: 1. EDIT IN PLACE

Add an "Edit" feature: clicking Edit turns the title into an input; pressing Enter sends a `PUT` with the new title. What data do you add and which request?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 2. WHICH STATUS CODES

For the Flask API above, what status does each give? (a) POST with `{}`; (b) DELETE of id 999; (c) POST with `{"title": "x"}`; (d) DELETE of an existing id.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 3. FIX THE BUG

A student writes `async loadTasks() { const res = fetch(API); this.tasks = res.json(); }` and the list stays empty with an error in the console. List the problems.

▸ **Need a hint?**

▸ **Show the full solution**

IN YOUR PROJECT

When the examiner says "show me an API failing", stop Flask and reload the page. Your UI must show a clear error message (not a blank page or a stuck "Loading..."). Also show a 400 from a bad form. Be ready to explain where you put the `try/catch/finally` and why `loading` goes back to `false`.

Idea 5: CORS: why the browser blocks and how to fix it

The browser has a safety rule, the **same-origin policy**: JavaScript from one origin may not **read** responses from another origin, unless that server says it is allowed. An **origin** is **scheme + host + port**; all three must match.

Your Vue dev page is at `http://localhost:8080`. Flask is at `http://127.0.0.1:5000`. Those are different origins (different host and port), so a `fetch` from the page to Flask is a **cross-origin** request, and without CORS headers the browser blocks the response and prints a CORS error in the console.

Origin comparison (a recurring exam question), page loaded from `https://abc.com`:

TARGET	SAME ORIGIN?	CORS ERROR BY DEFAULT?
`https://abc.com`	yes	no
`https://abc.com/page`	yes (the path does not matter)	no

TARGET	SAME ORIGIN?	CORS ERROR BY DEFAULT?
`https://xyz.abc.com`	**no** (different host: a subdomain is another origin)	**yes**
`https://abc8.com`	no (different host)	**yes**
`http://abc.com`	no (different scheme)	yes
`https://abc.com:8443`	no (different port)	yes

THE IDEA

How CORS works

- It is enforced by the **browser**, not the server. A request from `curl` or Postman is never blocked.
- The **server** opts in by sending a header: `Access-Control-Allow-Origin: http://localhost:8080` (or `*` for any origin). CORS lets the server say **which origins may load its resources**; it is not the client that chooses.
- **Same-origin requests need no CORS.**
- For "non-simple" requests (methods PUT/DELETE, a `Content-Type: application/json` body, or an `Authorization` header) the browser first sends a **preflight** `OPTIONS` request. The server must reply with `Access-Control-Allow-Methods` and `Access-Control-Allow-Headers`; only then is the real request sent.
- A blocked request often **still reaches the server and runs**; the browser just refuses to give your JavaScript the answer. That is why CORS is not a protection against someone who calls your API directly.

Enabling CORS in Flask

```
from flask import Flask
from flask_cors import CORS

app = Flask(__name__)
CORS(app)                                    # allow every origin (fine for local
learning)
# or restrict:
# CORS(app, resources={r"/api/*": {"origins": "http://localhost:8080"}})
```

I tested the restricted version: a request with `Origin: http://localhost:8080` gets `Access-Control-Allow-Origin: http://localhost:8080`; a request with `Origin: http://localhost:5173` gets **no** such header (status 200, but the browser blocks it); a preflight `OPTIONS` gets the allowed methods and headers.

SOLVED EXAMPLE: 3. WILL THE REQUEST SUCCEED?

```
CORS(app, resources={r"/api/*": {"origins": "http://localhost:3000"}})

@app.route("/api/data")
def data():
    return {"msg": "Success"}
```

```
fetch("http://localhost:5000/api/data").then(r ⇒ r.json()).then(console.log);
```

The Vue app runs at `http://localhost:5173`. Only `http://localhost:3000` is allowed, so Flask sends no allow header for origin 5173. **The browser shows a CORS error** (it is not a 404 and the server does not crash; the server did answer). This is a real paper question.

A real alternative sets the header by hand: `return "Endpoint 1 Accessed", 200, {"Access-Control-Allow-Origin": "*"}`. That is the same thing without the extension.

Other ways to avoid CORS

1. **Dev proxy.** The Vue dev server forwards some paths to Flask, so the page calls its own origin:

```
// vue.config.js (Vue CLI)
module.exports = {
  devServer: {
    proxy: { "/api": { target: "http://localhost:5000", changeOrigin: true } },
  },
};
```

Now the page at `http://localhost:8080` calls `fetch("/api/tasks")` (a relative URL). The dev server passes it to Flask. The browser sees one origin, so no CORS. (With Vite the same idea is `server.proxy` in `vite.config.js`.) The proxy is for **development only**.

1. **Serve both from one origin** (next idea). In production the built Vue files and the API come from Flask, so there is no cross-origin call.

Idea 6: Putting a built Vue app inside Flask (outline)

`npm run build` (Vue CLI) produces a `dist/` folder: `index.html`, plus `js/` and `css/` with minified files. These are plain static files. Flask can serve them:

```
from flask import Flask, send_from_directory
```

```
app = Flask(__name__, static_folder="dist", static_url_path="")

@app.get("/")
def index():
    return send_from_directory(app.static_folder, "index.html")

# the API lives under /api/... , so the same origin serves page and data
```

I checked that `/` returns the built `index.html`. Steps in practice:

1. In the Vue project run `npm run build`.
2. Copy or point to `dist/` from your Flask folder (or set the build output there).
3. Use **relative** API URLs: `fetch("/api/tasks")` works in dev (through the proxy) and in production (same origin).
4. If you use the Vue Router in `history` mode, add a Flask catch-all that returns `index.html` for unknown non-API paths, otherwise a page refresh on `/about` gives Flask's 404 (week 6).

Idea 7: The response lifecycle and debugging

What happens when a user clicks "Load":

```
click (@click)  →  method runs  →  loading = true  →  fetch() sends the request
      |                                                        |
Vue re-renders "Loading..."                          Flask route runs, returns JSON +
status
      |                                                        |
   response arrives (browser checks CORS) →  res.ok? → res.json() → this.tasks = data
      |
finally: loading = false  →  Vue re-renders the list
```

Debugging with the browser Network tab (F12, then Network):

- Each request is a row: **name, status, type, size, time**. Red means failure.
- Click a row: **Headers** (request URL, method, status, request and response headers, including `Access-Control-Allow-Origin`), **Payload** (what you sent), **Preview / Response** (what came back), **Timing**.
- A CORS failure shows a message in the **Console** ("blocked by CORS policy"), and the row shows a CORS error or no response.
- Typical diagnosis: status **404** = wrong URL; **405** = wrong method; **400** = bad body or missing `Content-Type`; **415** = Flask did not get JSON; **500** = look at the Flask terminal; **no row at all** = the code never ran or Flask is not running.
- Add `console.log(res.status)` and `console.log(data)` while learning. Use the **Vue Devtools** extension to see `data` change.

Timing questions (very common in the papers)

```
@app.route("/endpoint1")
def method1():
    time.sleep(20)
    return "Endpoint 1 Accessed", 200, {"Access-Control-Allow-Origin": "*"}
@app.route("/endpoint2")
def method2():
    time.sleep(30)
    return "Endpoint 2 Accessed", 200, {"Access-Control-Allow-Origin": "*"}

app.run(threaded=False)
```

A page issues `fetch` to both URLs one after the other. The browser sends both at once, but a server started with `threaded=False` handles **one request at a time**, so total time until both are done is 20 + 30 = **50 seconds**. With a threaded server (the default for `flask run` and `app.run()` in Flask 1.0 and later) they run in parallel and take about the larger one, **30 seconds**. A variant: two routes sleeping 10 and 30 seconds with `app.run(debug=True)` (threaded), the page opened at 10:10:10, each route returning `datetime.now().second`. The logs show 20 (10 + 10) and 40 (10 + 30). The page itself loads at once in all cases (answer "0 seconds" for "when does the static page show"), because `fetch` does not block rendering.

Other ways to get fresh data

TECHNIQUE	IDEA
Polling	client asks again every few seconds, whether or not there is news
Long polling	client asks once; the server holds the request open until data exists, then answers
WebSockets	one persistent two-way connection (chat)
Webhooks	the server calls **your** URL when an event happens
Pub/Sub	publishers send messages to a topic; many subscribers receive them

Match used in the papers: webhooks = notification when an event occurs; websockets = real-time chat; API = request/response over HTTP; polling = fetch periodically; pub/sub = broadcast to many subscribers.

Exam radar

EXAM RADAR

The evidence file counts questions about this week's topics as about **45 of 608 Quiz 2** and **90 of 1471 End Term** questions (keyword match, so approximate; none from Quiz 1), and about 36 distinct questions were read. They cluster in these shapes:

- **Predict the output of fetch/async code** with `try/catch` inside `try/catch` (success, HTTP error code, HTML instead of JSON, network failure). Answers hinge on which block catches what.
- **CORS facts and origin comparisons** (`https://xyz.abc.com` and `https://abc8.com` give errors, `https://abc.com/page` does not; same-origin needs no CORS; CORS lets the server say which origins may load resources) and Flask-CORS with a restricted origin.
- **Timing with** `threaded=False` (50 seconds) or threaded (parallel), and "End of script" before the data.
- **Required** `Content-Type: application/json` **header** for a JSON POST.
- **Created hook with an async fetch** and what the page shows (week 4 and here).
- **Security reading questions**: string-built SQL in Flask (SQL injection), no authentication check on the route, plain-text passwords, no token mechanism. The usual answer is "all of these".
- **Concept matching**: webhooks, websockets, polling, long polling, pub/sub; REST versus GraphQL (REST responses can be cached; GraphQL works with any language that can make HTTP requests; it is not "always better").
- **Page speed**: factors are the number of HTTP requests and the file sizes.

WATCH OUT - REAL EXAM TRAP

- `fetch` + 404 does not hit `.catch`.
- `await` is not allowed outside `async` functions (in a plain method it is a syntax error).
- "Which origin gives a CORS error": compare scheme, host **and** port; paths do not matter; subdomains do.
- CORS is enforced by the browser; server-to-server calls and Postman ignore it.
- A DELETE returning `204` has no body: `res.json()` fails.
- Do not build SQL with f-strings from request data (SQL injection); use parameters or the ORM.

Quick quiz

QUICK QUIZ

`fetch("/api/missing")` receives a `404` response with a JSON body. What happens?

- the promise rejects and `.catch` runs
- the promise resolves; `res.ok` is false and `res.status` is 404

fetch throws a SyntaxError

the response is silently dropped

Check

An async function does `const res = await fetch(url); const data = await res.json();` in a `try`, and the `catch` returns `"failed"`. The URL returns HTTP 200 with an HTML page. What is returned?

the HTML text

the string "200"

"failed"

undefined

Check

Which header is necessary for a `fetch` POST that sends `JSON.stringify(data)` as the body to a Flask route reading `request.get_json()`?

`'Content-Type': 'multipart/form-data'`

`'Content-Type': 'application/text'`

`'Content-Type': 'application/json'`

`'Accept': '*/*'`

Check

A page at `https://abc.com` calls `fetch`. Select the targets that cause a CORS error by default.
(select all that apply)

`https://abc.com/page`

`https://xyz.abc.com`

`https://abc8.com`

`https://abc.com`

Check

Flask has `CORS(app, resources={r"/api/*": {"origins": "http://localhost:3000"}})`. The Vue app at `http://localhost:5173` calls `http://localhost:5000/api/data`. What happens?

- the request succeeds
- a CORS error occurs in the browser
- the server crashes
- a 404 is returned

Check

A page fires two `fetch` calls at once to Flask routes that sleep 20 and 30 seconds. Flask runs with `threaded=False`. About how long until both responses are logged?

- 20 seconds
- 30 seconds
- 10 seconds
- 50 seconds

Check

Select all TRUE statements about CORS. (select all that apply)

- It lets a server say which origins may load its resources.
- Same-origin requests do not need CORS.
- The client page decides which servers it may call, by setting a header.
- It is enforced by Flask, so Postman is blocked too.

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- **How does your Vue app get data from the Flask backend?**
- **What is the difference between fetch and axios?**
- **Show me how you send a JSON POST. Why that header?**
- **What is CORS and how did you solve it?**
- **The page shows "Loading..." forever. What do you check?**
- **How do you tell a network error from an HTTP error in fetch?**
- **How do you deploy the Vue frontend with Flask?**

Recap

REMEMBER THIS

- `fetch(url, options)` returns a promise of a Response. Call `res.json()` (also a promise). It rejects only on network or CORS failure; check `res.ok` for HTTP errors.
- Use `async` / `await` in methods with `try/catch/finally`; keep `loading`, `error` and the data in `data`. Start with `[]` or `{}` (not `null`) or guard with `v-if`.
- JSON POST/PUT: method, `Content-Type: application/json`, `JSON.stringify` body. 201 Created, 204 No Content (no body), 400, 401, 403, 404, 500.
- CRUD in Vue: list in `created`, `push` after create, copy the response after update, `filter` after delete, one helper for status checking.
- CORS: browser rule on origins (scheme + host + port); the server opts in with `Access-Control-Allow-Origin`; preflight `OPTIONS` for JSON/PUT/DELETE/Authorization; fix with Flask-CORS, a dev proxy or same-origin serving.
- A non-threaded Flask server runs requests one by one (20 + 30 = 50 s); async code lets "End of script" print first.
- Debug in the Network tab: status, request payload, response, headers, timing.

CODE AND FACTS TO KEEP

- `const res = await fetch(url); if (!res.ok) throw new Error(res.status); const data = await res.json();`
- `fetch(url, { method: "POST", headers: { "Content-Type": "application/json" }, body: JSON.stringify(obj) })`
- axios: `const { data } = await axios.get(url)`, `axios.post(url, obj)`, errors in `err.response.status`
- Flask: `CORS(app)` or `CORS(app, resources={r"/api/*": {"origins": "http://localhost:8080"}})`
- Return codes: `return jsonify(obj), 201`, `return "", 204`, `return jsonify({"error": "..."}), 404`
- Dev proxy (`vue.config.js`): `devServer: { proxy: { "/api": { target: "http://localhost:5000", changeOrigin: true } } }`
- Serve build: `Flask(__name__, static_folder="dist", static_url_path="")` and `send_from_directory(app.static_folder, "index.html")`
- Origin = scheme + host + port; path does not matter; subdomain differs
- Loading pattern: `loading = true; try { ... } catch (e) { error = e.message } finally { loading = false }`

MAD2 week 6: Advanced Vue: components, props, events and Vue Router

> **Official syllabus line:** Advanced Vuejs

The one-minute story

THINK OF IT LIKE THIS

Components are Lego bricks, the router is the signboard, and props and events are the parent and child talking. A big page (a dashboard) is not one huge block. You build small bricks: a navigation bar, a card, a button. Each brick has its own look and its own little notebook of data. You can use the same brick many times.

Bricks talk in only two ways, like a parent and a child at home. The parent gives the child things (**props**: "here is the name to show"). The child cannot open the parent's cupboard; if it wants something changed, it calls out (**events**: "I was clicked!") and the parent decides. **Props down, events up.**

The **router** is the signboard at a railway station. It does not build new pages from the server; it just decides which brick sits in the big empty frame (`router-view`) for the address in the URL. You press a link, the address changes, and a different brick takes the frame, with no full page reload.

By the end of this week you can:

- create components (global and local), pass **props** with validation, and send **events** back with `$emit`;
- use **slots** (default, named, scoped), dynamic components and `keep-alive`;
- read and write a **single file component** (`.vue`) and set up a Vue CLI project;
- configure **Vue Router 3**: routes, `router-link`, `router-view`, dynamic segments, query, named routes, nested routes, guards, lazy loading;
- predict what a component tree or a router configuration renders for a given URL;
- state the lifecycle order for a parent and its child.

Words you need

WORDS YOU NEED

- **Component** - a reusable piece of UI with its own template, data and logic.
- **Global / local registration** - global: `Vue.component("name", { … })` works everywhere; local: listed in a component's `components` option, works only inside that one.
- **Prop** - data a parent passes to a child (`:title="x"`); read-only in the child.
- **Custom event** - a message from child to parent: `this.$emit("name", payload)`, heard with `@name="handler"`.
- **Slot** - a hole in a child's template that the parent fills with its own HTML.
- **Single file component (SFC)** - a `.vue` file with `template`, `script` and `style` parts, compiled by a build tool.
- **Vue CLI** - the command-line tool that creates and builds Vue projects (`vue create`).
- **Vue Router** - the official router library; version 3 goes with Vue 2.
- **Route** - a mapping from a URL path to a component.
- **Dynamic segment** - a path part such as `:id` that matches any value; read it as `$route.params.id`.
- **Navigation guard** - a function that runs before a route change and can allow, redirect or cancel it.
- **Mixin** - an object of options merged into a component to share code.
- **Plugin** - a library installed with `Vue.use()` (Vue Router and Vuex are plugins).
- **provide / inject** - an ancestor offers a value and any descendant picks it up, skipping the levels between.

Idea 1: Components: global and local

A component is almost a small Vue instance, with two differences: it has **no** `el`, and `data` must be a **function** that returns an object (so each copy gets its own data, and a plain object triggers a warning I reproduced).

```
<div id="app">
  <hello-card></hello-card>
  <hello-card></hello-card>
```

```
</div>
<script>
  // Global registration: available in every template
  Vue.component("hello-card", {
    data() { return { likes: 0 }; },
    template: `<div><button @click="likes++">Like</button> {{ likes }}</div>`,
  });
  new Vue({ el: "#app" });
</script>
```

The page shows two independent buttons, because `data()` runs once per copy. If `data` were an object `{ likes: 0 }` all copies would share it.

```
// Local registration: only the registering component can use it
const NavBar = { template: `<nav>Home | About</nav>` };
const Page = {
  components: { NavBar },
  template: `<div><nav-bar></nav-bar><p>Content</p></div>`,
};
new Vue({ el: "#app", components: { Page }, template: `<page></page>` });
```

THE IDEA

Global versus local

	GLOBAL `VUE.COMPONENT(...)`	LOCAL `COMPONENTS: {...}`
Where usable	any instance and component template	only in the component that lists it
Must be done	**before** the root instance is created	in that component's options
Downside	always bundled, even if unused	none; clearer and preferred

Naming: register as `NavBar` or `"nav-bar"` ; in an **in-DOM** HTML template always write kebab-case tags (`<nav-bar>`), because HTML is case-insensitive. In string templates both work.

SOLVED EXAMPLE: 1. WHAT DOES THE PAGE SHOW? (REGISTRATION)

```
Vue.component("NavBar", { template: "<p>This is navbar</p>" });
const Model = { template: "<p>This is model</p>" };
new Vue({ el: "#app", template: "<nav-bar></nav-bar>" });
```

Answer: `This is navbar.` `NavBar` is global and used. `Model` is just a JavaScript object that is never registered or used, so it never renders. A related real question: `Batsman` has `components: { NavBar }` and its template contains `<nav-bar>` ; the root registers only `Batsman` . The root can use `batsman` , and `Batsman` can use `nav-bar` , but the root **cannot** use `nav-bar` directly.

WATCH OUT - REAL EXAM TRAP

- A local component is **not** inherited by its children: if `A` registers `C` and `A` uses `B` , then `B` cannot use `C` unless `B` registers it too.
- A component that is defined but not registered (or registered but not used in a template) renders nothing.
- `data` must be `data() { return { … } }` in a component. Two copies must not share the same object.

Idea 2: Props down

Props are the input of a component.

```
Vue.component("player-card", {
  props: {
    playerName: { type: String, default: "Unknown Player" },
    age: { type: Number, required: true },
    tags: { type: Array, default: () => [] },       // object/array defaults are
functions
    level: { validator: (v) => ["a", "b"].includes(v) },
  },
  template: `<div><h2>{{ playerName }}</h2><p>Age {{ age }}</p></div>`,
});
```

```
<player-card player-name="Rahul" :age="21"></player-card>
<player-card :age="30"></player-card>
```

- A camelCase prop (`playerName`) is passed with **kebab-case** in HTML: `player-name=" … "` .
- `age="21"` passes the **string** "21"; `:age="21"` passes the **number** 21. Use `v-bind` (`:`) for anything that is not a string.

- If the parent does not pass the prop, the `default` is used: the second card above shows `Unknown Player`. I ran a card like this: with no `player-name` it printed `Unknown Player`.
- Short form: `props: ["id", "name"]` (no checks). Type checks only warn in the console; they do not stop the app. I reproduced `Invalid prop: type check failed` for a string passed to a `Number` prop.

THE IDEA

One-way data flow. Data flows down from parent to child. A child must **not change a prop**: Vue warns "Avoid mutating a prop directly" (I reproduced this by doing `n++` on a prop) because the parent's next re-render would overwrite it. To change something:

- copy it to local state: `data() { return { count: this.initial }; }`, or
- use a computed property for a transformed version, or
- **emit an event** so the parent changes its own data (Idea 3).

If a prop is an object or array, the child *can* mutate its inside (it shares the reference), but that is bad style.

SOLVED EXAMPLE: 2. DEFAULT PROP PLUS SLOT (A REAL EXAM SHAPE)

```
<div id="app">
  <player-card>
    <p>Rahul is a versatile player, skilled in defense and attack.</p>
    <p>Position: Captain</p>
  </player-card>
</div>
<script>
  Vue.component("player-card", {
    props: { playerName: { type: String, default: "Unknown Player" } },
    template: `<div>
      <h2>{{ playerName }}</h2>
      <slot>This player is an important part of the team.</slot>
    </div>`,
  });
  new Vue({ el: "#app" });
</script>
```

The parent never passes `player-name`, so the heading is the **default** `Unknown Player` (not "Rahul"). The content between the tags **replaces** the slot's fallback text, so the fallback sentence disappears and the two paragraphs show. Rendered: `Unknown Player`, `Rahul is a versatile player ...`, `Position: Captain`. (I ran this and got exactly that.) Both `Position` lines appear because **all** unnamed content goes to the default slot.

YOUR TURN: 1. PROPS OR DATA?

A `star-rating` component receives `value` (1 to 5) from the parent and has a button that adds one. What is wrong with `methods: { more() { this.value++; } }` and how do you fix it?

▸ **Need a hint?**

▸ **Show the full solution**

Idea 3: Events up and v-model on components

The child calls `this.$emit("eventName", payload)`. The parent listens with `@eventName="handler"`.

```
<div id="app">
  <p>{{ message }}</p>
  <child-comp @message="updateMessage"></child-comp>
</div>
<script>
  Vue.component("child-comp", {
    template: `<button @click="send">Click</button>`,
    methods: {
      send() { this.$emit("message", "Hello"); },
    },
  });
  new Vue({
    el: "#app",
    data: { message: "Waiting ... " },
    methods: { updateMessage(msg) { this.message = msg; } },
  });
</script>
```

Clicking the button changes the text from `Waiting ...` to `Hello` (tested). The payload is passed as the handler's first argument. Both of these are correct ways (a real MSQ):

- `@message="updateMessage"` with `this.$emit("message", "Hello")` (handler receives the payload);
- `@message="updateMessage('Hello')"` with `this.$emit("message")` (the parent supplies the argument in the inline call).

Wrong: `this.emit( ... )` (it is `$emit`), `:message=" ... "` (that is a prop, not a listener), or an event name that differs from the `@` name (names must match exactly, including case).

v-model on a component

In Vue 2, `v-model` on a component is shorthand for a prop named **`value`** and an event named **`input`**:

```
Vue.component("my-input", {
  props: ["value"],
  template: `<input :value="value" @input="$emit('input', $event.target.value)">`,
});
```

```
<my-input v-model="name"></my-input>
```

Typing in the box changes the parent's `name` (tested). The correct way to do "two-way binding in a custom component" is `v-model` **with a prop and an emit**; not mutating the prop, not a plain `v-bind`. There is also `:title.sync="x"`, which pairs the prop `title` with an event `update:title`.

In newer Vue 3: `v-model` uses the prop `modelValue` and the event `update:modelValue`; components can declare `emits: [...]`.

THE IDEA

Parent and child talk: props down, events up.

DIRECTION	MECHANISM	EXAMPLE
parent to child	props	`<child :name="user.name">`
child to parent	events	`this.$emit("saved", item)`
ancestor to deep descendant	provide / inject (Idea 7)	`provide: { theme }`
unrelated components	shared state: Vuex (week 7)	`$store`
parent peeks at child	`ref` and `this.$refs.child`	use sparingly

YOUR TURN: 2. WIRE A COUNTER

Write a `counter-button` child that shows `count` from a prop and, when clicked, asks the parent to increase it by 1. Show the parent.

▸ Need a hint?

▸ Show the full solution

Idea 4: Slots

A **slot** lets the parent decide part of the child's markup. Content in the parent's template is compiled with the **parent's** data.

```
Vue.component("data-panel", {
  template: `
    <div>
      <header><slot name="header">Default title</slot></header>
      <main><slot>Default body</slot></main>
      <footer><slot name="footer"></slot></footer>
    </div>`,
});
```

```
<data-panel>
  <template v-slot:header><h1>Navigation Menu</h1></template>
  <p>Main body text</p>
  <template v-slot:footer>Last updated: Today</template>
</data-panel>
```

- **Default slot** (`<slot>`): receives all content that is **not** in a named template. Text inside `<slot>…</slot>` is **fallback** shown only if the parent provides nothing.
- **Named slot** (`<slot name="header">`): filled by `<template v-slot:header>` (shorthand `#header`). Old syntax: `slot="header"` attribute (Vue 2.5 and before; still works).
- **Scoped slot**: the child passes data **up into** the slot so the parent decides how to show each item:

```
Vue.component("item-list", {
  props: ["items"],
  template: `<ul><li v-for="i in items"><slot :item="i">{{ i }}</slot></li></ul>`,
});
```

```
<item-list :items="[1, 2]">
  <template v-slot:default="sp">#{{ sp.item }}</template>
</item-list>
```

renders `<ul><li>#1</li><li>#2</li></ul>` (tested). Without a slot template, the fallback `{{ i }}` would print `1` and `2` .

WATCH OUT - REAL EXAM TRAP

What disappears. I ran these cases:

- A child **without** a `<slot>` throws away everything placed between its tags. `<main-comp> <nest-comp></nest-comp></main-comp>` where `main-comp` 's template is just `<p>This is main Component</p>` renders only `This is main Component` : the nested component never appears (a real exam question).
- Content for a **named slot that the child does not define** is silently dropped. If the child has `header` , the default slot and `actions` but no `footer` , then `Last updated: Today` placed in `v-slot:footer` is **not rendered**. A real paper's output is `Navigation Menu` , `Main Dashboard ...` , `Extra content outside templates` , `Save Changes Cancel` , without the "Last updated" line, for this reason.
- Unnamed content outside any `template` goes to the **default** slot.

Idea 5: Dynamic components and keep-alive

`<component :is="name">` renders whichever component `name` currently refers to: a tab view without a router.

```
new Vue({
  el: "#app",
  data: { current: "tab-a" },
  components: {
    "tab-a": { template: "<p>A <input></p>" },
    "tab-b": { template: "<p>B</p>" },
  },
  template: `<div>
      <button @click="current = 'tab-a'">A</button>
      <button @click="current = 'tab-b'">B</button>
      <keep-alive><component :is="current"></component></keep-alive>
    </div>`,
});
```

- Switching destroys the old component and creates the new one, so typed text in `A` 's input is lost.
- Wrapping in **`<keep-alive>`** keeps the inactive component alive in memory with its state. I tested: `created` runs once, switching away runs **`deactivated`** , switching back runs **`activated`** , and `created` does **not** run again.
- `keep-alive` also works around `<router-view>` to keep a page's state while navigating away.

Idea 6: Single file components and the Vue CLI

In a real project, components live in **`.vue` files**:

```
<!— src/components/TaskItem.vue —>
<template>
  <li :class="{ done: task.done }" @click="$emit('toggle', task.id)">
    {{ task.title }}
  </li>
</template>

<script>
export default {
  name: "TaskItem",
  props: { task: { type: Object, required: true } },
};
</script>

<style scoped>
.done { text-decoration: line-through; }
</style>
```

- `template` : exactly one root element in Vue 2.
- `script` : `export default { … }`, the options object (name, props, data, methods...).
- `style` : CSS. `scoped` makes the rules apply **only to this component's** elements (Vue adds a unique `data-v-xxxx` attribute to them).
- A browser cannot run a `.vue` file. It must be **compiled** by a tool (vue-loader in Vue CLI, or Vite). You cannot load a `.vue` file in a script tag. Real MSQ: ".vue files typically contain template, script and style" is true; "must be compiled before they run in the browser" is true; "can be imported directly without preprocessing" is false.

Using it:

```
<script>
import TaskItem from "./components/TaskItem.vue";
export default {
  components: { TaskItem },
  data() { return { tasks: [{ id: 1, title: "Learn", done: false }] }; },
  methods: { toggle(id) { const t = this.tasks.find((x) ⇒ x.id ≡ id); t.done =
!t.done; } },
};
</script>
<!— in the template: —>
<!— <ul><task-item v-for="t in tasks" :key="t.id" :task="t" @toggle="toggle"></task-
item></ul> —>
```

Vue CLI

```
npm install -g @vue/cli        # once
vue create my-app              # choose Vue 2 (and Router if you want it)
cd my-app
npm run serve                  # dev server with hot reload, http://localhost:8080
npm run build                  # production files in dist/
```

Project layout: `public/index.html` (the page with `div id="app"`), `src/main.js` (creates the root instance), `src/App.vue` (root component), `src/components/`, `src/views/` (page-level components), `src/router/index.js`, `package.json` (scripts and dependencies), `node_modules/` (never edit), `vue.config.js` (optional settings, such as the proxy from week 5).

```
// src/main.js (Vue 2)
import Vue from "vue";
import App from "./App.vue";
import router from "./router";

new Vue({
  router,
  render: (h) => h(App),
}).$mount("#app");
```

`render: h => h(App)` is used because the CLI build has no template compiler at run time. **Vite** is the newer, faster tool (Vue 3 projects usually use it) and Vue CLI is in maintenance mode, but this course uses Vue CLI and Vue 2.

Idea 7: Mixins, plugins and provide / inject at a glance

Mixin. An object of options merged into a component, to share code.

```
const logMixin = {
  created() { console.log("mixin created"); },
  methods: { hello() { return "hi"; } },
};
export default { mixins: [logMixin], created() { console.log("component created"); } };
```

Both hooks run; the **mixin's hook first**, then the component's. If a name clashes (a method with the same name), the component's own wins.

Plugin. A library installed once with `Vue.use(plugin, options)`. A plugin is an object with an `install(Vue, options)` function that can add global components, directives or `Vue.prototype.$something`. `Vue.use(VueRouter)` and `Vue.use(Vuex)` are exactly that. (With the CDN scripts of Vue and Vue Router 3, the router installs itself.)

provide / inject. A far ancestor offers; any descendant takes, without passing props through every level.

```
// ancestor
provide: { theme: "dark" },
// any descendant
inject: ["theme"],      // then use this.theme  (shows "dark", tested)
```

It is **not reactive by default** for plain values; use it for configuration. For shared changing state use Vuex.

Idea 8: Vue Router 3

Vue Router maps URLs to components in a single-page app. Install `npm install vue-router@3` (for Vue 2) or use the CDN script, which gives `VueRouter`.

```
const Home = { template: "<p>Home</p>" };
const User = { template: "<p>User {{ $route.params.id }}</p>" };

const router = new VueRouter({
  routes: [
    { path: "/", component: Home },
    { path: "/user/:id", name: "user", component: User },
  ],
});
new Vue({ el: "#app", router });
```

```
<div id="app">
  <router-link to="/">Home</router-link>
  <router-link :to="{ name: 'user', params: { id: 7 } }">User 7</router-link>
  <router-view></router-view>
</div>
```

- `router-link` renders an `a` tag (`href="/user/7"`) and navigates without reloading. The active link gets the classes `router-link-active` and `router-link-exact-active`.
- `router-view` is the frame where the matched component is shown.
- Routes are tried **in the order written**; the first match wins. Put the catch-all `{ path: "*", component: NotFound }` **last**.
- Inside any component: `this.$route` is the current route (read it); `this.$router` is the router (use it to navigate). Mixing them up is a common wrong option (`this.$router.params` is wrong; `this.$route.params.username` is right).

Params, query and named routes

PART OF THE URL	ROUTE CONFIG	READ IT
dynamic segment `/user/7`	`path: "/user/:id"`	`this.$route.params.id` (a **string**, "7")
query string `/search?q=vue&page=2`	none needed	`this.$route.query.q`
hash `#top`	none	`this.$route.hash`

A real question: `message() { return this.$route.query.update ? "This is create page" : "This is update page" }` at `/#/?update=true`. The query value is the string `"true"`, truthy, so it gives **This is create page**. Even `?update=false` is the string `"false"`, which is also truthy (I checked: it prints the "create" text too); only a **missing** query gives "update".

Props from the route. `{ path: "/dashboard/:id", component: Dashboard, props: true }` passes `:id` to the component as a **prop** `id` (still a string). With `id` "8" and a lookup that falls back to the last user when none matches, `/dashboard/8` shows `This is dashboard for User3` (a real question; the compare uses `==` between a number and a string, so "1" matches 1).

Named routes. `name: "settings"` on the route lets you navigate by name: `{ name: "settings" }`. In a real question, the answer was `{ "name": "Settings" }`: the **route's** `name` (not the component's name, and `path: "Settings"` would be missing the slash and capital mismatch).

Programmatic navigation

```
this.$router.push("/user/7");
this.$router.push({ path: "/search", query: { q: "vue" } });   // /search?q=vue
this.$router.push({ name: "user", params: { id: 7 } });
this.$router.replace("/login");   // like push, but no new history entry
this.$router.go(-1);              // one step back in history (like the Back button)
```

`push` adds a history entry, `replace` does not. In Vue Router 3.1+ `push` returns a promise that rejects with `NavigationDuplicated` if you push the route you are already on (I reproduced it); add `.catch(() ⇒ {})` if that noise matters.

Nested routes

A route can have `children`. The **parent component** must contain its own `router-view` where the child appears. Child paths without a leading slash are relative to the parent.

```
const user = { template: "<div>Welcome {{ $route.params.name }}<router-view></router-
view></div>" };
const home = { template: "<p>This is home page</p>" };
const profile = { template: "<p>This is profile page of {{ $route.params.name }}</p>" };
```

```
const routes = [
  { path: "/", component: home },
  {
    path: "/user/:name",
    component: user,
    children: [
      { path: "", component: home },
      { path: "home", component: home },
      { path: "profile", component: profile },
    ],
  },
];
```

I rendered these for different URLs (abstract mode, same logic):

URL	RENDERED
`/user/mohan`	`Welcome mohan` then `This is home page` (the child with path `""` is the default)
`/user/mohan/profile`	`Welcome mohan` then `This is profile page of mohan`

SOLVED EXAMPLE: 3. NESTED ROUTE WITH A MISSING CHILD (A REAL SHAPE)

```
const routes = [
  { path: "/", component: Home },
  { path: "/dashboard", component: Dashboard,   // template: "Dashboard" + router-
view
    children: [{ path: "stats", component: Stats }, { path: "settings",
component: Settings }] },
  { path: "*", component: NotFound },
];
```

- `/dashboard/stats` renders `Dashboard` followed by `Stats Component` : parent text first, child in the parent's `router-view` . (The real paper uses `"**"` ; in my test it also acted as a catch-all, but write `*` .)
- `/dashboard` alone renders only `Dashboard` : there is no child with path `""` , so the inner `router-view` is empty.
- `/zzz` matches `*` and renders `404 Not Found` .

Similarly: a route `/` with children `profile` and `*` , and a parent template `This is header` plus `router-view` : URL `/` renders `This is header404 Page not found` (the child `*` also matches `/` , tested), and `/profile` renders `This is headerWelcome to profile page` . The parent always shows, the child follows it.

A further real question: a root template with `top-bar` , `nav-bar` and `router-view` , and a route `/video/:id` with child `comments/newest` , at `/video/5/comments/newest` : `Topbar` and `Navbar` stay visible (they are outside the `router-view`), and `Newest Comments` appears **inside** `Video Details` .

Hash mode and history mode

	HASH (DEFAULT)	HISTORY
URL	`http://host/#/records/2`	`http://host/records/2`
Config	nothing	`mode: "history"`
Server	needs nothing (the server only sees `/`)	must return `index.html` for **every** path, or a refresh gives 404
Looks	has `#`	clean URLs

Consequences I verified with a `/records/:id` route and a `*` fallback:

MODE	URL	RESULT
hash	`/#/records/2`	`Pianobind has 13 songs ...`
hash	`/records/1` (no `#`)	**not** the record: the router sees path `/`, so it matches `*` and shows `Unknown record!`
hash	`/#/records` (no id)	`*` matches: `Unknown record!`
hash	`/#/records/3`	route matches, but `records[3]` is `undefined`; `record.albums` throws; the component shows **nothing** and the console has an error. **Not** "Unknown record!"
history	`/inventory/1`	the product
history	`/#/inventory/1`	the hash is ignored; path is `/`, so `*` shows `Item not found`
history	`/inventory/3`	route matches, `item` undefined: **error and blank**, not the not-found page

The lesson: `*` only catches URLs that **no route matches**. An id that matches the pattern but has no data must be handled **in your component** (`v-if="record"`).

Same component, changing parameter

Going from `/u/1` to `/u/2` **reuses** the same component instance, so `created` and `mounted` do **not** run again. I logged: `created 1` , then on navigating `beforeRouteUpdate 2` and a watcher `1→2` . React with a watcher or the in-component guard:

```
data() { return { item: inventory[this.$route.params.productId] }; },
watch: {
  $route(to, from) { this.item = inventory[to.params.productId]; },     // as in the
paper
  // or just the param:  "$route.params.productId"(id) {  ... }
},
```

Another real question watches `'$route.path'` , sets a heading and shows links with `v-if` . At first load none of the watch runs (a watch is not `immediate`), so the heading shows the initial `data` value (`Vue Router Demo`). It then changes on each route change. A computed property that reads `$route.params` also updates by itself, because `$route` is reactive.

Navigation guards

A **guard** runs before the route changes: global `router.beforeEach((to, from, next) ⇒ { … })`, per-route `beforeEnter`, or in-component `beforeRouteEnter`, `beforeRouteUpdate`, `beforeRouteLeave`. You must call `next()` (allow), `next(false)` (cancel) or `next("/login")` (redirect), exactly once in Vue Router 3.

```
router.beforeEach((to, from, next) ⇒ {
  const loggedIn = !!localStorage.getItem("token");
  if (to.meta.requiresAuth && !loggedIn) next("/login");
  else next();
});
// route: { path: "/profile", component: Profile, meta: { requiresAuth: true } }
```

I tested a guard with `meta: { auth: true }`: asking for `/settings/7` ended at `/login` and the guard also ran again for the redirect target. `meta` is any extra data you attach to a route.

Lazy loading and other options

```
{ path: "/about", component: () ⇒ import("./views/About.vue") }   // loaded only when
visited
{ path: "/old", redirect: "/new" }                                  // redirect
{ path: "/search", component: Search, props: (route) ⇒ ({ q: route.query.q }) }
```

Lazy loading splits the build into chunks, so the first page loads faster.

In newer Vue 3: Vue Router 4 uses `createRouter({ history: createWebHistory(), routes })`, `useRoute()`, `useRouter()`, and a catch-all is `/:pathMatch(.*)*`.

Idea 9: Lifecycle of a parent and child

When a parent contains a child, hooks interleave. I logged both with a mixin:

```
create:   parent beforeCreate, parent created, parent beforeMount,
          child beforeCreate, child created, child beforeMount,
          child mounted, parent mounted
update:   parent beforeUpdate, child beforeUpdate, child updated, parent updated
destroy:  parent beforeDestroy, child beforeDestroy, child destroyed, parent destroyed
```

- Creation goes down (parent first), but `mounted` **goes up: the child's** `mounted` **runs before the parent's**. So in the parent's `mounted` the children are in the DOM.
- A component's hooks are **independent** of the parent's: it has its own lifecycle (a real MSQ).
- Typical use: load initial data in `created()`, start a timer or DOM work in `mounted()`, clear the timer in `beforeDestroy()`; `destroyed()` is after teardown, not for declaring data.
- With `keep-alive`, `activated` and `deactivated` replace the destroy and create on switching.

YOUR TURN: 3. PREDICT THE ROUTER

Routes: `{ path: "/player/:name", component: Player, props: true, children: [{ path: "", component: OneDay }, { path: "test", component: Test }] }` and `{ path: "*", component: Test }`. `Player` shows `Name: {{ name }}` and a `router-view`; `OneDay` shows `Oneday Runs: 10000`; `Test` shows `Test Runs: 5000`. What do `/#/player/Rohit`, `/#/player/Rohit/test` and `/#/player` render?

▸ **Need a hint?**

▸ **Show the full solution**

IN YOUR PROJECT

In the viva, be ready to draw your component tree: which component is the **view** (one per route), which are **reusable** (card, navbar, form), what **props** each receives, and what **events** it emits. Show a navigation guard that sends a logged-out user to the login page, and be ready to explain why you used hash mode (no server setup) or history mode (Flask catch-all route).

Exam radar

EXAM RADAR

The evidence file counts questions on this week's topics (components, props, events, router, lifecycle) as about **26 of 848 Quiz 1**, **95 of 608 Quiz 2** and **125 of 1471 End Term** questions (a keyword match, approximate), with about 75 distinct questions read. The router questions are by far the most repeated; the same nested-route problems come back across several exams. Typical shapes:

- **"What will the browser render for this URL"** for nested routes (`children`, empty child path, parent text then child text), hash versus history URLs, `*` fallback, dynamic `:id`, route `props: true`, `$route.query`, and a route whose data lookup is `undefined`.
- **Components and slots**: global versus local registration, unregistered components, default prop values, content inside a component without a slot, missing named slots.
- **Events**: choose the correct `@event` and `$emit` pair; `v-model` on a component with a prop and an emit; "props pass data from child to parent" is false.
- **API facts**: `$route` versus `$router`, named route navigation, `$router.go(-1)`, `*` as a fallback, `children` for nested routes.
- **Lifecycle hooks in components**: which hook for loading data (`created`), timers (`mounted`) and cleanup (`beforeDestroy`); `beforeDestroy` runs before teardown, `destroyed` after; data must be a function.
- **True or false**: `.vue` files and compilation; advantages of components (reuse, maintainability, easier UI management; not "automatic SQL optimization"); a few Vuex questions (week 7).

WATCH OUT - REAL EXAM TRAP

- Child `router-view` needed inside the parent component, or the child never shows.
- A child route path with a leading slash (`/profile`) is treated as **absolute**, not nested under the parent URL.
- `this.$route` (read) versus `this.$router` (act).
- `*` catches only URLs that no route matches; a matching URL with missing data fails in the component.
- In hash mode the part after `#` is the route; in history mode the `#` part is ignored.
- Props down, events up; props are never mutated by the child.
- A route `props: true` delivers params as **strings**.

Quick quiz

QUICK QUIZ

A child component's template is `<p>This is main Component</p>` (no slot). The parent writes `<main-comp><nest-comp></nest-comp></main-comp>` and both are registered. What is rendered?

- This is main Component and This is nested Component
- This is nested Component
- This is main Component
- nothing

Check

A route `{ path: "/", component: Header, children: [{ path: "profile", component: Profile }, { path: "*", component: Err }] }` and `Header` renders `This is header` and a `router-view`. The URL is `http://127.0.0.1:8080/` (hash mode). What shows?

- This is header only
- Welcome to profile page
- This is header followed by 404 Page not found
- 404 Page not found followed by This is header

Check

In a Vue 2 component, which is the right way to read `username` from the URL `/user/:username`?

- `this.$router.params.username`
- `this.$route.params.username`
- `this.$router.query.username`
- `this.$route.query.username`

Check

A child must update the parent's `message` to "Hello" when a button is clicked. Select correct pairs. (select all that apply)

- parent `@message="updateMessage"`, child `this.$emit("message", "Hello")`
- parent `@message="updateMessage('Hello')"`, child `this.$emit("message")`
- parent `:message="updateMessage"`, child `this.emit("message", "Hello")`
- parent `@updateMessage="message"`, child `this.$emit("updateMessage")`

Check

The component is `props: ["id"]` with a lookup that returns the third user when no user matches, the route is `/dashboard/:id` with `props: true`, and the URL is `/#/dashboard/8`. What shows?

- This is dashboard for User1
- This is dashboard for User2
- This is dashboard for User3
- nothing

Check

Which statements about `.vue` files are true? (select all that apply)

- They usually contain template, script and style sections.

- They must be compiled (Vue CLI or Vite) before running in a browser.
- JavaScript can import and run them with no preprocessing.
- They can be loaded with a plain script tag.

Check

For parent P and child C (child inside the parent's template), in what order do the `mounted` hooks run?

- parent then child
- child then parent
- only the parent
- they run in parallel

Check

The URL `http://localhost:8080/#/records/3` matches `/records/:id`, but the data lookup `records[3]` is undefined and the template reads `record.albums`. What does the page show?

- Unknown record! from the `*` route
- nothing, and an error appears in the console
- the first record
- a 404 from the server

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- Why do we split an app into components?
- How do components communicate?
- What is the difference between global and local registration?
- What are slots and when do you use them?
- What is a single file component and what does `scoped` do?
- How does Vue Router work? Explain `$route` and `$router`.
- How do you protect a page so only logged-in users see it?
- What is the difference between hash mode and history mode?

Recap

REMEMBER THIS

- Components: `Vue.component` (global) or `components: {}` (local). `data` must be a function. Registered children are not inherited by grandchildren.
- Props down (kebab-case in HTML, `:` for non-strings, validation with type, default, required; object and array defaults are functions); never mutate a prop. Events up with `$emit`. `v-model` on a component is `value` plus `input`.
- Slots: default, named (`v-slot:name`), scoped. A child without a slot drops the content; content for an undefined named slot is dropped.
- Dynamic components with `:is` ; `keep-alive` keeps state and uses `activated` / `deactivated` .
- SFC: template, script, style (scoped); needs Vue CLI or Vite. `npm run serve` and `npm run build` .
- Router: `routes` , `router-link` , `router-view` , `:id` in `$route.params` , query in `$route.query` , named routes, `push` , `replace` , `go(-1)` , `children` need a `router-view` in the parent, `*` last, guards with `next()` , lazy `() ⇒ import( … )` , hash versus history.
- Same component with a changed param is reused: use a watcher on `$route` .
- Lifecycle: parent create, child create, child mounted, parent mounted; destroy is parent before, child, child destroyed, parent destroyed.

CODE AND FACTS TO KEEP

- Global: `Vue.component("my-comp", { props: [ … ], data() { return { … }; }, template: " … " })`
- Local: `components: { MyComp }` (use `<my-comp>`)
- Prop: `props: { n: { type: Number, required: true, default: 0 } }` ; pass `:n="5"`
- Event: child `this.$emit("name", payload)` ; parent `@name="handler"`
- v-model on component: prop `value` , event `input`
- Slots: `<slot name="x">fallback</slot>` ; `<template v-slot:x>` (or `#x`); scoped `<slot :item="i">` with `v-slot:default="sp"`
- Dynamic: `<component :is="name">` , `<keep-alive>`
- Router: `new VueRouter({ mode: "history", routes: [ … ] })` , `{ path: "/u/:id", name: "user", component: U, props: true, children: [ … ] }`
- Read: `this.$route.params.id` , `this.$route.query.q` ; go: `this.$router.push({ name: "user", params: { id: 1 } })` , `.replace()` , `.go(-1)`
- Guard: `router.beforeEach((to, from, next) ⇒ { … next() })` ; catch-all `{ path: "*", component: NotFound }`
- Lazy: `component: () ⇒ import("./views/About.vue")`
- Vue CLI: `vue create app` , `npm run serve` , `npm run build` ; Vue 3 router: `createRouter` , `createWebHistory`

MAD2 week 7: State management with Vuex

> **Official syllabus line:** Advanced State Management

The one-minute story

> **THINK OF IT LIKE THIS**
>
> **Vuex is the school notice board and the office clerk.** Imagine a school where every class keeps its own copy of the timetable. When the principal changes a period, 30 classes must be told, one by one, and some will be told wrong. So the school puts **one timetable on one notice board** (the **state**). Anyone can *read* it (**getters** are like a ready-made "today's free periods" summary). But nobody scribbles on the board directly. To change it you fill a **request slip** (an **action** or a **commit**) and give it to the **clerk** (a **mutation**), who is the only person allowed to hold the pen. Because every change passes through the clerk, you can keep a log of who changed what, and replay it.
>
> Vuex is that notice board for a Vue app: **one shared store, read from anywhere, changed only by mutations.**

By the end of this week you can:

- explain why shared state is needed and why prop drilling and an event bus break down;
- write a Vuex 3 store with **state, getters, mutations, actions** and **modules**;
- predict the output of code that mixes `commit`, `dispatch`, `async` actions and getters;
- use `this.$store`, `mapState`, `mapGetters`, `mapMutations` and `mapActions` correctly;
- keep an auth token in the store and survive a page refresh with `localStorage`;
- say when **not** to use Vuex, and what Pinia is.

Words you need

WORDS YOU NEED

- **State** - the data the whole app shares (cart items, logged-in user, token).
- **Store** - the single object that holds state, getters, mutations and actions.
- **Getter** - a read-only computed value derived from state (like a computed property for the store).
- **Mutation** - a plain, synchronous function that changes the state. The only legal way to change it.
- **Action** - a function that may be asynchronous (API calls, timers). It never changes state itself: it commits mutations.
- **commit** - the call that runs a mutation: `store.commit('name', payload)`.
- **dispatch** - the call that runs an action: `store.dispatch('name', payload)`.
- **Payload** - the one extra argument you pass to a mutation or action. Need two values? Pass one object.
- **Module** - a slice of the store with its own state, getters, mutations and actions. **Namespaced** means its names get a prefix like `cart/`.
- **Prop drilling** - passing a prop through many middle components that do not need it, just to reach a deep child.
- **Event bus** - an empty Vue instance used as a global message channel (`$emit` / `$on`).
- **Strict mode** - a development setting that complains if state is changed outside a mutation.
- **Persistence** - keeping data after a page refresh, for example in `localStorage`.
- **Reactivity** - Vue noticing a data change and re-rendering by itself.

Idea 1: Why shared state is needed

In a small Vue app each component owns its data. Two components that need the same data can talk through **props down** and **events up**. That is fine for parent and child. It breaks when many far-apart components need the same thing: the logged-in user, the cart, the token, the notification count.

APPROACH	WHAT GOES WRONG
Prop drilling	App passes `user` to Layout, which passes it to Sidebar, which passes it to Avatar. Layout and Sidebar never use it. Every rename touches every level.
Event bus (`new Vue()` as a channel)	Anyone can emit, anyone can listen. Who changed the value? Who is listening? No central record, no devtools history, listeners leak if you forget `$off`.
Global variable	Changing it does **not** re-render components (not reactive). Nothing tracks who changed it.
Vuex store	One reactive source of truth. Every change is a named mutation that devtools can list and replay.

THE IDEA

State management rule. Data used by **one** component stays in that component (`data()`). Data used by **many** components, or that must survive navigation between routes, goes in the **store**. Vuex gives a **single source of truth**: components *read* from it and *request* changes; they do not own copies.

Short-lived UI bits (a dropdown open or closed, text half-typed in a field) are called **ephemeral state**. Keep them local. Putting everything in Vuex is a common over-engineering mistake (see Idea 8).

Idea 2: The one-way data flow

Vuex is built on a loop where data moves in **one direction only**:

```
Component --dispatch⟶ Action --commit⟶ Mutation --mutates⟶ State --renders⟶
Component
 (user click)        (async work,        (sync, the only       (reactive)         (view
updates)
                      API calls)          writer of state)
```

Components never write to state. The only road to state is **commit a mutation**. The road to an async job is **dispatch an action**, which later commits.

THE IDEA

Order for async work: `dispatch` an action, then the action `commit` s a mutation, then the **state changes**. Wrong orders such as "mutation, then action, then state" or "state, then dispatch" are false. For a simple synchronous change you may commit straight from the component and skip the action.

Idea 3: A complete store, line by line

Vue 2 + Vuex 3 loaded from a CDN, store injected into the root instance:

```
const store = new Vuex.Store({
  state: {
    todos: [{ text: 'Learn Vuex', done: true }, { text: 'Write app', done: false }],
    count: 0
  },
  getters: {
    doneCount(state) {                        // like a computed property
      return state.todos.filter(t ⇒ t.done).length
    },
    summary(state, getters) {                 // getters can use other getters
```

```
      return getters.doneCount + ' of ' + state.todos.length + ' done'
    }
  },
  mutations: {
    increment(state, n = 1) {                // (state, payload)
      state.count += n
    },
    addTodo(state, todo) {
      state.todos.push(todo)
    }
  },
  actions: {
    async incrementLater(context, n) {       // context has commit, dispatch, state,
getters
      await new Promise(resolve ⇒ setTimeout(resolve, 1000))
      context.commit('increment', n)
    }
  }
})

new Vue({ el: '#app', store })               // injects store as this.$store everywhere
```

What each part receives:

PART	SIGNATURE	CAN DO
mutation	`(state, payload)`	change state, **synchronously**
getter	`(state, getters)`	read, return a derived value
action	`(context, payload)`	`context.commit`, `context.dispatch`, read `context.state`; may be `async`

SOLVED EXAMPLE: 1. PREDICT THE OUTPUT

Using the store above (with `strict: false`):

```
store.commit('increment')            // payload missing, default n = 1
store.commit('increment', 5)
console.log(store.state.count)       // 6
console.log(store.getters.doneCount) // 1
console.log(store.getters.summary)   // "1 of 2 done"

const p = store.dispatch('incrementLater', 10)
console.log(store.state.count)       // still 6, the action is waiting 1 second
p.then(() ⇒ console.log(store.state.count)) // 16 after one second
```

`dispatch` always returns a **Promise**, even if your action is not declared `async`. The log right after `dispatch` shows the **old** value because the timer has not fired. I ran this against Vuex 3.

