Table of Content

S.NO	TOPIC	SUB TOPIC
1	Node.js + Express + MongoDB	Node.js Intro, npm & Scripts
2	Node.js + Express + MongoDB	Express App & Routes
3	Node.js + Express + MongoDB	Middleware in Express
4	Node.js + Express + MongoDB	Mongoose Setup & DB Connection
5	MongoDB + Mongoose CRUD	Define Schema & Models
6	MongoDB + Mongoose CRUD	CRUD Routes & DB Ops
7	MongoDB + Mongoose CRUD	API Testing with Postman
8	REST API Design + Postman	REST Design & Endpoints
9	REST API Design + Postman	Status Codes & Input Validation
10	REST API Design + Postman	Error Handling & Resilience
11	REST API Design + Postman	Project Structure
12	Authentication & Integration	User Auth – Register/Login
13	Authentication & Integration	JWT Token Authentication
14	Authentication & Integration	Auth Middleware & Access
15	Authentication & Integration	Final Project – Setup
16	Authentication & Integration	Final Project – Logic
17	Authentication & Integration	Final Project – Security
18	Authentication & Integration	Testing & Wrap-up

Unit 1: Node.js

1.1 What is Node.js?

Definition:

Node.js is a **JavaScript runtime** built on **Chrome's V8 JavaScript engine**. It allows running JavaScript code **outside the browser**, typically on the **server-side**.

Features:

- Non-blocking, event-driven I/O
- Ideal for building fast, scalable network apps (e.g., APIs)
- Uses single-threaded event loop for concurrency

1.2 Installing Node.js

Download and install from the official Node.js site.

Once installed, verify:

```
node -v       # Node.js version
npm -v        # npm version
```

1.3 Node.js REPL (Read-Eval-Print Loop)

What is REPL?

An interactive shell where you can:

- Type JavaScript expressions
- Evaluate them immediately

Try it:

```
node
> 5 + 3
8
> const name = "Alice";
> name.toUpperCase()
'ALICE'
> .exit
```

1.4 Command-Line Scripts

You can run a .js file using:

```
node app.js
```

Example: app.js

```
console.log("Welcome to Node.js!");
```

Run it:

```
node app.js
```

1.5 npm & package.json

What is npm?

npm (Node Package Manager) helps manage packages and project dependencies.

Initialize a project:

```
mkdir my-app
cd my-app
npm init -y     # creates package.json with default settings
```

Your package.json:

```
{
  "name": "my-app",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  }
}
```

1.6 Creating and Using Scripts

Example project file: index.js

```
console.log("Node app is running...");
```

Add custom scripts in package.json:

```
"scripts": {
  "start": "node index.js",
  "dev": "nodemon index.js"
}
```

Run them using:

```
npm start
npm run dev     # if nodemon installed
```

Install nodemon:

```
npm install --save-dev nodemon
```

1.7 Product vs. Development Dependencies

What are Dependencies?

Dependencies are packages or modules your project needs to function.

1. Product (Regular) Dependencies

Definition:
Packages that your app **needs to run in production**—that is, when it's deployed to users.

Example:

- express – used to build the server
- mongoose – used to connect to MongoDB

Install command:

```
npm install express
npm install mongoose
```

Saved in:

```
"dependencies": {
  "express": "^4.18.2",
  "mongoose": "^7.0.0"
}
```

2. Development Dependencies

Definition:
Packages only needed **during development**, such as tools for testing, linting, or restarting the server automatically.

Example:

- nodemon – automatically restarts server on file change
- eslint – linter for checking code quality

Install command:

```
npm install --save-dev nodemon
npm install --save-dev eslint
```

Saved in:

```
"devDependencies": {
  "nodemon": "^3.0.1",
  "eslint": "^8.56.0"
}
```

Scripts Example Using Both:

In package.json:

```
"scripts": {
  "start": "node index.js",
  "dev": "nodemon index.js"
}
```

- Use npm start for production
- Use npm run dev during development

Why the Difference Matters

- Production servers install **only dependencies** by default.
- Keeps production lightweight and secure.
- Avoids shipping unnecessary code (like testing or monitoring tools).

Try It Yourself:

1. Run:

```
npm install express
npm install --save-dev nodemon
```

2. Check your package.json. It should have two sections: dependencies and devDependencies.

1.7 Summary

Concept	Skill
Node.js CLI	Run JavaScript on server
REPL	Test code snippets live
npm	Manage project dependencies
Scripts	Automate tasks like start or dev

Unit 2: Express App & Routes

2.1 What is Express?

Definition:

Express.js is a **minimalist web application framework** for Node.js used to build fast, scalable, and modular web servers and APIs.

Why use Express?

- Simplifies routing compared to raw `http` module
- Built-in middleware and ability to use third-party middleware
- Works well with MongoDB, JWT, templating engines, etc.

Real-world analogy:

Think of Node.js as a **car engine**, and Express.js as the **steering wheel, pedals, and dashboard** — it makes controlling the car much easier!

2.2 Setup and Installation

Prerequisite: Initialized Node project

```
npm init -y
```

Install Express:

```
npm install express
```

2.3 Creating a Basic Express Server

File: index.js

```
const express = require('express');
const app = express(); // create express app

const PORT = 3000;

app.get('/', (req, res) => {
  res.send('Hello, Express!');
});

app.listen(PORT, () => {
  console.log(`Server is running at http://localhost:${PORT}`);
});
```

Explanation:

- express() creates an app instance
- app.get() defines a GET route
- app.listen() starts the server

Try:

```
node index.js
```

Then open browser at http://localhost:3000/

2.4 Understanding Routes

What is a Route?

A route defines how the server responds to a client request for a given endpoint and HTTP method.

Syntax:

```
app.METHOD(PATH, HANDLER);
```

Examples:

Method	Path	Description
GET	/about	Get static page or resource
POST	/register	Submit new data

2.5 GET and POST in Detail

GET Request Example

```
app.get('/greet', (req, res) => {
  res.send('Good day!');
});
```

Test in browser:

```
http://localhost:3000/greet
```

POST Request Example (with JSON body)

app.use(express.json()); // built-in body parser middleware

```
app.post('/submit', (req, res) => {
  const { name, age } = req.body;
  res.send(`Received: ${name}, Age: ${age}`);
});
```

Test with Postman:

- Method: POST
- URL: http://localhost:3000/submit
- Body (raw JSON):

```
{
  "name": "Sam",
  "age": 23
}
```

2.6 Using Route Parameters and Query Strings

Route Parameters

```
app.get('/user/:id', (req, res) => {
  res.send(`User ID: ${req.params.id}`);
});
```

Try in browser:

```
http://localhost:3000/user/101
```

Query Parameters

```
app.get('/search', (req, res) => {
  const { q } = req.query;
  res.send(`Search term: ${q}`);
});

Try in browser:

http://localhost:3000/search?q=express
```

2.7 Breakdown of an HTTP Request

Part	Description
URL	/user/42?q=profile
Method	GET, POST, etc.
Headers	Metadata like Content-Type, Authorization
Body	JSON or form data (in POST/PUT)
Params	/user/:id → id
Query	/user?q=abc → q

2.9 Testing Routes with Postman

Setup:

1. Open Postman
2. Select method: GET or POST
3. Enter URL: http://localhost:3000/your-endpoint
4. For POST: Select "Body → raw → JSON", enter payload

2.10 Summary

Term	Definition
Express	Web framework for Node.js
Route	URL + HTTP method combination
Middleware	Function that processes request before route
Request Params	Variables in URL path (:id)
Request Body	Data sent with POST/PUT request

Unit 3: Middleware in Express

3.1 What Is Middleware?

Simple Definition:

Middleware is just a **function** that runs **before your final route handler**. It processes requests, adds features, logs data, or checks permissions.

