Building RESTful CRUD APIs with Express.js

Course: MERN Backend Development

Instructor: Joel Odufu Ekowoicho
Topic: Creating CRUD APIs with Express.js
Date: April 2025
Duration: 2 Hours

Learning Objectives

By the end of this lesson, students will be able to:

1. Understand RESTful APIs and the CRUD operations (Create, Read, Update, Delete).
2. Set up an Express.js server and handle API requests.
3. Use different input methods: request body, URL parameters, query parameters, and headers.
4. Implement best practices for API responses, including HTTP status codes and consistent JSON structure.
5. Build and test a complete CRUD API using in-memory data.

1. Introduction to RESTful APIs

What is a RESTful API?

- **REST** (Representational State Transfer) is an architectural style for designing APIs.
- APIs enable communication between clients (e.g., web browsers) and servers.
- RESTful APIs use **HTTP methods** to perform **CRUD** operations:
  - **C**reate: Add new data (POST).
  - **R**ead: Retrieve data (GET).
  - **U**pdate: Modify data (PUT).
  - **D**elete: Remove data (DELETE).

Key Concepts

- **Resources**: Data objects (e.g., users) accessed via URLs (e.g., `/users`).
- **HTTP Methods**: Define the action (e.g., GET to read).
- **Status Codes**:
  - `200 OK`: Success for GET, PUT, DELETE.
  - `201 Created`: Success for POST.
  - `400 Bad Request`: Invalid input.
  - `404 Not Found`: Resource not found.
  - `500 Internal Server Error`: Server issue.
- **JSON**: Standard format for data exchange.

Example

- Request: `GET /users/1`
- Response: `{ "id": 1, "name": "Alice" }`
- Status: 200 OK

2. Setting Up Express.js

Express.js is a Node.js framework for building APIs and web applications.

Step 1: Initialize a Project

1. Create a directory and initialize a Node.js project:

```
mkdir express-crud-api
cd express-crud-api
npm init -y
```

2. Install Express:

```
npm install express
```

3. Create a file named `index.js`.

Step 2: Basic Express Server with Dummy Data

Below is a fragmented code example to introduce Express, including middleware and dummy data.

```
// Import Express
const express = require("express");
const app = express();
const port = 3000;

// Middleware to parse JSON request bodies
app.use(express.json());

// Dummy data: Array of users
let users = [
  { id: 1, name: "Alice", email: "alice@example.com", role: "user" },
  { id: 2, name: "Bob", email: "bob@example.com", role: "admin" },
  { id: 3, name: "Charlie", email: "charlie@example.com", role: "user" },
];

// Helper function to generate unique IDs
const generateId = () => {
  return users.length > 0 ? Math.max(...users.map((u) => u.id)) + 1 : 1;
};

// Test route
app.get("/", (req, res) => {
  res.json({ status: "success", message: "Welcome to the API!" });
});

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

Explanation

- `express.json()` : Parses incoming JSON requests (e.g., `req.body`).
- `users` : In-memory array simulating a database.
- `generateId` : Creates unique IDs for new users.
- Run with `node index.js` and test at `http://localhost:3000` .

3. Handling Inputs in Express

Express supports multiple ways to receive data from clients:

- **Request Body (`req.body`)**: Used in POST/PUT for JSON data (e.g., `{ "name": "Dave" }`).
- **URL Parameters (`req.params`)**: Used in URLs to identify resources (e.g., `/users/:id`).
- **Query Parameters (`req.query`)**: Used for filtering/sorting (e.g., `/users?role=admin`).
- **Headers (`req.headers`)**: Used for metadata or authentication (e.g., `Authorization: Bearer token`).

Best Practices for Responses

- **Consistent JSON Structure**: Use `{ status, message, data }` for all responses.
- **Validate Inputs**: Check data to prevent errors or security issues.
- **Error Handling**: Return clear error messages with appropriate status codes.

4. CRUD Operations

We'll build a CRUD API for the `users` resource, with each operation covered in its own subtopic. Each includes validation, input handling, and response formatting.

4.1. Create (POST /users)

Purpose

Add a new user to the `users` array using data from the request body.

Input

- **req.body**: JSON payload with `name` , `email` , and optional `role` .

Code Example