The payload rule: one value only

`commit('name', a, b)` passes only `a` to the mutation. The second extra value is ignored. This is a classic exam trap.

```
// WRONG: price is lost
this.$store.commit('update_total_cost', count, price)

// RIGHT: send ONE object
this.$store.commit('update_total_cost', { count: count, price: price })
// mutation:
update_total_cost(state, payload) {
  state.total_cost = payload.count * payload.price
}
```

Also note the **object-style commit**: `store.commit({ type: 'increment', n: 1 })` passes the **whole object** (including `type`) as the payload. With `increment(state, n)` that gives `state.count += {type … , n … }`, which is garbage (string concatenation). Use it only when the mutation reads `payload.n`.

Context versus `this.$store`

Inside an **action** the store is reached through its first argument: `context.commit( … )` or the destructured `{ commit }`. Inside a **component** it is `this.$store.commit( … )`. Mixing them up is the standard wrong option in "fill code1 and code2" questions: `this.$store` is not available inside a store action, and `context` does not exist in a component method.

SOLVED EXAMPLE: 2. AWARDEES: THE QUESTION PATTERN THE PAPERS KEEP REPEATING

Fill code1 (inside an action) and code2 (inside the mutation `update_final`) so that `awardees` gets the companies that have a website and capital above 40000, plus a token sent from the action.

```
actions: {
  send_task(context) {
    const token = Math.random().toString(36).slice(2)
    context.commit('update_final', { min: 40000, token: token })   // code1
  }
},
mutations: {
  update_final(state, payload) {                                    // code2
    state.companies.forEach(company ⇒ {
      if (company.website ≠ null && company.capital > payload.min) {
        state.awardees.push({ name: company.name, token: payload.token })
      }
    })
  }
}
```

Reject wrong options by these rules: (1) in an action use `context.commit`, never `this.$store.commit`; (2) pass **one** payload object, never `commit('x', 40000, token)`; (3) inside a mutation reach data as `state.companies`, never bare `companies` or `capital`; (4) push into `state.awardees`, not a local name; (5) read payload fields as `payload.min`, `payload.token`.

Idea 4: Mutations are sync, actions may be async

THE IDEA

Why mutations must be synchronous. Devtools records state **before** and **after** each mutation, so you can time-travel. If a mutation changes state later inside a timer or a fetch callback, devtools cannot tell *which* mutation caused the change. So: **mutations = synchronous, directly modify state. Actions = may be asynchronous and call APIs; they change state only by committing mutations.**

```
// GOOD mutation
ADD_TO_CART(state, product) { state.cart.push(product) }

// BAD mutations: async code inside
ADD_TO_CART(state, product) { setTimeout(() ⇒ state.cart.push(product), 500) }
```

```
ADD_TO_CART(state, product) { fetch('/api/add').then(() ⇒ state.cart.push(product)) }
ADD_TO_CART(state, product) { Promise.resolve().then(() ⇒ state.cart.push(product)) }
```

Even the third one, which looks instant, is async: `then` runs after the current code finishes.

SOLVED EXAMPLE: 3. THREE WAYS TO CHANGE THE CART

```
state: { cart: [] },
mutations: { ADD_TO_CART(state, product) { state.cart.push(product) } },
actions: {
  addToCart({ commit }, product) {
    setTimeout(() ⇒ commit('ADD_TO_CART', product), 500)
  }
}
```

```
addItem_A(p) { this.$store.commit('ADD_TO_CART', p) },    // sync, in devtools
immediately
addItem_B(p) { this.$store.dispatch('addToCart', p) },    // in devtools after
500 ms
addItem_C(p) { this.$store.state.cart.push(p) }           // bypasses mutations
```

All three change the cart (C works in non-strict mode because Vue still sees the array change). But devtools mutation history shows only **A** (at once) and **B** (with a delay). C is invisible in history. In **strict mode** C logs an error. This is almost word for word a real Quiz 2 question.

Dispatch and the Promise

An action returns a Promise, so a component can wait for it:

```
actions: {
  async addTodoAsync({ commit }, text) {
    commit('addTodo', { text, done: false })   // runs right now, before the first await
    return 'done'
  }
}
store.dispatch('addTodoAsync', 'Learn Vuex').then(msg ⇒ console.log(msg))
console.log(store.state.todos.length)
```

Output: the **length prints first** (the `commit` already ran synchronously inside the async function, so it prints the new length, for example 1 if the list was empty), then `done`. Code in an `async` function runs synchronously until the first `await`; the `.then` always runs later.

Idea 5: Using the store from components

```
Vue.component('score-display', {
  template: `
    <div>
      <p>Current Score: {{ score }}</p>
      <button @click="boostNow">Boost Now</button>
      <button @click="boostLater">Boost After 1s</button>
    </div>`,
  computed: {
    score() { return this.$store.state.score }       // read through a computed
  },
  methods: {
    boostNow()   { this.$store.commit('increaseScore') },
    boostLater() { this.$store.dispatch('increaseScoreAsync') }
  }
})
```

- Read state in a **computed** property (so the view updates). Do not copy it once into `data()`: the copy would not follow later changes.
- `this.$store.increaseScoreAsync()` is **wrong**: actions are not methods of the store; use `dispatch`. `this.$vuexstore` does not exist; the name is `$store`.
- Every component in the tree gets `$store` once the root Vue instance has `store` (not only direct children).

The map helpers

They generate the repetitive code. Import them first: `import { mapState, mapGetters, mapMutations, mapActions } from 'vuex'`.

```
computed: {
   ... mapState(['count', 'todos']),            // this.count, this.todos
   ... mapState({ n: 'count' }),                // renamed: this.n
   ... mapGetters(['doneCount', 'summary'])     // this.doneCount
},
methods: {
   ... mapMutations(['increment']),             // this.increment(5) means
this.$store.commit('increment', 5)
   ... mapActions(['incrementLater'])           // this.incrementLater(10) means dispatch
}
```

HELPER	GOES INTO	REPLACES
`mapState`, `mapGetters`	`computed`	`this.$store.state.x`, `this.$store.getters.x`
`mapMutations`, `mapActions`	`methods`	`this.$store.commit(...)`, `this.$store.dispatch(...)`

The **spread** `...` is required when you also have your own computed properties: `mapState([...])` alone returns an object, and without the spread it would replace the whole `computed`. A real question: the correct code is `computed: { ...mapState(['stateA','stateB','stateC']) }`. `mapState` expects the **names that exist in the store state**.

YOUR TURN: 1. FILL THE COMPONENT

The store has `state.score` and the mutation `increaseScore`. Write a component computed property and a method using the map helpers.

▸ **Need a hint?**

▸ **Show the full solution**

Idea 6: Getters, reactivity and the evidence-style trace

Getters are **cached computed values** of the store: they recompute only when the state they read changes. They are **reactive**, so the claim "getters are not reactive" is false.

SOLVED EXAMPLE: 4. TRACE A GETTER AFTER A MUTATION

```
state: { users: [
  { id: 1, name: 'Alice',   status: 'active',   role: 'admin' },
  { id: 2, name: 'Bob',     status: 'inactive', role: 'user'  },
  { id: 3, name: 'Charlie', status: 'active',   role: 'user'  } ] },
getters: {
  activeUsers(state) { return state.users.filter(u ⇒ u.status ⩵ 'active') },
  activeUserCount(state, getters) { return getters.activeUsers.length },
  adminUsers(state) { return state.users.filter(u ⇒ u.role ⩵ 'admin') }
},
mutations: {
  updateUserStatus(state, { userId, status }) {
    const user = state.users.find(u ⇒ u.id ⩵ userId)
    if (user) user.status = status
  }
}
```

Now `store.commit('updateUserStatus', { userId: 2, status: 'active' })`.

- `activeUsers` now holds Alice, Bob, Charlie.
- `activeUserCount` returns **3** (it was 2).
- `adminUsers` is unchanged: only Alice.

If an action toggles users **asynchronously** (for example after a 1 second delay each), the count changes only when each delayed `commit` runs: it becomes 1 after the first completes and 2 after the second. A getter always shows the state *at that moment*; it never "runs ahead".

A **getter with arguments** returns a function: `byDone: state ⇒ d ⇒ state.todos.filter(t ⇒ t.done ⩵ d)`, used as `store.getters.byDone(false)`. Such getters are not cached.

Reactivity gotchas in Vue 2

Vue 2 cannot see two kinds of change. I tested both:

```
// not detected: no re-render
state.user.age = 9          // adding a NEW property to an object
state.list[0] = 100         // setting an array item by index

// detected
Vue.set(state.user, 'age', 9)          // or this.$set( … )
state.list.splice(0, 1, 100)
state.user = { …state.user, age: 9 } // replace the whole object
```

Declare every property in the initial `state`, even as `null`, and Vue tracks it from the start. Array methods like `push`, `splice`, `pop` are fine. (Vue 3 and Pinia use Proxies and do not have this gotcha.)

Idea 7: Modules, namespacing, strict mode

As the app grows, split the store into **modules**:

```
const cart = {
  namespaced: true,
  state: () ⇒ ({ items: [] }),
  getters: { count: s ⇒ s.items.length },
  mutations: { add(s, item) { s.items.push(item) } },
  actions: { addLater({ commit }, item) { commit('add', item) } }
}
const store = new Vuex.Store({ modules: { cart } })

store.commit('cart/add', 'pen')
store.dispatch('cart/addLater', 'ink')
console.log(store.state.cart.items)       // [ 'pen', 'ink' ]
console.log(store.getters['cart/count'])  // 2
```

- Module state is nested: `store.state.cart.items`.
- Without `namespaced: true`, mutations and actions of all modules share **one global name space** (names can clash). With it you add the prefix: `'cart/add'`, and `getters['cart/count']` (bracket access, because of the slash).
- Module `state` should be a **function** if the module may be reused.
- Map helpers take a namespace first: `...mapState('cart', ['items'])`, `...mapActions('cart', ['addLater'])`.

Strict mode: `new Vuex.Store({ strict: true, ... })`. If anything changes state outside a mutation (for example `this.$store.state.count = 99`), Vuex logs the error `do not mutate vuex store state outside mutation handlers`. Use it in **development only**; it has a performance cost, so turn it off in production (`strict: process.env.NODE_ENV ≠= 'production'`).

Idea 8: API calls, the token, persistence, and when not to use Vuex

Call APIs from actions, then commit the result:

```
actions: {
  async login({ commit }, creds) {
    const res = await fetch('http://localhost:5000/login?include_auth_token', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(creds)
    })
```

```
    if (!res.ok) throw new Error('Login failed')       // component can catch it
    const data = await res.json()
    commit('setToken', data.response.user.authentication_token)
  },
  async fetchValue({ commit }) {
    const value = await new Promise(r => setTimeout(() => r('API Value'), 500))
    commit('setValue', value)
  }
}
```

Only actions should hold `fetch`. Mutations only assign the already-fetched data to state. In the real paper, a component does `await this.$store.dispatch('fetchValue')` and after 500 ms the page shows `API Value`.

Auth token in the store and the page refresh problem

State lives in **JavaScript memory**. A refresh reloads the page and the memory is wiped, so the token vanishes and the user is logged out. Fix: keep a copy in **web storage** and load it when the store is created.

```
const store = new Vuex.Store({
  state: {
    token: sessionStorage.getItem('authToken') || null     // initial value from storage
  },
  mutations: {
    setToken(state, token) {
      state.token = token
      sessionStorage.setItem('authToken', token)           // keep a copy
    },
    clearToken(state) {
      state.token = null
      sessionStorage.removeItem('authToken')
    }
  },
  actions: {
    login({ commit }, token) { commit('setToken', token) },
    logout({ commit }) { commit('clearToken') }
  }
})
```

After login and a refresh, the store starts again, reads `sessionStorage`, and the token is back in `state.token`. Storage facts the papers test:

STORAGE	LIFETIME	SHARED ACROSS TABS	SENT TO SERVER AUTOMATICALLY
Vuex state / JS variable	until refresh	no	no
`sessionStorage`	until that tab closes (survives refresh)	**no**	no
`localStorage`	until deleted (survives browser restart)	yes	no
cookie	until expiry; session cookie until browser closes	yes	**yes**, on every request

Both web storages hold **strings** as key-value pairs: store objects with `JSON.stringify` and read with `JSON.parse`. Tradeoffs of storing tokens are in MAD2 week 8.

WATCH OUT - REAL EXAM TRAP

- Persisting only *in the store* does not survive refresh; **the initial `state` must read from storage**.
- `localStorage` is not cleared on refresh, and not "synchronised across devices". `sessionStorage` is not shared across tabs.
- In a Vuex store, the mutation (not the component) is the right place to write storage, so the two copies never drift apart.

When NOT to use Vuex

- Small app or a page with one or two components: props and events are enough.
- Data used by one component (form fields, open/closed menus): local `data()`.
- Data that is just a copy of the server and can be re-fetched: fetch it where needed; a cache layer is simpler.
- Using Vuex is **not** always hard when apps grow; the opposite: it exists to keep large apps manageable. A component can still have its own local state when Vuex is used.

THE IDEA

In newer Vue (Pinia). Vue 3 recommends **Pinia**, the successor of Vuex. It drops mutations (you change state in actions, or directly), has no modules (each store is its own file), and works with TypeScript and Proxies. The ideas you learn here (single source of truth, getters, actions) carry over. This course exam uses Vue 2 + Vuex 3, so write Vuex in answers.

IN YOUR PROJECT

Your project almost certainly needs a **token and user role in the store**. Examiner questions: "What happens on refresh?" (show the `sessionStorage` or `localStorage` restore), "Where is the API call made?" (an action), "Why a mutation to set the token?" (devtools history, single writer). Keep everything else local unless two far-apart components need it. Also be ready to say that hiding a button by role in Vue is a convenience: the Flask API must check the role too.

YOUR TURN: 2. TRUE OR FALSE

(a) A mutation may contain a `fetch` call. (b) `this.$store.dispatch('x')` returns a Promise. (c) Getters are not reactive. (d) Only the direct children of the root component have `$store`. (e) `commit('m', a, b)` gives the mutation both a and b.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 3. PREDICT THE OUTPUT

```
const store = new Vuex.Store({
  state: { n: 1 },
  mutations: { add(state, k) { state.n += k } },
  actions: { addTwice({ commit }, k) { commit('add', k); commit('add', k) } }
})
store.dispatch('addTwice', 3)
console.log(store.state.n)
```

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 4. WHY DOES THE USER GET LOGGED OUT?

The `login` action calls `commit('setToken', token)` which sets `state.token = token`. Users report being logged out on refresh. Fix it.

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

Counted by keywords in the graded papers, about **48 distinct Vuex-related questions** were found, with roughly 59 of 608 Quiz 2 questions and 82 of 1471 End Term questions matching the week's keywords (approximate, the keyword match also catches neighbours). Vuex is a **favourite in Quiz 2 and the End Term**. The real questions test:

- the **"fill code1 and code2" store question** (awardees, students, total cost): `context.commit` in an action, one payload object, `state.` prefix inside the mutation (seen in 2022, 2023, 2024 papers);
- **which statements about Vuex are true**: single source of truth, common solution for parent-child and sibling data, multiple views on the same state, actions can be async; false: " `this.$vuexstore` ", "a component cannot have local state", "getters are not reactive";
- **mutation versus action**: mutations directly modify state and are synchronous; only actions make API calls;
- **sequence**: dispatch, commit, state change;
- **predict the output** of an action with `setTimeout` / `await` and a console log, getter counts after a mutation, devtools history of commit vs dispatch vs direct push;
- **mapState** with the spread syntax; and token persistence via `sessionStorage` after refresh.

WATCH OUT - REAL EXAM TRAP

- `this.$store.commit` inside an action, or `context.commit` inside a component: wrong place.
- `commit('x', a, b)` : only `a` arrives.
- Writing `awardees.push(...)` or `capital` without `state.` / `company.` inside a mutation.
- Declaring a mutation async or putting `fetch` in it.
- `mapState([...])` without the spread when you also have other computed properties.
- Thinking the first console line after `dispatch` shows the final value.

Quick quiz

QUICK QUIZ

In Vuex, what is the purpose of mutations?

- To commit asynchronous operations
- To directly (and synchronously) change the state
- To get the state

To define computed properties

Check

Which statements about Vuex are correct? (select all that apply)

Vuex is preferred when multiple views depend on the same state.

Vuex provides actions for asynchronous operations.

Vuex getters are not reactive.

Only direct children of the root can access the store.

Check

What is the correct sequence to update state after an async operation?

State, dispatch, commit, state change

Dispatch an action, commit a mutation, state change

Commit a mutation, dispatch an action, state change

State change, commit, dispatch

Check

Which snippet is a correct mutation?

```
ADD(state, p) { state.cart.push(p) }
```

```
ADD(state, p) { setTimeout(() => state.cart.push(p), 500) }
```

```
ADD(state, p) { fetch('/api').then(() => state.cart.push(p)) }
```

```
ADD(state, p) { Promise.resolve().then(() => state.cart.push(p)) }
```

Check

Inside a Vuex action, which line correctly updates the store with a min value and a token?

```
this.$store.commit('update_final', 40000, token)
```

`context.commit('update_final', 40000, token)`

`context.commit('update_final', { min: 40000, token: token })`

`this.$store.update_final({ min: 40000 })`

Check

After login the token is saved in `state.token` and also in `sessionStorage`, and `state.token` starts as `sessionStorage.getItem('authToken') || null`. The user refreshes. What happens?

Both are lost

Vuex keeps it, sessionStorage resets

The Vuex state loads the token from sessionStorage again

sessionStorage is cleared on refresh

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- **Why did you use Vuex in your project?**
- **Difference between a mutation and an action?**
- **Why must mutations be synchronous?**
- **What happens to the store when I refresh the page?**
- **Where do you call your API and why?**
- **What are getters and what are modules?**
- **When would you not use Vuex?**

Recap

REMEMBER THIS

- Vuex = one reactive **store** shared by all components; data flows one way: dispatch, commit, mutate, render.
- **State** is read (via computed, `mapState`); **getters** are cached derived values; **mutations** change state synchronously; **actions** do async work and commit.
- `commit('name', payload)` and `dispatch('name', payload)`: **one** payload. Components use `this.$store`, actions use `context`.
- `dispatch` returns a Promise; code before the first `await` in an action runs immediately.
- Modules split the store; `namespaced: true` adds the `module/` prefix. Strict mode is for development.
- The token in the store is lost on refresh; restore it from `localStorage` or `sessionStorage` in the initial state.
- Vue 2 cannot detect new properties or index assignment: use `Vue.set` or replace the object.
- Newer Vue uses Pinia; do not use Vuex for tiny or purely local data.

CODE AND FACTS TO KEEP

- `new Vuex.Store({ state, getters, mutations, actions, modules, strict })`
- mutation `(state, payload)`; getter `(state, getters)`; action `(context, payload)` with `{ commit, dispatch, state, getters }`
- component: `this.$store.state.x`, `this.$store.getters.x`, `this.$store.commit('m', p)`, `this.$store.dispatch('a', p)`
- helpers: `...mapState(['x'])`, `...mapGetters(['g'])` in `computed`; `...mapMutations(['m'])`, `...mapActions(['a'])` in `methods`
- namespaced: `commit('cart/add', x)`, `getters['cart/count']`, `mapState('cart', ['items'])`
- persistence: `sessionStorage.setItem(k, v)`, `getItem(k)`, `removeItem(k)`; same API on `localStorage`; strings only
- Reactivity fix: `Vue.set(obj, 'key', val)`, `arr.splice(i, 1, val)`

MAD2 week 8: Token authentication, API design, CORS and cookies

> **Official syllabus line:** Authentication and Designing APIs

The one-minute story

THINK OF IT LIKE THIS

Token authentication is a cinema wristband. At the ticket counter (the **login endpoint**) you show your ID and pay once. They give you a wristband (the **token**) with a stamp that only the cinema can make (the **signature**) and an expiry time printed on it. After that, every guard at every door just looks at the wristband: no need to go back to the counter, no register of who is inside. If the band is torn, fake or old, the guard says no.

Session authentication is the older way: the counter writes your name in a register (the **session on the server**) and gives you only a number. Each door guard must phone the register to check. Fine for one building; awkward for ten buildings.

Then there are **rules of the street**: **CORS** (which other websites may use this cinema's doors), **cookies** (wristbands the browser attaches by itself), and **API design** (how to name doors and what signs to show when someone is refused).

By the end of this week you can:

- explain the difference between session and token authentication and trace a token flow from login to expiry;
- read a **JWT**, say what is in each part, and explain why the payload is readable but cannot be forged;

- protect Flask routes with Flask-Security-Too or Flask-JWT-Extended, and check roles on the server and in Vue Router;
- choose where to store a token and name the XSS and CSRF risks;
- explain **CORS**, simple versus preflighted requests, and cookie attributes `SameSite`, `HttpOnly`, `Secure`;
- design a clean REST API: names, verbs, status codes, error JSON, pagination, versioning.

Words you need

WORDS YOU NEED

- **Authentication** - proving who you are (login). **Authorization** - what you are allowed to do (roles).
- **Session** - login state kept **on the server**, found with an id that usually travels in a cookie.
- **Token** - a string the client gets at login and sends with every request instead of a password.
- **JWT (JSON Web Token)** - a token made of three base64url parts, `header.payload.signature`, that carries claims and is signed.
- **Claim** - one field inside a JWT payload (`sub`, `exp`, `role`).
- **Bearer token** - "whoever bears (carries) this token gets in"; sent as `Authorization: Bearer <token>`.
- **Refresh token** - a long-lived token used only to get a new short-lived access token.
- **RBAC** - role-based access control: permissions come from the user's role.
- **Origin** - scheme + host + port, for example `http://localhost:8080`.
- **CORS** - the browser rule (relaxed by server headers) for letting JavaScript from one origin read responses from another.
- **Preflight** - the automatic `OPTIONS` request a browser sends before a "non-simple" cross-origin request.
- **XSS** - cross-site scripting: attacker's script runs in your page. **CSRF** - cross-site request forgery: another site makes your browser send an authenticated request.
- **Idempotent** - repeating the request has the same effect as doing it once (GET, PUT, DELETE).
- **Pagination** - returning a big list in pages.

Idea 1: Session versus token authentication

	SESSION (COOKIE)	TOKEN (HEADER)
Where login state lives	**Server** memory or database	**In the token** itself (JWT) or a server lookup (opaque token)
What the client keeps	A session id in a cookie	The token in memory, `localStorage` or a cookie
How it is sent	Browser attaches the cookie **automatically**	Your JavaScript adds an `Authorization` or `Authentication-Token` header

	SESSION (COOKIE)	TOKEN (HEADER)
Scaling to many servers	Needs shared session storage	Any server can verify the signature, no shared store
Works for mobile apps and other backends	Awkward	Natural
Main extra risk	**CSRF** (the cookie goes automatically)	**XSS** if the token sits in `localStorage`

A real question: a Vue frontend, a mobile app and another service call the same API on several servers. JWT fits, because each request carries the token and any server can validate it **without storing a per-user session in memory**. Saying that sessions are easier to scale because "no shared storage is needed" is false.

THE IDEA

Token flow, end to end.

1. Client sends credentials: `POST /login` with JSON `{username, password}`.
2. Server checks the password (against a **hash**), creates a token and returns it, with an **expiry**.
3. Client stores the token (say, in the Vuex store, backed by browser storage).
4. **Every** later request carries the token: `Authorization: Bearer eyJ...` (Flask-Security: `Authentication-Token: ...`).
5. Server validates signature and expiry, finds the user, runs the route, returns data. Missing or bad token: **401**. Valid token but not allowed: **403**.
6. When the token expires, the user logs in again (or a refresh token fetches a new one).

A false statement in the papers: "The client must send the token with the first request and need not send it in the subsequent requests". Wrong: it must be sent with **every** protected request, because the server keeps no memory of you. Another: "with Flask-Security all endpoints are protected by default". Wrong: you protect each route with a decorator such as `@auth_required('token')`.

Idea 2: JWT structure, and why you can read it but cannot forge it

A JWT is three base64url strings joined by **dots**: `header.payload.signature`.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI0MiIsInJvbGUiOiJhZG1pbiIsImV4cCI6MTc5MTI
3MTM0NH0.qEbqtAXKX3D5uxeiTLxLNjdK8Ychmy2str4jB1JBUAY
```

PART	CONTAINS	DECODED (THIS EXAMPLE)
Header	algorithm and type	`{"alg":"HS256","typ":"JWT"}`

PART	CONTAINS	DECODED (THIS EXAMPLE)
Payload	the **claims**	`{"sub":"42","role":"admin","exp":1791271344}`
Signature	HMAC of `header.payload` with a secret key	binary, not JSON

Each of the first two parts is **JSON encoded in base64url** (base64 with `-` and `_` and no `=` padding). Base64 is an **encoding, not encryption**: anyone can decode it. So the payload is **readable by anyone who holds the token**. Never put passwords or secrets in it.

The **signature** is what makes it trustworthy: the server computes `HMAC_SHA256(secret, header + "." + payload)`. If someone edits the payload (say `role` to `superadmin`), the signature no longer matches and the server rejects the token, because the attacker does not know the secret. So: **readable but tamper-evident**.

I ran this with PyJWT:

```
import jwt, time
SECRET = "use-a-long-random-secret-of-at-least-32-bytes"
token = jwt.encode({"sub": "42", "role": "admin", "exp": int(time.time()) + 3600},
                   SECRET, algorithm="HS256")
print(jwt.decode(token, SECRET, algorithms=["HS256"]))
# {'sub': '42', 'role': 'admin', 'exp':  ...}

jwt.decode(token, "wrong-key", algorithms=["HS256"])
# jwt.exceptions.InvalidSignatureError: Signature verification failed
```

An expired token raises `ExpiredSignatureError`, and a token whose payload was edited raises `InvalidSignatureError`. In recent PyJWT the `sub` claim must be a **string**, which is why the identity above is `"42"`. Always pass `algorithms=[...]` to `decode`.

> **THE IDEA**
>
> **Standard claims:** `sub` (subject, the user id), `exp` (expiry time), `iat` (issued at), `iss` (issuer). **HS256** uses one shared secret key to sign and verify. **RS256** uses a private key to sign and a public key to verify. A JWT is **stateless**: the server needs no lookup, but this also means you cannot easily revoke one before it expires. That is why access tokens are made **short-lived**.

Real statements: "JWT has three components, header, payload and signature" (true); "each component is base64 encoded" (true); "components are joined by a period" (true); "it contains XML" or "only the payload is contained" (false).

Idea 3: Token auth in Flask

Flask-Security-Too (the course's usual choice)

```
from flask_security import Security, SQLAlchemyUserDatastore, auth_required,
roles_required

app.config["SECRET_KEY"] = "dev-secret"
app.config["SECURITY_PASSWORD_SALT"] = "dev-salt"
app.config["SECURITY_TOKEN_AUTHENTICATION_HEADER"] = "Authentication-Token"   # the
default
app.config["WTF_CSRF_ENABLED"] = False        # typical for a pure JSON API

datastore = SQLAlchemyUserDatastore(db, User, Role)
app.security = Security(app, datastore)

@app.get("/api/profile")
@auth_required("token")                         # only token auth accepted here
def profile():
    return {"email": current_user.email}

@app.get("/api/admin")
@auth_required("token")
@roles_required("admin")                        # 403 if the user lacks the role
def admin():
    return {"ok": True}
```

Login: `POST /login?include_auth_token` with JSON credentials. The response JSON holds the token (in `response.user.authentication_token`); the client sends it back in the header `Authentication-Token`. Passwords are stored **hashed** (Flask-Security hashes them; with plain Flask use `generate_password_hash` and `check_password_hash`). Take the exact config names and the exact response shape from your course notes and the version you installed.

Flask-JWT-Extended outline (tested)

```
from flask_jwt_extended import (JWTManager, create_access_token, jwt_required,
                                get_jwt_identity, get_jwt)
app.config["JWT_SECRET_KEY"] = "a-long-random-secret"
app.config["JWT_ACCESS_TOKEN_EXPIRES"] = timedelta(minutes=15)
JWTManager(app)

@app.post("/api/v1/login")
def login():
    d = request.get_json(silent=True) or {}
    u = USERS.get(d.get("username"))
    if not u or not check_password_hash(u["hash"], d.get("password", "")):
        return jsonify(error="bad credentials"), 401
```

```
    token = create_access_token(identity=str(u["id"]), additional_claims={"role":
u["role"]})
    return jsonify(access_token=token)

@app.get("/api/v1/me")
@jwt_required()
def me():
    return jsonify(id=get_jwt_identity(), role=get_jwt()["role"])
```

I ran this: with no header, `/me` gives **401** `{'msg': 'Missing Authorization Header'}`; with `Authorization: Bearer <token>` it returns `{'id': '1', 'role': 'admin'}`. `get_jwt_identity()` reads the identity stored in the token.

Idea 4: Roles, and why the server check is the real security

RBAC: each user has a role (`admin`, `user`, `sponsor`...). The role decides which routes and actions are allowed.

```
def role_required(role):
    def deco(fn):
        @wraps(fn)
        @jwt_required()
        def wrapper(*args, **kwargs):
            if get_jwt().get("role") ≠ role:
                return jsonify(error="forbidden"), 403
            return fn(*args, **kwargs)
        return wrapper
    return deco

@app.get("/api/v1/admin")
@role_required("admin")
def admin_area(): ...
```

The Vue side (**Vue Router navigation guard**) only improves the user experience:

```
router.beforeEach((to, from, next) ⇒ {
  const token = store.state.token
  const role = store.state.role
  if (to.meta.requiresAuth && !token) next('/login')
  else if (to.meta.role && to.meta.role ≠≠ role) next('/forbidden')
  else next()
})
// routes: { path: '/admin', component: Admin, meta: { requiresAuth: true, role:
'admin' } }
```

THE IDEA

The browser is not trusted. A user can open devtools, edit the Vuex state, or call your API with `curl`, skipping the Vue router completely. A hidden button or a router guard stops honest users from wandering; only the **server check** (decorator, 401/403) protects data. Always do both, but never rely on the front end alone.

SOLVED EXAMPLE: 1. WHICH STATUS CODE?

Match each situation to a code.

- Request with no token to a protected route: **401 Unauthorized** (you are not identified).
- Valid token of a normal user calling an admin-only route: **403 Forbidden** (identified, not allowed).
- Valid token, expired: **401** (the client should log in again or refresh).
- Wrong password at login: **401**.
- Token is fine, resource id does not exist: **404**.

Idea 5: Where to keep the token, XSS and CSRF

PLACE	SURVIVES REFRESH	JAVASCRIPT CAN READ IT	SENT AUTOMATICALLY	MAIN RISK
JS memory (Vuex, a variable)	no	yes	no	lost on refresh; XSS can still use it while the page runs
sessionStorage	yes (same tab)	yes	no	**XSS** can steal it
localStorage	yes (also after restart)	yes	no	**XSS** can steal it; lives long
Cookie, plain	yes	yes	**yes**	**CSRF**, and XSS can read it
Cookie, `HttpOnly`	yes	**no**	yes	**CSRF** (needs `SameSite` and a CSRF token), safe from script theft

- **XSS**: the attacker gets their script to run on your site (for example through an unescaped comment). That script can read `localStorage` and send your token away. Defence: escape output (Vue's `{{ }}` escapes by default), avoid `v-html` on user input, keep tokens short-lived.
- **CSRF**: you are logged in to a bank in one tab; a malicious page triggers a request such as an image whose `src` is `http://example1.com/delete?id=1`. The browser **attaches your cookie automatically** and the server thinks it is you. This real exam scenario is CSRF, not XSS. It works because **cookies are sent automatically**. A token placed in a header by your own JavaScript is not

sent by a foreign page. Defences: **CSRF tokens**, `SameSite` cookies, requiring a custom header, never changing data with GET.
- Flask does **not** enforce CSRF protection by default, and does **not** send CORS headers by default.

Refresh token idea: the login returns two tokens. The **access token** lives 5 to 15 minutes and goes on every request. The **refresh token** lives days, is sent only to a `/refresh` endpoint and returns a new access token. If the access token is stolen the damage window is short, and the user does not have to type the password every 15 minutes.

Idea 6: CORS in depth

Two URLs have the **same origin** only if scheme, host **and port** all match. `http://localhost:8080` (Vue dev server) and `http://localhost:5000` (Flask) are **different origins** because the port differs.

Browsers enforce the **same-origin policy**: JavaScript from origin A may not **read** a response from origin B unless B says so. **CORS** is how B says so, with response headers. Notes the papers test:

- CORS **lets a server declare which origins may load its resources**. It does not encrypt, compress or authenticate.
- Same-origin requests need no CORS.
- It is the **browser** that blocks. `curl` and Postman ignore CORS. The request often still reaches the server; it is the *response* the page may not read.
- By default Flask sends no CORS headers, so a `fetch` from another origin fails until you enable them.

```
from flask_cors import CORS
CORS(app, origins=["http://localhost:8080"])      # allow just the Vue dev server
```

Simple versus preflighted. A *simple* request is a GET, HEAD or POST with a basic content type (`text/plain`, form data) and no custom headers. Sending JSON (`Content-Type: application/json`), an `Authorization` header, or using PUT or DELETE makes it **non-simple**. Then the browser first sends an automatic **preflight**:

```
OPTIONS /api/v1/me
Origin: http://localhost:8080
Access-Control-Request-Method: GET
Access-Control-Request-Headers: authorization
```

The server answers (I captured this from Flask-CORS):

```
Access-Control-Allow-Origin: http://localhost:8080
Access-Control-Allow-Methods: DELETE, GET, HEAD, OPTIONS, PATCH, POST, PUT
Access-Control-Allow-Headers: authorization
```

If these match, the browser sends the real request, and the real response also carries `Access-Control-Allow-Origin`. For an origin that is not on the list the header is simply **missing**, and the browser refuses to give the response to the page.

HEADER	MEANING
`Access-Control-Allow-Origin`	which origin may read the response (`*` or one origin)
`Access-Control-Allow-Methods`	methods allowed (preflight answer)
`Access-Control-Allow-Headers`	request headers allowed, e.g. `Authorization, Content-Type`
`Access-Control-Allow-Credentials: true`	cookies may be sent and read; then `Allow-Origin` **cannot be** `*`, it must name the origin
`Access-Control-Max-Age`	seconds the browser may cache the preflight

On the client, cookies cross origins only with `fetch(url, { credentials: 'include' })`.

Idea 7: Cookies, briefly

A cookie is a small string the server sets with `Set-Cookie` and the browser sends back on later requests to that site.

```
resp = make_response({"ok": True})
resp.set_cookie("session_id", "abc123",
                httponly=True,        # JavaScript cannot read it (document.cookie)
                secure=True,          # sent only over HTTPS
                samesite="Strict",    # not sent on cross-site requests
                max_age=3600)         # seconds; omit for a session cookie
```

ATTRIBUTE	EFFECT
`HttpOnly`	hides it from JavaScript; blocks theft through XSS
`Secure`	only over HTTPS
`SameSite=Strict`	never on cross-site requests; `Lax` sends on top-level navigation by GET; `None` always (and needs `Secure`)
`Domain`, `Path`	which hosts and URL paths receive it
`Max-Age` / `Expires`	lifetime; **without it, a session cookie that is deleted when the browser closes**

Real answers: the pair that makes a cookie **HTTPS-only and blocks cross-site requests** is `secure=True, samesite='Strict'`. A cookie is the right choice for data that "must be sent to the server with every request"; a session cookie is sent with each request **by default**. A session cookie vanishes when the browser closes; `sessionStorage` survives a refresh but dies with the tab.

Idea 8: Designing good APIs

PRACTICE	GOOD	BAD
Nouns, plural, lower case	`/api/v1/products`	`/getAllProducts`
Hierarchy	`/api/v1/users/7/orders`	`/api/v1/getOrdersOfUser?id=7`
Verb is the HTTP method	`DELETE /products/5`	`POST /deleteProduct`
Right status codes	`201` after create, `204` after delete, `404` missing, `400` / `422` bad input	`200` for everything
One error shape	`{"error": "name is required", "field": "name"}`	sometimes text, sometimes HTML
Pagination	`GET /products?page=2&per_page=20`	return 50,000 rows
Filtering and sorting	`?category=toys&sort=price`	separate endpoints per filter
Versioning	`/api/v1/...` so old clients keep working	breaking changes in place

```
@app.get("/api/v1/products")
def list_products():
    page = request.args.get("page", 1, type=int)
    per_page = min(request.args.get("per_page", 20, type=int), 100)
    q = Product.query.order_by(Product.id)
    p = db.paginate(q, page=page, per_page=per_page, error_out=False)
    return jsonify(items=[x.to_dict() for x in p.items], page=p.page,
                   pages=p.pages, total=p.total)
```

- **Idempotency:** GET, PUT and DELETE repeated give the same final state. POST is **not** idempotent (two POSTs may create two rows). For safe retries of POST, clients may send an idempotency key.
- **Rate limiting:** limit calls per user or IP (Flask-Limiter); respond **429 Too Many Requests**.
- **HATEOAS:** the response contains **links** to what you can do next (`"links": {"self": ..., "next": ...}`) so a client follows links instead of building URLs. It is an idea to know; few APIs do it fully.

- **Permalink:** a stable URL for one resource that does not change, not a session-bound or temporary URL.
- **REST versus GraphQL:** REST has many endpoints and may over-fetch or need several calls ("chatty"); GraphQL has one endpoint and the client asks for exactly the fields, usually returning 200 even for errors, with an `errors` field; writes are explicit **mutations**.
- **OpenAPI recap:** a YAML description of your endpoints, parameters and responses, used for documentation and testing. Write it once, tools generate Swagger pages.
- **Sending email from an API:** do not send inside the request. Use `smtplib`, Flask-Mail or a mail service, and move the sending to a background job (week 9). Never put the SMTP password in code; read it from the environment.

SOLVED EXAMPLE: 2. A LOGIN IN VUE THAT SURVIVES REFRESH AND SENDS THE HEADER

```
// store action
async login({ commit }, creds) {
  const res = await fetch('http://localhost:5000/api/v1/login', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(creds)
  })
  if (!res.ok) throw new Error('bad credentials')
  const { access_token } = await res.json()
  commit('setToken', access_token)            // mutation also writes
sessionStorage
}

// any API call
fetch('http://localhost:5000/api/v1/me', {
  headers: { Authorization: 'Bearer ' + this.$store.state.token }
})
```

Because the requests are to another origin, carry JSON and an `Authorization` header, the browser sends a preflight `OPTIONS` first: the Flask side must allow the origin and the `Authorization` header, or you will see a CORS error in the console even though the API is fine.

WATCH OUT - REAL EXAM TRAP

- 401 means "not authenticated", 403 means "authenticated but forbidden". They are not the same.
- Base64 is **not** encryption: a JWT payload is readable.
- A JWT is signed, not secret: do not store passwords in it.
- Same origin means same **port** too.
- Wildcard `Access-Control-Allow-Origin: *` cannot be combined with credentials.
- Hiding a Vue route is not security; the server must check.
- A token must be sent on **every** request; the server remembers nothing.

IN YOUR PROJECT

Your project likely has roles (admin, sponsor, influencer, doctor, and so on). Examiner questions: "How do you block a normal user from an admin API?" (show the decorator that returns 403), "What is in your token and how long does it live?", "Where is the token stored and why?", "Why do you get CORS errors, and how did you fix them?" (Flask-CORS with your Vue origin). Know your status codes and keep one error JSON shape.

YOUR TURN: 1. DECODE THE CLAIM

A JWT payload decodes to `{"sub":"7","role":"user","exp":1700000000}`. A user changes `role` to `admin` in the payload and sends the token back. What does the server do and why?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 2. PREFLIGHT OR NOT?

A page at `http://localhost:8080` runs `fetch('http://localhost:5000/api/v1/me', {headers: {Authorization: 'Bearer x'}})`. Is it same-origin? Is there a preflight? Which header must the server allow?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 3. FIX THE API NAMES

Improve: `POST /createNewUser`, `GET /getUserOrders?userId=3`, `POST /removeOrder?id=9`. Give the status code for a successful delete.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 4. CHOOSE THE COOKIE FLAGS

You set a login cookie. It must not be readable by JavaScript, must go only over HTTPS and must not be sent from other sites. Write the call.

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

In the graded papers, keyword matching finds about **77 distinct questions** touching this week's themes (token auth, JWT, CORS, cookies, CSRF, web storage, REST versus GraphQL, and ephemeral state), with about 76 of 608 Quiz 2 and 146 of 1471 End Term questions matching; the count is approximate and overlaps neighbouring weeks, and the **token auth, JWT and CORS** content shows up as a handful of questions each year rather than a whole paper. What they ask:

- **JWT facts**: three parts, base64, joined by periods; contains encoded JSON, not XML; used as an access token; the Flask-JWT-Extended `get_jwt_identity()`; why JWT suits many servers.
- **Token auth true/false**: tokens expire (and the time can be customised); the token goes with every request; Flask-Security does not protect all endpoints by default.
- **CORS**: servers declare allowed origins; not needed for same origin; primary purpose "control which domains can access resources"; Flask does not send CORS headers by default.
- **Cookies and CSRF**: `secure=True, samesite='Strict'`; a cookie is the way to send data back automatically; the `delete?id=1` image scenario is CSRF; the purpose of a CSRF token.
- **Browser storage**: `sessionStorage` per tab and survives refresh, `localStorage` persists; predict what a Vue component shows after refresh when it reads and writes storage; **REST versus GraphQL** statements.

WATCH OUT - REAL EXAM TRAP

- "Flask enforces CSRF by default" and "Flask returns CORS headers by default" are both false.
- `sessionStorage` is **not** shared across tabs and is **not** lost on refresh.
- Local storage data is not synced across devices.
- Selecting `secure=False` in the cookie question, or `samesite='None'` for blocking cross-site, is wrong.

Quick quiz

QUICK QUIZ

Which are true about a JWT? (select all that apply)

- It has three components: header, payload and signature.
- Each component is base64-encoded.
- The components are joined with periods.
- It contains XML-encoded data.

Check

Which statement about token-based authentication is FALSE?

- The token usually expires after a configurable time.
- The client sends the token only with the first request.
- A cross-origin fetch to Flask fails by default because of CORS.
- Flask-Security protects a route only if you decorate it.

Check

Which pair makes a cookie HTTPS-only and blocks cross-site sending?

- `secure=False, samesite='Strict'`
- `secure=True, samesite='None'`
- `secure=True, samesite='Strict'`

`secure=False, samesite='Lax'`

Check

A malicious page with an image whose address is `http://example1.com/delete?id=1` deletes a logged-in user's resource. This attack is...

- Cross-site scripting
- Cross-site request forgery
- Session hijacking by packet sniffing
- SQL injection

Check

Select the correct statements about CORS. (select all that apply)

- It lets a server say which origins may load its resources.
- It is not required for same-origin requests.
- It lets the client choose which servers it will accept.
- It encrypts traffic between browser and server.

Check

A user with a valid token but the wrong role calls an admin-only API. The best status code is...

- 200
- 401
- 403
- 404

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- How does token authentication work in your project?
- What is a JWT and why is the payload readable but safe from tampering?
- How do you implement roles?
- Where do you store the token, and what are the risks?
- What is CORS and why did you get an error?
- Give three API design rules you followed.
- What is the difference between 401 and 403?
- What is a refresh token?

Recap

REMEMBER THIS

- **Session**: state on the server, id in a cookie. **Token**: state in the signed token, sent in a header on **every** request; scales across servers.
- **JWT** = `header.payload.signature`, base64url, joined with dots. Readable, not forgeable. Short-lived access token plus refresh token.
- Flask-Security: `@auth_required('token')`, header `Authentication-Token`, login with `?include_auth_token`. Flask-JWT-Extended: `create_access_token`, `@jwt_required()`, `get_jwt_identity()`, header `Authorization: Bearer`.
- **RBAC**: check the role on the server (403). Vue Router guards only improve the experience.
- Token storage: memory, `sessionStorage`, `localStorage` have XSS risk; cookies add CSRF risk unless `HttpOnly`, `Secure`, `SameSite`.
- **CORS**: origin = scheme + host + port; non-simple requests trigger a preflight `OPTIONS`; servers answer with `Access-Control-Allow-*`.
- API design: plural nouns, verbs, status codes, one error JSON, pagination, versioning, idempotency, rate limiting (429).

CODE AND FACTS TO KEEP

- Send token: `Authorization: Bearer <token>` or `Authentication-Token: <token>`
- 200 OK, 201 Created, 204 No Content, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 409 Conflict, 422 Unprocessable, 429 Too Many Requests, 500 Server Error
- PyJWT: `jwt.encode(payload, key, algorithm="HS256")`, `jwt.decode(token, key, algorithms=["HS256"])`
- Flask-JWT-Extended: `create_access_token(identity=str(id))`, `@jwt_required()`, `get_jwt_identity()`, `get_jwt()`
- Flask-CORS: `CORS(app, origins=["http://localhost:8080"])`
- Cookie: `resp.set_cookie(name, val, httponly=True, secure=True, samesite="Strict", max_age=3600)`
- fetch with cookies cross-origin: `credentials: 'include'`, server needs `Access-Control-Allow-Credentials: true`
- Idempotent: GET, PUT, DELETE. Not idempotent: POST.

MAD2 week 9: Asynchronous jobs: Redis and Celery

> **Official syllabus line:** Asynchronous Jobs

The one-minute story

THINK OF IT LIKE THIS

A restaurant with a waiter, an order board and a kitchen. You (the browser) tell the waiter (the Flask server) "one biryani". If the waiter walked into the kitchen, cooked for 30 minutes and only then came back, nobody else in the restaurant would be served. So the waiter writes the order on a slip, pins it on the **order board** (the **message broker**, here Redis) and comes straight back: "Order taken, your token number is 17." The **cooks** (Celery **workers**) take slips from the board whenever they are free and cook. When the food is ready the cook leaves the plate on the **pickup counter** (the **result backend**). You come back with token 17 and ask "Is it ready?" (**polling**). There is also a clock on the wall: every morning at 6 it automatically pins a slip "Make the daily tea" on the board (**Celery beat**).

Slow jobs do not belong inside a web request. They belong on a board, for cooks.

By the end of this week you can:

- explain synchronous versus asynchronous work and why slow jobs must leave the request;
- explain the roles of broker, worker and result backend, and what Redis does in this setup;
- write a Celery task, call it with `.delay()` and `.apply_async()`, and predict what it returns;
- configure Celery with Flask (app context) and start a worker and a beat scheduler;

- design a **CSV export** flow (trigger, background build, status polling, download) and a daily reminder;
- explain retries, idempotency and failure handling.

Words you need

WORDS YOU NEED

- **Synchronous** - do the job and wait until it finishes before moving on.
- **Asynchronous** - start the job and carry on; the result comes later.
- **Message broker** - a middleman that receives messages from senders (producers) and holds them until receivers (consumers) take them.
- **Redis** - a very fast **in-memory** data store, used here as the broker and result store (also as a cache).
- **Celery** - a Python library for a **task queue**: it runs functions in separate worker processes.
- **Task** - a Python function registered with Celery (`@celery.task`).
- **Worker** - a process that fetches tasks from the broker and runs them.
- **Result backend** - where task states and return values are stored.
- **Producer** - the code that sends a task (your Flask app).
- **Polling** - the client asks repeatedly "is it done yet?". Short polling asks on a timer.
- **Celery beat** - the scheduler that sends periodic tasks to the broker at the right times.
- **Crontab** - a schedule written with minute, hour, day fields, such as "every day at 18:00".
- **Idempotent task** - running it twice has the same effect as once.
- **Webhook** - the server calls a URL of the client when something happens (push, not poll).

Idea 1: Synchronous versus asynchronous, and why jobs leave the request

A Flask worker handles one request at a time (per worker process). A route that builds a PDF and sends an email takes, say, 12 seconds. During that time that worker is busy: other users queue up and the page feels dead, and the browser may time out. Using Celery **does not make the job faster** (it does not add CPU cores); it moves the job to another process so the web request can return in milliseconds.

```
# synchronous: the user waits 12 seconds, the server is blocked
@app.get("/report")
def report():
    pdf = build_pdf()          # 10 s
    send_email(pdf)            # 2 s
    return {"done": True}

# asynchronous: respond at once, work happens elsewhere
@app.get("/report")
def report():
```

```
    task = build_and_send.delay()
    return {"task_id": task.id}, 202      # 202 Accepted: work has been queued
```

Good fits for Celery: sending emails, generating reports and CSV/PDF files, image or video processing, calling slow external APIs, nightly cleanups, **periodic** emails or reminders. Bad fits: handling real-time user interactions, generating the page the user is waiting for, serving live notifications (that is for WebSockets or Server-Sent Events).

Do not use `time.sleep()` as a trick to wait for a job inside a route: it blocks the worker, which is the very problem we are solving.

Idea 2: Message broker concept

Think of 50 servers that must talk to each other. **Point-to-point**: every pair needs its own connection, so the number of links grows as **O(n squared)**. With a **central broker**: each server connects only to the broker, so links grow as **O(n)**. This counting question appears again and again; remember: *point-to-point O(n^2), broker O(n)*.

Benefits of a broker: it **decouples producers and consumers** (they need not run at the same time, or know each other), makes the system **scalable** (add more workers), **absorbs traffic spikes** (messages wait in the queue until processed) and enables **batch processing**. If the consumer is slower than the producer, messages **accumulate in the queue** (they are not dropped, the producer does not block). A **topic** is a named channel to which messages are published. A broker does **not** let servers talk "directly without an intermediary", that is the opposite.

Two delivery styles: **push queues** (the broker pushes to consumers, good for real-time) and **pull queues** (consumers fetch when ready, good for batch work). Celery workers *pull*.

Idea 3: Redis, the board in the kitchen

> **THE IDEA**
>
> **Redis is an in-memory data structure store.** It keeps data in RAM (so it is extremely fast, hundreds of thousands of operations per second), is not a relational database and does not use SQL. It is used as a **database, cache and message broker**, supports **Pub/Sub**, and can be a Celery **broker** and a Celery **result backend**.

It is more than strings. Typical data types and commands (I checked the behaviour with a Redis emulation):

TYPE	COMMANDS	MEANING
String	`SET name Asha`, `GET name`, `INCR visits`	a value; numbers can be incremented atomically
List	`LPUSH q a b`, `RPUSH q c`, `LRANGE q 0 -1`, `RPOP q`	ordered; a natural **queue**

TYPE	COMMANDS	MEANING
Set	`SADD s x x y`, `SMEMBERS s`	unique members: `x` stored once
Hash	`HSET user:1 name A age 20`, `HGETALL user:1`	field-value map (like a small dict)
Sorted set	`ZADD lb 10 a 5 b`, `ZRANGE lb 0 -1 WITHSCORES`	members ordered by score (leaderboards)

```
127.0.0.1:6379> SET name Asha
OK
127.0.0.1:6379> EXPIRE name 60
(integer) 1
127.0.0.1:6379> TTL name
(integer) 60            # seconds left; -1 = no expiry, -2 = key does not exist
127.0.0.1:6379> LPUSH q a b
(integer) 2
127.0.0.1:6379> LRANGE q 0 -1
1) "b"
2) "a"
```

`LPUSH` pushes to the **left**, so the last pushed (`b`) is first. A key with `EXPIRE` disappears by itself, which is how caches clean up. `redis-cli` opens the interactive shell; `PING` should answer `PONG`.

Persistence. Because data lives in memory, a restart would lose it unless persistence is on: **RDB** takes periodic snapshots to a file; **AOF** (append-only file) logs every write. Redis can use both. For Celery as a queue and cache, losing some data on restart is often acceptable; for your only copy of important data, keep it in the SQL database.

Python client: `r = redis.Redis(host="localhost", port=6379, db=0, decode_responses=True)`, then `r.set("k","v")`, `r.get("k")`. Redis has numbered databases `0..15` inside one server; the URL `redis://localhost:6379/1` means database 1. Redis also does not have to be the broker: Celery can use RabbitMQ and others, and a system can have several brokers.

Redis on Windows

Redis is built for Linux. On Windows: use **WSL** (install Ubuntu, then `sudo apt install redis-server`, `sudo service redis-server start`) or **Docker** (`docker run -p 6379:6379 redis`). Port 6379 is then reachable as `localhost`. Check with `redis-cli ping`.

Idea 4: Celery architecture and your first task

```
Flask app           Redis (broker)         Celery worker(s)        Redis (backend)
(producer)  --1──> [ task queue ] --2──> run task function --3──> [ state + result ]
```

```
    |                                                                  ^
    +———————————————————— 4. AsyncResult(id).state / .get() ————————+
```

COMPONENT	JOB
Producer / Celery app	your Flask code that calls `.delay()`; creates a message
Broker	stores messages until a worker takes them (Redis, RabbitMQ)
Worker	separate process; imports your tasks, runs them, can run in parallel with other workers
Result backend	stores state and return value (Redis, a database); optional but needed for `.get()` and status polling

In the real paper the code `Celery('tasks', broker='redis://localhost:6379/1', backend='redis://localhost:6379/2')` is explained like this: the **broker stores task requests until a worker picks and executes them; the backend stores the result after execution**. They can use the same Redis server in different databases; nothing forces different ports. The broker does not "track which worker processes which task".

```
# tasks.py
from celery import Celery
import time

celery = Celery("tasks",
                broker="redis://localhost:6379/0",
                backend="redis://localhost:6379/1")

@celery.task
def long_task():
    time.sleep(10)
    return "Task completed"

@celery.task
def add(x, y):
    return x + y
```

Start the worker (in its own terminal, from the folder with `tasks.py`), then call from Python or Flask:

```
celery -A tasks worker -l info          # for a Flask app: celery -A app.celery worker -
l info
```

```
result = add.delay(4, 6)
print(result)            # a task id string such as 9c1f... : it is an AsyncResult, NOT
10
print(type(result))      # <class 'celery.result.AsyncResult'>
print(result.status)     # 'PENDING' right away (worker not finished or not started)
print(result.get(timeout=5))   # 10, waits up to 5 s for the worker
print(result.status)     # 'SUCCESS'
```

SOLVED EXAMPLE: 1. WHAT DOES THIS PRINT?