What Can Middleware Do?

- Log every request
- Parse incoming data (like JSON)
- Check if a user is logged in
- Block requests (e.g., rate limiting)
- Handle errors

3.2 Structure of Middleware Function

```
function myMiddleware(req, res, next) {
  // do something with req or res
  next(); // pass control to next middleware or route
}
```

Parameter	Description
req	Incoming request (data from client)
res	Response (data to send back)
next()	Function to continue to next middleware or route

How Middleware Works (Real-World Analogy)

Imagine an **airport security check**:

- You (the request) must pass multiple checkpoints.
- Each checkpoint is a middleware.
- If one checkpoint fails, you're stopped.
- If all succeed, you reach your gate (final route).

3.3 Code a Simple Express App with Middleware

Setup (if not already done)

```
npm init -y
npm install express
```

File: index.js

```
const express = require('express');
const app = express();
const PORT = 3000;

// Built-in middleware to parse JSON
app.use(express.json());
```

```
// Custom middleware
function logRequest(req, res, next) {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next(); // MUST call next()
}

app.use(logRequest); // apply globally

// Sample route
app.get('/', (req, res) => {
  res.send('Welcome to Express Middleware!');
});

app.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}`);
});
```

3.4 Built-in Middleware in Express

1. express.json()

- Parses JSON body from client request.
- Required when receiving POST/PUT JSON data.

```
app.use(express.json());
```

Needed when:

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

Example:

```
app.post('/data', (req, res) => {
  console.log(req.body); // logs parsed object
  res.send('Data received');
});
```

2. express.urlencoded()

Not needed for API development with JSON clients like Postman or React.

Use only if you're dealing with **HTML form submissions**:

```
<form method="POST" action="/submit">
  <input name="username" />
  <button>Send</button>
</form>
```

Then in Express:

```
app.use(express.urlencoded({ extended: true }));
```

3.5 Custom Middleware Examples

Example 1: Logger

```
function logger(req, res, next) {
  console.log(`Request: ${req.method} ${req.url}`);
  next();
}

app.use(logger);
```

Example 2: Add Timestamp to Request

```
function addTime(req, res, next) {
  req.requestTime = new Date().toISOString();
  next();
}

app.use(addTime);

app.get('/time', (req, res) => {
  res.send(`Requested at ${req.requestTime}`);
});
```

3.6 Order of Middleware Matters

```
app.use(middlewareA);
app.use(middlewareB);

middlewareA will run before middlewareB.
Always define middleware before your routes.
```

3.7 Middleware for Specific Routes

```
Apply to one route only:

function verifyUser(req, res, next) {
  if (req.query.admin === 'true') {
    next();
  } else {
    res.send('Access Denied');
  }
}

app.get('/admin', verifyUser, (req, res) => {
  res.send('Welcome Admin');
});
```

Try:

http://localhost:3000/admin?admin=true

Middleware Without next() = Broken Route

If you forget next(), your app **hangs forever**.

```
function badMiddleware(req, res, next) {
 console.log("Oops... forgot next");
 // next() is missing — the app freezes
}
```

3.8 Summary Table

Term	Description
middleware	Function that runs before final response
express.json()	Parses JSON bodies
next()	Passes control to next middleware/route
req, res	Request/Response objects
Custom middleware	Functions like logger, auth checker

Unit 4: Mongoose Setup & MongoDB Connection

4. 1 What is Mongoose?

Definition:

Mongoose is a Node.js **ODM (Object Data Modeling)** library that helps interact with MongoDB using **JavaScript objects**.

Why use Mongoose?

Without Mongoose:

```
db.collection('users').insertOne({ name: 'John' });
```

With Mongoose:

```
const user = new User({ name: 'John' });
user.save();
```

Mongoose simplifies:

- Schema definition (structure of your documents)
- Validations
- Query building
- Relationship between data (via refs)
- Middleware and hooks (advanced)

4.2 Setting Up Mongoose

Step 1: Install Mongoose

```
npm install mongoose
```

Step 2: Import and Connect

const mongoose = require('mongoose');

```
mongoose.connect('mongodb://localhost:27017/myapp')
  .then(() => console.log("Connected to MongoDB"))
  .catch((err) => console.error("Connection error:", err));
```

4.3 Connect to Local MongoDB

Make sure MongoDB is running locally (default port: 27017)

db.js

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

const connectDB = async () => {
  try {
    await mongoose.connect('mongodb://localhost:27017/mydb');
    console.log('MongoDB Connected');
  } catch (error) {
    console.error('Error connecting to MongoDB:', error);
    process.exit(1); // exit app on error
  }
};

module.exports = connectDB;
```

index.js

```
const express = require('express');
const app = express();
const connectDB = require('./db');

// Middleware
app.use(express.json());

// Connect to DB
connectDB();

// Simple route
app.get('/', (req, res) => {
  res.send('MongoDB is connected!');
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

4.4 Connect to MongoDB Atlas (Cloud)

Step-by-step:

1. Go to MongoDB Atlas
2. Create a **free cluster**
3. Add a **Database User** (with password)
4. Allow **IP Whitelist** (use 0.0.0.0/0 for development)
5. Copy the connection string, it looks like:

mongodb+srv://<username>:<password>@cluster0.abcd.mongodb.net/myDatabase?retryWrites=true&w=majority

Replace <username>, <password>, and myDatabase accordingly.

Connect Atlas in db.js

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

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGO_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log('Connected to MongoDB Atlas');
  } catch (err) {
    console.error('MongoDB Atlas Connection Error:', err);
    process.exit(1);
  }
};

module.exports = connectDB;
```

4.5 Use .env for Safe Credentials

Install dotenv

```
npm install dotenv
```

.env

```
MONGO_URI=mongodb+srv://user:password@cluster0.mongodb.net/myDatabase
```

index.js

```
require('dotenv').config(); // load .env variables
const connectDB = require('./db');
connectDB();
```

4.6 Organize Your Project

```
project/
├── db.js
├── models/
├── routes/
├── controllers/
├── .env
├── index.js
```

4.7 Common Errors & Fixes

Error	Likely Cause	Fix
Connection timeout	MongoDB not running	Start MongoDB locally
Auth error	Wrong username/password	Double-check Atlas URI
IP not allowed	IP not whitelisted	Add your IP to Atlas whitelist
Deprecation warnings	Old options missing	Use useNewUrlParser, useUnifiedTopology

4.8 Summary Table

Concept	Description
Mongoose	ODM library to work with MongoDB
mongoose.connect()	Connects to database
then()/catch() or async/await	Handle success/failure
MongoDB Atlas	Cloud version of MongoDB
.env	Stores sensitive credentials

Unit 5: Define Mongoose Schema & Models

5.1 What Is a Schema in Mongoose?

A **schema** defines the **structure** of documents in a MongoDB collection.

Like a **blueprint**:

- What fields exist
- What data type each field must be
- What rules to follow (required, min/max, etc.)

Example Schema

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

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

5.2 Your First Schema + Model

models/User.js

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

// Step 1: Define schema
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  age: Number
});

// Step 2: Create model
const User = mongoose.model('User', userSchema);

module.exports = User;
```

In index.js

```
const express = require('express');
const connectDB = require('./db');
const User = require('./models/User');

const app = express();
app.use(express.json());

connectDB();
```

```
app.post('/users', async (req, res) => {
  const user = new User(req.body);
  await user.save();
  res.json(user);
});
```

Now, try sending a POST request to /users with JSON:

```
{
  "name": "Aanya",
  "email": "aanya@gmail.com",
  "age": 22
}
```

