Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

ADVANCED FULL STACK DEVELOPMENT (WEEKS 25-30)

STRUCTURING A FULL STACK PROJECT

CHAPTER 1: INTRODUCTION TO FULL STACK PROJECT STRUCTURE

1.1 What is a Full Stack Project?

A **Full Stack project** consists of both **frontend (client-side) and backend (server-side) components,** often with a **database** for persistent storage.

- **Why Proper Structure is Important?**

✓ Improves **code organization** and maintainability.
✓ Enhances **scalability** for future growth.
✓ Ensures **separation of concerns** (Frontend, Backend, Database).

- **Common Full Stack Technologies**

Layer	Technology Example
Frontend	React.js, Vue.js, Angular
Backend	Node.js (Express), Django, Spring Boot

Database	MongoDB, MySQL, PostgreSQL

CHAPTER 2: PROJECT FOLDER STRUCTURE FOR FULL STACK APPLICATIONS

2.1 Standard Full Stack Project Structure

📌 Example Folder Structure:

```
fullstack-app/
│── backend/        # Backend (Express, Django, etc.)
│   ├── models/     # Database Models
│   ├── routes/     # API Routes
│   ├── controllers/  # Business Logic
│   ├── middleware/   # Middleware (Auth, Validation)
│   ├── config/     # Database & App Configuration
│   ├── index.js    # Main Server Entry
│
│── frontend/       # Frontend (React, Vue, Angular)
│   ├── src/
│   │   ├── components/  # Reusable UI Components
│   │   ├── pages/     # Page Components (Home, Dashboard)
│   │   ├── services/   # API Requests (Axios, Fetch)
```

```
│   │   ├──── App.js      # Main Component
│   │   ├──── index.js    # React/Vue Entry Point
│   ├──── public/         # Static Assets
│
├── database/         # Database Initialization (Optional)
├── .env              # Environment Variables
├── package.json      # Backend Dependencies
├── frontend/package.json # Frontend Dependencies
├── README.md         # Project Documentation
```

✓ **Separates backend and frontend** into different folders.
✓ **Organizes reusable code** (models, controllers, services).
✓ **Environment variables (.env)** store sensitive information.

CHAPTER 3: BACKEND STRUCTURE (NODE.JS & EXPRESS.JS EXAMPLE)

3.1 Setting Up the Backend Server

📌 **Initialize a Node.js project & install dependencies:**

```
mkdir backend
cd backend
npm init -y
npm install express mongoose dotenv cors bcryptjs jsonwebtoken
```

✓ express → Handles HTTP requests.
✓ mongoose → Connects to MongoDB.
✓ dotenv → Loads environment variables.
✓ cors → Allows frontend to access backend.

3.2 Setting Up index.js (Backend Entry Point)

📌 **Create backend/index.js**

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

const mongoose = require("mongoose");

const dotenv = require("dotenv");


dotenv.config();

const app = express();


app.use(express.json());

app.use(require("cors")());


mongoose.connect(process.env.MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true })

  .then(() => console.log("MongoDB Connected"))

  .catch(err => console.error("MongoDB Connection Error:", err));
```

```
app.get("/", (req, res) => res.send("API Running..."));


const PORT = process.env.PORT || 5000;

app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

✓ **Connects to MongoDB** and starts Express server.

3.3 Organizing Models, Routes, and Controllers

📌 **Example: User Model (models/User.js)**

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


const UserSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true },
  password: String
});


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

✓ Defines a **User Schema** for MongoDB.

📌 **Example: User Controller (controllers/userController.js)**

```
const User = require("../models/User");
```

```
exports.getUsers = async (req, res) => {
   const users = await User.find();
   res.json(users);
};
```

✓ **Handles user-related operations** (retrieving users).

📌 **Example: User Routes (routes/userRoutes.js)**

```
const express = require("express");
const { getUsers } = require("../controllers/userController");
const router = express.Router();

router.get("/", getUsers);

module.exports = router;
```

✓ Defines **routes separately** for modularity.

📌 **Integrate Routes in index.js**

```
const userRoutes = require("./routes/userRoutes");
app.use("/api/users", userRoutes);
```

✓ Now API is available at: http://localhost:5000/api/users.

CHAPTER 4: FRONTEND STRUCTURE (REACT.JS EXAMPLE)

4.1 Setting Up React Frontend

📌 Initialize React App:

npx create-react-app frontend

cd frontend

npm install axios react-router-dom

✓ axios → Handles API requests.
✓ react-router-dom → Enables page navigation.

4.2 Organizing React Components

📌 Example Folder Structure:

frontend/src/

```
├── components/
│   ├── Navbar.js
│   ├── UserList.js
├── pages/
│   ├── Home.js
│   ├── Dashboard.js
├── services/
│   ├── api.js
├── App.js
├── index.js
```

✓ components/ → Reusable UI components.
✓ pages/ → Full pages like Home, Dashboard.
✓ services/ → Handles API requests.

4.3 Setting Up API Service

📌 Example: Fetching Users from Backend (services/api.js)

```
import axios from "axios";

const API_URL = "http://localhost:5000/api/users";

export const fetchUsers = async () => {
  const response = await axios.get(API_URL);
  return response.data;
};
```

✓ fetchUsers() fetches **users from backend API**.

📌 Using API Data in a Component (components/UserList.js)

```
import React, { useEffect, useState } from "react";
import { fetchUsers } from "../services/api";

function UserList() {
  const [users, setUsers] = useState([]);
```

```
  useEffect(() => {
    fetchUsers().then(data => setUsers(data));
  }, []);

  return (
    <ul>
      {users.map(user => <li key={user.id}>{user.name}</li>)}
    </ul>
  );
}

export default UserList;
```

✓ Displays **list of users** from API.

CHAPTER 5: CONNECTING FRONTEND & BACKEND

5.1 Setting Up Proxy for API Calls

📌 **Modify frontend/package.json**

```
"proxy": "http://localhost:5000"
```

✓ Allows **frontend to communicate with backend** without CORS issues.

5.2 Displaying API Data in React UI

📌 Modify App.js to Include UserList Component

```
import React from "react";
import UserList from "./components/UserList";

function App() {
  return (
    <div>
      <h1>User List</h1>
      <UserList />
    </div>
  );
}

export default App;
```

✓ Now the frontend **fetches user data from backend API**.

Case Study: How Airbnb Structures Full Stack Projects

Challenges Faced by Airbnb

✓ Handling **millions of user bookings**.
✓ Managing **frontend-backend communication efficiently**.
✓ Ensuring **high-speed performance for users worldwide**.

Solutions Implemented

✓ Used **React for UI, Node.js/Express for API**.
✓ Optimized **database queries for faster responses**.
✓ Modularized **frontend and backend for scalability**.

