Node.js – HTTP Module, Views & Layouts, Middleware, Routing

1. The HTTP Module

➤ **What is the HTTP Module?**

- It is a core module in Node.js.
- Allows you to create a web server that can handle requests and responses.
- No need to install external libraries.

➤ **Features:**

- Create server applications.
- Handle different types of HTTP requests (GET, POST, etc.).
- Serve HTML, JSON, or other content.

Syntax Example:

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

const server = http.createServer((req, res) => {
   res.writeHead(200, {'Content-Type': 'text/plain'});
   res.end('Hello from Node.js server!');
});

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

✓ Output:
Server starts and when you visit http://localhost:3000, it shows "Hello from Node.js server!"

➤ **Important Methods:**

- http.createServer(callback) – Creates an HTTP server.
- req – Contains request details.
- res – Used to send the response.
- listen(port, callback) – Starts the server.

Teaser before Views and Layouts

What is Express in Node.js?

Express is a **web application framework** for **Node.js**. It provides a set of tools and features that make it easier to build web servers, APIs, and dynamic web applications without having to write repetitive or low-level code.

➤ **Key Features of Express:**

✓ **Simplified Server Creation** – Easily set up web servers and define how they respond to requests.
✓ **Routing** – Handle different URLs and HTTP methods like GET, POST, PUT, DELETE.
✓ **Middleware Support** – Add functions that process requests before sending a response (e.g., for authentication, logging).
✓ **Template Rendering** – Integrate with view engines like EJS, Pug, or Handlebars to render dynamic HTML pages.
✓ **Flexible & Minimal** – Lightweight framework that allows you to choose the components you need.

Why Use Express?

Without Express	With Express
Need to manually handle HTTP requests and responses	Provides methods like app.get() and app.post() to easily define routes
Difficult to manage middleware or request parsing	Built-in and third-party middleware make tasks easier
Server code can be cluttered	Structure your application with routes, views, and middleware

How Express Works

- You create an instance of the Express application.
- You define routes to handle incoming requests.
- You use middleware to process requests before sending a response.
- You start the server to listen on a specific port.

How Express Fits in the Node.js Ecosystem

1. **Node.js** – Provides the runtime environment and low-level modules like http.
2. **Express** – Offers higher-level abstractions and tools to build web apps faster.
3. **Middleware** – Enhances functionality, like parsing request bodies or handling authentication.
4. **Template Engines** – Render dynamic pages by combining data and templates.

Popular Use Cases of Express

✓ Building REST APIs
✓ Serving web pages with dynamic content
✓ Handling form submissions
✓ Managing sessions and authentication
✓ Connecting to databases like MongoDB or MySQL

Conclusion

Express is one of the most widely used frameworks for building web applications in Node.js. It simplifies server creation, routing, and middleware management while providing flexibility to build scalable and maintainable applications.

✅ How to Set Up Express in a Node.js Project

✅ Prerequisites

✓ Make sure **Node.js** and **npm** are installed on your system.
To check, run these commands in your terminal:

node -v

npm -v

If not installed, download from https://nodejs.org.

✅ Step 1 – Create a New Project Folder

mkdir my-express-app

cd my-express-app

✅ Step 2 – Initialize the Project

npm init -y

This creates a package.json file with default settings.

✅ Step 3 – Install Express

npm install express

This downloads and installs the Express framework into your project.

✅ Step 4 – Create the Entry Point File

Create a file named app.js or index.js.

touch app.js

✅ Step 5 – Write Your First Express Server

Open app.js and write the following code:

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

const app = express();
```

```
const port = 3000;


// Define a simple route
app.get('/', (req, res) => {
   res.send('Hello from Express!');
});
// Start the server
app.listen(port, () => {
   console.log(`Server is running at http://localhost:${port}`);
});
```

✅ **Step 6 – Run the Server**

In the terminal, run:

```
node app.js
```

✓ You should see:
Server is running at http://localhost:3000

Visit http://localhost:3000 in your browser, and you will see:
Hello from Express!

✅ **Optional – Install Nodemon for Auto Restart**

To automatically restart the server when you save changes:

1. Install nodemon globally (or locally as a dev dependency):