5.3 Mongoose Schema Data Types

Type	Description
String	Text data
Number	Integer or float
Boolean	true or false
Date	Timestamp
Array	List of values
ObjectId	Reference to another document
Mixed	Any type (use sparingly)

5.4 Validations in Mongoose

Example with Validation

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true, minlength: 3 },
  email: { type: String, required: true, unique: true },
  age: { type: Number, min: 0, max: 100 }
});
```

Option	Meaning
required	Field must be present
unique	No duplicates allowed
min/max	For numbers
minlength	For strings
validate()	Custom validation logic

5.5 Auto Timestamps

Want to track **when a document was created or updated?**

Add this option to schema:

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

Now each document will have:

```
{
  "createdAt": "2025-06-19T14:22:00Z",
  "updatedAt": "2025-06-19T14:30:00Z"
}
```

5.6 Mongoose Model

A **Model** is a **constructor** that lets you interact with a collection using the schema.

Example:

```
const User = mongoose.model('User', userSchema);
```

Use it to:

- Create new documents: new User()
- Save to DB: .save()
- Query documents: User.find(), User.findById(), etc.

5.7 Custom Methods (Optional, Advanced)

You can define **methods** inside the schema:

```
userSchema.methods.greet = function() {
  return `Hello, ${this.name}!`;
};

const user = new User({ name: 'Kiran' });
console.log(user.greet()); // Hello, Kiran!
```

5.8 Summary Table

Concept	Purpose
Schema	Defines structure of documents
Model	Interface to work with a MongoDB collection

type	Data type of a field
required, unique	Validation options
timestamps: true	Adds createdAt, updatedAt
.methods	Add functions to documents

Unit 6: CRUD Routes & DB Operations

6.1 Real-Time Use Case: E-commerce Product Management

Entity: Product

Fields to manage:

- title (String, required)
- description (String)
- price (Number, required, non-negative)
- category (String)
- stock (Number)
- createdAt, updatedAt (auto timestamps)

Step 1: Define Mongoose Schema and Model

models/Product.js

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

const productSchema = new mongoose.Schema({
  title: { type: String, required: true },
  description: String,
  price: { type: Number, required: true, min: 0 },
  category: String,
  stock: { type: Number, default: 0 }
}, {
  timestamps: true
});

const Product = mongoose.model('Product', productSchema);

module.exports = Product;
```

Step 2: Setup Express Routes for CRUD

routes/productRoutes.js

```
const express = require('express');
const router = express.Router();
const Product = require('../models/Product');

// CREATE product (POST /products)
router.post('/', async (req, res) => {
  try {
    const newProduct = new Product(req.body);
    const saved = await newProduct.save();
    res.status(201).json(saved);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
```

```
});
```

```
// READ all products (GET /products)
router.get('/', async (req, res) => {
  try {
    const products = await Product.find();
    res.json(products);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

// READ one product by ID (GET /products/:id)
router.get('/:id', async (req, res) => {
  try {
    const product = await Product.findById(req.params.id);
    if (!product) return res.status(404).json({ error: 'Not found' });
    res.json(product);
  } catch (err) {
    res.status(500).json({ error: 'Invalid ID' });
  }
});

// UPDATE product (PUT /products/:id)
router.put('/:id', async (req, res) => {
  try {
    const updated = await Product.findByIdAndUpdate(
      req.params.id,
      req.body,
      { new: true, runValidators: true }
    );
    if (!updated) return res.status(404).json({ error: 'Product not found' });
    res.json(updated);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});

// DELETE product (DELETE /products/:id)
router.delete('/:id', async (req, res) => {
  try {
    const deleted = await Product.findByIdAndDelete(req.params.id);
    if (!deleted) return res.status(404).json({ error: 'Product not found' });
    res.json({ message: 'Product deleted successfully' });
  } catch (err) {
    res.status(500).json({ error: 'Invalid ID' });
  }
});

module.exports = router;
```

Step 3: Connect Routes to Main App

index.js

```
const express = require('express');
const connectDB = require('./db');
const productRoutes = require('./routes/productRoutes');

require('dotenv').config();
connectDB();

const app = express();
app.use(express.json());

app.use('/products', productRoutes);

app.listen(3000, () => {
  console.log('Server is running on http://localhost:3000');
});
```

Step 4: Try the API in Postman

Method	Route	Description
POST	/products	Add new product
GET	/products	Get all products
GET	/products/:id	Get one product
PUT	/products/:id	Update product
DELETE	/products/:id	Delete product

Sample POST Body:

```
{
  "title": "Red T-shirt",
  "description": "100% Cotton",
  "price": 499,
  "category": "Clothing",
  "stock": 20
}
```

6.2 Error Handling Examples

Case	Error
Missing title or price	400 Bad Request
Invalid ID	500 Internal Error
Product not found	404 Not Found

Best Practices in Error Handling

We use:

- try/catch blocks for **async/await**
- runValidators: true during **update**
- Appropriate **HTTP status codes**
- Clear **error messages** for clients

Updated CRUD Code with Error Handling

routes/productRoutes.js

```
const express = require('express');
const router = express.Router();
const Product = require('../models/Product');

// CREATE Product
router.post('/', async (req, res) => {
  try {
    const { title, price } = req.body;

    // Check required fields manually (extra protection)
    if (!title || price == null) {
      return res.status(400).json({ error: 'Title and price are required.' });
    }

    const newProduct = new Product(req.body);
    const saved = await newProduct.save();
    res.status(201).json(saved);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});

// GET All Products
router.get('/', async (req, res) => {
  try {
    const products = await Product.find();
    res.json(products);
  } catch (err) {
    res.status(500).json({ error: 'Server error. Please try again later.' });
  }
});

// GET Product by ID
router.get('/:id', async (req, res) => {
  try {
    const product = await Product.findById(req.params.id);

    if (!product) {
      return res.status(404).json({ error: 'Product not found.' });
    }

    res.json(product);
  } catch (err) {
```

```
    res.status(500).json({ error: 'Invalid product ID format.' });
  }
});

// UPDATE Product
router.put('/:id', async (req, res) => {
  try {
    const updated = await Product.findByIdAndUpdate(
      req.params.id,
      req.body,
      {
        new: true,            // return updated document
        runValidators: true   // validate new data
      }
    );

    if (!updated) {
      return res.status(404).json({ error: 'Product not found.' });
    }

    res.json(updated);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});

// DELETE Product
router.delete('/:id', async (req, res) => {
  try {
    const deleted = await Product.findByIdAndDelete(req.params.id);

    if (!deleted) {
      return res.status(404).json({ error: 'Product not found.' });
    }

    res.json({ message: 'Product deleted successfully.' });
  } catch (err) {
    res.status(500).json({ error: 'Invalid product ID format.' });
  }
});

module.exports = router;
```

Summary of HTTP Status Codes Used

Status Code	Meaning	When It's Used
201	Created	After successful POST
400	Bad Request	Missing/invalid input
404	Not Found	Product ID doesn't exist
500	Server Error	Invalid ID format or database failure

Common Mistakes to avoid

- Omitting runValidators in PUT — causes invalid data to be saved
- Not checking null result after findById*() — leads to false success
- Trying to save incomplete data — leads to validation error

6.3 Optional Enhancements

- **Search by category**: GET /products?category=Electronics
- **Filter by price**: GET /products?min=100&max=500
- **Pagination**: Add limit and page query params

