

ChatGPT - Abdurrrehman



i want to study web programming i want you to make me a master in everything ill give u the syllabus and the old papers madam have prepared so u can have a better idea of what type of questions madam makes also il give u the slides

Perfect — I'm fully ready to help you **master web programming** based on your **syllabus**, **slides**, **and old exam papers**. Here's how we'll proceed step-by-step:

Step-by-Step Plan to Master Web Programming

Step 1: Upload Everything

- Please upload the syllabus
- Upload any **slides** you've received (PDFs, PPTs, etc.)
- Upload old question papers your teacher (madam) provided

Step 2: I'll Analyze and Break Down:

- The **topics** you need to cover
- The question styles your madam prefers (MCQs, coding, theory, etc.)
- The exam structure and key marking areas

Step 3: Your Personal Study Plan

Based on everything, I'll create a **custom weekly study plan**:

- Detailed topic-by-topic explanations
- Code exercises for hands-on learning
- Mock tests similar to madam's style
- Slide summaries and cheat sheets

Step 4: 1-on-1 Practice

- I'll help you write, debug, and improve **HTML**, **CSS**, **JavaScript**, **PHP**, or whatever languages are in your syllabus
- You'll get real examples, past paper solutions, and practice questions with answers

M Once you upload the materials, I'll start by:

- Giving you a topic map
- Highlighting **important patterns** in the past papers

• Creating your first batch of lessons + questions

Please go ahead and upload the **syllabus**, **slides**, **and past papers**, and I'll take it from there.



syllabus

Dear students, the syllabus for final exam is:

- 1. Web Architecture
- 2. HTML
- 3. HTTP
- 4. JS (let, const, arrays, objects, loops, conditions, array and object destructuring, arrow functions, DOM Manipulation, event handlers)
- 5. call backs, promises, async/await, event loop
- 6. fetch api
- 7. react (functional components, props in functional components, useEffect, useState, useRef, conditional rendering, list rendering, react router)
- 8. Redux toolkit
- 9. Mongoose models, mongoose basic crud operations (create, delete, find)
- 10. Express APIs and middleware
- 11. express router middleware and controllers
- 12. req (req.param, req.query, req.body, req.headers, etc.) and res (res.status(), res.json(), res.send(), etc) objects in express APIs
- 13. authentication using JWT tokens, authorization
- 14. cookies

Please note that styling (including css), class components, simple redux and lazy loading are not included in the exam.

TWO A4 Size handwritten cheat sheets with Roll# written on top of both sheets and stapled together are allowed. Photocopied or typed cheat sheets are not allowed.

Simple redux is also not included.

Great — thanks for sharing the syllabus. You're aiming to master a very practical and in-demand tech stack, often called the **MERN stack** (MongoDB, Express, React, Node.js) plus core web technologies (HTML, HTTP, JS).

Next Step: Upload the Slides & Old Papers

Please upload:

- **!** Lecture slides
- **Old question papers** your madam has given (midterms, quizzes, etc.)

Once you upload them, I will:

1. Analyze past paper patterns



- 2. Create chapter-wise breakdowns
- 3. Start training you with examples, practice questions, and quick summaries

Here's How We'll Cover Your Syllabus:

Торіс	What You'll Learn
1. Web Architecture	Client-server, HTTP cycle, backend/frontend split
2. HTML	Tags, forms, input types, semantic elements
3. HTTP	Methods, headers, status codes
4. JavaScript	Variables, logic, data structures, DOM, ES6 features
5. Async JS	Callbacks, Promises, `async/await`, Event Loop
6. Fetch API	Making network requests, handling responses/errors
7. React	Functional components, Hooks, rendering logic, Router
8. Redux Toolkit	Slices, store, dispatching actions
9. Mongoose	Models, Schemas, CRUD in MongoDB
10–12. Express	Building APIs, routers, middleware, ` req `/` res `
13–14. Auth & Cookies	JWT tokens, protected routes, cookie handling

Once I get your materials, we'll begin structured prep with:

- Q Concept + Code walkthroughs
- **?** Past-paper-style mock questions
- 📤 Please upload the old papers and slides now.



there is still some slides left ill give u them next

Perfect — I've now reviewed all your uploaded slides across topics like:

- Web Architecture
- V HTML & HTTP
- **V** JavaScript (Basics, DOM, Advanced, Async, Events, Destructuring)
- V React Setup & Components (State, Props, useEffect, useState, etc.)

Next, I'll wait for the remaining slides and the **old papers** from madam to analyze the type of questions she asks.

While You Upload, Here's Your Custom Study Structure:



I've broken your syllabus into **14 core topics** based on the slides + syllabus:

#	Торіс	Ready?
1	Web Architecture	V
2	HTML	✓
3	НТТР	V
4	JavaScript (let, const, loops, conditions)	V
5	Arrays & Objects	V
6	Arrow Functions	V
7	DOM Manipulation	✓
8	Event Handlers	V
9	Callbacks, Promises, Async/Await, Event Loop	V
10	Fetch API	V
11	React Functional Components + Props + Hooks	V
12	React Router	🔀 Waiting for slides
13	Redux Toolkit	🔀 Waiting for slides
14	Express, Mongoose, JWT, Cookies	🔀 Waiting for slides

<u>煄</u> Coming Up For You:

Once you upload the rest:

- I'll map questions madam likes (e.g., code tracing, explanation-based, syntax filling, true/false)
- I'll generate **practice questions** and **mock exam sets**
- You'll get **cheat-sheet-style notes** (you're allowed two handwritten A4 sheets, right? I'll help make those efficient)

Sample Learning Output You'll Get:

```
javascript

// DOM Event Example
const btn = document.getElementById("clicker");
btn.addEventListener("click", () => alert("Button Clicked!"));
```

- ✓ Breakdown of concept
- ✓ What madam might ask
- ✓ Real code
- ✓ Trick questions to expect

... Go ahead and upload the remaining slides and madam's past papers now — I'll turn this into your personalized full exam-prep plan.



All slides are now received and processed. You're giving me everything I need to make you a Web Programming master by final exam time.