```
// Input validation middleware
const validateUserInput = (req, res, next) => {
  const { name, email, role } = req.body;
  if (!name || typeof name !== "string") {
    return res.status(400).json({
      status: "error",
      message: "Name is required and must be a string",
    });
  }
  if (!email || !/^\S+@\S+\.\S+$/.test(email)) {
    return res.status(400).json({
      status: "error",
      message: "Valid email is required",
    });
  }
  if (role && !["user", "admin"].includes(role)) {
    return res.status(400).json({
      status: "error",
      message: "Role must be either 'user' or 'admin'",
    });
  }
  next();
};

// CREATE: Add a new user
app.post("/users", validateUserInput, (req, res) => {
  // Extract data from request body
  const { name, email, role = "user" } = req.body;

  // Create new user
  const newUser = {
    id: generateId(),
    name,
    email,
    role,
  };

  // Add to dummy data
  users.push(newUser);

  // Respond with 201 Created
```

```
    res.status(201).json({
      status: "success",
      message: "User created successfully",
      data: newUser,
    });
  });
```

Explanation

- **Validation**: Middleware ensures `name` is a string, `email` is valid, and `role` (if provided) is "user" or "admin".
- **Action**: Creates a user with a unique ID and adds it to `users`.
- **Response**: Returns 201 with the new user.
- **Error Handling**: Returns 400 for invalid inputs.

Testing

```
curl -X POST http://localhost:3000/users \
-H "Content-Type: application/json" \
-d '{"name":"Dave","email":"dave@example.com","role":"user"}'
```

Response (201):

```
{
  "status": "success",
  "message": "User created successfully",
  "data": { "id": 4, "name": "Dave", "email": "dave@example.com", "role": "user" }
}
```

4.2. Read (GET /users and GET /users/:id)

Purpose

Retrieve all users (with optional filtering) or a single user by ID.

Input

- **req.query**: Filter users (e.g., `?role=admin`).
- **req.params**: Identify a user (e.g., `/users/1`).

Code Example

```
// READ: Get all users with optional filtering
app.get("/users", (req, res) => {
  // Extract query parameter for filtering
  const { role } = req.query;

  // Filter users if query param exists
  let filteredUsers = users;
  if (role) {
    filteredUsers = users.filter((user) =>
      user.role.toLowerCase() === role.toLowerCase()
    );
  }

  // Respond with 200 OK
  res.status(200).json({
    status: "success",
    message: filteredUsers.length > 0 ? "Users retrieved successfully" : "No users found",
    data: filteredUsers,
    total: filteredUsers.length,
  });
});

// READ: Get a single user by ID
app.get("/users/:id", (req, res) => {
  // Extract ID from URL params
  const id = parseInt(req.params.id);

  // Find user
  const user = users.find((u) => u.id === id);

  // Check if user exists
  if (!user) {
    return res.status(404).json({
      status: "error",
      message: `User with ID ${id} not found`,
    });
  }

  // Respond with 200 OK
  res.status(200).json({
    status: "success",
```

```
    message: "User retrieved successfully",
    data: user,
  });
});
```

Explanation

- **GET /users**:
  - Uses `req.query` to filter by `role` (case-insensitive).
  - Returns all users if no filter is applied.
  - Includes `total` in the response for clarity.
- **GET /users/:id**:
  - Uses `req.params.id` to find a user.
  - Returns 404 if the user doesn't exist.
- **Response**: Both return 200 with consistent JSON structure.

Testing

1. **Get All Users**:

   ```
   curl http://localhost:3000/users
   ```

   Response (200):

   ```
   {
     "status": "success",
     "message": "Users retrieved successfully",
     "data": [
       { "id": 1, "name": "Alice", "email": "alice@example.com", "role": "user" },
       { "id": 2, "name": "Bob", "email": "bob@example.com", "role": "admin" },
       { "id": 3, "name": "Charlie", "email": "charlie@example.com", "role": "user" },
       { "id": 4, "name": "Dave", "email": "dave@example.com", "role": "user" }
     ],
     "total": 4
   }
   ```

2. **Filter by Role**:

   ```
   curl http://localhost:3000/users?role=admin
   ```

   Response (200):