Project Structure (Recommended)

```
project/
├── controllers/
│   └── productController.js
├── routes/
│   └── productRoutes.js
├── models/
│   └── Product.js
├── app.js
└── ...
```

controllers/productController.js

```
const Product = require('../models/Product');

// @desc    Get all products with optional filters
// @route   GET /products
// @access  Public
const getAllProducts = async (req, res) => {
  try {
    const { category, min, max, page = 1, limit = 10 } = req.query;

    const filter = {};

    // 🔍 Filter by category
    if (category) {
      filter.category = category;
    }

    // 💰 Filter by price range
    if (min !== undefined || max !== undefined) {
      filter.price = {};
      if (min !== undefined) filter.price.$gte = parseFloat(min);
      if (max !== undefined) filter.price.$lte = parseFloat(max);
    }

    // 📄 Pagination
    const pageNumber = parseInt(page);
    const limitNumber = parseInt(limit);
    const skip = (pageNumber - 1) * limitNumber;
```

```
    // 🧠 Query DB with filters and pagination
    const products = await Product.find(filter)
      .skip(skip)
      .limit(limitNumber);

    const total = await Product.countDocuments(filter);

    res.json({
      total,
      currentPage: pageNumber,
      totalPages: Math.ceil(total / limitNumber),
      products
    });

  } catch (error) {
    res.status(500).json({
      error: 'Failed to fetch products',
      details: error.message
    });
  }
};

module.exports = {
  getAllProducts,
};
```

routes/productRoutes.js

```
const express = require('express');
const router = express.Router();
const { getAllProducts } = require('../controllers/productController');

// GET /products with filters
router.get('/', getAllProducts);

module.exports = router;
```

models/Product.js (Sample Model)

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

const productSchema = new mongoose.Schema({
  title: {
    type: String,
    required: true,
  },
  price: {
    type: Number,
    required: true,
  },
  category: String,
}, { timestamps: true });

module.exports = mongoose.model('Product', productSchema);
```

app.js or server.js (Basic Setup)

```
const express = require('express');
const mongoose = require('mongoose');
const productRoutes = require('./routes/productRoutes');

const app = express();

app.use(express.json());
app.use('/products', productRoutes);

mongoose.connect('your_mongo_uri')
  .then(() => {
    console.log('MongoDB connected');
    app.listen(3000, () => console.log('Server running on port 3000'));
  })
  .catch(err => console.error('MongoDB error:', err));
```

Sample Request

```
GET /products?category=Shoes&min=500&max=1500&page=2&limit=5
```

Returns:

```
{
  "total": 22,
  "currentPage": 2,
  "totalPages": 5,
  "products": [ ... ]
}
```

6.4 Summary Table

Operation	Method	Route	Mongoose Function
Create	POST	/products	new Product(), .save()
Read All	GET	/products	Product.find()
Read One	GET	/products/:id	Product.findById()
Update	PUT	/products/:id	Product.findByIdAndUpdate()
Delete	DELETE	/products/:id	Product.findByIdAndDelete()

6.5 Recommended Folder Structure

```
project/
├── models/
│   └── Product.js
├── routes/
│   └── productRoutes.js
├── controllers/      <-- Optional: separate route logic
├── db.js
├── .env
├── index.js
```

Unit 7: API Testing with Postman

7.1 API Routes to Test

Method	Endpoint	Purpose
GET	/products	Fetch all products
GET	/products/:id	Get one product
POST	/products	Create new product
PUT	/products/:id	Update a product
DELETE	/products/:id	Delete a product

Test GET /products

Steps

- Open Postman
- Select **GET** method
- Enter URL: http://localhost:3000/products
- Click **Send**

Sample Response

```
[
  {
    "_id": "6661abc12345",
    "title": "Sneakers",
    "price": 799,
    "category": "Shoes"
  },
  ...
]
```

Test POST /products

Steps

- Select **POST**
- URL: http://localhost:3000/products
- Go to **Body** → **raw** → select **JSON**
- Paste sample product:

```
{
  "title": "T-shirt",
  "price": 499,
  "category": "Clothing"
}
```

- Click **Send**

Expected

- Returns 201 Created
- Response contains _id and product details

Test GET /products/:id

Steps

- Copy a valid product _id from previous response
- Select **GET**
- URL: http://localhost:3000/products/6661abc12345
- Click **Send**

If ID is invalid:

```
{
  "error": "Invalid product ID format."
}
```

Test PUT /products/:id

Steps

- Select **PUT**
- URL: http://localhost:3000/products/6661abc12345
- Body (JSON):

```
{
  "price": 549
}
```

- Click **Send**

Expected

- Returns updated product
- Validates fields using runValidators

Test DELETE /products/:id

Steps

- Select **DELETE**
- URL: http://localhost:3000/products/6661abc12345
- Click **Send**

Expected

```
{
  "message": "Product deleted successfully."
}
```

HTTP Status Codes Recap

Code	Meaning	When It's Used
200	OK	GET success
201	Created	POST success
400	Bad Request	Missing/invalid input
404	Not Found	No product with given ID
500	Server Error	Invalid ID or DB issues

Unit 8: REST API Design & Endpoints

8.1 What is REST?

REST (REpresentational State Transfer) is an architectural style for designing web APIs. A REST API allows systems to communicate over HTTP using standardized principles.

Key REST Principles

Principle	Description
Stateless	No client context is stored on the server. Each request must contain all necessary information.
Client-Server	The client and server operate independently. Server focuses on data, client on UI.
Uniform Interface	Consistent resource access using a common structure (URI + HTTP method).
Cacheable	Responses can be cached to improve performance.
Layered System	You can add middle layers (e.g., proxy, auth) without client knowing.

HTTP Methods (Verbs)

Verb	Purpose	Idempotent?	Example
GET	Read data	Yes	GET /products
POST	Create data	No	POST /products
PUT	Update entire resource	Yes	PUT /products/123
PATCH	Partial update	Yes	PATCH /products/123
DELETE	Delete resource	Yes	DELETE /products/123

8.2 RESTful URI Design

General Rules

1. **Use nouns** in URIs (not verbs):
   /getAllProducts → /products
2. **Use plural forms**:
   /users not /user
3. **Use nested resources** for relationships:
   /users/1/orders → Get orders for user 1
4. **Use query parameters** for filters/search:
   /products?category=clothing&min=100&max=500

Example: Product API Endpoints

Method	Endpoint	Purpose
GET	/products	Get all products
GET	/products/:id	Get product by ID
POST	/products	Create new product
PUT	/products/:id	Replace product
PATCH	/products/:id	Modify some fields
DELETE	/products/:id	Delete product

Statelessness in REST

- Each request must be independent.
- Server **doesn't store session** data (unlike traditional login sessions).
- Clients send all required data in every request (e.g., via headers or tokens).

Unit 9: Status Codes & Input Validation

9.1. HTTP Status Codes – What & Why?

HTTP status codes indicate the **result of an API call**.

Code	Meaning	When to Use
200 OK	Success	Resource fetched or action completed
201 Created	Created	New resource created (POST /products)
204 No Content	Success, no response	Successful deletion or update with no response body
400 Bad Request	Client error	Missing or invalid input
401 Unauthorized	Invalid auth token	User not logged in or JWT invalid
403 Forbidden	Authenticated, but not allowed	User lacks permission (e.g., role restrictions)
404 Not Found	Resource missing	Non-existent product/user
500 Internal Server Error	Code crash	Unhandled exception, server problem

9.2. What is express-validator?

express-validator is a middleware for validating and sanitizing user input in Express apps.

🔧 **Install it**:

```
npm install express-validator
```

📦 **Import tools from the package**:

```
const { check, validationResult } = require('express-validator');
```

9.3. Basic Validation Example: Create Product API