What's Next: Upload Old Question Papers

Please upload the **past papers or sample papers** your madam prepared (quizzes, midterms, mock questions, or anything similar). These help me:

- Identify her **question style** (coding, explainers, MCQs, true/false)
- Detect repeat topics
- Prepare realistic mock tests for you

📕 Summary of What You've Uploaded (So Far)

Section	Slides	
✓ Core Topics	Web Architecture, HTML, HTTP, JavaScript, DOM, Events	
✓ React	Setup, Functional Components, State, Props, Hooks, List Rendering, Prop Drilling	
✓ Redux Toolkit	Full slice-based architecture	
✓ Backend	Node.js, Express, CRUD APIs, Mongoose Models, JWT, Cookies	
🔀 Awaiting	Old papers for pattern detection	

Once You Upload Papers, I'll Deliver:

- 1. Madam-style Mock Exam Questions
- 2. Mini Projects or Exercises per Topic
- 3. Custom Cheat Sheet (2 A4 handwritten allowed)
- 4. Smart Revision Plan for Exam Week
- 5. Code + Theory Combo Questions
- ♣ Upload old question papers now I'll generate personalized practice questions for you right away.



u have everything now

Perfect. V You've now shared:

- All your syllabus + slides
- Both Midterm papers (Mid 1 and Mid 2)

Analysis of Madam's Paper Style

Key Patterns from Midterms:

- 1. Conceptual + Code Hybrid: Explaining concepts and writing small JS/React code snippets
- 2. **Short Qs (2-5 marks)** on:
 - o Definitions (e.g. "What is the DOM?")
 - Code predictions (e.g. "What will this print?")
 - Difference-type (e.g. `let` vs `const`, `props` vs `state`)

3. Medium Qs (5-8 marks):

- Tracing code (e.g. React or JavaScript execution)
- DOM + Events
- o Async flow, Event loop, Promises
- 4. Long Qs (10+ marks):
 - Explain a concept with examples
 - Write code (e.g., make a form, build a component, simulate Redux logic)

o Your Personal Web Programming Mastery Plan

Phase 1: Core JavaScript & Browser Skills

- Variables, arrays, destructuring
- Loops, conditions, arrow functions
- DOM Manipulation, Event Listeners
- Callbacks, Promises, Async/Await, Event Loop

Phase 2: React Mastery

- Functional Components, `useState`, `useEffect`, `useRef`
- Conditional & List Rendering
- Prop Drilling, Lifting State
- React Router Basics

Phase 3: Advanced State & API

- Redux Toolkit (Slices, Store, Dispatch)
- Fetch API & Async API calls
- CRUD with Mongoose
- Express Routing, Middleware, Controllers
- `req` and `res` usage

Coming Up for You:

I'll now create:

- 1. Topic-by-topic short notes
- 2. Mock questions like madam's
- 3. A personalized 7-day revision plan
- 4. Your 2-page handwritten cheat sheet blueprint
- Let's Begin with First Study Pack:

Would you like to start:

- 1. From the **beginning (HTML/HTTP/JS)**
- 2. From React
- 3. From Backend (Express + Mongoose + JWT)
- Just tell me **where you want to begin**, and I'll drop your first learning pack with notes, code, and practice questions.



first tell me what slides are excluded from the syllabus

Great question. Based on your **syllabus** and the **slides you uploaded**, here is a clear list of **excluded vs included** content:

✓ Included in Final Exam (According to Your Syllabus)

Торіс	Slides That Are Relevant
Web Architecture	`Lecture 1 - Web Architecture` 🗹
HTML	`Lecture 2`, `Lecture 3+4` 🗸 (CSS excluded)
НТТР	`Lecture 2` 🗸
JavaScript (basics + DOM + events)	`Lecture 5`,`Lecture 6`,`Lecture 7`,`Lecture 8` 🗹
Callbacks, Promises, Async/Await, Event Loop	`Lecture 7` 🗸
Fetch API	`Lecture 7` 🗸
React (Functional)	`Lecture 9`,`Lecture 10`,`Lecture 11` 🗸
React Router	`Lecture 9`,`Lecture 10` 🗹
Redux Toolkit	`Lecture 12 & 13` ✓
Mongoose (models, CRUD, relations)	`Lecture 15`,`Lecture 16` 🗹
Express (API, middleware, router, controller)	`Lecture 17`,`Lecture 18` 🗹
req/res objects	`Lecture 18` 🗹



Торіс	Slides That Are Relevant
JWT Authentication + Authorization	Covered indirectly (in context of cookies, `Lecture 19`) 🗸
Cookies	`Lecture 19` 🗸

Excluded Topics (Do NOT Study These for Final)

Slide	Excluded Content	
`Lecture 3+4`	X CSS (selectors, borders, flex, media queries, etc.)	
`Lecture 10`	X Class Components (just skip this part of the React lecture)	
`Lecture 12 & 13`	X Simple Redux (only Redux Toolkit is included)	
`Lecture 15— 16`	✓ Mongoose is included , but advanced schema validators not explicitly mentioned – study basic CRUD and basic relationships only	
`Lecture 19`	☑ Cookies are included , but don't go too deep into domain/subdomain rules unless asked specifically	

Final Summary

Focus your time on:

- JavaScript logic + DOM + async flow
- React functional components and Hooks
- Backend logic (Express, Mongoose)
- Redux Toolkit
- APIs, JWT, and cookies

Avoid studying:

- CSS
- React class components
- Simple Redux (non-toolkit version)
- Lazy loading (not mentioned at all)

Ready for your first personalized learning module or mock question set? Just tell me the topic you want to start with.



now i am ready lets start teach me everything and make me a master in web, lets go over topic by topic and as u suggested do madam style practice questions at the end with answers start with the first topic

Perfect. Let's begin your journey to **mastering Web Programming** — one topic at a time, with **clear explanations**, **real code**, **and madam-style practice questions**.



Topic 1: Web Architecture

What Is Web Architecture?

Web architecture is how different components of a web application work together — it's the **backbone** of the web. It defines:

- How the client (browser) talks to the server
- How data flows between frontend, backend, and database
- How apps scale and remain responsive

Key Concepts

1. Client-Server Model

- Client: Browser or mobile app that sends requests.
- **Server**: Receives requests, processes data, and sends responses.
- **Database**: Stores persistent data (e.g. MongoDB).