🔹 **Key Takeaways from Airbnb's Strategy:**
✓ **Modular folder structures improve scalability.**
✓ **API services enhance frontend-backend communication.**
✓ **Using best practices ensures a maintainable Full Stack project.**

📝 **Exercise**

✅ Implement **CRUD operations** for a "Tasks" API.
✅ Connect **React frontend** to backend using Axios.
✅ Secure API using **JWT authentication**.
✅ Deploy Full Stack project to **Heroku & Vercel**.

Conclusion

✓ **Full Stack projects should follow a structured folder hierarchy.**
✓ **Express.js manages backend logic and API requests.**
✓ **React.js handles frontend rendering and UI interactions.**
✓ **Proper integration between frontend & backend ensures a smooth user experience.**

CONNECTING FRONTEND (REACT) WITH BACKEND (NODE.JS)

CHAPTER 1: INTRODUCTION TO CONNECTING REACT WITH NODE.JS

1.1 Why Connect React with Node.js?

React is a **frontend framework** for building user interfaces, while Node.js with Express serves as the **backend API** to handle database interactions and business logic. Connecting them allows React to **fetch and display data dynamically** from a backend server.

- **Benefits of Connecting React with Node.js:**

✓ Enables **real-time data fetching** (e.g., user authentication, blog posts, product listings).
✓ Separates **frontend and backend logic**, making the application scalable.
✓ Ensures a **seamless user experience** with dynamic UI updates.

- **How the Connection Works:**

1. **Frontend (React)** → Sends HTTP requests to the **backend API** using fetch() or Axios.
2. **Backend (Node.js/Express)** → Processes requests, interacts with a **database**, and responds with JSON data.
3. **React Component** → Receives JSON data and updates the UI dynamically.

CHAPTER 2: SETTING UP THE BACKEND (NODE.JS + EXPRESS)

2.1 Initialize a Node.js Project

📌 Create a backend directory and initialize Node.js:

mkdir backend

cd backend

npm init -y

✓ Creates a package.json file for managing dependencies.

📌 Install Express and CORS for Backend API:

npm install express cors dotenv

✓ express → Web framework for handling API requests.
✓ cors → Allows frontend requests from a different domain.
✓ dotenv → Stores environment variables securely.

2.2 Creating an Express Server

📌 Create server.js and set up a basic API:

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

const cors = require("cors");

const dotenv = require("dotenv");


dotenv.config();

const app = express();
```

```
app.use(express.json());  // Enable JSON parsing
app.use(cors());  // Allow cross-origin requests

const PORT = process.env.PORT || 5000;

app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

✓ **Starts an Express server** on port 5000.
✓ Enables **CORS** to allow frontend requests.

2.3 Creating API Endpoints

📌 **Example: Defining a Simple GET API (/api/message)**

```
app.get("/api/message", (req, res) => {
  res.json({ message: "Hello from Node.js backend!" });
});
```

✓ Sends a **JSON response** when accessed.

CHAPTER 3: SETTING UP THE FRONTEND (REACT APP)

3.1 Create a React App

📌 **Navigate to the root folder and create a React app:**

```
npx create-react-app frontend
```

cd frontend

npm install axios

✓ create-react-app → Generates a **React project structure**.
✓ axios → Installs a package to **fetch API data from the backend**.

3.2 Fetching Data from Backend in React

📌 **Modify src/App.js to Fetch Data from Express API:**

```
import React, { useState, useEffect } from "react";
import axios from "axios";

function App() {
  const [message, setMessage] = useState("");

  useEffect(() => {
    axios.get("http://localhost:5000/api/message")
      .then(response => setMessage(response.data.message))
      .catch(error => console.error("Error fetching data:", error));
  }, []);

  return (
    <div>
```

```
        <h1>React & Node.js Integration</h1>
        <p>Backend Response: {message}</p>
      </div>
    );
}

export default App;
```

✓ axios.get("http://localhost:5000/api/message") fetches data from the **Node.js API**.
✓ The **backend response updates the React state dynamically**.

3.3 Running the React Frontend

📌 **Navigate to the React project and start the frontend:**

```
cd frontend
npm start
```

✓ Opens the React app in **http://localhost:3000/**.

CHAPTER 4: HANDLING CORS ISSUES BETWEEN REACT & NODE.JS

4.1 What is CORS (Cross-Origin Resource Sharing)?

By default, browsers **block API requests** between different origins (e.g., localhost:3000 → localhost:5000).

- **Fix:** Use the **CORS middleware** in Express.

📌 **Modify server.js to Allow Frontend Requests:**

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


app.use(cors({ origin: "http://localhost:3000" }));
```

✓ Ensures the **backend allows requests from React frontend**.

Chapter 5: Connecting React with a MongoDB Database via Node.js

5.1 Install MongoDB and Mongoose

📌 **Navigate to the backend folder and install MongoDB dependencies:**

```
npm install mongoose
```

✓ mongoose → Connects Node.js to MongoDB.

5.2 Connecting Backend to MongoDB

📌 **Modify server.js to Connect to MongoDB:**

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


mongoose.connect("mongodb://127.0.0.1:27017/myDatabase", {
  useNewUrlParser: true,
  useUnifiedTopology: true
```

```
})
.then(() => console.log("Connected to MongoDB"))
.catch(err => console.error("MongoDB connection error:", err));
```

✓ Connects the backend to **MongoDB running locally**.

5.3 Creating a Mongoose Schema

📌 **Create models/User.js in the backend folder:**

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

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

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

✓ Defines a **User schema** with name and email.

5.4 Creating an API to Save Data to MongoDB

📌 **Modify server.js to Include a POST API for Users:**

```
const User = require("./models/User");
```

```
app.post("/api/users", async (req, res) => {
  try {
    const newUser = new User(req.body);
    await newUser.save();
    res.status(201).json(newUser);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

✓ Receives user data from **React frontend** and saves it in MongoDB.

5.5 Sending Data from React to Express API

📌 Modify App.js in React to Submit User Data:

```
import React, { useState } from "react";
import axios from "axios";

function App() {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");

  const handleSubmit = async (e) => {
```

```
    e.preventDefault();
    await axios.post("http://localhost:5000/api/users", { name, email });
    alert("User added successfully!");
  };

  return (
    <div>
      <h1>Add User</h1>
      <form onSubmit={handleSubmit}>
        <input type="text" placeholder="Name" onChange={(e) => setName(e.target.value)} required />
        <input type="email" placeholder="Email" onChange={(e) => setEmail(e.target.value)} required />
        <button type="submit">Submit</button>
      </form>
    </div>
  );
}

export default App;
```

✓ The React form **sends user data to the backend API**.

Case Study: How Twitter Uses React & Node.js Together

Challenges Faced by Twitter

✓ Handling **millions of real-time tweets and interactions**.
✓ Ensuring **fast communication between frontend and backend**.

Solutions Implemented

✓ Used **React for a fast, interactive UI**.
✓ Implemented **Node.js & Express.js API** for real-time tweet processing.
✓ Stored tweets in **MongoDB for efficient retrieval**.

🔹 **Key Takeaways from Twitter's Tech Stack:**

✓ **Fast API responses improve user experience**.
✓ **MongoDB enables efficient storage and retrieval**.
✓ **Handling CORS issues ensures smooth frontend-backend communication**.

📝 Exercise

✅ Modify the API to **fetch user details and display them in React**.
✅ Implement **form validation** to prevent empty user submissions.
✅ Deploy both the **backend (Node.js) and frontend (React) using Heroku or Vercel**.

Conclusion

✓ React interacts with Node.js via API requests.

✓ CORS handling ensures seamless frontend-backend communication.

✓ MongoDB integrates easily with Express.js for dynamic data storage.

✓ Building full-stack applications improves scalability and performance.

STATE MANAGEMENT & API INTEGRATION

CHAPTER 1: UNDERSTANDING STATE MANAGEMENT

1.1 What is State Management?

State management refers to the process of **storing, updating, and sharing application data** across components in a React application. It ensures that the UI **reflects real-time data changes** efficiently.

- **Why is State Management Important?**

✓ Maintains **data consistency** across the app.

✓ Reduces **unnecessary re-renders**.

✓ Helps manage **complex application states** (e.g., user authentication, theme, API responses).

- **Types of State in React Apps:**

Type	Description	Example
Local State	Data managed within a single component	useState() to manage form inputs
Global State	Data shared across multiple components	Using Redux or Context API for authentication
Server State	Data fetched from an API	Managing API responses in state
UI State	Temporary UI changes	Modal visibility, loading indicators

CHAPTER 2: MANAGING STATE WITH USESTATE AND USECONTEXT

2.1 Using useState for Local State

📌 Example: Managing a Counter with useState

```
import React, { useState } from "react";


function Counter() {
  const [count, setCount] = useState(0);


  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}


export default Counter;
```

✓ useState(0) initializes **count** with 0.
✓ setCount(count + 1) updates **state dynamically**.

2.2 Using useContext for Global State

📌 Example: Creating a Theme Context

```
import React, { createContext, useState, useContext } from "react";

const ThemeContext = createContext();

export const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState("light");
  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};

export const useTheme = () => useContext(ThemeContext);
```

✓ createContext() creates **a global state provider**.
✓ useContext(ThemeContext) allows components to **access state anywhere**.

📌 **Example: Using Theme Context in Components**

```
import { useTheme } from "./ThemeProvider";

function ToggleTheme() {
  const { theme, setTheme } = useTheme();
```

```
  return (
    <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>
      Switch to {theme === "light" ? "Dark" : "Light"} Mode
    </button>
  );
}
```

✓ Enables **global theme switching**.

CHAPTER 3: USING REDUX TOOLKIT FOR ADVANCED STATE MANAGEMENT

3.1 Setting Up Redux Toolkit

📌 **Install Redux Toolkit:**

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

✓ Installs **Redux Toolkit** and **React bindings for Redux**.

3.2 Creating a Redux Store

📌 **Create store.js to Configure Redux Store**

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

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

✓ configureStore() **creates the Redux store**.

📌 **Wrap the App with Redux Provider (index.js)**

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

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById("root")
);
```

✓ **Makes Redux state available** throughout the app.

3.3 Managing State with Redux Slices

📌 Create counterSlice.js for Counter State Management

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

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

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

✓ Defines **initial state and reducers** in a single file.

📌 Using Redux State in a Component (Counter.js)

```
import { useSelector, useDispatch } from "react-redux";
import { increment, decrement } from "./counterSlice";
```

```
function Counter() {
  const count = useSelector(state => state.counter.value);
  const dispatch = useDispatch();

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => dispatch(increment())}>+</button>
      <button onClick={() => dispatch(decrement())}>-</button>
    </div>
  );
}

export default Counter;
```

✓ **useSelector() gets the state** from Redux.
✓ **useDispatch() sends actions** to modify state.

CHAPTER 4: API INTEGRATION IN REACT

4.1 Fetching Data from an API

📌 Example: Fetching Data with fetch()

```
import React, { useState, useEffect } from "react";
```

```
function UsersList() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(response => response.json())
      .then(data => setUsers(data));
  }, []);

  return (
    <ul>
      {users.map(user => <li key={user.id}>{user.name}</li>)}
    </ul>
  );
}

export default UsersList;
```

✓ Uses **useEffect() to fetch API data on mount**.

4.2 Using Axios for API Calls

📌 Install Axios:

npm install axios

📌 Example: Fetching API Data with Axios

```
import React, { useState, useEffect } from "react";
import axios from "axios";

function UsersList() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    axios.get("https://jsonplaceholder.typicode.com/users")
      .then(response => setUsers(response.data));
  }, []);

  return (
    <ul>
      {users.map(user => <li key={user.id}>{user.name}</li>)}
    </ul>
  );
}
```

```
export default UsersList;
```

✓ **Axios automatically parses JSON responses**.

4.3 Integrating API Calls with Redux Toolkit

📌 Modify userSlice.js to Fetch Data Asynchronously

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

export const fetchUsers = createAsyncThunk("users/fetchUsers", async () => {
  const response = await axios.get("https://jsonplaceholder.typicode.com/users");
  return response.data;
});

const userSlice = createSlice({
  name: "users",
  initialState: { users: [], loading: false },
  extraReducers: (builder) => {
    builder
      .addCase(fetchUsers.pending, (state) => { state.loading = true;
})
```

```
      .addCase(fetchUsers.fulfilled, (state, action) => {
        state.loading = false;
        state.users = action.payload;
      });
  }
});

export default userSlice.reducer;
```

📌 Using Redux API Data in a Component

```
import { useEffect } from "react";
import { useDispatch, useSelector } from "react-redux";
import { fetchUsers } from "./userSlice";

function Users() {
  const dispatch = useDispatch();
  const { users, loading } = useSelector(state => state.users);

  useEffect(() => {
    dispatch(fetchUsers());
  }, [dispatch]);
```

```
  return loading ? <p>Loading...</p> : users.map(user =>
<p>{user.name}</p>);