```
check = add.delay(4, 6)
print(check.status)              # (a)
print(check.get(timeout=5))      # (b)
```

(a) `PENDING`: `.delay()` returns **immediately** with an `AsyncResult`, the work has not finished. (b) `10` after the worker finishes. And for `print(add.delay(4, 6))` you see **a task id**, never 10.

Calling `long_task.delay()` with workers running: the call returns at once; the task executes **asynchronously** and its result `'Task completed'` is available after about 10 seconds. Tasks are **asynchronous by default** (not synchronous), can run on **another machine** than the Flask app, and need a running worker.

delay versus apply_async

`task.delay(a, b)` is a shortcut for `task.apply_async(args=(a, b))`. Use `apply_async` for options:

```
send_email.delay("a@x.com")                                    # send now
send_email.apply_async(args=["a@x.com"], countdown=10)         # run 10 s later
send_email.apply_async(args=["a@x.com"], eta=datetime_utc)     # run at a time
send_email.apply_async(args=["a@x.com"], expires=60)           # drop if not started in
60 s
```

With `countdown=10` the message is sent to the queue **immediately** but the worker starts it only after the 10 seconds have passed (it does not wait 10 seconds before dispatching). `.delay()` dispatches immediately and does not wait for any user input.

Task states

STATE	MEANING
`PENDING`	not started, or **unknown id** (also what you see for a task id that never existed)
`STARTED`	worker began (only recorded if `task_track_started` is on)

STATE	MEANING
`RETRY`	failed and will be retried
`SUCCESS`	finished, `.get()` returns the value
`FAILURE`	raised an exception; `.get()` re-raises it

Idea 5: Celery with Flask, and the app context

The worker is a separate process. Your task often needs the database or `current_app`, which only exist inside a Flask **app context**. The standard pattern (from the Flask docs) makes every task run inside one:

```
# celery_app.py
from celery import Celery, Task
from flask import Flask

def celery_init_app(app: Flask) -> Celery:
    class FlaskTask(Task):
        def __call__(self, *args, **kwargs):
            with app.app_context():
                return self.run(*args, **kwargs)

    celery = Celery(app.name, task_cls=FlaskTask)
    celery.config_from_object(app.config["CELERY"])
    celery.set_default()
    app.extensions["celery"] = celery
    return celery

# app.py
app = Flask(__name__)
app.config["CELERY"] = dict(
    broker_url="redis://localhost:6379/0",
    result_backend="redis://localhost:6379/1",
    timezone="Asia/Kolkata",
)
celery = celery_init_app(app)
```

I ran this pattern with eager mode: a task that returns `current_app.name` returned the app's name, proving the context works. Then start the worker with `celery -A app.celery worker -l info` (module `app`, the variable `celery`). Some course templates instead create the `Celery` object and then set `celery.conf.update(...)`: check your course notes for the exact style. Rules:

- The worker must be able to **import** your tasks (`include=["tasks"]` or put them in the module you pass to `-A`).
- Restart the worker after you change task code; it loads code once.

- Arguments and return values travel as **JSON**, so pass simple things: ids, strings, numbers, **not SQLAlchemy objects** (`job_report.delay(user)` is bad; pass `user.id` and load the user inside the task).
- On **Windows** the default pool does not work well; start the worker with `celery -A app.celery worker -l info -P solo`. WSL or Docker avoids this.

Idea 6: Scheduled (beat) versus on-demand tasks

	ON-DEMAND	SCHEDULED (PERIODIC)
Triggered by	your code calling `.delay()` (a user clicked "export")	the clock, via **Celery beat**
Example	export my data as CSV	daily reminder at 18:00, monthly report on the 1st
Processes needed	Redis + worker	Redis + worker + **beat**

Celery beat is only a scheduler. At the due time it puts a message in the broker and stops. A **worker** must be running to pick it up. Without a worker the morning digest is never sent. (This exact claim appeared as a question: "the worker is optional, beat handles execution" is false.) You start beat separately: `celery -A app.celery beat -l info`. Do not schedule with `@app.task_periodic` (no such thing); a fixed-interval email needs **Celery beat**, no external scheduler.

```
from celery.schedules import crontab

celery.conf.beat_schedule = {
    "daily-reminder": {
        "task": "tasks.send_reminders",
        "schedule": crontab(hour=18, minute=0),              # every day 18:00
    },
    "monthly-report": {
        "task": "tasks.monthly_report",
        "schedule": crontab(minute=0, hour=8, day_of_month=1),  # 1st of month, 08:00
    },
    "every-10-min": {
        "task": "tasks.send_email",
        "schedule": 600.0,                                   # seconds
        "args": ("admin@x.com",),
    },
}
```

I printed these schedules: `crontab(hour=18, minute=30)` shows as `30 18 * * *`; `crontab(minute='*/10')` is every 10 minutes; `crontab(hour=7, minute=0, day_of_week='mon')` is

Mondays at 07:00. The task name is the module path plus function name (`tasks.send_reminders`). Set `timezone` so "18:00" means India time.

YOUR TURN: 1. WHICH PROCESSES DO I START?

Your app has an "export CSV" button and a daily 6 pm reminder. List what must be running.

▸ **Need a hint?**

▸ **Show the full solution**

Idea 7: The async CSV export flow

Problem: exporting all of a user's data takes 20 to 40 seconds. A `fetch('/static/data.csv')` after a fixed `setTimeout` of 41 seconds is a bad design: the job may take longer, and the user waits blindly. The right design is **trigger, build, poll, download**:

1. Browser calls `POST /api/export` . Flask starts the task and returns `202` with a task id.
2. The task builds the file in the background and returns its filename.
3. Browser **short-polls** `GET /api/export/<task_id>` every few seconds until state is `SUCCESS` .
4. Browser then fetches the download link.

```
# tasks.py
import csv, os
from celery_app import celery
from models import Item

@celery.task
def export_items(user_id):
    path = f"exports/items_{user_id}.csv"
    os.makedirs("exports", exist_ok=True)
    with open(path, "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(["id", "name"])
        for it in Item.query.filter_by(user_id=user_id):
            w.writerow([it.id, it.name])
    return path
```

```
# routes
from celery.result import AsyncResult

@app.post("/api/export")
@jwt_required()
def start_export():
```

```
    task = export_items.delay(int(get_jwt_identity()))
    return {"task_id": task.id}, 202

@app.get("/api/export/<task_id>")
@jwt_required()
def export_status(task_id):
    res = AsyncResult(task_id, app=celery)
    if res.state == "SUCCESS":
        return {"state": "SUCCESS", "download": f"/api/download/{task_id}"}
    if res.state == "FAILURE":
        return {"state": "FAILURE"}, 500
    return {"state": res.state}          # PENDING / STARTED / RETRY
```

```
// Vue method: short polling
async startExport() {
  const r = await fetch('/api/export', { method: 'POST', headers: this.authHeader })
  const { task_id } = await r.json()
  const timer = setInterval(async () => {
    const s = await (await fetch('/api/export/' + task_id, { headers: this.authHeader
})).json()
    if (s.state === 'SUCCESS') { clearInterval(timer); this.downloadUrl = s.download }
    if (s.state === 'FAILURE') { clearInterval(timer); this.error = 'Export failed' }
  }, 3000)
}
```

Short polling = asking on a fixed timer; it is the way to learn the **state** of an async task and trigger an action when done. **Long polling** = the server holds the request open until something happens, giving near real-time updates over plain HTTP (so "long polling cannot be done over HTTP" is false). A **webhook** is not short polling: the server **pushes** a message to a URL when an event happens. **Server-Sent Events (SSE)** keep one connection open for the server to push to the browser (one-way; Flask-SSE uses Redis as a shared channel).

SOLVED EXAMPLE: 2. ONE BIG TASK OR MANY SMALL TASKS?

Generate 1000 reports.

Approach 1: `for user in users: job_report.delay(user.id)` creates 1000 small tasks. Approach 2: one task `job_report()` loops over all users.

With **more than one worker**, approach 1 spreads work across workers and finishes **sooner**. With **one worker**, both take about the same time (a single worker runs them one after another; approach 1 only adds queue overhead). Likewise for sending bulk emails: one `send_bulk_emails` task is slower than many `send_email(email)` tasks when more than 2 workers exist, and about equal with a single worker. Many small tasks also fail independently: one bad email does not stop the others.

Idea 8: Retries, idempotency, failure handling, monitoring

```
@celery.task(bind=True, autoretry_for=(ConnectionError,), retry_backoff=True,
max_retries=3)
def send_mail(self, to):
    smtp_send(to)

# manual style
@celery.task(bind=True, max_retries=3)
def send_mail2(self, to):
    try:
        smtp_send(to)
    except ConnectionError as exc:
        raise self.retry(exc=exc, countdown=30)    # try again in 30 s
```

- **Retries:** transient failures (network, SMTP) are retried a limited number of times. After `max_retries` the state becomes `FAILURE`.
- **Idempotency:** a message may be delivered **twice** (worker crash, retry). Design tasks so repeating them is harmless: check "already sent?" before sending, use `INSERT ... ON CONFLICT` or unique keys, write output to a file named by task or user id so a rerun overwrites the same file instead of duplicating.
- **Failure handling:** log exceptions, set a `FAILURE` state, show "Export failed, try again" in the UI; never let the polling loop run forever (set a maximum number of polls).
- **Pass ids not objects**, and re-read the database inside the task so you work on fresh data.
- **Monitoring: Flower** is a web dashboard for Celery (`celery -A app.celery flower`) showing workers, tasks and states. `celery -A app.celery inspect active` lists running tasks.
- **Queue growth:** if tasks arrive faster than workers finish, messages accumulate in the queue; add workers (`-c 4` concurrency, or more worker machines).

WATCH OUT - REAL EXAM TRAP

- `.delay()` returns an `AsyncResult` (prints a task id), not the function's return value.
- `.status` right after `.delay()` is `PENDING`.
- `apply_async(countdown=10)` dispatches immediately and **runs** 10 s later.
- Beat alone does not run tasks: a worker is needed.
- Celery does not need Redis: other brokers work; and Redis is not a relational database (no SQL).
- Passing a SQLAlchemy object as a task argument; use ids.
- `time.sleep()` in a route is not asynchronous; it blocks.

IN YOUR PROJECT

Your project almost certainly needs a **daily reminder** and a **monthly report**, plus an **export** job. In the viva: "Show me the beat schedule", "What happens if the worker is down?" (messages wait in Redis until a worker starts), "Why is export a task?" (it takes long, the request would block), "How does the front end know it finished?" (polling a status endpoint). Keep a one-line start-up recipe: Redis, `flask run`, worker, beat.

YOUR TURN: 2. PREDICT THE BEHAVIOUR

A route calls `report.delay()` and then immediately returns `{"ok": true}`. The Celery worker is not running. What happens to the HTTP response and to the task?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 3. WRITE THE SCHEDULE

Write a beat entry that runs `tasks.weekly_digest` every Monday at 07:30.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 4. REDIS COMMANDS

Predict the output: `RPUSH q a b c`, then `LPUSH q z`, then `LRANGE q 0 -1`, then `SADD s 1 1 2`, then `SCARD s`.

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

About **39 distinct questions** touch this week's keywords, almost all from the **End Term** (roughly 114 of 1471 End Term keyword matches, but only about 3 of 608 in Quiz 2, so this week is an **End Term topic**; counts are approximate and this week has fewer matched questions than week 7). The questions are mostly **concept MCQ/MSQ**, few code questions. They ask:

- **Celery basics**: it is a task queue, runs tasks asynchronously, needs a broker and workers, can schedule tasks; Redis is **not** the only broker; a system may have multiple brokers;
- **delay versus apply_async(countdown=...)**, what `.delay()` returns, `PENDING` then the value (a 2026 paper);
- **broker versus backend** in `Celery(...,broker=...,backend=...)`; the role of Redis in Celery (store scheduled tasks, act as broker);
- **Beat**: scheduling `send_email` every 10 minutes; a 2026 question that beat needs a worker;
- **message broker** theory: decoupling, scalability, O(n) versus O(n squared) connections, spike absorption, topics, slower consumer causes queue growth;
- **polling**: short versus long polling, webhook is not polling; the CSV-file question (poll the task state instead of `setTimeout`); one big task versus many small tasks; why a slow Flask route blocks other users;
- **Redis facts**: in-memory data structure store, cache and broker, pub/sub, not SQL, not strings only.

WATCH OUT - REAL EXAM TRAP

- "Redis is a relational database" and "Redis only stores strings": both false.
- "Message broker lets servers talk directly" is false.
- With one worker, one big task and many small tasks take about the same time.
- Wrong complexity pair: broker links grow as O(n), point-to-point as O(n squared).
- `@app.task_periodic` and `@app.on_periodic` do not exist.

Quick quiz

QUICK QUIZ

What will `long_task.delay()` do, when `long_task` sleeps 10 seconds and then returns 'Task completed', and workers are running?

- Block the caller for 10 seconds
- Return 'Task completed' at once
- Return immediately; the task runs in a worker and the result is available after about 10 seconds
- Fail, because tasks cannot sleep

Check

You want a Celery task `send_email` to run every 10 minutes. How?

- With `@app.task_periodic`
- With `@app.on_periodic`
- With Celery beat and a beat schedule
- Celery cannot do it; use an external scheduler

Check

Select all TRUE statements about Celery and Redis. (select all that apply)

- A Celery system needs worker processes.
- Redis can be a broker and a result backend.
- Redis must be used as the broker.
- Redis stores data with SQL.

Check

In `Celery('tasks', broker='redis://localhost:6379/1', backend='redis://localhost:6379/2')`, what is the difference between broker and backend?

- The backend tracks which worker runs which task
- The broker holds task requests until a worker takes them; the backend stores the result afterwards
- They must run on different ports
- The backend is needed only for email tasks

Check

Which is a TRUE statement about `send_email.apply_async(countdown=10)`?

- The message is sent after waiting 10 seconds

- The message is dispatched immediately and the worker starts the task after the 10 seconds
- It waits for user input
- It runs synchronously

Check

A file is generated by a Celery task in 20 to 40 seconds. What is the best way for the page to fetch it?

- Call `setTimeout` for 41 seconds, then fetch
- Fetch the file in a tight loop forever
- Short-poll the task state every few seconds and fetch the file once it succeeds
- Block the Flask route until the file exists

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- **Why did you use Celery in your project?**
- **Explain the roles of Redis, the worker and the backend.**
- **What is Celery beat and how is it different from a normal task?**
- **How does the front end know that the CSV export is ready?**
- **What happens if the worker is down when a task is sent?**
- **What is idempotency and why does it matter for tasks?**
- **What is Redis and what can it store?**
- **How do you run Redis on Windows?**

Recap

REMEMBER THIS

- Slow work must leave the request: queue it, return `202` quickly, let a worker do it.
- A broker decouples producers and consumers; point-to-point links grow as O(n squared), with a broker as O(n); queues grow if consumers are slow.
- **Redis** is an in-memory data structure store (strings, lists, sets, hashes, sorted sets), usable as broker, result backend and cache; not SQL.
- **Celery**: app (producer), broker, worker, result backend. `@celery.task`, `.delay()`, `.apply_async(countdown=…)`, `AsyncResult`.
- `.delay()` returns an `AsyncResult` at once; `.status` starts `PENDING`; `.get()` waits for the value.
- Flask integration needs the app context inside tasks; start `celery -A app.celery worker -l info` (add `-P solo` on Windows).
- **Beat** + `crontab` schedules periodic tasks, but a **worker** still runs them.
- CSV export: trigger, build in a task, poll status, download. Make tasks idempotent, pass ids, retry transient errors, watch with Flower.

CODE AND FACTS TO KEEP

- `celery = Celery("tasks", broker="redis://localhost:6379/0", backend="redis://localhost:6379/1")`
- `@celery.task` / `@celery.task(bind=True, max_retries=3)`; `raise self.retry(exc=exc, countdown=30)`
- `t.delay(a, b)` is `t.apply_async(args=(a, b))`; options `countdown=`, `eta=`, `expires=`
- `res = AsyncResult(task_id, app=celery)`; `res.state` (`PENDING`, `STARTED`, `RETRY`, `SUCCESS`, `FAILURE`); `res.get(timeout=5)`
- `celery -A app.celery worker -l info`, `celery -A app.celery beat -l info`, `celery -A app.celery flower`
- `crontab(hour=18, minute=0)`; `crontab(minute='*/10')`; `crontab(hour=8, minute=0, day_of_month=1)`; `crontab(hour=7, minute=30, day_of_week='mon')`
- Redis: `SET`, `GET`, `EXPIRE k 60`, `TTL k`, `INCR`, `LPUSH`, `RPUSH`, `LRANGE q 0 -1`, `SADD`, `SMEMBERS`, `HSET`, `HGETALL`, `ZADD`, `ZRANGE`; `redis-cli ping` gives `PONG`
- Windows: `docker run -p 6379:6379 redis` or Redis inside WSL

MAD2 week 10: Messaging: webhooks, server-sent events and email

> **Official syllabus line:** Inter-Service Messaging and Webhooks

The one-minute story

THINK OF IT LIKE THIS

Three ways to hear about a parcel.

Polling is you walking to the gate every five minutes to ask "has my parcel come?". Most trips are wasted.

A webhook is you leaving your phone number with the courier: "call me the moment it arrives." One call, exactly when something happens. The courier (a server) calls *your* number (a server). Nobody walks anywhere.

Server-Sent Events (SSE) is a radio in your room tuned to the courier's channel. You switch it on once and the courier keeps announcing news. You can only listen; you cannot talk back.

A WebSocket is a phone call: both sides can talk at any time.

A message queue is the post office sorting room: a sender drops a letter in a box and goes away; a worker picks it up later. **Email** is the old, reliable letter that every app can send.

This week is about *services telling each other things*, and *the app telling the user things* without the user pressing refresh.

By the end of this week you can:

- explain **synchronous vs asynchronous** messaging, and **queue vs publish/subscribe**;
- write a **webhook sender** (Python `requests`) and a **webhook receiver** (Flask) that checks a signature;
- write a Flask **SSE** endpoint and the browser `EventSource` that reads it, and choose between SSE, WebSockets, polling and long polling;
- send an **HTML email** from Flask and test it with **MailHog**;
- schedule a **daily reminder** and a **monthly report** with Celery beat, and export a **PDF** or **CSV**;
- predict the answer to the typical "which statement about webhooks / SSE is true" question.

Words you need

WORDS YOU NEED

- **Synchronous** - the caller waits for the answer before doing anything else.
- **Asynchronous** - the caller sends the message and carries on; the answer (if any) comes later.
- **Message queue** - a list where producers put messages and workers take them out, one worker per message (Celery with Redis or RabbitMQ).
- **Publish/subscribe (pub/sub)** - a publisher sends to a named channel; every current subscriber of that channel gets a copy. The publisher does not know the subscribers.
- **Message broker** - the server that holds queues or channels (Redis, RabbitMQ, Kafka).
- **Webhook** - an HTTP POST that a server sends to a URL *you* registered, when an event happens. Also called a "reverse API" or HTTP callback.
- **Polling** - the client asks again and again ("short polling") whether there is something new.
- **Long polling** - the client asks once; the server holds the request open until there is news, then answers; the client asks again.
- **SSE (Server-Sent Events)** - one long-lived HTTP GET over which the server keeps pushing text events to the browser. One direction only.
- **WebSocket** - a persistent two-way (full-duplex) connection, started with an HTTP GET handshake and then upgraded.
- **SMTP** - the protocol used to *send* (deliver) email. **IMAP** is used to *read* (retrieve) email.
- **MailHog** - a fake local mail server: it catches every email your app sends so you can look at it in a web page. Nothing is really delivered.
- **HMAC signature** - a hash made with a secret key, sent with a webhook so the receiver can check who sent it.
- **Idempotent** - doing it twice has the same effect as doing it once.

Idea 1: Services talking to each other

A real application is rarely one program. In your project there is a Flask API, a Celery worker, Redis, maybe a mail server. They need to **pass messages**. There are two styles.

	SYNCHRONOUS	ASYNCHRONOUS
Caller	waits for the reply	sends and moves on
Example	`requests.get(url)` in a view; a normal API call	`task.delay()`, a message put in a queue, a webhook
Failure	the caller sees the error at once	the caller may not know; you need retries and logs
Good for	"give me this data now"	slow work, notifications, fan-out to many services

Queue vs publish/subscribe

THE IDEA

Queue (point to point). A producer puts a message in a queue. **Exactly one** consumer takes it. Use it to share *work* among workers. Celery tasks are queue messages.

Publish/subscribe. A publisher sends to a **channel**. **Every** subscriber currently listening gets its own copy. Use it to *broadcast* an event ("order placed") to many services.

In both, the sender does not need to know who receives it, and they communicate **asynchronously**. A **broker** sits in the middle, which also makes the system easy to scale: add more consumers without touching the producer.

Brokers you may hear about: **Redis** (queues, lists and pub/sub, the one used in your course), **RabbitMQ** (a full message broker, a common Celery broker), **Kafka** (a durable log for very high volume). For the course you only need to know what they are for.

Redis pub/sub in a few lines

This needs a Redis server running (see the week 12 lesson for Windows tips). The `redis` package is `pip install redis`.

```
# subscriber.py (run first, in terminal 1)
import redis
r = redis.Redis(host="localhost", port=6379, decode_responses=True)
p = r.pubsub()
p.subscribe("orders")
for msg in p.listen():
    if msg["type"] == "message":
        print("got:", msg["data"])
```

```
# publisher.py (terminal 2)
import redis
r = redis.Redis(host="localhost", port=6379, decode_responses=True)
n = r.publish("orders", "order 42 placed")
print("delivered to", n, "subscribers")
```

Terminal 1 prints `got: order 42 placed`. Terminal 2 prints `delivered to 1 subscribers`. If **no one** is subscribed, the number is `0` and the message is **lost**: Redis pub/sub does not store messages. (A Celery queue, in contrast, keeps the task until a worker takes it.)

WATCH OUT - REAL EXAM TRAP

- "In pub/sub the publisher must know all subscribers" is **false**. It is exactly the thing pub/sub removes.
- "Pub/sub communication is synchronous" is **false**; it is asynchronous.
- A message broker between services makes them **loosely coupled** and easier to scale compared with point-to-point calls.
- A queue gives each message to one worker; pub/sub gives it to all subscribers.

YOUR TURN: 1. QUEUE OR PUB/SUB?

Choose queue or pub/sub: (a) 1000 report-generation jobs shared among 4 workers; (b) one "user signed up" event that must reach the email service, the analytics service and the CRM; (c) a video upload that must be processed once.

▸ **Need a hint?**

▸ **Show the full solution**

Idea 2: Webhooks

THE IDEA

A **webhook** is an **HTTP POST** that service A sends to a URL of service B **when an event happens in A**. B registered that URL earlier. The body (the **payload**) is usually JSON.

- It is **event-driven** and **server to server** (machine to machine).
- It uses the **HTTP** protocol, normally **POST**. It does **not** use WebSockets.
- It is **one way**: A sends, B answers with a small status code (200 means "got it").
- It replaces **polling**: B no longer has to keep asking A "anything new?".
- Webhook receiver and sender do **not** have to be on the same origin (CORS is a browser rule; it does not apply between servers).

Webhook vs API call vs polling

	API CALL	POLLING	WEBHOOK
Who starts it	the client, when it wants	the client, again and again	the **server**, when an event happens
Wasted requests	none	many	none
Delay	none	up to one interval	near real time

"A webhook is triggered by an event, while an API call requires a client request" is the textbook difference. A webhook is **not** the same as short polling, and **not** the same as a WebSocket.

Sending a webhook with Python `requests`

Google Chat and Slack both give you an **incoming webhook URL**. You POST a small JSON message to it and the text appears in a chat room. The URL is a secret: anyone who has it can post. Keep it in an environment variable, never in Git.

```
import os
import requests

URL = os.environ["CHAT_WEBHOOK_URL"]   # looks like
https://chat.googleapis.com/v1/spaces/ ... /messages?key= ...

def notify(text):
    r = requests.post(URL, json={"text": text}, timeout=5)
    r.raise_for_status()               # turns 4xx/5xx into an exception
    return r.status_code

notify("Daily report is ready: 12 users inactive.")
```

`json=` sets the body to JSON **and** the header `Content-Type: application/json`. Google Chat accepts `{"text": "…"}`; Slack incoming webhooks also take `{"text": "…"}`. Always pass a `timeout`, otherwise a dead server can hang your worker forever.

Receiving a webhook in Flask, with a signature check

Anyone on the internet can POST to your receiver URL. How do you know it was really the sender? The sender and you share a **secret**. The sender computes an **HMAC** of the raw body with the secret and sends it in a header. You recompute and compare.

```
import hmac, hashlib
from flask import Flask, request, jsonify

app = Flask(__name__)
SECRET = b"s3cret-shared-key"          # in real code: os.environ

def sign(body: bytes) → str:
    return hmac.new(SECRET, body, hashlib.sha256).hexdigest()

@app.post("/hooks/payment")
def payment_hook():
    body = request.get_data()                        # the RAW bytes
    got = request.headers.get("X-Signature", "")
    if not hmac.compare_digest(got, sign(body)):
```

```
        return jsonify(error="bad signature"), 401
    event = request.get_json()
    print("event:", event["type"])
    return jsonify(ok=True), 200                    # answer quickly
```

I ran this with Flask's test client: a body signed with the right secret gives `200`, a body with no signature gives `401`.

Why these details:

- Sign the **raw bytes** (`request.get_data()`), not a re-dumped JSON, because spaces and key order change the hash.
- `hmac.compare_digest` compares in constant time, so an attacker cannot learn the signature by timing.
- Return `200` **fast** (within a few seconds). Do slow work in a Celery task: `process_event.delay(event)`. Senders give up on slow receivers.
- Use HTTPS, so the secret payload is not readable on the way.

Retries and idempotency

> **THE IDEA**
>
> Networks fail, so a good sender **retries** when it does not get a 2xx answer (often with growing waits: 1 min, 5 min, 30 min). This means your receiver may get the **same event twice**. Webhooks therefore do **not** guarantee exactly-once or in-order delivery. Make your receiver **idempotent**: each event carries a unique `id`; store the ids you have handled and ignore a repeat.

```
processed = set()          # in real code: a database table or Redis set

def handle(event):
    if event["id"] in processed:
        return "duplicate, ignored"
    processed.add(event["id"])
    #  ...  do the work once  ...
    return "done"
```

Calling `handle({"id": "evt_1"})` twice returns `done` then `duplicate, ignored`.

WATCH OUT - REAL EXAM TRAP

- Webhooks **do not** guarantee ordering or exactly-once delivery (a real exam option, marked false). Retries are done by the *sender* and only if it was built to; it is not automatic.
- A webhook is **not** a WebSocket and does not keep a connection open.
- Webhooks are **push** (the source calls you); polling is **pull**. "A webhook pulls messages from an application" is false.
- A webhook receiver is just an ordinary HTTP endpoint, so it needs authentication: signature, a secret token in the URL or header, HTTPS.

YOUR TURN: 2. WHICH SNIPPET IS THE RECEIVER?

Snippet A calls `requests.get(url)` every 10 seconds in a loop. Snippet B defines `@app.post("/hook")` that reads `request.json`. Snippet C does `r.publish("chan", msg)`. Snippet D opens `EventSource`. Which is a webhook receiver, which is pub/sub, which is polling?

▸ **Need a hint?**

▸ **Show the full solution**

IN YOUR PROJECT

Examiners like "how does the daily reminder reach the user?" If you post to a Google Chat webhook as an extra reminder channel, say: the URL is a secret kept in an environment variable; it is called from a Celery task, not from the request; and a failure is logged but does not break the page.

Idea 3: Server-Sent Events (SSE)

Polling asks again and again. A webhook goes server to server. But the **browser** is not a server, it cannot receive a POST. How does a page learn "your export is done" without reloading? One answer is **SSE**.

THE IDEA

SSE keeps **one HTTP connection** open. The browser sends a normal **GET**; the server answers with `Content-Type: text/event-stream` and never finishes: it writes small text blocks whenever it has news.

- **One direction**: server to client only.
- Plain **HTTP** (no special protocol). Works through normal proxies.
- The browser object is `EventSource`; you register an `onmessage` handler.
- It **reconnects automatically** if the connection drops.
- Message format: lines `data: …`, ended by a **blank line** (`\n\n`).

The wire format

```
data: Order placed

data: Preparing

event: done
data: {"status": "delivered"}
```

Each event ends with an empty line. A line `data: x` fires `onmessage`. A line `event: done` gives it a **name**, and the browser must listen with `addEventListener("done", ...)`. An optional `id: 7` line lets the browser tell the server where it stopped (the `Last-Event-ID` header on reconnect). A line `retry: 3000` sets the reconnect delay in milliseconds.

Flask streaming generator

```
import time
from flask import Flask, Response

app = Flask(__name__)

def event_stream():
    for i in range(5):
        yield f"data: Message {i}\n\n"     # note the blank line
        time.sleep(1)

@app.get("/events")
def events():
    return Response(event_stream(), mimetype="text/event-stream")
```

A view that returns a **generator** inside a `Response` streams each `yield` to the client as it is produced. I checked that the response header is `text/event-stream` and the body is the five `data:` blocks.

The browser side (plain JavaScript and Vue 2)

```
const es = new EventSource("http://localhost:5000/events");
es.onmessage = (event) => console.log(event.data);   // "Message 0", "Message 1", ...
es.onerror = () => console.log("lost connection, the browser will retry");
// es.close();   // stop listening
```

```
// inside a Vue 2 component
export default {
  data() { return { messages: [] }; },
```

```
  mounted() {
    this.es = new EventSource("http://localhost:5000/events");
    this.es.onmessage = (e) => { this.messages.push(e.data); };
  },
  beforeDestroy() { if (this.es) this.es.close(); }
};
```

`event.data` is always a **string**: send JSON text and use `JSON.parse(event.data)`.

SOLVED EXAMPLE: 1. PREDICT THE PAGE

Question (the shape of a real End Term question). The Flask code above runs on port 5000 and the Vue page opens an `EventSource` to it. The server yields 5 messages with a one second sleep. What does the user see?

Solution. The browser sends a GET, the server keeps the connection open and flushes one message per second. The page receives **5 messages, one per second** (not all at once, and the server does not crash: Flask can stream). If the page is served from a different origin, the server also needs CORS headers (see week 12), but the question's answer is "5 messages, one per second". After the generator ends the connection closes, and `EventSource` will **reconnect** and start the stream again unless you call `close()`.

A tiny practical note: the real exam code wrote `yield f"data: Message {i}"` without the two newlines. A real browser needs the blank line to fire `onmessage`, so write `\n\n` in your own code.

SSE with many clients and Redis

A generator like the one above is tied to **one** process. If a worker or another request must push an event to many connected browsers, they need a shared **channel**. The **Flask-SSE** extension does this: you call `sse.publish({"msg": "low stock"}, type="alert")` anywhere, and **Redis** acts as the shared channel (pub/sub) that carries the message to every web process holding open connections. Redis here is a message channel, not a login checker, and not mandatory storage of old events. Flask-SSE needs a streaming server (gunicorn with gevent), not the plain development server with many clients.

SSE vs WebSockets vs polling vs long polling

	SHORT POLLING	LONG POLLING	SSE	WEBSOCKET
Protocol	HTTP	HTTP	HTTP	starts as HTTP GET, then upgrades
Direction	client asks	client asks	**server to client**	**both ways** (full duplex)
Connections	a new request each time	one request held open, then repeated	one long connection	one long connection

	SHORT POLLING	LONG POLLING	SSE	WEBSOCKET
Auto reconnect	you write it	you write it	**built in**	you write it
Best for	rare, simple checks	older push emulation	live feeds, progress bars, notifications	chat, games, collaboration

- **Long polling:** the client makes a single request and the **server holds it open until new data exists**, then answers; the client immediately asks again. (Not "repeated requests at fixed intervals": that is short polling.)
- When people just say "polling" they mean **short polling**, over HTTP.
- SSE is **not** an extension of WebSockets and does **not** need service workers or HTTP/2. It cannot send data from the client, but the client can use a normal `fetch` POST for that.
- SSE and the WebSocket handshake both start with an HTTP **GET**.
- A food-delivery tracker (server pushes "Preparing", "Out for delivery") is a **textbook SSE** case: one direction is enough.

YOUR TURN: 3. CHOOSE THE TECHNOLOGY

Pick polling, long polling, SSE, WebSocket or webhook: (a) a live cricket score on a page, server pushes only; (b) a two-player online game; (c) Razorpay-style "payment done" call to your server; (d) a page that checks every minute if a Celery export finished, simplest code.

▸ **Need a hint?**

▸ **Show the full solution**

WATCH OUT - REAL EXAM TRAP

- SSE is **one-way** (server to client); WebSockets are **two-way**. Options that swap them are false.
- SSE uses **GET**, webhooks use **POST**.
- SSE is based on **HTTP**; "SSE switches to WebSocket automatically" is false.
- The browser **does** reconnect automatically in SSE. An answer that says "SSE does not reconnect" is false.
- SSE suits sending updates to **many clients** at once.

Idea 4: Sending email from Flask

Email is the oldest "message to a human". Two protocols: **SMTP** sends (delivers) mail from your app to a mail server; **IMAP** (and POP3) lets a mail program **read** it. Mixing these up is a classic exam trick: "SMTP retrieves email" is false.

MailHog for local testing

Never test with real addresses. **MailHog** is a fake mail server: it listens for SMTP on port **1025** and shows every message in a web page on port **8025** (`http://localhost:8025`). Download its single executable, run it, and point Flask-Mail at it. Nothing is delivered to the outside world.

Flask-Mail

```
pip install flask-mail
```

```
from flask import Flask
from flask_mail import Mail, Message

app = Flask(__name__)
app.config.update(
    MAIL_SERVER="localhost",
    MAIL_PORT=1025,            # MailHog's SMTP port
    MAIL_USE_TLS=False,
    MAIL_USE_SSL=False,
    MAIL_DEFAULT_SENDER="noreply@quizmaster.local",
)
mail = Mail(app)

def send_reminder(to, name):
    msg = Message(
        subject="Your daily reminder",
        recipients=[to],
        body=f"Hi {name}, you have not visited today.",              # plain-text part
        html=f"<h2>Hi {name}</h2><p>You have <b>not</b> visited today.</p>",  # HTML
part
    )
    mail.send(msg)

with app.app_context():
    send_reminder("student@example.com", "Asha")
```

With MailHog running, the message appears in its inbox page. `body` is the plain text version, `html` the rich one; mail programs show the HTML if they can. Flask-Mail needs an **application context** (inside a view, or `with app.app_context()` in a script or Celery task). For real Gmail you would use `smtp.gmail.com` , port 587, `MAIL_USE_TLS=True` , and an **app password** kept in an environment variable (never in code). The standard library `smtplib` does the same job with more lines.

Rendering the HTML body from a Jinja template keeps the code clean:

```
from flask import render_template
html = render_template("email/reminder.html", name="Asha", pending=3)
```

Daily reminders and monthly reports with Celery beat

Sending 500 mails inside a request would freeze the page, so email goes in a **Celery task**, and **Celery beat** (the scheduler) starts it at fixed times.

```
from celery import Celery
from celery.schedules import crontab

celery = Celery("app", broker="redis://localhost:6379/0",
backend="redis://localhost:6379/1")

@celery.task
def daily_reminder():
    # in a real app: query users who have not visited, then mail each
    return "reminders sent"

@celery.task
def monthly_report():
    return "reports sent"

celery.conf.beat_schedule = {
    "daily-6pm": {"task": "app.daily_reminder", "schedule": crontab(hour=18, minute=0)},
    "monthly-1st": {"task": "app.monthly_report",
                    "schedule": crontab(day_of_month=1, hour=8, minute=0)},
}
celery.conf.timezone = "Asia/Kolkata"
```

Run three processes: Redis, a worker (`celery -A app.celery worker -l info` ; on Windows add `-P solo`) and beat (`celery -A app.celery beat -l info`). The task name defaults to `module.function` , here `app.daily_reminder` because the file is `app.py` . A `crontab(hour=18, minute=0)` means "every day at 18:00".

PDF reports and CSV exports

A monthly report is usually HTML turned into a PDF. The pipeline: **data, then Jinja2 template to HTML, then a library renders the PDF**, then the PDF is attached to the email.

```
from flask import render_template
from weasyprint import HTML        # pip install weasyprint (pdfkit needs the
wkhtmltopdf program instead)
```

```
def make_pdf(user):
    html = render_template("report.html", user=user, stats=compute_stats(user))
    HTML(string=html).write_pdf(f"reports/{user.id}.pdf")
```

Attach it with `msg.attach("report.pdf", "application/pdf", pdf_bytes)`.

A **CSV export** needs no extra library:

```
import csv, io
from flask import Response

@app.get("/export.csv")
def export_csv():
    out = io.StringIO()
    w = csv.writer(out)
    w.writerow(["id", "title", "score"])
    w.writerow([1, "Quiz 1", 8])
    return Response(out.getvalue(), mimetype="text/csv",
                    headers={"Content-Disposition": "attachment; filename=export.csv"})
```

`Content-Disposition: attachment` makes the browser **download** the file. For big exports, run it as a Celery task (it can take 20 to 40 seconds), then tell the page when it is done. The page should **poll the task state** every few seconds, or listen via SSE; guessing "wait 41 seconds with `setTimeout`" is **wrong** because the time varies.

SOLVED EXAMPLE: 2. DAILY REMINDER END TO END

Question. Describe, in order, what happens at 18:00 when the daily reminder fires.

Solution.

1. **Celery beat** sees the schedule entry and puts a `daily_reminder` message into the Redis queue.
2. A **Celery worker** takes it, runs the task, queries the database for users who did not visit.
3. For each user the task renders `reminder.html` and sends it with Flask-Mail through SMTP.
4. In development MailHog catches the mails; you open `localhost:8025` and see them.
5. If the worker is down the message waits in Redis, and runs when the worker returns.

YOUR TURN: 4. FIX THE MAIL CODE

A student's view does `for u in User.query.all(): mail.send(make_msg(u))` and the page takes 40 seconds with 1000 users. Fix it.

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 5. SMTP OR IMAP

True or false: (a) SMTP delivers mail; (b) IMAP is used to send mail; (c) MailHog sends your mail to Gmail; (d) Flask-Mail `html=` carries the rich version.

▸ **Need a hint?**

▸ **Show the full solution**

IN YOUR PROJECT

Examiners often ask "show me the daily reminder". Have MailHog open in a tab, trigger the Celery task by hand once (`daily_reminder.delay()`) and show the mail arriving, then show the beat schedule entry. Say which user gets reminded and why (for example "no visit today").

Idea 5: Security and design of messaging endpoints

THE IDEA

Every endpoint that accepts messages from outside is a door. Lock it.

1. **HTTPS** only.
2. **Verify the sender**: HMAC signature header, or a shared token, or an allow-list of IPs.
3. **Validate the payload**: expected fields and types; never trust it.
4. **Answer fast**, process later in Celery.
5. **Idempotent** handling, with event ids.
6. **Secrets** (webhook URLs, SMTP password, signing key) in environment variables, not in Git or the frontend.
7. **Limit** what an event may do; never run SQL built from payload text (SQL injection).

Webhook secrets and SMTP passwords must never reach the browser. This week's exam also showed a login example mixing three mistakes: building SQL by string formatting (**SQL injection**), storing and comparing **plain-text passwords**, and returning no **token**. The right answer there was "all of these". Use parameterised queries, store password **hashes**, and return a token (see the earlier token weeks).

YOUR TURN: 6. WEAK RECEIVER

A receiver does `data = request.json; db.execute(f"UPDATE orders SET s='{data['s']}' WHERE id={data['id']}")` and has no signature check. List three problems and the fixes.

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

This week's evidence file has about **37** distinct matching questions (keyword matching is rough and some are about Vue or JavaScript from neighbouring weeks). Counts by paper: roughly **104 of 1471** End Term questions, **4 of 608** Quiz 2 and **5 of 848** Quiz 1 match this week's keywords, so it is mainly an **End Term topic** and rare in the quizzes. Weeks 10 to 12 have few matched questions, so treat the numbers as a guide only.

What the real questions test, in order of frequency:

- **Webhooks, about 13 questions**: HTTP POST, event driven, server to server, not polling, not WebSocket, no same-origin rule, not guaranteed in order or exactly once, not automatically retried.
- **SSE and WebSockets, about 10**: one way vs two way, HTTP GET, onmessage on EventSource, automatic reconnect, "what happens when the user opens this Flask SSE page", Flask-SSE with Redis as the shared channel.
- **Polling and long polling, about 4**: long polling holds the connection until data exists.
- **Pub/sub and queues, about 4**: publisher need not know subscribers, asynchronous, Celery/RabbitMQ for video processing, "match technologies to uses".
- **Email, about 3**: SMTP sends, IMAP reads.
- **Celery CSV export, about 2**: use polling on the task state, not a guessed delay. One question compared one Celery task per user with one task looping over users (answers were disputed in the key; know both sides: comparable with enough workers, a single big task serialises).

A few Vue and JavaScript questions (`this` in arrow functions, `filter` with `endsWith`, `localStorage` after submit) landed in this file only through the word "email"; they belong to the Vue weeks.

WATCH OUT - REAL EXAM TRAP

- Picking the option that swaps "one way" and "two way" for SSE and WebSockets.
- Thinking webhooks need a polling client or keep a connection open.
- Saying SMTP *retrieves* mail.
- Believing webhooks arrive exactly once and in order.
- Forgetting that SSE and webhooks are both plain HTTP, and that SSE uses GET but a webhook uses POST.

Quick quiz

QUICK QUIZ

Select all TRUE statements about webhooks. (select all that apply)

- They are typically sent as HTTP POST requests.
- They are mainly for server-to-server communication.
- They require the receiver to poll the sender.
- They guarantee ordered, exactly-once delivery.

Check

Which statement about long polling is true?

- The client sends a request every few seconds regardless of news.
- The client sends one request and the server holds it until data is available.
- It uses a WebSocket.
- It lets the server send many events over one request and the client never asks again.

Check

A live order tracker only needs the server to push status changes to the customer. Which choice is best?

- WebSockets, because SSE cannot send JSON.
- Polling every 100 ms.

Server-Sent Events.

A webhook to the customer's browser.

Check

A Flask route returns `Response(gen(), mimetype="text/event-stream")` where `gen` yields `data: hi\n\n` five times with a one second sleep. A page opens `new EventSource(url)`. What happens?

All five messages appear instantly.

The page receives five messages, one per second.

The server crashes.

Nothing, because EventSource cannot use GET.

Check

Select all FALSE statements. (select all that apply)

SMTP is used to retrieve email from a server.

In pub/sub the publisher must know every subscriber.

Pub/sub messaging is asynchronous.

A webhook normally uses HTTP.

Check

In a Flask-SSE setup with several web processes, Redis is used as...

the login checker for connected users.

a shared channel through which published events reach connected clients.

permanent storage that replays every old event.

a CSS cache.

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- What is a webhook and how is it different from polling?
- How would you make sure a webhook request really came from the sender?
- What happens if a webhook is delivered twice?
- When would you use SSE and when WebSockets?
- How does your project send the daily reminder and monthly report?
- Why not send the emails inside the request?
- Difference between a queue and pub/sub?

Recap

REMEMBER THIS

- **Synchronous** waits; **asynchronous** does not. Queues share work; pub/sub broadcasts.
- **Webhook** = event-triggered HTTP POST, server to server, push not pull. Verify with an HMAC, answer fast, be idempotent, expect retries and duplicates.
- **SSE** = one HTTP GET kept open, `text/event-stream`, one way, `EventSource` with `onmessage`, auto reconnect. WebSocket = two way. Long polling = request held open until data.
- **SMTP** sends, **IMAP** reads. Flask-Mail plus **MailHog** (SMTP 1025, web 8025) for local tests; HTML part with `html=`.
- **Celery beat** schedules the daily reminder and the monthly report; the worker does the work.
- PDF = Jinja2 to HTML to WeasyPrint/pdfkit; CSV = `csv` module plus `Content-Disposition`.

CODE AND FACTS TO KEEP

- Webhook send: `requests.post(URL, json={"text": "hi"}, timeout=5)`
- Signature: `hmac.new(SECRET, body, hashlib.sha256).hexdigest()` and `hmac.compare_digest(a, b)`
- SSE response: `Response(gen(), mimetype="text/event-stream")` ; frame = `"data: text\n\n"`
- Browser: `new EventSource(url)` , `es.onmessage = e ⇒ ...` , `es.close()`
- Redis pub/sub: `r.publish("ch", msg)` , `p = r.pubsub(); p.subscribe("ch")`
- Flask-Mail: `Message(subject, recipients=[...], body=..., html=...)` , `mail.send(msg)`
- MailHog: SMTP localhost:1025, inbox `http://localhost:8025`
- Beat: `crontab(hour=18, minute=0)` ; monthly `crontab(day_of_month=1, hour=8, minute=0)`
- CSV download header: `Content-Disposition: attachment; filename=export.csv`
- Next: week 11, performance; week 12, the project; viva: MAD2 viva

MAD2 week 11: Performance: measuring and caching

> **Official syllabus line:** Performance

The one-minute story

THINK OF IT LIKE THIS

A tea stall at a railway station. Every customer orders "one chai". If the stall-owner boils fresh water, adds tea leaves and waits four minutes each time, the queue grows and customers walk away. So the owner makes a big pot in the morning and pours from it. That pot is a **cache**: a ready answer to a question that gets asked again and again.

But the pot gets cold (the answer becomes **stale**), so after an hour the owner throws it away and brews again (**expiry**, or TTL). If the price of sugar changes, he must brew a new pot at once (**invalidation**).

Before you buy a bigger stove, you **measure**: how long does one chai take, how many chai per hour, which step is slow? Maybe the real problem is a customer who keeps asking for 300 cups one at a time (the **N+1 problem**), or a slow tap (a missing **index**). Performance work is: measure, find the bottleneck, fix that one thing, measure again.

By the end of this week you can:

- name the numbers that describe speed (**latency, throughput, percentiles, TTFB, page load**) and the tools that measure them;

- find a slow spot: **N+1 queries**, a **missing index**, a heavy page;
- use **browser caching headers**, **Flask-Caching with Redis**, `@cache.cached` and `@cache.memoize`, and **predict response times** in exam code;
- design cache keys, choose an expiry and decide when to invalidate;
- speed up a Vue front end (lazy loading, compression, debounce) and add **pagination**;
- run a before and after measurement.

Words you need

WORDS YOU NEED

- **Latency** - the time one request takes, from sending to receiving the answer.
- **Throughput** - how many requests per second the server finishes.
- **Percentile (p50, p95, p99)** - p95 = 95 percent of requests were faster than this number. It shows the slow tail that an average hides.
- **TTFB (time to first byte)** - time until the first byte of the response arrives.
- **FCP / TTI / TBT / CLS** - Lighthouse metrics: First Contentful Paint, Time to Interactive, Total Blocking Time, Cumulative Layout Shift.
- **Profiling** - measuring which part of the code takes the time.
- **N+1 queries** - one query for a list plus one extra query for each item.
- **Index** - a lookup structure that lets the database find rows without scanning the whole table.
- **Cache** - stored copies of answers, reused instead of recomputing.
- **Hit / miss / hit ratio** - found in cache / not found / hits divided by all lookups.
- **TTL (time to live)** - seconds after which a cache entry expires.
- **Invalidation** - removing or refreshing a cache entry when its data changed.
- **Memoisation** - caching a function's return value by its arguments.
- **CDN** - Content Delivery Network: servers around the world that deliver your static files from a place near the user.
- **Debounce** - wait until the user stops typing before acting.

Idea 1: What to measure, and with what

"The site is slow" is not a measurement. Pick numbers.

THE IDEA

- **Latency / response time**: one request, in milliseconds.
- **Throughput**: requests per second under load.
- **Percentiles**: report **p50** (median), **p95**, **p99**. An average of 100 ms can hide a p99 of 3 seconds.
- **TTFB**: how long the server (plus network) took to start answering.
- **Page load**: the whole page, shaped mostly by the **number of HTTP requests** and the **size of files** (images, JavaScript).

Lighthouse **Performance** score is built from FCP, TTI (Time to Interactive), TBT, Speed Index and layout shift (CLS). It does **not** include "database query time" or "HTTPS usage".

TOOL	WHAT IT DOES	TYPICAL USE
Browser dev tools, **Network** tab	every request, size, time, headers, cache status	which file is big or slow?
Dev tools, **Performance** tab	flame chart of scripting, rendering	why is the page janky?
Lighthouse (in Chrome dev tools)	scores and advice; can **emulate throttled network and mobile devices**	whole-page audit
Apache Bench `ab`	fires many requests at one URL	simple throughput/latency test
Locust	Python-scripted virtual users	realistic load with logins
`time` / `time.perf_counter()`	stopwatch for a command or code	one function
Flask debug toolbar, `cProfile`	which Python function is slow	profiling

Apache Bench example (100 requests, 10 at a time):

```
ab -n 100 -c 10 http://127.0.0.1:5000/api/books
```

Read `Requests per second` (throughput), `Time per request` (latency) and the `Percentage of the requests served within a certain time` table (percentiles).

A tiny stopwatch in Python:

```
import time

def timed(fn, *a):
    t = time.perf_counter()
    result = fn(*a)
    return result, round((time.perf_counter() - t) * 1000, 1)   # milliseconds
```

YOUR TURN: 1. WHICH NUMBER?

(a) "Most users wait under 400 ms but a few wait 5 s." Which metric exposes this? (b) "The server handled 800 requests in 10 seconds." What is the throughput? (c) Which Lighthouse item is not a performance metric: FCP, TTI, TBT, DQT?

▸ **Need a hint?**

▸ **Show the full solution**

Idea 2: Finding the bottleneck

Do not guess. Measure the layers in order: **network** (dev tools), **server** (profiling, logs), **database** (query count and time), **front end** (scripts).

N+1 queries

You load 3 authors, then for each author you touch `author.books`. That is 1 query for the list plus 3 more: **N+1**. With 1000 authors it is 1001 queries.

```
from sqlalchemy import select
from sqlalchemy.orm import selectinload

# slow: lazy loading, one query per author
for a in db.session.scalars(select(Author)):
    print(a.name, len(a.books))

# fast: load all books for all authors in one extra query
stmt = select(Author).options(selectinload(Author.books))
for a in db.session.scalars(stmt):
    print(a.name, len(a.books))
```

I counted the SQL statements with SQLAlchemy's `before_cursor_execute` event on 3 authors with 2 books each: the first loop ran **4 queries**, the second only **2**. A related fix is a SQL `JOIN`.

Missing indexes

Without an index the database reads **every row** (a "full table scan") to find `WHERE author_id = 7`. An index on that column lets it jump straight to the rows.

```
class Book(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    author_id = db.Column(db.Integer, db.ForeignKey("author.id"), index=True)
    title = db.Column(db.String(100))
```

In SQLite you can ask for the plan: `EXPLAIN QUERY PLAN SELECT * FROM book WHERE author_id = 7;` shows `SEARCH ... USING INDEX` when an index is used and `SCAN` when it is not. Indexes speed up reads but slow down writes a little and take space, so index the columns you filter or join on, not everything.

Other usual suspects

- Returning **too much data** (a full table instead of one page): use **pagination**.
- **Heavy work in the request**: sending mail, generating a PDF, a CSV of 100000 rows. Move it to a Celery task (week 10).
- **Big images and scripts** on the front end.
- Repeating the **same expensive answer** for every user: cache it.

WATCH OUT - REAL EXAM TRAP

- Optimising before measuring. The slow part is often not the one you suspect.
- Judging by the **average**. Look at p95 and p99.
- Testing on your fast laptop with a tiny database. Use realistic data volume.

Idea 3: Cache layers

A request may be answered at several layers. The earlier, the cheaper.

LAYER	WHERE	EXAMPLE	SHARED BY USERS?
Browser cache	the user's browser (a **private** cache)	`Cache-Control: max-age`	no, one per client
CDN / shared cache / reverse proxy	servers between user and you	CDN serving your JS and images	yes
Application cache	memory or Redis, near your Flask code	Flask-Caching	yes
Database cache	the database's own buffers or a stored result	query cache, a summary table	yes

A **CDN** is a group of servers that delivers web content fast, usually from a server near the client (usually, not always). A **shared cache** may be used for many users, so it must not hold **personalised** responses. A **private cache** is tied to one client. A reverse proxy (such as Nginx) can cache responses for similar later requests. A forward proxy protects clients, not servers, so it is a different thing.

The main benefits of caching: lower latency, **lower load on the origin server**, less bandwidth, better handling of traffic spikes (scalability). It does **not** give encryption, DDoS protection or remove the need for the database altogether.

Browser caching headers

The server tells the browser how to cache with response headers.

HEADER	MEANING
`Cache-Control: max-age=3600`	reuse for 3600 seconds without asking
`Cache-Control: no-store`	do not store at all (sensitive data)
`Cache-Control: no-cache`	may store, but **revalidate** before each reuse
`Cache-Control: public` / `private`	shared caches may / may not store it
`Expires`	an older absolute date for expiry
`ETag`	a fingerprint of the content
`Last-Modified`	when the content last changed

Revalidation: the browser sends `If-None-Match: <etag>` (or `If-Modified-Since`). If nothing changed, the server replies **304 Not Modified** with an empty body, and the browser reuses its copy. Cheap, but still one round trip.