```
{
  "status": "success",
  "message": "Users retrieved successfully",
  "data": [{ "id": 2, "name": "Bob", "email": "bob@example.com", "role": "admin" }],
  "total": 1
}
```

3. **Get User by ID**:

```
curl http://localhost:3000/users/1
```

Response (200):

```
{
  "status": "success",
  "message": "User retrieved successfully",
  "data": { "id": 1, "name": "Alice", "email": "alice@example.com", "role": "user" }
}
```

4.3. Update (PUT /users/:id)

Purpose

Modify an existing user's details using data from the request body.

Input

- **req.params**: Identify the user (e.g., `/users/1`).
- **req.body**: Updated fields (e.g., `{ "name": "Alicia" }`).

Code Example

```
// UPDATE: Update a user by ID
app.put("/users/:id", validateUserInput, (req, res) => {
  // Extract ID from URL params
  const id = parseInt(req.params.id);

  // Extract data from request body
  const { name, email, role } = req.body;

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

  // Check if user exists
  if (userIndex === -1) {
    return res.status(404).json({
      status: "error",
      message: `User with ID ${id} not found`,
    });
  }

  // Update user
  users[userIndex] = {
    ...users[userIndex],
    name,
    email,
    role: role || users[userIndex].role,
  };

  // Respond with 200 OK
  res.status(200).json({
    status: "success",
    message: "User updated successfully",
    data: users[userIndex],
  });
});
```

Explanation

- **Validation**: Reuses `validateUserInput` middleware from Create.
- **Action**: Updates the user's `name`, `email`, and `role` (if provided).
- **Error Handling**: Returns 404 if the user doesn't exist.

- **Response**: Returns 200 with the updated user.

Testing

```
curl -X PUT http://localhost:3000/users/1 \
-H "Content-Type: application/json" \
-d '{"name":"Alicia","email":"alicia@example.com","role":"admin"}'
```

Response (200):

```
{
  "status": "success",
  "message": "User updated successfully",
  "data": { "id": 1, "name": "Alicia", "email": "alicia@example.com", "role": "admin" }
}
```

4.4. Delete (DELETE /users/:id)

Purpose

Remove a user from the `users` array by ID.

Input

- **req.params**: Identify the user (e.g., `/users/1`).

Code Example

```
// DELETE: Delete a user by ID
app.delete("/users/:id", (req, res) => {
  // Extract ID from URL params
  const id = parseInt(req.params.id);

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

  // Check if user exists
  if (userIndex === -1) {
    return res.status(404).json({
      status: "error",
      message: `User with ID ${id} not found`,
    });
  }

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

  // Respond with 200 OK
  res.status(200).json({
    status: "success",
    message: "User deleted successfully",
    data: deletedUser,
  });
});
```

Explanation

- **Action**: Removes the user by ID using `splice`.
- **Error Handling**: Returns 404 if the user doesn't exist.
- **Response**: Returns 200 with the deleted user for confirmation.

Testing

```
curl -X DELETE http://localhost:3000/users/1
```

Response (200):

```
{
  "status": "success",
  "message": "User deleted successfully",
  "data": { "id": 1, "name": "Alicia", "email": "alicia@example.com", "role": "admin" }
}
```

5. Bonus: Protected Route with Headers

Purpose

Demonstrate using `req.headers` for authentication (e.g., protecting a route).

Code Example

```
// PROTECTED: Access users with authentication
app.get("/users/protected", (req, res) => {
  // Extract token from headers
  const token = req.headers["authorization"];

