Previously

- Javascript basics (scope, variables, functions)
- Asynchronous programming
- Nodejs installation and introduction
- Nodejs modules and it usage
- Express JS introduction and a demo application

Day 2: Deep Dive on HTTP Methods & Express.js

- Server Architecture
- HTTP Request Methods
- Different URL Part
- Express project setup
- Debugging Techniques

Section 1:
Server Architecture

Server Architecture

- Stateful Server
- Stateless Server

AMIT DHOJU

Stateful Server

- Keeps track of the client’s state between requests.
- The server stores session data (e.g., in memory or in a session store) and
- uses it to serve future requests.
- State is stored on the server (e.g., in memory, sessions, database).

- Example:
  - You log in → server keeps your session in memory → you browse pages without logging in again.
  - Online banking sessions — server remembers your login and account state.
  - Multiplayer game servers tracking player positions.

AMIT DHOJU

Stateful Server: Pros & Cons

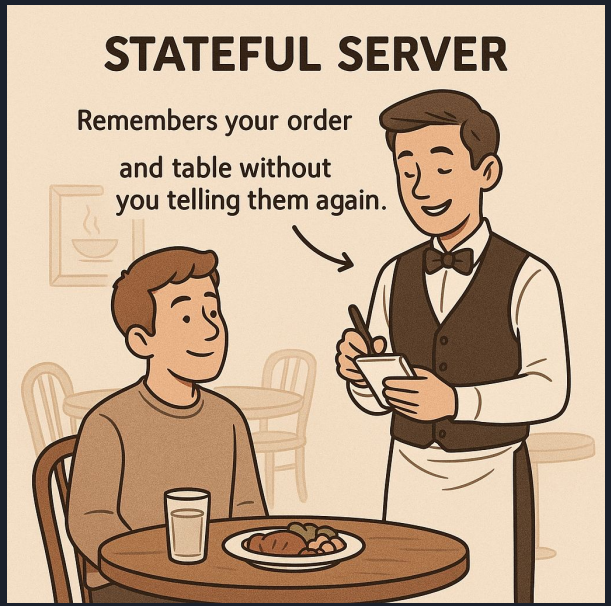
- Pros:
  - Good for long, interactive sessions (e.g., online banking, multiplayer games)
  - Easier to manage ongoing user interactions.
  - Less data sent with each request.

- Cons:
  - Harder to scale — need to share session state between servers (all servers need the same session data).
  - More memory usage on the server.

AMIT DHOJU

Stateful Server: Analogy

- Like a waiter who remembers your order and table without you telling them again.



AMIT DHOJU

Stateless Server

- Does not remember client state between requests. Each request is independent..
- **How it works**: Client must send all necessary context (e.g., authentication token, request data) with every request.
- State is stored on the client (e.g., in cookies, tokens)

- Example:
  - REST APIs using JWT tokens for authentication.
  - REST API where you send your authentication token with every request.

AMIT DHOJU

Stateful Stateless: Pros & Cons

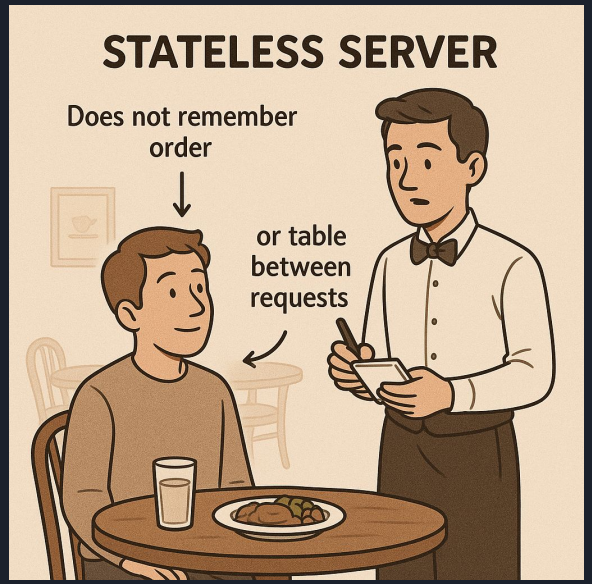
- Pros:
  - Easy to scale horizontally (add more servers without syncing state).
  - Simpler architecture.
  - More fault-tolerant.