}
```

✓ **Redux manages API requests efficiently**.

Conclusion

✓ **State management ensures smooth UI updates.**
✓ **Redux Toolkit simplifies global state handling.**
✓ **API integration fetches dynamic data for real-time applications.**
✓ **Efficient state and API management improves performance.**

ROLE-BASED AUTHENTICATION (ADMIN, USER) IN NODE.JS & EXPRESS.JS

CHAPTER 1: INTRODUCTION TO ROLE-BASED AUTHENTICATION

1.1 What is Role-Based Authentication?

Role-Based Authentication (RBAC) is a security mechanism that **assigns different levels of access** to users based on their roles (e.g., **Admin, User, Editor**). This ensures that only authorized users can **perform specific actions** in an application.

- **Why Use Role-Based Authentication?**

✓ Restricts **sensitive operations** to specific roles.
✓ Enhances **security** by preventing unauthorized access.
✓ Ensures **users only access permitted functionalities**.

- **Common Role-Based Access Examples:**

Role	Permissions	Example
Admin	Full access (CRUD)	Manage users, delete posts
User	Limited access (Read/Write)	View & edit personal profile
Editor	Moderate content	Edit blog posts, approve comments

CHAPTER 2: SETTING UP THE PROJECT

2.1 Installing Required Packages

📌 Initialize a Node.js project and install dependencies:

mkdir role-based-auth

cd role-based-auth

npm init -y

npm install express mongoose bcryptjs jsonwebtoken dotenv cors

✓ express → Handles API routing.
✓ mongoose → Manages MongoDB interactions.
✓ bcryptjs → Hashes passwords securely.
✓ jsonwebtoken → Handles authentication with JWT.
✓ dotenv → Stores configuration variables securely.
✓ cors → Enables API requests from frontend apps.

2.2 Setting Up the Express Server

📌 Create server.js and configure Express.js:

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

const mongoose = require("mongoose");

const dotenv = require("dotenv");

const cors = require("cors");


dotenv.config();

const app = express();

app.use(express.json()); // Middleware to parse JSON requests
```

```
app.use(cors()); // Enables Cross-Origin Resource Sharing


const PORT = process.env.PORT || 5000;


// Connect to MongoDB
mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log("MongoDB Connected"))
.catch(err => console.error("MongoDB connection error:", err));


app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

✓ **Starts a Node.js API server** on port 5000.
✓ **Connects to MongoDB** using mongoose.

📌 **Create a .env file for database credentials:**

```
MONGO_URI=mongodb+srv://your_user:your_password@cluster.mongodb.net/roleDB
JWT_SECRET=your_jwt_secret
```

✓ Stores **MongoDB connection string securely**.

CHAPTER 3: DEFINING THE USER MODEL WITH ROLES

📌 Create models/User.js to define the schema:

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


const UserSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true },
  password: String,
  role: { type: String, enum: ["user", "admin"], default: "user" } // Default role is 'user'
});


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

✓ Defines a **User model** with name, email, password, and role.
✓ enum: ["user", "admin"] restricts roles to **specific values**.

CHAPTER 4: IMPLEMENTING USER REGISTRATION & ROLE ASSIGNMENT

📌 Create routes/auth.js for authentication routes:

```
const express = require("express");
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");
const User = require("../models/User");
```

```
const router = express.Router();


// Register User with Role
router.post("/register", async (req, res) => {
  try {
    const { name, email, password, role } = req.body;


    // Check if user exists
    if (await User.findOne({ email })) {
      return res.status(400).json({ message: "Email already exists" });
    }


    // Hash password
    const hashedPassword = await bcrypt.hash(password, 10);


    // Save user with role
    const newUser = new User({ name, email, password:
hashedPassword, role });
    await newUser.save();
```

```
    res.status(201).json({ message: "User registered successfully", role: newUser.role });
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

module.exports = router;
```

✓ **Assigns user roles dynamically** (admin or user).
✓ **Hashes passwords** before saving to the database.

- **Testing with Postman:**
  - Method: **POST**
  - URL: http://localhost:5000/auth/register
  - Body (JSON):

```
{
  "name": "Admin User",
  "email": "admin@example.com",
  "password": "securepassword",
  "role": "admin"
}
```

CHAPTER 5: IMPLEMENTING USER LOGIN & TOKEN GENERATION

📌 Add a login route to routes/auth.js:

```
router.post("/login", async (req, res) => {
  try {
    const { email, password } = req.body;

    // Check if user exists
    const user = await User.findOne({ email });
    if (!user) return res.status(400).json({ message: "Invalid credentials" });

    // Validate password
    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) return res.status(400).json({ message: "Invalid credentials" });

    // Generate JWT Token with role
    const token = jwt.sign(
      { userId: user._id, role: user.role },
      process.env.JWT_SECRET,
      { expiresIn: "1h" }
```

```
    );


    res.json({ token, role: user.role });
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});
```

✓ **Validates user credentials** and generates a JWT token.
✓ Includes **role information** in the token.

CHAPTER 6: PROTECTING ROUTES WITH ROLE-BASED ACCESS CONTROL

6.1 Creating Authentication Middleware

📌 **Create middleware/authMiddleware.js:**

```
const jwt = require("jsonwebtoken");


module.exports = (role) => {
  return (req, res, next) => {
    const token = req.header("Authorization");
    if (!token) return res.status(401).json({ message: "Access Denied" });
```

```
    try {
      const verified = jwt.verify(token, process.env.JWT_SECRET);
      req.user = verified;

      if (role && req.user.role !== role) {
        return res.status(403).json({ message: "Forbidden: Insufficient Permissions" });
      }

      next();
    } catch (error) {
      res.status(400).json({ message: "Invalid Token" });
    }
  };
};
```

✓ **Verifies JWT tokens** and extracts user role.
✓ Restricts access to **specific user roles**.

6.2 Creating a Protected Admin Route

📌 **Modify routes/auth.js to add an admin-only route:**

