Node.js Express Architecture – Routes, Controllers & Models (E-commerce Example)

---

0. What is an API?

An **API (Application Programming Interface)** is a set of rules that allows two systems to communicate with each other.

In a web application: - The **client** (browser, mobile app, frontend) sends a request - The **server** (Node.js) processes the request - The **server** sends a response back to the client

APIs act as a **bridge** between client and server.

0.1 What is a REST API?

REST stands for **Representational State Transfer**.

A REST API is an API that follows standard HTTP rules and uses: - URLs to identify resources - HTTP methods to perform actions - JSON to exchange data

Key principles of REST API: - Stateless (each request is independent) - Client-server separation - Uses standard HTTP methods - Lightweight and fast

REST APIs are easy to build, easy to scale, and easy to consume.

0.2 What is a SOAP API?

SOAP stands for **Simple Object Access Protocol**.

SOAP APIs: - Use XML format - Are very strict and standardized - Require more configuration - Are heavier compared to REST

SOAP is mostly used in **legacy and enterprise systems** like banking.

0.3 REST API vs SOAP API (Difference)

Feature	REST API	SOAP API
Data Format	JSON	XML
Speed	Fast	Slow

Feature	REST API	SOAP API
Flexibility	High	Low
Ease of Use	Easy	Complex
Modern Apps	✅Yes	❌Rare

0.4 Why We Use REST APIs in This Application?

We use REST APIs because: - Node.js works naturally with JSON - REST APIs are lightweight - Easy integration with frontend (React, mobile apps) - Better performance - Industry standard for modern applications

This is why REST APIs are used in e-commerce, social media, and SaaS products.

0.5 What is JSON and Why Do We Need It?

JSON (JavaScript Object Notation) is a lightweight data format used to exchange data between client and server.

Example JSON:

```
{
  "name": "Ravi",
  "email": "ravi@gmail.com",
  "role": "customer"
}
```

Why JSON is important: - Easy to read and write - Native to JavaScript - Supported by all languages - Smaller size compared to XML

JSON is the **default language of REST APIs**.

0.6 Client → Server → Database Flow (Simple Explanation)

1. Client sends HTTP request (GET / POST)
2. Server receives request in route
3. Middleware processes request
4. Controller executes business logic
5. Database stores or fetches data
6. Server sends response to client

Example: - Client sends POST /users - Server validates data - Database saves user - Server returns success response

1. What is Node.js?

Node.js is a **JavaScript runtime environment** that allows us to run JavaScript **outside the browser**.

It is built on Google’s **V8 engine** and is designed for: - Building fast backend servers - Handling multiple requests efficiently - Creating REST APIs

Node.js uses a **non-blocking, event-driven architecture**, which makes it highly suitable for real-time and scalable applications like e-commerce, chat apps, dashboards, etc.

---

2. What is Express.js?

Express.js is a **lightweight framework built on top of Node.js**.

Express helps us: - Create APIs easily - Handle HTTP methods (GET, POST, PUT, DELETE) - Organize application structure - Add middleware support

Without Express, writing backend logic in Node.js becomes complex and repetitive.

---

3. What is Express Router?

Express Router is a **mini Express application** that allows us to define routes in a **modular and organized way**.

A router helps us group multiple HTTP methods (GET, POST, PUT, PATCH, DELETE) under the **same base path**.

Instead of writing all routes in `server.js`, we create routers for each feature (users, products, orders).

This makes the application: - Clean - Scalable - Easy to maintain - Easy to explain in interviews

---

4. Why Do We Need Express Router?

In real applications, a single resource (like users) needs **multiple operations**: - Get users - Create user - Update user - Delete user

Using Express Router, we can define all these operations **using the same router object**.

This avoids code duplication and improves readability.

---

5. How Router Handles Multiple HTTP Methods (Very Important)

When we create a router, we can attach **multiple HTTP methods** to it.

Example:

```
router.get("/", getUsers);
router.post("/", createUser);
router.put("/:id", updateUser);
router.patch("/:id", partialUpdateUser);
router.delete("/:id", deleteUser);
```

All these routes: - Use the same router - Share the same base path (`/users`) - Perform different actions based on HTTP method

---

Why This is Powerful?

Because: - Same URL can behave differently for different HTTP methods - REST API design becomes clean - Easy to explain in interviews as **resource-based routing**

Interview Explanation (You Can Say This)

> Express Router allows us to group all CRUD operations of a resource under a single router. Using `router.get`, `router.post`, `router.put`, `router.patch`, and `router.delete`, we can handle different operations on the same endpoint in a clean and scalable way.

---

6. Including Router as Middleware in Server

Router is added to the main app using `app.use()`.

```
app.use("/users", userRouter);
```

This means: - `/users` is the base path - All router methods work under this path

---

7. Why Router is Mandatory in Large Applications

Without router: - All code goes into one file - Difficult to maintain - Not scalable

With router: - Feature-based separation - Team-friendly development - Industry best practice

---

Express Router helps us **divide routes into separate files** instead of writing everything in one file.

Why Router is important: - Keeps code clean and organized - Makes large applications manageable - Separates concerns (users, orders, products) - Helps in team development

In real projects, we never write all routes in `server.js`.

---

4. E-commerce Application Structure

```
project-root
│
├── server.js
├── config
│   └── db.js
├── routes
│   ├── user.routes.js
│   ├── product.routes.js
│   ├── order.routes.js
│   └── customer.routes.js
├── controllers
│   ├── user.controller.js
│   ├── product.controller.js
│   ├── order.controller.js
│   └── customer.controller.js
├── models
│   ├── user.model.js
│   ├── product.model.js
│   ├── order.model.js
│   └── customer.model.js
├── middleware
│   └── auth.middleware.js
├── .env
└── package.json
```

This structure is **industry-standard** and commonly used in interviews and real projects.