```
npm install -g nodemon
```

or

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

2. Run the server using nodemon:

```
nodemon app.js
```

Now, whenever you save the file, the server will restart.

✅ Project Structure Example

my-express-app/

|

├── node_modules/

├── package.json

├── package-lock.json

└── app.js

✅ Next Steps After Setup

✓ Add more routes with app.get(), app.post(), etc.
✓ Use middleware with app.use()
✓ Integrate template engines like EJS or Pug
✓ Connect to databases like MongoDB or MySQL
✓ Add error handling and environment variables

✅ Summary

1. Create a project folder.
2. Initialize npm with npm init -y.
3. Install Express using npm install express.
4. Create app.js with server code.
5. Run the server using node app.js.
6. Optionally use nodemon for auto-restarts during development.

Express is now set up and ready for you to build web apps!

Let me know if you want sample projects or further guidance on middleware, routing, or database integration.

2. Views and Layouts

What are Views and Layouts?

- Views are templates that define how data is presented.
- Layouts are templates that wrap around views to avoid repeating common structures like headers, footers, etc.
- Often used with template engines like **EJS, Pug, or Handlebars**.

Using EJS with Express (Example)

1. **Install dependencies:**

```
npm install express ejs
```

2. **Set up views and layout:**

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

app.set('view engine', 'ejs');

app.get('/', (req, res) => {
  res.render('index', { title: 'Home Page' });
});

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

3. **Folder Structure:**

```
project/
|
├── views/
|   ├── layout.ejs
|   └── index.ejs
```

```
├── app.js
└── package.json
```

4. **layout.ejs Example:**

```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
</head>
<body>
  <header>
    <h1>My Website</h1>
  </header>

  <main>
    <%- body %>
  </main>

  <footer>
    <p>Footer Content</p>
  </footer>
</body>
</html>
```

5. **index.ejs Example:**

```
<h2>Welcome to the Home Page</h2>
<p>This is a sample view rendered using EJS.</p>
```

➤ **Why Views and Layouts are useful:**

✓ Avoid code duplication
✓ Separate logic from presentation
✓ Easier maintenance and scalability

✅ 3. Middleware

➤ What is Middleware?

- Functions that execute during the lifecycle of a request.
- Can modify request and response objects.
- Used for logging, authentication, error handling, etc.

➤ Types of Middleware:

1. **Application-level middleware**
2. **Router-level middleware**
3. **Built-in middleware (like express.json())**
4. **Third-party middleware (like cors, body-parser)**

➤ Example:

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

// Custom middleware
app.use((req, res, next) => {
  console.log(`${req.method} request to ${req.url}`);
  next(); // Pass to the next middleware
});

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

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

✓ Output: Logs every request method and URL.

➤ **Middleware Functions:**

- req – Request object
- res – Response object
- next – Pass control to the next middleware

✅ **4. Routing**

➤ **What is Routing?**

- The mechanism to handle different endpoints in your application.
- Routes map URLs to specific functions.

➤ **Basic Routing Example:**

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

app.get('/', (req, res) => {
   res.send('Home Page');
});

app.get('/about', (req, res) => {
   res.send('About Page');
});

app.post('/submit', (req, res) => {
   res.send('Form submitted');
});
```

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

✓ Routes defined: /, /about, /submit.

➤ **Route Parameters Example:**

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

✓ Access by visiting /user/123, response: User ID: 123

➤ **Route Grouping using express.Router:**

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

router.get('/dashboard', (req, res) => {
   res.send('Dashboard');
});

app.use('/admin', router);
```

✓ Now /admin/dashboard will be accessible.

✅ **Summary**

Topic	**Key Takeaways**
HTTP Module	Used to create web servers with requests and responses.
Views & Layouts	Templates for rendering dynamic content with reusable structures.
Middleware	Functions that process requests/responses, useful for logging, authentication, etc.

Topic	Key Takeaways
Routing	Maps URLs to functions to handle requests and responses.

✅ **Key Terms**

✓ http.createServer() – Creates server
✓ res.render() – Renders views
✓ app.use() – Applies middleware
✓ req.params – Holds route parameters
✓ next() – Moves to the next middleware

Let me know if you want exercises, diagrams, or sample projects based on these topics.