Think of:

User opens website \rightarrow Browser sends request \rightarrow Server processes \rightarrow Responds with HTML/JSON

2. Architecture Tiers

Туре	Layers	
2-tier	Client ↔ Server/Database	
3-tier	Client ↔ App Server ↔ Database	
N-tier	Add more layers like: Authentication, Caching, etc.	

3-tier is most common in modern apps

3. Microservices Architecture

- System split into independent small services (e.g., Auth Service, Product Service)
- Improves scalability, maintainability, and fault tolerance

4. RESTful Design Principles

- Stateless: Server doesn't remember users every request must carry all needed info (like JWT)
- **Uniform interface**: Standard URL routes



• HTTP Methods:

o `GET`: Read data

o `POST`: Create

∘ `PUT`: Update

○ `DELETE`: Delete

5. Scalability

Туре	Description	
Horizontal scaling	Add more servers (scale out)	
Vertical scaling	Add more RAM/CPU to one server (scale up)	

6. TCP/IP & HTTP

- TCP/IP: Foundation protocol of internet communication
- **HTTP**: Protocol for **web page delivery** (we'll do this in Topic 2)

Bonus: What is a URL?

Uniform Resource Locator

Example: `https://www.facebook.com/login`

- `https` → Protocol
- `www.facebook.com` → Domain
- `/login` → Path (resource)

Madam-Style Practice Questions (with Answers)



📝 Q1: Define Web Architecture. (2 marks)

Answer:

Web architecture is the structure and design of how web components — like the frontend, backend, and database — interact using protocols like HTTP. It ensures data flows efficiently and securely between client and server.



📝 Q2: What is the difference between 2-tier and 3-tier architecture? (3 marks)

Answer:

2-Tier	3-Tier
Client talks directly to database	Has middle layer (App Server)
Less secure, faster	More secure, scalable



📝 Q3: What are benefits of Microservices Architecture? (3 marks)

Answer:

- Easier to maintain and update individual services
- Independent scalability
- Fault isolation: One failing service doesn't crash entire app

📝 Q4: Label this architecture diagram: (4 marks)

```
less
[Browser] → [App Server] → [MongoDB]
              Backend API
 Frontend
```



📝 Q5: Define RESTful API. (2 marks)

Answer:

A RESTful API is a stateless web API that follows standard HTTP methods (GET, POST, PUT, DELETE) and returns resources in formats like JSON.

? Q6 (Code-style):

Which HTTP method would you use to update a user's email in the database? (1 mark) Answer: `PUT` or `PATCH`

📝 Q7 (Real-life logic):

Why is horizontal scaling preferred in cloud systems? (2 marks)

Answer: It allows adding more servers for handling load, which is cheaper and more flexible than upgrading a single server.

Ready to move on to **Topic 2: HTML + HTTP**?

Or do you want a quick quiz on Web Architecture before we continue?



next

Great! Let's now dive into:



Topic 2: HTML + HTTP

We'll split this into two parts:

- 1. **HTML** structure of web pages
- 2. **HTTP** how browsers and servers talk

Part A: HTML – Building Blocks of Web Pages

What is HTML?

HTML (HyperText Markup Language) is the language that structures content on the web. It defines headings, paragraphs, links, images, forms, and more.

✓ Basic Structure:

```
html

<!DOCTYPE html>
<html>
  <head>
        <title>My Page</title>
        </head>
        <body>
            <h1>Welcome!</hl>
            This is a paragraph.
            </body>
            </html>
```

Common Tags:

Element	Use
` <h1>` to `<h6>`</h6></h1>	Headings
``	Paragraph
` `	Link
` `	Image
` `,``,``	Lists
` <form>`,`<input/>`,`<textarea>`</th><th>Forms</th></tr><tr><th>``,``,``,``</th><th>Tables</th></tr><tr><th>`<div>`</th><th>Section/Box</th></tr><tr><th>`<button>`</th><th>Button</th></tr></tbody></table></textarea></form>	

Form Input Types:

- `type="text"` → Input box
- $\bullet \ \ \, \textbf{`type="password"`} \to Hidden\ text$
- $\bullet \ \texttt{`type="checkbox"`, `type="radio"`} \to Options \\$
- `type="date"`, `type="file"` \rightarrow Date, file upload
- `type="submit"` → Submit button

Semantic Tags (HTML5):

Тад	Purpose
` <header>`,`<footer>`</footer></header>	Top/Bottom content
` <article>`,`<section>`</section></article>	Structured blocks
` <aside>`</aside>	Side content
` <nav>`</nav>	Navigation bar

Part B: HTTP - Web Communication Protocol

What is HTTP?

HTTP (HyperText Transfer Protocol) is how your browser communicates with a server.

- HTTP Request Flow:
 - 1. Client (browser) sends a request → `GET /index.html`
 - 2. **Server** processes and returns a response \rightarrow `200 OK + HTML`

Request Example:

```
http

GET /home HTTP/1.1

Host: www.example.com
User-Agent: Chrome
```

Response Example:

```
h

HTTP/1.1 200 OK

Content-Type: text/html

<html>...</html>
```

• Common HTTP Methods:

Method	Use
`GET`	Retrieve data
`POST`	Send new data
`PUT`	Update existing data
`DELETE`	Remove data

Status Codes:



Code	Meaning
`200`	ОК
`404`	Not Found
`500`	Server Error
`403`	Forbidden
`301`	Moved Permanently

V

Madam-Style Practice Questions

Q1: What is HTML? Give one example of a heading tag. (2 marks)

Answer:

HTML is the standard language for creating webpages. Example: `<h1>This is a heading</h1>`

📝 Q2: What will the following HTML display? (2 marks)

```
html

    Cat
    Dog
```

Answer:

A bullet list with:

- Cat
- Dog

Q3: Difference between `<input type="radio">` and `<input type="checkbox">` (3 marks)

Answer:

- `radio`: only one option can be selected
- `checkbox`: multiple options can be selected

Q4: Write HTML to create a form with name and password fields and a submit button. (5 marks)

Answer:

```
html

<form>
    <label>Name:</label>
    <input type="text" name="name">
```

```
<label>Password:</label>
  <input type="password" name="pass">
     <input type="submit" value="Login">
     </form>
```

📝 Q5: What is HTTP? Is it stateful or stateless? (2 marks)

Answer:

HTTP is the protocol for web communication. It is **stateless**, meaning each request is independent.