```
from flask import Flask, make_response, request, jsonify
app = Flask(__name__)

@app.get("/data")
def data():
    r = make_response(jsonify(x=1))
    r.headers["Cache-Control"] = "public, max-age=60"
    r.add_etag()
    return r.make_conditional(request)
```

I tested this with the Flask test client: the first request returned `200` with an `ETag`; repeating it with `If-None-Match: <that etag>` returned **304** and an empty body.

Related facts that appear in exams: `localStorage` has more space than cookies and keeps data after the browser closes; `sessionStorage` **is cleared when the tab or browser closes**. A **hard refresh** bypasses

the cache for the page's files but does not delete the whole browser cache of the site. Browsers **can** cache JavaScript files, and normally only **GET** requests (without a body) are cached.

YOUR TURN: 2. HEADERS

Which of these control caching: `Cache-Control`, `Expires`, `Bearer`, `Content-Type`, `ETag`? What status code means "your copy is still valid"?

▸ **Need a hint?**

▸ **Show the full solution**

Idea 4: Application caching with Flask-Caching and Redis

```
pip install flask-caching redis
```

```
import time
from flask import Flask
from flask_caching import Cache

app = Flask(__name__)
app.config.update(
    CACHE_TYPE="RedisCache",                # or "SimpleCache" (in-process memory) for
tests
    CACHE_REDIS_URL="redis://localhost:6379/1",
    CACHE_DEFAULT_TIMEOUT=300,              # seconds, applies when a decorator gives
no timeout
)
cache = Cache(app)
```

`SimpleCache` keeps entries in the Python process (lost on restart, not shared between processes), good for learning. `RedisCache` stores them in Redis: shared between workers, survives an app restart.

`@cache.cached` caches the whole view

```
@app.get("/time")
@cache.cached(timeout=5)
def get_time():
    return str(time.time())
```

The **first** request runs the function and stores the response. Requests within 5 seconds get the stored copy: **the function is skipped**. After 5 seconds the entry has expired and the function runs again.

SOLVED EXAMPLE: 1. PREDICT THE TIMESTAMPS

Question (like a real End Term question). `CACHE_DEFAULT_TIMEOUT = 5` and `@cache.cached()` with no argument on `get_time`, which returns the current time. Request 1 at 00:00:00, request 2 at 00:00:02, request 3 at 00:00:06. What does each return?

Solution.

- Request 1: miss, runs, returns `00:00:00` and stores it (expires at 00:00:05).
- Request 2: 2 s later, still fresh: hit, returns `00:00:00`.
- Request 3: 6 s later, expired: miss, runs again, returns `00:00:06`.

Answer: `00:00:00`, `00:00:00`, `00:00:06`. I checked the same logic with `SimpleCache` and a 2 second timeout: two quick calls matched, a call after the timeout differed.

The key matters: `cached` vs `memoize`

THE IDEA

`@cache.cached(timeout=60)` caches a **view**. By default its key is the **URL path** (`request.path`). Two different paths are two different entries. But if you pass `key_prefix='compute'`, the key is that **fixed string**, so different URLs share one entry!

`@cache.memoize(timeout=60)` caches a **function's return value** and builds the key from the **function and its arguments**. `get_data("a")` and `get_data("b")` are separate entries; `get_data("a")` twice is one computation. This is **memoisation**.

```
calls = []

@cache.memoize(timeout=60)
def square(n):
    calls.append(n)
    return n * n

square(3); square(3); square(4)
print(calls)                        # [3, 4]  (the second square(3) was a hit)
cache.delete_memoized(square, 3)    # invalidate only square(3)
square(3)
print(calls)                        # [3, 4, 3]
```

I ran exactly this and got `[3, 4]` and then `[3, 4, 3]`.

SOLVED EXAMPLE: 2. CACHED WITH A KEY_PREFIX: A CLASSIC TRAP

Question.

```
@app.route('/compute/<int:x>/<int:y>')
@cache.cached(timeout=60, key_prefix='compute')
def compute(x, y):
    return jsonify({'result': x * y})
```

`/compute/10/20` is requested, then `/compute/5/10` within 60 seconds. What are the two responses?

Solution. The key is the fixed string `compute`, not the URL. The first call stores `{"result": 200}` under that key. The second URL finds the same key and gets `{"result": 200}` again, although 5 times 10 is 50. So the answers are 200 and 200. Without `key_prefix` (default key from the path) they would be 200 and 50.

SOLVED EXAMPLE: 3. TOTAL TIME WITH MEMOIZE

Question.

```
@cache.memoize(timeout=300)
def compute_data(user_id):
    time.sleep(5)
    return f"Data for user {user_id}"
```

Requests for user ids 1, 2, 1 arrive within 5 minutes. Total time?

Solution. id 1: miss (5 s). id 2: a different argument, miss (5 s). id 1 again: hit (about 0 s). Total **10 seconds**.

Variation: a view `@cache.cached(timeout=40)` that sleeps 30 s, visited by user 1, and by user 2 three minutes later: the entry (40 s) has expired, so user 2 also waits 30 s, and the difference in response times is about **0 seconds**. With `timeout=120` also 3 minutes later, again expired. And "two requests within 50 seconds to a view cached for 100 seconds with a 25 s sleep": the first takes 25 s, the second about 0, so the difference is **25 seconds**. Always compare the gap between requests with the timeout.

A more careful look at how long entries live

- The `timeout` in the decorator overrides `CACHE_DEFAULT_TIMEOUT`.
- A memoize timeout of 180 s with visits at 0, 150 s and 210 s: the first is a miss, the second a hit, the third is only 60 s after the second but 210 s after the **store**, so it is a miss. (TTL counts from when the entry was **stored**, not from the last visit.)
- Expiry is a **time to live** and does not refresh on a hit.

WATCH OUT - REAL EXAM TRAP

- "`@cache.cached` caches by function arguments." No: it caches a view by its **path** (or by `key_prefix`); `memoize` uses the **arguments**. The statement that "memoize does not include parameters in the key" is **false**.
- "Caching removes the need for a database." No, only for repeated lookups.
- A cached **personalised** page (a page showing "Hello Asha") returned to a second user is a bug and a privacy leak. Put the user id in the key, or do not cache it.
- TTL runs from the **store time**.
- Time-based questions: draw a timeline, mark the expiry, and then see which requests land before it.

Idea 5: Cache keys, expiry and invalidation

Key design

A good key is **unique for every different answer** and **identical for every repeat of the same answer**.

ANSWER DEPENDS ON	KEY SHOULD INCLUDE
nothing about the user (a list of public books)	`books:page:2`
one user	`user:7:dashboard`
a filter	`books:genre:sci:page:2`
a score per country	`score:country:India`

Bad keys: too coarse (all users share one entry), or including something that changes every time (a timestamp), which gives a hit ratio of zero.

Expiry (TTL)

Pick it by how stale the data may be. A dashboard that can be 5 minutes old: TTL 300. A social feed or stock price: **short** TTL. Caching a feed for a long time is a bad idea (the question that calls it "recommended" is false). Rarely changing data (list of countries): long TTL.

Invalidation strategies

STRATEGY	HOW	GOOD	BAD
Time-based	just let the TTL expire	simplest	stale until expiry
Delete on write	when data changes, `cache.delete(key)` (or `delete_memoized`)	always fresh next read	you must remember every write path

STRATEGY	HOW	GOOD	BAD
Write-through	on write, update the database **and** the cache	reads stay fast and fresh	more code, two stores to keep consistent

```
@app.post("/books")
def add_book():
    # ... save to the database ...
    cache.delete("view//books")       # or cache.delete_memoized(list_books)
    return {"ok": True}, 201
```

To find the key of a cached view: with the default settings it is `view/` plus the path, for example `view//books` (the double slash is correct, as the path starts with `/`). Easier and safer: use `cache.delete_memoized(function)` for memoised functions, or give views an explicit `key_prefix`.

Hit ratio and stampede (concept)

Hit ratio = hits / (hits + misses). 90 hits and 10 misses is 0.9. A low ratio means bad keys or a TTL that is too short.

Cache stampede: a popular entry expires and 500 requests arrive at the same moment. All see a miss and all hit the database together. Remedies at concept level: one request recomputes while the others wait (a lock), refresh a little before expiry, add a small random amount to each TTL so entries do not all expire together, or warm the cache from a background job.

YOUR TURN: 3. DESIGN THE KEY AND THE INVALIDATION

`GET /scores/countries/<country_name>` returns the total of all players' scores for a country. (a) Where can you cache it? (b) Give a key. (c) When must it be invalidated?

▸ **Need a hint?**

▸ **Show the full solution**

YOUR TURN: 4. PREDICT

```
@cache.cached(timeout=100)
def home():
    time.sleep(30)
    return "hello"
```

User 1 visits. User 2 visits 3 minutes later. What is the difference in their response times? Same question if user 2 visited after 50 seconds.

▸ **Need a hint?**

▸ **Show the full solution**

IN YOUR PROJECT

Examiners ask "what do you cache and what happens when data changes?" Have a one-sentence answer for each cached item, for example: "dashboard summary, key user id, 5 minutes TTL, deleted when the user adds a task". Also demonstrate it: open dev tools or print a timestamp to show the second call is fast. A cache with no expiry and no invalidation will be questioned.

Idea 6: Front-end performance and pagination

Most of the delay a user sees is in the **browser**: many requests, big files, and heavy scripts.

TECHNIQUE	WHAT IT DOES
Lazy loading routes / components	load a page's JavaScript only when the user visits it
Code splitting	the build tool breaks the app into chunks; the initial download is smaller
Minification	strips spaces and shortens names in JS/CSS
Compression (gzip, brotli)	the server sends the file compressed; the browser unpacks it
Image optimisation	right size, modern format (WebP), `loading="lazy"` on images
Fewer requests	bundle small files, use a CDN for libraries
Debounce	wait for a pause in typing before calling the API

Lazy-loaded route in Vue 2 + Vue Router 3 (this needs a build setup such as Vue CLI; it will not work from a CDN `script` tag):

```
const routes = [
  { path: "/", component: Home },
  { path: "/reports", component: () => import("./views/Reports.vue") }   // downloaded
on first visit
];
```

Debounce, so typing "abc" in a search box makes one API call, not three. I ran this in Node:

```
function debounce(fn, ms) {
  let t;
  return (...args) => {
    clearTimeout(t);
    t = setTimeout(() => fn(...args), ms);
  };
}
const search = debounce((q) => console.log("search", q), 100);
search("a"); search("ab"); search("abc");   // quick typing
setTimeout(() => search("abcd"), 300);     // later
```

It printed `search abc` and then `search abcd`. The first three calls collapsed into the last one.

Pagination

Never send 50000 rows. Send a page at a time.

```
@app.get("/api/books")
def books():
    page = request.args.get("page", 1, type=int)
    p = db.paginate(select(Book).order_by(Book.id), page=page, per_page=20,
error_out=False)
    return jsonify(items=[b.title for b in p.items], page=p.page, pages=p.pages,
has_next=p.has_next)
```

With 6 books and `per_page=4`, page 2 holds 2 items, `pages` is 2, `has_next` is False (I ran it). Always add an `order_by` so pages are stable.

Async jobs for heavy work

Anything slower than a second or two (PDF, CSV export, sending mail to everyone) goes to a **Celery** task: the API replies at once with a task id, and the page **polls** the status or is told when it finishes (see week 10).

SOLVED EXAMPLE: 4. BEFORE AND AFTER, A MEASUREMENT EXERCISE

Task. The endpoint `/api/authors` is slow. Improve it and prove it.

1. **Measure before:** `ab -n 200 -c 10 http://127.0.0.1:5000/api/authors`. Write down requests per second and the 95 percent time. Suppose you get 12 requests/s, p95 = 900 ms.
2. **Find the cause:** count queries (the N+1 listener from Idea 2). 1 + 50 queries per request: N+1.
3. **Fix 1:** `selectinload(Author.books)`. Measure again: for example 40 requests/s, p95 = 250 ms.
4. **Fix 2:** add `@cache.cached(timeout=60)`. Measure again: for example 600 requests/s, p95 = 20 ms.
5. **Check correctness:** add an author, confirm the list updates (delete on write or wait 60 s).

Record a small table of before, after 1, after 2 in your report. (The numbers above are only an illustration; your machine gives its own.) One change at a time, so you know what helped.

YOUR TURN: 5. FRONT-END ADVICE

A Vue app loads one 3 MB bundle and a search box calls the API on every key press. List three fixes.

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

The evidence file has about **42** distinct matching questions. Keyword counts: about **134 of 1471** End Term, **12 of 608** Quiz 2 and **6 of 848** Quiz 1 questions match (rough; several matches are really Vue Router questions from neighbouring weeks, and weeks 10 to 12 have few matched questions overall). Honest reading: this is an **End Term topic**, and the pattern is very stable.

What the real questions test:

- **Predict the time with Flask-Caching (about 10 questions).** Code with `@cache.cached(timeout=...)` or `@cache.memoize(timeout=...)`, a `time.sleep(N)`, and two or three visits at given times. You choose "0 seconds", "N seconds" or "difference N". Also the `time.time()` version with request timestamps.
- `@cache.cached` **vs** `@cache.memoize`: view vs function arguments; memoisation definition.
- **Caching in general (about 8)**: benefit (lower server load, lower latency, scalability), shared vs private cache, CDN, what to cache or not, hard refresh, "a browser cannot cache JavaScript" (false).
- **HTTP headers (about 3)**: `Cache-Control`, `Expires`, `ETag`, `no-store`; and the "which statement is false" with `sessionStorage`.
- **Lighthouse (about 3)**: FCP, TTI, Layout Shift count; "Database Query Time" is not one; it can emulate throttling and devices.
- **Page load factors**: number of HTTP requests and file sizes.
- Redis as a cache store; Celery for async work and CSV or reports (week 12 evidence).

WATCH OUT - REAL EXAM TRAP

- Compare the gap between requests with the **timeout**, not with the sleep time.
- `key_prefix` makes different URLs share an entry.
- Different arguments to a memoised function are different entries.
- `no-store` prevents caching; `no-cache` means "revalidate".
- `sessionStorage` does **not** survive closing the browser.
- Caching does not add security.
- A shared cache must not store personalised responses.

Quick quiz

QUICK QUIZ

`@cache.cached(timeout=100)` **wraps a view that sleeps 25 seconds. A client requests the same URL twice within 50 seconds. The approximate difference between the two latencies is...**

0 seconds

25 seconds

- 50 seconds
- 100 seconds

Check

A memoised function with `time.sleep(5)` is called for ids 1, 2, 1 within its timeout. Total time?

- 5 seconds
- 10 seconds
- 15 seconds
- 0 seconds

Check

Select all TRUE statements. (select all that apply)

- `@cache.memoize` builds the key from the function arguments.
- A CDN is a group of servers that delivers content quickly.
- `sessionStorage` data stays after the browser is closed and reopened.
- A shared cache is suitable for personalised responses.

Check

Which is NOT a Lighthouse performance metric?

- First Contentful Paint
- Time to Interactive
- Total Blocking Time
- Database Query Time

Check

With `@cache.cached(timeout=60, key_prefix='compute')` on `/compute/<x>/<y>`, what do `/compute/10/20` then `/compute/5/10` return (multiplication)?

- 200 then 50
- 200 then 200
- 50 then 200
- 100 then 100

Check

Which fixes an N+1 problem?

- Adding `@cache.cached` to a Python `print`
- Loading related rows with `selectinload` or a join
- Using `no-store`
- Increasing the timeout

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- How did you measure your app's performance?
- What did you cache and how do you keep it fresh?
- Difference between `cache.cached` and `cache.memoize`?
- What is a cache stampede?
- How do browser caching headers work?
- Name three things you did to make the front end faster.
- Why is a heavy task like CSV export done in Celery?

Recap

REMEMBER THIS

- **Measure first**: latency, throughput, p95/p99, TTFB. Tools: dev tools, Lighthouse, `ab`, Locust, profilers.
- Common bottlenecks: **N+1 queries**, missing **indexes**, too much data, heavy work inside requests, big front-end files.
- **Cache layers**: browser (private), CDN and proxies (shared), application (Redis), database.
- `@cache.cached` = view, key by path (or `key_prefix`); `@cache.memoize` = function, key by arguments. TTL counts from the store time.
- Keep caches fresh with **TTL**, **delete on write** or **write-through**; watch the **hit ratio** and stampedes.
- Front end: lazy routes, code splitting, minify, gzip, images, debounce. Server: pagination, Celery for heavy jobs.

CODE AND FACTS TO KEEP

- `ab -n 100 -c 10 URL` (requests, concurrency)
- Hit ratio = hits / (hits + misses); throughput = requests / seconds
- `Cache-Control: max-age=60`, `no-store`, `no-cache`, `public`, `private`; `ETag` plus `If-None-Match` gives `304`
- Config: `CACHE_TYPE="RedisCache"`, `CACHE_REDIS_URL="redis://localhost:6379/1"`, `CACHE_DEFAULT_TIMEOUT=300`
- `@cache.cached(timeout=60)`; `@cache.cached(timeout=60, key_prefix="k")`; `@cache.memoize(timeout=60)`
- `cache.delete("key")`, `cache.delete_memoized(fn, *args)`, `cache.clear()`
- N+1 fix: `select(Author).options(selectinload(Author.books))`
- Index: `db.Column(db.Integer, index=True)`
- Pagination: `db.paginate(select(Book).order_by(Book.id), page=2, per_page=20)`
- Lazy route: `component: () ⇒ import("./views/X.vue")`
- Next: week 12, the project; earlier: week 10

MAD2 week 12: The project: putting it all together

> **Official syllabus line:** Project

The one-minute story

THINK OF IT LIKE THIS

Building a small restaurant, not just cooking one dish. Until now you practised single dishes: a Flask route, a Vue component, a Celery task, a cache. The project is the whole restaurant: a kitchen (Flask API and database), a dining room (Vue front end), a waiter who checks reservations (token login and roles), a clock that rings at closing time to tidy up (Celery beat), a pantry with ready-made sauces (Redis cache), and a place to test the fire alarm without a real fire (MailHog).

A restaurant is not judged by how fancy one dish is. It is judged by whether guests are seated, served and billed correctly, and whether the owner can explain how it all works. That last part is your **viva**.

TIP

Honest note on scope. This lesson is a guide, not an official rulebook. The official site lists only the single word "Project" for this week. The **project statement of your term** (published on the course page) is the only authority on what is required and what is optional. The lists below come from the course material and what students report; if the two differ, follow the statement and ask on the course forum.

By the end of this week you can:

- draw the **architecture** of a MAD2 project and say what each part is for;
- plan the **folders, database tables, API routes and Vue structure** for a given statement;
- explain the **token flow** and implement a **role guard** on the server and in the router;
- schedule Celery jobs (daily reminder, monthly report, async CSV export) and cache with expiry;
- run everything locally on Windows, test it, and prepare the README, report and submission;
- check your own project against a rubric and avoid the common mistakes.

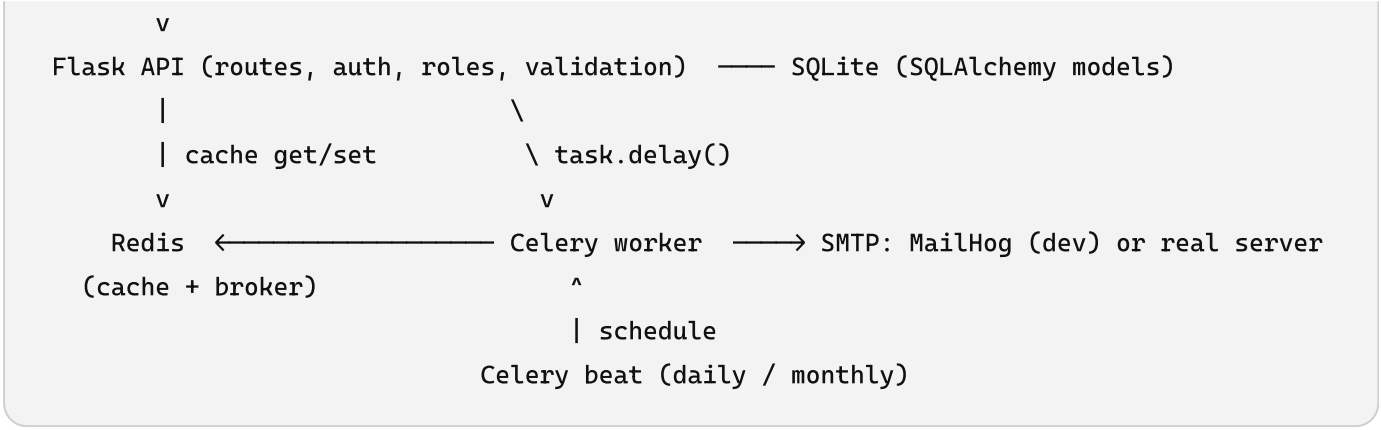
Words you need

WORDS YOU NEED

- **Architecture** - the big picture: which programs exist and how they talk.
- **API-driven** - the backend only serves data (JSON); the front end renders it.
- **SPA (single page application)** - one HTML page; Vue swaps components without full reloads.
- **Token** - a string the server gives after login; the client sends it with every request.
- **RBAC (role-based access control)** - what you may do depends on your role (admin, user...).
- **Guard** - a check that blocks access: a decorator on the server, `beforeEach` in the router.
- **Blueprint** - a way to split a Flask app into modules.
- **OpenAPI / Swagger** - a standard file describing your API routes (YAML), used as documentation.
- **CORS** - the browser rule that blocks a page from one origin from calling another unless the server allows it.
- **Celery worker / beat** - the process that runs tasks / the scheduler that starts them on time.
- **MailHog** - a fake mail server for local testing.
- **WSL** - Windows Subsystem for Linux: a Linux inside Windows, convenient for Redis and Celery.
- **Seed data** - starter rows (the admin user) created on first run.
- **Viva** - the oral examination on your own project.

Idea 1: The architecture of a typical project

```
Browser (Vue 2 SPA: components, Vue Router, Vuex)
       |  HTTP + JSON, header  Authorization: Bearer <token>
```



PART	JOB	TYPICAL TOOLS
Front end	pages, forms, calls the API	Vue 2, Vue Router 3, Vuex 3, `fetch`
Backend	business rules, auth, JSON	Flask, Flask-RESTful or plain routes, Flask-SQLAlchemy
Database	durable data	SQLite
Redis	cache and Celery broker	Redis, Flask-Caching
Celery workers	slow or scheduled jobs	Celery, `delay()`, beat
Mail	reminders, reports	Flask-Mail, MailHog

Core versus optional (as students report)

Typical features reported by students for MAD2 projects:

USUALLY CORE	OFTEN OPTIONAL OR BONUS
Role-based access (for example admin and user)	Scheduled jobs: daily reminder, monthly report
Token login (not only server sessions)	CSV export (async, with a download)
CRUD for the main entities	Caching with expiry
Backend as an **API**, a Vue front end	PDF report, extra charts, search

The point to remember: **read your term's project statement**. Marks follow it. If it makes a feature mandatory, it is mandatory, whatever this table says.

> **THE IDEA**
> A good MAD2 project is **small but complete**: every required feature works end to end, the code is organised, secrets are not in the repository, and you can explain every line. Ten half-working features lose to five solid ones.

YOUR TURN: 1. DRAW THE BOXES

Your statement says: "Librarians add books; members request books; members get a reminder mail every evening if a book is due tomorrow; librarians can export all issued books as CSV." List the processes you need to run on your laptop to demo this.

▸ **Need a hint?**

▸ **Show the full solution**

Idea 2: Folders, database and API plan

Folder structure

```
project/
  backend/
    app.py               # create_app(), registers extensions and blueprints
    config.py            # settings read from environment
    extensions.py        # db, jwt, cache, mail, celery objects
    models.py            # SQLAlchemy models
    routes/
      auth.py            # /api/login, /api/register
      books.py           # CRUD
      admin.py           # admin-only routes
    tasks.py             # Celery tasks and beat schedule
    templates/email/     # reminder.html, report.html
    requirements.txt
  frontend/
    src/
      main.js
      router/index.js
      store/index.js     # Vuex
      api.js             # fetch wrapper that adds the token
      components/ views/
    package.json
  README.md
  openapi.yaml           # API description
  report.pdf
```

Why: `extensions.py` avoids circular imports; routes in blueprints keep each file short; templates for emails keep HTML out of Python code. A simpler layout (a single `app.py`) is fine for a small project, as long as it is tidy.

Database design: design from the statement

Underline nouns in the statement: they become **tables**. Underline "has many": they become **foreign keys**. Take the library statement above.

TABLE	COLUMNS	NOTES
`user`	id, username (unique), email, password_hash, role, last_visit	role is `admin` or `member`
`book`	id, title, author, available (bool)	index on title if you search
`issue`	id, user_id (FK), book_id (FK), issued_on, due_on, returned_on (nullable)	one row per loan; a many-to-many relation turned into a table

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50), unique=True, nullable=False)
    email = db.Column(db.String(120), nullable=False)
    password_hash = db.Column(db.String(200), nullable=False)
    role = db.Column(db.String(10), nullable=False, default="member")

class Issue(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey("user.id"), nullable=False,
index=True)
    book_id = db.Column(db.Integer, db.ForeignKey("book.id"), nullable=False)
    due_on = db.Column(db.Date, nullable=False)
    returned_on = db.Column(db.Date)
```

Store a **hash** (`werkzeug.security.generate_password_hash`) never the password. Create the admin by **seeding** on first run, not through a public register page.

API route plan

Use nouns, the right verb, the right status code (the earlier weeks).

METHOD AND PATH	WHO	SUCCESS	ERRORS
`POST /api/login`	anyone	200 + token	401
`POST /api/register`	anyone	201	400, 409
`GET /api/books?page=1`	logged in	200	401
`POST /api/books`	admin	201	400, 401, 403

METHOD AND PATH	WHO	SUCCESS	ERRORS
`PUT /api/books/<id>`	admin	200	404
`DELETE /api/books/<id>`	admin	204	404
`POST /api/books/<id>/issue`	member	201	409 if unavailable
`POST /api/export`	admin	202 + task id	
`GET /api/export/<task_id>`	admin	200 state / file	

OpenAPI. Describe the same table in `openapi.yaml`; it doubles as documentation for the examiner:

```
openapi: 3.0.0
info: {title: Library API, version: "1.0"}
paths:
  /api/books:
    get:
      summary: List books (paginated)
      parameters:
        - {name: page, in: query, schema: {type: integer}}
      responses:
        "200": {description: A page of books}
        "401": {description: Missing or bad token}
```

SOLVED EXAMPLE: 1. FROM STATEMENT TO PLAN

Statement. "Doctors add available slots; patients book a slot; a patient gets a mail the evening before the appointment."

Solution. Nouns: user (role doctor or patient), slot, booking. Tables: `user`, `slot(id, doctor_id FK, starts_at, booked bool)`, `booking(id, slot_id FK unique, patient_id FK)`. Unique `slot_id` in booking stops double booking; a second attempt returns **409**. Routes: `POST /api/slots` (doctor, 201), `GET /api/slots?page=1` (patient), `POST /api/slots/<id>/book` (patient, 201 or 409). Job: beat at 18:00 mails patients whose slot starts tomorrow.

YOUR TURN: 2. DESIGN TABLES AND ROUTES

Statement: "A quiz platform. Admin creates quizzes with questions. Students attempt a quiz and see their score. Admin sees all scores. Students get a daily mail if they have not attempted any quiz today." Give the tables with keys and eight routes.

▸ **Need a hint?**

▸ **Show the full solution**

Idea 3: Token login, roles and guards

THE IDEA

Token flow.

1. The Vue login form sends `POST /api/login` with username and password (HTTPS in real life).
2. Flask checks the password hash and replies with a **token** (for example a JWT that carries the user id, role and an expiry).
3. Vue stores the token (Vuex, and `localStorage` if the user should stay logged in after reload).
4. Every later request sends the header `Authorization: Bearer <token>`.
5. Flask checks the token on each protected route; a missing or bad token gives **401**; a valid token with the wrong role gives **403**.
6. Logout: delete the token in the client. A token also expires on its own.

401 vs 403: 401 means "I do not know who you are" (no or invalid token). 403 means "I know who you are, and you may not do this".

Server side with Flask-JWT-Extended, including a role decorator (I ran this with the test client: no token gives 401, a student token on the admin route gives 403, an admin token gives 200):

```
from datetime import timedelta
from functools import wraps
from flask import Flask, jsonify, request
from flask_jwt_extended import JWTManager, create_access_token, jwt_required, get_jwt

app = Flask(__name__)
app.config["JWT_SECRET_KEY"] = "change-me-and-read-from-environment"
app.config["JWT_ACCESS_TOKEN_EXPIRES"] = timedelta(hours=1)
JWTManager(app)

@app.post("/api/login")
def login():
```

```
    d = request.get_json()
    user = find_user(d.get("username"))                # your own lookup
    if not user or not check_password(user, d.get("password")):
        return jsonify(error="bad credentials"), 401
    token = create_access_token(identity=user.username, additional_claims={"role":
user.role})
    return jsonify(token=token, role=user.role)

def role_required(role):
    def deco(fn):
        @wraps(fn)
        @jwt_required()
        def wrapper(*a, **k):
            if get_jwt().get("role") ≠ role:
                return jsonify(error="forbidden"), 403
            return fn(*a, **k)
        return wrapper
    return deco

@app.get("/api/admin/stats")
@role_required("admin")
def stats():
    return jsonify(users=2)
```

Your course may use a different token library (for example Flask-Security token authentication). The flow is the same: use what the lectures and the statement ask for.

Vue side: Vuex store, API helper, router guard

```
// store/index.js (Vuex 3)
import Vue from "vue";
import Vuex from "vuex";
Vue.use(Vuex);

export default new Vuex.Store({
  state: {
    token: localStorage.getItem("token") || null,
    role: localStorage.getItem("role") || null
  },
  mutations: {
    setAuth(state, { token, role }) {
      state.token = token; state.role = role;
      localStorage.setItem("token", token); localStorage.setItem("role", role);
    },
    clearAuth(state) {
```

```
      state.token = null; state.role = null;
      localStorage.removeItem("token"); localStorage.removeItem("role");
    }
  },
  actions: {
    async login({ commit }, creds) {
      const res = await fetch("http://localhost:5000/api/login", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(creds)
      });
      if (!res.ok) throw new Error("Login failed");
      commit("setAuth", await res.json());
    }
  }
});
```

```
// api.js
import store from "./store";
export function api(path, options = {}) {
  return fetch("http://localhost:5000" + path, {
    ...options,
    headers: {
      "Content-Type": "application/json",
      Authorization: "Bearer " + store.state.token,
      ...(options.headers || {})
    }
  });
}
```

```
// router/index.js (Vue Router 3)
import VueRouter from "vue-router";
import store from "../store";

const routes = [
  { path: "/login", component: Login },
  { path: "/books", component: Books, meta: { requiresAuth: true } },
  { path: "/admin", component: Admin, meta: { requiresAuth: true, role: "admin" } }
];
const router = new VueRouter({ routes });

router.beforeEach((to, from, next) => {
  if (to.meta.requiresAuth && !store.state.token) return next("/login");
  if (to.meta.role && to.meta.role !== store.state.role) return next("/books");
```

```
  next();
});
export default router;
```

WATCH OUT - REAL EXAM TRAP

- **A client-side guard is only for convenience.** A user can edit `localStorage` and open `/admin`. The **server** must check the role on every protected route. This is the most asked viva question.
- Do not keep the signing secret, the SMTP password or webhook URLs in Vue code, or in Git.
- A token is not encrypted by default (a JWT is only signed): do not put the password in it.
- Do not use `GET` for login with a password in the URL.
- A guard that checks only on page load misses expiry: handle a `401` response by clearing the token and redirecting to login.

SOLVED EXAMPLE: 2. ONE REQUEST THROUGH THE WHOLE SYSTEM

Question. An admin clicks "Export CSV". Trace what happens.

Solution. (1) Vue calls `POST /api/export` with the Bearer token. (2) Flask checks the token and the admin role (else 401 or 403). (3) It calls `export_csv.delay()`, so a message goes into Redis, and replies **202** with a task id. (4) A Celery worker takes the message and writes the file. (5) The page polls `GET /api/export/<id>` every few seconds. (6) On `SUCCESS` the page shows a download link.

YOUR TURN: 3. 401 OR 403

(a) No `Authorization` header on `/api/books`. (b) A student token on `POST /api/books` (admin only). (c) Admin token on `GET /api/books/9999` which does not exist. (d) A student edits `localStorage.role` to `admin` and opens `/admin` in Vue; what stops the data from leaking?

▸ **Need a hint?**

▸ **Show the full solution**

Idea 4: Celery jobs, caching and CORS in the project

Celery jobs

Three typical jobs (details in week 10):

JOB	TRIGGER	NOTES
Daily reminder	beat, `crontab(hour=18, minute=0)`	query who needs it, send an HTML mail
Monthly report	beat, `crontab(day_of_month=1, hour=8, minute=0)`	Jinja2 to HTML (optionally PDF), mail it
Async CSV export	user clicks a button, `export.delay()`	API returns **202** with a task id; page **polls** `GET /api/export/<task_id>` ; then offers the download

Two design questions from past papers: for 1000 users, should the report job be one task per user or one task that loops? One task per user can spread over many workers; one big task is simpler but runs in series. Either can be right; state the trade-off. And for a CSV that takes 20 to 40 seconds, the front end should **poll the task state** (say every 3 to 5 seconds), not wait a guessed fixed time.

```
from celery.result import AsyncResult

@app.post("/api/export")
@role_required("admin")
def start_export():
    task = export_csv.delay()
    return jsonify(task_id=task.id), 202

@app.get("/api/export/<task_id>")
@role_required("admin")
def export_status(task_id):
    r = AsyncResult(task_id, app=celery)
    return jsonify(state=r.state, file="/static/export.csv" if r.state == "SUCCESS"
else None)
```

Caching with expiry

Cache the read endpoints that many users repeat and that tolerate a little staleness, such as a summary or a book list. Use a TTL and delete the key when data changes (see week 11). Say your numbers: "book list, 60 seconds, deleted on add or delete".

CORS

The Vue dev server (`http://localhost:8080`) and Flask (`http://localhost:5000`) are **different origins** (different port), so the browser blocks the call unless Flask allows it. Install `flask-cors` :

```
from flask_cors import CORS
```

```
CORS(app, resources={r"/api/*": {"origins": ["http://localhost:8080"]}})
```

I tested that a request carrying `Origin: http://localhost:8080` receives `Access-Control-Allow-Origin: http://localhost:8080`. Allow only your front end origin, not `*` in the final version. If the browser sends an `Authorization` header, it first sends an `OPTIONS` **preflight**; `flask-cors` answers it.

Idea 5: Run it, test it, document it

Running everything on Windows

PIECE	WINDOWS WAY
Flask	`python -m venv venv`, `venv\Scripts\activate`, `pip install -r requirements.txt`, `flask run`
Vue	`npm install`, `npm run serve`
Redis	not native on Windows: install **WSL** (`wsl --install`), then `sudo apt install redis-server` and `redis-server`; reachable from Windows on `localhost:6379` in most setups
Celery worker	in WSL: `celery -A app.celery worker -l info`. On plain Windows it needs `-P solo` (or `-P threads`)
Celery beat	`celery -A app.celery beat -l info`
MailHog	run the downloaded `.exe`; SMTP 1025, inbox `http://localhost:8025`

If one of them does not start, the demo breaks, so rehearse the start-up order: Redis, MailHog, Flask, worker, beat, Vue. Write it in the README as numbered steps. Keep a `requirements.txt` (`pip freeze > requirements.txt`).

Testing

Test the backend with `pytest` and Flask's test client (no server needed):

```
def test_admin_route_needs_admin(client, student_token, admin_token):
    h = lambda t: {"Authorization": "Bearer " + t}
    assert client.get("/api/admin/stats").status_code == 401
    assert client.get("/api/admin/stats", headers=h(student_token)).status_code == 403
    assert client.get("/api/admin/stats", headers=h(admin_token)).status_code == 200
```

Test login (good and bad password), one CRUD path, one 404, and one validation error (400). Test the Celery task by calling the function directly (`export_csv()`), without a worker.

README and report checklist

- **README:** what the project is, how to install, **exact run order**, default admin login, ports, how to run tests, optional features and where they are.
- **Report** (short; use the template your term gives): statement, architecture diagram, database ER diagram, API list (or `openapi.yaml`), key design decisions, screenshots, how features map to the statement.
- **Never** commit `venv`, `node_modules`, `.env`, real passwords or webhook URLs.

Submission checklist

1. Project statement requirements ticked one by one.
2. Fresh clone works: new virtual environment, `pip install -r requirements.txt`, `npm install`, run order in the README.
3. Seed admin exists; sample data loads.
4. The `.zip` or repository is within the stated size and format; correct file names.
5. Report and screen-recording or links (if the term asks) in the right place before the deadline.
6. You can start every process within two minutes.

YOUR TURN: 4. FIND THE PROBLEMS

A project has: the admin check only in Vue, the SMTP password in `app.py`, the Celery task sending mail for 1000 users inside the `/send` route, no `requirements.txt`, and CORS `origins="*"`. Fix each in one line.

▸ **Need a hint?**

▸ **Show the full solution**

IN YOUR PROJECT

Self-review rubric (score yourself 0, 1 or 2 per row; below 14 of 20 means fix before submitting).

ROW	2 MEANS
Requirements	every item of the statement works end to end
Auth and roles	token login; server checks roles; sensible 401 and 403
API and data	clean routes, status codes, validation, hashed passwords, indexes where used
Front end	components, router, store; loading and error states shown
Background jobs	at least the required jobs run and can be shown on demand
Performance	pagination; cache with TTL and invalidation (if required)
Code quality	folders, naming, no secrets, comments where it is not obvious
Tests	key paths covered; tests pass
Documentation	README, report, API description
Viva readiness	you can explain every file and change something live

Idea 6: A six-week timeline, common mistakes and the viva

A project timeline of 6 weeks

WEEK	GOAL	DONE WHEN
1	Read the statement, list features, design tables and routes	a one-page plan; a repository with folders
2	Backend: models, seed, login, token, roles	login works in a REST client
3	Backend: CRUD, validation, errors, pagination	all required routes pass tests
4	Vue: router, store, login, main pages	a user can log in and use the core flow
5	Redis cache, Celery jobs, mail, CSV export	jobs shown with MailHog; cache has TTL

WEEK	GOAL	DONE WHEN
6	Tests, README, report, rehearse the viva, submit early	fresh-clone test passes

Start week 5 early if scheduled jobs are mandatory; they need the most set-up time.

Common mistakes

- Building features **not in the statement** before the required ones.
- **Secrets in Git** (token key, SMTP password, webhook URL).
- Role checks **only in the front end**.
- Plain-text passwords; SQL built by string formatting.
- Heavy work **inside** a request (mail, PDF, CSV).
- Cache with no expiry and no invalidation; or caching personalised data under a shared key.
- Works only on your laptop (missing `requirements.txt`, hard-coded paths and ports).
- Forgetting CORS, then discovering it on demo day.
- Not understanding code taken from tutorials or generated by tools. The viva will find out.

How this connects to the viva

The project viva is you walking the examiner through your own work. Questions follow your code: the architecture, why this table, how login works, what the cache stores, how the job is scheduled, and "change X live". Practise with the question banks: project viva and MAD2 viva.

YOUR TURN: 5. BE THE EXAMINER

Write five questions an examiner could ask about your own project, and the one-sentence answer for each.

▸ **Need a hint?**

▸ **Show the full solution**

Exam radar

EXAM RADAR

The evidence for this week is thin. About **4** distinct questions matched; keyword counts are roughly **11 of 1471** End Term, **6 of 848** Quiz 1 and **0 of 608** Quiz 2 questions. Weeks 10 to 12 have few matched questions, so this is not a topic with its own question bank: the **project is assessed in the viva and report**, and the written papers test the *ingredients* from other weeks.

What the few matched questions looked like:

- A Vue `v-for` list from the project-style data (**predict what is rendered**).
- A UI state question after a series of clicks on tasks.
- **Celery design**: generating 1000 reports with one task per user versus one task looping over users.
- **CSV generation in Celery** taking 20 to 40 seconds, and how the front end should wait: poll the task state, not a guessed `setTimeout`.

So revise the project ingredients: token auth, roles, Celery and beat, Redis cache, CORS (an origin is scheme, host **and port**: a script from `http://example.com` cannot call `http://new.example.com` unless allowed), plus week 10 and week 11.

WATCH OUT - REAL EXAM TRAP

- Treating the project as "extra". It carries its own marks and the viva.
- Using a fixed delay instead of polling for an async job.
- Forgetting that different subdomains and different ports are different origins.
- Mixing up 401 and 403.

Quick quiz

QUICK QUIZ

A student edits `localStorage` so the Vuex role is `admin` and opens the admin page. The admin API route checks the role claim in the signed token. What happens when the page calls the API?

- The data is shown, because the front end said admin.
- The API returns 403, because the signed token still carries the role member.
- The API returns 404.
- The API returns 200 but with empty data.

Check

Which design is best for an export that takes 20 to 40 seconds?

- Compute it inside the request and increase the browser timeout.
- Wait 41 seconds with `setTimeout` and then fetch the file.
- Start a Celery task, return 202 with a task id, and poll the status until it is SUCCESS.
- Ask the user to refresh until it works.

Check

Which processes does a demo of "daily reminder mail" need running, in addition to Flask?

- Only MailHog.
- Redis, a Celery worker, Celery beat and MailHog.
- Only the Vue server.
- Only SQLite.

Check

Select all statements that are true. (select all that apply)

- A route guard in Vue improves the user experience but is not real security.
- The Vue dev server on port 8080 and Flask on 5000 are different origins, so CORS must be configured.
- A 403 means the token is missing.
- A password may be stored in plain text if the database is local.

Check

Which belongs in `.gitignore` (not in the repository)?

- `requirements.txt`
- `README.md`

`.env` with the SMTP password and `venv/`

`models.py`

Check

Which statement about the project is the most accurate?

This lesson's list of optional features overrides the project statement.

The project statement of the term is the authority on what is required.

Only the front end is marked.

Features beyond the statement always earn extra marks.

Check

Viva: say it out loud

VIVA: SAY IT OUT LOUD

- ▸ **Draw and explain the architecture of your project.**
- ▸ **How does login work?**
- ▸ **How do you stop a normal user from using admin features?**
- ▸ **Why did you design the tables this way?**
- ▸ **What does your Celery beat do, and how can you show it now?**
- ▸ **What do you cache and how do you invalidate it?**
- ▸ **What would you change with more time?**
- ▸ **Change something live: add a field to a model.**

Recap

REMEMBER THIS

- The project is the **whole system**: Vue SPA, Flask API, SQLite, Redis, Celery worker and beat, MailHog.
- **Your term's project statement** decides what is required. Typical core: roles, token login, CRUD, API backend. Typical optional: scheduled jobs, CSV export, caching.
- Design from the statement: **nouns to tables**, relations to foreign keys, routes with verbs and status codes, an OpenAPI file.
- **Server-side role checks** always; client guards are convenience. 401 is unknown, 403 is forbidden.
- Heavy work goes to **Celery**; the page polls task state; caches have **TTL and invalidation**; CORS allows only your front end.
- Prepare a **README with run order**, report, tests, a fresh-clone check and a 6 week plan. Rehearse the viva.

CODE AND FACTS TO KEEP

- Header: `Authorization: Bearer <token>` ; 401 = no or bad token, 403 = not allowed, 409 = conflict, 202 = accepted for background work
- Flask: `create_access_token(identity= … , additional_claims={"role": … })` , `@jwt_required()` , `get_jwt()`
- Vue router guard: `router.beforeEach((to, from, next) ⇒ { … next("/login") })`
- Vuex: `state` , `mutations` (sync), `actions` (async), `commit` , `dispatch`
- CORS: `CORS(app, resources={r"/api/*": {"origins": ["http://localhost:8080"]}})`
- Beat: `crontab(hour=18, minute=0)` , `crontab(day_of_month=1, hour=8, minute=0)`
- Celery on Windows: `celery -A app.celery worker -l info -P solo` ; beat: `celery -A app.celery beat -l info`
- MailHog: SMTP 1025, inbox `http://localhost:8025`
- Run order: Redis, MailHog, Flask, worker, beat, Vue
- `pip freeze > requirements.txt` ; keep secrets in `.env` , never in Git
- Related: week 10, week 11, project viva, MAD2 viva

MAD1 + MAD2: the full-stack map

One app, two courses

THINK OF IT LIKE THIS

Imagine a restaurant. **MAD1** builds the **whole restaurant in one building**: the kitchen (Flask and the database), the menu cards printed fresh for each guest (Jinja2 templates), and a waiter who walks the card to the table (the browser getting a ready-made page). **MAD2** turns it into a **modern chain**: the kitchen becomes a **service window** that only hands out ingredients (a JSON API), a **tablet at each table** (a Vue app in the browser) builds the menu by itself, a **ticket system** remembers who is allowed in (tokens), a **back-room team** prepares big orders without making guests wait (Celery and Redis), and a **warming shelf** keeps popular dishes ready (caching).

Same restaurant, same food, one big idea moved: **who builds the page, the server or the browser.**

The full-stack map

LAYER	MAD1 (SERVER BUILDS THE PAGE)	MAD2 (BROWSER BUILDS THE PAGE)
Talking	HTTP requests and responses (week 1)	the same HTTP, now carrying JSON (week 5)
Page structure	HTML and CSS (week 2)	HTML inside Vue templates (week 4)

LAYER	MAD1 (SERVER BUILDS THE PAGE)	MAD2 (BROWSER BUILDS THE PAGE)
Filling the page	Jinja2 on the server (week 3)	Vue directives in the browser (week 4), components (week 6)
Data	SQLite and SQLAlchemy (week 4)	the same database, hidden behind the API
Logic	Flask routes (week 5)	Flask API plus Vue methods
Contract	REST API and OpenAPI (week 6)	API design, tokens (week 8)
Speed	indexes, FTS, caching idea (week 7)	Redis cache, measuring (week 11)
Interactivity	a little JavaScript, fetch (week 8)	all of it: JavaScript (weeks 1 and 2), Vue, Vuex (week 7)
Safety	sessions, login, attacks (week 9)	tokens, CORS, cookies (week 8)
Quality	pytest (week 10)	tests plus performance checks
Shipping	Git, WSGI, servers (week 12)	Flask + Vue + Redis + Celery together (week 12)
Background work	cron and batch ideas (week 7)	Celery workers and beat (week 9), webhooks and email (week 10)

The recommended order

1. **MAD1 weeks 1 to 6**: build the server-side app. Do not skip week 4 (database design): every later week reuses it.
2. **MAD1 week 8 (JavaScript and fetch)** before MAD2 week 1, even if you have finished all of MAD1: it is the doorway.
3. **MAD1 weeks 7, 9, 10, 11, 12** (backend systems, security, testing, HTML5, deployment): they are the "grown-up" topics that MAD2 builds on.
4. **MAD2 weeks 1 to 3** (JavaScript, async, frontend ideas), then **weeks 4 to 7** (Vue, APIs, components, Vuex), then **week 8** (tokens, API design), then **weeks 9 to 11** (Celery, messaging, performance), and finally **week 12** (the project).

> **TIP**
>
> **Both Quiz 1 and Quiz 2 of each course reuse earlier weeks.** The past papers show that fundamentals such as HTTP, HTML, CSS and SQL keep coming back even in later quizzes, so revise the early weeks on the day before every exam.

Five ideas that cross both courses

THE IDEA

1. Request and response. Everything in both courses is a browser asking and a server answering. If you can say what is in a request (method, URL, headers, body) and a response (status code, headers, body), you can explain 80 percent of the rest.

2. Separation of concerns (MVC). MAD1: models (SQLAlchemy), views (Jinja2), controllers (Flask routes). MAD2 keeps the same split but moves the "view" into the browser (Vue) and keeps "model + controller" in Flask behind the API.

3. Stateless servers. HTTP forgets you after each request. MAD1 fixes this with **sessions and cookies**; MAD2 adds **tokens**. Both are answers to "how does the server know who you are?".

4. Do slow work somewhere else. Indexes (MAD1), cache (MAD1, MAD2), Celery workers (MAD2): all ways of not making the user wait.

5. Trust nothing from the client. Validate on the server, check roles on the server, hash passwords on the server. This is the answer to half of the security and viva questions.

Three problems that cross both courses

YOUR TURN: 1. ONE ROUTE, TWO WORLDS (MAD1 WEEK 5 AND MAD2 WEEK 5)

A Flask route `@app.route("/books")` returns a rendered HTML page in MAD1. (a) What must change so that a Vue app can use the same data? (b) What does the Vue side do with the response?

▸ Need a hint?

▸ Show the full solution

YOUR TURN: 2. SESSION OR TOKEN? (MAD1 WEEK 9 AND MAD2 WEEK 8)

A user logs in. (a) In MAD1 with Flask sessions, what is sent back and what does the browser do next? (b) In MAD2 with a token, what is different?

▸ Need a hint?

▸ Show the full solution

YOUR TURN: 3. MAKE IT FAST (MAD1 WEEK 7 AND MAD2 WEEK 11)

A page lists 50 products and takes 2 seconds because the query scans a large table every time. Give three ways to speed it up, in the order you would try them.

▸ Need a hint?

▸ Show the full solution

Short on time?

THE IDEA

Crash path for the viva (about a week). Day 1: MAD1 weeks 1 and 5 (HTTP, Flask routes). Day 2: MAD1 weeks 3 and 4 (Jinja2, SQL, SQLAlchemy). Day 3: MAD1 weeks 6 and 9 (REST, security). Day 4: MAD2 weeks 4 to 6 (Vue basics, components, router). Day 5: MAD2 weeks 7 and 8 (Vuex, tokens). Day 6: MAD2 weeks 9 and 11 (Celery, caching). Day 7: the project guide and a mock viva.

Project viva guide

What the viva is, and what is (and is not) known

THINK OF IT LIKE THIS

Think of the project viva as a **driving test for the app you built**. The examiner does not care that a car exists; they want to see that **you** can drive it, explain what the pedals do, and handle a small surprise ("please turn left here"). In the viva that surprise is usually: "change one thing in your code, right now."

TIP

Be careful about what is official. The official IIT Madras pages for the project courses (BSCS2003P for MAD1, BSCS2006P for MAD2) describe the skills only. They do **not** publish the viva format, the marking scheme or the deadlines. Everything about levels, marks and rules below comes from **student write-ups and summaries**, and each point is tagged. For your own term, the **project statement and the announcements on the course portal are the only authority**: read them before you trust anything here.

POINT	WHAT WE FOUND	CONFIDENCE
The project is checked in a one-to-one viva	IIT Madras BS programme material	official, general

POINT	WHAT WE FOUND	CONFIDENCE
	mentions project vivas	
MAD1 project: a multi-user Flask app (Jinja2 templates, SQLite, Bootstrap/CSS, secure login) that automates a manual task	student write-ups and the "best projects" site	medium
MAD2 project: Flask API + Vue frontend, token authentication, optional Celery/Redis jobs and caching	official course page text, student repos	high for the skills, medium for details
The viva has two levels: you must clear Level 1 to be taken to Level 2	secondary summary of student documents	medium
One summary of a MAD2 orientation session quotes Level 1 as 40 marks and Level 2 as 60 marks, with a roughly 80 percent originality expectation	an automatic video summary, so it may be garbled	low to medium: **verify**
Viva is scheduled only after you submit; the submission is a one-time upload (a zip, under a size limit)	same orientation summary	medium: **verify**

What examiners seem to check

From student reports (not official), the questions fall into five families:

1. **Explain your project.** What does it do, who are the users, which features are core, how is the code organised?
2. **Explain a piece of your code.** "Open this route. What does `redirect(url_for( ... ))` do? Why POST here and not GET?"
3. **Explain the basics behind it.** HTTP methods and status codes, sessions vs tokens, what an ORM is, what a template is, what cache means, what RBAC means, how hashing works.
4. **Prove it works.** Demo each core feature, with the database visible, with a normal user and an admin.
5. **Change something live.** Add a field, change a rule, add a filter, fix a small bug. This is the family that separates people who built the project from people who copied it.

> **THE IDEA**
>
> **The one rule that decides everything: ownership.** You will be judged on whether the project is **yours**. If you can explain every file, every route and every table without opening the editor first, you will pass the "explain" questions. If you can add a small feature in five minutes, you will pass the "live" questions.

Your two-minute project story (fill in the blanks)

Practise saying this out loud until it is smooth. Do not read it.

```
My project is <name>. It helps <who> to <do what>.
There are <n> kinds of users: <admin / creator / user> and each can <...>.
The stack is Flask for the backend, <Jinja2 + Bootstrap | Vue> for the frontend, SQLite
with <SQLAlchemy> for data.
The main tables are <A>, <B> and <C>; <A> has many <B> (one-to-many), and <B> and <C>
are many-to-many through <link table>.
Login works by <session cookie | token>. Passwords are stored as <hashes>, never as
plain text.
The hardest part was <problem> and I solved it by <idea>.
Extra features: <search / charts / CSV export / daily reminder / caching>.
```

The five-minute demo script

1. Start the app from a clean terminal (virtual environment, one command). It should start with **no errors** and create the database by itself if it does not exist.
2. Log in as **admin**: create something, edit it, delete it. Show that the change is visible in the database (for example with `sqlite3` or a DB browser).
3. Log in as a **normal user** in a private window: show that the user can do their own tasks and **cannot** open an admin page by typing its URL.
4. Show **one validation**: submit an empty form or a duplicate and show the friendly error.
5. Show **search or filter**, then your **extra** feature.
6. For MAD2, additionally: show the API with a tool such as the browser network tab or `curl`, a **token** in the request header, and the **Celery job** or **cache** working (a log line, a CSV file arriving, a repeated request becoming faster).

Checklist before the viva

YOUR TURN: 1. RUN THROUGH IT HONESTLY

Tick each item only if you could demonstrate it right now.

▸ **Need a hint?**

▸ **Show the full solution**

Five live-change drills (do them on your own project)

Time yourself. Each should take under five minutes. Do not use a chatbot.

1. Add a new **column** to one table and show it on the page (and decide how you handle old rows).
2. Add a **filter or search box** to a list page.
3. Make one page **admin-only** and show what a normal user sees.
4. Change a **validation rule** (for example a longer minimum password or no duplicate names) and show the error message.

5. (MAD2) Add one more **API endpoint** and call it from the Vue page; or change the **cache expiry** and show the effect.

WATCH OUT - REAL EXAM TRAP

- **"It worked yesterday."** Run the whole demo on the computer you will use, a day before.
- **Role checks only in the frontend.** An examiner types the admin URL. Always check on the server.
- **Plain-text passwords** or secrets in the code.
- **No delete rules.** Deleting a parent row that has children (a user with bookings, a category with items) crashes or leaves orphans; decide what happens (cascade, block, or soft-delete) and be ready to explain it.
- **Not knowing your own folder structure.**
- **Learning the answers to the question bank but not your project.** The question banks on this site teach the *ideas*; the examiner will ask them about *your code*.

How to use this site for the viva

1. Finish the lessons for the ideas your project uses and do the **"Viva: say it out loud"** box at the end of each lesson.
2. Work through the MAD1 question bank and the MAD2 question bank. Open an answer **only after** you have said yours out loud.
3. Take two or three mock vivas, then revise the questions you marked "need to revise".
4. Rehearse your **two-minute story**, the **five-minute demo** and the **five live-change drills** on your own project.