- Cons:
  - Client sends more data with each request.
  - Can be less efficient for continuous sessions.

AMIT DHOJU

Stateful Stateless: Analogy

- Like ordering food at a fast-food counter — every order is independent, you must give your details every time..



AMIT DHOJU

Server API Style

- SOAP (Simple Object Access Protocol)
- REST (Representational State Transfer)
- GraphQL

SOAP API

- **Principle**: XML-based protocol for exchanging structured information.
- **Uses**: Often in enterprise applications (banking, government systems).
- More strict structure, with a WSDL (Web Services Description Language).

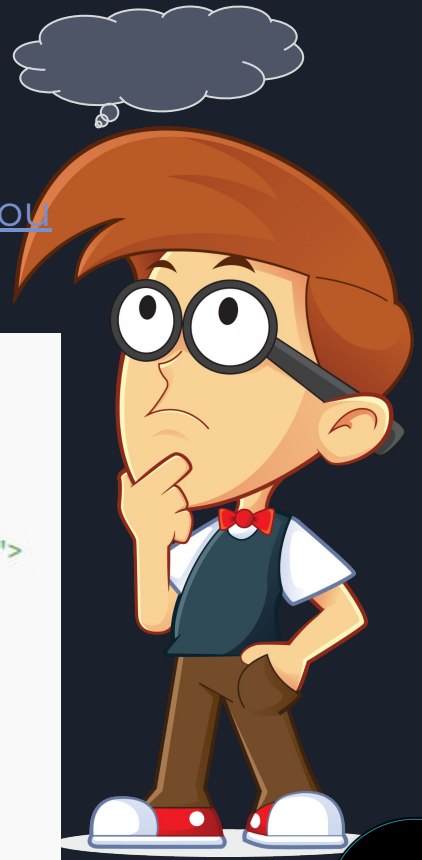
- **Pros**:
  - Strong standards and strict structure.
  - Built-in error handling and security
- **Cons**:
  - Verbose XML format.
  - More overhead compared to REST/GraphQL.

AMIT DHOJU

SOAP API : Example

- http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso?WSDL

```
fetch('http://localhost:8000/wsdl', {
  method: 'POST',
  headers: { 'Content-Type': 'text/xml' },
  body: `
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <GetUser/>
  </soapenv:Body>
</soapenv:Envelope>`
})
  .then(res => res.text())
  .then(console.log);
```



AMIT DHOJU

REST API

- **Principle:** Represent resources via unique URLs, use standard HTTP methods.
- **Methods:** GET, POST, PUT, DELETE, PATCH.
- **Data Format:** JSON (most common), XML possible.

- **Pros:**
  - Simple, widely adopted.
  - Stateless by default.
- **Cons**:
  - Can result in over-fetching or under-fetching data.

REST API : Example

- https://jsonplaceholder.typicode.com/

```
fetch('http://localhost:3000/users')
  .then(res => res.json())
  .then(data => console.log(data));
```

AMIT DHOJU

GraphQL API

- **Principle:** Client specifies exactly what data it needs.
- **Endpoints:** Usually a single /graphql endpoint.
- **Data Format:** JSON
- Reduces over-fetching and under-fetching.
- **Pros:**
  - No over-fetching — returns exactly what you request.
  - Strongly typed schema.
- **Cons**:
  - More complex setup.
  - Caching can be trickier than REST.

AMIT DHOJU

GraphQL API : Example

- https://countries.trevorblades.com/

```
{
  countries {
    capital
    name
  }

  continents{
    name
code
  }
}
```