Q6: Which method is used to update a user in HTTP API? (1 mark)

Answer: `PUT`

📝 Q7: Match the code to status:

- 200 → OK
- 404 → ?
- 500 → **?**

Answer:

- 404 → Not Found
- 500 → Server Error
- You're now done with Topic 2!
- Next up: JavaScript Basics (let, const, arrays, loops, conditions, destructuring, arrow functions).
 Shall we begin that?



yeah next

Awesome! Now let's master:

Topic 3: JavaScript Basics

Covers:

- `let`, `const`, `var`
- Arrays & Objects
- Loops & Conditions
- Destructuring
- Arrow Functions

1. Variable Declarations: `var` vs `let` vs `const`

Keyword	Scope	Reassignable?	Redeclarable?
`var`	Function-scoped	✓ Yes	✓ Yes
`let`	Block-scoped	✓ Yes	X No
`const`	Block-scoped	X No	X No

Prefer `let` or `const`. Only use `const` if you won't reassign.

2. Data Types

• 3. Conditions

```
if (age >= 18) {
  console.log("Adult");
} else {
  console.log("Minor");
}
```

4. Loops

`for` loop

```
js

for (let i = 0; i < 5; i++) {
   console.log(i);
}</pre>
```

`for...of` (arrays)

```
for (let fruit of fruits) {
   console.log(fruit);
}
```

`for...in` (objects)

```
for (let key in user) {
   console.log(key, user[key]);
}
```

5. Arrays & Objects

6. Destructuring

Array Destructuring:

```
js

const [first, second] = ["Ali", "Ahmed"];
console.log(first); // "Ali"
```

V Object Destructuring:

```
const user = { name: "Zain", age: 30 };
const { name, age } = user;
console.log(name); // "Zain"
```

7. Arrow Functions

Basic:

```
js
const add = (a, b) => a + b;
```

With single param:

```
js
const greet = name => console.log("Hi", name);
```

Multiline:

```
js

const square = n => {
   return n * n;
};
```

Madam-Style Practice Questions

Q1: What is the difference between `let` and `var`? (2 marks)

Answer:

`let` is block-scoped and not redeclarable.

`var` is function-scoped and can be redeclared.

Q2: Output? (2 marks)

```
js

const a = 5;
a = 10;
console.log(a);
```

Answer: ★ Error — `const` cannot be reassigned.

Q3: Write a loop to print numbers from 1 to 5. (2 marks)

```
js

for (let i = 1; i <= 5; i++) {
   console.log(i);
}</pre>
```

Q4: Use destructuring to extract `age` from this object: (2 marks)

```
js
const student = { name: "Ali", age: 22 };
```

Answer:

```
js
const { age } = student;
```

📝 Q5: Write an arrow function that multiplies two numbers. (2 marks)



```
js
const multiply = (a, b) => a * b;
```

📝 Q6: What is the output? (3 marks)

```
const arr = ["apple", "banana"];
const [a, b] = arr;
console.log(a + " & " + b);
```

Answer: `apple & banana`

Ready for **Topic 4: DOM Manipulation** (document selectors, events, style updates)? Or want a **quick JavaScript quiz** first?



yeah next

Perfect — now let's tackle:

📕 Topic 4: DOM Manipulation

This is where **JavaScript interacts with HTML** to make web pages dynamic.

• What is the DOM?

DOM (Document Object Model) is a **tree-like structure** of your HTML elements that JavaScript can:

- Access
- Modify
- Add or delete
- PDOM is stored in memory, not in the HTML file itself.

Accessing DOM Elements

Method	Description
`document.getElementById("id")`	Select by ID
`document.getElementsByClassName("class")`	Select by class
`document.getElementsByTagName("tag")`	Select by tag name
`document.querySelector("selector")`	First matching CSS selector
`document.querySelectorAll("selector")`	All matching elements

```
js

const para = document.getElementById("myPara");
const boxes = document.querySelectorAll(".box");
```

Changing Content and Styles

```
element.innerHTML = "<b>New HTML</b>"; // Set HTML
element.textContent = "Just text"; // Set plain text
element.style.color = "red"; // Change style
```

Creating & Appending Elements

```
const newDiv = document.createElement("div");
newDiv.textContent = "I'm new!";
document.body.appendChild(newDiv);
```

Removing/Replacing Elements

```
element.remove(); // removes the element itself
parent.removeChild(child); // classic way
parent.replaceChild(newEl, oldEl);
```

Event Listeners

```
document.getElementById("btn").addEventListener("click", () => {
   alert("Clicked!");
});
```

Event Propagation

There are **3 phases** of event propagation:

- 1. Capturing: top-down
- 2. Target: event reaches target element
- 3. **Bubbling (default)**: bottom-up

```
// Add event in bubbling phase (default)
element.addEventListener("click", handler);
// Add event in capturing phase
element.addEventListener("click", handler, true);
```

DOM Traversal

```
el.parentNode; // Parent element
el.children; // All children
el.firstElementChild; // First child
el.lastElementChild; // Last child
el.nextElementSibling; // Next element
el.previousElementSibling; // Previous element
```

🔽 Example Code

```
html

<div id="box">Click me</div>
<script>
    const box = document.getElementById("box");
    box.addEventListener("click", () => {
        box.textContent = "You clicked me!";
        box.style.backgroundColor = "yellow";
    });
</script>
```

Madam-Style Practice Questions

2 Q1: What is the DOM? (2 marks)

Answer:

DOM is a tree-like structure that represents HTML elements in memory. JavaScript uses it to read and modify the page dynamically.