```
// routes/productRoutes.js
const express = require('express');
const { check, validationResult } = require('express-validator');
const router = express.Router();
const productController = require('../controllers/productController');

router.post(
  '/products',
  [
    check('title', 'Title is required').not().isEmpty(),
    check('price', 'Price must be a number').isFloat({ gt: 0 }),
    check('category', 'Category must be at least 3 chars').isLength({ min: 3 })
  ],
  productController.createProduct
);
```

```
module.exports = router;
// controllers/productController.js
exports.createProduct = async (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({
      success: false,
      message: 'Invalid input',
      errors: errors.array(),
    });
  }

  // Proceed with DB save...
  res.status(201).json({ success: true, message: 'Product created' });
};
```

9.4. Common Input Validation Scenarios

Field	Rule	Example
email	isEmail()	Valid email format
password	isLength({ min: 6 })	Minimum password length
price	isFloat({ gt: 0 })	Price must be a positive number
category	optional().isLength({ min: 3 })	Optional field with minimum 3 characters
id	isMongoId()	Validate MongoDB ObjectId

9.5. Secure Your API Input

- Sanitize inputs: .trim().escape() to prevent injection.
- Never trust incoming data blindly.
- Use schema-level validation as a second safety net (Mongoose).

9.6. Consistent Error Format

Standardize validation errors so frontend knows how to handle them:

```
{
  "success": false,
  "message": "Validation failed",
  "errors": [
    {
      "msg": "Price must be a number",
      "param": "price",
      "location": "body"
    }
  ]
}
```

Unit 10: Error Handling & Resilience

10.1 Why Error Handling Matters

- **Avoid Crashes**: Unhandled errors may bring down the server.
- **Improve Developer UX**: Clear error messages help debugging.
- **Improve Client UX**: Frontend apps can handle and show user-friendly messages.
- **Keep Logic DRY**: One centralized handler avoids duplication.

10.2. Basic try-catch Usage in Controllers

Wrap your controller logic in try/catch to safely catch runtime and DB errors:

```
// controllers/productController.js
exports.getProductById = async (req, res, next) => {
  try {
    const product = await Product.findById(req.params.id);

    if (!product) {
      return res.status(404).json({ success: false, message: 'Product not found' });
    }

    res.status(200).json({ success: true, data: product });

  } catch (error) {
    next(error); // forward error to centralized error handler
  }
};
```

10.3. Centralized Error Middleware

Place this middleware **after all routes** in app.js:

```
// middlewares/errorHandler.js
module.exports = (err, req, res, next) => {
  console.error(err.stack);

  const statusCode = res.statusCode === 200 ? 500 : res.statusCode;

  res.status(statusCode).json({
    success: false,
    message: err.message || 'Internal Server Error',
    stack: process.env.NODE_ENV === 'production' ? undefined : err.stack,
  });
};
// In app.js or server.js
const errorHandler = require('./middlewares/errorHandler');
app.use(errorHandler); // after all routes
```

10.4. Create a Custom Error Class

Optional but **professional**: Create your own AppError class.

```
// utils/AppError.js
class AppError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;

    Error.captureStackTrace(this, this.constructor);
  }
}
```

module.exports = AppError;

Usage Example:

```
const AppError = require('../utils/AppError');

if (!product) {
  throw new AppError('Product not found', 404);
}
```

Now the centralized error handler can use err.statusCode directly.

10.5. Mongoose-Specific Errors

- CastError (invalid ID format)
- ValidationError (schema-level failures)
- Use logic to catch and customize these

```
if (err.name === 'CastError') {
  message = 'Invalid ID format';
  statusCode = 400;
}
```

You can add this logic in your error handler.

10.6. Final Error Response Format

Standard structure to maintain consistency:

```
{
  "success": false,
  "message": "Product not found",
  "stack": "..."  // optional in dev mode
}
```

Unit 11: Project Structure & Modularity

11.1. Why Structure Matters

- Keeps code **organized and maintainable**
- Enables team collaboration
- Eases debugging and testing
- Promotes **separation of concerns** (e.g., routes ≠ business logic)

11.2. Standard Project Structure

```
project-root/
│
├── controllers/        → All business logic
│   ├── productController.js
│   └── userController.js
│
├── routes/             → Route definitions
│   ├── productRoutes.js
│   └── userRoutes.js
│
├── models/             → Mongoose schemas
│   ├── Product.js
│   └── User.js
│
├── middlewares/        → Custom middleware & error handlers
│   └── errorHandler.js
│
├── config/             → Configuration files (DB, JWT, etc.)
│   └── db.js
│
├── utils/              → Helper functions, error classes, etc.
│   └── AppError.js
│
├── .env                → Environment variables
├── app.js              → Express setup & middleware
├── server.js           → Starts the server
└── package.json
```

11.3. server.js vs app.js

File	Responsibility
app.js	Create Express app, use middleware, attach routes
server.js	Import app.js and listen on a port (starts the server)

Example server.js:

```
const app = require('./app');
const dotenv = require('dotenv');

dotenv.config();

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

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

Example app.js:

```
const express = require('express');
const app = express();
const productRoutes = require('./routes/productRoutes');
const userRoutes = require('./routes/userRoutes');
const errorHandler = require('./middlewares/errorHandler');

// Built-in middleware
app.use(express.json());

// API routes
app.use('/api/products', productRoutes);
app.use('/api/users', userRoutes);

// Error handling middleware
app.use(errorHandler);

module.exports = app;
```

11.4. Controller Example

```
// controllers/productController.js
const Product = require('../models/Product');

exports.getAllProducts = async (req, res, next) => {
  try {
    const products = await Product.find();
    res.status(200).json({ success: true, data: products });
  } catch (err) {
    next(err);
  }
};
```

11.5. Route File Example

```
// routes/productRoutes.js
const express = require('express');
const router = express.Router();
const productController = require('../controllers/productController');

router.get('/', productController.getAllProducts);

module.exports = router;
```

11.6. Mongoose Model Example

```
// models/Product.js
const mongoose = require('mongoose');

const productSchema = new mongoose.Schema({
  title: {
    type: String,
    required: [true, 'Product title is required'],
  },
  price: {
    type: Number,
    required: true,
    min: [1, 'Price must be above zero'],
  },
  category: String,
});

module.exports = mongoose.model('Product', productSchema);
```

11.7. Configuring Environment Variables

.env:

```
PORT=5000
MONGODB_URI=mongodb+srv://yourdb.mongodb.net/mydb
JWT_SECRET=supersecretkey
```

Use with dotenv:

```
require('dotenv').config();
const db = process.env.MONGODB_URI;
```

11.8. Middleware File Example

```
// middlewares/errorHandler.js
module.exports = (err, req, res, next) => {
  const status = err.statusCode || 500;
  res.status(status).json({
    success: false,
    message: err.message || 'Internal Server Error',
  });
};
```

Unit 12: REST API Design & Best Practices

12.1. Standard API Response Format

Instead of sending raw objects or arrays, define a **standard shape** for all API responses:

Example

```
{
  "success": true,
  "message": "Product fetched successfully",
  "data": {
    "title": "Laptop",
    "price": 800
  }
}
```

Benefits

- Easy for frontend to parse
- Unified error/success messaging
- Consistent structure improves debugging and logging

Code Implementation

In controller:

```
res.status(200).json({
  success: true,
  message: 'Product fetched successfully',
  data: product,
});
```

In error handler:

```
res.status(err.statusCode || 500).json({
  success: false,
  message: err.message || 'Something went wrong',
});
```

12.2. API Versioning

Include version numbers in route prefixes:

```
app.use('/api/v1/products', productRoutes);
app.use('/api/v1/users', userRoutes);
```

Why it matters:

- Enables smooth updates without breaking old clients
- Can run /v1, /v2 side-by-side if needed

12.3. CORS (Cross-Origin Resource Sharing)

Helps your API accept requests from other domains (like React/Vue frontend).

