Week3 - Working with Express & Node Architecture & ES6 - Part 1

Amr A Khllaf

August 25, 2025

Contents

1 Introduction To Express.js

1.1 Why We Need To Learn Express.js

Express.js is **a Fast and Unopinionated and minimal and flexible Node.js web application framework** that provides a robust set of features for web and mobile applications. Learning Express.js is essential for several reasons:

1. **Simplifies Server-Side Development**: Express makes it easier to build web applications and APIs with Node.js
2. **Industry Standard**: It's one of the most popular frameworks in the Node.js ecosystem
3. **Middleware Support**: Offers a powerful middleware system for handling requests and responses
4. **Routing Capabilities**: Provides intuitive routing to organize your application
5. **Performance**: Designed to be fast and efficient for real-world applications
6. **Flexibility**: Can be used for various types of applications, from simple APIs to full-featured web servers

1.2 Opinionated Vs Unopinionated Frameworks

When selecting a framework for your project, understanding whether it's opinionated or unopinionated is crucial:

1.2.1 Opinionated Frameworks

These frameworks impose specific patterns, structures, and methodologies:

- **Defined Structure**: Clear guidelines for file organization and architecture
- **Conventions**: Strong conventions that limit configuration options
- **Development Speed**: Faster initial development with fewer decisions to make
- **Learning Curve**: Steeper initial learning but more consistency across projects
- Like: Ruby on Rails, Django, Angular, NestJS

1.2.2 Unopinionated Frameworks

These frameworks prioritize flexibility and developer freedom:

- **Flexible Structure**: Minimal constraints on application organization
- **Configuration**: Many configuration options and architectural choices
- **Customization**: Easily adaptable to different project requirements
- **Learning Curve**: Requires more decisions but allows incremental adoption
- Like: Express.js, Flask, React

1.2.3 Comparison

Aspect	Opinionated (Rails)	Unopinionated (Express)
Structure	Predefined	Developer-defined
Configuration	Convention over configuration	Explicit configuration
Learning Curve	Steeper initially	Gradual
Flexibility	Limited	High
Best For	Standardized applications	Custom solutions

Express.js is deliberately unopinionated, giving you the freedom to architect your application as needed without enforcing a rigid structure.

---

2 Hello World! With ExpressJS

To create a simple "Hello World" application using Express.js, follow these steps:

1. **Initialize a New Node.js Project**:

```
mkdir express-hello-world
cd express-hello-world
npm init -y
```

2. **Install Express.js**:

```
npm install express
```

3. **Create the Application**: Create a file named `app.js` and add the following code:

```
const express = require("express"); // Import the express module
// express is a Third Party Module (Node Module)
const app = express(); // Create an instance of the express
↪  application

app.get("/", (req, res) => {
  res.send("Hello World!"); // Send a response to the client instead
↪  of end() or write() in basics
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

4. **Run the Application**:

```
node app.js
or
nodemon app.js
```

5. **Access the Application**: Open your web browser and go to `http://localhost:3000`. You should see "Hello World!" displayed on the page.

This simple Express.js application sets up a server that listens on port 3000 and responds with "Hello World!" when the root URL is accessed.

2.1 Explain The Hello World Code

- `const express = require("express");`: This line imports the Express.js module, allowing us to use its features in our application.
- `const app = express();`: Here, we create an instance of the Express application, which will be used to define our routes and middleware.
- `app.get("/", (req, res) => { res.send("Hello World!"); });`: This line sets up a route for the root URL ("/"). When a GET request is made to this URL, the server responds with "Hello World!".
- `const PORT = process.env.PORT || 3000;`: This line defines the port on which the server will listen. It first checks if a PORT environment variable is set; if not, it defaults to port 3000.
- `app.listen(PORT, () => { console.log(Server is running on port ${PORT}); });`: Finally, we start the server and listen for incoming requests on the specified port. A message is logged to the console indicating that the server is running.

2.2 Response Methods in Express

2.2.1 The `send()` Method

The `send()` method is one of Express's most versatile response methods that:

- Automatically sets the appropriate Content-Type header
- Sends the response with the correct encoding

- Handles different data types intelligently

2.2.2 Key Features

```
res.send(body);
```

- **Automatic Content Detection**: Detects and formats based on the argument type
- **Multiple Data Types**: Handles strings, objects, arrays, Buffers, and more
- **Content-Type Setting**:
  - Objects/Arrays → `application/json`
  - Strings → `text/html`
  - Buffers → `application/octet-stream`

2.2.3 Examples