```
const authMiddleware = require("../middleware/authMiddleware");
```

```
// Admin-only Route
router.get("/admin", authMiddleware("admin"), (req, res) => {
  res.json({ message: "Welcome, Admin!" });
});
```

✓ Only users with **role: "admin"** can access /admin.

- **Testing Admin Access with Postman:**
  - Method: **GET**
  - URL: http://localhost:5000/auth/admin
  - Headers:
    - Authorization: Bearer YOUR_JWT_TOKEN

Case Study: How Netflix Uses Role-Based Authentication

Challenges Faced by Netflix

✓ Restricting **admin-level content management**.
✓ Managing **user subscriptions and access levels**.

Solutions Implemented

✓ Used **role-based authentication** to limit admin access.
✓ Implemented **tier-based subscription plans** (Basic, Premium).
✓ Used **JWT tokens** for secure session handling.

- **Key Takeaways:**

✓ **Role-based access prevents unauthorized actions.**
✓ **JWT ensures secure authentication across platforms.**
✓ **Access control improves data security and compliance.**

Conclusion

✓ Express.js enables role-based authentication for web applications.

✓ JWT provides secure, token-based authentication.

✓ Middleware restricts access based on user roles.

✓ Real-world applications rely on RBAC for security and efficiency.

OAuth Integration (Google, Facebook Login)

Chapter 1: Introduction to OAuth Authentication

1.1 What is OAuth?

OAuth (Open Authorization) is a **secure authentication standard** that allows users to log in to applications **without sharing their passwords**. Instead, they use third-party authentication providers like **Google, Facebook, GitHub, etc.**

- **Why Use OAuth?**

✓ **Enhances security** (no need to store user passwords).
✓ **Improves user experience** (faster login with existing accounts).
✓ **Increases trust** by using well-known providers like Google & Facebook.

- **How OAuth Works:**

1. **User clicks "Login with Google/Facebook"**.
2. The app **redirects to the OAuth provider (Google/Facebook)**.
3. The user **grants permissions** to share their profile.
4. The provider **returns an access token** to the app.
5. The app uses the token to **authenticate the user**.

Chapter 2: Setting Up OAuth in a Node.js & Express App

2.1 Installing Required Dependencies

📌 Create a Node.js project & install required packages:

mkdir oauth-authentication

cd oauth-authentication

npm init -y

npm install express passport passport-google-oauth20 passport-facebook dotenv cors express-session

✓ passport → Handles authentication.
✓ passport-google-oauth20 → Google OAuth strategy.
✓ passport-facebook → Facebook OAuth strategy.
✓ express-session → Manages user sessions.
✓ dotenv → Stores sensitive credentials securely.

2.2 Setting Up Express Server

📌 Create server.js to initialize Express

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

const passport = require("passport");

const session = require("express-session");

const dotenv = require("dotenv");


dotenv.config();

const app = express();
```

```
// Session Middleware
app.use(session({ secret: "secretkey", resave: false,
saveUninitialized: true }));


// Passport Middleware
app.use(passport.initialize());
app.use(passport.session());


app.get("/", (req, res) => {
   res.send("<h1>OAuth Authentication App</h1><a
href='/auth/google'>Login with Google</a>");
});


const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`Server running on port
${PORT}`));
```

✓ Sets up **Express server** with passport for authentication.
✓ Uses express-session to **store user sessions**.

CHAPTER 3: GOOGLE OAUTH INTEGRATION

3.1 Creating Google OAuth Credentials

📌 **Steps to Get Google OAuth Credentials:**

1. Go to Google Developer Console.
2. Click **"Create Project"** → Name it **OAuth-Project**.
3. Navigate to **APIs & Services** → **Credentials**.
4. Click **"Create Credentials"** → **OAuth Client ID**.
5. Configure **Redirect URI:**
    - Development: http://localhost:5000/auth/google/callback
    - Production: https://yourdomain.com/auth/google/callback
6. Copy **Client ID & Client Secret** into .env file.

📌 **Update .env File with Google Credentials:**

```
GOOGLE_CLIENT_ID=your_google_client_id
GOOGLE_CLIENT_SECRET=your_google_client_secret
```

3.2 Configuring Google OAuth in Passport.js

📌 **Create auth/googleAuth.js**

```
const passport = require("passport");
const GoogleStrategy = require("passport-google-oauth20").Strategy;

passport.use(new GoogleStrategy({
  clientID: process.env.GOOGLE_CLIENT_ID,
```

```
  clientSecret: process.env.GOOGLE_CLIENT_SECRET,
  callbackURL: "/auth/google/callback"
}, (accessToken, refreshToken, profile, done) => {
  done(null, profile);
}));

passport.serializeUser((user, done) => {
  done(null, user);
});

passport.deserializeUser((user, done) => {
  done(null, user);
});
```

✓ Configures **Google OAuth strategy** using Passport.
✓ Stores user details in **session** for authentication.

3.3 Implementing Google OAuth Routes

📌 Update server.js to Include Google Auth Routes

```
require("./auth/googleAuth");

app.get("/auth/google", passport.authenticate("google", { scope: ["profile", "email"] }));
```

```
app.get("/auth/google/callback", passport.authenticate("google", {
  successRedirect: "/dashboard",
  failureRedirect: "/"
}));

app.get("/dashboard", (req, res) => {
  res.send(`<h1>Welcome ${req.user.displayName}</h1><a href='/logout'>Logout</a>`);
});

app.get("/logout", (req, res) => {
  req.logout(() => res.redirect("/"));
});
```

✓ **Redirects user to Google login**.
✓ On success, redirects to **dashboard** with user info.
✓ **req.logout()** logs out the user and redirects to home.

CHAPTER 4: FACEBOOK OAUTH INTEGRATION

4.1 Creating Facebook OAuth Credentials

📌 Steps to Get Facebook OAuth Credentials:

1. Go to Facebook Developer Console.

2. Click **"Create App"** → Choose **"Consumer"**.
3. Go to **Settings** → **Basic** → Add **App Domains**.
4. Navigate to **Facebook Login** → **Settings**.
5. Add **Valid OAuth Redirect URIs:**
    - http://localhost:5000/auth/facebook/callback
6. Copy **App ID & App Secret** into .env file.

📌 **Update .env File with Facebook Credentials:**

```
FACEBOOK_APP_ID=your_facebook_app_id
FACEBOOK_APP_SECRET=your_facebook_app_secret
```

4.2 Configuring Facebook OAuth in Passport.js

📌 **Create auth/facebookAuth.js**

```
const passport = require("passport");
const FacebookStrategy = require("passport-facebook").Strategy;

passport.use(new FacebookStrategy({
  clientID: process.env.FACEBOOK_APP_ID,
  clientSecret: process.env.FACEBOOK_APP_SECRET,
  callbackURL: "/auth/facebook/callback",
  profileFields: ["id", "displayName", "emails"]
}, (accessToken, refreshToken, profile, done) => {
```

```
  done(null, profile);
}));

passport.serializeUser((user, done) => {
  done(null, user);
});

passport.deserializeUser((user, done) => {
  done(null, user);
});
```

✓ Configures **Facebook OAuth strategy** using Passport.

4.3 Implementing Facebook OAuth Routes

📌 Update server.js to Include Facebook Auth Routes

```
require("./auth/facebookAuth");

app.get("/auth/facebook", passport.authenticate("facebook"));

app.get("/auth/facebook/callback",
passport.authenticate("facebook", {
  successRedirect: "/dashboard",
```

```
  failureRedirect: "/"
}));
```

✓ **Redirects user to Facebook login**.
✓ On success, redirects to **dashboard** with user info.

CHAPTER 5: SECURING OAUTH IMPLEMENTATION

5.1 Using Environment Variables (dotenv)

✓ **Never hardcode OAuth credentials** in code.
✓ Use .env to store sensitive keys.

5.2 Implementing Session-Based Authentication

📌 **Ensure Secure Session Storage in server.js**

```
app.use(session({
  secret: process.env.SESSION_SECRET,
  resave: false,
  saveUninitialized: true,
  cookie: { secure: false }
}));
```

✓ Stores **user sessions securely** after authentication.

Case Study: How Spotify Uses OAuth for Secure Authentication

Challenges Faced by Spotify

✓ Ensuring **secure user authentication** across millions of devices.
✓ Preventing **unauthorized access to user data**.

Solutions Implemented

✓ Used **OAuth 2.0 for login with Google & Facebook**.
✓ Implemented **session management & token expiration**.