Q2: Write code to change the color of a paragraph with ID "para" to blue. (2 marks)

```
js
document.getElementById("para").style.color = "blue";
```

Q3: Add a new paragraph using JavaScript and append it to a `div` with ID `container`. (3 marks)



```
let para = document.createElement("p");
para.textContent = "New paragraph!";
document.getElementById("container").appendChild(para);
```

📝 Q4: Write code to remove a paragraph with ID `text1`. (2 marks)

```
js
document.getElementById("text1").remove();
```

📝 Q5: Describe the three phases of event propagation. (3 marks)

Answer:

- Capturing: Event travels from root to target
- Target: Event hits the target element
- **Bubbling**: Event bubbles up to parent elements (default)

📝 Q6: What will this output when the button is clicked? (3 marks)

```
html

<button id="btn">Click</button>
  <script>
    document.getElementById("btn").addEventListener("click", () => {
        alert("Clicked!");
    });
    </script>
```

Answer: A popup alert saying `"Clicked!"`

Ready to move on to **Topic 5: Callbacks, Promises, Async/Await, and Event Loop**? This is important for both JS and API handling.



yeah next

Great! Let's now master one of the most critical and often tested topics:

Topic 5: Callbacks, Promises, Async/Await, and the Event Loop

This topic is about handling asynchronous operations like API calls, timers, and file loading.

1. Callback Functions

A **callback** is a function passed as an argument to another function, to be **called later**.

```
function greet(name, callback) {
  console.log("Hi " + name);
  callback();
}
greet("Ali", () => console.log("Welcome!"));
```

Used in: `setTimeout`, `event listeners`, `fetch`, etc.

2. Promises

A **promise** represents a value that will be available in the **future**.

```
let promise = new Promise((resolve, reject) => {
  let success = true;
  if (success) resolve("Done!");
  else reject("Error!");
});

promise
  .then(result => console.log(result))
  .catch(error => console.log(error));
```

- States of a promise:
 - `pending`
 - `fulfilled`
 - `rejected`

3. Async/Await

Cleaner syntax to work with promises.

```
async function fetchData() {
   try {
    let res = await fetch("https://jsonplaceholder.typicode.com/posts/1");
   let data = await res.json();
    console.log(data);
   } catch (err) {
    console.error(err);
   }
}
fetchData();
```

💡 `await` pauses the function until the promise resolves or rejects.

4. Event Loop: Behind-the-Scenes Magic

JavaScript is **single-threaded**, but it handles async code using:

Component	Description
Call Stack	Executes functions (synchronous only)
Web APIs	Handle async tasks (timers, fetch)
Callback Queue	Stores tasks waiting to run
Microtask Queue	Stores `.then() `callbacks
Event Loop	Moves tasks to the stack when it's empty

Microtask vs Macrotask

Queue	Used For	Priority
Microtask	Promises, `async/await`	☑ Higher
Macrotask	`setTimeout`, UI Events	X Lower

Example:

```
console.log("1");
setTimeout(() => console.log("2"), 0); // Macrotask
Promise.resolve().then(() => console.log("3")); // Microtask
console.log("4");
// Output: 1 → 4 → 3 → 2
```

✓ Madam-Style Practice Questions

Q1: What is a callback function? Give one example. (2 marks)

Answer:

A function passed into another function to be called later.

```
js
setTimeout(() => console.log("Waited!"), 1000);
```

📝 Q2: What are the 3 states of a promise? (2 marks)

Answer:

- Pending
- Fulfilled
- Rejected

📝 Q3: Convert this callback to a promise-based version. (3 marks)

```
function doTask(cb) {
  cb("Task done!");
}
```

Answer:

```
function doTask() {
   return Promise.resolve("Task done!");
}
doTask().then(msg => console.log(msg));
```

2 Q4: What will this log? Explain why. (3 marks)

```
console.log("A");
setTimeout(() => console.log("B"), 0);
Promise.resolve().then(() => console.log("C"));
console.log("D");
```

Answer:

 $A \rightarrow D \rightarrow C \rightarrow B$

Because microtasks (`C`) are handled before macrotasks (`B`).

2 Q5: Fill in the blank: (2 marks)

```
async function example() {
  let res = await fetch("url");
  let data = await res.json();
  return data;
}
```

Q: What is the keyword used to wait for a promise?

Answer: `await`

Q6: Write a function that waits 2 seconds and then logs "Done Waiting" using async/await. (3 marks)

```
function wait(ms) {
  return new Promise(res => setTimeout(res, ms));
}
async function run() {
  await wait(2000);
  console.log("Done Waiting");
```

```
}
run();
```

You're now done with one of the trickiest JS topics.

Ready for Topic 6: Fetch API? (This is how frontend sends/receives data from the backend)



yeah

Awesome — let's jump into:

📕 Topic 6: Fetch API

This is how your frontend communicates with servers (like your Express API).

What is Fetch?

The Fetch API allows JavaScript to make HTTP requests (GET, POST, PUT, DELETE) to servers and APIs.

It's **promise-based** and works well with `async/await`.

1. Basic GET Request

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

- Fetch returns a promise
- `res.json()` parses the body to a JS object

2. Using Async/Await

```
async function getPost() {
   try {
     const res = await fetch("https://jsonplaceholder.typicode.com/posts/l");
     const data = await res.json();
     console.log(data);
   } catch (err) {
     console.error(err);
   }
}
```

3. POST Request (Send Data)

```
async function addUser() {
  const res = await fetch("https://jsonplaceholder.typicode.com/users", {
    method: "POST",
    headers: {
        "Content-Type": "application/json"
    },
    body: JSON.stringify({ name: "Ali", age: 20 })
});
  const data = await res.json();
  console.log(data);
}
```

4. PUT & DELETE Requests

```
// PUT: Update
fetch("/api/users/123", {
  method: "PUT",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ name: "Updated Name" })
});

// DELETE
fetch("/api/users/123", {
  method: "DELETE"
});
```

5. Common Mistakes

Mistake	Fix
Forgot `await` before `fetch`	`await fetch()`
Missing `"Content-Type"` in headers	Add` "Content-Type": "application/json" `
Using `.json()` without `await`	Use `await res.json()`

Madam-Style Practice Questions

Q1: Write a fetch GET request to get data from `/api/products`. (2 marks)

```
fetch("/api/products")
  .then(res => res.json())
  .then(data => console.log(data));
```

Q2: What does `res.json()` do? (2 marks)

Answer:

It converts the response body from JSON to a JavaScript object.