```
// Sending HTML
res.send("<h1>Hello World!</h1>");

// Sending JSON
res.send({ user: "John", status: "active" });

// Sending plain text
res.send("Plain text response");

// Sending a status with content
res.status(201).send("Resource created");
```

2.2.4 Compared to Node.js Core

Unlike Node's native `res.end()`, the Express `send()` method:

- Handles content-type detection automatically
- Manages HTTP headers more intelligently
- Supports chaining with other Express methods
- Simplifies sending different response formats

2.3 Send() Vs. Json() in Express.js

Both `send()` and `json()` are response methods in Express.js, but they serve slightly different purposes and have different behaviors.

2.3.1 send() Method

```
res.send(data);
```

- **Versatile**: Handles multiple data types (strings, objects, arrays, buffers)
- **Intelligent Content-Type**: Automatically sets appropriate Content-Type header
- **Flexible**: Can send HTML, plain text, JSON, or binary data
- **Content Detection**: Determines format based on the argument type
- **Chainable**: Works with other Express methods like `status()`

2.3.2 json() Method

```
res.json(data);
```

- **JSON-specific**: Explicitly converts data to JSON format
- **Forced Content-Type**: Always sets `Content-Type: application/json`
- **Type Conversion**: Will convert non-objects (like null or undefined) to valid JSON
- **Parameter Handling**: Handles JSON replacer and spacing parameters
- **Security**: Applies JSON security measures to prevent attacks

2.3.3 Comparison

Feature	send()	json()
Primary Purpose	General-purpose response	JSON-specific response
Content-Type	Varies based on data	Always application/json
Data Handling	Handles multiple formats	Forces JSON conversion
Common Use Cases	Mixed content responses	API endpoints
Object Handling	Converts objects to JSON	Same, but with more options
Non-Object Data	Sends as appropriate type	Converts to JSON representation

2.3.4 When to Use Each

- **Use send() when:**
  - You're responding with different types of content in different routes
  - You need the automatic content-type detection
  - You're sending non-JSON responses like HTML or plain text
- **Use json() when:**
  - You're building an API that always returns JSON
  - You want to ensure the response is always treated as JSON
  - You need JSON-specific parameters like replacers
  - You want to be explicit about the response format

2.3.5 Example

```
// Using send() for different response types
app.get("/html", (req, res) => {
  res.send("<h1>HTML Response</h1>"); // Content-Type: text/html
});

app.get("/data", (req, res) => {
  res.send({ name: "John", age: 30 }); // Content-Type: application/json
});

// Using json() specifically for JSON responses
app.get("/api/user", (req, res) => {
  res.json({ name: "John", age: 30 }); // Always application/json
});
```

3 Send Files With Express.js

Express provides a powerful method to serve static files to clients through the `sendFile()` method.

3.1 The sendFile() Method

```
res.sendFile(path, [options], [callback]);
```

3.1.1 Key Features

- **Streams Files**: Efficiently transfers file data to the client
- **Content-Type Detection**: Automatically sets MIME type based on file extension
- **Range Support**: Handles partial content requests for large files
- **Error Handling**: Provides callback for tracking transfer completion or errors

3.1.2 Basic Usage Example

```
app.get("/download", (req, res) => {
  res.sendFile("/path/to/file.pdf", (err) => {
    if (err) {
      console.error("Error sending file:", err);
      res.status(500).send("Error delivering file");
    }
  });
});
```

3.1.3 Options Object

```
res.sendFile("/path/to/file.pdf", {
  root: __dirname, // Root directory for relative paths
  dotfiles: "deny", // How to handle dotfiles (deny, allow, ignore)
  headers: {
    // Custom headers to include
    "x-sent": true,
  },
  maxAge: 86400000, // Cache control max-age in milliseconds (1 day)
});
```

3.1.4 Working With Absolute vs Relative Paths

For relative paths, use the `root` option:

```
// Using relative path with root option
app.get("/download", (req, res) => {
  res.sendFile("public/files/document.pdf", { root: __dirname });
});

// Using absolute path (no root option needed)
```

```
app.get("/download", (req, res) => {
  res.sendFile(__dirname + "/public/files/document.pdf");
});
```

- Explain `__dirname`: A Node.js global variable that represents the directory name of the current module. It provides an absolute path to the directory containing the currently executing file.

3.1.5 Practical Example: File Download Server

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

// Serve a downloadable file
app.get("/files/:filename", (req, res) => {
  const filename = req.params.filename;
  const filePath = path.join(__dirname, "uploads", filename);

  res.sendFile(filePath, (err) => {
    if (err) {
      res.status(404).send("File not found");
    }
  });
});

app.listen(3000, () => {
  console.log("File server running on port 3000");
});
```

`sendFile()` is preferable to manual file handling as it handles streaming, content types, and error scenarios automatically.

