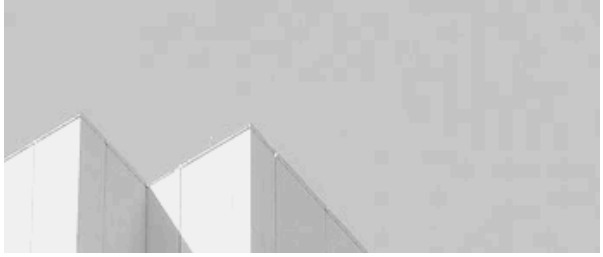
ENSA KHOURIBGA ÉCOLE NATIONALE DES SCIENCES APPLIQUÉES



جامعة السلطان مولاي سليمان
Université Sultan Moulay Slimane

21/10/2024

Rapport TP2 :

Express.JS

Réalisé Par :
Sad FETTAH

1. What is Express.js and What Can We Make with It?

Express.js is a minimal, flexible, and fast Node.js web application framework. It simplifies the process of building web servers and APIs by providing robust tools and utilities for handling HTTP requests, routing, middleware integration, and much more

What Can We Build with Express.js?

- Web Applications:
- RESTful APIs
- Real-Time Applications

2. What are Middlewares and How are They Used in Express.js?

Middlewares in Express.js are functions that process requests before they reach the final route handler. They have access to the request (req), response (res), and the next() function. They can modify requests/responses, run code, or end the request-response cycle.

How Are Middlewares Used in Express.js?

Middlewares are used to:

- Modify requests (e.g., parsing JSON, adding headers).
- Handle tasks like logging, authentication, or error handling.
- Chain execution using next() to move through middleware layers. They can be applied globally or to specific routes and run in the order they are defined.

1. Server Setup and Initialization

```js
const express = require('express');
const app = express();
```

- express is imported, allowing the use of the Express framework.
- app is initialized as an instance of the Express application, which will handle incoming requests and responses.

2. Middleware: Parsing JSON

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

express.json() is middleware that parses incoming request bodies in JSON format. This is necessary to handle data sent in POST and PUT requests.

3. Server Setup

```
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

The server is set to listen on PORT 3000. When the server starts, it logs a message indicating the URL where the server is running.

4 . Creating a New Item (POST request)

```
app.post('/items', (req, res) => {
  const item = req.body;
  items.push(item);
  res.status(201).send(item);
});
```

- Endpoint: POST /items
- This route allows the client to send a JSON object (req.body) representing an item to the server. The item is added to the items array.
- The response status is 201 (Created) and the newly created item is sent back to the client.

5 . Retrieving All Items (GET request)

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

- Endpoint: GET /items
- This route sends back all the items stored in the items array. No filtering or processing is applied; it simply returns the current state of the array.

6 . Retrieving a Single Item by ID (GET request)

```
app.get('/items/:id', (req, res) => {
  const id = parseInt(req.params.id, 10);
  const item = items.find(i => i.id === id);
  if (item) {
    res.send(item);
  } else {
    res.status(404).send('Item not found');
  }
});
```

- Endpoint: GET /items/:id
- This route retrieves a single item based on the id parameter in the URL. The id is extracted from req.params.id and parsed as an integer.
- It searches for the item using find(). If found, the item is returned. Otherwise, it returns a 404 status with the message 'Item not found'.

7 . Updating an Item by ID (PUT request)

```
app.put('/items/:id', (req, res) => {
  const id = parseInt(req.params.id, 10);
  const itemIndex = items.findIndex(i => i.id === id);

  if (itemIndex !== -1) {
    items[itemIndex] = { ...items[itemIndex], ...req.body };
    res.send(items[itemIndex]);
  } else {
    res.status(404).send('Item not found');
  }
});
```

- Endpoint: PUT /items/:id
- This route updates an existing item by its id. First, the id is parsed from the URL, and the itemIndex is found using findIndex().
- If the item is found, it updates the item's properties using object spread syntax {...items[itemIndex], ...req.body}. This merges the existing item with the updated properties sent in the request body.
- If the item is not found, a 404 status is returned.

8 . Deleting an Item by ID (DELETE request)

```
app.delete('/items/:id', (req, res) => {
  const id = parseInt(req.params.id, 10);
  const itemIndex = items.findIndex(i => i.id === id);

  if (itemIndex !== -1) {
    const deletedItem = items.splice(itemIndex, 1);
    res.send(deletedItem);
  } else {
    res.status(404).send('Item not found');
  }
});
```

- Endpoint: DELETE /items/:id
- This route deletes an item by its id. The id is parsed, and findIndex() is used to find the index of the item in the array.
- If the item exists, it is removed from the items array using splice(). The removed item is returned in the response. If the item is not found, a 404 status is sent.


Best regards,