🔹 **Key Takeaways from Spotify's OAuth Strategy:**
✓ **OAuth authentication improves user security & trust.**
✓ **Session management prevents unauthorized access.**
✓ **Storing OAuth credentials securely prevents breaches.**

📝 **Exercise**

✅ Implement **GitHub OAuth authentication** in Passport.js.
✅ Add **JWT authentication** after OAuth login.
✅ Store authenticated users in a **MongoDB database**.

Conclusion

✓ OAuth provides a secure way to authenticate users via Google, Facebook, etc.
✓ Passport.js simplifies OAuth integration in Express.js applications.
✓ Using session-based authentication ensures users stay logged in.
✓ OAuth is widely used by apps like Spotify, Airbnb, and Netflix for secure logins.

CACHING AND OPTIMIZING API CALLS

CHAPTER 1: INTRODUCTION TO CACHING AND API OPTIMIZATION

1.1 What is Caching?

Caching is a technique that **stores frequently accessed data** temporarily to **reduce response time and server load**. Instead of making repetitive API requests, cached data can be retrieved quickly from memory, improving performance.

- **Why Use Caching?**

✓ **Reduces API response time** by storing previously requested data.
✓ **Minimizes database load,** preventing frequent queries.
✓ **Improves scalability**, especially for high-traffic applications.

- **Common Caching Mechanisms:**

Caching Method	Description	Example
Memory Cache	Stores API responses in memory	Redis, Node.js Cache
Browser Cache	Stores API responses in the client's browser	LocalStorage, IndexedDB
CDN Caching	Caches static files and API responses globally	Cloudflare, AWS CloudFront

CHAPTER 2: IMPLEMENTING CACHING IN NODE.JS APIS

2.1 Using Memory Cache with node-cache

📌 Step 1: Install node-cache package

npm install node-cache

📌 Step 2: Implement Basic Caching in Express API (server.js)

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

const NodeCache = require("node-cache");


const app = express();

const cache = new NodeCache({ stdTTL: 60 }); // Cache expires after 60 seconds


// Sample API Route with Caching

app.get("/api/data", (req, res) => {

  const cachedData = cache.get("data");


  if (cachedData) {

    return res.json({ source: "cache", data: cachedData });

  }


  // Simulated API Response

  const apiData = { message: "Hello from the server!" };

  cache.set("data", apiData); // Store data in cache
```

```
  res.json({ source: "server", data: apiData });
});

app.listen(5000, () => console.log("Server running on port 5000"));
```

✓ **Checks cache before making an API call**.
✓ **Stores response in cache for 60 seconds** to reduce load.

2.2 Using Redis for High-Performance Caching

Redis is an **in-memory key-value store** that provides **fast caching** for API calls.

📌 **Step 1: Install Redis and redis npm package**

```
npm install redis dotenv
```

📌 **Step 2: Set Up Redis Connection in server.js**

```
const redis = require("redis");
const client = redis.createClient();

client.connect().then(() => console.log("Connected to Redis"));
```

📌 **Step 3: Cache API Responses with Redis**

```
app.get("/api/data", async (req, res) => {
  const cachedData = await client.get("data");
```

```
  if (cachedData) {
    return res.json({ source: "cache", data: JSON.parse(cachedData) });
  }

  // Simulated API Response
  const apiData = { message: "Hello from the server!" };
  await client.setEx("data", 60, JSON.stringify(apiData)); // Store in Redis for 60 seconds

  res.json({ source: "server", data: apiData });
});
```

✓ Uses **Redis for persistent in-memory caching**.
✓ **Faster than traditional database queries**.

CHAPTER 3: OPTIMIZING API CALLS IN FRONTEND (REACT)

3.1 Using localStorage for Caching API Responses

📌 Example: Storing API Data in Browser Storage (App.js)

```
import { useState, useEffect } from "react";
import axios from "axios";
```

```
function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    const cachedData = localStorage.getItem("apiData");

    if (cachedData) {
      setData(JSON.parse(cachedData));
    } else {
      axios.get("http://localhost:5000/api/data").then(response => {
        setData(response.data);
        localStorage.setItem("apiData",
JSON.stringify(response.data));
      });
    }
  }, []);

  return <div>{data ? data.message : "Loading..."}</div>;
}

export default App;
```

✓ **Stores API response in localStorage**.
✓ Reduces **unnecessary API calls** on page reload.

3.2 Using React Query for API Optimization

📌 Step 1: Install React Query

npm install @tanstack/react-query

📌 Step 2: Implement Caching in React API Calls (App.js)

```
import { useQuery } from "@tanstack/react-query";
import axios from "axios";

function fetchData() {
  return axios.get("http://localhost:5000/api/data").then(res => res.data);
}

function App() {
  const { data, isLoading } = useQuery(["apiData"], fetchData, { staleTime: 60000 });

  return <div>{isLoading ? "Loading..." : data.message}</div>;
}
```

export default App;

✓ Uses **React Query to cache API responses for 60 seconds**.
✓ Prevents **redundant API calls when switching between pages**.

CHAPTER 4: IMPLEMENTING API RATE LIMITING FOR PERFORMANCE OPTIMIZATION

4.1 Using express-rate-limit to Prevent API Overuse

📌 **Step 1: Install express-rate-limit**

npm install express-rate-limit

📌 **Step 2: Apply Rate Limiting in server.js**

```
const rateLimit = require("express-rate-limit");


const limiter = rateLimit({
  windowMs: 1 * 60 * 1000, // 1 minute
  max: 10, // Limit to 10 requests per minute
  message: "Too many requests. Please try again later."
});