AMIT DHOJU

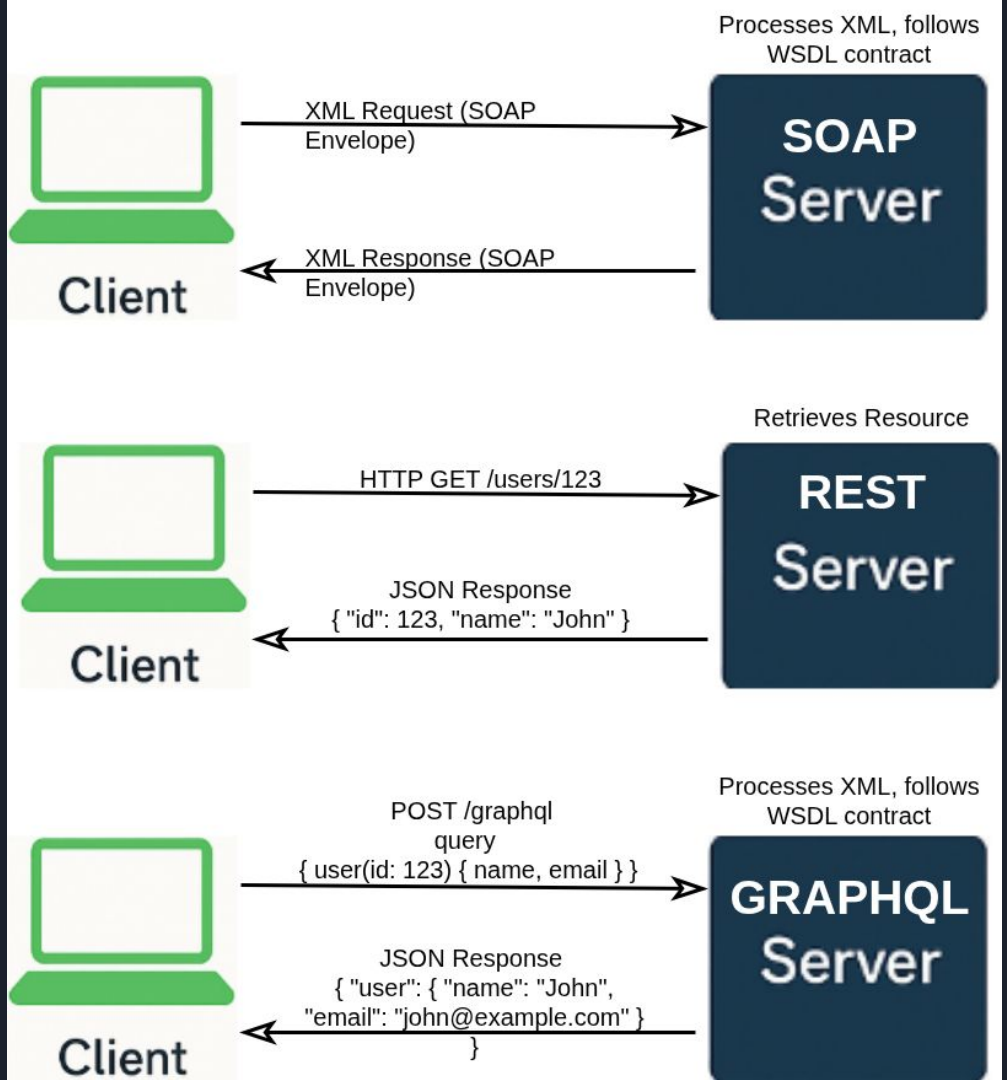
SOAP vs REST vs GRAPHQL

- **SOAP –** Like a government post office: strict rules, fixed format, reliable but heavy.

- **REST** – Like ordering from a restaurant menu: simple, predefined options, sometimes extra you don’t need (eg:fish with fries).

- **GraphQL –** Like a custom smoothie bar: you ask for exactly what you want in one go (eg: content and flavour).

AMIT DHOJU


Processes XML, follows WSDL contract
XML Request (SOAP Envelope)
SOAP Server
XML Response (SOAP Envelope)
Client
Retrieves Resource
HTTP GET /users/123
REST Server
JSON Response
{ "id": 123, "name": "John" }
Client
Processes XML, follows WSDL contract
POST /graphql
query
{ user(id: 123) { name, email } }
GRAPHQL Server
JSON Response
{ "user": { "name": "John",
"email": "john@example.com" }
}
Client

AMIT DHOJU

Any Questions??