3.2 Path Module

The Node.js `path` module is an essential utility for working with file and directory paths in a cross-platform manner.

3.2.1 Why Use Path Module?

- **Platform Independence**: Handles path differences between Windows (backslashes) and Unix-based systems (forward slashes)
- **Path Manipulation**: Provides methods to extract, join, normalize and resolve paths
- **Path Components**: Easily access filename, extension, and directory information

3.2.2 Core Path Methods

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

// Join multiple path segments
path.join("/users", "data", "file.txt"); // '/users/data/file.txt'
```

```
// Get file extension
path.extname("document.pdf"); // '.pdf'

// Get filename (with or without extension)
path.basename("/users/docs/report.txt"); // 'report.txt'
path.basename("/users/docs/report.txt", ".txt"); // 'report'

// Get directory name
path.dirname("/users/docs/report.txt"); // '/users/docs'
```

3.3 The resolve() Method

The `path.resolve()` method is particularly powerful as it resolves a sequence of paths or path segments into an absolute path.

```
path.resolve([...paths]);
```

3.3.1 Key Features of resolve()

- **Absolute Path Creation**: Always returns an absolute path
- **Right-to-Left Processing**: Processes path segments from right to left
- **Current Directory Awareness**: Uses current working directory when needed
- **Path Segment Handling**: Handles both relative and absolute path segments

3.3.2 Examples

```
// Basic usage - resolves to absolute path from current directory
path.resolve("images", "logo.png");
// e.g., '/current/working/directory/images/logo.png'

// With absolute path segments (resets the path)
path.resolve("/tmp", "files", "../config", "settings.json");
// Result: '/tmp/config/settings.json'

// Mixing with __dirname
path.resolve(__dirname, "public", "assets");
// e.g., '/absolute/path/to/current/directory/public/assets'
```

3.3.3 When to Use resolve()

- When you need an absolute file path regardless of the current working directory
- For constructing file paths in Express applications:

```
app.get("/download", (req, res) => {
  const filePath = path.resolve(__dirname, "files", req.query.filename);
  console.log(filePath); //
  ↪  '/absolute/path/to/current/directory/files/filename'
  res.sendFile(filePath);
});
```

- When handling paths that might contain relative segments (`../` or `./`)

```
app.get("/assets", (req, res) => {
  const assetPath = path.resolve(__dirname, "public", "assets",
  ↪  req.query.file);
  console.log(assetPath); //
  ↪  '/absolute/path/to/current/directory/public/assets/file'
  res.sendFile(assetPath);
});
```

3.3.4 When to Use join()

- When you need to construct a path from multiple segments
- For creating paths that are relative to the current directory
- Example on using join()

```
app.get("/report", (req, res) => {
  const reportPath = path.join(__dirname, "files", "report.txt");
  console.log(reportPath); //
  ↪  '/absolute/path/to/current/directory/files/report.txt'
  res.sendFile(reportPath);
});
```

3.3.5 resolve() vs. join()

Feature	resolve()	join()
Purpose	Creates absolute path	Joins path segments
Behavior	Right-to-left processing	Left-to-right concatenation
Absolute segments	Resets the path	Preserves all segments
Result type	Always absolute path	Can be relative path
Common use	File access, configuration	Path construction

The `path.resolve()` method is indispensable for Express applications when you need to safely construct absolute file paths for operations like `sendFile()`.

3.4 resolve() vs. ___dirname

Understanding the difference between `path.resolve()` and `__dirname` is essential for effective path handling in Node.js applications:

3.4.1 ___dirname

- **What it is**: A global variable in Node.js that contains the absolute path to the directory of the current module (file)
- **Static value**: Does not change during execution
- **Not a function**: It's a string value, not a method
- **Example**: In a file `/home/user/app/server.js`, `__dirname` would be `/home/user/app`

3.4.2 path.resolve()

- **What it is**: A function that resolves a sequence of paths or path segments into an absolute path
- **Dynamic operation**: Processes its arguments to generate a path
- **Flexible**: Can take multiple arguments and handles relative paths
- **Example**: `path.resolve('folder', 'file.txt')` might return `/current/working/directory/fo`

3.4.3 How They Work Together