---

5. What is a Route?

A route defines **which URL should execute which logic**.

Example: - `/users` → user logic - `/products` → product logic - `/orders` → order logic

Routes only handle **request & response mapping**, not business logic.

---

6. Creating Routes Using Express Router

User Routes (`routes/user.routes.js`)

```
const express = require("express");
const router = express.Router();
const userController = require("../controllers/user.controller");

router.get("/", userController.getUsers);
router.post("/", userController.createUser);

module.exports = router;
```

Here: - `router.get()` maps GET request - Logic is redirected to controller

---

7. Including Routes in Server File

`server.js`

```
const express = require("express");
const app = express();

app.use(express.json());

app.use("/users", require("./routes/user.routes"));
app.use("/products", require("./routes/product.routes"));
app.use("/orders", require("./routes/order.routes"));
app.use("/customers", require("./routes/customer.routes"));

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

This is how **router middleware** is registered in Express.

---

8. What is a Controller?

Controllers contain **business logic** of the application.

Why controllers are important: - Keeps routes clean - Separates request handling from logic - Easy to test and maintain - Reusable logic

Routes should only redirect to controllers.

---

9. Creating Controllers

User Controller (`controllers/user.controller.js`)

```
exports.getUsers = (req, res) => {
  res.json({ message: "Fetching users" });
};

exports.createUser = (req, res) => {
  res.json({ message: "User created" });
};
```

Controllers receive: - `req` → request data - `res` → response object

---

10. What is Middleware?

Middleware is a function that **executes before the final request handler**.

Middleware is used for: - Authentication - Authorization - Logging - Validation

---

11. Middleware Example

```
module.exports = (req, res, next) => {
  console.log("Middleware executed");
  next();
};
```

Middleware must call `next()` to continue the request flow.

---

12. MongoDB Connection Types in Node.js (Very Important)

Before working with models, it is very important to understand **how Node.js connects to MongoDB** and the **different connection types**.

12.1 What is `mongod`?

`mongod` is the **MongoDB database server process**.

In simple English: - `mongod` is the **actual database engine** - It runs in the background - It stores data on disk

Key points: - Without `mongod`, MongoDB does not work - It listens on port `27017` - It must be running before any Node.js app connects

You usually start it automatically (Windows service) or manually.

12.2 What is MongoDB Client (`MongoClient`)?

`MongoClient` is the **official MongoDB driver for Node.js**.

It is used to: - Connect Node.js to MongoDB - Run queries directly on collections - Work without schemas

This is called a **native MongoDB connection**.

Example: MongoClient Connection (Without Mongoose)

```
const { MongoClient } = require("mongodb");

const url = "mongodb://localhost:27017";
const client = new MongoClient(url);

async function connectDB() {
  await client.connect();
  const db = client.db("ecommerce");
  const users = db.collection("users");

  const data = await users.find().toArray();
  console.log(data);
}

connectDB();
```

When to use MongoClient: - Small applications - Microservices - When you want full control

---

12.3 What is Mongoose?

Mongoose is an **ODM (Object Data Modeling) library** built on top of MongoDB.

Mongoose provides: - Schemas - Models - Validation - Middleware (hooks)

It makes MongoDB more **structured and manageable**.

Example: Mongoose Connection

```
const mongoose = require("mongoose");

mongoose.connect(process.env.MONGO_URL)
  .then(() => console.log("MongoDB connected via Mongoose"));
```

When to use Mongoose: - Medium to large applications - Structured data - Team projects

---

12.4 MongoClient vs Mongoose (Comparison)

Feature	MongoClient	Mongoose
Schema	❌No	✅Yes
Validation	❌No	✅Yes
Structure	Low	High
Learning Curve	Easy	Moderate
Best For	Small apps	Large apps

---

13. What is a Model?

A model represents **database structure**.

Models define: - Fields - Data types - Validation rules

In MongoDB, models are created using **Mongoose**.

---

13. Creating MongoDB Models (Mongoose)

User Model (`models/user.model.js`)

```
const mongoose = require("mongoose");

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

module.exports = mongoose.model("User", userSchema);
```

---

14. MongoDB Connection File

`config/db.js`

```
const mongoose = require("mongoose");

const connectDB = async () => {
  await mongoose.connect(process.env.MONGO_URL);
  console.log("MongoDB connected");
};

module.exports = connectDB;
```

---

15. Using Environment Variables (`.env`)

Environment variables are used to store **configuration values** that should not be hard-coded in the application.

Examples: - Database URL - Port number - API keys - Secrets

Why environment variables are important: - Improves security - Different values for development & production - Prevents exposing sensitive data

---

How to Access Environment Variables in Node.js

1. Install dotenv package:

```
npm install dotenv
```

2. Load environment variables in `server.js`:

```
require("dotenv").config();
```

3. Access variables using `process.env`:

```
const port = process.env.PORT;
const mongoUrl = process.env.MONGO_URL;
```

Node.js provides `process.env` globally to access environment variables.

```
PORT=3000
MONGO_URL=mongodb://localhost:27017/ecommerce
```

Why `.env` is important: - Keeps secrets secure - Environment-specific configuration - Best practice for production

---

16. Final Application Flow (Very Important)

```
Request → Route → Middleware → Controller → Model → Database
```

Understanding this flow means you understand **Node.js backend completely**.

---

17. Interview & Teaching Perspective

After understanding this document, you should be able to: - Design Node.js project structure - Explain routes vs controllers - Build scalable backend apps - Answer interview questions confidently - Teach this concept to others

---

This document is written to help you **build, explain, and teach Node.js applications confidently**.