AMIT DHOJU

Section 2: HTTP Request Methods

HTTP Request Methods

- GET - Retrieve Data
- POST - Create New Data
- PUT - Update Entire Resource
- PATCH - Partial Updates
- DELETE - Remove Data

AMIT DHOJU

HTTP GET

- Used to request data from the server.
- Does not change data on the server.
- Typically used to fetch resources.

```
app.get('/users', (req, res) => {
  // Pretend we get users from database
  const users = [{ id: 1, name: 'Amit' }, { id: 2, name: 'John' }];
  res.json(users);
});
```

AMIT DHOJU

HTTP POST

- Used to send data to the server to create a new resource.
- Data sent usually in request body (JSON, form data, etc.).

```
app.post('/users', (req, res) => {
  const newUser = req.body;  // Assuming middleware like express.json() is used
  // Save newUser to database (mock)
  newUser.id = 3; // Example ID assigned
  res.status(201).json(newUser);
});
```

AMIT DHOJU

HTTP PUT

- Replaces the entire resource with new data.
- Requires sending the full updated object.

```
app.put('/users/:id', (req, res) => {
  const userId = req.params.id;
  const updatedUser = req.body;
  updatedUser.id = userId;
  // Replace user in database (mock)
  res.json(updatedUser);
});
```

AMIT DHOJU

HTTP PATCH

- Updates only specified fields of a resource.
- Does not require sending the full object.

```
app.patch('/users/:id', (req, res) => {
  const userId = req.params.id;
  const partialUpdate = req.body;  // e.g., { name: 'New Name' }
  // Update user partially in database (mock)
  const updatedUser = { id: userId, name: partialUpdate.name || 'Existing Name' };
  res.json(updatedUser);
});
```

AMIT DHOJU

HTTP DELETE

- Deletes the resource identified by the URL.

```
app.delete('/users/:id', (req, res) => {
  const userId = req.params.id;
  // Delete user from database (mock)
  res.status(204).send();  // 204 No Content means successful deletion
});
```

AMIT DHOJU

HTTP SUMMARY

Method	Action	Idempotent	Request Body	Example Usage
GET	Retrieve data	✅	❌	Get user list or detail
POST	Create new data	❌	✅	Add a new user
PUT	Update whole data	✅	✅	Replace user with new data
PATCH	Partial update	❌	✅	Change only some user fields
DELETE	Delete resource	✅	❌	Remove a user

AMIT DHOJU

Any Questions??

La yo chai
sodhnu parla

AMIT DHOJU

Section 3: Different URL Part

URL Parts

- Query Parameters (?key=value)
- Route Parameters (:id)
- Hash Fragment (#something)

```
https://example.com/users/123?sort=asc&page=2#contact
          |       |       |            |     |
      protocol   path   route param   query  fragment
```

Query Parameters (?key=value)

- **Syntax**: ?param1=value1&param2=value2
- **Purpose**: Send extra information to the server, usually for filtering, searching, or pagination.
- **Example:** https://example.com/products?category=boot&color=red
- **Notes**:
  - Appear after ?.
  - Multiple parameters are separated by &.
  - Order doesn't matter for most APIs (but can matter if the backend is coded that way).

AMIT DHOJU

Route Parameters (:id)

- **Syntax (in backend frameworks)** : /users/:id
- **Purpose**: Represent dynamic parts of the path.
- **Example:** https://example.com/users/123
- Notes:
  - Here **123** is the id of the user.
  - Often used for resource identification.
  - The colon : is only in the route definition (e.g., Express.js), not in the actual URL.

AMIT DHOJU

Hash Fragment (#something)

- **Syntax** : #section-name
- **Purpose**:
  - On the web: Scrolls to a specific part of the page (anchor).
  - In SPAs (Single Page Apps): Can be used for client-side routing.
- **Example:** https://example.com/about#team
- Notes:
  - Loads about page and jumps to the team section..
  - Never sent to the server — handled entirely by the browser.
  - Common in old hash-based routing: /#/home.

AMIT DHOJU

Any Questions??