```
// Common pattern in Express applications
const absolutePath = path.resolve(__dirname, "public", "index.html");
// If __dirname is '/home/user/app', this resolves to
↪  '/home/user/app/public/index.html'

// Another common pattern
app.get("/", (req, res) => {
  res.sendFile(path.resolve(__dirname, "views", "home.html"));
});
```

3.4.4 When to Use Each

- Use **___dirname** when you need a reference to the directory containing your current file
- Use **path.resolve()** when you need to build absolute paths from relative segments
- Use **both together** when you need to reference files relative to your current script location

The combination provides reliable absolute paths regardless of the current working directory from which your Node.js application is started.

3.5 send() Vs. sendFile()

Express.js provides different methods for sending responses to clients, with `send()` and `sendFile()` being two of the most commonly used. Understanding when to use each is important for efficient web application development.

3.5.1 send() Method

```
res.send(data);
```

- **Purpose**: Sends various types of responses to the client
- **Data Types**: Handles strings, HTML, JSON objects, arrays, Buffers
- **Content Handling**:
  - Automatically sets appropriate Content-Type header
  - Converts objects to JSON
  - Sets proper encoding
- **Use Cases**: API responses, simple text/HTML content, error messages

3.5.2 sendFile() Method

```
res.sendFile(path, [options], [callback]);
```

- **Purpose**: Serves file content to the client

- **Functionality**:
  - Streams file content instead of loading it into memory
  - Sets appropriate Content-Type based on file extension
  - Handles file transfer details (range requests, streaming)
- **Use Cases**: Serving documents, images, downloadable files, HTML pages

3.5.3 Key Differences

Feature	send()	sendFile()
Content Source	In-code data	External files
Memory Usage	Holds content in memory	Streams content (better for large files)
Performance	Better for small data	Better for large files
Use Case	Dynamic content	Static assets
Path Handling	N/A	Requires file paths

3.5.4 Code Comparison

```
// Using send() for dynamic content
app.get("/api/user", (req, res) => {
  res.send({ name: "John", age: 30 });
});

// Using sendFile() for file content
app.get("/download", (req, res) => {
  res.sendFile(path.resolve(__dirname, "documents/report.pdf"));
});
```

3.5.5 When to Choose Each

- Choose **send()** when:
  - Sending data generated within your application code
  - Working with small amounts of data
  - Need to send different data types dynamically
- Choose **sendFile()** when:
  - Serving existing files from the file system
  - Dealing with large files that shouldn't be loaded into memory
  - Delivering downloadable content to users

---

4 Static Files in Express

- Express makes it easy to serve static files, such as images, CSS files, and JavaScript files. This is done using the built-in `express.static` middleware.
- This Public Folder is the one who is sent to the client or Frontend.

4.0.1 Setting Up Static File Serving

To serve static files, you need to specify a directory that contains the files you want to serve. This is typically done in your main application file (e.g., `app.js` or `server.js`).

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

const app = express();

// Serve static files from the "public" directory
app.use(
  express.static(path.join(__dirname, "public"), {
    maxAge: "1d", // Cache static assets for 1 day
  })
);
```

4.0.2 Directory Structure

Assuming you have the following directory structure:

```
/project
  /public
    /images
      logo.png
    /css
      styles.css
    /js
      scripts.js
```

With the above setup, you can access your static files using the following URLs:

- `http://localhost:3000/images/logo.png`
- `http://localhost:3000/css/styles.css`
- `http://localhost:3000/js/scripts.js`

4.0.3 Best Practices

- **Organize Static Files**: Keep your static files organized in a dedicated directory (e.g., `public`) to make it easier to manage and serve them.
- **Cache Control**: Use caching headers to improve performance for static assets. You can set cache control headers using middleware.
- **Minification**: Minify CSS and JavaScript files for production to reduce file size and improve load times.

```
app.use(
  express.static(path.join(__dirname, "public"), {
    maxAge: "1d", // Cache static assets for 1 day
  })
);
```

4.0.4 Conclusion

Serving static files in Express is straightforward with the `express.static` middleware. By following best practices for organization, caching, and minification, you can ensure efficient delivery of static assets in your web applications.

---

5 JSON With ExpressJS

Express provides powerful methods for handling JSON data in both requests and responses.