Q3: Complete this POST request to send `{ name: "Sara" }` to `/api/users` (3 marks)

```
fetch("/api/users", {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({ name: "Sara" })
});
```

📝 Q4: What is the output? (2 marks)

```
js

const res = await fetch("/api/posts/1");
console.log(typeof res);
```

Answer:

`object` - it's a `Response` object

Z Q5: Which header must be added in a POST request when sending JSON? (1 mark)

Answer:

`"Content-Type": "application/json"`

You're now ready to start frontend-backend communication!

Next up: React Functional Components, Props, useState, useEffect, useRef, Conditional & List Rendering?

Shall we begin the **React section**?



yeah react is big and combines multiple lectures so i want comprehensive notes

Excellent decision! React is a core part of your exam and syllabus — and yes, it combines multiple lectures. Here's how we'll break it down comprehensively:

React (Complete Overview for Final)

Covers **functional components**, **hooks**, **props**, **state**, **list rendering**, **conditional rendering**, **lifting state**, **prop drilling**, **React Router**, **and useRef** — exactly as per your syllabus.

1. Setting Up a React App



```
npm create vite@latest my-app -- --template react
cd my-app
npm install
npm install react-router-dom
npm run dev
```

Your project has:

- `main.jsx` entry point
- `App.jsx` root component
- `components/` reusable components

2. Functional Components

```
jsx

function Greet() {
  return <h1>Hello!</h1>;
}
export default Greet;
```

- Always return JSX.
- Should be **pure** (no side-effects inside body).
- Prefer arrow functions:

```
jsx
const Greet = () => <h1>Hello</h1>;
```

3. JSX Syntax

- JSX lets you write HTML in JS
- Use `{}` for JS expressions
- One parent element is required

```
jsx

const name = "Ali";
return <h1>Hello, {name}</h1>;
```

4. Props (Read-only Data from Parent)

```
jsx
const Welcome = ({ name }) => <h2>Hi {name}</h2>;
```

```
// Usage
<Welcome name="Ali" />
```

Props cannot be changed inside the child.

5. useState (Component State)

6. useEffect (Side Effects)

Runs after render. You can control **when** it runs using dependencies.

```
isx

useEffect(() => {
    console.log("Component mounted");
}, []); // only once

useEffect(() => {
    console.log("Count changed");
}, [count]); // every time `count` changes
```

7. Conditional Rendering

```
jsx
{isLoggedIn ? <Logout /> : <Login />}

jsx

{user && Welcome back!}
```

8. List Rendering with `map()`

Always add a unique key prop!

9. useRef (Access DOM / Store Persistent Value)

10. Lifting State Up

Move state **from child to parent** to share it between multiple children.

11. Prop Drilling

Passing props down through many levels unnecessarily:

```
jsx

<Grandparent>
    <Parent>
        <Child someData={data} />
        </Parent>
        </Grandparent>
```

Avoid using Context API or Redux.

12. React Router (Navigation)



- Use `<Link to="/about">About</Link>` instead of `<a>`
- You need to `npm install react-router-dom`

Madam-Style Practice Questions

📝 Q1: What is the difference between props and state? (3 marks)

Answer:

Props	State
Passed from parent	Local to component
Read-only	Can be updated
Immutable	Mutable via `setState`

iv Q2: What does `useEffect(() => {}, [])` mean? *(2 marks)*

Answer:

Runs only once after the initial render (componentDidMount behavior).

📝 Q3: Convert this to list rendering: (3 marks)

Q4: Write a React component with a button that toggles "Show" / "Hide". (4 marks)

```
jsx

const Toggle = () => {
  const [show, setShow] = useState(true);
```



📝 Q5: What will this print? (2 marks)

```
jsx

const [num, setNum] = useState(0);
setNum(num + 1);
setNum(num + 1);
console.log(num);
```

Answer:

`0` — because `setNum` is batched, and both use the same `num = 0`. Correct way is `setNum(prev => prev + 1)`.

You're now done with **React Core + Hooks + Routing**.

✓ Ready for Redux Toolkit next?



yeah

Great — let's now master:

📘 Topic 8: Redux Toolkit (RTK)

This is your React app's **global state manager**, and much better than old-school Redux.

What is Redux Toolkit?

Redux Toolkit is the **official, recommended way** to use Redux today. It simplifies:

- Creating a global **store**
- Managing multiple states via slices
- Writing less boilerplate code

Redux Toolkit Key Concepts

Concept	Role
Store	Holds the global state
Slice	Contains a part of the state + its actions + reducers

Concept	Role
Dispatch	Sends an action to update the store
useSelector	Access state from the store
useDispatch	Send actions to the store

Project Setup

```
bash
npm install @reduxjs/toolkit react-redux
```

Step-by-Step: Counter Example

`features/counterSlice.js`

```
import { createSlice } from "@reduxjs/toolkit";

const counterSlice = createSlice({
    name: "counter",
    initialState: { value: 0 },
    reducers: {
        increment: (state) => { state.value += 1; },
        decrement: (state) => { state.value -= 1; },
        addByAmount: (state, action) => { state.value += action.payload; }
    }
});

export const { increment, decrement, addByAmount } = counterSlice.actions;
export default counterSlice.reducer;
```

app/store.js`

```
import { configureStore } from "@reduxjs/toolkit";
import counterReducer from "../features/counterSlice";

export const store = configureStore({
   reducer: { counter: counterReducer }
});
```

main.jsx`

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";
import { Provider } from "react-redux";
import { store } from "./app/store";

ReactDOM.createRoot(document.getElementById("root")).render(
```

```
<Provider store={store}>
     <App />
     </Provider>
);
```

`components/Counter.jsx`

- Important RTK Features
 - Mutating state is allowed (internally handled by Immer)
 - **Each slice** can handle only its state
 - Actions auto-created for every case reducer

Madam-Style Practice Questions

Q1: What is the purpose of a Redux slice? (2 marks)

Answer:

It defines a part of the state along with its actions and reducers in one file.

Q2: What library must be used to connect React to Redux? (1 mark)

Answer:

`react-redux`

Q3: Complete this reducer logic to subtract payload from `count`. (3 marks)