  // Simulate token validation
  if (!token || token !== "Bearer my-secret-token") {
    return res.status(401).json({
      status: "error",
      message: "Unauthorized: Invalid or missing token",
    });
  }

  // Respond with protected data
  res.status(200).json({
    status: "success",
    message: "Accessed protected user data",
    data: users,
  });
});
```

Explanation

- **Input**: `req.headers["authorization"]` checks for a token.
- **Action**: Returns all users if the token is valid.

- **Response**: Returns 200 or 401 Unauthorized.

Testing

```
curl http://localhost:3000/users/protected \
-H "Authorization: Bearer my-secret-token"
```

Response (200):

```
{
  "status": "success",
  "message": "Accessed protected user data",
  "data": [{ "id": 2, ... }, ...]
}
```

6. Error Handling

Add a global error handler to catch unexpected issues.

```
// Error handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({
    status: "error",
    message: "Something went wrong on the server",
  });
});
```

- **Why?** Ensures the API doesn't crash on errors and returns a user-friendly response.

7. Testing the Full API

1. Save all code in `index.js`.
2. Run the server:

```
node index.js
```

3. Use **Postman** or **curl** to test each endpoint.
4. Try invalid inputs (e.g., missing `name` in POST) to see error responses.

8. Best Practices Recap

- **Validation**: Validate all inputs to ensure data integrity.
- **Responses**: Use `{ status, message, data }` for consistency.
- **Status Codes**: Match the operation (e.g., 201 for Create, 200 for Read).
- **Error Handling**: Handle 400, 404, and 500 errors gracefully.
- **REST Conventions**: Use proper HTTP methods and URL structures.

9. Practical Exercise

Task: Build a CRUD API for a `products` resource (e.g., items in a store).

Requirements

1. Create a `products` array with fields: `id`, `name`, `price`, `category` (e.g., "electronics").
2. Implement CRUD routes:
   - **POST /products**: Create a product (`req.body`).
   - **GET /products**: List products, filter by category (`req.query`).
   - **GET /products/:id**: Get a product (`req.params`).
   - **PUT /products/:id**: Update a product (`req.body`, `req.params`).
   - **DELETE /products/:id**: Delete a product (`req.params`).
3. Add validation middleware to ensure:
   - `name` is a non-empty string.
   - `price` is a positive number.
   - `category` is "electronics", "clothing", or "books".
4. Use `{ status, message, data }` response format.
5. Test all routes.

Starter Code

```
let products = [
  { id: 1, name: "Laptop", price: 999.99, category: "electronics" },
];

// Validation middleware
const validateProductInput = (req, res, next) => {
  // Add validation logic
};

// CRUD routes
app.post("/products", validateProductInput, (req, res) => {
  // Add logic
});
```

Submission

- Submit `index.js` with the implemented routes.
- Include screenshots of three successful API calls (e.g., POST, GET, DELETE).

10. Additional Tips for Production

- **Database**: Use MongoDB or PostgreSQL instead of in-memory data.
- **Validation**: Use `express-validator` or `joi` for robust checks.
- **Authentication**: Implement JWT for secure routes.
- **Logging**: Add `morgan` for request logging.
- **Rate Limiting**: Use `express-rate-limit` to prevent abuse.

11. Q&A and Discussion

- What's the difference between `req.body` and `req.query`?
- Why use middleware for validation?
- How would you add pagination to GET `/users`?
- What status code should you use for a failed authentication?

12. References

- Express.js Documentation: expressjs.com
- REST API Best Practices: restfulapi.net
- HTTP Status Codes: developer.mozilla.org/en-US/docs/Web/HTTP/Status

Connection to Previous Lessons

This lesson builds on your JavaScript knowledge from our MERN Backend course, where we covered variables, functions, and async operations. The CRUD API prepares you for integrating MongoDB in the next session, as outlined in our course schedule [Memory: April 1, 2025]. The exercise aligns with your teaching approach at Aptech, emphasizing hands-on practice.