5.1 Sending JSON Responses

5.1.1 The json() Method

```
res.json(data);
```

- **Purpose**: Specifically designed for sending JSON responses
- **Content-Type**: Automatically sets `Content-Type: application/json`
- **Data Conversion**: Converts non-objects to proper JSON format
- **Security**: Implements JSON security measures to prevent injection attacks

5.1.2 Example: Basic JSON API

```
app.get("/api/user", (req, res) => {
  res.json({
    id: 1,
    name: "John Doe",
    email: "john@example.com",
    roles: ["user", "admin"],
  });
});
```

5.2 Receiving JSON Data

Express provides the `express.json()` middleware to parse incoming JSON requests:

```
// Add JSON parsing middleware
app.use(express.json()); // Solve the problem of parsing JSON request
↪  bodies which is unhandled by default

// Now req.body will contain parsed JSON data
app.post("/api/users", (req, res) => {
  const newUser = req.body; // Parsed JSON object in req.body
  console.log(newUser); // Parsed JSON object

  // Process the data...

  res.status(201).json({
    message: "User created",
    user: newUser,
```

```
  });
});
```

5.3 JSON vs Other Response Types

Feature	json()	send()	sendFile()
Primary Use	API data	Mixed content types	File delivery
Data Format	Always JSON	Varies based on input	Binary file content
Content-Type	application/json	Auto-detected	Based on file extension
Best For	RESTful APIs, data services	General-purpose responses	Documents, images, downloads

5.4 Best Practices

1. **Consistent Response Structure**:

```
// Good practice - consistent structure
app.get("/api/items", (req, res) => {
  res.json({
    success: true,
    data: items,
    count: items.length,
  });
});
```

2. **Proper Error Handling**:

```
app.get("/api/resource/:id", (req, res) => {
  try {
    const item = findItem(req.params.id);
    if (!item) {
      return res.status(404).json({
        success: false,
        error: "Resource not found",
      });
    }
    res.json({ success: true, data: item });
  } catch (err) {
    res.status(500).json({
      success: false,
      error: "Server error",
    });
  }
});
```

3. **Content Negotiation**:

```
app.get("/users/:id", (req, res) => {
  const user = getUser(req.params.id);

  // Respond based on Accept header
  res.format({
    "application/json": () => res.json(user),
    "text/html": () => res.render("user", { user }),
    default: () => res.status(406).send("Not Acceptable"),
  });
});
```

JSON handling is fundamental to modern Express applications, especially when building APIs that interact with frontend JavaScript frameworks.

---

6 Params and Query Params With ExpressJS

Express.js provides two primary ways to extract data from client requests: route parameters and query parameters. Understanding both methods is essential for building flexible and interactive web applications.

6.1 Route Parameters

Route parameters capture values specified at specific positions in the URL path.

6.1.1 Basic Syntax

```
app.get("/users/:userId", (req, res) => {
  const userId = req.params.userId;
  res.send(`Looking at user profile: ${userId}`);
});
```

6.1.2 Multiple Parameters

```
app.get("/products/:category/:productId", (req, res) => {
  const { category, productId } = req.params;
  res.json({
    category,
    productId,
    message: `Viewing product ${productId} in ${category}`,
  });
});
```

6.1.3 Parameter Constraints

You can use regular expressions to constrain parameter formats:

```
// Only match numeric IDs
app.get("/users/:userId(\\d+)", (req, res) => {
  res.send(`Looking at numeric user ID: ${req.params.userId}`);
});
```

```
// Only match specific patterns
app.get("/files/:filename.:ext(png|jpg|gif)", (req, res) => {
  const { filename, ext } = req.params;
  res.send(`Requested ${filename} with extension ${ext}`);
});
```

6.2 Query Parameters

Query parameters are key-value pairs appended to the URL after a question mark (?).

6.2.1 Basic Usage

```
// URL: /search?q=express&limit=10
app.get("/search", (req, res) => {
  const query = req.query.q; // Search query
  const limit = req.query.limit || 20; // Default value if not provided

  res.json({
    searchTerm: query,
    limit: limit,
    message: `Searching for "${query}" with limit ${limit}`,
  });
});
```

6.2.2 Multiple Query Parameters

```
// URL:
↪  /products?category=electronics&minPrice=100&maxPrice=500&sort=price
app.get("/products", (req, res) => {
  const { category, minPrice, maxPrice, sort } = req.query;

  res.json({
    filters: { category, minPrice, maxPrice },
    sortBy: sort,
    message: `Filtered ${category} products from $${minPrice} to
    ↪  $${maxPrice}`,
  });
});
```

6.3 Params vs Query Params: When to Use Each

Route Parameters	Query Parameters
Part of the URL path	Appended after `?` in URL
Required by design	Optional by nature
Best for: identifying specific resources	Best for: filtering, sorting, pagination
Example: `/users/42`	Example: `/users?role=admin`
Less flexible, part of route definition	More flexible, can be added without changing routes
Cleaner URLs for essential data	Better for optional parameters