Install & Use

```
npm install cors
const cors = require('cors');
app.use(cors()); // allow all origins by default
```

You can also restrict origins:

```
app.use(cors({ origin: 'https://myfrontend.com' }));
```

12.4. Handle Unknown Routes (404)

Catch-all route should be placed **after all other routes**:

```
app.all('*', (req, res) => {
  res.status(404).json({
    success: false,
    message: `Route ${req.originalUrl} not found`,
  });
});
```

Ensures all invalid API calls return a clean 404 response.

12.5. Rate Limiting (Optional)

Use to prevent abuse or DoS attacks.

Install:

```
npm install express-rate-limit
```

Setup:

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

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // limit each IP
  message: 'Too many requests, please try again later.',
});

app.use(limiter);
```

12.6. Using Postman Collections & Environments

Example:

Set environment:

```
base_url = http://localhost:5000/api/v1
token = your_jwt_token
```

Use in requests:

```
{{base_url}}/products
Authorization: Bearer {{token}}
```

Bonus: Status Code Summary Table

Status	Use Case
200	OK – data successfully fetched
201	Created – new resource added
400	Bad Request – invalid input
401	Unauthorized – token missing/invalid
403	Forbidden – insufficient permissions
404	Not Found – resource doesn't exist
500	Internal Server Error – something failed

Unit 13 : User Authentication – Register/Login

13.1. Introduction to User Authentication

Authentication is the process of verifying a user's identity.

Real-world example:

When you log in to Amazon, the system checks whether your email and password match what's stored in their database.

Tools & Libraries

Tool	Purpose
bcryptjs	Hashing passwords before storing them
express-validator	Validating user inputs (email, password)
mongoose	Storing and querying user data in MongoDB
dotenv	Securely storing secrets like DB connection strings

13.2. Project Structure (Simplified)

```
/auth-app
  ├── models
  │    └── User.js
  ├── routes
  │    └── auth.js
  ├── .env
  ├── server.js
  └── package.json
```

Mongoose User Model

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

const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, "Name is required"],
    minlength: 3,
  },
  email: {
    type: String,
    required: [true, "Email is required"],
    unique: true,
    match: [/\S+@\S+\.\S+/, "Email format is invalid"],
  },
  password: {
    type: String,
    required: [true, "Password is required"],
    minlength: 6,
```

```
  }
}, { timestamps: true });

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

Hashing Passwords with bcryptjs

```
npm install bcryptjs
```

Hashing During Registration

```
const bcrypt = require("bcryptjs");

// Inside register controller
const salt = await bcrypt.genSalt(10);
const hashedPassword = await bcrypt.hash(password, salt);
```

Password Comparison on Login

```
const isMatch = await bcrypt.compare(inputPassword, user.password);
```

Register Route (with Validation)

```
// routes/auth.js
const express = require("express");
const { body, validationResult } = require("express-validator");
const bcrypt = require("bcryptjs");
const User = require("../models/User");

const router = express.Router();

router.post(
  "/register",
  [
    body("name").notEmpty().withMessage("Name is required"),
    body("email").isEmail().withMessage("Enter a valid email"),
    body("password").isLength({ min: 6 }).withMessage("Password must be 6+ chars")
  ],
  async (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });

    const { name, email, password } = req.body;

    try {
      const existingUser = await User.findOne({ email });
      if (existingUser) return res.status(400).json({ msg: "Email already registered"
});

      const salt = await bcrypt.genSalt(10);
      const hashedPassword = await bcrypt.hash(password, salt);

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

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

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

Login Route (Verify Credentials)

```
// routes/auth.js (continued)
router.post(
  "/login",
  [
    body("email").isEmail().withMessage("Enter valid email"),
    body("password").exists().withMessage("Password required"),
  ],
  async (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });

    const { email, password } = req.body;

    try {
      const user = await User.findOne({ email });
      if (!user) return res.status(400).json({ msg: "Invalid email or password" });
```

```
      const isMatch = await bcrypt.compare(password, user.password);
      if (!isMatch) return res.status(400).json({ msg: "Invalid email or password" });

      res.status(200).json({ msg: "Login successful", user: { name: user.name, email:
user.email } });
    } catch (err) {
      res.status(500).json({ msg: "Server error", error: err.message });
    }
  }
);
```

13.3. Testing with Postman

Register Request

- Method: POST
- URL: http://localhost:5000/api/auth/register
- Body: raw > JSON

```
{
  "name": "Alice",
  "email": "alice@example.com",
  "password": "securepass123"
}
```

Login Request

- Method: POST
- URL: http://localhost:5000/api/auth/login

```
{
  "email": "alice@example.com",
  "password": "securepass123"
}
```

13. 4. Summary

Concept	Outcome
Hashing	Secure passwords before storing
Validation	Prevents invalid input
Error handling	Send proper HTTP responses
Credential checks	Ensure login safety

Unit 14: JWT Token Authentication

14.1. What is JWT?

A **JWT (JSON Web Token)** is a signed token that stores user info. Commonly used for stateless authentication.

Structure:

```
header.payload.signature
```

14.2. JWT Implementation

Setup: Install JSON Web Token

```
npm install jsonwebtoken cookie-parser
```

In your server.js or app.js:

```
const cookieParser = require('cookie-parser');
app.use(cookieParser());
```

Generate Token on Login and Set Cookie

In authController.js:

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

// helper to create token
const createToken = (user) => {
  return jwt.sign(
    { userId: user._id, role: user.role }, // payload
    process.env.JWT_SECRET,
    { expiresIn: '1d' }
  );
};
```

```
const login = async (req, res) => {
  const { email, password } = req.body;

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

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

  const token = createToken(user);

  // Set token in HTTP-only cookie
  res.cookie('token', token, {
    httpOnly: true,
    secure: process.env.NODE_ENV === 'production', // only over HTTPS in production
```

```
    sameSite: 'Strict',
    maxAge: 24 * 60 * 60 * 1000, // 1 day
  });
```

```
  res.status(200).json({ message: 'Login successful' });
};
```

Read Token from Cookie in Middleware

verifyToken.js middleware:

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

const verifyToken = (req, res, next) => {
  const token = req.cookies.token;

  if (!token) {
    return res.status(401).json({ message: 'Unauthorized: No token' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded; // attach user info to request
    next();
  } catch (err) {
    res.status(401).json({ message: 'Unauthorized: Invalid token' });
  }
};

module.exports = verifyToken;
```

Logout: Clear the Cookie

```
const logout = (req, res) => {
  res.clearCookie('token', {
    httpOnly: true,
    secure: process.env.NODE_ENV === 'production',
    sameSite: 'Strict',
  });
  res.json({ message: 'Logged out successfully' });
};
```

14.3. Testing in Postman

Login

- Method: POST /api/auth/login
- Body: raw JSON

```
{
  "email": "test@example.com",
  "password": "password123"
}
```

- Postman will automatically store cookies

Access Protected Route

- Use GET /api/user/dashboard
- Must include verifyToken middleware
- No need to manually set Authorization header
- Go to Postman → Cookies tab → Check token is stored

Logout

- Method: POST /api/auth/logout
- Clears the HTTP-only cookie

14.4. Why Use HTTP-only Cookies?

Storage Method	Vulnerable to XSS?	Can be Accessed by JS?	Secure?
localStorage	Yes	Yes	No
httpOnly cookie	No	No	Yes

