Building RESTful APIs

Building RESTful APIs with Express.js, a popular Node.js framework, is a common approach for creating backend services that interact with databases, handle user authentication, and serve data to front-end applications. In this guide, we'll cover the basics of setting up an Express.js project, handling routing, and managing middleware for creating a RESTful API.

1. Setting Up an Express.js Project

Before you start, ensure you have Node.js installed. Then, you can set up a new Express.js project.

a. Initialize the Project

Create a new directory for your project:
bash

```
mkdir my-express-api
cd my-express-api
```

1.

Initialize a new Node.js project:
bash

```
npm init -y
```

2. This command creates a `package.json` file with default settings.

Install Express.js:
bash

```
npm install express
```

3.

b. Basic Express Server Setup

Create an `index.js` file in your project directory:

javascript

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

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

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

- **`express()`**: Initializes an Express application.
- **`app.get('/', ...)`**: Sets up a route to handle GET requests to the root URL (`/`).
- **`app.listen(PORT, ...)`**: Starts the server and listens on the specified port.

To run the server, execute:

bash

```
node index.js
```

Visit `http://localhost:3000` in your browser, and you should see "Hello, World!".

2. Handling Routing in Express.js

Routing refers to how an application responds to client requests to particular endpoints (URIs).

a. Creating Basic Routes

Express allows you to define routes for different HTTP methods (GET, POST, PUT, DELETE, etc.).

Example: Basic CRUD Routes

javascript

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

// Sample data
let items = [
  { id: 1, name: 'Item 1' },
  { id: 2, name: 'Item 2' }
];

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

// GET: Retrieve all items
app.get('/items', (req, res) => {
  res.json(items);
});

// GET: Retrieve a single item by ID
app.get('/items/:id', (req, res) => {
  const item = items.find(i => i.id ===
parseInt(req.params.id));
  if (!item) return res.status(404).send('Item not found');
  res.json(item);
});

// POST: Create a new item
app.post('/items', (req, res) => {
  const newItem = {
    id: items.length + 1,
    name: req.body.name
  };
  items.push(newItem);
```

```
  res.status(201).json(newItem);
});

// PUT: Update an item by ID
app.put('/items/:id', (req, res) => {
  const item = items.find(i => i.id ===
parseInt(req.params.id));
  if (!item) return res.status(404).send('Item not found');

  item.name = req.body.name;
  res.json(item);
});

// DELETE: Remove an item by ID
app.delete('/items/:id', (req, res) => {
  const itemIndex = items.findIndex(i => i.id ===
parseInt(req.params.id));
  if (itemIndex === -1) return res.status(404).send('Item not
found');

  items.splice(itemIndex, 1);
  res.status(204).send();
});

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

- **`app.get('/items', ...)`**: Handles GET requests to retrieve all items.
- **`app.get('/items/:id', ...)`**: Handles GET requests to retrieve a specific item by ID.
- **`app.post('/items', ...)`**: Handles POST requests to create a new item.
- **`app.put('/items/:id', ...)`**: Handles PUT requests to update an existing item by ID.

- **`app.delete('/items/:id', ...)`**: Handles DELETE requests to remove an item by ID.

b. Route Parameters

Route parameters are dynamic segments of the URL that act as placeholders.

javascript

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

In this example, `:userId` is a route parameter that will be captured and made available in `req.params`.

3. Managing Middleware

Middleware functions are functions that have access to the request object (`req`), the response object (`res`), and the next middleware function in the application's request-response cycle.

a. Using Built-In Middleware

Express provides built-in middleware functions such as `express.json()` and `express.urlencoded()`.

javascript

```
app.use(express.json()); // Parses incoming JSON requests
app.use(express.urlencoded({ extended: true })); // Parses
URL-encoded data
```

b. Creating Custom Middleware

You can create your own middleware functions to perform tasks like logging, authentication, or modifying requests and responses.

Example: Custom Logging Middleware

javascript

```
const logger = (req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next(); // Passes control to the next middleware function
};

app.use(logger); // Use the custom logger middleware

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

- **`logger`**: A custom middleware function that logs the HTTP method and URL of incoming requests.
- **`next()`**: A function that passes control to the next middleware in the stack.

c. Error-Handling Middleware

Error-handling middleware is used to catch and handle errors in your application.

Example: Error-Handling Middleware

javascript

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something went wrong!');
});
```

- **Error-handling middleware**: Defined with four parameters (`err`, `req`, `res`, `next`). This middleware catches any errors that occur during the request-response cycle.

4. Organizing Routes with Express Router

As your application grows, you can use `express.Router` to modularize and organize your routes.

Example: Organizing Routes with Express Router

Create a separate file for routes (e.g., `routes/items.js`):
javascript

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

let items = [
  { id: 1, name: 'Item 1' },
  { id: 2, name: 'Item 2' }
];

router.get('/', (req, res) => {
  res.json(items);
});

router.get('/:id', (req, res) => {
  const item = items.find(i => i.id ===
parseInt(req.params.id));
  if (!item) return res.status(404).send('Item not found');
  res.json(item);
});

router.post('/', (req, res) => {
  const newItem = {
    id: items.length + 1,
    name: req.body.name
  };
  items.push(newItem);
  res.status(201).json(newItem);
});
```

```
router.put('/:id', (req, res) => {
  const item = items.find(i => i.id ===
parseInt(req.params.id));
  if (!item) return res.status(404).send('Item not found');

  item.name = req.body.name;
  res.json(item);
});

router.delete('/:id', (req, res) => {
  const itemIndex = items.findIndex(i => i.id ===
parseInt(req.params.id));
  if (itemIndex === -1) return res.status(404).send('Item not
found');

  items.splice(itemIndex, 1);
  res.status(204).send();
});

module.exports = router;
```

1.

Import and use the router in `index.js`:
javascript

```
const express = require('express');
const app = express();
const itemsRouter = require('./routes/items');
const PORT = 3000;

app.use(express.json());
app.use('/items', itemsRouter);

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

```
});
```

2.

- **`itemsRouter`**: The router defined in the `routes/items.js` file, which is then used in `index.js` with the `/items` path prefix.

5. Testing and Running the API

a. Using Postman or CURL

To test your API, you can use tools like Postman or CURL.

Example CURL Commands:

bash

```
# Get all items
curl -X GET http://localhost:3000/items

# Get a single item by ID
curl -X GET http://localhost:3000/items/1

# Create a new item
curl -X POST -H "Content-Type: application/json" -d
'{"name":"Item 3"}' http://localhost:3000/items

# Update an item
curl -X PUT -H "Content-Type: application/json" -d
'{"name":"Updated Item"}' http://localhost:3000/items/1

# Delete an item
curl -X DELETE http://localhost:3000/items/1
```

Summary

- **Setting Up Express.js**: Initialize your Node.js project, install Express, and create a basic server.
- **Handling Routing**: Define RESTful routes for handling CRUD operations using HTTP methods.
- **Managing Middleware**: Use built-in middleware for tasks like parsing JSON and create custom middleware for logging, authentication, etc.
- **Organizing Routes**: Use `express.Router` to modularize and organize your application's routes.
- **Testing API**: Use tools like Postman or CURL to test your RESTful API.