app.use("/api", limiter);
```

✓ Prevents **excessive API requests from a single user**.

CHAPTER 5: USING CDN FOR API CACHING

5.1 Using Cloudflare to Cache API Responses

CDNs like **Cloudflare** can cache API responses globally to **reduce latency and server load**.

- **Steps to Enable API Caching with Cloudflare:**

1. Go to **Cloudflare Dashboard**.
2. Navigate to **Caching → Configuration**.
3. Set **Caching Level** to **Standard** for API requests.
4. Enable **Edge Cache TTL** to store responses globally.

✓ Reduces **server requests by serving cached responses**.
✓ Speeds up API calls for **global users**.

Case Study: How Twitter Uses Caching & API Optimization

Challenges Faced by Twitter

✓ Handling **millions of API requests per second**.
✓ Ensuring **real-time content updates** with minimal latency.
✓ Preventing **server crashes due to high load**.

Solutions Implemented

✓ Used **Redis for caching frequently accessed tweets**.
✓ Implemented **API rate limiting** to prevent abuse.
✓ Cached API responses using **Cloudflare CDN** for global access.

- **Key Takeaways from Twitter's API Optimization:**

✓ **Caching improves speed and reduces server load**.

✓ **Rate limiting prevents API abuse and improves reliability**.
✓ **CDNs enable faster API responses for global users**.

📝 Exercise

✅ Implement **Redis caching** in an Express API.
✅ Store API responses in **localStorage** to reduce frontend API calls.
✅ Add **rate limiting** to an API using express-rate-limit.
✅ Configure **Cloudflare caching** for API endpoints.

Conclusion

✓ **Caching reduces API response time and improves scalability.**
✓ **Redis offers high-performance in-memory storage for APIs.**
✓ **Frontend caching with localStorage minimizes redundant requests.**
✓ **Rate limiting protects APIs from excessive traffic and abuse.**

DEPLOYING FULL STACK APPLICATIONS

CHAPTER 1: INTRODUCTION TO FULL STACK DEPLOYMENT

1.1 What is Full Stack Deployment?

Deploying a full-stack application means making the **frontend (React, Angular, Vue)** and **backend (Node.js, Express, Django, Flask)** available for public or production use. It includes **setting up servers, databases, and networking** for seamless interaction between frontend, backend, and database.

- **Why is Deployment Important?**

✓ Makes the application **accessible to users worldwide**.
✓ Ensures **scalability and high availability**.
✓ Supports **database connections and API integrations**.

- **Key Components in Deployment:**

Component	Purpose	Examples
Frontend	UI/Client-side application	React, Angular, Vue
Backend	API/Server-side logic	Node.js, Express, Django
Database	Stores application data	MongoDB, PostgreSQL, MySQL
Hosting Service	Runs the application	Vercel, Netlify, AWS, DigitalOcean

Domain Name	Human-readable address	myapp.com

CHAPTER 2: DEPLOYING THE FRONTEND (REACT)

2.1 Choosing a Frontend Deployment Platform

Popular frontend hosting services:

✓ **Netlify** (best for React apps).

✓ **Vercel** (ideal for Next.js).

✓ **GitHub Pages** (good for static sites).

2.2 Deploying React on Netlify

📌 **Step 1: Build the React App**

npm run build

✓ Generates an optimized build/ folder for deployment.

📌 **Step 2: Deploy on Netlify**

1. **Sign up** at Netlify.
2. Click **New Site** → Connect **GitHub repository**.
3. Select **React project** → Set build command to:
4. npm run build
5. Click **Deploy** → Netlify **automatically deploys** the React app.

✓ **Netlify provides a live URL** (e.g., myapp.netlify.app).

2.3 Deploying React on Vercel

📌 **Steps:**

1. Install Vercel CLI:
2. npm install -g vercel
3. Login to Vercel:
4. vercel login
5. Deploy project:
6. vercel

✓ **Vercel provides automatic previews for GitHub commits.**

CHAPTER 3: DEPLOYING THE BACKEND (NODE.JS & EXPRESS.JS)

3.1 Choosing a Backend Hosting Service

✓ **Render** (free Node.js hosting).
✓ **Heroku** (easy deployment for APIs).
✓ **AWS, DigitalOcean, Linode** (scalable infrastructure).

3.2 Deploying Express.js on Render

📌 **Step 1: Create server.js in Your Project**

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

const app = express();

app.get("/", (req, res) => res.send("Backend is running!"));
```

```
const PORT = process.env.PORT || 5000;

app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

✓ This **starts an Express.js server on port 5000**.

📌 Step 2: Create package.json (Ensure Start Command is Defined)

```
"scripts": {
  "start": "node server.js"
}
```

📌 Step 3: Deploy Backend on Render

1. Sign up on Render.
2. Click **New Web Service** → Connect **GitHub repo**.
3. Set **Build Command:**
4. npm install
5. Set **Start Command:**
6. node server.js
7. Click **Deploy** → Render provides a **public API URL**.

✓ Your **backend API is now live** (e.g., https://myapp-api.onrender.com).

3.3 Deploying Express.js on Heroku

📌 Step 1: Install Heroku CLI

npm install -g heroku

📌 **Step 2: Login to Heroku**

heroku login

📌 **Step 3: Deploy App on Heroku**

heroku create my-app

git add .

git commit -m "Deploy backend"

git push heroku main

✓ Heroku **deploys the backend API** and provides a live URL.

CHAPTER 4: DEPLOYING THE DATABASE (MONGODB & POSTGRESQL)

4.1 Deploying MongoDB on MongoDB Atlas

📌 **Step 1: Create a MongoDB Atlas Cluster**

1. Sign up at MongoDB Atlas.
2. Create a **new cluster** → Choose **Free Tier**.
3. Click **Connect** → Select **Node.js Driver**.
4. Copy the connection string:
5. mongodb+srv://username:password@cluster.mongodb.net/mydatabase

📌 **Step 2: Use the Connection in Express.js**

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

mongoose.connect(process.env.MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true })

  .then(() => console.log("Connected to MongoDB Atlas"))

  .catch(err => console.error(err));
```

✓ Express.js **connects to the remote MongoDB database**.

4.2 Deploying PostgreSQL on ElephantSQL

📌 Step 1: Create a PostgreSQL Database on ElephantSQL

1. Sign up at ElephantSQL.
2. Click **Create New Instance** → Select **Free Plan**.
3. Copy the **PostgreSQL Connection URL**:
4. postgres://user:password@host:port/database

📌 Step 2: Connect PostgreSQL to Express.js

```
const { Pool } = require("pg");

const pool = new Pool({

  connectionString: process.env.POSTGRES_URL,

  ssl: { rejectUnauthorized: false }

});
```

✓ Express.js **connects to PostgreSQL** on a remote server.

CHAPTER 5: CONNECTING FRONTEND TO BACKEND

5.1 Handling CORS Issues

📌 Install cors Package in Backend

npm install cors

📌 Enable CORS in Express.js (server.js)

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

app.use(cors());
```

✓ Prevents **CORS errors** when frontend calls backend APIs.

5.2 Making API Calls from React Frontend

📌 Example: Fetching Data from Backend API

```
import { useState, useEffect } from "react";


function Users() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch("https://myapp-api.onrender.com/api/users")
      .then(response => response.json())
      .then(data => setUsers(data));
  }, []);
```