14.5. Summary

Action	Code Snippet
Generate JWT	jwt.sign({userId}, secret, {expiresIn})
Store token	res.cookie('token', token, { httpOnly: true })
Read token	req.cookies.token
Logout	res.clearCookie('token')

Unit 15: Auth Middleware & Access Control

15.1. What is Middleware?

In Express, **middleware** is a function that has access to:

(req, res, next)

It can:

- Inspect/modify the request or response
- End the request-response cycle
- Call next() to pass control to the next middleware/handler

Example middleware:

```
const logger = (req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next(); // Pass control
};
```

Use it globally:

```
app.use(logger);
```

Authentication Middleware

We already created verifyToken in Unit 2. Let's review it:

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

const verifyToken = (req, res, next) => {
  const authHeader = req.headers.authorization;

  if (!authHeader || !authHeader.startsWith('Bearer ')) {
    return res.status(401).json({ message: 'Unauthorized: Token missing' });
  }

  const token = authHeader.split(' ')[1];

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded; // attach user data to request
    next();
  } catch (err) {
    res.status(401).json({ message: 'Unauthorized: Invalid token' });
  }
};

module.exports = verifyToken;
```

Applying Middleware to Routes

```
const verifyToken = require('../middleware/verifyToken');

router.get('/dashboard', verifyToken, (req, res) => {
  res.json({ message: `Welcome, user ${req.user.userId}` });
});
```

Only logged-in users with a valid token can access this route.

15.2 Optional: Role-Based Access

If your users have roles like admin, seller, customer, you can protect routes accordingly.

1. Add a role field in your user model:

```
role: {
  type: String,
  enum: ['user', 'admin'],
  default: 'user',
}
```

2. Add role info in token:

```
const token = jwt.sign(
  { userId: user._id, role: user.role },
  process.env.JWT_SECRET,
  { expiresIn: '1d' }
);
```

3. Create role-check middleware:

```
const restrictTo = (...roles) => {
  return (req, res, next) => {
    if (!roles.includes(req.user.role)) {
      return res.status(403).json({ message: 'Access denied' });
    }
    next();
  };
};

module.exports = restrictTo;
```

4. Use in route:

```
const restrictTo = require('../middleware/restrictTo');

router.delete('/admin/delete-user/:id', verifyToken, restrictTo('admin'), async (req,
res) => {
  // Only admins can delete users
});
```

15.3. Testing Access with Postman

1. Login as a normal user and as an admin.
2. Try to access an **admin-only** route with a **normal user token**.
   It should return **403 Forbidden**.
3. Try again with **admin token**.
   It should allow access.

15.4. Common Access Scenarios

Use Case	Middleware Chain
Authenticated users only	verifyToken
Admin-only route	verifyToken, then restrictTo('admin')
Logged-in users + specific role	verifyToken, then restrictTo('seller')

15.5 Middleware Flow Recap

Request → verifyToken → restrictTo → Controller logic

Middleware stack ensures:

- Authentication is checked first
- Authorization (who can access) is checked next
- Finally, the actual route logic runs

15.6. Summary

Concept	Middleware Name	Use
Check login/token	verifyToken	All secure routes
Limit access by role	restrictTo(...)	Admin or custom access
General purpose logging	logger	Dev/debug

Unit 4: Final Project – Setup (Scaffold Secure CRUD API)

Objective:

Build a **secure, modular Express + MongoDB API** where:

- Users can register/login
- Only authenticated users can manage products
- Code is organized into folders (routes, controllers, models, middleware)
- Uses HTTP-only JWT-based auth
- Is scalable and clean for teams

16.1. Project Folder Structure

Organize the backend like this:

```
secure-api-project/
├── controllers/
│   ├── authController.js
│   └── productController.js
├── models/
│   ├── User.js
│   └── Product.js
├── routes/
│   ├── authRoutes.js
│   └── productRoutes.js
├── middleware/
│   └── verifyToken.js
├── config/
│   └── db.js
├── .env
├── server.js
├── package.json
```

Install Dependencies

npm install express mongoose bcryptjs jsonwebtoken dotenv cookie-parser express-validator

Setup server.js

```
const express = require('express');
const mongoose = require('mongoose');
const cookieParser = require('cookie-parser');
const dotenv = require('dotenv');
dotenv.config();

const authRoutes = require('./routes/authRoutes');
const productRoutes = require('./routes/productRoutes');
const connectDB = require('./config/db');
```

```
const app = express();
app.use(express.json());
app.use(cookieParser());

connectDB();

app.use('/api/auth', authRoutes);
app.use('/api/products', productRoutes);

app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ message: 'Server Error' });
});

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

config/db.js – MongoDB Connection

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

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGO_URI);
    console.log('MongoDB connected');
  } catch (err) {
    console.error(err.message);
    process.exit(1);
  }
};

module.exports = connectDB;
```

.env Sample

```
PORT=5000
MONGO_URI=mongodb://localhost:27017/secure-crud
JWT_SECRET=yourSuperSecretKey
NODE_ENV=development
```

Modular Route Setup

```
routes/authRoutes.js

const express = require('express');
const { register, login, logout } = require('../controllers/authController');
const router = express.Router();

router.post('/register', register);
router.post('/login', login);
router.post('/logout', logout);

module.exports = router;

routes/productRoutes.js
```

```
const express = require('express');
const { createProduct, getProducts, updateProduct, deleteProduct } =
require('../controllers/productController');
const verifyToken = require('../middleware/verifyToken');
```

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

router.use(verifyToken); // all routes below this are protected

router.post('/', createProduct);
router.get('/', getProducts);
router.put('/:id', updateProduct);
router.delete('/:id', deleteProduct);

module.exports = router;
```

Empty Controllers for Now

```
controllers/authController.js

exports.register = (req, res) => {
  res.send('Register');
};
exports.login = (req, res) => {
  res.send('Login');
};
exports.logout = (req, res) => {
  res.send('Logout');
};

controllers/productController.js

exports.createProduct = (req, res) => {
  res.send('Create product');
};
exports.getProducts = (req, res) => {
  res.send('List products');
};
exports.updateProduct = (req, res) => {
  res.send('Update product');
};
exports.deleteProduct = (req, res) => {
  res.send('Delete product');
};
```

Reuse Middleware from Unit 2

```
middleware/verifyToken.js

const jwt = require('jsonwebtoken');

const verifyToken = (req, res, next) => {
  const token = req.cookies.token;
  if (!token) return res.status(401).json({ message: 'Unauthorized' });

  try {
```

```
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch {
    res.status(401).json({ message: 'Invalid Token' });
  }
};

module.exports = verifyToken;
```

16.2. Summary

You now have a **ready-to-build scaffolded secure CRUD API**:

- Uses modular structure
- Protects routes with token middleware
- Serves as a base for building real CRUD logic

Unit 17: Final Project – Logic

17.1. Auth Controller Logic (controllers/authController.js)

```
const User = require('../models/User');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const { validationResult } = require('express-validator');

// REGISTER
exports.register = async (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });

  const { name, email, password } = req.body;
  try {
    const existing = await User.findOne({ email });
    if (existing) return res.status(400).json({ message: 'Email already registered' });

    const hashed = await bcrypt.hash(password, 10);
    const user = await User.create({ name, email, password: hashed });

    res.status(201).json({ message: 'User created', user: { id: user._id, email:
user.email } });
  } catch (err) {
    res.status(500).json({ message: 'Server Error' });
  }
};

// LOGIN
exports.login = async (req, res) => {
  const { email, password } = req.body;
  try {
    const user = await User.findOne({ email });
    if (!user) return res.status(401).json({ message: 'Invalid credentials' });

    const match = await bcrypt.compare(password, user.password);
    if (!match) return res.status(401).json({ message: 'Invalid credentials' });

    const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, { expiresIn: '1h'
});
    res.cookie('token', token, { httpOnly: true, secure: false }); // secure: true for
HTTPS
    res.status(200).json({ message: 'Login successful' });
  } catch {
    res.status(500).json({ message: 'Server Error' });
  }
};

// LOGOUT
exports.logout = (req, res) => {
  res.clearCookie('token');
  res.json({ message: 'Logged out' });
};
```

