
Chapter 4: Building a Server with Node.js and Express

Introduction

In the previous chapter, you learned how JavaScript makes web pages interactive by manipulating the content and responding to user actions directly within the browser. Now, it's time to take the next step and understand how the client and server communicate. The server plays a crucial role in serving content, processing data, and responding to requests from users.

In this chapter, you will learn how to **build a simple server using Node.js and Express.js**. We will go through everything from setting up the environment to writing code, explaining each concept in depth. You will also see how this server can handle basic requests like loading a webpage or receiving data from a user.

By the end of this chapter, you will:

1. Understand what Node.js is and how it works.
2. Learn why Express.js is a powerful tool for building servers.
3. Create a simple server that can respond to requests.
4. Understand how routes work and how the server handles different kinds of requests.
5. Practice by creating example projects with detailed explanations.

Let’s begin!

What is Node.js?

Node.js is a **JavaScript runtime environment** that allows you to run JavaScript code outside of the browser. Traditionally, JavaScript was only used in browsers to make web pages interactive. However, with Node.js, you can write server-side applications in JavaScript.

Why use Node.js?

- **Same Language Everywhere**: You can use JavaScript for both front-end and back-end development, which makes learning easier.
- **Non-blocking Architecture**: Node.js handles multiple requests at the same time without waiting for each one to finish.
- **Package Manager**: Node.js comes with `npm` (Node Package Manager), which lets you easily install and manage libraries and tools.

How does Node.js work?

Node.js uses an event-driven model where it listens for requests and handles them without blocking other operations. This is especially useful when many users access your server simultaneously.

What is Express.js?

Express.js is a web framework built on top of Node.js that makes it easier to write server applications. It provides helpful tools and methods that allow you to define routes, handle requests, and structure your server efficiently.

Why use Express.js?

- It simplifies server setup.
- It provides routing capabilities.
- It integrates easily with middleware (additional functions that process requests).
- It's widely used and well-documented.

Setting Up Your Development Environment

Before writing any code, you need to set up your environment. Follow these steps carefully.

Step 1 – Install Node.js

1. Download Node.js from the official website (https://nodejs.org – this is for installation only, no need to reference it further in your code).
2. Follow the instructions to install Node.js for your operating system.
3. Verify the installation by opening your terminal or command prompt and typing:

```
node -v
```

This should print the version of Node.js installed, like `v18.12.1`.

4. Also, check the version of `npm` by typing:

```
npm -v
```

This shows the version of Node Package Manager installed.

Step 2 – Create Your Project Folder

1. Create a folder named `myserver` or any name you like.
2. Open a terminal inside this folder.

```
mkdir myserver
cd myserver
```

3. Initialize the project by typing:

```
npm init -y
```

This creates a `package.json` file that manages your project's dependencies.

Step 3 – Install Express.js

Now that the project is initialized, install Express by running:

```
npm install express
```

This will download and install the Express library into your project.

Writing Your First Server

Now it’s time to create a file that will run your server.

Step 4 – Create `server.js`

Inside your `myserver` folder, create a file named `server.js`. This will be the main file for your server.

Here’s the complete code:

```
// Import the express library
const express = require('express');

// Create an instance of express
const app = express();

// Define the port number
const port = 3000;

// Define a route for the home page
app.get('/', (req, res) => {
    res.send('Hello! Welcome to my first server.');
});

// Start the server and listen on the defined port
app.listen(port, () => {
    console.log(`Server is running at http://localhost:${port}`);
});
```

Explanation of the Code

- `const express = require('express');`
  This imports the Express library so you can use it.

- `const app = express();`
  Here, you're creating an instance of Express. Think of it as your server object.

- `const port = 3000;`
  This defines the port where your server will listen for requests.

- `app.get('/', (req, res) => { ... });`
  This defines a **route** for the home page (`/`). When a user visits the root URL, this function will run and send the text "Hello! Welcome to my first server."

- `app.listen(port, () => { ... });`
  This starts the server and tells it to listen on the specified port. When the server starts successfully, it will print a message in the terminal.

Step 5 – Run the Server

In your terminal, type the following command:

node server.js

You should see:

Server is running at http://localhost:3000

Now open your browser and go to `http://localhost:3000`. You will see the message:

"Hello! Welcome to my first server."

Understanding Routes and Request Handling

What is a Route?

A route is an endpoint defined on the server that listens for specific requests. It usually consists of:

1. The **URL path**.
2. The **HTTP method** (GET, POST, etc.).
3. The **handler function** that processes the request and sends a response.

Adding More Routes

Let’s add more routes to see how the server can respond differently depending on the URL.