La yo chai
sodhnu parla

AMIT DHOJU

Section 4:
Express project setup

Express project setup

- Manual setup : using npm initialization
- Fast setup : using express generator

Project setup: manual setup

- Create all necessary files and folder manually

1. Initialize project with npm: `***npm init -y***`
2. Install Express: npm install express
3. Create your project structure manually:
    a. /project-root
    b. /routes
    c. /controllers
    d. /middleware
    e. /views
    f. app.js

AMIT DHOJU

Project setup: manual setup

4. Use nodemon for easier development:
   `***npm install --save-dev nodemon***`
5. Add a script in package.json:
   "scripts": {
   "dev": "nodemon app.js"
   }
6. Run development server: `***npm run dev***`

AMIT DHOJU

Project setup: Fast setup

- Using Express Generator for full setup

1. Run express generator: `***npx express-generator***`
2. This scaffolds the project with routes, views, public folder, and basic structure.
3. Install dependencies: `***npm install***`
4. Use nodemon (install if needed):
   `***npm install --save-dev nodemon***`

AMIT DHOJU

Project setup: Fast setup

5. Add dev script to package.json

```
"scripts": {
  "dev": "nodemon ./bin/www"
}
```

6. Run development server: `` `npm run dev` ``

AMIT DHOJU

Any Questions??

La yo chai
sodhnu parla

AMIT DHOJU

Section 5: Express project directory description

```
/project-root
|
├── /routes
|     └── index.js
|
├── /controllers
|     └── userController.js
|
├── /middleware
|     └── logger.js
|
├── /views
|     └── index.pug
|
├── /public
|     ├── /css
|     └── /js
|
├── app.js
├── package.json
└── README.md
```

Express project folders

- **/routes**
  - **Purpose**: Define the app routes and map routes to controller functions.
  - Keeps routing logic organized and separate from business logic.
  - **Example:** /routes/index.js

AMIT DHOJU

Express project folders

- **/controllers**
  - Optional(better if used)
  - **Purpose**: Contains the business logic and handlers for routes.
  - Keeps route files clean and focused on routing only.
  - Can handle DB calls, validations, or other logic.
  - **Example:** /controllers/userController.js

Express project folders

- **/middleware**
  - **Purpose**: Contains custom middleware functions to handle repeated tasks
  - Middleware can process requests before they reach routes/controllers.
  - **Example:** /middleware/logger.js

Express project folders

- **/views**
  - **Purpose**: Stores template files (if using a view engine like Pug, EJS)..
  - For server-side rendering of HTML pages..
  - **Example:** /views/index.pug

```
app.set('view engine', 'pug');
app.set('views', path.join(__dirname, 'views'));
```

AMIT DHOJU

Express project folders

- **/public**
  - **Purpose**: Static files like CSS, JavaScript, images, fonts.
  - Express serves these directly to the client.
  - **Example:** /public/{*}.{css | js}

Express project folders

- **app.js**
  - **Purpose**: Main entry point of the app..
  - Sets up Express app, middleware, routes, and error handlers..
  - **Example:** app.js

Section 6: Debugging Techniques for Node.js Express App

Debugging technique

- try-catch
- console functions
- debugger usage
- node inspector
- postman

Try-Catch Blocks

- Wrap your synchronous code or async functions with try-catch to catch runtime errors.
- Use in middleware or route handlers to gracefully handle exceptions.

```
app.get('/user/:id', async (req, res) => {
  try {
    const user = await getUserById(req.params.id);
    res.json(user);
  } catch (error) {
    res.status(500).json({ message: 'Server Error', error: error.message });
  }
});
```

AMIT DHOJU

Console Functions

- Use console.log(), console.error(), console.warn(), and console.table() for quick insights.
- Add meaningful messages and variable outputs for tracing app flow.

```
console.log('User ID:', req.params.id);
console.error('Database connection failed');
console.table([{ id: 1, name: 'Amit' }, { id: 2, name: 'John' }]);
```

AMIT DHOJU

Debugger Usage

- Use the built-in Node.js debugger.
- Start your app with
  - **node inspect app.js**
  - or **node --inspect app.js**
- Insert debugger; statement in your code where you want to pause execution.
- Then use Chrome DevTools or VSCode debugger to step through your code.

```
app.get('/data', (req, res) => {
  debugger;  // Execution will pause here if debugger is attached
  res.send('Debug me');
});
```

AMIT DHOJU

Node Inspector

- Start your app with the inspector enabled
  - **node --inspect-brk app.js**
- --inspect-brk pauses execution on the first line.
- Open Chrome browser and navigate to chrome://inspect to connect and debug.
- Supports breakpoints, call stack inspection, variable watching.

