



Rapport de Tp Nodejs(expressjs):

Made By : Abdelmounaim Hmamed

# Express.js CRUD Application with Express Router

## 1. Introduction

This document provides a detailed explanation of the CRUD (Create, Read, Update, Delete) application developed using Express.js, an efficient Node.js web framework. The application demonstrates how to implement CRUD operations using the Express Router and a file-based data storage system. The structure, code implementation, and overall architecture will be covered in this report.

## 2. Objective

The main objective of this project is to create a simple CRUD application using Express.js. This involves building endpoints for managing items, using Express Router for modular routing, and storing data in a JSON file. The endpoints allow the user to create, read, update, and delete items.

## 3. Architecture Overview

The application follows a modular architecture where the logic is divided into separate components. The project consists of three primary components:

1. **Main Application File (`server.js`)\*\*: Initializes the Express server and uses the Express Router.**
2. **Router File (`routes/route.js`)\*\*: Handles all CRUD operations through defined endpoints.**
3. **Data Storage (`data/data.json`)\*\*: Stores data in a JSON file to keep the application simple.**

## 4. Directory Structure

The project directory structure is designed to maintain code modularity and separation of concerns:

```
...
express-crud-app/
|
├── app.js
├── routes/
│   └── route.js
├── data/
│   └── data.json
...
```

- **server.js**: Main entry point of the application, responsible for setting up the server and routes.
- **routes/route.js**: Contains routing logic for CRUD operations.
- **data/data.json**: JSON file for persisting item data.


## 5. Implementation Details

### 5.1 server.js

The `app.js` file is the main entry point of the application. It sets up the Express server and uses the Express Router to route the requests to the appropriate handler. It also uses the `body-parser` middleware to parse incoming JSON data.

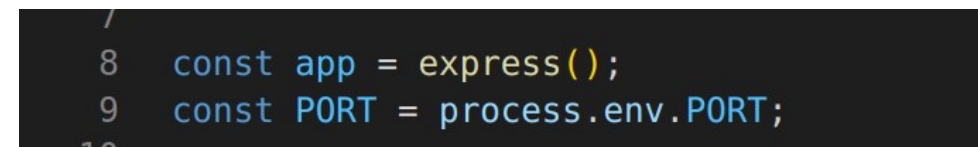
Code Breakdown:

Import necessary modules.

A screenshot of a code editor with a dark theme. The top bar shows several open files: package.json, server.js (active), route.js, request.http, .env, and data.json. The main editor area shows the following code in server.js:

```
1 const express = require('express');
2 const bodyParser = require('body-parser');
3 const itemRoutes = require('./routes/route');
4 const dotenv = require("dotenv");
5 dotenv.config();
6
7
```

Initialize the Express application and define a port.

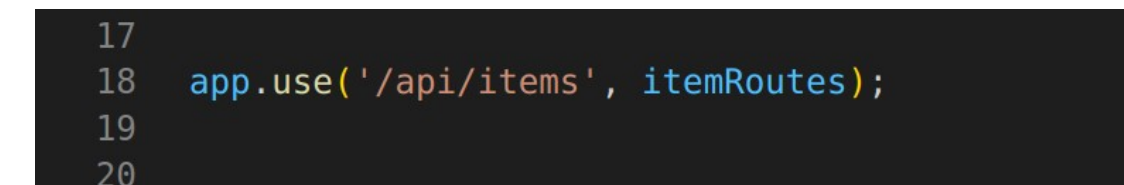
A screenshot of a code editor showing the following code:

```
8 const app = express();
9 const PORT = process.env.PORT;
```

Use `body-parser` middleware and mount the router from `route.js`.

A screenshot of a code editor showing the following code:

```
app.use(bodyParser.json());
app.use(express.json())
```

A screenshot of a code editor showing the following code:

```
17
18 app.use('/api/items', itemRoutes);
19
20
```

Start the server using `app.listen` and listen on the defined port.

```
21 app.listen(PORT, () => {  
22   console.log(`Server is running on port : ${PORT}`);  
23 });  
24
```

## 5.2 routes/route.js

The `routes/items.js` file handles the routing logic for all CRUD operations. It uses Express Router to define endpoints for creating, reading, updating, and deleting items stored in the JSON file.

Code Breakdown:

Import required modules, including Express, `fs`, and `path`.

Initialize the Express Router.



```
routes > JS route.js > router.post('/') callback  
1 const express = require('express');  
2 const fs = require('fs');  
3 const path = require('path');  
4  
5 const router = express.Router();  
6 const dataFile = path.join(__dirname, '../data/data.json');  
7
```

Define helper functions to read and write data from/to the JSON file.

```
7  
8 const readData = () => {  
9   const data = fs.readFileSync(dataFile);  
10  return JSON.parse(data);  
11 };  
12  
13 const writeData = (data) => {  
14   fs.writeFileSync(dataFile, JSON.stringify(data, null, 2));  
15 };  
16
```