Modify your `server.js` as follows:

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

// Home route
app.get('/', (req, res) => {
    res.send('Hello! Welcome to my first server.');
});

// About route
app.get('/about', (req, res) => {
    res.send('This is the About page.');
});

// Contact route
app.get('/contact', (req, res) => {
    res.send('You can contact us at contact@example.com.');
});

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

Try accessing:

- `http://localhost:3000/` → Shows the home message.
- `http://localhost:3000/about` → Shows the about message.
- `http://localhost:3000/contact` → Shows the contact message.

Handling Request Parameters

Sometimes you want the server to respond differently depending on the data in the request. Express allows you to define **parameters** in the URL.

Example: Greeting by Name

Add this route:

```
app.get('/greet/:name', (req, res) => {
    const userName = req.params.name;
    res.send(`Hello, ${userName}! Welcome to the server.`);
});
```

Here's how it works:

- `:name` is a route parameter.
- `req.params.name` extracts the value from the URL.

If you visit `http://localhost:3000/greet/Alice`, the server will respond with:

"Hello, Alice! Welcome to the server."

Handling Query Strings

You can also pass data through query strings like this:

`http://localhost:3000/search?term=nodejs`

Example: Searching by Query Parameter

Add this route:

```
app.get('/search', (req, res) => {
    const searchTerm = req.query.term;
    res.send(`You searched for: ${searchTerm}`);
});
```

Here's what's happening:

- `req.query.term` extracts the `term` from the query string.

If you access `http://localhost:3000/search?term=nodejs`, you will see:

"You searched for: nodejs"

Sending HTML as a Response

So far, we've only sent plain text as responses. Servers can also send HTML content, which browsers render as webpages.

Example: Serving HTML

```
app.get('/html', (req, res) => {
    res.send(`
        <h1>Welcome to the HTML page!</h1>
        <p>This page is being served by Express.js.</p>
    `);
});
```

When you visit `http://localhost:3000/html`, you'll see a formatted webpage.

Using Middleware

Middleware functions are functions that run between the request and response. They can modify the request, handle errors, or perform other tasks.

Example: Simple Logging Middleware

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

Place this at the top of your file. It will log each incoming request before handling it.

Handling Static Files

You can serve static files like images, stylesheets, and JavaScript files using Express.

Example: Serving Static Files

1. Create a folder named `public` in your project directory.

2. Add a file `style.css` in `public`:

```
body {
```

```
    font-family: Arial, sans-serif;
    background-color: #f4f4f4;
}
```

3. In `server.js`, add this line before defining routes:

```
app.use(express.static('public'));
```

Now, any file in the `public` folder can be accessed directly.

Visit `http://localhost:3000/style.css` to see the CSS file.

Handling POST Requests

So far, we've only handled GET requests where the browser fetches data. POST requests are used when sending data to the server.

Example: Handling Form Data

1. Create a route to display the form:

```
app.get('/form', (req, res) => {
    res.send(`
        <form action="/submit" method="POST">
            <input type="text" name="username" placeholder="Enter your name" required>
            <button type="submit">Submit</button>
        </form>
    `);
});
```

2. Install the middleware to parse form data:

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

3. Handle the form submission:

```
app.post('/submit', (req, res) => {
    const username = req.body.username;
```

```
    res.send(`Form submitted! Hello, ${username}.`);
});
```

Now, when you visit `http://localhost:3000/form`, enter a name, and submit, the server will greet you.

Understanding the Request and Response Objects

In Express, each route handler receives two important objects:

- `req` (short for **request**): Contains all the information about the client's request, including parameters, query strings, body data, headers, etc.
- `res` (short for **response**): Contains methods to send a response back to the client.

Common `req` properties

- `req.params`: Route parameters like `/user/:id`.
- `req.query`: Query strings like `?term=value`.
- `req.body`: Data sent in the request body, often from forms.
- `req.method`: The HTTP method used (GET, POST, etc.).
- `req.url`: The full URL requested.

Common `res` methods

- `res.send()`: Sends text, JSON, or HTML.
- `res.json()`: Sends JSON formatted data.
- `res.redirect()`: Redirects the user to another page.
- `res.status()`: Sets the HTTP status code.

Organizing Code for Larger Applications

While this chapter focuses on simple examples, you should be aware that real-world servers are structured more carefully.

Best practices

1. **Separate routes** into different files.
2. **Use middleware** for common tasks like parsing requests.
3. **Use environment variables** for configuration like port numbers.
4. **Handle errors** gracefully using error-handling middleware.

These concepts will be covered in advanced chapters, but it's good to keep them in mind.

Summary

In this chapter, you learned:

- What Node.js is and how it allows JavaScript to run outside the browser.
- Why Express.js is used to build servers easily.
- How to set up your project with Node.js and Express.
- How to create a server that listens on a specific port.
- How routes work and how the server handles GET and POST requests.
- How to extract route parameters and query strings from requests.
- How to serve HTML and static files.
- How middleware functions help process requests.
- How to handle form submissions using POST requests.
- How the `req` and `res` objects work to send and receive data.