MAD1 viva question bank (258 questions)

Web basics and HTTP

Q1. What is HTTP and why do we say it is stateless? L1

HTTP is the application-layer request-response protocol the browser uses to talk to a web server. It is stateless because the server treats every request as independent and does not remember earlier ones. To remember a logged-in user we add state on top using cookies and sessions.

Q2. Explain the client-server model with the example of your app. L1

The browser is the client: it sends an HTTP request such as `GET /dashboard`. The Flask server receives it, runs the matching view, talks to the database and returns an HTML response. The client only renders; all data and business rules live on the server.

Q3. What is the difference between GET and POST? L1

GET asks for a resource, puts its parameters in the URL query string, should not change server state and is safe to bookmark or refresh. POST sends data in the request body and is used to create or change something, such as submitting a registration form. Never use GET for actions that modify data, and never put passwords in a URL.

Q4. What does it mean for a method to be idempotent? Which methods are? L1

A method is idempotent if repeating the same request leaves the server in the same state as sending it once. GET, PUT, DELETE, HEAD and OPTIONS are idempotent; POST is not, since two POSTs can create two records. This is why a double-click on a booking button needs separate protection.

Q5. Name the common HTTP status codes and what they mean. L1

200 OK, 201 Created, 204 No Content, 301/302 redirects, 400 Bad Request, 401 Unauthorized (not authenticated), 403 Forbidden (authenticated but not allowed), 404 Not Found, 405 Method Not Allowed, 409 Conflict and 500 Internal Server Error. The first digit gives the class: 2xx success, 3xx redirect, 4xx client error, 5xx server error.

Q6. Walk through what happens when you type a URL and press Enter. L1

The browser resolves the domain to an IP address via DNS, opens a TCP connection (plus a TLS handshake for HTTPS) and sends an HTTP request. The server routes it, runs code, and returns a response with a status line, headers and body. The browser parses the HTML, fetches linked CSS, JS and images, and renders the page.

Q7. Break a URL into its parts. L1

In `https://example.com:8080/books/5?sort=asc#reviews`, `https` is the scheme, `example.com` the host, `8080` the port, `/books/5` the path, `sort=asc` the query string and `reviews` the fragment. The fragment is handled only by the browser and is never sent to the server.

Q8. What are cookies and how do they differ from sessions? L1

A cookie is a small piece of data the server asks the browser to store and send back with every request to that site. A session is the per-user state kept between requests; in Flask the session dict is

stored in a signed cookie, while server-side sessions keep only an id in the cookie. Cookies are the transport, sessions are the idea.

Q9. What is the difference between query parameters, path parameters and form data? L1

Query parameters come after `?` in the URL and are read with `request.args`. Path parameters are part of the route, like `/book/<int:id>`, and arrive as function arguments. Form data is in the POST body and is read with `request.form`.

Q10. What is the Post/Redirect/Get pattern and why use it? L1

After a successful POST the server replies with a redirect (302/303) to a GET page instead of rendering directly. If the user refreshes, only the harmless GET is repeated, so the form is not resubmitted and no duplicate record is created.

```
@app.route("/add", methods=["POST"])
def add():
    db.session.add(Item(name=request.form["name"]))
    db.session.commit()
    return redirect(url_for("list_items"))
```

Q11. What is the difference between HTTP and HTTPS? L1

HTTPS is HTTP carried over TLS. It encrypts traffic so passwords and cookies cannot be read on the network, protects against tampering, and lets the browser verify the server's identity through a certificate. Plain HTTP sends everything readable.

Q12. What is the difference between a static and a dynamic website? L1

A static site serves fixed files that are the same for every visitor. A dynamic site generates the HTML per request using code and a database, for example showing your own bookings. Flask with Jinja2 builds dynamic pages.

Q13. What are server-side and client-side rendering? L1

In server-side rendering the server builds the final HTML, as Jinja2 does, and the browser just displays it. In client-side rendering the server sends mostly data (JSON) and JavaScript builds the DOM in the

browser. MAD1 mainly uses server-side rendering, with JavaScript used for extras.

Q14. What are HTTP headers? Give examples. L1

Headers are key-value metadata sent with requests and responses. Examples are `Content-Type` (format of the body), `Set-Cookie`, `Cookie`, `Location` (for redirects), `Accept` and `Authorization`. They let client and server agree on formats, caching and identity.

Q15. A form submits via GET and the password appears in the URL. How do you fix it? L2

GET puts fields in the query string, which ends up in history, logs and referrers. Change the form to POST and make the route accept POST.

```
<form method="post" action="{{ url_for('login') }}">
```

```
@app.route("/login", methods=["GET", "POST"])
```

Q16. Your route only has methods=['POST'] and visiting it in the browser gives 405. Why, and how do you handle it? L2

Typing a URL makes a GET request, and the route does not allow GET, so Flask returns 405 Method Not Allowed. Add `"GET"` to methods and branch on `request.method`, or serve the form on a separate GET route. 405 is the correct response when the verb is not supported.

Q17. Why is a 302 redirect after login better than rendering the dashboard directly in the POST handler? L2

Rendering in the POST leaves the browser URL as `/login` and a refresh re-sends the credentials. Redirecting to the dashboard sets a clean URL and a GET page, so refresh is safe and the logic of building the dashboard stays in one place.

Q18. When should an API return 401 versus 403? L2

Return 401 when the caller is not authenticated, meaning there is no valid login or token. Return 403 when the caller is identified but lacks permission, for example a normal user opening an admin route. Mixing them up gives clients the wrong signal.

Q19. The browser caches an old version of your CSS. How do you verify and fix it? L2

Open the dev tools Network tab, check whether the CSS request is served from cache, and do a hard reload with the cache disabled. To fix it permanently, change the file name or add a version query like `style.css?v=2` so the browser treats it as a new resource.

HTML and CSS

Q20. What is semantic HTML and why does it matter? L1

Semantic HTML uses tags that describe meaning, such as `header`, `nav`, `main`, `section`, `article` and `footer`, rather than generic `div`s. It helps screen readers, search engines and other developers understand the page structure, and makes code easier to maintain.

Q21. What is the difference between block and inline elements, and between div and span? L1

Block elements like `div`, `p` and `h1` start on a new line and take the full width. Inline elements like `span`, `a` and `strong` flow within text and only take their content's width. `div` is a generic block container and `span` a generic inline container.

Q22. How does an HTML form send data to the server? L1

The `form` tag has `action` (URL) and `method` (get/post). On submit, the browser collects every named control as `name=value` pairs and sends them in the query string or the request body. Controls without a `name` attribute are not sent.

Q23. Why do input fields need the name attribute? L1

The `name` is the key under which the value reaches the server, for example `request.form['email']`. The `id` is only used by CSS, JavaScript and `label for`. A field without a name is silently dropped.

Q24. What is the difference between id and class? L1

An `id` must be unique in a page and identifies one element; a class can be reused on many elements and one element can have several. In CSS, ids are selected with `#` and classes with `.`. Classes are preferred for styling, ids for anchors and JavaScript hooks.

Q25. Explain the CSS box model. L1

Every element is a box made of content, padding, border and margin from inside out. Width and height apply to the content box by default, so padding and border add to the total size. `box-sizing: border-box` makes width include padding and border.

Q26. Describe inline, internal and external CSS and which is best. L1

Inline CSS uses the `style` attribute on a tag, internal CSS sits in a `<style>` block in the head, and external CSS is a separate `.css` file linked with `<link rel="stylesheet">`. External is best for real apps: one file styles many pages and the browser caches it.

```
<link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
```

Q27. What are CSS selectors? Give element, class, id, descendant and child examples. L1

A selector chooses which elements a rule applies to. `p` is element, `.card` class, `#main` id, `nav a` descendant (any a inside nav) and `ul > li` child (direct children only). Pseudo-classes such as `a:hover` target states.

Q28. How do CSS specificity and the cascade decide which rule wins? L1

Inline styles beat ids, ids beat classes (and attributes and pseudo-classes), and classes beat element selectors. If specificity ties, the rule that appears later wins. `!important` overrides these but should be avoided.

Q29. Compare Flexbox and Grid. L1

Flexbox lays items out in one dimension, a row or a column, and is great for navbars and centring. Grid is two-dimensional, controlling rows and columns together, and suits page layouts and card galleries. They can be combined.

Q30. What is responsive design and how do you achieve it? L1

Responsive design makes a page adapt to different screen sizes. Use the viewport meta tag, flexible units like percentages and `rem`, media queries such as `@media (max-width: 768px)`, and a grid

system like Bootstrap's.

Q31. What is the viewport meta tag? L1

`<meta name="viewport" content="width=device-width, initial-scale=1">` tells mobile browsers to use the device width instead of pretending to be a wide desktop. Without it, responsive CSS and Bootstrap breakpoints do not work properly on phones.

Q32. What are the common input types and attributes for validation? L1

Types include `text`, `email`, `password`, `number`, `date`, `checkbox`, `radio` and `file`. Attributes like `required`, `minlength`, `max` and `pattern` give browser-side validation. This only improves user experience; the server must validate again.

Q33. How do you serve an HTML page and a CSS file from Flask? L1

Put templates in a `templates` folder and render them with `render_template`. Put CSS, JS and images in a `static` folder and refer to them with `url_for('static', filename= ... )`. Flask serves `/static/<path>` automatically.

Q34. How do you use browser developer tools while building a page? L1

Right-click and choose Inspect. The Elements panel shows the DOM and the CSS applied, and lets you edit styles live; the Console shows JS errors; the Network tab shows each request, its status, headers and timing. They are the first stop for debugging layout and request problems.

Q35. Centre a div horizontally and vertically using CSS. L2

Use flexbox on the parent with a height.

```
.parent { display: flex; justify-content: center; align-items: center; height:
100vh; }
```

For horizontal centring alone, a block with a fixed width can use `margin: 0 auto`.

Q36. Two rules style the same paragraph and the colour is not changing. How do you debug it? L2

Inspect the element in dev tools; the Styles pane shows crossed-out overridden rules. Check specificity, order of stylesheets and typos in the selector or file path. Fix it by making the intended selector more specific or reordering, rather than adding `!important`.

Q37. Make a table that has alternate row colours and highlights on hover. L2

Use `nth-child` and `:hover`, which need no JavaScript.

```
tr:nth-child(even) { background: #f2f2f2; }
tr:hover { background: #ddd; }
```

Q38. Open a link in a new tab safely. L2

Use `target="_blank"` together with `rel="noopener noreferrer"` so the new page cannot access `window.opener` or see the referrer.

```
<a href="https://example.com" target="_blank" rel="noopener noreferrer">Docs</a>
```

Q39. The form has a checkbox but request.form does not contain it when unchecked. Why? L2

Browsers only send checked checkboxes. An unchecked one has no entry, so `request.form['agree']` raises a 400 error. Use `request.form.get('agree')` or `'agree' in request.form` to read it safely.

Jinja2 and Bootstrap

Q40. What is Jinja2 and how does Flask use it? L1

Jinja2 is a templating engine that mixes HTML with placeholders and simple logic. `render_template('page.html', items=items)` loads the file from `templates`, fills in the variables and returns the final HTML string to the browser.

Q41. What are the three Jinja2 delimiters? L1

`{{ ... }}` prints an expression, `{% ... %}` holds statements like `for`, `if` and `block`, and `{#`

`... #}` is a comment that does not appear in the output.

Q42. Show a for loop and conditional in Jinja2. L1

```
{% for b in books %}
  <li>{{ loop.index }}. {{ b.title }}{% if b.available %} (in stock){% endif %}</li>
{% else %}
  <li>No books</li>
{% endfor %}
```

`loop.index` starts at 1 and the `else` branch runs when the list is empty.

Q43. Explain template inheritance with extends and block. L1

A base template defines the common layout with named blocks; child templates use `{% extends 'base.html' %}` and fill the blocks. Anything outside blocks in a child is ignored. This avoids repeating the navbar and CSS links on every page.

```
{% extends "base.html" %}
{% block content %}<h1>Home</h1>{% endblock %}
```

Q44. What is the difference between include and extends? L1

`extends` makes the current page a child of a layout and overrides its blocks. `include` simply inserts another template's rendered output at that spot, which is good for reusable fragments like a footer or a card.

Q45. What is a macro in Jinja2? L1

A macro is a reusable template function with parameters. Define it once and call it wherever needed, for example to render a form field or a card.

```
{% macro card(title, body) %}<div class="card"><h5>{{ title }}</h5><p>{{ body }}</p>
</div>{% endmacro %}
{{ card("Hi", "Welcome") }}
```

Q46. What are Jinja2 filters? Give examples. L1

Filters transform a value using a pipe: `{{ name|upper }}`, `{{ items|length }}`, `{{ x|default('N/A') }}`, `{{ price|round(2) }}`, `{{ text|truncate(50) }}`. They can be chained. Custom filters can be registered with `@app.template_filter`.

Q47. What is autoescaping and what does the safe filter do? L1

Flask auto-escapes HTML in variables, turning `<` into `<`, so user input cannot inject scripts (XSS). The `|safe` filter marks text as trusted and turns that protection off, so use it only for content you generated yourself, never user input.

Q48. How does url_for work in templates and why use it? L1

`url_for('endpoint', **values)` builds a URL from the view function name. If you later change the route path, links update automatically. It also builds static file URLs: `url_for('static', filename='style.css')`.

Q49. How do flash messages work? L1

`flash('Saved')` in a view stores a message in the session; the next rendered template reads it with `get_flashed_messages()` and shows it once.

```
{% with msgs = get_flashed_messages(with_categories=true) %}
  {% for cat, m in msgs %}<div class="alert alert-{{ cat }}">{{ m }}</div>{% endfor
%}
{% endwith %}
```

Q50. What is Bootstrap and how do you include it? L1

Bootstrap is a CSS and JS framework with a responsive grid and ready components such as navbar, cards, buttons and modals. Include it from a CDN with a link and script tag in the base template, or serve local copies from the static folder.

Q51. Explain the Bootstrap grid system. L1

A `container` holds a `row`, and a row holds up to 12 columns. Classes like `col-12 col-md-6 col-lg-4` set how many columns an element spans at each breakpoint, so layout changes between phone, tablet and desktop.

```
<div class="row"><div class="col-12 col-md-6">A</div><div class="col-12 col-md-6">B</div></div>
```

Q52. CDN versus local Bootstrap files: what are the trade-offs? L1

A CDN is quick to set up and often cached, but needs internet and depends on a third party. Local files work offline and are under your control but you must ship and update them. For a demo that might run offline, local files are safer.

Q53. How do you pass data from a Flask view to a template? L1

Pass keyword arguments to `render_template`; they become variables. Objects work too, so `{{ user.name }}` accesses attributes. A context processor can add variables to every template.

Q54. How would you show the same navbar on every page and highlight the active link? L2

Put the navbar in `base.html` and compare `request.endpoint` to mark the active link.

```
<a class="nav-link {% if request.endpoint == 'home' %}active{% endif %}" href="{{ url_for('home') }}">Home</a>
```

Q55. Create a custom Jinja filter to format a date as DD-MM-YYYY. L2

Register a function with the template_filter decorator.

```
@app.template_filter("dmy")
def dmy(value):
    return value.strftime("%d-%m-%Y")
```

Use it as `{{ booking.date|dmy }}`.

Q56. A user enters <script>alert(1)</script> as their name and it displays literally. Is that correct? L2

Yes, that is autoescaping working: the text is shown as characters instead of running. If it executed, the template must be using `|safe` or building HTML by string concatenation, which is an XSS bug to fix.

Q57. Build breadcrumbs on a child page. L2

Pass a list of (label, url) pairs from the view, or define a block in the base template that children fill.

```
<nav><ol class="breadcrumb">
{% for label, url in crumbs %}<li class="breadcrumb-item"><a href="{{ url }}">{{
label }}</a></li>{% endfor %}
</ol></nav>
```

Q58. Shade alternate table rows using only Jinja. L2

Use `loop.cycle` to alternate a class name.

```
{% for r in rows %}<tr class="{{ loop.cycle('odd', 'even') }}">…</tr>{% endfor %}
```

Q59. The Bootstrap layout looks unstyled on mobile, though it is fine on desktop. What do you check? L2

Check that the viewport meta tag is present, that the Bootstrap CSS link loaded (the Network tab shows no 404), and that column classes include a small-screen class such as `col-12`. A missing viewport tag is the commonest cause.

Databases, SQL and ER diagrams

Q60. Why use a database instead of files for your app data? L1

A database gives structured storage, fast queries with indexes, constraints that keep data valid, transactions for safe concurrent updates and a standard query language. Plain files make searching, relating and protecting data much harder.

Q61. What is SQLite and why is it a good fit for MAD1? L1

SQLite is a serverless, file-based SQL database embedded in the application process. It needs no installation or configuration, comes with Python, and is perfect for a single-machine project. It is less suited to many concurrent writers or large multi-server deployments.

Q62. What are primary key and foreign key? L1

A primary key uniquely identifies each row and cannot be null. A foreign key is a column that refers to the primary key of another table and enforces that the referenced row exists, which is how relationships are stored.

Q63. Explain one-to-many and many-to-many relationships with examples. L1

One-to-many: one user has many bookings, so the foreign key `user_id` sits on the bookings table. Many-to-many: students and courses; you need an association table holding `(student_id, course_id)` with both as foreign keys, optionally with extra columns like enrolled date.

Q64. What is an ER diagram and what does it show? L1

An entity-relationship diagram shows entities (tables), their attributes, and the relationships between them with cardinality such as 1:N or M:N. It is the design blueprint you draw before creating tables.

Q65. What is normalization? Explain the first three normal forms briefly. L1

Normalization organises tables to reduce redundancy and update anomalies. 1NF: each cell holds one atomic value. 2NF: no partial dependence of non-key columns on part of a composite key. 3NF: no non-key column depends on another non-key column.

Q66. What are ACID properties? L1

Atomicity: a transaction fully happens or not at all. Consistency: it moves the database from one valid state to another. Isolation: concurrent transactions do not see each other's partial work. Durability: committed data survives crashes.

Q67. Write SQL for CREATE TABLE with constraints. L1

`NOT NULL`, `UNIQUE`, `PRIMARY KEY` and `REFERENCES` are constraints enforced by the database.

```
CREATE TABLE booking (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  user_id INTEGER NOT NULL REFERENCES user(id),
  slot TEXT NOT NULL,
  UNIQUE (user_id, slot)
);
```

Q68. Explain INNER JOIN and LEFT JOIN with an example. L1

INNER JOIN returns only rows that match in both tables; LEFT JOIN returns all rows from the left table with NULLs when there is no match. The query below lists every user including those with zero bookings.

```
SELECT u.name, COUNT(b.id)
FROM user u LEFT JOIN booking b ON b.user_id = u.id
GROUP BY u.id;
```

Q69. What is the difference between WHERE and HAVING? L1

WHERE filters rows before grouping, HAVING filters groups after aggregation. For example `GROUP BY user_id HAVING COUNT(*) > 3` keeps only users with more than three rows.

Q70. What is the difference between DELETE, TRUNCATE and DROP? L1

DELETE removes selected rows and can be rolled back in a transaction. DROP removes the whole table and its structure. TRUNCATE empties a table quickly in other databases, but SQLite has no TRUNCATE; you use `DELETE FROM table`.

Q71. What is a transaction and when do you need one? L1

A transaction groups several statements so they commit together or roll back together. You need it when steps depend on each other, for example reducing stock and inserting an order; if one fails, both must be undone.

Q72. What is an index and what is its downside? L1

An index is a sorted structure (a B-tree in SQLite) on one or more columns that makes lookups and sorting faster. The cost is extra disk space and slower inserts and updates, since the index must be maintained.

Q73. What does ON DELETE CASCADE do in SQLite and what must be enabled? L1

It automatically deletes child rows when the parent row is deleted. SQLite does not enforce foreign keys unless you run `PRAGMA foreign_keys = ON` for each connection, so cascades silently do nothing without it.

Q74. How do you design an ER diagram from a problem statement? L1

Underline the nouns to get entities, decide attributes and a primary key for each, then find relationships between entities and their cardinality. Resolve M:N with an association table, then check for redundancy using normalization.

Q75. Find the top three users by number of bookings. Write the SQL. L2

Group by user, count, sort descending and limit.

```
SELECT user_id, COUNT(*) AS n
FROM booking
GROUP BY user_id
ORDER BY n DESC
LIMIT 3;
```

Q76. You need to store a list of tags per post. Why not a comma separated column? L2

A comma separated value breaks first normal form: you cannot index, search or join on individual tags, and constraints cannot be applied. Use a separate `tag` table and a `post_tag` association table.

Q77. How do you find rows in table A that have no match in table B? L2

Use a LEFT JOIN and filter NULLs, or NOT EXISTS.

```
SELECT a.* FROM a LEFT JOIN b ON b.a_id = a.id WHERE b.id IS NULL;
```

Q78. Add a column to an existing SQLite table without losing data. L2

Use ALTER TABLE. Note that `create_all()` will not alter existing tables, so in a real app use a migration tool such as Flask-Migrate.

```
ALTER TABLE user ADD COLUMN phone TEXT;
```

SQLAlchemy and Flask-SQLAlchemy

Q79. What is an ORM and why use SQLAlchemy? L1

An ORM maps Python classes to tables and instances to rows, so you write Python instead of SQL strings. SQLAlchemy also builds parameterised queries, which prevents SQL injection, and makes switching databases easier.

Q80. Explain engine, session and model in SQLAlchemy. L1

The engine manages the connection to the database. A session is a workspace that tracks objects, collects changes and sends them in a transaction on commit. A model is a class mapped to a table whose attributes are columns.

```
engine = create_engine("sqlite:///app.db")
Session = sessionmaker(bind=engine)
```

Q81. How do you define a model in Flask-SQLAlchemy? L1

Subclass `db.Model` and declare columns with `db.Column`; relationships are declared with `db.relationship`.

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
```

```
    email = db.Column(db.String(120), unique=True, nullable=False)
    bookings = db.relationship("Booking", backref="user")
```

Q82. How do you configure Flask-SQLAlchemy and create tables? L1

Set `SQLALCHEMY_DATABASE_URI`, create `db = SQLAlchemy(app)` and call `db.create_all()` inside an app context. `create_all` creates only missing tables and never alters existing ones.

```
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///app.db"
db = SQLAlchemy(app)
with app.app_context():
    db.create_all()
```

Q83. What is the difference between flush, commit and rollback? L1

`flush` sends pending SQL to the database inside the open transaction without ending it. `commit` flushes and makes the transaction permanent. `rollback` discards uncommitted changes, and is needed after an error.

Q84. Show common queries: get by primary key, filter, order, count. L1

`first()` returns None when nothing matches, while `first_or_404()` aborts with 404.

```
db.session.get(User, 1)
User.query.filter_by(email=e).first()
Booking.query.filter(Booking.price > 100).order_by(Booking.date.desc()).all()
Booking.query.filter_by(user_id=1).count()
```

Q85. How do you define a relationship and what do backref and back_populates do? L1

`relationship` creates a Python-level link between models using the foreign key. `backref` automatically adds the reverse attribute on the other class, while `back_populates` declares both sides explicitly. The foreign key column is still required.

Q86. Explain cascade options such as delete-orphan. L1

Cascade controls what happens to related objects when the parent changes. `cascade='all, delete-orphan'` deletes children when the parent is deleted or when a child is removed from the parent's collection. It acts in Python; database-level `ON DELETE` is separate.

Q87. What is lazy loading and the N+1 problem? L1

By default related rows load lazily on first access, so looping over 100 users and reading `user.bookings` issues 1 query plus 100 more. Fix it with eager loading using `joinedload` or `selectinload`.

```
from sqlalchemy.orm import joinedload
User.query.options(joinedload(User.bookings)).all()
```

Q88. How do you do a join and an aggregate with SQLAlchemy? L1

`func.count`, `func.sum` and `func.avg` map to the SQL aggregates.

```
from sqlalchemy import func
rows = (db.session.query(User.name, func.count(Booking.id))
        .join(Booking).group_by(User.id).all())
```

Q89. How does SQLAlchemy protect against SQL injection? L1

It sends user values as bound parameters, separate from the SQL text, so input is never interpreted as SQL. The danger returns if you build raw strings with f-strings; for raw SQL use `text()` with bind parameters.

Q90. How do you ensure uniqueness at the database level? L1

Add `unique=True` on the column or a `UniqueConstraint` for a combination. The database then rejects duplicates with an IntegrityError even under race conditions, which a check in Python alone cannot guarantee.

Q91. What is Flask-Migrate and what do upgrade and downgrade do? L1

Flask-Migrate wraps Alembic to version-control schema changes. `flask db init` creates the migrations folder, `flask db migrate` autogenerates a script from model changes, `flask db upgrade` applies it and `flask db downgrade` reverts the latest one. It changes the schema without dropping your data.

Q92. Add a new column to a model without losing data. L2

Edit the model, then generate and apply a migration. Check the generated script first; for SQLite a non-null new column needs a default.

```
flask db migrate -m "add phone"
flask db upgrade
```

Q93. Handle a duplicate email on registration safely. L2

Catch the IntegrityError, roll back and tell the user.

```
from sqlalchemy.exc import IntegrityError
try:
    db.session.add(User(email=email)); db.session.commit()
except IntegrityError:
    db.session.rollback(); flash("Email already used")
```

Q94. Soft delete: implement it so deleted records are hidden but kept. L2

Add a boolean column and filter by it. Deleting becomes `item.is_deleted = True` followed by commit.

```
is_deleted = db.Column(db.Boolean, default=False)
Item.query.filter_by(is_deleted=False).all()
```

Q95. Queries raise 'PendingRollbackError' after an earlier failure. What happened? L2

A previous flush failed and the session is still in a failed state. Call `db.session.rollback()` in the except block so the session can be used again. Always rollback after a failed commit.

Q96. Why does 'working outside of application context' appear when running db.create_all() in a script? L2

Flask-SQLAlchemy needs the app context to know which app and database to use. Wrap the call in `with app.app_context():` or run it from the Flask shell.

Q97. Change the type of a column in a live project. What are the risks? L2

`create_all` will not change existing tables, and SQLite supports limited ALTER. Use a migration with batch mode (`render_as_batch=True`), which recreates the table and copies data, and back up the database first.

Flask and MVC

Q98. What is Flask and what are its main components? L1

Flask is a lightweight Python web microframework built on Werkzeug (WSGI and routing) and Jinja2 (templates). It provides routing, request and response objects, sessions and templates, and leaves choices like database and forms to extensions.

Q99. Explain MVC and map it to your Flask project. L1

Model holds data and rules (SQLAlchemy classes), View is what the user sees (Jinja templates), Controller handles requests and ties them together (Flask route functions). Flask itself calls the controller a view function, so some people describe it as MVT; the separation of concerns is the same.

Q100. What is a route and how do you define one with variables and methods? L1

A route maps a URL pattern to a function with a decorator. Converters like `int` , `string` and `path` validate and convert the URL part.

```
@app.route("/book/<int:book_id>", methods=["GET", "POST"])
def book(book_id):
    ...
```

Q101. What is the difference between request.form, request.args and request.json? L1

`request.form` has POST form fields, `request.args` has query-string parameters and `request.get_json()` parses a JSON body. Use `.get()` to avoid a 400 when a key is missing.

Q102. How do redirect and url_for work together? L1

`url_for('endpoint')` builds the URL and `redirect(url)` returns a 302 response with a Location header. Together they let you move to another page without hard-coding paths.

Q103. What is the Flask session and why does it need a SECRET_KEY? L1

`session` is a dict-like object stored in a cookie that Flask signs with `SECRET_KEY`, so the user cannot alter it undetected. It is signed, not encrypted, so never store secrets in it. Without a key Flask raises an error when you use the session.

Q104. What are Blueprints and when would you use them? L1

A Blueprint groups related routes, templates and static files into a module that is registered on the app. It keeps larger projects organised, for example separate `auth`, `admin` and `api` blueprints.

Q105. What is an application factory? L1

A function `create_app(config)` that builds and returns the app and initialises extensions. It allows different configs for testing and production and avoids circular imports.

Q106. Explain error handlers and abort. L1

`abort(404)` raises an HTTP exception immediately. `@app.errorhandler(404)` registers a function that returns a custom page for that error.

```
@app.errorhandler(404)
def not_found(e):
    return render_template("404.html"), 404
```

Q107. What do before_request, after_request and the g object do? L1

`before_request` functions run before each view, for example to load the current user. `after_request` runs after the view and can modify the response. `g` stores data for the duration of one request only.

Q108. What is a decorator and how would you write login_required? L1

A decorator is a function that wraps another function to add behaviour.

```
from functools import wraps
def login_required(f):
    @wraps(f)
    def wrapper(*a, **kw):
        if "user_id" not in session:
            return redirect(url_for("login"))
        return f(*a, **kw)
    return wrapper
```

Q109. Describe a typical Flask project structure. L1

A top-level `app.py` or package with `models.py`, routes (or blueprints), a `templates` folder with `base.html`, a `static` folder for css, js and images, `config.py`, `requirements.txt` and the SQLite database file. Separate files keep models, logic and views apart.

Q110. How do you handle a file upload in Flask? L1

Use a form with `enctype="multipart/form-data"`, read `request.files['file']`, sanitise the name with `secure_filename`, check the extension and size, and save to an upload folder.

Q111. What does debug=True do and why avoid it in production? L1

It enables the auto-reloader and the interactive debugger. The debugger lets anyone who reaches an error page execute code on the server, so it must be off in production.

Q112. Add a new page /about that renders about.html. What are the steps? L2

Create `templates/about.html` extending the base, add a route, and link to it in the navbar with `{{ url_for('about') }}`.

```
@app.route("/about")
def about():
    return render_template("about.html")
```

Q113. A route works for /items but /items/ gives 404. Why? L2

Flask treats a rule defined with a trailing slash (`/items/`) as a folder and redirects `/items` to it, while a rule defined without the slash does not match `/items/` . Choose one form consistently, or set `strict_slashes=False` .

Q114. Restrict a page to admins only. L2

Make a decorator that checks the role and aborts with 403.

```
def admin_required(f):
    @wraps(f)
    def w(*a, **k):
        if session.get("role") ≠ "admin":
            abort(403)
        return f(*a, **k)
    return w
```

Q115. Two routes use the same function name and Flask raises an AssertionError. Why? L2

Flask uses the function name as the endpoint and endpoints must be unique. Rename one function or give an explicit `endpoint=` argument. This also bites decorators that forget `functools.wraps` , because every wrapped function then gets the name `wrapper` .

Q116. Read the logged-in user's id from the session and show only their records. L2

Filter the query by the session id rather than any id given in the URL.

```
uid = session["user_id"]
bookings = Booking.query.filter_by(user_id=uid).all()
```

REST APIs and OpenAPI

Q117. What is a REST API? L1

REST is an architectural style where resources are identified by URLs and manipulated with standard HTTP methods, with stateless requests and usually JSON representations. For example `GET /api/books/5` returns book 5.

Q118. How do CRUD operations map to HTTP methods and status codes? L1

Create is POST (201 Created), Read is GET (200), Update is PUT for full replacement or PATCH for partial (200), and Delete is DELETE (204 or 200). Errors use 400, 404 or 409 as appropriate.

Q119. What are good REST URL design rules? L1

Use plural nouns for resources, not verbs, and nest only when meaningful: `/api/users/5/bookings` rather than `/getBookings`. Use query strings for filtering and sorting, and version the API like `/api/v1/`.

Q120. What is OpenAPI and what does an OpenAPI YAML contain? L1

OpenAPI is a machine-readable standard to describe a REST API: servers, paths, methods, parameters, request bodies, responses and schemas. Tools like Swagger UI render it as interactive docs and can generate client code and tests.

```
paths:
  /api/books/{id}:
    get:
      responses:
        "200": {description: A book}
        "404": {description: Not found}
```

Q121. What is Flask-RESTful and how does a Resource differ from a plain route? L1

Flask-RESTful lets you write a class per resource with methods named `get`, `post`, `put` and `delete`; the HTTP method picks the function. You register it with `api.add_resource`.

```
class BookAPI(Resource):
    def get(self, id):
        ...
api.add_resource(BookAPI, "/api/books/<int:id>")
```

Q122. What is reqparse used for? L1

`reqparse.RequestParser` declares expected request arguments with type, required flag and help message and validates them, returning 400 with an error message if they are wrong.

```
parser = reqparse.RequestParser()
parser.add_argument("title", type=str, required=True, help="title needed")
args = parser.parse_args()
```

Q123. What does marshal_with do? L1

It formats the returned object according to a dict of fields, converting SQLAlchemy objects into JSON with only the chosen fields.

```
book_book_fields = {"id": fields.Integer, "title": fields.String}
@marshal_with(book_fields)
def get(self, id):
    return Book.query.get_or_404(id)
```

Q124. How do you return a custom status code from a Flask-RESTful resource? L1

Return a tuple of data and status, or raise `abort(404, message="Not found")` from flask_restful.

```
return {"message": "created"}, 201
```

Q125. What is the difference between PUT and PATCH? L1

PUT replaces the whole resource and sends all fields; PATCH changes only the supplied fields. Both are normally idempotent when used correctly.

Q126. What is the difference between jsonify and returning a dict? L1

`jsonify` creates a JSON response with the right Content-Type header. Newer Flask versions also convert a returned dict to JSON automatically, and Flask-RESTful serialises returned dicts for you.

Q127. Why is an API useful in addition to your web pages? L1

It lets other clients such as JavaScript, a mobile app or scripts use the same data in JSON, separates backend from frontend and is easy to test with curl or Postman.

Q128. How do you test your API without a frontend? L1

Use curl, Postman or Python requests, or the Swagger UI from the OpenAPI spec. For automated checks use the Flask test client in pytest.

```
curl -X POST http://localhost:5000/api/books -H "Content-Type: application/json" -d
'{"title":"X"}'
```

Q129. Add a DELETE endpoint for a book that returns 404 when missing. L2

```
def delete(self, id):
    book = db.session.get(Book, id)
    if not book:
        abort(404, message="Book not found")
    db.session.delete(book); db.session.commit()
    return "", 204
```

Q130. The API returns 500 for bad input instead of 400. How do you fix it? L2

Validate with reqparse (required, type, choices) or manual checks and return 400 with a message before touching the database. A 500 means an unhandled exception on the server; a client mistake should be a 4xx.

Q131. Add a query parameter ?q= for searching books in the API. L2

Read the argument and filter.

```
q = request.args.get("q", "")
Book.query.filter(Book.title.ilike(f"%{q}%")).all()
```

Q132. When should POST return 409 Conflict? L2

When the request is valid but conflicts with current state, such as creating a user whose unique email already exists or booking a slot already taken. Return 409 with a message rather than a generic 400 or 500.

Q133. Nest a related list in a marshalled response. L2

Use `fields.List(fields.Nested( ... ))`.

```
book_fields = {"id": fields.Integer, "reviews":
fields.List(fields.Nested(review_fields))}
```

Backend: indexing, search, caching and logging

Q134. What is a database index and how do you create one in SQL and SQLAlchemy? L1

An index is a separate sorted structure that lets the database find rows without scanning the whole table. Create it for columns you often filter, join or sort on.

```
CREATE INDEX idx_booking_user ON booking(user_id);
```

```
user_id = db.Column(db.Integer, db.ForeignKey("user.id"), index=True)
```

Q135. How do you check whether a query uses an index? L1

Run `EXPLAIN QUERY PLAN` before the query in SQLite. `SEARCH ... USING INDEX` means the index is used, while `SCAN` means a full table scan.

```
EXPLAIN QUERY PLAN SELECT * FROM booking WHERE user_id = 5;
```

Q136. When is an index not helpful? L1

On tiny tables, on columns with very few distinct values such as a boolean, when the query wraps the column in a function or uses a leading wildcard like `LIKE '%abc'`, or on write-heavy tables where the maintenance cost outweighs the gain.

Q137. What is a composite index and does column order matter? L1

A composite index covers several columns, for example `(user_id, date)`. Order matters: it helps queries filtering on the leftmost column or on both, but not on `date` alone.

Q138. How is basic search done with SQL LIKE and what are its limits? L1

`WHERE title LIKE '%word%'` matches a substring and `ilike` in SQLAlchemy is case-insensitive. A leading wildcard cannot use a normal index, so it scans every row, and there is no ranking or word-based matching.

Q139. What is SQLite FTS5? L1

FTS5 is a SQLite extension for full-text search. You create a virtual table with the columns to search and query it with `MATCH`, which gives fast word-based search and ranking, unlike `LIKE`.

```
CREATE VIRTUAL TABLE docs USING fts5(title, body);
SELECT * FROM docs WHERE docs MATCH 'flask' ORDER BY rank;
```

Q140. What is caching and where can an app cache? L1

Caching stores the result of expensive work so repeated requests are served faster. It can be in the browser (HTTP cache headers), in the app (Flask-Caching, memoised functions), in a store like Redis, or at a proxy or CDN. The trade-off is stale data.

Q141. How do you use Flask-Caching? L1

Configure a cache type, then decorate a view or function with a timeout.

```
cache = Cache(app, config={"CACHE_TYPE": "SimpleCache"})
@app.route("/stats")
```

```
@cache.cached(timeout=60)
def stats(): ...
```

`SimpleCache` lives in process memory; use Redis for multiple workers.

Q142. What is cache invalidation and why is it hard? L1

Invalidation means removing or updating cached data when the source changes. It is hard because you must know every write that affects the cached value. Simple strategies are short timeouts or deleting the key on updates with `cache.delete`.

Q143. What are logging levels and how do you configure logging in Python? L1

From low to high: DEBUG, INFO, WARNING, ERROR, CRITICAL. A logger prints messages at or above its level.

```
import logging
logging.basicConfig(level=logging.INFO, filename="app.log",
                    format="%(asctime)s %(levelname)s %(message)s")
app.logger.info("User %s logged in", user_id)
```

Q144. Why use logging instead of print? L1

Logging has levels, timestamps, destinations such as files, and can be switched off or tuned without editing code. In production print output may be lost, while logs support diagnosing problems later.

Q145. How do you debug a Flask app? Mention pdb and the Flask shell. L1

Read the traceback first, add log lines, and use `breakpoint()` to open pdb where commands like `n`, `s`, `c`, `p var` and `l` step through code. The `flask shell` command opens a Python shell with the app context so you can try queries interactively.

Q146. What is pagination and why use it? L1

Pagination returns results in chunks instead of loading everything. It reduces memory and response time.

```
page = Item.query.paginate(page=1, per_page=10)
page.items, page.has_next
```

Q147. A page is slow because it lists bookings with their user names in a loop. How do you diagnose and fix it? L2

Log or count queries; this looks like an N+1 problem. Use a join or eager loading, add an index on the foreign key and paginate. Confirm with `EXPLAIN QUERY PLAN` and compare timings.

Q148. Log every failed login without logging the password. L2

Log the username and time, never the secret.

```
app.logger.warning("Failed login for %s from %s", email, request.remote_addr)
```

Q149. How would you log exceptions with the stack trace? L2

Use `logging.exception` or `app.logger.exception` inside an except block, which records the message at ERROR level together with the traceback.

```
try: risky()
except Exception:
    app.logger.exception("risky failed")
```

Q150. Cached dashboard shows old numbers after a new booking. What do you do? L2

Delete or refresh the cache key after the write, for example `cache.delete('stats')` after commit, or shorten the timeout. Decide how much staleness the page can tolerate.

JavaScript and the frontend

Q151. What is the DOM? L1

The Document Object Model is the browser's tree representation of the HTML page. JavaScript can

read and change it, for example changing text, adding elements or reacting to clicks.

Q152. How do you select elements and change them with JavaScript? L1

```
const el = document.getElementById("msg");
el.textContent = "Saved";
document.querySelector(".btn").classList.add("active");
```

`querySelector` accepts any CSS selector; `textContent` is safer than `innerHTML` for user text.

Q153. How do you handle events? L1

Attach a listener with `addEventListener`.

```
document.getElementById("btn").addEventListener("click", () => {
  alert("clicked");
});
```

`event.preventDefault()` stops default behaviour such as a form submit.

Q154. What is fetch and how do you call an API with it? L1

`fetch` makes HTTP requests from JavaScript and returns a Promise.

```
fetch("/api/books")
  .then(r => r.json())
  .then(data => console.log(data));
```

Q155. Show a POST request with JSON using fetch. L1

```
fetch("/api/books", {
  method: "POST",
  headers: {"Content-Type": "application/json"},
  body: JSON.stringify({title: "X"})
}).then(r => r.json());
```

Without the Content-Type header, `request.get_json()` on the server fails.

Q156. What are Promises and async/await? L1

A Promise represents a value that will arrive later. `async/await` is syntax to write asynchronous code in a sequential style.

```
async function load() {
  const r = await fetch("/api/books");
  const data = await r.json();
}
```

Q157. Does fetch reject on a 404 or 500 response? L1

No. It rejects only on network failure. Check `response.ok` or `response.status` yourself and handle errors explicitly.

Q158. What is the difference between let, const and var? L1

`const` cannot be reassigned, `let` is block-scoped and reassignable, `var` is function-scoped and hoisted, which causes surprises, so prefer `const` and `let`.

Q159. What is the difference between == and === in JavaScript? L1

`==` converts types before comparing, so `0 == '0'` is true. `===` compares without conversion, so it is false. Prefer `===`.

Q160. When would you use JavaScript in a MAD1 project? L1

For small enhancements: client-side validation, confirming deletes, live search filter, fetching JSON from your API to update part of a page without reload, or drawing charts. The core logic and validation remain on the server.

Q161. Add a confirm dialog before deleting. L2

```
<form method="post" action="/delete/5" onsubmit="return confirm('Delete this
item?');">
  <button class="btn btn-danger">Delete</button>
</form>
```

Q162. Live filter a table as the user types. L2

```
document.getElementById("q").addEventListener("input", e => {
  const q = e.target.value.toLowerCase();
  document.querySelectorAll("tbody tr").forEach(tr => {
    tr.style.display = tr.textContent.toLowerCase().includes(q) ? "" : "none";
  });
});
```

Q163. Your fetch to the Flask API returns 400. How do you debug in the browser? L2

Open the Network tab, click the request and check the request payload, Content-Type header and the response body, which usually contains the error message. Also check the Console for JavaScript errors.

Q164. Why can JavaScript fail to find an element and return null? L2

The script may run before the element exists, so place it at the end of body or use `defer`, or the id or selector may be misspelled. Check the Elements panel for the real id.

Security and authentication

Q165. What is the difference between authentication and authorization? L1

Authentication verifies who you are, for example by password. Authorization decides what you are allowed to do, for example whether you can open the admin page. Both are needed.

Q166. How should passwords be stored? L1

Only as salted hashes produced by a slow hashing function, never plain text. Werkzeug provides this.

```
from werkzeug.security import generate_password_hash, check_password_hash
h = generate_password_hash("secret")
check_password_hash(h, "secret")
```

Q167. Why can hashes not be reversed, and why use a salt? L1

A hash is a one-way function, so you verify by hashing the input again and comparing. A random salt per password makes identical passwords produce different hashes and defeats precomputed rainbow tables.

Q168. What is session-based login in Flask? L1

After checking credentials you store the user id in `session`; later requests carry the signed cookie so the server knows who the user is. Logout clears the session with `session.clear()`.

Q169. What is role-based access control (RBAC)? L1

Users are assigned roles such as admin, customer or sponsor, and each route or action is allowed only for particular roles. It is enforced in the backend with decorators, not by merely hiding links.

Q170. What is Flask-Security and what does it provide? L1

Flask-Security (Flask-Security-Too) adds registration, login, logout, password hashing, roles and role-protected routes on top of Flask-Login. You define User and Role models and use decorators like `@login_required` and `@roles_required('admin')`.

Q171. What is XSS and how is it prevented? L1

Cross-site scripting is injecting script into a page so it runs in another user's browser. Prevent it by escaping output, which Jinja2 does by default, avoiding `|safe` on user input and avoiding `innerHTML` with user text.

Q172. What is CSRF and how do you defend against it? L1

Cross-site request forgery makes a logged-in user's browser send an unwanted request to your site. Defend with a secret per-form token (Flask-WTF `CSRFProtect`), SameSite cookies and using POST for

state changes.

Q173. What is SQL injection and how do you prevent it? L1

It is when user input is concatenated into SQL and changes the query, for example `' OR 1=1 --`. Use the ORM or parameterised queries so input is only data.

Q174. What is IDOR and how does your app avoid it? L1

Insecure direct object reference means a user changes an id in the URL to access someone else's record. Always check ownership on the server by filtering with the logged-in user's id, and return 403 or 404 otherwise.

Q175. Why is client-side validation not enough? L1

Anyone can bypass the browser using dev tools, curl or a script. The server must validate every input for type, length, uniqueness and permission.

Q176. Which cookie flags improve security? L1

`HttpOnly` stops JavaScript reading the cookie, `Secure` sends it only over HTTPS and `SameSite` limits cross-site sending. In Flask: `SESSION_COOKIE_HTTPONLY`, `SESSION_COOKIE_SECURE`, `SESSION_COOKIE_SAMESITE`.

Q177. How do you keep secrets like SECRET_KEY out of code? L1

Read them from environment variables or an untracked config file and add that file to `.gitignore`.

```
app.config["SECRET_KEY"] = os.environ["SECRET_KEY"]
```

Q178. A normal user types /admin in the URL and sees the page. Fix it. L2

The route lacks a server-side role check. Add a decorator that verifies the logged-in role and aborts with 403, and also test it manually and in pytest.

```
@app.route("/admin")
@login_required
@roles_required("admin")
def admin(): ...
```

Q179. Change the stored password hashing so existing users keep working. L2

Hash on registration with the new method and verify with `check_password_hash`, which reads the method from the stored hash. For stronger scheme migration, rehash the password on the next successful login.

Q180. Why is session['user'] = user_id safer than a cookie holding the plain user id? L2

The Flask session cookie is signed, so tampering invalidates it, while a plain id cookie could be edited to impersonate someone. Still, the session is only signed, so do not store sensitive data in it.

Testing

Q181. What is the difference between unit tests and functional tests? L1

A unit test checks one small piece such as a function in isolation. A functional (integration) test exercises a feature through several layers, for example posting a form through the test client and checking the response and database.

Q182. What is pytest and how do you write a test? L1

pytest finds functions named `test_*` and uses plain `assert` statements.

```
def test_add():
    assert 1 + 2 == 3
```

Run with `pytest -v`.

Q183. What is a fixture? L1

A fixture is reusable setup and teardown code injected into tests by name. A typical one creates an app and an empty test database.

```
@pytest.fixture
def client():
    app = create_app({"TESTING": True, "SQLALCHEMY_DATABASE_URI": "sqlite://"})
    with app.app_context():
        db.create_all()
        yield app.test_client()
```

Q184. How do you test a Flask route? L1

Use the test client and assert on status code and content.

```
def test_home(client):
    r = client.get("/")
    assert r.status_code == 200
    assert b"Welcome" in r.data
```

Q185. How do you test a POST with login? L1

```
def test_login(client):
    r = client.post("/login", data={"email": "a@b.com", "password": "x"},
                    follow_redirects=True)
    assert r.status_code == 200
```

`follow_redirects=True` follows the 302 to the final page.

Q186. Why use a separate test database? L1

Tests must not damage real data and must start from a known state so they are repeatable. In-memory SQLite (`sqlite://`) or a temporary file is created and thrown away for each test.

Q187. What is code coverage and how do you measure it? L1

Coverage is the percentage of lines your tests execute. Run `coverage run -m pytest` then `coverage report`, or use `pytest --cov=app`. High coverage does not prove correctness but low coverage shows untested code.

Q188. What edge cases should you test in a booking app? L1

Duplicate registration, empty or invalid fields, nonexistent ids, double booking the same slot, unauthorised access to another user's data and admin routes, and deleting a parent with children.

Q189. What is a test of a REST API like? L1

Send requests with the test client and check status code and JSON.

```
r = client.post("/api/books", json={"title": "X"})
assert r.status_code == 201
assert r.get_json()["title"] == "X"
```

Q190. What is manual testing and when is it still needed? L1

Manual testing means trying scenarios yourself in a browser. It finds layout and usability problems that automated tests miss, but is slow and not repeatable, so combine with automated tests.

Q191. A test passes alone but fails when run with others. Why? L2

Tests share state such as the database. Make each test use a fresh fixture with its own app and tables, and avoid depending on test order.

Q192. Test that a normal user gets 403 on an admin route. L2

```
def test_admin_forbidden(client, login_user):
    r = client.get("/admin")
    assert r.status_code == 403
```

Q193. Your test hits 'working outside of application context'. Fix it. L2

The code touching the database needs an app context. Create the app in the fixture and use `with app.app_context():` around setup and queries.

HTML5, accessibility and encoding

Q194. What is the doctype and why is it needed? L1

`<!DOCTYPE html>` tells the browser to use standards mode for HTML5. Without it browsers may render in quirks mode with older, inconsistent behaviour.

Q195. What are the new HTML5 semantic elements and form features? L1

Semantic elements include `header`, `nav`, `main`, `section`, `article`, `aside` and `footer`. Forms gained types like `email`, `date`, `number`, `range` and attributes such as `required`, `placeholder` and `pattern`. HTML5 also has `audio`, `video` and `canvas`.

Q196. What is accessibility (a11y) and why does it matter? L1

Accessibility means people with disabilities, for example screen reader or keyboard-only users, can use the site. It matters for fairness, legal reasons and also improves usability and SEO for everyone.

Q197. How do you make a page accessible? L1

Use semantic tags, give images `alt` text, connect labels to inputs, keep sufficient colour contrast, ensure everything works via keyboard with visible focus, and use ARIA only when native elements are not enough.

Q198. How do you associate a label with an input? L1

Use `for` matching the input's `id`, or wrap the input in the label.

```
<label for="email">Email</label>
<input type="email" id="email" name="email">
```

Clicking the label focuses the input, and screen readers announce it.

Q199. What is the alt attribute? L1

It provides text for an image when it cannot be seen, read out by screen readers and shown if the image fails to load. Decorative images get an empty `alt=""`.

Q200. What is a character encoding and why use UTF-8? L1

Encoding maps characters to bytes. UTF-8 can represent every Unicode character, is backward-compatible with ASCII and is the web standard. Declare it with `<meta charset="utf-8">` so non-English text displays correctly.

Q201. What is the difference between ASCII and Unicode? L1

ASCII has 128 characters, mainly English letters and symbols. Unicode assigns a code point to characters of all scripts and emoji; UTF-8 is one way to encode those code points into bytes.

Q202. What are HTML entities? L1

Entities represent reserved or special characters, for example `<` for `<`, `>`, `&` for `&` and `"`. They stop the browser treating text as markup and are what Jinja2 autoescaping outputs.

Q203. What are ARIA attributes? L1

ARIA attributes such as `role`, `aria-label` and `aria-live` add accessibility information to elements where plain HTML cannot express it. The first rule of ARIA is to use a native element like `button` if one exists.

Q204. What is URL encoding? L1

Special characters in URLs are replaced with percent codes, for example a space becomes `%20` and `&` becomes `%26`. Query values should be encoded, and Flask decodes them in `request.args`.

Q205. A screen reader reads 'button button' for an icon-only button. Fix it. L2

Give the button an accessible name with `aria-label` or visible hidden text.

```
<button aria-label="Delete item"><i class="bi bi-trash"></i></button>
```

Q206. Hindi or accented text shows as garbled characters. What do you check? L2

Ensure the HTML has `<meta charset="utf-8">`, the file is saved as UTF-8, and the response Content-Type charset is UTF-8. Also check that the database and any file reads use UTF-8.

Q207. Why should a form error message not rely on colour alone? L2

Colour-blind users or screen readers may miss it. Add text such as 'Error: email required', an icon, and link it to the field with `aria-describedby`.

Deployment and Git

Q208. What is WSGI? L1

WSGI (Web Server Gateway Interface) is the standard Python interface between web servers and applications. Flask is a WSGI app, so servers like gunicorn can call it. Flask's built-in server is only for development.

Q209. Why not use flask run in production? L1

The development server is single-purpose, not hardened and weak under load. Use a production WSGI server such as gunicorn with several workers, normally behind nginx.

```
gunicorn -w 4 -b 127.0.0.1:8000 app:app
```

Q210. What does nginx do in front of gunicorn? L1

It acts as a reverse proxy: handles HTTPS, serves static files efficiently, buffers slow clients and forwards dynamic requests to gunicorn. This improves security and performance.

Q211. What is a virtual environment and why use one? L1

A virtual environment is an isolated Python installation per project so package versions do not clash.

```
python -m venv venv
source venv/bin/activate    # Windows: venv\Scripts\activate
pip install -r requirements.txt
```

Q212. What is requirements.txt? L1

It lists the project's dependencies so anyone can reproduce the environment. Generate it with `pip freeze > requirements.txt` and install with `pip install -r requirements.txt`.

Q213. What is Docker and why use it? L1

Docker packages the app with its dependencies into an image that runs the same on any machine as a container. It removes 'works on my machine' problems and simplifies deployment.

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["gunicorn", "-b", "0.0.0.0:8000", "app:app"]
```

Q214. What is the difference between a Docker image and a container? L1

An image is the read-only template built from the Dockerfile; a container is a running instance of an image.

Q215. What are the basic Git commands you use? L1

```
git init
git add .
git commit -m "message"
git branch feature
git checkout feature
git push origin main
```

Git tracks changes so you can see history, work on branches and undo mistakes.

Q216. What belongs in .gitignore? L1

Files that should not be versioned: the virtual environment, `__pycache__`, `.db` files, uploaded files, `.env` and secrets, and editor settings.

Q217. How do you manage configuration differences between development and production? L1

Use separate config classes or environment variables for debug flag, database URI and secret key, and select by environment. Never commit production secrets.

Q218. How would you deploy a Flask app to a server in outline? L1

Copy the code, create a virtualenv, install requirements, set environment variables, run migrations, start gunicorn under a process manager such as systemd, and configure nginx as the reverse proxy with HTTPS.

Q219. The project works on your laptop but fails on the examiner's machine. What do you check? L2

Missing or version-mismatched dependencies (update requirements.txt), hard-coded absolute paths, a database that is not created at startup, a missing environment variable, or OS-specific path separators. Provide clear run instructions and test them in a fresh virtualenv.