```
  return users.map(user => <p key={user.id}>{user.name}</p>);
}
```

✓ Fetches **backend API data dynamically**.

Case Study: How Airbnb Deploys Full Stack Applications

Challenges Faced by Airbnb

✓ Deploying **frontend, backend, and databases at scale**.
✓ Handling **millions of users and listings**.
✓ Ensuring **fast response times and security**.

Solutions Implemented

✓ Used **AWS for backend hosting** (EC2, Lambda).
✓ Hosted frontend on **Vercel for fast static content delivery**.
✓ Managed **databases with PostgreSQL & MongoDB** for scalability.

- **Key Takeaways from Airbnb's Deployment Strategy:**

✓ **Cloud hosting enables global availability.**
✓ **Frontend and backend should be independently scalable.**
✓ **Database optimizations improve performance.**

📝 **Exercise**

✅ Deploy a **React frontend** using **Netlify** or **Vercel**.
✅ Deploy an **Express.js API** using **Render** or **Heroku**.

✅ Deploy a **MongoDB or PostgreSQL database** on a cloud service.
✅ Connect **frontend to backend** and handle **CORS issues**.

Conclusion

✓ Netlify, Vercel, Render, and Heroku simplify deployment.
✓ MongoDB Atlas and ElephantSQL provide cloud-hosted databases.
✓ Handling CORS ensures smooth API integration.
✓ Optimized deployments enhance speed and scalability.

ASSIGNMENT:

DEVELOP A TASK MANAGEMENT APP WITH USER AUTHENTICATION.

Assignment Solution: Develop a Task Management App with User Authentication

Step 1: Setting Up the Project

1.1 Initialize a Node.js Project

📌 **Run the following commands:**

mkdir task-manager

cd task-manager

npm init -y

✓ Creates a **Node.js project** with package.json.

1.2 Install Dependencies

📌 **Run:**

npm install express mongoose bcryptjs jsonwebtoken dotenv cors

✓ express → Handles API requests.
✓ mongoose → Connects to MongoDB.
✓ bcryptjs → Hashes passwords securely.
✓ jsonwebtoken → Manages authentication with JWT.
✓ dotenv → Stores environment variables.
✓ cors → Enables frontend API requests.

Step 2: Setting Up Express Server

📌 Create server.js and configure Express.js:

```
const express = require("express");
const mongoose = require("mongoose");
const dotenv = require("dotenv");
const cors = require("cors");

dotenv.config();
const app = express();

app.use(express.json()); // Parse JSON data
app.use(cors()); // Enable CORS

const PORT = process.env.PORT || 5000;

// Connect to MongoDB
mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log("MongoDB Connected"))
.catch(err => console.error("MongoDB connection error:", err));

app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

✓ **Connects to MongoDB** and starts the Express server.

📌 **Create a .env file for MongoDB connection:**

MONGO_URI=mongodb+srv://your_user:your_password@cluster.mongodb.net/taskDB

JWT_SECRET=your_jwt_secret

Step 3: Creating Models for Users & Tasks

3.1 Creating the User Model

📌 **Create models/User.js for user authentication:**

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

const UserSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true },
  password: String,
  role: { type: String, enum: ["user", "admin"], default: "user" }
});

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

✓ Defines **user schema with roles** (user or admin).
✓ unique: true ensures **no duplicate emails**.

3.2 Creating the Task Model

📌 Create models/Task.js to store tasks:

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


const TaskSchema = new mongoose.Schema({
  title: { type: String, required: true },
  description: String,
  status: { type: String, enum: ["pending", "in-progress", "completed"], default: "pending" },
  userId: { type: mongoose.Schema.Types.ObjectId, ref: "User", required: true }
});


module.exports = mongoose.model("Task", TaskSchema);
```

✓ **Stores tasks** with title, description, and status.
✓ userId links tasks to a specific **authenticated user**.

Step 4: Implementing User Authentication

📌 Create routes/auth.js for authentication routes:

```
const express = require("express");
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");
const User = require("../models/User");
```

```
const router = express.Router();


// Register User
router.post("/register", async (req, res) => {
  try {
    const { name, email, password, role } = req.body;


    if (await User.findOne({ email })) {
      return res.status(400).json({ message: "Email already exists" });
    }


    const hashedPassword = await bcrypt.hash(password, 10);
    const newUser = new User({ name, email, password:
hashedPassword, role });
    await newUser.save();


    res.status(201).json({ message: "User registered successfully" });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});


// Login User
```

```
router.post("/login", async (req, res) => {
  try {
    const { email, password } = req.body;
    const user = await User.findOne({ email });

    if (!user || !(await bcrypt.compare(password, user.password))) {
      return res.status(400).json({ message: "Invalid credentials" });
    }

    const token = jwt.sign({ userId: user._id, role: user.role },
process.env.JWT_SECRET, { expiresIn: "1h" });
    res.json({ token, role: user.role });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

module.exports = router;
```

✓ Uses **bcrypt** for password hashing.
✓ Generates **JWT token** with user role.

Step 5: Protecting Routes with Authentication Middleware

📌 **Create middleware/authMiddleware.js:**

```
const jwt = require("jsonwebtoken");


module.exports = (role) => {
  return (req, res, next) => {
    const token = req.header("Authorization");
    if (!token) return res.status(401).json({ message: "Access Denied" });


    try {
      const verified = jwt.verify(token, process.env.JWT_SECRET);
      req.user = verified;


      if (role && req.user.role !== role) {
        return res.status(403).json({ message: "Forbidden: Insufficient Permissions" });
      }


      next();
    } catch (error) {
      res.status(400).json({ message: "Invalid Token" });
    }
  };
};
```

✓ Ensures **only authenticated users** access routes.
✓ Restricts **role-based permissions**.

Step 6: Implementing Task Management CRUD Operations

📌 Create routes/task.js for handling tasks:

```
const express = require("express");
const authMiddleware = require("../middleware/authMiddleware");
const Task = require("../models/Task");

const router = express.Router();

// Create Task
router.post("/", authMiddleware("user"), async (req, res) => {
  const newTask = new Task({ ...req.body, userId: req.user.userId });
  await newTask.save();
  res.status(201).json(newTask);
});

// Get All Tasks for Authenticated User
router.get("/", authMiddleware("user"), async (req, res) => {
  const tasks = await Task.find({ userId: req.user.userId });
  res.json(tasks);
```

```
});

// Update Task Status
router.put("/:id", authMiddleware("user"), async (req, res) => {
  const task = await Task.findById(req.params.id);
  if (!task || task.userId.toString() !== req.user.userId) {
    return res.status(403).json({ message: "Unauthorized" });
  }
  task.status = req.body.status || task.status;
  await task.save();
  res.json(task);
});

// Delete Task
router.delete("/:id", authMiddleware("user"), async (req, res) => {
  const task = await Task.findById(req.params.id);
  if (!task || task.userId.toString() !== req.user.userId) {
    return res.status(403).json({ message: "Unauthorized" });
  }
  await task.deleteOne();
  res.json({ message: "Task deleted successfully" });
});
```

```
module.exports = router;
```

✓ **Users can create, view, update, and delete tasks.**
✓ **Each task is linked to the authenticated user.**

Step 7: Running & Testing the API

📌 **Modify server.js to use routes:**

```
const authRoutes = require("./routes/auth");
const taskRoutes = require("./routes/task");

app.use("/api/auth", authRoutes);
app.use("/api/tasks", taskRoutes);
```

✓ **Registers authentication and task routes.**

📌 **Start the Server:**

```
npm start
```

✓ API runs on http://localhost:5000.