AMIT DHOJU

Testing with Postman

- Postman is a GUI tool to test API endpoints.
- Helps verify request methods, payloads, headers, and responses.
- Useful to isolate backend issues by manually triggering routes.
- Can save test collections for repeated debugging or regression tests.

AMIT DHOJU

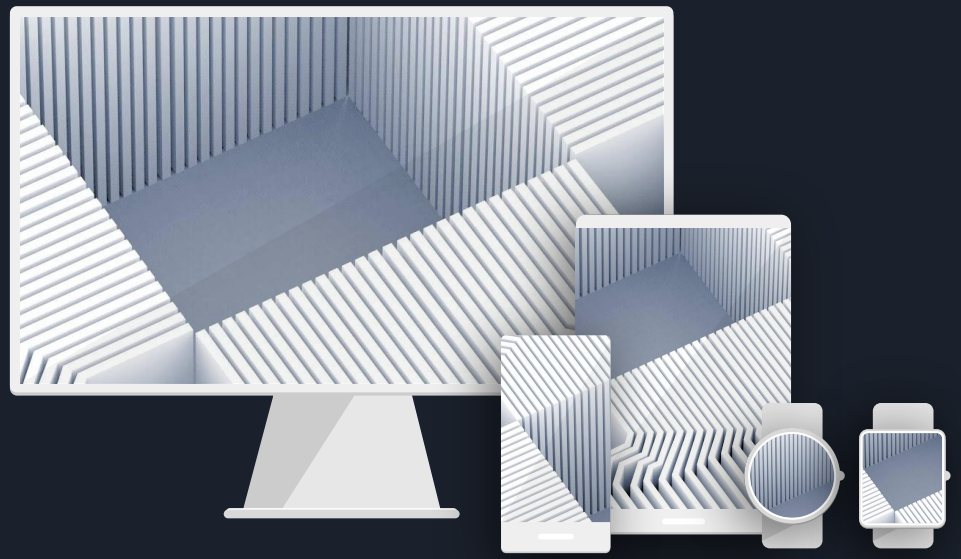
Any Questions??

La yo chai
sodhnu parla

AMIT DHOJU

Hands-On Exercise:

Express app: create some more routes



AMIT DHOJU

Section 7: System Design Basics

What is System Design?

- process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements.

- how to build systems that work well under load, are maintainable, scalable, and fault tolerant.

AMIT DHOJU

Key Concepts

- **Scalability**: Ability of the system to handle increasing load by adding resources.
- **Vertical scaling**: Adding more power (CPU, RAM) to a single machine.
- **Horizontal scaling**: Adding more machines to distribute load.
- **Availability**: The system's ability to remain operational and accessible.

AMIT DHOJU

Key Concepts

- **Reliability**: The system works correctly even when some components fail.
- **Latency**: Time taken to respond to a request.
- **Throughput**: Number of requests handled per unit time.

AMIT DHOJU

Core Components of a System

- **Client**
- **API Layer / Load Balancer**
- **Application Servers.**
- **Database**
- **Cache**
- **Message Queues**
- **CDN: Content Delivery Network**

AMIT DHOJU

Core Components of a System

- Client: User or service that interacts with the system.
- API Layer / Load Balancer: Distributes requests, balances traffic.
- Application Servers: Handle business logic.
- Database: Stores data (SQL or NoSQL).
- Cache: Temporary fast storage for frequently accessed data.
- Message Queues: Asynchronous communication for decoupling components.
- CDN: Content Delivery Network to serve static files quickly.

AMIT DHOJU

Design Principles

- **Separation of Concerns**: Divide system into distinct features/modules.
- **Loose Coupling**: Components should be independent to reduce impact of change.
- **High Cohesion**: Related functionality grouped together.
- **Fail Fast**: Detect and handle errors quickly.
- **Idempotency**: Ensure operations can be safely repeated.

AMIT DHOJU

Common Design Patterns

- **Client-Server Architecture**
- **Microservices Architecture**: Splitting system into small independent services.
- **Event-Driven Architecture**: Components react to events/messages.
- **Load Balancing**: Distribute requests to multiple servers.

- **Caching Strategies**: Cache Aside, Write Through, Write Back.

- **Database Sharding**: Splitting data horizontally to improve performance.

AMIT DHOJU