Q220. You accidentally committed a .db file and a secret. What now? L2

Remove the file from tracking with `git rm --cached`, add it to `.gitignore` and commit. A secret that was pushed must be treated as leaked: rotate it, since history still contains it.

Q221. Run the app on a different port or host. L2

```
flask run --port 5001 --host 0.0.0.0
```

Or `app.run(port=5001)`. Binding to `0.0.0.0` exposes the dev server to the network, so do it only deliberately.

Your project (design choices)

Q222. Give a short overview of your project and its users. L1

State the problem it automates, name the roles such as admin and user, and list the main features in one breath, for example registration, browsing, booking and an admin dashboard. Then name the stack: Flask, Jinja2, SQLite with SQLAlchemy, Bootstrap. Keep it under thirty seconds and let them ask for detail.

Q223. Why did you choose SQLite? L1

SQLite is serverless, needs no setup, ships with Python and is a single file, so the examiner can run the project anywhere. For the scale of this project, with modest concurrent writes, it is enough. If it grew I would move to PostgreSQL by changing the connection URI, as SQLAlchemy abstracts the database.

Q224. Why Flask and not Django? L1

Flask is lightweight and explicit, so I understand every part of the request flow: routing, templates and database. It suits a small to medium project and the course teaches it. Django offers an admin and ORM out of the box, but that hides things I wanted to learn.

Q225. Explain your database schema and why you designed it that way. L1

Name each table, its primary key and the relationships: for example `User 1–N Booking N–1 Slot`. Explain that repeated information is stored once and referenced by foreign keys, which follows third normal form, and mention the constraints you added like unique and not null.

Q226. How does your app prevent a user from seeing another user's data? L1

Every protected route requires login and every query is filtered by the logged-in user's id from the session, never by an id trusted from the URL alone. For a record opened by id, I check `record.user_id == current_user.id` and abort with 403 or 404 otherwise. I tested this by trying another user's URLs.

```
b = Booking.query.get_or_404(id)
if b.user_id ≠ session["user_id"]:
    abort(403)
```

Q227. How do you stop duplicate bookings or registrations? L1

At two levels. In the database a `UniqueConstraint` on the columns that must be unique, such as `(user_id, slot_id)` or email, guarantees it even with two simultaneous requests. In code I check first for a friendly message and also catch `IntegrityError` and roll back.

Q228. How is the admin created and how is admin access protected? L1

The admin is created programmatically at first startup if none exists, so there is no public admin signup. Admin routes are guarded by a role check that returns 403 for others, in the backend, not just by hiding menu links.

Q229. Where do you validate input and why in both places? L1

In the browser with `required`, types and patterns for quick feedback, and on the server for every field: presence, type, length, uniqueness and permission. The server is the authority because browser checks can be bypassed.

Q230. How is search implemented in your app? L1

Describe your real implementation: typically a `filter` with `ilike` on name or category using a query parameter, optionally ordered and paginated. Say that if data grew, I would add an index or use SQLite FTS5.

Q231. Walk me through one feature end to end. L1

Pick one, for example booking: the user clicks a button which is a form POST to a route; the route checks login, validates input, creates a `Booking` through SQLAlchemy, commits and redirects to the bookings page; the template loops over the user's bookings. Name the template, route function and model at each step.

Q232. How do you handle deleting a record that other records refer to? L1

Either cascade the delete to children with `cascade='all, delete-orphan'`, block deletion while children exist with a friendly message, or soft delete with a flag so history is kept. I chose whichever fits the business rule, for example soft delete for items that appear in past bookings.

Q233. What happens if a user submits invalid data or the database fails? L1

Invalid input is caught by validation, which re-renders the form with a flash message. A database error is caught, the session rolled back so nothing half-saved remains, and a friendly message or error page is shown while the details go to the log.

Q234. What would you improve with more time? L1

Pagination and caching for large lists, automated pytest tests, a proper token-based REST API with documentation, CSRF protection if not present, migrations with Flask-Migrate, and deployment with gunicorn and Docker.

Q235. What was the hardest part and how did you solve it? L1

Give a specific example, such as modelling relationships or preventing double booking. Describe the symptom, how you debugged it using the traceback, logging or the Flask shell, and the fix. A concrete story shows you wrote the code.

Q236. Did you use external help or AI? Can you explain every line? L1

Answer honestly. Say what you took from documentation, tutorials or tools, and that you reviewed, tested and understand all code. Then be ready to explain any file you are shown, because that is what the viva checks.

Q237. Show me where in your code a user is authenticated, and change the redirect after login live. L2

Open the login route, point to the credentials check and the `session` assignment, then edit the redirect target.

```
return redirect(url_for("user_dashboard"))   # was "home"
```

Save, restart if needed, and demonstrate the new behaviour.

Q238. Add a new field to a model in your project and show it in the UI. L2

Add the column to the model, apply the schema change (migration, or recreate the dev database), add the input to the form, read it with `request.form`, and display it in the template. Mention that

existing rows will have a default or NULL.

Q239. How would your project behave with two users booking the last slot at the same moment? L2

Without protection both could succeed. A unique constraint or a conditional update inside one transaction makes the database accept only one and raise IntegrityError for the other, which I catch and report as slot taken. Checking in Python alone is not enough because of the race.

Code-change challenges (Level 2 style)

Q240. Add a new route /hello/<name> that greets the user. L2

```
@app.route("/hello/<name>")
def hello(name):
    return f"Hello, {name}!"
```

The `<name>` part of the URL arrives as the function argument. In real pages use a template so the name is escaped.

Q241. Add a column 'phone' to the User model and show it on the profile page. L2

```
phone = db.Column(db.String(15))
```

Run a migration or recreate the database in development, add an input named `phone` in the form, save it with `user.phone = request.form['phone']`, and show `{{ user.phone }}` in the template.

Q242. Change the dashboard to show only the latest 5 records. L2

```
items = Item.query.order_by(Item.created_at.desc()).limit(5).all()
```

`order_by( ... desc())` puts the newest first and `limit(5)` keeps five.

Q243. Add a search box that filters records by name. L2

```
q = request.args.get("q", "")
items = Item.query.filter(Item.name.ilike(f"%{q}%")).all()
```

```
<form method="get"><input name="q" value="{{ request.args.get('q','') }}">
<button>Search</button></form>
```

Q244. Make the delete button work with POST rather than a link. L2

A link makes a GET, which should not change data. Use a form.

```
<form method="post" action="{{ url_for('delete_item', id=item.id) }}">
<button>Delete</button></form>
```

```
@app.route("/delete/<int:id>", methods=["POST"])
```

Q245. Disable a user (block) without deleting them and stop them logging in. L2

Add `is_active = db.Column(db.Boolean, default=True)` and check it in the login route.

```
if not user.is_active:
    flash("Account blocked"); return redirect(url_for("login"))
```

Blocking sets the flag to False and commits.

Q246. Show a count of bookings per user on the admin page. L2

```
rows = (db.session.query(User.name, func.count(Booking.id))
        .outerjoin(Booking).group_by(User.id).all())
```

In the template loop `{% for name, n in rows %}`.

Q247. Add a unique constraint on (user_id, slot_id) and handle the error. L2

```
__table_args__ = (db.UniqueConstraint("user_id", "slot_id"),)
```

Then catch `IntegrityError` around commit, call `db.session.rollback()` and flash a message.

Q248. Change the Bootstrap layout so cards show 3 per row on large screens and 1 on phones. L2

```
<div class="row">
  {% for c in cards %}
  <div class="col-12 col-lg-4 mb-3"><div class="card"> … </div></div>
  {% endfor %}
</div>
```

Q249. Add a flash message after a successful save and display it on the page. L2

```
flash("Saved successfully", "success")
return redirect(url_for("list_items"))
```

Show it in `base.html` with a loop over `get_flashed_messages(with_categories=true)` and a Bootstrap alert.

Q250. Add pagination of 10 items per page. L2

```
page = request.args.get("page", 1, type=int)
p = Item.query.paginate(page=page, per_page=10)
```

In the template loop `p.items` and link to `url_for('items', page=p.next_num)` when `p.has_next`.

Q251. Add a JSON endpoint /api/items that returns all items. L2

```
@app.route("/api/items")
def api_items():
```

```
    return jsonify([{"id": i.id, "name": i.name} for i in Item.query.all()])
```

Q252. Sort a table by price when the user clicks a link. L2

Pass `?sort=price` and map it to a column.

```
sort = request.args.get("sort", "name")
col = {"name": Item.name, "price": Item.price}.get(sort, Item.name)
items = Item.query.order_by(col).all()
```

Using a whitelist prevents arbitrary column names.

Q253. Show a custom 404 page. L2

```
@app.errorhandler(404)
def nf(e):
    return render_template("404.html"), 404
```

Return the status code 404 with the page so browsers and tools see a real not found.

Q254. Hash the password at registration and verify on login. L2

```
user = User(email=email, password=generate_password_hash(pw))
 ...
if user and check_password_hash(user.password, pw): session["user_id"] = user.id
```

The password column must be long enough, for example 255 characters.

Q255. Log out so that the back button does not show a protected page. L2

Clear the session and redirect, and add no-store headers on protected pages.

```
session.clear()
@app.after_request
def nocache(r): r.headers["Cache-Control"] = "no-store"; return r
```

Q256. Add a created_at timestamp to a model. L2

```
from datetime import datetime, timezone
created_at = db.Column(db.DateTime, default=lambda: datetime.now(timezone.utc))
```

Pass a function so the time is computed per row, not once at import.

Q257. Prevent a past date being booked. L2

Validate on the server.

```
d = datetime.strptime(request.form["date"], "%Y-%m-%d").date()
if d < date.today():
    flash("Date must be today or later"); return redirect(url_for("book"))
```

Add `min` on the date input too for the browser.

Q258. Add an index on a column and prove it is used. L2

```
name = db.Column(db.String(80), index=True)
```

Then run `EXPLAIN QUERY PLAN SELECT * FROM item WHERE name = 'x';` and look for `USING INDEX`.

MAD2 viva question bank (282 questions)

JavaScript core

Q1. What is the difference between var, let and const? L1

`var` is function-scoped and hoisted with the value `undefined`. `let` and `const` are block-scoped and live in the temporal dead zone until their declaration line runs, so using them earlier throws a ReferenceError. `const` stops you from re-assigning the variable, but an object or array held in a `const` can still be mutated. In my code I use `const` by default and `let` only when the value must change.

Q2. What is the difference between == and === in JavaScript? L1

`==` compares after type coercion, so `0 == "0"` and `null == undefined` are true. `===` compares both type and value with no coercion, so `0 === "0"` is false. Strict equality is predictable, so I use `===` everywhere. The only common use of `==` is `x == null` to catch both null and undefined.

Q3. What is a closure? Give a small example. L1

A closure is a function that keeps access to the variables of the scope where it was created, even after that outer function has returned. The variables stay alive because the inner function still references them.

```
function counter() {
  let n = 0;
  return () => ++n;
}
const next = counter();
next(); // 1
next(); // 2
```

Here `n` is private to `counter` and only `next` can change it. Closures are what make callbacks, event handlers and module-style privacy work.

Q4. This loop prints 3, 3, 3. Why, and how do you make it print 0, 1, 2? ```js for (var i = 0; i < 3; i++) { setTimeout(() => console.log(i), 0); } ``` L2

`var i` is a single function-scoped variable shared by all three callbacks. The callbacks run after the loop ends, when `i` is already 3. Changing `var` to `let` creates a fresh binding of `i` for each iteration, so the closures capture 0, 1 and 2. The older fix is wrapping the body in an IIFE that receives `i` as a parameter.

Q5. What is hoisting? L1

Before running the code, JavaScript registers declarations in their scope. Function declarations are hoisted completely, so you can call them before they appear. `var` variables are hoisted and initialised to `undefined`. `let` and `const` are hoisted too, but stay uninitialised in the temporal dead zone, so reading them early throws an error. Function expressions and arrow functions assigned to a variable follow that variable's rule.

Q6. How is an arrow function different from a normal function? L1

An arrow function has no `this`, `arguments` or `super` of its own; it uses the `this` of the surrounding scope (lexical `this`). It also cannot be used as a constructor with `new`. That is why arrows are good for callbacks inside methods, while object methods and Vue 2 `methods` should be normal functions so that `this` points to the instance.

Q7. What will `this` be inside a Vue 2 method if you write the method as an arrow function, and what is the fix? ```js methods: { save: () => { this.count++; } // wrong } ``` L2

The arrow function takes `this` from the enclosing module scope, not from the Vue instance, so `this.count` is undefined or throws an error. Vue binds methods to the instance, but it cannot rebind an arrow. Write `save() { this.count++; }` or `save: function () { ... }`. Arrows are fine for callbacks nested inside a method, because they inherit the method's `this`.

Q8. What does `this` refer to in JavaScript? L1

`this` is decided by how a function is called, not where it is written. In a method call `obj.f()` it is `obj`; with `new` it is the new object; with `call`, `apply` or `bind` it is whatever you pass; in a plain call it is `undefined` in strict mode (or the global object otherwise). Arrow functions do not have their own `this` and use the surrounding one.

Q9. Explain prototypes and prototype-based inheritance in JavaScript. L1

Every object has an internal link to another object called its prototype. When you read a property that the object does not have, JavaScript walks up this prototype chain until it finds the property or reaches `null`. Functions used with `new` set the new object's prototype to `Fn.prototype`, and `class` syntax is mostly a cleaner way to write the same mechanism. Methods put on the prototype are shared by all instances instead of copied.

Q10. What is an execution context and the call stack? L1

An execution context is the environment in which code runs: it holds the variables, the scope chain and the value of `this`. There is a global context, and each function call creates a new one, which is pushed on the call stack and popped when the function returns. JavaScript has a single call stack, so it runs one thing at a time. Asynchronous callbacks only run when the stack is empty.

Q11. What do map, filter, forEach and reduce do? L1

`map` returns a new array with each element transformed. `filter` returns a new array with only the elements for which the callback returns true. `forEach` just runs a function for each element and returns `undefined`. `reduce` folds the array into a single value using an accumulator.

```
const total = [1, 2, 3].reduce((acc, x) ⇒ acc + x, 0); // 6
```

Q12. What are destructuring and the spread operator? L1

Destructuring unpacks values from arrays or objects into variables: `const { id, name } = user;` or `const [a, b] = arr;`. Spread `...` expands an iterable or object in place, which is handy for copying and merging: `const next = { ...state, loading: false };` or `[...list, item]`. I use spread to produce new objects instead of mutating, for example inside Vuex mutations or when copying a prop to local data.

Q13. What are ES modules and why do we need a bundler in a Vue CLI project? L1

ES modules let each file keep its own scope and share code with `export` and `import`. In a plain browser page you need `<script type="module">` and real file paths. In a Vue CLI project, webpack bundles the modules, resolves npm packages, compiles `.vue` files and minifies the output for production. Modules avoid global variable clashes and make dependencies explicit.

Q14. What is the difference between a shallow copy and a deep copy? L1

A shallow copy such as `{...obj}` or `arr.slice()` copies only the top level, so nested objects are still shared between the original and the copy. A deep copy duplicates nested structures as well; `structuredClone(obj)` works for plain data, and `JSON.parse(JSON.stringify(obj))` works for JSON-safe data but loses functions, dates and undefined. This matters in Vue and Vuex when you copy state and then edit it: a shallow copy can silently mutate the original.

Q15. What do truthy and falsy mean? List the falsy values. L1

In a condition JavaScript converts a value to a boolean. The falsy values are `false`, `0`, `-0`, `0n`, `""`, `null`, `undefined` and `NaN`; everything else is truthy, including `"0"`, `[]` and `{}`. This is why `v-if="items"` is true for an empty array, and you should write `items.length` when you mean non-empty.

Q16. Why does `typeof null` give "object" and how do you check for NaN? L2

`typeof null === "object"` is a historical bug in the language that was never fixed for compatibility, so test null with `x === null`. `NaN` is the only value not equal to itself, so `NaN === NaN` is false; use `Number.isNaN(x)`. Arrays also report "object", so use `Array.isArray(x)`.

Q17. What are JSON.stringify and JSON.parse used for? L1

`JSON.stringify(obj)` turns a JavaScript value into a JSON string, and `JSON.parse(str)` turns a JSON string back into a value. We use them to send a body with fetch (`body: JSON.stringify(data)`) and to keep objects in localStorage, which only stores strings. `JSON.parse` throws on invalid input, so wrap it in try/catch when the data is not trusted.

Q18. Explain event bubbling and event delegation. What would the order of handlers be if a click happens on a table cell? L2

When an element is clicked, the event first travels down in a capture phase and then bubbles up from the target through its ancestors, so listeners on the `td`, then `tr`, then `table`, then `body` fire in that order (for the default bubbling listeners). `event.stopPropagation()` stops it going further. Event delegation means attaching one listener to a parent and checking `event.target`, which works well for long or dynamic lists. Capturing listeners (third argument `true`) run on the way down, before the bubbling ones.

Q19. What are call, apply and bind? L2

All three set `this` explicitly. `fn.call(obj, a, b)` runs the function immediately with arguments listed, `fn.apply(obj, [a, b])` does the same with an array of arguments, and `fn.bind(obj)` returns a new function with `this` fixed for later. `bind` is useful when passing a method as a callback, such as `setTimeout(this.save.bind(this), 100)`.

Q20. What is the output and why? ```js const o = { name: "A", hi() { return this.name; } }; const f = o.hi; console.log(o.hi(), f()); ``` L2

`o.hi()` is a method call so `this` is `o` and it prints "A". `f()` is a plain call: `this` is undefined in strict mode (which throws if you access `.name`) or the global object in sloppy mode (usually `undefined`/empty name in a browser). Detaching a method loses its `this`. Fix with `o.hi.bind(o)`.

Async JavaScript

Q21. Explain the JavaScript event loop. L1

JavaScript runs on one thread with a call stack. Asynchronous work, such as timers, fetch and DOM events, is handled by the browser, and when it finishes a callback is placed in a queue. The event loop takes the next callback from the queue only when the call stack is empty. Promise callbacks go into the microtask queue, which is fully drained before the next macrotask such as a `setTimeout` callback.

Q22. What is the order of output and why? ```js console.log("A"); setTimeout(() => console.log("B"), 0); Promise.resolve().then(() => console.log("C")); console.log("D"); ``` L2

The output is A, D, C, B. A and D are synchronous. The promise callback is a microtask and runs right after the current script finishes. The timer callback is a macrotask and runs later, even with a delay of 0.

Q23. What problem do promises solve compared with callbacks? L1

Nested callbacks lead to deeply indented "callback hell" with scattered error handling. A promise represents a future result and can be chained with `.then()`, with one `.catch()` for any failure in the chain. It has three states: pending, fulfilled and rejected, and once settled it never changes. `async` / `await` is syntax that lets you consume promises in a sequential style.

Q24. What does an async function return, and what does await do? L1

An async function always returns a Promise; a returned value is wrapped in a resolved promise and a thrown error becomes a rejection. `await` pauses only that function until the awaited promise settles, then gives you the value or throws the rejection. The rest of the program is not blocked, because the function is suspended and the event loop continues.

Q25. How do you handle errors with async/await? L1

Wrap the awaited calls in try/catch (optionally with finally to reset a loading flag), or attach `.catch()` to the returned promise.

```
async function load() {
  try {
    const res = await fetch("/api/lots");
    if (!res.ok) throw new Error("HTTP " + res.status);
    return await res.json();
  } catch (e) {
    console.error(e);
  } finally {
```

```
    // hide spinner
  }
}
```

Q26. Does fetch reject the promise when the server returns 404 or 500? L1

No. `fetch` rejects only on network-level failure such as no connection or a CORS block. A 404 or 500 still resolves with a Response object, so you must check `res.ok` or `res.status` yourself. Axios differs here: it rejects for any status outside 2xx by default.

Q27. How do you send JSON with a POST request using fetch? L1

```
const res = await fetch("/api/login", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ email, password }),
});
const data = await res.json();
```

`res.json()` itself returns a promise, so it needs `await` or `.then`. Without the Content-Type header, Flask's `request.get_json()` will not parse the body as JSON by default.

Q28. What is the difference between Promise.all, Promise.allSettled, Promise.race and Promise.any? L1

`Promise.all` resolves with an array of all results, but rejects as soon as any one promise rejects. `Promise.allSettled` waits for all and returns each outcome as fulfilled or rejected, never rejecting. `Promise.race` settles with the first promise to settle, whether success or failure. `Promise.any` resolves with the first promise that fulfils and only rejects if all of them reject.

Q29. You need three independent API calls. Which is faster, awaiting them one by one or using Promise.all, and show the code? L2

Awaiting one by one runs them sequentially, so total time is the sum of all three. Starting them together and awaiting with `Promise.all` runs them in parallel, so total time is about the slowest one.

```
const [lots, users, stats] = await Promise.all([
  fetch("/api/lots").then(r => r.json()),
  fetch("/api/users").then(r => r.json()),
  fetch("/api/stats").then(r => r.json()),
]);
```

Only do sequential awaits when a call depends on the result of the previous one.

Q30. Why does `items.forEach(async x => { await save(x); })` not wait for the saves to finish? L2

`forEach` ignores the promises returned by the callback, so it returns immediately and the code after it runs before the saves complete. To run them one after another use `for (const x of items) { await save(x); }`. To run them in parallel and wait, use `await Promise.all(items.map(x => save(x)))`.

Q31. A team of two works in parallel (3 s and 2 s) and a solo player takes 4 s. How long does Promise.all take, and how long would sequential awaits take? L2

`Promise.all` finishes when the slowest promise finishes, so for tasks of 3 s and 2 s started together it takes about 3 s. Awaiting them one after another takes about 5 s (3 + 2). The solo 4 s task would beat the sequential team but lose to the parallel team. This shows that parallel promises take the maximum time, sequential ones take the sum.

Q32. What is debouncing and where did you use or would you use it? L1

Debouncing delays running a function until the user has stopped triggering it for a fixed time, for example 300 ms after the last keystroke in a search box. It avoids sending an API request for every key press. In Vue you can implement it with `setTimeout` and `clearTimeout` inside a watcher or method.

```
let t;
function onInput(q) {
  clearTimeout(t);
  t = setTimeout(() => search(q), 300);
}
```

Q33. What is setTimeout(fn, 0) and does it run immediately? L1

It schedules `fn` as a macrotask after at least the given delay, so it runs only after the current code and all pending microtasks (promise callbacks) have finished. The delay is a minimum, not a guarantee, because the call stack must be empty first.

Q34. How can you cancel a fetch request, for example when the user types a new search? L2

Create an `AbortController`, pass its `signal` to `fetch`, and call `controller.abort()` to cancel; the fetch promise rejects with an `AbortError`, which you should ignore.

```
const c = new AbortController();
fetch("/api/search?q=abc", { signal: c.signal });
c.abort();
```

Axios supports cancelling too (newer versions accept the same AbortController signal).

Q35. What happens if a promise rejects and nobody handles it? L2

The browser reports an "unhandled promise rejection" in the console and the failure is easy to miss. Always add `.catch()` or try/catch around `await`. In a Vue app you can also add a global handler with `window.addEventListener("unhandledrejection", …)` for logging.

Q36. How would you poll an API every few seconds from a Vue page, and what must you remember to do? L1

Start a `setInterval` in `mounted` that calls the status endpoint and stores the result in `data`; stop it with `clearInterval` once the job is done and also in `beforeDestroy`. If you forget to clear it, the timer keeps running after the component is gone, causing memory leaks and extra requests.

```
mounted() { this.timer = setInterval(this.check, 3000); },
beforeDestroy() { clearInterval(this.timer); }
```

Q37. In the following code, which line prints first, and what is the type of `r`? ```js async function f() { return 5; } const r = f(); console.log(r); ``` L2

It prints `Promise { 5 }` (a resolved promise), not the number 5, because async functions always wrap their return value. You need `await f()` or `f().then(v ⇒ …)` to get 5.

Web frontend and browser storage

Q38. What is a single-page application (SPA) and how does it differ from a multi-page app? L1

An SPA loads one HTML page and then updates the view in the browser with JavaScript, getting data from APIs as JSON. A traditional multi-page app asks the server for a whole new HTML page on each navigation. SPAs feel faster after the first load and separate frontend from backend, but the first load is heavier and SEO needs extra work.

Q39. What is the difference between client-side and server-side rendering, and which does your project use? L1

In server-side rendering (Flask with Jinja) the server builds the HTML and sends it ready to display. In client-side rendering (Vue) the server sends a small HTML shell plus JavaScript, and the browser builds the page from JSON API data. My project uses Vue for client-side rendering, with Flask serving only JSON APIs plus the static build.

Q40. What is a Progressive Web App (PWA)? L1

A PWA is a web app that can be installed and can work offline or on poor networks. It needs HTTPS, a web app manifest (name, icons, start URL, display mode) and a service worker. The service worker is a script that runs in the background and can intercept network requests and serve cached files.

Q41. What does a service worker do and what are its limits? L2

A service worker sits between the page and the network, so it can cache static files and respond when the user is offline, and it can handle push notifications. It runs in a separate thread, cannot access the DOM directly, works only over HTTPS (localhost is allowed), and has its own lifecycle of install, activate and fetch events. Because it can serve stale files, the cache must be versioned and old caches deleted on activate.

Q42. Compare localStorage, sessionStorage and cookies. L1

`localStorage` keeps strings with no expiry until cleared, per origin, about 5 MB, and is not sent to the server. `sessionStorage` is the same but lives only for that tab. Cookies are small (about 4 KB), can have an expiry, and are sent to the server automatically with every matching request; they can be marked HttpOnly so JavaScript cannot read them. Use cookies when the server must receive the value each time, and storage for client-only data.

Q43. What is IndexedDB and when would you use it over localStorage? L1

IndexedDB is an asynchronous, transactional, in-browser database for large amounts of structured data, including files and blobs. localStorage is synchronous, string-only and small. Use IndexedDB for large offline data, for example cached lists in a PWA; use localStorage for tiny settings such as a theme or a token.

Q44. What is ephemeral state in a frontend app? Give examples. L1

Ephemeral state is temporary UI state that is not worth saving: whether a dropdown is open, the current text typed in an input, scroll position or a loading spinner flag. It lives in component `data` and is lost on refresh, unlike persistent state such as user records, which belong in the database, or the token, which you may keep in localStorage.

Q45. Where do you store the auth token on the client, and what are the risks? L2

I store it in localStorage (and also in Vuex so components can react to it), and send it in an Authorization or Authentication-Token header. The risk is that any injected script (XSS) can read localStorage and steal the token. An HttpOnly cookie cannot be read by scripts, but then you must defend against CSRF using SameSite and CSRF tokens. Either way, keep token lifetime short and escape all user content.

Q46. An application is built entirely on the client with JavaScript and no server. What can and can't it do? L1

It can show UI, do calculations and keep data in the browser with localStorage or IndexedDB. But data is stored per browser and per user, it is not shared across devices or users, and the user can inspect or edit it. Secrets, access control and shared state need a server, since anything in client code is visible to the user.

Q47. What is Lighthouse and what does it measure? L1

Lighthouse is a Chrome DevTools audit tool that scores a page on Performance, Accessibility, Best Practices, SEO and (optionally) PWA checks. The performance score is computed from metrics such as First Contentful Paint, Largest Contentful Paint, Total Blocking Time, Cumulative Layout Shift and Speed Index. It gives suggestions like compressing images, removing unused JavaScript and caching static files.

Q48. What is the DOM, and how is it different from the HTML source? L1

The DOM is the browser's in-memory tree of objects built from the HTML, which JavaScript can read and modify. The HTML source is just text. After scripts change the DOM, what you see on screen can differ from the original file, which is exactly what Vue does when it renders.

Q49. What does it mean that HTTP is stateless and how do web apps still remember the user? L1

Each HTTP request is independent, and the server does not remember earlier requests by itself. Apps add state on top: a session cookie holding an id that points to server-side session data, or a token sent by the client on each request that identifies the user. Token-based auth is common for SPAs and mobile apps because the API stays stateless.

Q50. What is the same-origin policy? L2

The same-origin policy is a browser rule that stops a page from one origin from reading data from another origin. An origin is scheme + host + port, so `http://localhost:8080` and `http://localhost:5000` are different origins. It protects users from malicious sites reading their data on other sites; CORS is the controlled way for a server to relax it.

Q51. How does the browser decide what to cache, and which headers control it? L2

The server controls it with HTTP headers: `Cache-Control` (for example `max-age=3600`, `no-store`, `no-cache`, `public`, `private`), `ETag` and `Last-Modified` for validation, and `Expires`. With validators the browser sends `If-None-Match` or `If-Modified-Since`, and the server may reply `304 Not Modified` with no body. This is separate from server-side caching with Redis.

Q52. What makes a web page load slowly, and how does it affect Lighthouse? L1

Large JavaScript bundles and images, render-blocking scripts and CSS, many requests, slow server response (TTFB), no compression or caching, and layout shifts. These directly hurt metrics like LCP, TBT and CLS. Fixes are code splitting, lazy loading, compressing and resizing images, gzip or brotli, caching and a CDN.

Vue basics

Q53. What is Vue.js and what does "progressive framework" mean? L1

Vue is a JavaScript framework for building user interfaces from components, with reactive data binding between state and DOM. "Progressive" means you can adopt it gradually: drop it in with one

script tag to enhance a page, or use Vue CLI with the router and Vuex for a full SPA. This course uses Vue 2 (Options API); in newer Vue 3 there is also the Composition API.

Q54. Why did you choose Vue CLI instead of the CDN version (or the reverse)? L1

CDN is just a script tag, with no build step, so it is quick for small pages and demos. Vue CLI gives single-file components (`.vue`), npm packages, hot reload, ES modules, linting and an optimised production build, which suits a multi-component SPA with Vue Router and Vuex. I would pick CLI for a project with many components and shared state, and say clearly which one I actually used and why.

Q55. What is the difference between v-if and v-show? L1

`v-if` actually adds or removes the element from the DOM (and supports `v-else-if` and `v-else`), so it has a higher cost when toggling but nothing is rendered when false. `v-show` always renders the element and only toggles `display: none`, so toggling is cheap but the initial render cost is paid. Use `v-show` for something toggled often, such as a dropdown, and `v-if` when the condition rarely changes or the content is heavy. `v-show` does not work on `<template>`.

Q56. Why do we use :key with v-for? L1

`key` gives each item a stable unique identity so Vue can reuse, move or remove the right DOM elements when the list changes, instead of patching in place and mixing up state (like input text). Use a unique id from the data, not the array index when items can be reordered or removed.

```
<li v-for="lot in lots" :key="lot.id">{{ lot.name }}</li>
```

Q57. Explain v-model. L1

`v-model` is two-way binding on form inputs. On a text input it is shorthand for `:value="x"` plus `@input="x = $event.target.value"`. It has modifiers `.trim`, `.number` and `.lazy`. On a custom component in Vue 2 it binds the `value` prop and the `input` event.

Q58. Differentiate data, computed, methods and watch. L1

`data` holds the reactive state. `computed` properties are derived values that are cached and recalculated only when their reactive dependencies change. `methods` run every time they are called

(and on each re-render if used in a template). `watch` runs a function for side effects when a property changes, such as calling an API when a search term changes.

Q59. When would you use computed instead of a method for a filtered list, and write it. L2

Use computed because the result is cached and re-evaluated only when `items` or `query` change, whereas a method in the template would run on every re-render.

```
computed: {
  filtered() {
    return this.items.filter(i ⇒
      i.name.toLowerCase().includes(this.query.toLowerCase()));
  }
}
```

Computed must be pure: no API calls or data changes inside it.

Q60. How does reactivity work in Vue 2? L1

Vue 2 walks through the `data` object and converts each property into a getter and setter using `Object.defineProperty`. When a render function or computed reads a property, Vue records the dependency, and when the setter runs it tells the dependents to re-run, so the DOM updates. In Vue 3 this uses Proxies instead.

Q61. Why doesn't the page update when you do `this.items[0] = x` or add a new property to an object in Vue 2, and how do you fix it? L2

Vue 2 cannot detect array index assignment or adding or deleting object properties because getters and setters are created only for properties existing at creation. Use `this.$set(this.items, 0, x)` (or `Vue.set`), `splice`, or replace the array or object with a new one: `this.user = { ...this.user, age: 20 }`. Declare all properties in `data` up front. (In Vue 3 Proxy-based reactivity removes this limitation.)

Q62. Why must `data` be a function inside a component? L1

Components are reused, and if `data` were a shared object all instances would share and change the same state. A function returns a fresh object for each instance. The root instance created with `new Vue` can use an object because it exists only once.

Q63. List the Vue 2 lifecycle hooks in order and say what each is used for. L1

`beforeCreate`, `created` (data and methods ready, no DOM yet; good for initial API calls), `beforeMount`, `mounted` (DOM is available; use for DOM access or third-party libraries), `beforeUpdate`, `updated` (after re-render), `beforeDestroy` (clean up timers and listeners) and `destroyed`. In Vue 3 the last two are renamed `beforeUnmount` and `unmounted`.

Q64. Where should you fetch the initial data and why? L2

Usually in `created` or `mounted`. `created` fires earlier, before the DOM is mounted, so the request starts sooner and `this` data is already reactive. `mounted` is needed only if you must touch the DOM (for example to initialise a chart on a canvas). In both cases show a loading flag, because data arrives after the first render.

Q65. Write the class binding that always applies `errorClass` and applies `activeClass` only when `isActive` is truthy. L2

```
<div class="errorClass" :class="{ activeClass: isActive }"></div>
<div :class="['errorClass', { activeClass: isActive }]"></div>
```

A plain `class` attribute and a `:class` binding are merged by Vue. The object syntax adds the class when the value is truthy; an array can mix strings and objects.

Q66. What are the shorthands for v-bind and v-on, and what is the difference between {{ }} and v-bind? L1

`:` is shorthand for `v-bind` and `@` for `v-on`. Mustache `{{ }}` interpolates text content only, while `v-bind` sets attributes and props, for example `:href="url"` or `:disabled="busy"`. You cannot use mustaches inside attributes.

Q67. What are event modifiers and what are they used for? L1

Modifiers change how an event handler behaves, without code in the method: `@submit.prevent` calls `preventDefault`, `@click.stop` stops propagation, `.once` runs only once, `.self`, `.capture`, and key modifiers like `@keyup.enter`. They keep methods free of DOM details.

Q68. What is the virtual DOM? L1

It is a lightweight JavaScript tree describing what the UI should look like. On a state change Vue renders a new virtual tree, compares it to the previous one (diffing), and applies only the necessary changes to the real DOM, which is much cheaper than rebuilding everything.

Q69. Why is using v-if and v-for on the same element discouraged? L2

Their priority differs between versions (v-for first in Vue 2, v-if first in Vue 3) and it also makes Vue loop over every item even when most are hidden. Instead, filter in a computed property and loop over the result, or move `v-if` to a wrapping `<template>`.

Q70. What is $nextTick and when is it needed? L2

After you change data, Vue updates the DOM asynchronously in a batch. If you need to read or use the updated DOM, for example to focus an input that just appeared via `v-if`, call `this.$nextTick(() ⇒ { ... })` (or `await this.$nextTick()`), which runs after the DOM update.

Q71. What is a template ref and how do you use it? L1

Add `ref="name"` to an element or component and access it as `this.$refs.name` after mount, for example `this.$refs.search.focus()`. Refs give direct access to the DOM or a child instance; prefer props and events for normal data flow because refs are not reactive.

Q72. What is the difference between Vue 2 and Vue 3? L1

Vue 3 adds the Composition API (`setup`, `ref`, `reactive`), Proxy-based reactivity, multiple root nodes in a template, better performance and TypeScript support, and its ecosystem is Vue Router 4 and Pinia (Vuex 4 still exists). Vue 2 uses the Options API and `Object.defineProperty`. This course works in Vue 2 with Vuex 3 and Vue Router 3.

Vue with APIs

Q73. How does your Vue frontend talk to the Flask backend? L1

The Vue app calls the Flask REST endpoints (for example `/api/lots`) using `fetch` or Axios, sends and receives JSON, and attaches the auth token in a request header. Flask returns JSON and status

codes, and Vue stores the result in `data` or the Vuex store, which triggers a re-render. The frontend and backend are separate apps that share only the API contract.

Q74. Show how you fetch a list from the API in a Vue 2 component. L1

```
export default {
  data() { return { lots: [], loading: true, error: null }; },
  async created() {
    try {
      const res = await fetch("/api/lots", {
        headers: { "Authentication-Token": localStorage.getItem("token") },
      });
      if (!res.ok) throw new Error("HTTP " + res.status);
      this.lots = await res.json();
    } catch (e) { this.error = e.message; }
    finally { this.loading = false; }
  },
};
```

The template then uses `v-if="loading"`, `v-else-if="error"` and `v-for` over `lots`.

Q75. Fetch vs Axios: which did you use and why? L1

Fetch is built into the browser, with no dependency, but it needs a manual `res.json()`, does not reject on HTTP errors, and has no interceptors. Axios parses JSON automatically, rejects on non-2xx, supports default headers, interceptors and cancellation, so it reduces repeated code. Either is fine; the point is to handle errors and attach the token consistently.

Q76. How do you attach the token to every request using an Axios interceptor? L2

```
import axios from "axios";
const api = axios.create({ baseURL: "http://localhost:5000" });
api.interceptors.request.use(cfg => {
  const t = localStorage.getItem("token");
  if (t) cfg.headers["Authentication-Token"] = t;
  return cfg;
});
api.interceptors.response.use(r => r, err => {
  if (err.response && err.response.status === 401) {
    localStorage.removeItem("token");
```

```
    window.location = "/login";
  }
  return Promise.reject(err);
});
```

The request interceptor avoids repeating the header in each call, and the response interceptor handles expired tokens in one place.

Q77. How do you show loading and error states in the UI while calling an API? L1

Keep `loading` and `error` flags in `data`. Set loading to true before the call, then in `finally` set it to false, and store a message in `error` in the `catch`. In the template show a spinner with `v-if="loading"`, an error message with `v-else-if="error"`, and the list otherwise. This keeps the user informed and avoids a blank page.

Q78. In development the Vue dev server runs on port 8080 and Flask on 5000. How do you avoid CORS errors without changing the backend? L2

Use the Vue CLI dev-server proxy so the browser sends requests to the same origin as the page, and the dev server forwards `/api` to Flask.

```
// vue.config.js
module.exports = {
  devServer: { proxy: { "/api": { target: "http://localhost:5000", changeOrigin:
true } } },
};
```

The browser sees only same-origin calls, so no CORS preflight is needed. In production, serve the built files and API from the same domain.

Q79. How is the built Vue app served together with Flask in production? L1

Run `npm run build` to produce static files in `dist/`. Flask can serve these as static files, with a route returning `index.html` for the root (and for all unknown paths when using history mode), while `/api/*` routes return JSON. Alternatively put Nginx in front: static files straight from disk and `/api` proxied to Flask.

Q80. Jinja and Vue both use {{ }}. How do you avoid the clash when Flask renders an HTML page that uses Vue from the CDN? L2

Either serve the Vue page as a static file (not through Jinja), wrap the Vue part in `{% raw %} … {% endraw %}`, or change Vue's delimiters with `delimiters: ["[[", "]]"]` in the Vue instance. The cleanest solution is what we do in an SPA: Flask serves `index.html` as a static file and sends only JSON, so Jinja never touches the Vue templates.

Q81. How do you handle a 401 response from the API in the frontend? L2

A 401 means the token is missing, invalid or expired, so the app should clear the stored token and user from Vuex and localStorage and redirect to the login page, ideally in one central Axios or fetch wrapper. A 403 means the user is logged in but not allowed, so show a "not permitted" message and do not log out.

Q82. How do you implement pagination and search from the frontend? L2

Send `page`, `per_page` and `q` as query parameters, `fetch("/api/lots?page=2&q=park")`, and let the backend apply `LIMIT/OFFSET` (for example `paginate()` in Flask-SQLAlchemy) and return the items with total count. The frontend keeps the page number in `data` or in the route query, and a debounced watcher on the search text reloads the first page. This keeps payloads small and fast.

Q83. How do you upload a file or submit a form from Vue to Flask? L2

For a file use `FormData`: `const fd = new FormData(); fd.append("file", this.$refs.file.files[0]); await axios.post("/api/upload", fd);` and do not set Content-Type manually, so the browser adds the multipart boundary. In Flask read it with `request.files["file"]`. For plain forms use `@submit.prevent` on the form and send JSON.

Q84. How do you configure the API base URL for development and production in Vue CLI? L2

Put it in environment files such as `.env.development` and `.env.production`; Vue CLI exposes only variables prefixed with `VUE_APP_`, for example `VUE_APP_API=http://localhost:5000`, and you read them as `process.env.VUE_APP_API`. This keeps URLs out of the code, and secrets must never go there because they end up in the public bundle.

Q85. Why call APIs from a Vuex action instead of directly in a component? L2

Actions centralise async logic and error handling, avoid repeating the same fetch in several components, and make the data available to every component through the store. The action calls the

API and commits a mutation with the result, so components only dispatch the action and read state. For simple local data used by only one component, calling in the component is fine.

Q86. What is the difference between how Jinja and Vue render? L1

Jinja renders on the server: Flask fills in the template and sends finished HTML. Vue renders in the browser from JavaScript and JSON, so navigation and updates need no full reload. In my project Jinja is not used for pages, except possibly for email or PDF report templates generated on the server.

Components, props and events

Q87. How do parent and child components communicate in Vue? L1

Down with props, up with events. The parent passes data through props (`<child :title="t">`), and the child sends a message by calling `this.$emit("saved", payload)`, which the parent listens to with `@saved="onSaved"`. This one-way flow keeps data changes traceable.

Q88. Why shouldn't a child mutate a prop? L1

Props follow one-way data flow: the parent owns the data. If a child changes it, the parent's state is not updated, and the next re-render overwrites the change; Vue also warns about it. Instead, copy the prop into local `data` when you need a local editable copy, or emit an event so the parent updates its own state.

Q89. Write a child component with a prop and an emitted event, and the parent that uses it. L2

```
Vue.component("lot-card", {
  props: { lot: { type: Object, required: true } },
  template: `<div>{{ lot.name }}
    <button @click="$emit('book', lot.id)">Book</button></div>`,
});
```

```
<lot-card v-for="l in lots" :key="l.id" :lot="l" @book="book"></lot-card>
```

`book(id)` in the parent receives the emitted id. Event names are best written in kebab-case in templates.

Q90. What are slots? L1

Slots let a parent pass template content into a child component. A default `<slot></slot>` renders whatever is placed between the child's tags, named slots (`<slot name="header">` with `<template v-slot:header>`) place content in specific spots, and scoped slots let the child pass data back to the slot content. They make components flexible, like a card or modal with customisable content.

Q91. How do you validate props? L2

Declare props as an object with types, defaults and validators.

```
props: {
  count: { type: Number, default: 0 },
  status: { type: String, validator: v ⇒ ["open", "closed"].includes(v) },
  user: { type: Object, default: () ⇒ ({}) },
}
```

Defaults for objects and arrays must be returned from a function. Invalid props only produce a console warning in development.

Q92. What is the difference between global and local component registration? L1

`Vue.component("name", { ... })` registers globally, so it can be used in any template but is always bundled. Local registration uses `components: { Name }` inside a component or view, so it is available only there and works better with tree shaking. Prefer local for components used in few places.

Q93. How does v-model work on a custom component in Vue 2? L2

By default the component receives a `value` prop and emits an `input` event, so `<my-input v-model="name">` is shorthand for `:value="name" @input="name = $event"` . Inside the component you emit `this.$emit("input", newValue)` . You can rename these with the `model: { prop, event }` option. Vue 2 also has the `.sync` modifier (`:title.sync="t"` with `$emit("update:title", v)`); in Vue 3 `v-model` uses `modelValue` and `update:modelValue` .

Q94. How do two sibling components share data? L1

Move the shared state to their common parent and pass it down as props with events up; for many components, or deep trees, use a Vuex store. An event bus (a separate Vue instance with `$on` and

`$emit`) works in small apps but becomes hard to track, and it was removed in Vue 3.

Q95. What is provide/inject and when would you use it instead of props? L2

`provide` in an ancestor makes values available, and `inject` in any descendant reads them without passing props through each level (prop drilling). It is useful for library-style components such as tabs, but the data flow is less visible and the injected values are not reactive by default in Vue 2, so for app-wide state a Vuex store is clearer.

Q96. What does `scoped` do in a single-file component's style tag? L2

`<style scoped>` adds a unique data attribute to the component's elements and rewrites the CSS selectors so that the styles apply only to this component and do not leak. Child component root elements are also affected by the parent's scoped styles. To style deeper inside a child you use a deep selector (`>>>` or `/deep/` in Vue 2).

Q97. What is a single-file component (SFC)? L1

A `.vue` file that contains a `<template>`, a `<script>` that exports the component options, and a `<style>`. A build tool such as webpack with vue-loader compiles it. SFCs keep markup, logic and styles of one component together and are why we need Vue CLI rather than the plain CDN.

Q98. What does the `key` attribute do on a component, and how can it be used to reset one? L2

`key` identifies the component to Vue's diff. If the key changes, Vue destroys the old instance and creates a new one, so its internal state is reset. For example `<edit-form :key="user.id">` gives a fresh form each time the selected user changes.

Q99. What is a dynamic component and when do you use it? L2

`<component :is="currentTab"></component>` renders the component whose name or object is in `currentTab`, useful for tabs or wizards. Wrap it in `<keep-alive>` to keep a deactivated component's state instead of destroying it.

Q100. How would you build a reusable modal that lets the parent decide its content and handle close? L2

Make a `Modal` component with a `<slot>` for the body, a `visible` prop, and a `close` event. The parent controls `visible` with its own data and listens to `@close`.

```
<modal :visible="show" @close="show = false">
  <p>Delete this lot?</p>
</modal>
```

This keeps the modal generic: the content comes from the slot and the state remains in the parent.

Q101. Why should you avoid this.$parent and this.$refs for communication? L2

`$parent` couples the child tightly to a specific parent, so the component can no longer be reused elsewhere. `$refs` are not reactive and exist only after mounting. Props and events keep dependencies explicit and make components testable; use `$refs` only for direct DOM actions like focus.

Vue Router

Q102. What is Vue Router and why use it in an SPA? L1

Vue Router (version 3 for Vue 2) maps URL paths to components so a single-page app can have multiple "pages" with working back and forward buttons and shareable links, without reloading. You declare routes, add `<router-view>` where the matched component should appear, and navigate with `<router-link>` or `this.$router.push`.

Q103. Show a basic Vue Router 3 setup. L2

```
import Vue from "vue";
import VueRouter from "vue-router";
Vue.use(VueRouter);
const routes = [
  { path: "/", component: Home },
  { path: "/lot/:id", name: "lot", component: Lot, props: true },
  { path: "*", component: NotFound },
];
export default new VueRouter({ mode: "history", routes });
```

The router is passed to `new Vue({ router, render: h => h(App) })`. The wildcard route must be last.

Q104. What is the difference between router-link and an anchor tag? L1

`<router-link to="/lots">` navigates inside the app through the history API without a page reload and keeps the Vue state, and it adds active classes automatically. A normal `<a href>` makes the browser request a new page from the server, reloading the whole app and losing the in-memory state.

Q105. What is the difference between hash mode and history mode? L1

Hash mode puts the route after `#` (`/#/lots`), which the server never sees, so it works with any static server. History mode uses clean paths (`/lots`) via the HTML5 history API, but a page refresh on `/lots` makes the browser ask the server for that path, so the server must fall back to `index.html` for unknown routes. In Flask add a catch-all route that returns `index.html` .

Q106. How do you read dynamic route parameters and query strings, and what is the difference? L2

Params are part of the path: for `/lot/:id` read `this.$route.params.id` . Query is after `?` : for `/lots?page=2` read `this.$route.query.page` . Both come back as strings. If the same component is reused when only the param changes, the component is not recreated, so watch `$route` or use `beforeRouteUpdate` .

Q107. How do you navigate programmatically with parameters? L2

```
this.$router.push({ name: "lot", params: { id: 5 } });
this.$router.push({ path: "/lots", query: { page: 2 } });
this.$router.replace("/login");
this.$router.go(-1);
```

`push` adds a history entry, `replace` replaces the current one, and `go(-1)` goes back. When using `path` the `params` option is ignored, so use `name` with `params` , or build the path string yourself.

Q108. What are navigation guards? Show how you protected the admin pages. L2

Guards run before a navigation completes and can allow, redirect or cancel it. A global `beforeEach` checks the token and role.

```
router.beforeEach((to, from, next) => {
  const token = localStorage.getItem("token");
  const role = localStorage.getItem("role");
  if (to.meta.requiresAuth && !token) return next("/login");
  if (to.meta.role && to.meta.role !== role) return next("/");
  next();
});
```

Routes carry `meta: { requiresAuth: true, role: "admin" }`. The callback must always call `next()` exactly once.

Q109. If you hide admin routes using a guard, is the app secure? L2

No. A guard only controls the UI, and the user can edit the JavaScript, the localStorage values, or call the API directly with curl. Real security is on the server: every admin API verifies the token and the role (`@roles_required("admin")`) and returns 401 or 403. Front-end guards just give a good user experience.

Q110. What are nested routes? L1

A route can have `children`, which render inside a second `<router-view>` in the parent component, like a dashboard layout with a sidebar and a changing content area. Child paths are relative to the parent, for example `/admin/users`.

Q111. How do you lazy-load a route and why? L2

Use a dynamic import: `{ path: "/admin", component: () => import("./views/Admin.vue") }`. Webpack splits that component into a separate chunk that downloads only when the route is visited, so the first load is smaller and faster. It is a good fit for rarely used pages like admin reports.

Q112. How do you set up redirects, a 404 page and named routes? L2

Use `{ path: "/home", redirect: "/" }` for redirects, `{ path: "*", component: NotFound }` as the last route to catch unknown paths, and `name: "lot"` on a route to navigate with `{ name: "lot", params: { ... } }` so changes to the path do not break links.

Q113. Which options do you have to pass route params as props, and why? L2

Set `props: true` on the route so `:id` arrives as a prop, `props: route ⇒ ({ page: Number(route.query.page) || 1 })` for custom mapping. This decouples the component from `$route`, so it can be reused and tested with plain props.

Q114. What does an exam-style route like `/slot/:flag` with a query `?offset=1` render? How do you reason about it? L2

Match the path to a route first, then read `$route.params.flag` as the string "true" (not the boolean) and `$route.query.offset` as the string "1". Then follow the template of that component, remembering that strings are always truthy and that `{{ }}` shows what is in params and query exactly. Check whether the path matches a nested child, or falls to the wildcard.

Vuex state management

Q115. What is Vuex and why did you use it? L1

Vuex (version 3 for Vue 2) is a centralised store for shared state with strict rules for how it changes. Without it, passing the logged-in user, token or cached lists through many levels of props and events gets messy. With Vuex any component reads the same state and updates it only through mutations, so state changes are predictable and debuggable in devtools. For a small app with a few components, local state and props are enough.

Q116. Explain state, getters, mutations and actions. L1

`state` is the single source of data. `getters` are computed properties of the store (such as a filtered list). `mutations` are the only way to change state, and they must be synchronous. `actions` can be asynchronous (API calls), and they commit mutations. Components call `this.$store.dispatch("fetchLots")`, an action calls the API and does `commit("SET_LOTS", data)`, and the mutation assigns `state.lots`.

Q117. What is the correct sequence for updating Vuex state after an asynchronous API call? L1

The component dispatches an action; the action performs the async work (API request) and waits for it; the action commits a mutation with the result; the mutation changes the state; the components that use that state re-render. Only the mutation touches state directly.

Q118. Why must mutations be synchronous? L1

Devtools record a snapshot before and after each mutation, which only works if the state change finishes at once. An asynchronous change would make it impossible to know when and why the state changed. Put async code in actions, and let the action commit when the result is ready.

Q119. Write a Vuex store with state, a mutation, a getter and an async action for loading lots. L2

```
export default new Vuex.Store({
  state: { lots: [], token: localStorage.getItem("token") },
  getters: { available: s ⇒ s.lots.filter(l ⇒ l.free > 0) },
  mutations: {
    SET_LOTS(state, lots) { state.lots = lots; },
  },
  actions: {
    async fetchLots({ commit }) {
      const res = await fetch("/api/lots");
      commit("SET_LOTS", await res.json());
    },
  },
});
```

A component uses `this.$store.state.lots`, `this.$store.getters.available` and `this.$store.dispatch("fetchLots")`.

Q120. How do you use mapState, mapGetters, mapMutations and mapActions? L2

They generate component members from the store so you do not write `this.$store…` repeatedly.

```
import { mapState, mapActions } from "vuex";
computed: { …mapState(["lots"]) },
methods: { …mapActions(["fetchLots"]) },
```

State and getters map to `computed`, while mutations and actions map to `methods`. The object spread operator is needed to combine them with other local properties.

Q121. In a component you must call the mutation `update_total_cost` with a value. How? L2

Use `this.$store.commit("update_total_cost", value)` or `…mapMutations(["update_total_cost"])` and then call `this.update_total_cost(value)`.

Mutations are called with `commit`, never `dispatch`; `dispatch` is for actions. A mutation receives `(state, payload)`, and when you need several values pass them as one object.

Q122. How do you keep the login state after a page refresh with Vuex? L2

Vuex state lives in memory and is lost on refresh. So when logging in, also save the token (and role) to localStorage, and initialise the store from it: `token: localStorage.getItem("token")`. On logout clear both. A plugin like `vuex-persistedstate` can automate this.

Q123. When would you not use Vuex? L1

When state is local to one component or passes only between a parent and a child, props and events are simpler. Vuex adds boilerplate, so it makes sense for data shared across many unrelated components: the current user, token, cart, notification counts and cached lists.

Q124. What are Vuex modules and namespacing? L2

Modules split a large store into several parts such as `auth` and `lots`, each with its own state, mutations, actions and getters. With `namespaced: true` they are accessed as `auth/login`, so action names do not clash: `this.$store.dispatch("auth/login", creds)`.

Q125. Why is it wrong to modify store state directly from a component? L2

Direct changes bypass mutations, so devtools cannot track them, and in strict mode (`strict: true`, used in development) Vuex throws an error. Keeping all writes in mutations makes it easy to find who changed what. The component should commit a mutation or dispatch an action.