```
js
subtract: (state, action) => {
   state.value -= action.payload;
```

}

📝 Q4: Fill in the blanks for Redux Toolkit setup. (5 marks)

```
import { configureStore } from "@reduxjs/toolkit";
import ___ from "./features/mySlice";

export const store = configureStore({
   reducer: {
      mySlice: ___
   }
});
```

Answer:

Replace blanks with `myReducer` (or `mySlice.reducer`)

📝 Q5: Why can we directly mutate state in Redux Toolkit? (2 marks)

Answer:

Because Redux Toolkit uses the **Immer** library, which makes mutations safe and converts them into immutable updates under the hood.

Done with Redux Toolkit!

You're now ready to build full-scale apps that manage state cleanly.

Ready for the next topic: Mongoose + MongoDB (models + CRUD)?



yeh next

Perfect! Let's now jump into the backend database part:

Topic 9: Mongoose + MongoDB (Models + CRUD + Relations)

MongoDB stores your app's data; Mongoose helps you work with it in Node.js easily.

What is MongoDB?

- A **NoSQL** database that stores data as **documents** (like JSON).
- Collection = Table
- Document = Row

```
json
{
    "_id": "abc123",
    "name": "Ali",
```

```
"age": 25,
"skills": ["JS", "React"]
}
```

What is Mongoose?

- An ODM (Object Data Mapper) for MongoDB
- Helps:
 - Define schemas
 - o Validate data
 - Write CRUD operations easily

Setup محر

```
bash

npm install mongoose

js

// db.js
const mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/myapp")
    .then(() => console.log("Connected"))
    .catch(err => console.log("Error:", err));
```

Defining Schema and Model

```
const mongoose = require("mongoose");
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  age: { type: Number, min: 18 }
});
const User = mongoose.model("User", userSchema);
```

CRUD Operations

✓ Create

```
js

const user = new User({ name: "Ali", age: 25 });
await user.save();
```

Read

```
js

const allUsers = await User.find();
const oneUser = await User.findOne({ name: "Ali" });
```

Update

```
js
await User.findByIdAndUpdate(id, { age: 30 }, { new: true });
```

V Delete

```
js
await User.findByIdAndDelete(id);
```

Mongoose Relations

📌 One-to-One

```
const profileSchema = new mongoose.Schema({
  bio: String,
  user: { type: mongoose.Schema.Types.ObjectId, ref: "User" }
});
```

📌 One-to-Many

```
const postSchema = new mongoose.Schema({
   title: String,
   userId: { type: mongoose.Schema.Types.ObjectId, ref: "User" }
});
```

Many-to-Many

```
const studentSchema = new mongoose.Schema({
  name: String,
  courses: [{ type: mongoose.Schema.Types.ObjectId, ref: "Course" }]
});
```

Madam-Style Practice Questions

? Q1: What is the role of Mongoose? (2 marks)

Answer:

It acts as an ODM to connect MongoDB with Node.js and provides schema, validation, and CRUD support.

📝 Q2: Create a schema for `Book` with `title (String)`, `pages (Number)` (2 marks)

```
const bookSchema = new mongoose.Schema({
   title: String,
   pages: Number
});
```

📝 Q3: What does this do? (2 marks)

```
js
User.findOne({ name: "Sara" });
```

Answer:

Finds the first user in the collection whose name is "Sara".

📝 Q4: Add a new user named "Ali", age 22. (3 marks)

```
js
const user = new User({ name: "Ali", age: 22 });
await user.save();
```

Q5: Define a one-to-many relation between `User` and `Post`. (4 marks)

```
const postSchema = new mongoose.Schema({
  title: String,
  user: { type: mongoose.Schema.Types.ObjectId, ref: "User" }
});
```

You're now done with MongoDB + Mongoose basics + relations + CRUD.

Next up: Express (APIs, middleware, routers, controllers)?



yeah next

Great — let's move into:

Topic 10 & 11: Express APIs, Middleware, Routers, Controllers

This is how we build the **backend logic** for your web apps.

What is Express?

Express is a fast and flexible web framework built on Node.js. It simplifies:

- Creating routes (API endpoints)
- Handling requests and responses
- Applying middleware (e.g., JSON parsing, authentication)

% Project Setup

```
npm install express mongoose cors dotenv
npm install --save-dev nodemon
```

File structure:

```
project/
— index.js
— db/
— config.js
— models/
— User.js
— controllers/
— user.js
— routes/
— user.js
```

index.js (Main File)

```
import express from "express";
import connectDB from "./db/config.js";
import userRouter from "./routes/user.js";

const app = express();
app.use(express.json()); // Middleware to parse JSON
app.use("/api/users", userRouter); // Mount user routes

connectDB();
app.listen(3000, () => console.log("Server running on port 3000"));
```

db/config.js

```
import mongoose from "mongoose";

const connectDB = async () => {
   await mongoose.connect("mongodb://localhost:27017/myDB");
   console.log("MongoDB connected");
};
export default connectDB;
```

models/User.js

```
import mongoose from "mongoose";

const userSchema = new mongoose.Schema({
   name: String,
   email: String
});

const User = mongoose.model("User", userSchema);
export default User;
```

routes/user.js

```
import express from "express";
import { getUsers, createUser } from "../controllers/user.js";

const router = express.Router();
router.get("/", getUsers);
router.post("/", createUser);
export default router;
```

controllers/user.js

```
import User from "../models/User.js";

export const getUsers = async (req, res) => {
   const users = await User.find();
   res.status(200).json(users);
};

export const createUser = async (req, res) => {
   const { name, email } = req.body;
   const user = await User.create({ name, email });
   res.status(201).json(user);
};
```

Middleware

Global Middleware:

```
js
app.use(express.json()); // Automatically parses JSON
```

Custom Middleware Example:

```
app.use((req, res, next) => {
   console.log("Request received");
   next();
});
```

Madam-Style Practice Questions

📝 Q1: What is middleware in Express? (2 marks)

Answer:

A function that intercepts requests and runs before the final route handler. Used for logging, parsing, authentication, etc.

Q2: What does `express.json()` do? (2 marks)

Answer:

It parses incoming JSON payloads and adds it to `req.body`.

📝 Q3: Define an Express route that returns all users. (3 marks)