Route Parameters	Query Parameters

6.3.1 Examples of Appropriate Use

Route Parameters:

- Resource identifiers: `/posts/123`
- Hierarchical data: `/departments/sales/employees/42`
- SEO-friendly URLs: `/articles/2023/01/express-tutorial`

Query Parameters:

- Search terms: `/search?q=express`
- Filtering: `/products?category=electronics&inStock=true`
- Pagination: `/users?page=2&limit=20`
- Sorting: `/comments?sort=newest&direction=desc`

6.4 Practical Examples

6.4.1 Combining Both Approaches

```
// URL: /users/42/posts?status=published&sort=newest
app.get("/users/:userId/posts", (req, res) => {
  const userId = req.params.userId;
  const { status, sort } = req.query;

  res.json({
    userId,
    filters: { status },
    sortBy: sort,
    message: `Fetching ${status} posts for user ${userId}, sorted by
    ↪  ${sort}`,
  });
});
```

Route parameters and query parameters provide complementary approaches to handling data in Express applications. Understanding when to use each helps create intuitive, RESTful APIs and web services.

---

7 404 Not Found With ExpressJS

Handling "Not Found" errors properly is essential for creating a professional Express application. When a user requests a route that doesn't exist, you should return a proper 404 response.

7.1 Basic 404 Handler

In Express, you can add a 404 handler by placing it after all other routes:

```
// Define all your routes first
app.get("/", (req, res) => {
  /* ... */
});
app.get("/about", (req, res) => {
  /* ... */
});

// 404 handler (must be placed after all other routes)
app.use((req, res) => {
  res.status(404).send("404 - Page Not Found");
});
```

7.2 Enhanced 404 Responses

7.2.1 JSON Response for APIs

```
app.use((req, res) => {
  res.status(404).json({
    success: false,
    error: "Resource not found",
    path: req.originalUrl,
  });
});
```

7.2.2 HTML Response with Templates

```
app.use((req, res) => {
  res.status(404).render("404", {
    title: "Page Not Found",
    path: req.originalUrl,
  });
});
```

7.2.3 Serving a Static 404 Page

```
app.use((req, res) => {
  res.status(404).sendFile(path.join(__dirname, "public", "404.html"));
});
```

7.3 Content Negotiation

You can provide different formats based on the client's Accept header:

```
app.use((req, res) => {
  res.status(404);

  res.format({
    "text/html": () => {
      res.render("404", { url: req.originalUrl });
    },
    "application/json": () => {
```

```
      res.json({ error: "Not found", path: req.originalUrl });
    },
    default: () => {
      res.send("404 - Resource not found");
    },
  });
});
```

7.4 Best Practices

1. **Set the correct status code** (404)
2. **Provide helpful information** about the error
3. **Match your response format** to your application type (API vs. website)
4. **Consider logging** 404s to identify broken links
5. **Place the 404 handler** after all other routes

Remember that Express processes middleware and routes in the order they are defined, so your 404 handler must come last to catch all unmatched routes.

---

8 Get All Users With Express.js

8.1 Creating a Users API Endpoint

To retrieve a list of all users in an Express.js application, you can implement a dedicated route handler:

```
// Define sample user data
const users = [
  { id: 1, name: "John Doe", email: "john@example.com" },
  { id: 2, name: "Jane Smith", email: "jane@example.com" },
  { id: 3, name: "Bob Johnson", email: "bob@example.com" },
];

// Create GET endpoint for all users
app.get("/api/users", (req, res) => {
  res.json({
    success: true,
    count: users.length,
    data: users,
  });
});
```

8.2 Adding Filtering Capabilities

You can enhance your endpoint with query parameter filtering:

```
app.get("/api/users", (req, res) => {
  let result = [...users];
```

```
  // Filter by name if provided in query
  if (req.query.name) {
    result = result.filter((user) =>
      user.name.toLowerCase().includes(req.query.name.toLowerCase())
    );
  }

  res.json({
    success: true,
    count: result.length,
    data: result,
  });
});
```

8.3 Error Handling

Implement proper error handling for robustness:

```
app.get("/api/users", (req, res) => {
  try {
    // User retrieval logic here

    res.json({
      success: true,
      count: users.length,
      data: users,
    });
  } catch (error) {
    res.status(500).json({
      success: false,
      error: "Failed to retrieve users",
    });
  }
});
```

Access this endpoint by sending a GET request to `/api/users`.