Q126. What changes in newer Vue regarding Vuex? L1

In newer Vue (Vue 3) the officially recommended store is Pinia, which has no mutations (you change state in actions or directly), simpler syntax and better TypeScript support. Vuex 4 still exists for Vue 3. Our course uses Vuex with Vue 2, so I explain state, getters, mutations and actions.

Q127. How would you make an action return something to the component, such as a success flag? L2

An action always returns a promise, so you can return values and the component can `await` it.

```
async login({ commit }, creds) {
  const res = await axios.post("/login?include_auth_token", creds);
  commit("SET_TOKEN", res.data.response.user.authentication_token);
  return true;
}
```

In the component: `const ok = await this.$store.dispatch("login", creds); if (ok) this.$router.push("/");`. If the call throws, the rejection reaches the component's try/catch.

Q128. Reading from a store getter versus computing in the component: which is better? L2

Put derived data that several components need in a getter, so the logic lives once and is cached while the state is unchanged. Use a component-level computed for logic that is specific to one component, such as sorting for its own view.

Token authentication and JWT

Q129. What is the difference between session-based and token-based authentication? L1

In session auth the server stores the session data and gives the browser a session id in a cookie, which is sent automatically and looked up on each request. In token auth the server issues a signed token after login, the client stores it and sends it in a header on every request, and the server verifies it without keeping session state. Tokens suit SPAs, mobile clients and multi-server setups; sessions are simple for server-rendered apps.

Q130. Which authentication did you use in your project and how does the login flow work? L1

The user posts credentials to the login endpoint, the server verifies the hashed password and returns a token and role. Vue saves the token (Vuex plus localStorage), and every API call sends it in the `Authentication-Token` (Flask-Security) or `Authorization: Bearer` header (JWT). The server decorator `@auth_required("token")` or `@jwt_required()` checks it, and role decorators restrict admin routes. Logout removes the token from the client.

Q131. What is a JWT and what are its parts? L1

A JSON Web Token is `header.payload.signature`, three Base64URL parts. The header names the algorithm (like HS256), the payload carries claims such as `sub`, `role`, `iat` and `exp`, and the

signature is made with the server's secret key. The payload is only encoded, not encrypted, so anyone can read it: never put passwords or secrets in it. The signature stops tampering.

Q132. How does the server verify a JWT? L1

It recomputes the signature over the header and payload with its secret key (HS256) or checks it with a public key (RS256), compares it with the signature in the token, and checks claims such as `exp`. If the signature is wrong or the token expired, it rejects with 401. No database lookup is needed, which is why JWTs are called stateless.

Q133. How does Flask-Security-Too token authentication work? L2

You enable it with a `SQLAlchemyUserDatastore`; the login endpoint (`POST /login?include_auth_token`, JSON body) returns an authentication token for the user. The client sends it in the `Authentication-Token` header, and routes use `@auth_required("token")` plus `@roles_required("admin")`. The token is a signed value checked by the server, and it can expire with the `SECURITY_TOKEN_MAX_AGE` setting.

Q134. Show a protected Flask route with role-based access using Flask-Security. L2

```
from flask_security import auth_required, roles_required, current_user

@app.route("/api/admin/users")
@auth_required("token")
@roles_required("admin")
def users():
    return jsonify([u.email for u in User.query.all()])
```

`auth_required` returns 401 when the token is missing or invalid, and `roles_required` returns 403 when the user lacks the role. `current_user` gives the logged-in user, so I can also scope data to the user.

Q135. Show a JWT login and a protected route using Flask-JWT-Extended. L2

```
from flask_jwt_extended import create_access_token, jwt_required, get_jwt_identity

@app.post("/api/login")
def login():
    user = check_credentials(request.json)
```

```
    if not user:
        return {"msg": "bad credentials"}, 401
    return {"token": create_access_token(identity=str(user.id))}

@app.get("/api/me")
@jwt_required()
def me():
    return {"id": get_jwt_identity()}
```

The token lifetime is set with `JWT_ACCESS_TOKEN_EXPIRES`, and the signing key with `JWT_SECRET_KEY`.

Q136. Where do you store the token on the client, and what are the trade-offs? L1

localStorage is simple and works across tabs, but any XSS can read it. An HttpOnly, Secure, SameSite cookie cannot be read by scripts, but is sent automatically and so needs CSRF protection. I choose localStorage plus Vuex with a short expiry and escaped output, and explain this trade-off.

Q137. Why do tokens have an expiry, and what are refresh tokens? L1

If a token is stolen, an expiry limits how long it can be used. A short-lived access token (minutes) is paired with a longer-lived refresh token that is used only to get a new access token. If the access token expires, the client uses the refresh token, or sends the user back to the login page. This balances security and convenience.

Q138. How would you log out with a stateless JWT, and what is the limitation? L2

On the client, delete the token from Vuex and localStorage. The server cannot forget a stateless token, so it stays valid until it expires; to truly revoke it you keep a blocklist of revoked token ids (jti) in Redis until they expire, or use short lifetimes. That is a trade-off between stateless simplicity and revocation.

Q139. How are passwords stored and what is a rainbow table? L1

Passwords are never stored in plain text; they are salted and hashed with a slow algorithm such as bcrypt, PBKDF2 or Argon2 (Flask-Security does this through passlib). A rainbow table is a precomputed table of hashes for common passwords, used to reverse unsalted hashes. A random salt per password makes such tables useless, and a slow hash makes brute force expensive.

Q140. What is the difference between authentication and authorisation, and what is RBAC? L1

Authentication checks who you are (login with credentials or a token), authorisation checks what you may do. Role-Based Access Control grants permissions through roles such as admin, professional and customer, and each endpoint specifies which roles are allowed. It is easier to manage than permissions assigned per person.

Q141. What is the difference between status codes 401 and 403? L1

401 Unauthorized means that authentication is missing or invalid, such as no token, a bad token or an expired token, so the client should log in again. 403 Forbidden means that the user is authenticated but not allowed to do this action, for example a customer calling an admin endpoint.

Q142. What is CSRF and why is token-in-header authentication largely safe against it? L2

Cross-Site Request Forgery tricks a logged-in user's browser into sending an unwanted request to your site, such as a malicious page containing `<img src="http://site/delete?id=1">`. It works because browsers attach cookies automatically. A token stored in localStorage and sent manually in a header is not attached by the browser to a cross-site request, so the attack fails. If you use cookies for auth, use SameSite cookies and a CSRF token; the CSRF token is a random value the attacker's page cannot read, which the server must check.

Q143. How does Vue escape output and what is XSS? L1

Cross-Site Scripting injects attacker script into your page, for example through a comment containing `<script>`. Vue's `{{ }}` interpolation escapes HTML by default, so it is safe; `v-html` inserts raw HTML and must never be used with untrusted user content. Server-side, return JSON, not HTML, and sanitise rich text if you need to allow it.

Q144. How do you prevent SQL injection in your Flask backend? L2

Use the SQLAlchemy ORM or parameterised queries so user input is sent as bound parameters, never concatenated into SQL strings. For example `User.query.filter_by(email=email)` is safe, whereas `db.session.execute("... WHERE email='" + email + "'")` is vulnerable.

Q145. What would you check to make a token-based API secure? L2

Use HTTPS, hash passwords with a salt, keep the secret key out of source control (environment variable), set a short token expiry, validate input, check roles on the server for every endpoint, check

object ownership (a user can only see their own bookings), restrict CORS origins, and rate-limit the login endpoint.

Q146. Where should the secret key or other secrets live? L2

In environment variables or a config file that is not committed to git, for example `app.config["SECRET_KEY"] = os.environ["SECRET_KEY"]`. A leaked signing key lets anyone forge tokens. Frontend environment variables are public in the bundle, so no secrets belong there.

Q147. A user changes `role` in localStorage to "admin". What happens? L2

The frontend might show the admin menu, but every API call goes to the server, which determines the role from the verified token and the database, not from what the client says, so admin endpoints return 403. This is why the role in localStorage is only for UI, and the server must be the real enforcement.

CORS and cookies

Q148. What is CORS and why does it exist? L1

Browsers apply the same-origin policy: JavaScript on one origin cannot read responses from another origin. CORS (Cross-Origin Resource Sharing) is the mechanism by which a server tells the browser, through `Access-Control-Allow-*` response headers, which other origins may read its responses. It exists to protect users' data on other sites, and it is enforced by the browser, not the server.

Q149. What is an origin? Which of these origins are the same as https://example.com? L1

An origin is the combination of scheme, host and port. `https://example.com` and `https://example.com/page` are the same origin, but `http://example.com` (different scheme), `https://api.example.com` (different host) and `https://example.com:8443` (different port) are different origins. By default the browser allows fetch calls that read the response only to the same origin; others need CORS headers.

Q150. How do you prevent CORS errors in your project? L1

In development, either configure the backend with Flask-CORS to allow the Vue dev origin, or use the dev-server proxy so calls become same-origin. In production serve the frontend and API from the

same domain, or allow exactly the frontend origin. Also make sure the `Authorization` or `Authentication-Token` header is allowed, because custom headers trigger a preflight.

Q151. Show how you enable CORS in Flask. L2

```
from flask_cors import CORS
CORS(app, resources={r"/api/*": {"origins": ["http://localhost:8080"]}},
     allow_headers=["Content-Type", "Authentication-Token", "Authorization"])
```

`CORS(app)` alone allows all origins, which is fine for local testing but should be limited in production. Flask-CORS adds the `Access-Control-Allow-Origin` header and answers the OPTIONS preflight requests.

Q152. What is a preflight request? L1

For non-simple cross-origin requests (methods like PUT or DELETE, a JSON content type, or custom headers such as Authorization) the browser first sends an `OPTIONS` request asking if the real request is allowed. The server answers with `Access-Control-Allow-Origin`, `Access-Control-Allow-Methods` and `Access-Control-Allow-Headers`; only then the browser sends the actual request. `Access-Control-Max-Age` lets the browser cache the preflight result.

Q153. Why does `Content-Type: application/json` trigger a preflight but a plain form POST doesn't? L2

The CORS "simple request" rules allow GET, HEAD and POST with only a few content types (`application/x-www-form-urlencoded`, `multipart/form-data`, `text/plain`) and no custom headers. `application/json` and headers such as `Authorization` are outside that list, so the browser must ask permission first with an OPTIONS preflight.

Q154. Is CORS a security feature of your server? Will it stop curl or Postman? L2

No. CORS is enforced only by browsers to protect the user; the server still receives the request and curl or Postman ignore CORS completely. So CORS is not access control. Authentication and authorisation must still be done on the server.

Q155. You get "No 'Access-Control-Allow-Origin' header is present" in the browser console while the same request works in Postman. What do you check? L2

It means that the response lacked the CORS header, so I check that the Flask-CORS or the headers are configured for this route and origin, that the preflight (OPTIONS) is handled and returns 200/204, and that the server did not crash with a 500 (error responses may lack CORS headers). Also check that the origin string matches exactly, including port and scheme. It works in Postman only because Postman does not enforce CORS.

Q156. How do cookies work with CORS? L2

For a cross-origin request to send and accept cookies, the client must opt in (`fetch(url, { credentials: "include" })` or `axios.defaults.withCredentials = true`), and the server must send `Access-Control-Allow-Credentials: true` with a specific origin, because `*` is not allowed with credentials. The cookie's `SameSite` setting also must permit cross-site sending.

Q157. Explain the cookie attributes HttpOnly, Secure and SameSite. L1

`HttpOnly` hides the cookie from JavaScript (`document.cookie`), reducing the impact of XSS. `Secure` sends it only over HTTPS. `SameSite` controls cross-site sending: `Strict` never sends it on cross-site requests, `Lax` sends it on top-level navigations with safe methods, and `None` (which requires `Secure`) allows all cross-site use. Together they protect against theft and CSRF.

Q158. What is a session cookie? L1

A session cookie has no expiry date, so the browser normally discards it when the browser session ends, and it commonly holds just a session id that the server maps to stored data. Persistent cookies have an `Expires` or `Max-Age` . A session cookie is sent automatically with every request to its domain, which is why CSRF protection matters.

Q159. Why is `*` not allowed as an allowed origin when using credentials? L2

If any website could read credentialed responses, a malicious site could make the user's browser send their cookies and read the private response. So the spec requires an explicit origin in `Access-Control-Allow-Origin` when `Allow-Credentials` is true. Flask-CORS with `supports_credentials=True` echoes the specific allowed origin.

Q160. How would a malicious site exploit an endpoint `GET /delete?id=1` and how do you stop it? L2

The attacker page includes `<img src="http://example1.com/delete?id=1">`; the browser sends the request with the victim's cookies, so the delete happens (CSRF). Prevent it by never using GET for state changes (use DELETE or POST), by requiring a CSRF token or a token header that a cross-site page cannot add, and by setting `SameSite` on cookies.

API design

Q161. What is a REST API and what are its main principles? L1

REST is an architectural style where resources are identified by URLs and manipulated through standard HTTP methods. It is stateless (each request carries all needed information), uses a uniform interface, and transfers representations such as JSON. Example: `GET /api/lots`, `POST /api/lots`, `GET /api/lots/3`, `PUT /api/lots/3`, `DELETE /api/lots/3`.

Q162. Explain the HTTP methods and which are safe or idempotent. L1

GET reads and is safe and idempotent. POST creates and is neither safe nor idempotent. PUT replaces a resource and is idempotent, as is DELETE; PATCH makes a partial update. Idempotent means repeating the request gives the same final state, which is why clients can safely retry PUT and DELETE.

Q163. Which HTTP status codes do you use and when? L1

200 OK, 201 Created for a successful POST, 204 No Content (for example a DELETE), 400 Bad Request for invalid input, 401 when not authenticated, 403 when not permitted, 404 when not found, 409 for a conflict such as a duplicate or an already-booked spot, 422 for validation errors in some APIs, and 500 for a server bug. Returning correct codes lets the frontend handle errors properly.

Q164. Show a Flask-RESTful resource with GET and POST. L2

```
from flask_restful import Api, Resource, fields, marshal_with
api = Api(app)
lot_fields = {"id": fields.Integer, "name": fields.String, "price": fields.Float}

class LotList(Resource):
    @marshal_with(lot_fields)
    def get(self):
        return Lot.query.all()
```

```
    def post(self):
        d = request.get_json()
        lot = Lot(name=d["name"], price=d["price"])
        db.session.add(lot); db.session.commit()
        return {"id": lot.id}, 201

api.add_resource(LotList, "/api/lots")
```

`marshal_with` converts SQLAlchemy objects to JSON-safe dictionaries.

Q165. Why do you need serialisation (marshalling) in an API? L1

SQLAlchemy objects cannot be turned into JSON directly. We convert them to dictionaries with chosen fields, using `marshal_with`, Marshmallow schemas, or a `to_dict()` method, so we control what is exposed (never the password hash) and the format the frontend gets.

Q166. How do you validate input in your API? L2

Read the JSON with `request.get_json()`, check required fields, types and ranges, and return `400` with a clear JSON message when invalid, before touching the database. Validate on the server even if the form is validated in Vue, because the client can be bypassed. Flask-RESTful's `reqparse` or Marshmallow can automate it.

Q167. How would you design pagination, filtering and sorting in the API? L2

Use query parameters: `GET /api/lots?page=2&per_page=10&q=mall&sort=price`. The server applies filters and `paginate()`, and returns a body like `{ "items": [ ... ], "page": 2, "total": 57 }`. Limiting `per_page` to a maximum protects the server.

Q168. Why is returning a different status code for each failure better than always returning 200 with an error message? L2

HTTP clients, browsers, caches and tools like Axios understand status codes: with Axios a 4xx or 5xx rejects the promise, so error handling goes in the catch block. Always returning 200 hides failures and makes caching, monitoring and debugging harder.

Q169. How do you handle two users booking the same slot at the same time? L2

Check and update in one database transaction, and let the database enforce it, for example with a unique constraint or `SELECT ... FOR UPDATE` style locking or an atomic conditional update (`UPDATE ... WHERE status='free'`) and check the affected row count. If the second user's attempt finds the slot taken, return `409 Conflict` . A simple "check then insert" without a transaction has a race condition. SQLite serialises writes, but the logic still needs to be atomic.

Q170. What is API versioning and why would you do it? L1

Versioning (like `/api/v1/lots`) lets you change the API without breaking existing clients, which keep using the old version. For a course project it is optional, but I would mention it as a good practice when clients such as a mobile app depend on the API.

Q171. Difference between PUT and PATCH, and which does your update endpoint use? L2

PUT replaces the whole resource, so the client sends all fields; PATCH changes only the fields sent. Many projects use PUT for edits and just update the supplied fields. State which you use and be consistent; both should return the updated resource or 204.

Q172. How do you document or test your API? L1

Test with Postman, curl or the browser, and write down endpoint, method, body, auth header and response codes (for example in a README or an OpenAPI/Swagger file). Consistent documentation helps the frontend developer and the examiner understand the API contract.

Q173. What is rate limiting and where would it be useful in your app? L2

Rate limiting caps how many requests a client can make in a period, protecting against abuse and brute force. It is most useful on the login endpoint and on expensive endpoints such as export. Flask-Limiter can do it, and it can use Redis to store the counters.

Q174. How can you make an API "stateless" and why is it good for scaling? L2

Do not keep per-client data in server memory between requests; each request carries the token and everything needed. Then any server instance can handle any request, so you can add more servers behind a load balancer. Shared state lives in the database or Redis.

Q175. What is the purpose of an API for the frontend/backend split? L1

The API is a contract between the two sides: the backend exposes data and operations as JSON endpoints, and any client (Vue app, mobile app, scripts) can use them. The teams can work independently and the backend can be reused, which is the reason to move from Jinja-rendered pages to Vue.

Redis and caching

Q176. What is Redis and what did you use it for? L1

Redis is an in-memory key-value data store that supports strings, hashes, lists, sets and sorted sets, with optional persistence. In the project I used it in two roles: as the message broker (and result backend) for Celery, and as a cache for expensive API responses. Because data is in RAM, reads and writes are very fast.

Q177. Why is Redis fast, and how is it different from a relational database like SQLite? L1

Redis keeps data in memory, uses simple data structures with direct key access, and avoids query parsing and disk reads. A relational database stores data durably on disk with schemas, joins and transactions, and handles complex queries. Redis is used for fast, often temporary data (cache, queues, counters), the SQL database as the source of truth.

Q178. Show some basic Redis commands. L1

```
redis-cli ping            # PONG
redis-cli set lots:1 "data"
redis-cli get lots:1
redis-cli expire lots:1 60
redis-cli ttl lots:1      # seconds left
redis-cli keys "*"
redis-cli del lots:1
```

`SETEX key seconds value` sets a value with expiry at once. `KEYS *` is fine for debugging but should not be used on big production databases (use `SCAN`).

Q179. What is a cache and why do you cache? L1

A cache stores the result of an expensive operation, such as a database query or computation, so the next request can be answered quickly. This reduces response time and database load. The cost is that

data may be stale, so we need an expiry and invalidation strategy.

Q180. How did you implement caching in Flask, and how would you show it works? L2

With Flask-Caching using a Redis backend.

```
from flask_caching import Cache
cache = Cache(app, config={"CACHE_TYPE": "RedisCache",
                           "CACHE_REDIS_URL": "redis://localhost:6379/1",
                           "CACHE_DEFAULT_TIMEOUT": 300})

@app.get("/api/stats")
@cache.cached(timeout=60)
def stats():
    return jsonify(compute_stats())
```

To prove it works: call the endpoint twice and show the second response is much faster, add a `print` in `compute_stats` that appears only once, and show the key with `redis-cli keys "*"` and its decreasing `ttl`.

Q181. What is the difference between `@cache.cached()` and `@cache.memoize()`? L2

`cached` builds the key from the request path (by default not from function arguments, and you can use `query_string=True` to include the query), so it suits a view with no arguments. `memoize` includes the function's arguments in the key, so `get_user(1)` and `get_user(2)` are cached separately, and you can invalidate them with `cache.delete_memoized(func, args)`.

Q182. Two requests go to /compute/10/20 and /compute/5/10 within 60 seconds, and the view is decorated with `@cache.cached(timeout=60)`. What does the second request return? L2

If the route has URL parameters, the default key is based on the request path, so each URL path gets its own entry: /compute/10/20 and /compute/5/10 are different keys, so each computes its own result on the first call and is cached after. If you wrongly used a fixed `key_prefix` (the same key for both), the second request would return the first request's cached response. Reading the key rule carefully decides the answer.

Q183. Why do we set an expiry (TTL) for a cache? L1

The expiry bounds how stale data can be and keeps memory from filling up. Fast-changing data such as available slots needs a short TTL (a few seconds or minutes), while rarely changing data such as a

category list can use a long one. I choose the TTL according to how often the data changes and how bad stale data is.

Q184. What is cache invalidation, and how did you handle it? L1

Invalidation means removing or updating cached data when the underlying data changes, otherwise users see stale values. After a successful POST, PUT or DELETE I delete the related keys with `cache.delete("lots")` or `cache.delete_memoized(get_lots)`, so the next read repopulates it. A TTL acts as the fallback in case invalidation is missed.

Q185. How do you design cache keys so one user doesn't see another user's data? L2

Include everything that makes the response different in the key: the user id (or role), the path, and the query parameters. For per-user data use `key_prefix=lambda: f"bookings:{current_user.id}"` or `memoize` with the user id as an argument. A key that omits the user would serve the first user's data to everybody, which is a serious leak.

Q186. What is a cache hit and a miss, and what is the cache-aside pattern? L1

A hit means the data is found in the cache; a miss means it is not, and we must load it from the database. In cache-aside the application first checks the cache; on a miss it reads the database, stores the result in the cache with a TTL and returns it. Writes update the database and invalidate the cache key.

Q187. Show a manual cache-aside implementation with the redis library. L2

```
import json, redis
r = redis.Redis(host="localhost", port=6379, db=1)

def get_lots():
    cached = r.get("lots")
    if cached:
        return json.loads(cached)
    data = [l.to_dict() for l in Lot.query.all()]
    r.setex("lots", 60, json.dumps(data))
    return data
```

After a write I call `r.delete("lots")`. Redis returns bytes, which `json.loads` accepts.

Q188. What should you not cache? L2

Highly dynamic or critical data where staleness is harmful, such as the real-time availability that decides if a booking succeeds (the booking check must hit the database); also responses with user-specific private data unless the key includes the user, and responses to POST or other state-changing requests.

Q189. Redis is in memory, so is the data lost on restart? L1

By default it can be, unless persistence is enabled: RDB takes periodic snapshots, and AOF logs every write. For a cache we can accept loss because the data can be rebuilt from the database; for broker data (Celery queues) loss means pending tasks may disappear, so enable persistence if that matters.

Q190. Is HTTP caching (Cache-Control) the same as Redis caching? L2

No. HTTP caching happens in the browser or intermediate proxies controlled with `Cache-Control`, `ETag` and `Expires` headers, and avoids requests reaching the server at all. Redis caching is server-side: the request arrives, but the app avoids recomputing by reading Redis. They can be used together.

Q191. How do you use two Redis databases or separate Redis roles for broker and cache? L2

Redis has numbered databases (0 to 15 by default), so you can use `redis://localhost:6379/0` for the Celery broker and `redis://localhost:6379/1` for the cache. This keeps keys separate, so `cache.clear()` does not delete queued tasks. In production you might run separate instances.

Q192. A flask app returns the current time with `@cache.cached(timeout=120)`. Visits occur at 0 s, 150 s and 210 s. What times are shown? L2

The first visit computes and shows the time at 0 s. The second at 150 s is after the 120 s expiry, so it is a miss: it recomputes and shows the time at about 150 s and the new entry lasts until 270 s. The third at 210 s is within that window, so it shows the time from 150 s again. So the displayed times are 0, 150 and 150 seconds.

Q193. Which statements about Redis are true? Is it a message broker? L1

Redis can act as a message broker because its lists and publish/subscribe features can carry messages, which is how Celery uses it. It is also a cache and a data store with TTLs. It is not a replacement for a

relational database for complex joins, and it is not "only" a cache.

Celery and asynchronous jobs

Q194. What is Celery and why do you need it? L1

Celery is a distributed task queue for Python. Your Flask app (the producer) puts a task message on a broker such as Redis, and a separate worker process picks it up and runs it in the background. We need it for slow work, such as exporting a CSV, sending emails or building reports, so that the HTTP request returns immediately and does not time out.

Q195. What are the components of a Celery setup? L1

The application that defines tasks and sends them, the broker (Redis or RabbitMQ) that holds the queue of messages, one or more workers that execute tasks, an optional result backend (such as Redis) that stores results and states, and Celery Beat, a scheduler that sends periodic tasks. All of these run as separate processes.

Q196. Show a minimal Celery setup with Flask and one task. L2

```
from celery import Celery

celery = Celery("app", broker="redis://localhost:6379/0",
                backend="redis://localhost:6379/0")

@celery.task
def add(x, y):
    return x + y
```

```
celery -A app.celery worker -l info
```

In the Flask view call `result = add.delay(2, 3)` and return `result.id`. The `-A` argument points to the module and the Celery object.

Q197. What is the difference between a scheduled task and an async (on-demand) task? L1

An async task is triggered by a user or code event, such as the user clicking "Export CSV", through `.delay()`. A scheduled (periodic) task is triggered by time, like the daily reminder at 6 PM or the

monthly report on the 1st, by Celery Beat. Both are executed by the same workers.

Q198. What does Celery Beat do? L1

Beat is a scheduler process that reads a schedule (crontab or interval) and sends the due tasks to the broker at the right time. It does not execute the tasks; a worker must be running too. So for periodic jobs you start the worker and Beat as two processes (or one with `-B` in development).

Q199. How do you schedule a daily reminder at 6 PM IST and a monthly report on the 1st? L2

```
from celery.schedules import crontab
celery.conf.timezone = "Asia/Kolkata"
celery.conf.beat_schedule = {
    "daily-reminder": {"task": "tasks.send_reminders",
                       "schedule": crontab(hour=18, minute=0)},
    "monthly-report": {"task": "tasks.monthly_report",
                       "schedule": crontab(day_of_month=1, hour=8, minute=0)},
}
```

```
celery -A app.celery beat -l info
```

Set the timezone explicitly, otherwise the schedule uses UTC.

Q200. How does the frontend know an async CSV export is finished? L2

The export endpoint starts the task and returns the task id immediately (202). The frontend polls a status endpoint that uses `AsyncResult(task_id).state`, and when it is `SUCCESS` it shows a download link, which hits a download endpoint that serves the file. Alternatives are SSE or a notification by email.

```
@app.get("/api/export/<task_id>")
def status(task_id):
    r = celery.AsyncResult(task_id)
    return {"state": r.state}
```

Q201. What are the Celery task states? L1

PENDING (waiting or unknown), STARTED (if enabled), SUCCESS, FAILURE, RETRY and REVOKED. Note that PENDING is also reported for an unknown task id. The state is stored in the result backend.

Q202. Why do tasks sometimes fail with "working outside of application context", and how do you fix it? L2

The task runs in the worker process, outside a Flask request, so `db`, `current_app` and config are not available automatically. Create the Flask app and push an application context inside the task, or subclass `celery.Task` with a context wrapper.

```
class ContextTask(celery.Task):
    def __call__(self, *a, **kw):
        with app.app_context():
            return self.run(*a, **kw)
celery.Task = ContextTask
```

Q203. How do you handle failures and retries in a task? L2

Use `bind=True` and call `self.retry` on a recoverable error, or `autoretry_for`.

```
@celery.task(bind=True, max_retries=3, default_retry_delay=60)
def send_mail(self, to):
    try:
        smtp_send(to)
    except ConnectionError as e:
        raise self.retry(exc=e)
```

Because a task can run more than once, it should be idempotent, so that a retry does not send duplicates or double-update.

Q204. What is the difference between `.delay()` and `.apply_async()`? L2

`delay(*args, **kwargs)` is a shortcut for `apply_async(args, kwargs)`. `apply_async` accepts options like `countdown=60` (run after 60 seconds), `eta`, `expires`, `queue` and `retry`. Both return an `AsyncResult` immediately without waiting for the task.

Q205. Why can't you call `result.get()` inside a Flask view for the CSV export? L2

`get()` blocks until the task finishes, so the request would hang and time out, which defeats the purpose of a background job. Return the id immediately and let the client poll or be notified.

Q206. Why Redis as a broker, and what are the alternatives? L1

Redis is easy to install and already used as a cache, so the project needs one service for both. RabbitMQ is a dedicated message broker with stronger delivery guarantees and routing features. For a course project Redis is simple and sufficient.

Q207. Why do many guides use WSL for Celery and Redis on Windows? L1

Redis has no official Windows build, and Celery 4 and later dropped official Windows support, as its default prefork pool does not work reliably there. WSL provides a Linux environment where both run normally. (Celery can be run on Windows with `--pool=solo`, only for development.)

Q208. Which situations are best suited for Celery tasks? L1

Long-running or heavy work that the user should not wait for: report generation, data analysis, sending bulk emails, image processing, calling slow external services, and scheduled jobs. It is not needed for quick operations like a simple database read.

Q209. Arrange these steps for a heavy analysis requested by millions of users: user request, return a task id, put the task in the queue, worker processes it, store the result, client fetches the result. L2

The client sends the request; the API validates it and puts a task on the broker queue; the API immediately returns a task id (HTTP 202); a worker takes the task and runs it; the result is stored in the result backend or database; the client polls (or is notified) and fetches the result. The key point is the API never blocks on the heavy work, and you scale by adding workers.

Q210. If the producer is faster than the consumer in a message queue, what happens first? L2

Messages accumulate in the queue, so the queue grows (and latency increases). You fix it by adding more consumers (workers), speeding up the task, or limiting the producer. The broker's memory can eventually run out if the backlog keeps growing.

Q211. Compare 1000 users getting a report in a loop inside one task versus one subtask per user. L2

One task with a loop is simple but runs sequentially in one worker and a failure may stop the rest. One subtask per user, started from a parent via `group()` or a loop of `.delay()`, runs in parallel across workers and retries failures individually, at the cost of more messages. For 1000 emails the per-user tasks are usually better; just limit the rate for the SMTP server.

Q212. How do you scale Celery? L2

Run more worker processes or machines pointed at the same broker, and raise the concurrency (`-c 4`). Use separate queues for different kinds of tasks, so slow reports do not block quick emails. Only one Beat scheduler should run, otherwise periodic tasks are sent twice.

Webhooks, SSE and email

Q213. What is a webhook? L1

A webhook is an HTTP callback: when an event happens, the source system sends an HTTP POST with the event data to a URL that you registered. It is "reverse API" or "don't call us, we'll call you", so you do not need to poll. Examples: payment gateway notifications, or posting a message to Google Chat or Slack with their incoming webhook URL.

Q214. Show a webhook sender and a receiver. L2

```
# sender: post a reminder to a chat webhook
import requests
requests.post(WEBHOOK_URL, json={"text": "Your parking ends in 10 minutes"})

# receiver: a Flask route
@app.post("/webhook/payment")
def payment_hook():
    event = request.get_json()
    process(event)
    return "", 204
```

The receiver should respond quickly (do heavy work in Celery) with a 2xx code, otherwise the sender may retry.

Q215. How do you secure a webhook receiver? L2

Use HTTPS, verify a shared secret or an HMAC signature header computed from the raw body, compare with `hmac.compare_digest`, validate the payload, ignore replays (timestamps or event ids), and make the handler idempotent because senders retry. Never trust a request just because it reaches your URL.

Q216. Webhook vs polling: when is each better? L1

Webhooks are event-driven: low latency, no wasted requests, but the receiver must be publicly reachable and handle retries. Polling is simple and works behind firewalls and when the source offers no webhooks, but it wastes requests and has delay up to the polling interval.

Q217. What are server-sent events (SSE)? L1

SSE is a one-way stream from server to browser over a normal HTTP connection with the `text/event-stream` content type. The browser uses `EventSource`, which reconnects automatically and can resume with `Last-Event-ID`. Each message is a block of text lines ending with a blank line, such as `data: hello`.

Q218. Show an SSE endpoint in Flask and the client code. L2

```
import time
from flask import Response

@app.get("/stream")
def stream():
    def gen():
        while True:
            yield f"data: {time.time()}\n\n"
            time.sleep(2)
    return Response(gen(), mimetype="text/event-stream")
```

```
const es = new EventSource("/stream");
es.onmessage = e ⇒ console.log(e.data);
```

Each message must end with a blank line. The server needs a threaded or async worker so a stream does not block other requests.

Q219. SSE vs WebSocket vs polling? L1

Polling asks repeatedly and is simple but wasteful. Long polling holds the request open until data is ready or a timeout. SSE is one-way server to client with automatic reconnect over HTTP. WebSocket is a two-way full-duplex channel over a single connection, better for chat and games.

Q220. Why can't EventSource send an Authorization header, and what do you do about it? L2

The standard `EventSource` API does not allow custom headers. So send the token as a query parameter (use a short-lived, single-purpose token, since URLs can be logged), or use cookies, or use `fetch` streaming. This detail matters if your SSE endpoint is protected.

Q221. What is the difference between short polling and long polling? L1

In short polling the client sends requests at fixed intervals and the server answers immediately, even with "no news". In long polling the server keeps the request open until it has new data or a timeout, then the client immediately asks again. Long polling reduces empty responses but holds server connections.

Q222. What is pub/sub and how is it different from point-to-point messaging? L1

In publish/subscribe a publisher sends a message to a topic and every subscriber of that topic receives a copy. In point-to-point (a queue) each message is delivered to exactly one consumer, which is how Celery workers share tasks. A message broker can support both.

Q223. What are the benefits of a message broker? L1

It decouples producer and consumer, so they need not be online together, it buffers bursts of work, allows load to be shared across multiple workers, enables retries and routing, and lets you scale parts independently. The cost is extra infrastructure and eventual (not immediate) results.

Q224. How do you send an email from Flask? L1

Use Python's `smtplib` with `email.message.EmailMessage`, or Flask-Mail, connecting to an SMTP server.

```
import smtplib
from email.message import EmailMessage

def send_mail(to, subject, html):
```

```
    msg = EmailMessage()
    msg["From"], msg["To"], msg["Subject"] = "app@example.com", to, subject
    msg.set_content("Please view this in an HTML client")
    msg.add_alternative(html, subtype="html")
    with smtplib.SMTP("localhost", 1025) as s:
        s.send_message(msg)
```

Sending is slow, so call it from a Celery task.

Q225. Why did you use MailHog (or a similar tool) in development? L1

MailHog is a fake SMTP server that accepts all mail and shows it in a web UI (port 8025, SMTP on 1025) so you can see emails without sending real ones and without credentials. In production you would swap the host and port to a real SMTP provider with TLS and authentication details taken from environment variables.

Q226. How would you generate a monthly HTML report and also attach a PDF? L2

A Celery beat task queries the month's data per user, renders an HTML report with a Jinja template, and sends it as the HTML body. For the PDF convert the same HTML using a library like WeasyPrint, pdfkit (wkhtmltopdf) or xhtml2pdf, and attach the bytes with `msg.add_attachment(pdf, maintype="application", subtype="pdf", filename="report.pdf")`.

Q227. How do you scale sending of many emails? L2

Send them from background tasks, one task per recipient (or in batches) so failures are retried individually, and rate-limit to respect the SMTP provider's limits. Reuse the SMTP connection within a batch, and log failures. Avoid sending in the web request.

Q228. Which statements are true for webhooks and SSE? L1

Both are server-initiated notifications, but differ: webhooks are server-to-server HTTP POSTs to a registered URL, while SSE is a long-lived server-to-browser stream. Webhooks need a publicly reachable receiver; SSE uses a normal browser connection and `EventSource`, and is one-directional. Neither is bidirectional like WebSockets.

Performance

Q229. How did you measure and improve the performance of your application? L1

I measured the response time with the browser DevTools Network tab, a simple timer, or a load tool before and after changes, and ran Lighthouse for the frontend. Improvements: Redis caching of expensive endpoints, background jobs for heavy work, database indexes and pagination, lazy-loaded routes, compressed and sized images. Always say what you measured and the numbers.

Q230. How would you demonstrate that caching improves performance? L1

Call the same endpoint twice and compare the times in the Network tab or with `curl -w "%{time_total}"`: the first (miss) is slow and the second (hit) is fast. Also show the key in `redis-cli`, and show that a database query log or a print statement appears only for the miss. Then change the data and show invalidation or the TTL expiry.

Q231. What is an N+1 query problem and how do you fix it? L2

It is when you run one query to get N parent rows and then one more query for each parent to get its related rows, giving N+1 queries. In SQLAlchemy fix it with eager loading such as `joinedload` or `selectinload`, or a single JOIN query. It is a common reason for slow list endpoints.

Q232. What is a database index and what is the cost? L1

An index is an additional data structure (usually a B-tree) that speeds up lookups, filters and sorts on a column, like a book index. The cost is extra storage and slower writes because the index must also be updated. Index columns used often in WHERE, JOIN or ORDER BY.

Q233. Which frontend techniques improve load time? L1

Code splitting and lazy-loaded routes, minified bundles, removing unused libraries, compressing and lazily loading images, gzip or brotli, HTTP caching for static files, a CDN, and avoiding render-blocking resources. These improve metrics measured by Lighthouse like LCP and TBT.

Q234. Which metrics are in the Lighthouse performance score? L1

First Contentful Paint (FCP), Largest Contentful Paint (LCP), Total Blocking Time (TBT), Cumulative Layout Shift (CLS) and Speed Index are the main ones. Things like Time to First Byte, bundle size and image weight influence them indirectly; accessibility or SEO checks don't change the performance score.

Q235. What is vertical vs horizontal scaling? L1

Vertical scaling gives one machine more CPU or RAM; it is simple but has a limit and a single point of failure. Horizontal scaling adds more machines behind a load balancer, which needs stateless application servers and a shared database or cache. Statelessness (token auth, Redis for shared state) makes horizontal scaling possible.

Q236. Why not run the Flask development server in production? L2

It is single-process and not designed for security, concurrency or reliability. Use a WSGI server such as Gunicorn with several workers behind Nginx, which serves static files, terminates TLS and proxies `/api` to Flask. This also lets many requests run in parallel.

Q237. How would you load-test your API? L2

Use a tool such as Apache Bench (`ab -n 1000 -c 20 http://localhost:5000/api/lots`), Locust or k6, and watch the response times, the error rate and the requests per second, with and without caching. The goal is to find the bottleneck, so you test one change at a time.

Q238. What does debounce or throttle do for performance on the frontend? L2

They limit how often an expensive handler runs. Debounce waits until the events stop (search while typing), throttle runs at most once per interval (scroll or resize handlers). Both reduce API calls and re-renders.

Q239. Why is a computed property better than a method for expensive calculations in Vue? L1

Computed values are cached based on their reactive dependencies, so they do not recompute on every render unless data changes, while a method call in the template runs on every re-render. For expensive derived data such as a filtered and sorted list, computed saves work.

Q240. Is caching always good? What are its drawbacks? L2

No. Caching adds complexity: stale data, invalidation bugs, memory use, and the risk of serving one user's data to another if keys are wrong. It helps most for data that is read often, expensive to compute, and changes rarely. Measure first, then cache the real bottleneck.

Your project (design choices)

Q241. Briefly describe the architecture of your project. L1

A Vue 2 single-page app (Vue CLI, Vue Router, Vuex) talks to a Flask REST API with token authentication and SQLAlchemy over SQLite. Redis serves as the cache and as the Celery broker; Celery workers run async jobs (CSV export), and Celery Beat triggers the daily reminders and monthly reports sent by email. Role-based access separates the admin and user. I explain this flow end to end with the main files of each part.

Q242. Why did you choose Vue CLI over the CDN? L1

My app has many components and shared state, so I needed single-file components, npm packages (axios, vue-router, vuex), hot reloading and a production build. The CDN version is simpler, but all templates would be strings or in one HTML page and it does not scale to many components. The trade-off is a build step and Node tooling.

Q243. Why did you use Vuex in your project? L1

Several unrelated components need the same data: the logged-in user, role and token, and some shared lists. Passing them through props and events would be tangled, so a store gives one source of truth. Auth state lives in Vuex (synced to localStorage for refresh), and API calls are in actions. If it were a tiny app, I would not use Vuex.

Q244. Why did you choose token-based authentication rather than sessions? L1

The frontend is a separate SPA, so a token in a header works the same for the SPA, curl or a future mobile app, and the API stays stateless. I used the token issued by Flask-Security (or JWT) and enforced it with `@auth_required("token")`. Cookies and sessions would need CORS credentials and CSRF handling.

Q245. How did you implement role-based access control? L1

Users have roles (admin, customer and so on) stored in the database with Flask-Security's user-role tables. Each API endpoint is protected with `@auth_required("token")` and `@roles_required("admin")` as needed, and customer endpoints filter by `current_user.id` so users only see their own data. The Vue router guards use the role to hide pages, but the server is the real enforcement.

Q246. Explain how you implemented the daily reminder. L1

A Celery Beat entry with `crontab(hour=18, minute=0)` triggers a task each day. The task queries users who have not visited or have pending items, builds a message and sends an email (or a Google Chat webhook message) through SMTP. It runs in a worker, uses the application context, and I tested it with a short schedule (every minute) before setting the real time.

Q247. Explain how you implemented the monthly activity report. L1

A Beat task with `crontab(day_of_month=1, ...)` runs on the first of each month. For each user it aggregates last month's data with SQL (counts, totals, most used items), renders an HTML template, and emails it (optionally with a PDF attachment). Heavy aggregation and email run in the Celery worker so the web server is not affected.

Q248. Explain your CSV export as an async job. L1

The user clicks Export and the API starts a Celery task with `.delay()` and returns a task id at once. The task queries the data, writes a CSV file and saves it (or emails a link). The frontend polls the status endpoint using the task id and shows a download button when the state is SUCCESS. This avoids request timeouts for large exports.

Q249. How did you use caching in your project, and how do you ensure it isn't stale? L1

I cached read-heavy endpoints, such as the list of lots or the dashboard statistics, in Redis with a TTL chosen from how fast the data changes (60 seconds for availability, longer for static data). Write endpoints invalidate the affected keys, and keys include the user id when the response is user-specific. If asked, I demonstrate it with the timing and `redis-cli`.

Q250. How is token expiry handled in your app? L1

The server sets a maximum token age; when the token expires the API responds with 401. The frontend's response interceptor catches the 401, clears the store and localStorage, and redirects to login with a message. A short expiry limits the damage of a stolen token; for better usability a refresh token could be added.

Q251. How did you design the database, and why SQLite? L1

Tables with relationships (users, roles, and the domain entities with foreign keys) defined using SQLAlchemy models, with unique constraints and indexes on frequently queried columns. SQLite was

required, needs no setup and is fine for a single-user demo; its limits are weak write concurrency and no network access, so for production I would use PostgreSQL.

Q252. What security measures did you take? L1

Hashed and salted passwords, token authentication with expiry, server-side role checks, input validation, ORM queries against SQL injection, Vue's output escaping against XSS, a limited CORS origin list, secrets in environment variables, and no sensitive data in the JWT payload. I would add HTTPS, rate limiting and refresh tokens in production.

Q253. What was the hardest part of the project and what did you learn? L1

Pick a real one, such as getting Celery and Redis to run together, debugging CORS and preflight errors, or keeping cache and database consistent. Describe the symptom, how you found the cause (logs, Network tab, `redis-cli`), and the fix. A concrete debugging story shows that you built it yourself.

Q254. What would you improve if you had more time? L1

Refresh tokens and token revocation, pagination everywhere, automated tests, Docker Compose for the services, PostgreSQL, rate limiting, better error handling and a CI pipeline. Naming trade-offs honestly shows understanding.

Q255. How do you run the whole project? L1

Start Redis, run the Flask server, start a Celery worker and Celery Beat, start MailHog for emails, and run the Vue dev server (`npm run serve`) or serve the build. I keep these commands in the README, and I can show each process's log when a task runs.

Q256. How would you scale your project to many users? L1

Use a stateless Flask app behind Nginx with several Gunicorn workers, move from SQLite to PostgreSQL, run multiple Celery workers, and use Redis for caching and shared state. Serve the Vue static build through a CDN. Because token auth is stateless, adding servers is straightforward.

Q257. Which parts of the code did you write yourself and how can you prove it? L1

Be ready to open any file and explain it line by line: the model, an API endpoint, a Vuex action, a Celery task. If you used a library or a template, say so and explain what it does. Examiners can ask you to change something live, so understand your code, not just run it.

Q258. Why did you choose to store the token in localStorage? L1

It is simple, survives refreshes, and works with the Authorization header. I know the risk is XSS, so I escape output, avoid `v-html` with user data, and use a short expiry. An HttpOnly cookie is an alternative that needs CSRF protection and CORS credentials.

Q259. How do you handle errors and notify the user in your app? L1

The API returns proper status codes and a JSON message, Axios or fetch errors are caught in actions or components, and the UI shows the message (a toast or an alert) and keeps the form data. Specific cases such as 401 trigger a redirect to login, and a failed Celery task is logged and, if it matters, retried.

Code-change challenges (Level 2 style)

Q260. Live change: add a "Refresh" button that reloads the list from the API. L2

Add a method and a button in the component.

```
<button @click="load">Refresh</button>
```

```
methods: {
  async load() {
    this.loading = true;
    const res = await fetch("/api/lots");
    this.lots = await res.json();
    this.loading = false;
  }
},
created() { this.load(); }
```

Reusing the same `load` method in `created` avoids duplicate code. Explain that the data assignment triggers a re-render.

Q261. Live change: show a "No items found" message when the list is empty. L2

```
<p v-if="!loading && lots.length === 0">No items found</p>
<ul v-else>
  <li v-for="l in lots" :key="l.id">{{ l.name }}</li>
</ul>
```

`v-else` must follow the `v-if` element immediately. Use `lots.length === 0` because an empty array is truthy.

Q262. Live change: add a search box that filters the displayed list in real time. L2

```
<input v-model="query" placeholder="Search">
<li v-for="l in filtered" :key="l.id">{{ l.name }}</li>
```

```
data() { return { query: "", lots: [] }; },
computed: {
  filtered() {
    const q = this.query.toLowerCase();
    return this.lots.filter(l => l.name.toLowerCase().includes(q));
  }
}
```

I use computed so it is cached and updates as `query` changes.

Q263. Live change: protect a new route `/reports` so that only admins can open it. L2

Add `meta` to the route and check it in the guard.

```
{ path: "/reports", component: Reports, meta: { requiresAuth: true, role: "admin" }
}
```

The existing `beforeEach` redirects when the role does not match. Then add `@roles_required("admin")` on the backend endpoint, because the guard alone is not security.

Q264. Live change: add a new API endpoint that returns the count of items, with admin-only access. L2

```
@app.get("/api/admin/count")
@auth_required("token")
@roles_required("admin")
def count():
    return {"count": Lot.query.count()}
```

Then test it in Postman with and without the token and as a non-admin to show 401 and 403. Mention that the decorator order matters: `@app.get` must be the outermost.

Q265. Live change: change the cache time of an endpoint from 60 seconds to 10 seconds, and show the effect. L2

Change `@cache.cached(timeout=60)` to `timeout=10`, restart Flask, and call the endpoint repeatedly. For the first 10 seconds the response is identical (hit); afterwards a recomputed value appears (miss). `redis-cli ttl <key>` shows the counting down. A shorter TTL gives fresher data but more database work.

Q266. Live change: invalidate the cache when a new item is created. L2

In the POST handler, after the commit, delete the cached key.

```
db.session.add(lot); db.session.commit()
cache.delete("view//api/lots")   # key used by the cached list view
```

In Flask-Caching the default key for `cached` is `view/<request.path>` (prefix `view/`), so inspect the key with `redis-cli keys "*"` or give the decorator an explicit `key_prefix="lots"` and delete that. Then GET again to show the new item appears.

Q267. Live change: change the daily reminder to run every 2 minutes for a demo. L2

Change the schedule in `beat_schedule`.

```
"daily-reminder": {"task": "tasks.send_reminders",
                   "schedule": crontab(minute="*/2")}
```

Restart Beat (the worker picks the task up automatically), and show the log lines appear every 2 minutes and the mail in MailHog. Using `schedule: 120.0` (seconds) also works.

Q268. Live change: add a new Celery task that sends an email to one user and call it from an endpoint. L2

```
@celery.task
def welcome_mail(to):
    send_mail(to, "Welcome", "<h1>Hello</h1>")

@app.post("/api/welcome/<int:uid>")
@auth_required("token")
def welcome(uid):
    user = User.query.get_or_404(uid)
    welcome_mail.delay(user.email)
    return {"msg": "queued"}, 202
```

Restart the worker, because it only knows tasks loaded at startup, then show the mail in MailHog.

Q269. Live change: make the Vuex store keep a cart total with a mutation and a getter. L2

```
state: { items: [] },
mutations: { ADD(state, item) { state.items.push(item); } },
getters: { total: s ⇒ s.items.reduce((t, i) ⇒ t + i.price, 0) },
```

```
this.$store.commit("ADD", { name: "Pen", price: 10 });
// template: {{ $store.getters.total }}
```

The getter recalculates automatically when `items` changes.

Q270. Live change: the page shows `undefined` for a nested value on first render. Why and how do you fix it? L2

The data arrives asynchronously, so on the first render `user` is still `null` or `{}` and `user.address.city` throws or shows nothing. Initialise data with the right shape, or guard the template with `v-if="user"`, or use optional access in a computed property (`this.user && this.user.address`). In Vue 2 templates the `?.` operator is not supported.

Q271. Live change: a button should be disabled while a request is running. L2

```
<button :disabled="busy" @click="save">Save</button>
```

```
async save() {
  this.busy = true;
  try { await axios.post("/api/lots", this.form); }
  finally { this.busy = false; }
}
```

Using `finally` makes sure the button is enabled again even when the call fails. It also prevents double submission.

Q272. Live change: add a loading spinner and an error message to a page that loads data. L2

Add `loading` and `error` to `data`, set them in the load method with try/catch/finally, and write `<p v-if="loading">Loading …</p><p v-else-if="error" class="err">{{ error }}</p>` before the list. Use `v-else-if` / `v-else` so only one block displays at a time.

Q273. Live change: the Vue app calls the API and the browser reports a CORS error. Fix it on the backend. L2

Install Flask-CORS and enable it for the frontend origin and allow the needed headers.

```
from flask_cors import CORS
CORS(app, resources={r"/api/*": {"origins": "http://localhost:8080"}},
     allow_headers=["Content-Type", "Authentication-Token"])
```

Restart Flask and check in the Network tab that the response has `Access-Control-Allow-Origin`. If the origin or port is different, the header will not match.

Q274. Live change: show different navbar links for admin and normal users. L2

Read the role from the store and use `v-if`.

```
<router-link v-if="role === 'admin'" to="/admin">Admin</router-link>
<router-link v-if="token" to="/profile">Profile</router-link>
```

```
computed: { ...mapState(["role", "token"]) }
```

Mention that this only hides links; the API must still check the role.

Q275. Live change: add pagination (page number and next/previous) to an API endpoint. L2

```
@app.get("/api/lots")
def lots():
    page = request.args.get("page", 1, type=int)
    p = Lot.query.paginate(page=page, per_page=10, error_out=False)
    return {"items": [l.to_dict() for l in p.items], "page": p.page, "pages":
p.pages}
```

In Vue keep a `page` in data, call `/api/lots?page=${this.page}` and disable Previous on page 1 and Next on the last page.

Q276. Live change: add a field (for example "phone") to the user model and the API. What else must change? L2

Add the column to the SQLAlchemy model, then update the database: SQLite will not change an existing table with `create_all`, so use a migration (Flask-Migrate) or recreate the DB in development. Then update the API's input handling and the output fields, the Vue form (`v-model`), and any validation. Forgetting the database migration is the common mistake.

Q277. Live change: the child must tell the parent when an item is deleted. Show the code. L2

```
// child
methods: { remove() { this.$emit("deleted", this.item.id); } }
```

```
<!-- parent -->
<item-row v-for="i in items" :key="i.id" :item="i" @deleted="onDeleted"></item-row>
```

```
onDeleted(id) { this.items = this.items.filter(i => i.id !== id); }
```

The parent owns the list, so it does the removal, and the child just reports the event.

Q278. Live change: persist the user's selected theme across refreshes. L2

Save it to localStorage when it changes and read it on load.

```
data() { return { theme: localStorage.getItem("theme") || "light" }; },
watch: { theme(v) { localStorage.setItem("theme", v); } }
```

Mention that localStorage holds strings and is per-origin, and that this is non-sensitive data.

Q279. Live change: add a watcher that calls the API when the search text changes, with debounce. L2

```
watch: {
  query() {
    clearTimeout(this.t);
    this.t = setTimeout(this.search, 300);
  }
},
methods: { async search() { /* call /api/search?q=this.query */ } }
```

Without the debounce each keystroke would send a request. Clear the timer in `beforeDestroy`.

Q280. Live change: make the token expire after 30 minutes. L2

For Flask-Security set `SECURITY_TOKEN_MAX_AGE = 1800` (seconds) in the config; for Flask-JWT-Extended set `JWT_ACCESS_TOKEN_EXPIRES = timedelta(minutes=30)`. Restart the server. After 30 minutes the API returns 401, and the frontend interceptor logs the user out. To demo, use a small number like 30 seconds.

Q281. Live change: make the Celery export task report progress. L2

Use `self.update_state` with a custom state and meta in a bound task, and read it in the status endpoint.

```
@celery.task(bind=True)
def export(self):
    for i, row in enumerate(rows):
```

```
        write(row)
        self.update_state(state="PROGRESS", meta={"done": i, "total": len(rows)})
```

The status endpoint returns `r.state` and `r.info`, and the frontend shows a progress bar.

Q282. Live change: add a Redis-backed counter for failed logins to block brute force. `L2`

```
key = f"fail:{email}"
n = r.incr(key)
r.expire(key, 600)
if n > 5:
    return {"msg": "Too many attempts"}, 429
```

`INCR` is atomic, and the expiry resets the window after 10 minutes. Delete the key after a successful login.