17.2. Product Controller Logic (controllers/productController.js)

```
const Product = require('../models/Product');
const { validationResult } = require('express-validator');

// CREATE
exports.createProduct = async (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });

  try {
    const { title, price, category } = req.body;
    const product = await Product.create({ title, price, category, user: req.user.id
});
    res.status(201).json({ message: 'Product created', product });
  } catch {
    res.status(500).json({ message: 'Server Error' });
  }
};

// READ
exports.getProducts = async (req, res) => {
  try {
    const products = await Product.find({ user: req.user.id });
    res.json(products);
  } catch {
    res.status(500).json({ message: 'Server Error' });
  }
};
```

```
// UPDATE
exports.updateProduct = async (req, res) => {
  try {
    const { id } = req.params;
    const product = await Product.findOne({ _id: id, user: req.user.id });

    if (!product) return res.status(404).json({ message: 'Product not found' });

    Object.assign(product, req.body);
    await product.save();
    res.json({ message: 'Product updated', product });
  } catch {
    res.status(500).json({ message: 'Server Error' });
  }
};
```

```
// DELETE
exports.deleteProduct = async (req, res) => {
  try {
    const product = await Product.findOneAndDelete({ _id: req.params.id, user:
req.user.id });
    if (!product) return res.status(404).json({ message: 'Product not found' });

    res.json({ message: 'Product deleted' });
  } catch {
    res.status(500).json({ message: 'Server Error' });
  }
};
```

17.3. Input Validation with express-validator

Add to routes:

routes/authRoutes.js

```
const { body } = require('express-validator');

router.post('/register', [
  body('name').notEmpty(),
  body('email').isEmail(),
  body('password').isLength({ min: 6 }),
], register);

router.post('/login', [
  body('email').isEmail(),
  body('password').notEmpty(),
], login);

routes/productRoutes.js

const { body } = require('express-validator');

router.post('/', [
  body('title').notEmpty(),
  body('price').isFloat({ gt: 0 }),
  body('category').notEmpty(),
], createProduct);
```

17.4. Summary

You now have a **fully functional, secure backend API** with:

- ✓ Register/Login with hashed passwords
- ✓ JWT auth stored in HTTP-only cookies
- ✓ User-protected CRUD product logic
- ✓ Validations and proper status code handling

Unit 18: Final Project – Security

18.1. Security Middleware Stack

Use the following packages:

npm install helmet xss-clean express-mongo-sanitize cors

Then apply them globally in app.js or server.js:

```
const helmet = require('helmet');
const xss = require('xss-clean');
const mongoSanitize = require('express-mongo-sanitize');
const cors = require('cors');

app.use(helmet()); // Sets secure HTTP headers
app.use(xss()); // Prevent XSS attacks
app.use(mongoSanitize()); // Prevent MongoDB injection
app.use(cors({ origin: 'http://localhost:3000', credentials: true }));
```

18.2. Environment Variables & Secrets

- Create a .env file to hide secrets:

```
PORT=5000
MONGO_URI=your_mongo_uri
JWT_SECRET=your_secret_key
```

- Use dotenv package:

```
npm install dotenv
```

In server.js:

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

Keep .env in .gitignore to avoid pushing secrets to GitHub.

18.3. Use HTTP-only Cookies for JWTs

Already implemented in Unit 2, but here's a reminder why:

- **HTTP-only** cookies are **not accessible via JavaScript**, so they protect tokens from XSS.
- Use { secure: true } in production for HTTPS.

Example:

```
res.cookie('token', token, {
  httpOnly: true,
  secure: process.env.NODE_ENV === 'production',
  sameSite: 'Strict',
  maxAge: 60 * 60 * 1000 // 1 hour
});
```

18.4. Limit Request Size

Prevent large payload attacks:

```
app.use(express.json({ limit: '10kb' }));
```

18.5. Rate Limiting (Prevent Brute Force)

Install:

```
npm install express-rate-limit

Usage:

const rateLimit = require('express-rate-limit');

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // limit each IP to 100 requests
  message: 'Too many requests from this IP, please try again later.'
});

app.use(limiter);
```

18.6. Role-Based Authorization (Optional)

If you want to allow only **admins** to create/delete products:

```
// middleware/authorizeRole.js
module.exports = (role) => {
  return (req, res, next) => {
    if (req.user.role !== role) {
      return res.status(403).json({ message: 'Access Denied' });
    }
    next();
  };
};
```

Use in route:

```
router.post('/', verifyToken, authorizeRole('admin'), createProduct);
```

18.7. Centralized Error Handling

In middlewares/errorHandler.js:

```
module.exports = (err, req, res, next) => {
  console.error(err.stack);
  res.status(err.statusCode || 500).json({
    message: err.message || 'Something went wrong',
    status: 'error'
  });
};
```

Use in app.js:

```
const errorHandler = require('./middlewares/errorHandler');
app.use(errorHandler);
```

18..8. Summary: Production-Ready API Security Checklist

Feature	Done?
Helmet + CORS + XSS + Sanitize	✅
Environment variables loaded	✅
Secure JWT in HTTP-only cookie	✅
Centralized error handling	✅
Rate limiting + request size cap	✅
Optional role-based access	✅

Unit 19: Final Project – Testing & Wrap-up

19.1. Test the Full API with Postman

Create and save a **Postman collection** with the following tests:

Public Routes

- POST /api/auth/register → User registration
- POST /api/auth/login → Login, check if token is set in cookie

Protected Routes (require verifyToken)

- GET /api/user/dashboard → Should return user data only if logged in
- POST /api/products → Add new product (admin-only, if role-based)
- GET /api/products → Fetch products
- PUT /api/products/:id → Update product
- DELETE /api/products/:id → Delete product

Add **Cookie in Postman** under "Cookies" → your localhost → Add token=<jwt_token>

19.2. Test All Edge Cases

Case	Expected Outcome
Invalid JWT token	401 Unauthorized
Missing fields on register	400 Bad Request
Bad product ID	500 Internal Server Error
Product not found	404 Not Found
Unauthorized product delete	403 Forbidden (if role-based)

19.3. Manual Security Checklist

Before deployment, verify:

- CORS settings are correct (origin set properly)
- .env secrets not pushed to Git
- HTTP-only cookies used
- MongoDB URI is secure and not public
- Rate limiting is on

19.4. Optional Deployment (Cloud)

If you want to deploy:

🔧 **Backend (Node + MongoDB)**

- Use Render, Railway, or [Vercel Functions]

- Push code to GitHub
- Set environment variables (JWT_SECRET, MONGO_URI, etc.)
- Connect to **MongoDB Atlas**

🧪 **Test Deployment**

- Deploy and test each endpoint with Postman or frontend

19.5. Final Folder Structure Recap

```
project/
├── controllers/
├── models/
├── routes/
├── middleware/
├── config/
├── utils/
├── app.js
├── server.js
├── .env
```

Keep all logic modular, clean, and testable.

19.6. Final Outcome

You've now built a **secure, full-featured REST API** with:

- Auth (Register/Login)
- JWT & Cookies
- Middleware
- MongoDB CRUD
- REST design
- Validation, error handling
- Security best practices
- Scalable project structure