(POST Endpoint)\*\*: Handles adding a new item to the list by reading the current data, appending the new item, and writing back to the file.

```
router.post('/', (req, res) => {  
  const items = readData();  
  const newItem = req.body;  
  newItem.id = items.length + 1;  
  items.push(newItem);  
  writeData(items);  
  res.status(201).json({ message: "item Added !!!!!", newItem });  
});
```

(GET Endpoint)\*\*: Fetches and returns all items.

```
26 router.get('/', (req, res) => {  
27   const items = readData();  
28   res.status(200).json(items);  
29 }  
30 );
```

(GET by ID)\*\*: Retrieves a specific item by ID from the list.

```
31 router.get('/:id', (req, res) => {  
32   const items = readData();  
33   const item = items.find(i => i.id === parseInt(req.params.id));  
34   if (item) {  
35     res.status(200).json(item);  
36   } else {  
37     res.status(404).json({ message: 'item not found by this id !' });  
38   }  
39 }  
40 );
```

(PUT Endpoint)\*\*: Updates an item by ID if it exists, using the provided data.

```

router.put('/:id', (req, res) => {
  const items = readData();
  const index = items.findIndex(i => i.id === parseInt(req.params.id));

  if (index !== -1) {
    items[index] = { ...items[index], ...req.body };
    writeData(items);
    res.status(200).json({ message: 'item has been updated successfully', updatedItem: items[index] });
  } else {
    res.status(404).json({ message: 'there is no items by this id ' });
  }
});

```

(DELETE Endpoint)\*\*: Deletes an item by ID if it exists, updating the list accordingly.

```

router.delete('/:id', (req, res) => {
  const items = readData();
  const filteredItems = items.filter(i => i.id !== parseInt(req.params.id));

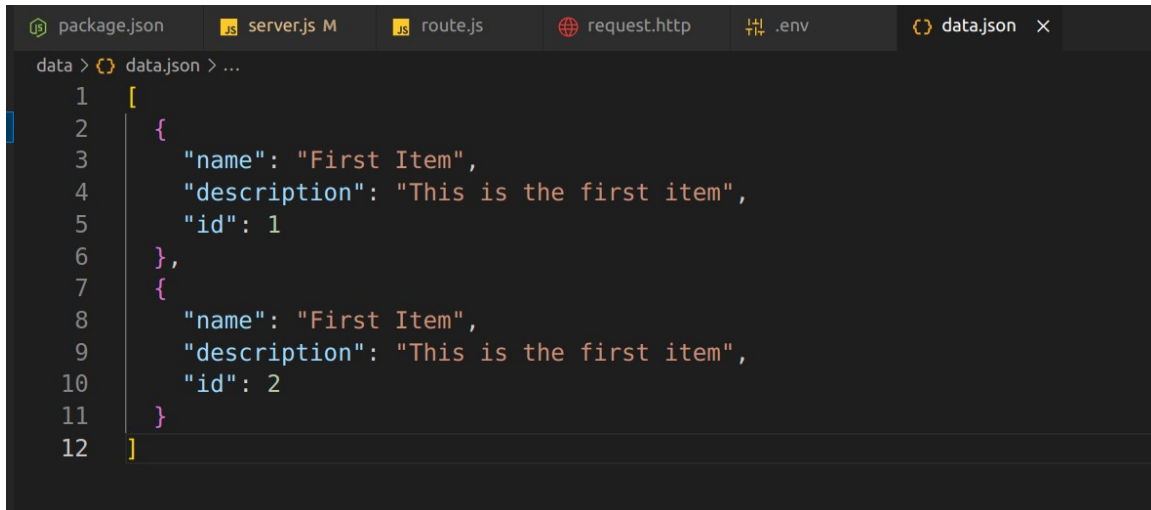
  if (filteredItems.length !== items.length) {
    writeData(filteredItems);
    res.status(200).json({ message: 'item deleted successfully' });
  } else {
    res.status(404).json({ message: "there is no item by this id !" });
  }
});

module.exports = router;

```

### 5.3 data/data.json

The `items.json` file is used for data storage. It keeps the list of items as a JSON array, which makes the application simple and file-based. This design allows for quick prototyping, where the focus is on CRUD operations rather than setting up a database.



```
data > data.json > ...
1  [
2    {
3      "name": "First Item",
4      "description": "This is the first item",
5      "id": 1
6    },
7    {
8      "name": "First Item",
9      "description": "This is the first item",
10     "id": 2
11   }
12 ]
```

## 6. CRUD Operations in Detail

- **Create (POST)**: Allows the user to add new items. A unique ID is assigned to each item based on the current length of the items array.
- **Read (GET)**: There are two GET endpoints—one to retrieve all items and another to retrieve an item by ID.
- **Update (PUT)**: Modifies an existing item identified by its ID. The updated data is provided in the request body.
- **Delete (DELETE)**: Removes an item identified by its ID from the list.

## 7. Conclusion

This Express.js CRUD application demonstrates the implementation of RESTful APIs using Express Router, a modular approach for routing, and a file-based storage system. The architecture maintains separation of concerns, with clear divisions between the main application logic, routing logic, and data storage. This project serves as a basic but comprehensive guide to developing a simple backend using Express.js.