MAD1 cheat sheet

Week 1: How the web works: clients, servers, URLs and HTTP

- Requests per second = 1 / (seconds per request); with `n` workers multiply by `n`
- Bits = bytes x 8; Gbps = 1000 Mbps; throughput = (requests per second) x (response size)
- Download time = size in megabits / speed in Mbps
- Defaults: http port 80, https 443, Flask 5000, `python -m http.server` 8000
- Flask: `request.args.get("k")` (query), `<int:id>` (path), `abort(404)`, `jsonify(x)` gives 200 and `application/json`
- Status: 200 OK, 201 Created, 301/302 redirect, 400 Bad Request, 401 not logged in, 403 forbidden, 404 not found, 405 wrong method, 500 server error
- curl: `curl -i URL`, `curl -X POST URL -H "Content-Type: application/json" -d '{"a": 1}'`

Week 2: HTML and CSS: building and styling pages

- Skeleton: `<!DOCTYPE html><html><head><title></title></head><body></body></html>`
- Links: `<link rel="stylesheet" href="s.css">`, `<script src="a.js"></script>`, `<a href="">`, `<img src="" alt="">`
- Specificity: inline 1000, `#id` 100, `.class [attr] :pseudo-class` 10, `tag ::pseudo-element` 1
- Visible width (content-box) = width + 2 padding + 2 border; space = + 2 margin
- Flex: `display: flex; justify-content: ...; align-items: ...; gap: ...`
- Bootstrap 5 CSS: `https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css`
- Breakpoints: sm 576, md 768, lg 992, xl 1200; grid = 12 columns

Week 3: Views: Jinja2 templates, Bootstrap and forms

- Render: `render_template("page.html", name=value)`
- Static link: `href="{{ url_for('static', filename='style.css') }}"`
- Loop: `{% for x in xs %}` … `{{ loop.index }}` … `{% else %}empty{% endfor %}`
- If: `{% if a %}`..`{% elif b %}`..`{% else %}`..`{% endif %}`
- Inheritance: `{% extends "base.html" %}` and `{% block content %}` … `{{ super() }}{% endblock %}`
- Macro: `{% macro name(arg, opt='x') %}` … `{% endmacro %}` then `{{ name(1) }}`
- Whitespace trim: `{%- … -%}`; comment: `{# … #}`
- Form data: `request.form.get("k")`, `request.args.get("k")`, `request.method`
- Count: `{{ items|length }}`; default: `{{ x|default('n/a') }}`
- `string.Template`: `$name`, `substitute()` (KeyError) / `safe_substitute()`

Week 4: Models: databases, SQL, ER diagrams and SQLAlchemy

- `CREATE TABLE t(id INTEGER PRIMARY KEY, name TEXT NOT NULL UNIQUE, n INTEGER DEFAULT 0 CHECK(n≥0));`
- `SELECT cols FROM t JOIN u ON t.id=u.t_id WHERE … GROUP BY … HAVING … ORDER BY … LIMIT n;`
- Aggregates: `COUNT SUM AVG MIN MAX`; NULL test: `IS NULL`
- SQLite file URI: `sqlite:///name.sqlite3`; memory: `sqlite:///:memory:`
- `db = SQLAlchemy(app)`; models extend `db.Model`; `db.Column(db.String(80), nullable=False, unique=True)`
- `db.relationship("Child", backref="parent")` plus `db.ForeignKey("parent.id")` on the child
- `db.session.add(obj)`, `db.session.commit()`, `db.session.rollback()`, `db.session.delete(obj)`
- `Model.query.all()` / `.first()` / `.filter_by(a=1)` / `.filter(Model.a > 1)` / `.order_by(…)` / `.count()`
- `db.session.query(func.max(M.col)).scalar()`
- `with app.app_context(): db.create_all()`

Week 5: Controllers: Flask routes and business logic

- `@app.route("/x/<int:id>", methods=["GET", "POST"])`
- `request.args.get("q")`, `request.form.get("f")`, `request.json`, `request.method`
- `return "text", 201`; `return {"a": 1}`; `return jsonify(list)`
- `redirect(url_for("endpoint", arg=1))`; `abort(404)`
- `@app.errorhandler(404)`; `flash("msg")`; `get_flashed_messages()`
- `app.secret_key = "…"`; `session["k"] = v`; `session.get("k")`; `session.pop("k", None)`
- `Blueprint("name", __name__, url_prefix="/p")`; `app.register_blueprint(bp)`
- `db.init_app(app)`; `def create_app(): …; return app`
- `python -m venv venv`; `pip install -r requirements.txt`; `pip freeze > requirements.txt`
- Status codes: 200 OK, 302 redirect, 400 bad request, 401 unauthorised, 403 forbidden, 404 not found, 405 wrong method, 415 wrong media type, 500 server error

Week 6: APIs and REST: OpenAPI and Flask-RESTful

- `api = Api(app)`; `api.add_resource(Cls, "/x", "/x/<int:id>")`
- `class Cls(Resource): def get(self, id): return {...}, 200`
- `parser = reqparse.RequestParser(); parser.add_argument("n", type=int, required=True, location="json")`; `args = parser.parse_args()`
- `@marshal_with({"id": fields.Integer, "name": fields.String})`
- `abort(404, message="...")` from `flask_restful`
- `curl -X POST URL -H "Content-Type: application/json" -d "{\"a\":1}"`; `curl -v URL`
- `requests.post(URL, json={...})`, `.status_code`, `.json()`
- 201 created, 204 no content, 400 bad input, 401 not logged in, 403 not allowed, 404 missing, 405 wrong verb, 409 conflict
- OpenAPI: `openapi: 3.0.3`, `info`, `paths: /x: get: parameters/responses`, `components: schemas`

Week 7: Backend systems: indexing, search, caching and logging

- Levels: DEBUG 10, INFO 20, WARNING 30 (default), ERROR 40, CRITICAL 50
- `logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')`
- `breakpoint()`; pdb: `n`, `s`, `c`, `p x`, `l`, `q`; `flask --app app shell`
- `CREATE INDEX idx ON t(col);` `EXPLAIN QUERY PLAN SELECT ...;` SCAN vs SEARCH USING INDEX
- `CREATE VIRTUAL TABLE t USING fts5(a, b);` `WHERE t MATCH 'word OR pre*'`
- `@cache.cached(timeout=60)`, `@cache.memoize()`, `cache.delete(key)`, `cache.clear()`
- `Cache-Control: public, max-age=60`, `ETag` and `If-None-Match` give 304
- DDR bandwidth = (bus bytes) x (2 x clock) x channels; worst disk latency = 60000 / RPM ms
- Round trip = 2 x distance / speed; requests per second = 1000 / RTT ms
- Data = rate (Mbit/s) x seconds / 8 (MB); 1.5 million x 4 KB = 6 GB

Week 8: The frontend: JavaScript, the DOM and fetch

- `<script src="app.js" defer></script>`
- `document.getElementById("id")`, `document.querySelector("#id .cls")`, `querySelectorAll("li")`
- `el.textContent`, `el.innerHTML`, `el.style.fontWeight = "bold"`, `el.classList.add("x")`
- `const li = document.createElement("li"); ul.appendChild(li)`
- `el.addEventListener("click", e => { e.preventDefault(); })`
- `JSON.parse(text)`, `JSON.stringify(obj)`
- `const res = await fetch(url, {method:"POST", headers:{"Content-Type":"application/json"}, body: JSON.stringify(data)})`, `res.ok`, `res.status`, `await res.json()`
- `localStorage.setItem(k, v)`, `getItem(k)`, `removeItem(k)`
- `display`: block, inline, inline-block (side by side), flex (row)

Week 9: Security: authentication, sessions, tokens and attacks

- `generate_password_hash(pw)` and `check_password_hash(stored, pw)` from `werkzeug.security`
- `login_user(user)`, `logout_user()`, `@login_required`, `current_user`, `@login_manager.user_loader`
- `@roles_required('admin')` (all), `@roles_accepted('a','b')` (any)
- `session['k'] = v`, `session.pop('k', None)`, `app.config['SECRET_KEY'] = os.environ['SECRET_KEY']`
- `SESSION_COOKIE_HTTPONLY = True`, `SESSION_COOKIE_SECURE = True`, `SESSION_COOKIE_SAMESITE = "Lax"`
- Cookie header: `Set-Cookie: session=…; HttpOnly; Secure; SameSite=Lax`
- Parameterised SQL: `cur.execute("select * from u where name=?", (n,))`
- Jinja: `{{ x }}` escapes; `{{ x|safe }}` does not
- Status codes: 401 unauthenticated, 403 forbidden, 404 not found, 429 too many requests, 302 redirect
- Basic auth = base64 (not encryption); Digest = hash; HTTPS = encrypted channel

Week 10: Testing web applications with pytest

- `pip install pytest pytest-cov`; run: `pytest`, `pytest -v`, `pytest -x`, `pytest file.py::test_name`
- Selection: `-k name_part`, `-m marker`, `-m "not slow"`, `-m "a and b"`
- Fixture: `@pytest.fixture` (use `yield` for teardown); parametrize: `@pytest.mark.parametrize("a,b", [(1,2),(3,4)])`
- Checks: `assert x == y`, `pytest.approx(0.3)`, `with pytest.raises(Err):`, `tmp_path`
- Flask: `app.test_client()`; `client.get("/")`; `client.post("/x", data={…})`; `client.post("/api", json={…})`
- Response: `.status_code`, `.data` (bytes), `.get_json()`, `.headers["Location"]`
- Test database: `{"TESTING": True, "SQLALCHEMY_DATABASE_URI": "sqlite:///:memory:"}`
- Mock: `from unittest.mock import patch, MagicMock`; `@patch("module.name")`
- Coverage: `pytest --cov=app --cov-branch --cov-report=term-missing`
- Summary words: passed, failed, deselected, error, warnings

Week 11: HTML5, accessibility, encoding and beyond HTML

- Bits per character (fixed, minimum) = ceiling of log2 of the number of distinct characters; total = characters x bits
- UCS-4 bits / 32 x 7 = ASCII 7-bit bits; UCS-2 bits / 16 x 7 = ASCII bits
- UTF-8: U+0000-007F 1 byte; to U+07FF 2; to U+FFFF 3; to U+10FFFF 4. UTF-32 always 4 bytes
- `<meta charset="utf-8">` and `<meta name="viewport" content="width=device-width, initial-scale=1">`
- `localStorage.setItem(k, v)`, `getItem(k)`, `removeItem(k)`; use `JSON.stringify` / `JSON.parse`
- POUR = Perceivable, Operable, Understandable, Robust; ARIA: `role`, `aria-label`, `aria-live`
- `ws://` / `wss://` WebSocket; `new EventSource(url)` for SSE; `navigator.serviceWorker.register("/sw.js")`
- Custom element: `customElements.define("my-tag", Class)` (name has a hyphen)

Week 12: Deployment: Git, WSGI, servers and the cloud

- Git setup: `git config --global user.name "N"`, `git config --global user.email "e"`
- Git basics: `git init`, `git status`, `git add .`, `git commit -m "msg"`, `git log --oneline`, `git diff`
- Branches: `git branch`, `git branch x`, `git checkout x` or `git switch x`, `git checkout -b x`, `git merge x`, `git branch -d x`
- Remote: `git clone URL`, `git remote add origin URL`, `git push -u origin main`, `git pull`
- Untrack but keep: `git rm --cached file`; ignore: `.gitignore`
- Env: `python -m venv venv`, `pip freeze > requirements.txt`, `pip install -r requirements.txt`
- Config: `os.environ.get("NAME", "default")`
- Serve: `gunicorn -w 4 -b 127.0.0.1:8000 app:app`; nginx `proxy_pass http://127.0.0.1:8000;`
- Docker: `docker build -t myapp .`, `docker run -p 8000:8000 myapp`, `docker compose up`
- Migrate: `flask db init`, `flask db migrate -m "msg"`, `flask db upgrade`, `flask db downgrade`
- Logging levels: DEBUG, INFO, WARNING, ERROR, CRITICAL

MAD2 cheat sheet

Week 1: JavaScript basics: values, functions and the DOM

- `typeof null` -> "object"; `typeof NaN` -> "number"; `Array.isArray(x)`
- Falsy: `false 0 -0 0n "" null undefined NaN`
- `"5" + 3` -> "53"; `"5" - 3` -> 2; `1 + 2 + "3"` -> "33"
- `for (const v of arr)` values; `for (const k in obj)` keys (strings)
- `arr.reduce((acc, x) => acc + x, 0)`; `arr.map(f)`; `arr.filter(f)`
- `const { a, b: { c } } = obj;` `const [x, y] = arr;` `{ ...obj, k: v }` `[...arr, v]`
- `JSON.stringify(obj)` / `JSON.parse(text)`
- `document.querySelector("#id")`, `el.addEventListener("click", fn)`, `el.textContent`

Week 2: Advanced JavaScript: closures, promises and async/await

- `const make = () => { let c = 0; return () => ++c; };`
- `f.call(obj, a)`, `f.apply(obj, [a])`, `const g = f.bind(obj)`
- `class B extends A { constructor(x) { super(x); } }`
- `export const x = 1; export default fn; import fn, { x } from "./m.js";`
- `try { } catch (e) { } finally { }`; `a?.b`, `a ?? b`
- Order: sync, microtasks (`then`, after `await`), tasks (`setTimeout`)
- `Promise.all([..])`, `Promise.race([..])`, `Promise.allSettled([..])`
- `const res = await fetch(url, { method: "POST", headers: { "Content-Type": "application/json" }, body: JSON.stringify(obj) }); if (!res.ok) throw new Error(res.status); const data = await res.json();`

Week 3: The web frontend: SPAs, browser storage and tooling

- `localStorage.setItem(k, v)`, `getItem(k)` (null if missing), `removeItem(k)`, `clear()`; `sessionStorage` has the same API
- `JSON.stringify(obj)` before storing, `JSON.parse(str)` after reading
- Sizes: cookie about 4 KB; web storage about 5 MB per origin
- `npm install`, `npm run serve` (dev, hot reload), `npm run build` (to `dist/`)
- `^2.6.14` = same major, newer minor and patch allowed
- `<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>`; `new Vue({ el: "#app", data: {} })`
- `navigator.serviceWorker.register("/sw.js")`; manifest = name, icons, start_url, display
- `<meta name="viewport" content="width=device-width, initial-scale=1">`

Week 4: Vue basics: instances, directives and reactivity

- CDN: `<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>`
- Create: `new Vue({ el: "#app", data: { ... }, methods: { ... }, computed: { ... }, watch: { ... } })`
- `:attr="x"` is `v-bind:attr`; `@click="f"` is `v-on:click`; `@submit.prevent`, `@click.stop`, `@click.once`, `@keyup.enter`
- `v-model.trim`, `v-model.number`, `v-model.lazy`
- `v-for="(item, i) in list"`, `(value, key, i) in obj`, `n in 5`; always `:key`
- `:class="{ active: isOn }"`, `:class="[a, { b: flag }]"`, `:style="{ color: c }"`
- Caveat fixes: `Vue.set(obj, key, val)`, `this.$set(arr, i, val)`, `arr.splice(i, 1, val)`
- `this.$nextTick(fn)`, `this.$refs.name`
- Hook order: beforeCreate, created, beforeMount, mounted, (beforeUpdate, updated)*, beforeDestroy, destroyed
- Vue 3: `createApp`, Proxy reactivity, `beforeUnmount` and `unmounted`

Week 5: Vue with APIs: fetch, axios and lifecycle

- `const res = await fetch(url); if (!res.ok) throw new Error(res.status); const data = await res.json();`
- `fetch(url, { method: "POST", headers: { "Content-Type": "application/json" }, body: JSON.stringify(obj) })`
- axios: `const { data } = await axios.get(url)`, `axios.post(url, obj)`, errors in `err.response.status`
- Flask: `CORS(app)` or `CORS(app, resources={r"/api/*": {"origins": "http://localhost:8080"}})`
- Return codes: `return jsonify(obj), 201`, `return "", 204`, `return jsonify({"error": "..."}), 404`
- Dev proxy (`vue.config.js`): `devServer: { proxy: { "/api": { target: "http://localhost:5000", changeOrigin: true } } }`
- Serve build: `Flask(__name__, static_folder="dist", static_url_path="")` and `send_from_directory(app.static_folder, "index.html")`
- Origin = scheme + host + port; path does not matter; subdomain differs
- Loading pattern: `loading = true; try { ... } catch (e) { error = e.message } finally { loading = false }`

Week 6: Advanced Vue: components, props, events and Vue Router

- Global: `Vue.component("my-comp", { props: [ … ], data() { return { … }; }, template: " … " })`
- Local: `components: { MyComp }` (use `<my-comp>`)
- Prop: `props: { n: { type: Number, required: true, default: 0 } }`; pass `:n="5"`
- Event: child `this.$emit("name", payload)`; parent `@name="handler"`
- v-model on component: prop `value`, event `input`
- Slots: `<slot name="x">fallback</slot>`; `<template v-slot:x>` (or `#x`); scoped `<slot :item="i">` with `v-slot:default="sp"`
- Dynamic: `<component :is="name">`, `<keep-alive>`
- Router: `new VueRouter({ mode: "history", routes: [ … ] })`, `{ path: "/u/:id", name: "user", component: U, props: true, children: [ … ] }`
- Read: `this.$route.params.id`, `this.$route.query.q`; go: `this.$router.push({ name: "user", params: { id: 1 } })`, `.replace()`, `.go(-1)`
- Guard: `router.beforeEach((to, from, next) ⇒ { … next() })`; catch-all `{ path: "*", component: NotFound }`
- Lazy: `component: () ⇒ import("./views/About.vue")`
- Vue CLI: `vue create app`, `npm run serve`, `npm run build`; Vue 3 router: `createRouter`, `createWebHistory`

Week 7: State management with Vuex

- `new Vuex.Store({ state, getters, mutations, actions, modules, strict })`
- mutation `(state, payload)`; getter `(state, getters)`; action `(context, payload)` with `{ commit, dispatch, state, getters }`
- component: `this.$store.state.x`, `this.$store.getters.x`, `this.$store.commit('m', p)`, `this.$store.dispatch('a', p)`
- helpers: `…mapState(['x'])`, `…mapGetters(['g'])` in `computed`; `…mapMutations(['m'])`, `…mapActions(['a'])` in `methods`
- namespaced: `commit('cart/add', x)`, `getters['cart/count']`, `mapState('cart', ['items'])`
- persistence: `sessionStorage.setItem(k, v)`, `getItem(k)`, `removeItem(k)`; same API on `localStorage`; strings only
- Reactivity fix: `Vue.set(obj, 'key', val)`, `arr.splice(i, 1, val)`

Week 8: Token authentication, API design, CORS and cookies

- Send token: `Authorization: Bearer <token>` or `Authentication-Token: <token>`
- 200 OK, 201 Created, 204 No Content, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 409 Conflict, 422 Unprocessable, 429 Too Many Requests, 500 Server Error
- PyJWT: `jwt.encode(payload, key, algorithm="HS256")`, `jwt.decode(token, key, algorithms=["HS256"])`
- Flask-JWT-Extended: `create_access_token(identity=str(id))`, `@jwt_required()`, `get_jwt_identity()`, `get_jwt()`
- Flask-CORS: `CORS(app, origins=["http://localhost:8080"])`

- Cookie: `resp.set_cookie(name, val, httponly=True, secure=True, samesite="Strict", max_age=3600)`
- fetch with cookies cross-origin: `credentials: 'include'`, server needs `Access-Control-Allow-Credentials: true`
- Idempotent: GET, PUT, DELETE. Not idempotent: POST.

Week 9: Asynchronous jobs: Redis and Celery

- `celery = Celery("tasks", broker="redis://localhost:6379/0", backend="redis://localhost:6379/1")`
- `@celery.task` / `@celery.task(bind=True, max_retries=3)`; `raise self.retry(exc=exc, countdown=30)`
- `t.delay(a, b)` is `t.apply_async(args=(a, b))`; options `countdown=`, `eta=`, `expires=`
- `res = AsyncResult(task_id, app=celery)`; `res.state` (`PENDING`, `STARTED`, `RETRY`, `SUCCESS`, `FAILURE`); `res.get(timeout=5)`
- `celery -A app.celery worker -l info`, `celery -A app.celery beat -l info`, `celery -A app.celery flower`
- `crontab(hour=18, minute=0)`; `crontab(minute='*/10')`; `crontab(hour=8, minute=0, day_of_month=1)`; `crontab(hour=7, minute=30, day_of_week='mon')`
- Redis: `SET`, `GET`, `EXPIRE k 60`, `TTL k`, `INCR`, `LPUSH`, `RPUSH`, `LRANGE q 0 -1`, `SADD`, `SMEMBERS`, `HSET`, `HGETALL`, `ZADD`, `ZRANGE`; `redis-cli ping` gives `PONG`
- Windows: `docker run -p 6379:6379 redis` or Redis inside WSL

Week 10: Messaging: webhooks, server-sent events and email

- Webhook send: `requests.post(URL, json={"text": "hi"}, timeout=5)`
- Signature: `hmac.new(SECRET, body, hashlib.sha256).hexdigest()` and `hmac.compare_digest(a, b)`
- SSE response: `Response(gen(), mimetype="text/event-stream")`; frame = `"data: text\n\n"`
- Browser: `new EventSource(url)`, `es.onmessage = e ⇒ …`, `es.close()`
- Redis pub/sub: `r.publish("ch", msg)`, `p = r.pubsub(); p.subscribe("ch")`
- Flask-Mail: `Message(subject, recipients=[…], body=…, html=…)`, `mail.send(msg)`
- MailHog: SMTP localhost:1025, inbox `http://localhost:8025`
- Beat: `crontab(hour=18, minute=0)`; monthly `crontab(day_of_month=1, hour=8, minute=0)`
- CSV download header: `Content-Disposition: attachment; filename=export.csv`
- Next: week 11, performance; week 12, the project; viva: MAD2 viva

Week 11: Performance: measuring and caching

- `ab -n 100 -c 10 URL` (requests, concurrency)
- Hit ratio = hits / (hits + misses); throughput = requests / seconds
- `Cache-Control: max-age=60`, `no-store`, `no-cache`, `public`, `private`; `ETag` plus `If-None-Match` gives `304`
- Config: `CACHE_TYPE="RedisCache"`, `CACHE_REDIS_URL="redis://localhost:6379/1"`, `CACHE_DEFAULT_TIMEOUT=300`
- `@cache.cached(timeout=60)`; `@cache.cached(timeout=60, key_prefix="k")`; `@cache.memoize(timeout=60)`

- `cache.delete("key")`, `cache.delete_memoized(fn, *args)`, `cache.clear()`
- N+1 fix: `select(Author).options(selectinload(Author.books))`
- Index: `db.Column(db.Integer, index=True)`
- Pagination: `db.paginate(select(Book).order_by(Book.id), page=2, per_page=20)`
- Lazy route: `component: () ⇒ import("./views/X.vue")`
- Next: week 12, the project; earlier: week 10

Week 12: The project: putting it all together

- Header: `Authorization: Bearer <token>`; 401 = no or bad token, 403 = not allowed, 409 = conflict, 202 = accepted for background work
- Flask: `create_access_token(identity= … , additional_claims={"role": … })`, `@jwt_required()`, `get_jwt()`
- Vue router guard: `router.beforeEach((to, from, next) ⇒ { … next("/login") })`
- Vuex: `state`, `mutations` (sync), `actions` (async), `commit`, `dispatch`
- CORS: `CORS(app, resources={r"/api/*": {"origins": ["http://localhost:8080"]}})`
- Beat: `crontab(hour=18, minute=0)`, `crontab(day_of_month=1, hour=8, minute=0)`
- Celery on Windows: `celery -A app.celery worker -l info -P solo`; beat: `celery -A app.celery beat -l info`
- MailHog: SMTP 1025, inbox `http://localhost:8025`
- Run order: Redis, MailHog, Flask, worker, beat, Vue
- `pip freeze > requirements.txt`; keep secrets in `.env`, never in Git
- Related: week 10, week 11, project viva, MAD2 viva

Glossary (386 terms)

.gitignore: A file listing files Git should not track, such as virtual environments and secrets.

200 OK: The request succeeded and the response carries the result.

201 Created: The request succeeded and a new resource was created.

204 No Content: The request succeeded but there is nothing to send back.

301 Moved Permanently: The resource now lives at a new URL for good, and clients should use the new one from now on.

302 Found: A temporary redirect to another URL; Flask `redirect()` sends this by default.

304 Not Modified: The cached copy is still valid, so the server sends no body.

400 Bad Request: The server cannot understand the request because it is malformed or has invalid data.

401 Unauthorized: The client is not authenticated, meaning it has not proved who it is.

403 Forbidden: The server knows who you are but you are not allowed to do this.

404 Not Found: The server has no resource at that URL.

405 Method Not Allowed: The URL exists but does not accept the method used, such as POST on a GET-only route.

409 Conflict: The request clashes with the current state of the resource, such as creating a duplicate.

429 Too Many Requests: The client has sent too many requests in a given time and is being rate limited.

500 Internal Server Error: The server hit an unexpected problem such as an unhandled exception in the code.

502 Bad Gateway: A proxy such as nginx got an invalid answer from the application server behind it.

503 Service Unavailable: The server cannot handle the request right now, for example because it is overloaded or down.

== and ===: `==` compares after converting types, while `===` compares value and type with no conversion, so `===` is safer.

Access token: A short-lived token used to call the API.

Access-Control-Allow-Origin: A response header that names the origins permitted to read the response.

Accessibility (a11y): Designing pages so people with disabilities, such as blind users with screen readers, can use them.

ACID: Four promises of reliable transactions: Atomicity, Consistency, Isolation and Durability.

Action: A Vuex function that may do asynchronous work such as API calls and then commits mutations.

addEventListener: A method that attaches a function to run when a given event happens on an element.

Aggregate function: A function like `COUNT`, `SUM`, `AVG`, `MIN` or `MAX` that turns many rows into one value.

AJAX: Updating part of a page with data fetched in the background, without reloading the whole page.

alt text: The `alt` attribute on an image that describes it in words for screen readers or when the image fails to load.

API: Application Programming Interface, a defined way for one program to ask another to do something or give data.

API design: Planning an API's URLs, methods, inputs, outputs and errors so it is clear, consistent and easy to use.

API versioning: Keeping old and new forms of an API available, for example `/api/v1/`, so existing clients keep working when it changes.

Application context: Flask's way of making app-level objects, like the current app and database, available while handling a request or command.

Application factory: A function that builds and returns a configured Flask app, making testing and setup easier.

ARIA: A set of attributes like `aria-label` and `role` that give assistive technology extra meaning about elements.

Arrow function: A short function written `(x) ⇒ x * 2` that does not have its own `this`.

assert: A statement that fails the test if its condition is false.

Association table: A table that holds two foreign keys to connect rows in a many-to-many relationship.

async function: A function marked `async` that always returns a promise and may use `await` inside.

Asynchronous: Code that starts a task and carries on, handling the result later when it finishes.

Attribute: Extra information inside a start tag, written as `name="value"`, like `href` or `class`.

Authentication: Checking who a user is, for example with a password.

Authorization: Deciding what an already identified user is allowed to do.

Auto-escaping: Jinja2 turns characters like `<` into safe text by default so user input cannot run as HTML or script.

await: A keyword that pauses an async function until a promise settles and then gives its value.

Axios: A JavaScript library for making HTTP requests that returns promises, often used with Vue.

B-tree: A balanced tree structure that databases commonly use for indexes, allowing fast search and ordered reading.

Background job: A task that runs outside the web request so the user does not wait for it.

Bearer token: A token sent in the header `Authorization: Bearer <token>`, where whoever holds it is trusted.

Block element: An element like `<div>` or `<p>` that starts on a new line and takes the full width.

Blueprint: A Flask feature that groups related routes and templates into a reusable part of an app.

Bootstrap: A popular CSS and JavaScript library with ready-made classes and components for building responsive pages quickly.

Bootstrap grid: A 12-column layout system using `container`, `row` and `col-*` classes.

Bottleneck: The slowest part of a system, which limits the speed of the whole.

Box model: The idea that each element is a box made of content, padding, border and margin.

Branch: A separate line of development in Git, letting you try changes without touching the main code.

Broker: In Celery, the service that carries task messages from the app to workers.

Browser: A program that sends HTTP requests and displays HTML, CSS and JavaScript results as pages.

Browser cache: Files the browser keeps locally so it need not download them again.

Cache hit: A request found in the cache, which is fast.

Cache invalidation: Removing or updating cached data when the original data changes so users do not see stale values.

Cache miss: A request not found in the cache, so the data must be fetched or computed.

Cache-Control: An HTTP header that tells browsers and proxies how long and whether to cache a response.

Caching: Keeping a copy of a result so it can be reused instead of being worked out again.

Call stack: The list of functions currently running; the one on top runs first.

Callback: A function passed to another function to be called later.

Callback URL: The address you give a service so it can send you a webhook request.

Candidate key: Any column or set of columns that could uniquely identify a row; one of them is chosen as the primary key.

Cardinality: How many instances of one entity can relate to instances of another: one-to-one, one-to-many or many-to-many.

Cascade: The rules a browser uses to pick a winning style from many sources, using origin, specificity and order.

CDN: A Content Delivery Network, a group of servers around the world that deliver files such as Bootstrap or Vue quickly from a nearby location.

Celery: A Python library that runs background tasks on separate worker processes.

Celery beat: A scheduler that sends tasks to the queue at set times, such as every night.

Celery task: A Python function marked with `@celery.task` that can be run in the background by a worker.

Character encoding: The rule that turns characters into bytes and back, which is why a wrong one produces garbled text.

Class (JavaScript): A cleaner syntax for making objects with a constructor and methods, built on prototypes.

Class selector: A selector written with a dot, like `.card`, that matches every element with that class.

Client: The program (usually a browser) that sends a request and shows the response to the user.

Client-server model: A design where clients ask for things and servers answer, so work is split between the two.

Client-side rendering: Building the HTML in the browser with JavaScript from data, instead of on the server.

Closure: A function that remembers the variables from the place it was created, even after that place has finished running.

Commit: Making the changes of a transaction permanent in the database.

Commit (Git): A saved snapshot of your tracked files with a message describing the change.

commit and dispatch: In Vuex, `commit` triggers a mutation and `dispatch` triggers an action.

Component: A reusable piece of interface with its own template, data and logic.

Computed property: A value worked out from other data that is cached and recomputed only when what it uses changes.

Constraint: A rule the database enforces on data, such as `NOT NULL`, `UNIQUE`, primary key or foreign key.

Container: A light, isolated running package of an app and its dependencies that shares the host's operating system.

Content-Type: A header that names the format of the body, such as `text/html` or `application/json`.

Cookie: A small piece of data a server asks the browser to store and send back with later requests.

Cookie vs token: A cookie is stored and sent by the browser by itself, while a token is usually stored by your code and added to the header on purpose.

CORS: Cross-Origin Resource Sharing, a way for a server to tell the browser that certain other origins may read its responses.

CORS preflight: An automatic `OPTIONS` request the browser sends before certain cross-origin requests to ask if the real one is allowed.

created: A lifecycle hook that runs after data is set up but before the component is shown on the page.

Crontab: A schedule format using minute, hour, day and month fields, also available in Celery beat.

CRUD: Create, Read, Update, Delete, the four basic actions on data, mapping to POST, GET, PUT or PATCH, and DELETE.

CSRF: Cross-Site Request Forgery, an attack that tricks a logged-in user's browser into sending an unwanted request to a site.

CSRF token: A secret random value placed in a form and checked by the server to prove the request came from the real page.

CSS: Cascading Style Sheets, the language that controls how HTML looks, such as colours, fonts and layout.

CSS Grid: A CSS layout system that places items in rows and columns of a two-dimensional grid.

CSV: Comma-Separated Values, a plain text format where each line is a row and commas separate the columns.

Custom event: An event a child sends up to its parent using `this` plus the `emit` method.

Data property: A value in the `data` option of Vue that is made reactive.

Data type: The kind of value a column holds, such as `INTEGER`, `TEXT` or `REAL`.

Data types (JavaScript): The basic kinds of value: number, string, boolean, `null`, `undefined`, symbol, bigint and object.

Database: An organised store of data that can be searched and changed efficiently.

db.Column: The way to declare a table column on a model class, giving its type and options.

db.Model: The base class in Flask-SQLAlchemy that you extend to make a model class mapped to a table.

db.relationship: A function that lets a model reach related rows as Python attributes without writing the join by hand.

db.session: The object that collects changes to models and sends them to the database on `commit()`.

DBMS: Database Management System, software that stores, manages and retrieves data in a database.

Debug mode: A Flask setting that reloads the server on code change and shows a detailed error page; it must never be on in production.

Debugging: Finding and fixing mistakes in code.

Decorator: A Python feature written with `@` above a function that wraps it to add behaviour, such as `@app.route`.

delay(): A method on a Celery task that sends it to the queue to run later.

DELETE: A method that removes a resource.

Deployment: Putting an app on a server so real users can use it.

Deserialisation: Turning received text or bytes, such as JSON, back into an object.

destroyed: A Vue 2 lifecycle hook that runs after a component is removed, a good place to clean up timers.

Destructuring: Pulling values out of arrays or objects into variables, like `const {a, b} = obj`.

Directive: A special attribute starting with `v-` that tells Vue to do something to an element.

DNS: The Domain Name System, which turns domain names into IP addresses.

Docker: A tool that packs an app and everything it needs into a container so it runs the same everywhere.

DOCTYPE: The first line `<!DOCTYPE html>` that tells the browser to treat the page as modern HTML5.

DOM: The Document Object Model, the browser's tree of objects that represents a page and that JavaScript can read and change.

Domain name: A human-friendly name like `example.com` that stands in for a numeric IP address.

Dynamic route: A route with a variable part like `/user/<int:id>` whose value is passed to the view function.

ECMAScript: The official standard that JavaScript follows; ES6 (2015) added `let`, `const`, arrow functions and classes.

emit: A Vue method a child uses to trigger a named event, optionally with data, that the parent listens to.

Endpoint: One URL, together with its allowed methods, where an API accepts requests.

Entity: A thing we want to store data about, such as a student or a course.

Environment variable: A named value outside the code, such as a secret key, that a program can read when it runs.

ER diagram: A drawing of a database design showing entities, their attributes and the relationships between them.

Error handler: A function marked with `@app.errorhandler` that makes a custom response for an error code.

Escaping: Turning special characters into harmless text so they are not treated as code.

ETag: A response header holding an identifier of a version of a resource, used to check if cached copies are still current.

Event (DOM): Something that happens in the page, such as a click or key press, that code can react to.

Event bubbling: An event first fires on the clicked element and then moves up through its parents.

Event delegation: Putting one listener on a parent to handle events from many children, using bubbling.

Event loop: The mechanism that takes finished tasks from queues and runs them when the call stack is empty, letting single-threaded JavaScript do async work.

Event modifier: A suffix like `.prevent` or `.stop` on `v-on` that changes how an event is handled.

EventSource: The browser object that opens an SSE connection and receives its events.

fetch: A browser function that sends an HTTP request and returns a promise of the response.

Fixture: A pytest function marked `@pytest.fixture` that prepares things like a test client or database for tests.

flash message: A one-time message stored in the session by `flash()` and shown on the next page.

Flask: A small Python web framework for building web apps and APIs.

Flask session: Flask's `session` object, which stores data in a signed cookie that the user can read but not change without the secret key.

Flask-Caching: A Flask extension that adds caching to views and functions, with Redis as one possible backend.

Flask-CORS: A Flask extension that adds CORS headers to responses.

Flask-JWT-Extended: A Flask extension for making and checking JWTs.

Flask-Migrate: A Flask extension that uses Alembic to track and apply changes to the database schema.

Flask-RESTful: A Flask extension that helps build REST APIs using classes, one method per HTTP verb.

Flask-Security: A Flask extension that adds login, registration, roles and password hashing.

Flask-SQLAlchemy: A Flask extension that connects SQLAlchemy to a Flask app with simple setup.

Flexbox: A CSS layout system, turned on with `display: flex`, that arranges items in a row or column and shares space between them.

Foreign key: A column that holds the primary key of a row in another table, linking the two tables.

Form: The `<form>` element that groups input fields and sends their values to the server with a chosen method and `action`.

Full-text search: Searching the words inside large text, ranking results by relevance, instead of simple pattern matching.

GET: A method that asks for a resource and should not change anything on the server.

Getter: A Vuex function that works out a value from the state, like a computed property for the store.

Git: A tool that records versions of your files so you can see history, undo changes and work with others.

GitHub: A website that hosts Git repositories online for sharing and teamwork.

Google Chat webhook: A URL that lets a program post a message into a Google Chat space using an HTTP POST.

GROUP BY: A SQL clause that bundles rows with the same value so aggregate functions can work on each bundle.

gunicorn: A production Python WSGI server that runs several worker processes to handle requests.

Hash function: A function that turns any input into a fixed-size value that cannot practically be reversed.

Hash mode and history mode: Vue Router ways to make URLs; hash mode uses `#` in the URL, while history mode makes normal URLs and needs server support.

Hashing vs encryption: Hashing is one-way and cannot be undone, while encryption can be reversed with a key.

HAVING: A SQL clause that filters groups after `GROUP BY`, whereas `WHERE` filters rows before grouping.

Higher-order function: A function that takes another function as input or returns one, such as `map` and `filter`.

Hoisting: JavaScript's moving of declarations to the top of their scope before running; `var` and function declarations are hoisted, while `let` and `const` stay unusable until declared.

HTML: HyperText Markup Language, the language that describes the structure and content of a web page using tags.

HTML5: The current version of HTML, which added semantic tags, audio, video, canvas and new form inputs.

HTTP: HyperText Transfer Protocol, the set of rules a client and server follow to exchange requests and responses on the web.

HTTP header: A `name: value` line in a request or response that carries extra information such as content type or cookies.

HTTP method: The verb in a request (GET, POST, PUT, PATCH, DELETE) that says what action the client wants.

HttpOnly: A cookie flag that stops JavaScript from reading the cookie, which reduces damage from XSS.

HTTPS: HTTP sent over an encrypted TLS connection so others cannot read or change the data on the way.

ID selector: A selector written with a hash, like `#header`, that matches the one element with that `id`.

Idempotency key: A unique value sent with a request so a repeated try is recognised and does not do the action twice.

Idempotent: Describes an operation that gives the same end result whether you do it once or many times; GET, PUT and DELETE are idempotent, POST is not.

In-memory database: A database that keeps data in RAM, which is fast but may lose data if not saved to disk.

Index: An extra data structure on one or more columns that lets the database find rows quickly without scanning the whole table.

IndexedDB: A browser database for storing larger amounts of structured data.

Inheritance (CSS): When a child element takes some property values, such as `color` and `font-family`, from its parent.

Inline element: An element like `<span>` or `<a>` that stays inside the line of text and takes only the width it needs.

Input element: The `<input>` tag, which makes a field such as text, password, checkbox or file, depending on its `type`.

Input validation: Checking that data from users has the right type, size and form before using it.

Integration test: A test that checks that several parts, such as a route and the database, work together.

Inverted index: A list that maps each word to the documents containing it, the core idea behind full-text search.

JavaScript: The programming language that runs in browsers (and in Node.js) to make pages interactive.

Jinja filter: A function applied to a value with a pipe, like `{{ name|upper }}`.

Jinja2: The templating engine used by Flask that lets HTML files contain variables, loops and conditions.

JOIN: A SQL operation that combines rows from two tables using a related column.

JSON: JavaScript Object Notation, a light text format of keys and values used to send data between programs.

JSON.parse: Turns a JSON string into a JavaScript value.

JSON.stringify: Turns a JavaScript value into a JSON string.

jsonify: A Flask function that turns Python data into a JSON response with the right content type.

JWT: JSON Web Token, a signed token with header, payload and signature that carries claims such as the user ID.

JWT signature: The part of a JWT made with a secret or private key that proves the token was not changed.

key attribute: A unique `:key` given to items in `v-for` so Vue can track and reuse elements properly.

Key-value store: A database that saves each value under a unique key and fetches it by that key.

Latency: The time it takes to get a response to a request.

Left join: A join that keeps every row from the left table and fills with NULL where the right table has no match.

Lifecycle hook: A function Vue calls at set stages of a component's life, such as `created`, `mounted`, `updated` and `destroyed`.

Load balancer: A server that spreads incoming requests over several copies of an app.

Load testing: Sending many requests to an app on purpose to see how it behaves under heavy use.

localhost: A name for your own computer, pointing to the IP address `127.0.0.1`.

localStorage: Browser storage that keeps string data with no expiry until it is cleared, for one site.

Log level: A label for how serious a message is: DEBUG, INFO, WARNING, ERROR or CRITICAL.

Logging: Writing messages about what a program is doing to a file or screen so problems can be traced later.

login_required: A decorator that blocks a route unless the user is logged in.

MailHog: A local fake mail server that catches emails during development so none go out for real.

Many-to-many: A relationship where rows on both sides can match many on the other, stored using a third link table.

map, filter, reduce: Array methods that transform every item, keep matching items, or combine all items into one value.

marshal_with: A Flask-RESTful decorator that shapes the returned object into a fixed set of output fields.

Media query: A CSS `@media` rule that applies styles only when conditions such as screen width are met.

Merge: Combining the changes from one Git branch into another.

Merge conflict: A clash when two branches changed the same lines and Git needs a person to choose.

Message broker: A middleman, such as Redis or RabbitMQ, that holds task messages from the app until a worker takes them.

Methods: Functions in the `methods` option of Vue that can be called from templates and events.

Microtask queue: A higher-priority queue for promise callbacks, which runs before the next ordinary task.

Migration: A saved step that changes a database schema, such as adding a column, which can be applied or undone.

Minification: Removing spaces and shortening names in code to make files smaller without changing behaviour.

Module (ES module): A JavaScript file that shares code using `export` and loads it using `import`.

Module (Vuex): A part of the Vuex store with its own state, getters, mutations and actions, used to split a big store.

mounted: A lifecycle hook that runs after the component has been put on the page and its element can be used.

Mustache interpolation: Showing data in a Vue template with double curly braces like `{{ message }}`.

Mutation: A Vuex function that is the only way to change state, and it must be synchronous.

MVC: Model-View-Controller, a way of splitting an app into data (model), what the user sees (view) and the logic joining them (controller).

Named slot: A slot with a `name` so a parent can fill different places in a component.

Navigation guard: A Vue Router function, like `beforeEach`, that runs before a route change and can allow, redirect or cancel it.

nginx: A fast web server often placed in front of an app to serve static files and pass other requests along.

Normalisation: Organising tables to cut repeated data and avoid update mistakes by splitting them in a careful way.

npm: The Node Package Manager, a tool and registry for installing JavaScript packages.

NULL: A special marker meaning a value is missing or unknown; it is not zero or an empty string.

One-to-many: A relationship where one row on one side matches many rows on the other, such as one teacher and many classes.

OpenAPI: A standard format, usually YAML, for describing an API's endpoints, inputs and outputs so tools can read it.

OPTIONS: A method that asks the server which methods or cross-origin actions are allowed; browsers use it for CORS preflight.

Origin: The combination of scheme, host and port of a URL; if any differs, the origin is different.

ORM: Object-Relational Mapper, a tool that lets you use Python classes and objects instead of writing SQL for tables and rows.

package.json: A file in a JavaScript project that lists its name, scripts and dependencies.

Pagination: Returning a large list in small pages so responses stay small and fast.

Parameterised query: A query where values are passed separately from the SQL text, so inputs cannot change the query's meaning.

Password hashing: Storing a one-way scrambled form of a password instead of the password itself.

PATCH: A method that changes only some fields of a resource.

Performance measurement: Measuring how fast an app is, for example response time and throughput, before and after changes.

Periodic task: A task that is run again and again on a schedule, set up with Celery beat.

Polling: The client asking the server again and again at intervals whether there is new data.

Port: A number that tells a computer which program should receive incoming data; HTTP uses 80 and HTTPS uses 443 by default.

POST: A method that sends data to the server, usually to create something or trigger an action.

Post/Redirect/Get: A pattern where a POST handler redirects to a GET page so refreshing does not resubmit the form.

Primary key: A column or set of columns whose values uniquely identify each row and are never empty.

Principle of least privilege: Giving a user or program only the permissions it really needs.

Production: The live setting where real users use the app, as opposed to development or testing.

Progressive Web App (PWA): A web app that can be installed, work offline and send notifications, using a manifest and a service worker.

Promise: An object standing for a value that will arrive later, which ends as fulfilled or rejected.

Promise.all: Waits for all promises to succeed and gives their results as an array, or fails as soon as one fails.

Props: Data a parent passes down to a child component, which the child should not change.

Props drilling: Passing data through many layers of components just to reach a deep one, which a store avoids.

Prototype: The object another object borrows properties and methods from, forming the prototype chain.

Push and pull: Push sends your commits to a remote repository; pull brings others' commits to you.

PUT: A method that replaces a whole resource with the data sent; repeating it gives the same result.

pytest: A Python testing tool that runs functions named `test_*` and checks them with plain `assert` statements.

Query (SQLAlchemy): An object, such as `User.query.filter_by(...)`, used to fetch rows as model objects.

Query string: The part of a URL after `?` that carries `key=value` pairs separated by `&`.

Race condition: A bug where the result depends on the unpredictable order of steps that run at the same time.

Rate limiting: Capping how many requests a client may make in a period to protect the server.

RBAC: Role-Based Access Control, where permissions are given to roles such as admin or user and users get roles.

Reactivity: Vue's ability to notice changes in data and update the page automatically.

Redirect: A response that tells the browser to go to a different URL.

redirect: A Flask function that tells the browser to go to another URL.

Redis: A very fast in-memory data store used for caching, queues and counters.

Refresh token: A longer-lived token used to get a new access token without logging in again.

Relational database: A database that keeps data in tables of rows and columns connected by keys.

Relationship: A link between two entities, such as a student enrolling in a course.

render_template: A Flask function that fills a Jinja2 template with data and returns the finished HTML.

Report generation: Collecting data and turning it into a document such as a PDF or CSV, usually in a background job.

reqparse: A Flask-RESTful helper for declaring and checking the inputs a request must have.

Request: A message from a client to a server that has a method, a URL, headers and sometimes a body.

Request body: The data a client sends along with a request, such as form fields or JSON.

request object: The Flask object that holds details of the current request such as `form`, `args`, `json` and `method`.

requirements.txt: A file listing the Python packages and versions a project needs.

Resource: A thing an API exposes, such as a user or a book, identified by a URL.

Resource class: A Flask-RESTful class with methods like `get` and `post`, attached to a URL by `api.add_resource`.

Response: The message a server sends back, with a status code, headers and usually a body.

Response.json(): A method that reads a fetch response body and parses it as JSON, returning a promise.

Responsive design: Building pages that adjust their layout to fit phones, tablets and desktops.

REST: Representational State Transfer, a style for designing web APIs around resources, URLs and standard HTTP methods.

RESTful API: An API that follows REST ideas, such as stateless requests and resources identified by URLs.

Result backend: A place where Celery stores task results and status so they can be read later.

Reverse proxy: A server that receives client requests and forwards them to another server behind it.

Rollback: Undoing the changes of a transaction that has not been committed.

Route: A rule linking a URL path to a Python function, made in Flask with `@app.route`.

Route params: Variable parts of a Vue route such as `/user/:id`, read with the current route object.

router-link: A Vue Router component that makes a link which changes the route without a page reload.

router-view: A Vue Router component that shows the component matching the current route.

Safe method: A method that is only meant to read data and not change the server, such as GET and HEAD.

Salt: Random data added to a password before hashing so equal passwords get different hashes.

Same-origin policy: A browser rule that stops a page from reading responses from a different origin.

SameSite: A cookie setting that limits when cookies are sent on cross-site requests, helping stop CSRF.

Scheduled job: A task set to run on its own at fixed times without anyone starting it.

Schema: The design of a database: its tables, columns, types and relationships.

Scope: The part of the code where a variable can be used.

Screen reader: Software that reads page content aloud for people who cannot see the screen.

SECRET_KEY: A private value Flask uses to sign cookies and tokens; it must be kept secret.

Selector: The part of a CSS rule that picks which elements to style, such as `p`, `.box` or `#main`.

Semantic HTML: Using tags like `<header>`, `<nav>`, `<main>` and `<footer>` that describe the meaning of content, not just how it looks.

Serialisation: Turning an object in memory into text or bytes, such as JSON, so it can be sent or stored.

Server: A program or computer that waits for requests from clients and sends back responses.

Server-sent events (SSE): A way for a server to push a stream of text messages to the browser over one long-lived HTTP connection, one way only.

Server-side rendering: Building the HTML on the server and sending the finished page to the browser.

Service worker: A script that runs in the background of the browser, between the page and network, enabling offline use and caching.

Session: Data the server remembers about one user across several requests, tied to that user by a session ID.

sessionStorage: Browser storage that keeps string data only until that tab is closed.

setInterval: A function that runs a callback again and again at a fixed interval until cleared.

setTimeout: A function that runs a callback once after a delay.

Single file component: A `.vue` file that holds a component's template, script and style together.

Single-page application (SPA): A web app that loads one page and then updates its content with JavaScript instead of loading new pages.

Slot: A placeholder in a component's template where the parent can insert its own content.

SMTP: Simple Mail Transfer Protocol, the protocol used to send email between servers.

Specificity: The scoring that decides which CSS rule wins when several match; ID beats class, and class beats element.

SQL: Structured Query Language, the language for creating, reading, changing and deleting data in relational databases.

SQL injection: An attack where a user types SQL into an input so it changes your query; stopped by parameterised queries or an ORM.

SQLAlchemy: A Python library with an ORM and a toolkit for working with SQL databases.

SQLite: A lightweight relational database that stores everything in a single file and needs no separate server.

Stack trace: The list of function calls shown when an error occurs, helping you find where it came from.

Stale data: Old cached data that no longer matches the real data.

State (Vuex): The single object in the Vuex store that holds the shared data.

Stateless: Describes a server that keeps no memory of earlier requests, so each request must carry everything needed to handle it.

Static file: A file such as an image, CSS or JavaScript file that is sent as it is, kept in Flask's `static` folder.

Status code: A three-digit number in a response that tells whether the request worked and why not if it failed.

Store: The Vuex object that holds state, getters, mutations and actions.

Subquery: A query written inside another query.

Swagger: A set of tools built around OpenAPI, including a page for trying out API calls in the browser.

Synchronous: Code that runs one step at a time, with each step waiting for the previous one.

Table: A grid in a database made of rows and columns that holds one kind of data.

Task queue: A waiting line of callbacks, such as timer callbacks, that run once the stack is empty.

Task queue (Celery): A list of jobs waiting for worker programs to pick them up and run them.

Template: An HTML file with placeholders that get filled with real data before it is sent to the user.

Template (Vue): HTML with Vue syntax that describes what a component shows based on its data.

Template inheritance: Making a base template with `{% block %}` areas that child templates fill in with `{% extends %}`.

Template literal: A string in backticks that can include values with a dollar sign and braces and span many lines.

Temporal dead zone: The time between entering a scope and the line where a `let` or `const` is declared, when using it throws an error.

Test client: A Flask tool, `app.test_client()`, that sends fake requests to the app without running a server.

text/event-stream: The content type a server uses to send server-sent events.

then, catch, finally: Promise methods that handle success, handle failure, and run code in either case.

Third-party cookie: A cookie set by a site other than the one in the address bar, often blocked by browsers.

this: A keyword whose value depends on how a function is called; arrow functions use the `this` of the code around them.

Throughput: How many requests a system handles in a given time.

Token authentication: Logging in once to get a token, then sending that token with each request instead of a password or session cookie.

Token expiry: The time after which a token is no longer accepted, limiting harm if it is stolen.

Transaction: A group of database steps that are treated as one unit and either all succeed or all are undone.

Truthy and falsy: Values that act as true or false in a condition; falsy ones are `false`, `0`, `""`, `null`, `undefined` and `NaN`.

try/catch with await: The way to handle errors from awaited promises, with `catch` receiving a rejection.

TTL: Time To Live, how long a cached item stays valid before it expires.

Unicode: A standard that gives a unique number, called a code point, to every character in almost every writing system.

Unit test: A small automatic test that checks one piece of code, such as one function.

updated: A lifecycle hook that runs after changed data has caused the page to re-render.

URL: The full address of a resource on the web, made of scheme, host, optional port, path, query string and fragment.

url_for: A Flask function that builds a URL from a view function name so links do not break when routes change.

UTF-8: A way to store Unicode characters as bytes, using one to four bytes per character and one byte for plain English letters.

v-bind: A directive that ties an attribute to data, often written as `:`, like `:src="url"`.

v-else: A directive used right after `v-if` that shows when the condition is false.

v-for: A directive that repeats an element for each item of a list.

v-if: A directive that adds or removes an element from the page depending on a condition.

v-model: A directive that keeps a form input and a data property in sync both ways.

v-on: A directive that listens to events, often written as `@`, like `@click="save"`.

v-show: A directive that hides or shows an element using CSS `display`, keeping it in the page.

View function: The Python function that runs for a route and returns the response.

Viewport meta tag: The tag `<meta name="viewport" ...>` that makes mobile browsers use the device width, needed for responsive pages.

Virtual DOM: A light copy of the page structure in JavaScript that Vue compares with its earlier copy to change only what differs in the real DOM.

Virtual environment: An isolated Python setup with its own installed packages for one project.

Vue CLI: A command-line tool for creating and building Vue projects with a ready setup.

Vue instance: The object made with `new Vue({ ... })` that links data, methods and a page element.

Vue Router: Vue's official library for showing different components for different URLs in a single-page app.

Vue.js: A JavaScript framework for building user interfaces from reactive data and components.

Vuex: Vue's official store library that keeps shared state for all components in one place.

Watcher: A function in `watch` that runs when a chosen data property changes, good for side effects such as fetching.

Web: A huge collection of pages and resources linked together and reached through browsers over the internet using URLs and HTTP.

Web app manifest: A JSON file that gives an app's name, icons and colours so it can be installed on a device.

Web storage: Browser storage for key-value data, made of `localStorage` and `sessionStorage`.

Webhook: A way for one service to call a URL you provide when something happens, so you get told instead of asking again and again.

Webpack: A tool that bundles many JavaScript, CSS and other files into a few files for the browser.

WebSocket: A connection that stays open and lets client and server send messages to each other both ways at any time.

Worker: A Celery process that takes tasks from the broker and runs them.

WSGI: Web Server Gateway Interface, the Python standard for how a web server talks to a Python web app.

XSS: Cross-Site Scripting, an attack where an attacker's script runs in other users' browsers, stopped by escaping output.