---

9 Add User With ExpressJS

9.1 Creating a POST Endpoint for User Creation

To add a new user in an Express.js application, you need to create a POST endpoint that accepts user data and processes it:

```
// Make sure to include the JSON parsing middleware first
app.use(express.json()); // Parse JSON request bodies

// POST endpoint for adding a user
app.post("/api/users", (req, res) => {
  const { name, email } = req.body;
```

```
  // Validate required fields
  if (!name || !email) {
    return res.status(400).json({
      success: false,
      error: "Please provide name and email",
    });
  }

  // Create new user object
  const newUser = {
    id: users.length + 1,
    name,
    email,
  };

  // Add to users array
  users.push(newUser);

  // Return success response with created user
  res.status(201).json({
    success: true,
    data: newUser,
  });
});
```

9.2 Handling Form Data

If you're accepting form submissions instead of JSON data, use the URL-encoded parser:

```
// Parse URL-encoded form data
app.use(express.urlencoded({ extended: false }));

// Handle form submissions
app.post("/users", (req, res) => {
  const { name, email } = req.body;

  // Process form data and create user
  // ...

  // Redirect after successful creation
  res.redirect("/users");
});
```

9.3 Input Validation

For robust applications, implement thorough validation:

```
app.post("/api/users", (req, res) => {
  const { name, email } = req.body;
```

```
  // Validate input
  if (!name || name.trim().length < 2) {
    return res.status(400).json({
      success: false,
      error: "Name must be at least 2 characters",
    });
  }

  if (!email || !email.includes("@")) {
    return res.status(400).json({
      success: false,
      error: "Please provide a valid email address",
    });
  }

  // Check for duplicate email
  if (users.find((user) => user.email === email)) {
    return res.status(409).json({
      success: false,
      error: "Email already in use",
    });
  }

  // Create and save user if validation passes
  const newUser = { id: Date.now(), name, email };
  users.push(newUser);

  res.status(201).json({
    success: true,
    data: newUser,
  });
});
```

9.4 Error Handling

Implement try-catch blocks for robust error handling:

```
app.post("/api/users", (req, res) => {
  try {
    // User creation logic here
    // ...

    res.status(201).json({
      success: true,
      data: newUser,
    });
  } catch (error) {
    console.error("Error creating user:", error);
    res.status(500).json({
      success: false,
      error: "Server error while creating user",
```

```
    });
  }
});
```

Access this endpoint by sending a POST request to `/api/users` with a JSON body containing the user data.

---

10 Update User With ExpressJS

10.1 Creating PUT and PATCH Endpoints for User Updates

Express.js provides simple ways to implement user update functionality through PUT and PATCH HTTP methods.

10.1.1 PUT vs PATCH Methods

Method	Purpose	Best For
PUT	Replace entire resource	Complete resource updates
PATCH	Apply partial modifications	Partial updates to specific fields

10.2 PUT Implementation (Full Update)

```
app.put("/api/users/:id", (req, res) => {
  const userId = parseInt(req.params.id); // Convert ID to integer
  const { name, email } = req.body; // Destructure request body

  // Validate required fields
  if (!name || !email) {
    return res.status(400).json({
      success: false,
      error: "Please provide name and email",
    });
  }

  // Find user index by ID
  const userIndex = users.findIndex((user) => user.id === userId);

  // Check if user exists
  if (userIndex === -1) {
    return res.status(404).json({
      success: false,
      error: "User not found",
    });
  }

  // Replace entire user object
  users[userIndex] = {
```

```
    id: userId,
    name,
    email,
  };

  res.json({
    success: true,
    data: users[userIndex],
  });
});
```

10.3 PATCH Implementation (Partial Update)

```
app.patch("/api/users/:id", (req, res) => {
  const userId = parseInt(req.params.id);
  const updates = req.body;

  // Find user by ID
  const userIndex = users.findIndex((user) => user.id === userId);

  // Check if user exists
  if (userIndex === -1) {
    return res.status(404).json({
      success: false,
      error: "User not found",
    });
  }

  // Apply partial updates
  users[userIndex] = {
    ...users[userIndex],
    ...updates,
  };

  res.json({
    success: true,
    data: users[userIndex],
  });
});
```