```
router.get("/", async (req, res) => {
  const users = await User.find();
  res.json(users);
});
```

2 Q4: Which file defines Express endpoints? (1 mark)

Answer:

`routes/user.js` (Router file)

Q5: Fill in the blanks to create an Express API:

```
import express from "express";
const app = express();
```

```
app.____(express.json());
app.____("/", handler);
```

Answer:

```
`use`, `get`
```

Ready for **Topic 12: Express** `req` and `res` objects (req.params, req.query, req.body, res.json, res.status)?



yeah next

Awesome — let's move into the heart of Express request handling:

Topic 12: Express `req` and `res` Objects

These two objects let you receive data from the client and send responses back.

- `req` Object (Incoming Request)
- `req.params`

Used for **URL path parameters**:

```
js

router.get("/user/:id", (req, res) => {
   console.log(req.params.id); // /user/123 → "123"
});
```

`req.query`

Used for **URL query strings**:

```
js

// /search?term=book
router.get("/search", (req, res) => {
   console.log(req.query.term); // "book"
});
```

\(\overline{\pi}\) `req.body`

Used in **POST/PUT/PATCH** requests:

```
router.post("/add", (req, res) => {
  const { name, email } = req.body;
});
```

Requires middleware: `app.use(express.json());`

`req.headers`

Contains all incoming headers:

```
router.get("/headers", (req, res) => {
   console.log(req.headers); // { host, user-agent, etc. }
});
```

`res` Object (Server Response)

`res.send()`

Sends plain text or HTML:

```
js
res.send("Hello");
```

`res.json()`

Sends JSON response:

```
js
res.json({ name: "Ali", age: 22 });
```

`res.status()`

Sets the **status code**:

```
res.status(404).send("Not found");
res.status(200).json({ success: true });
```

Examples Combined

```
router.post("/add", (req, res) => {
  const { name } = req.body;
  console.log(req.headers["user-agent"]);
  res.status(201).json({ message: `User ${name} added` });
});
```

Madam-Style Practice Questions

Q1: What is `req.params` used for? (2 marks)

Answer:

To get dynamic values from the route URL (like user ID in `/user/:id`).

Q2: What's the difference between `req.query` and `req.body`? (3 marks)

Answer:

`req.query`	`req.body`
From URL (` ?key=value`)	From request payload (POST/PUT)
Used in GET	Used in POST/PUT

Q3: Write an API that accepts a `name` in `req.body` and returns a welcome message. (4 marks)

```
router.post("/welcome", (req, res) => {
  const { name } = req.body;
  res.send(`Welcome, ${name}!`);
});
```

Q4: What status code means "Created"? (1 mark)

Answer: `201`

📝 Q5: Output of this code? (2 marks)

```
js
res.status(404).json({ error: "Not Found" });
```

Answer:

Status 404 and a JSON response: `{ error: "Not Found" }`

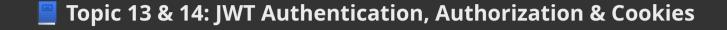
You're now comfortable handling Express API data flow.

Next up: JWT Authentication & Authorization + Cookies (final major backend topic)?



yeah

Perfect — let's wrap up the backend section with:



What is JWT?

JWT (JSON Web Token) is a secure way to **identify users** across API requests without storing session data on the server.

- When user logs in → server sends a **token**
- Token is stored (in **cookie** or **localStorage**)
- Token is attached in future requests to verify identity

✓ JWT Structure

A JWT has 3 parts:

```
php-template
<headler>.<payL0AD>.<sigNature>
```

Example:

```
eyJhbGciOiJI... (not human readable)
```

Setup کہ

```
bash
npm install jsonwebtoken
```

Generate Token (Login API)

```
js
import jwt from "jsonwebtoken";
```

```
const token = jwt.sign({ userId: user._id }, "secretKey", { expiresIn: "lh" });
res.cookie("access_token", token, {
  httpOnly: true,
  secure: true,
  sameSite: "None",
  maxAge: 3600000
});
```

Protect Route (Authorization)

```
const verifyToken = (req, res, next) => {
  const token = req.cookies.access_token;
  if (!token) return res.status(403).send("Not logged in");

  try {
    const decoded = jwt.verify(token, "secretKey");
    req.user = decoded;
    next();
  } catch (err) {
    res.status(401).send("Invalid token");
  }
};
```

Add to route:

```
router.get("/protected", verifyToken, (req, res) => {
  res.send("Secret Data");
});
```

Cookies in Express

Set a Cookie

```
res.cookie("name", "Ali", {
  httpOnly: true,
  secure: true,
  sameSite: "None",
  maxAge: 60000
});
```

Read a Cookie

```
import cookieParser from "cookie-parser";
app.use(cookieParser());
console.log(req.cookies.name); // "Ali"
```

Delete a Cookie (Logout)

```
res.clearCookie("access_token", {
  httpOnly: true,
  sameSite: "None",
  secure: true
});
res.send("Logged out");
```

Madam-Style Practice Questions

Q1: What is the purpose of JWT? (2 marks)

Answer:

To authenticate users across API calls without storing session state on the server.

2: What's the difference between Authentication and Authorization? (3 marks)

Auth	Authz
Verifies who you are	Verifies what you can access

📝 Q3: Write code to generate a JWT token. (3 marks)

```
js
const token = jwt.sign({ userId: user._id }, "secretKey", { expiresIn: "lh" });
```

📝 Q4: What does `http0nly: true` do in a cookie? (1 mark)

Answer:

Prevents JavaScript on the frontend from accessing the cookie (protects against XSS).

Q5: Middleware to verify JWT? (3 marks)

```
const verifyToken = (req, res, next) => {
  const token = req.cookies.access_token;
  if (!token) return res.sendStatus(403);

  try {
    const data = jwt.verify(token, "secretKey");
    req.user = data;
    next();
  } catch {
    res.sendStatus(401);
  }
};
```



You've now completed all major topics in your syllabus.

- **✓** Would you like me to now:
 - 1. Prepare a **revision day plan**
 - 2. Generate mock exam questions
 - 3. Build your **2-page cheat sheet template**
 - 4. All of the above?