10.4 Validation and Error Handling

For robust user updates, implement thorough validation:

```
app.patch("/api/users/:id", (req, res) => {
  try {
    const userId = parseInt(req.params.id);
    const updates = req.body;

    // Find user
    const userIndex = users.findIndex((user) => user.id === userId);
```

```
    if (userIndex === -1) {
      return res.status(404).json({
        success: false,
        error: "User not found",
      });
    }

    // Validate email if it's being updated
    if (updates.email && !updates.email.includes("@")) {
      return res.status(400).json({
        success: false,
        error: "Please provide a valid email address",
      });
    }

    // Check email uniqueness if changing email
    if (
      updates.email &&
      users.some((u) => u.email === updates.email && u.id !== userId)
    ) {
      return res.status(409).json({
        success: false,
        error: "Email already in use",
      });
    }

    // Apply updates
    users[userIndex] = {
      ...users[userIndex],
      ...updates,
    };

    res.json({
      success: true,
      data: users[userIndex],
    });
  } catch (error) {
    console.error("Error updating user:", error);
    res.status(500).json({
      success: false,
      error: "Server error while updating user",
    });
  }
});
```

Access these endpoints by sending PUT or PATCH requests to `/api/users/:id` with a JSON body containing the update data.

---

11 Delete User With ExpressJS

11.1 Creating a DELETE Endpoint for User Removal

Implementing user deletion functionality in Express.js is straightforward using the DELETE HTTP method:

```
app.delete("/api/users/:id", (req, res) => {
  const userId = parseInt(req.params.id);

  // Find user index by ID
  const userIndex = users.findIndex((user) => user.id === userId);

  // Check if user exists
  if (userIndex === -1) {
    return res.status(404).json({
      success: false,
      error: "User not found",
    });
  }

  // Remove user from array
  const deletedUser = users.splice(userIndex, 1)[0];

  // Return success response
  res.json({
    success: true,
    message: "User deleted successfully",
    data: deletedUser,
  });
});
```

11.2 Error Handling

Add robust error handling to ensure reliable operation:

```
app.delete("/api/users/:id", (req, res) => {
  try {
    const userId = parseInt(req.params.id);

    // Find user index
    const userIndex = users.findIndex((user) => user.id === userId);

    if (userIndex === -1) {
      return res.status(404).json({
        success: false,
        error: "User not found",
      });
    }

    // Delete the user
    const deletedUser = users.splice(userIndex, 1)[0];
```

```
    res.json({
      success: true,
      message: "User deleted successfully",
      data: deletedUser,
    });
  } catch (error) {
    console.error("Error deleting user:", error);
    res.status(500).json({
      success: false,
      error: "Server error while deleting user",
    });
  }
});
```

11.3 splice() Return Value: Understanding the [0] Notation

When deleting a user with `splice()`, you'll notice two common patterns:

```
// Version 1: With [0]
const deletedUser = users.splice(userIndex, 1)[0];

// Version 2: Without [0]
users.splice(userIndex, 1);
```

11.3.1 Key Differences

- **Return Value**: `splice()` always returns an array containing the deleted elements
- **With [0]**: Extracts the deleted user object from the returned array
- **Without [0]**: Ignores the returned array entirely

11.3.2 Why Use [0]?

```
// The splice method returns an array of removed elements
const deletedUserArray = users.splice(userIndex, 1); // Returns: [{ id: 2,
↪  name: "Jane", ... }]

// Using [0] accesses the user object directly
const deletedUser = users.splice(userIndex, 1)[0]; // Returns: { id: 2,
↪  name: "Jane", ... }
```

11.3.3 Best Practice

- Use `[0]` when you need to access or return the deleted item
- Skip it when you only care about removing the item

```
// When returning the deleted user in the response:
const deletedUser = users.splice(userIndex, 1)[0];
res.json({ success: true, data: deletedUser });

// When simply confirming deletion without needing the details:
```

```
users.splice(userIndex, 1);
res.json({ success: true, message: "User deleted" });
```

11.4 Soft Delete Alternative

For applications that need to preserve data, implement a soft delete approach:

```
app.delete("/api/users/:id", (req, res) => {
  const userId = parseInt(req.params.id);

  // Find user
  const userIndex = users.findIndex((user) => user.id === userId);

  if (userIndex === -1) {
    return res.status(404).json({
      success: false,
      error: "User not found",
    });
  }

  // Mark as deleted instead of removing
  users[userIndex] = {
    ...users[userIndex],
    isActive: false,
    deletedAt: new Date().toISOString(),
  };

  res.json({
    success: true,
    message: "User deactivated successfully",
    data: users[userIndex],
  });
});
```

Access this endpoint by sending a DELETE request to `/api/users/:id` where `:id` is the ID of the user to delete.