

Rapport du TP (Express Js)

Préparé par : Boudehane Salma

1. What is Express.js and What Can We Make With It

Express.js is a web application framework built on top of Node.js, designed to simplify and streamline the process of building robust and scalable web applications. It provides a minimalistic approach while offering flexible, powerful features to handle the complexities of modern web development.

Key Features of Express.js:

- **Minimalistic:** Express.js doesn't enforce a specific structure or pattern, allowing developers to structure their applications as they see fit.
- **Middleware Support:** Express is built around middleware functions, making it easy to extend or modify behavior at any point in the request-response cycle.
- **Routing:** Express offers an intuitive and powerful routing system to define URL patterns and handle HTTP requests.
- **Templating Engines:** You can integrate templating engines like Pug, EJS, and Handlebars for dynamic HTML rendering.
- **Database Integration:** It can work seamlessly with various databases like MongoDB, MySQL, PostgreSQL, etc.
- **Extensibility:** The framework is easily extensible with various libraries and tools available in the Node.js ecosystem.

What Can You Build with Express.js?

- **Web Applications:** You can develop full-stack or backend-focused web applications, handling server-side logic, API routes, sessions, and more.
- **RESTful APIs:** Express is widely used to build RESTful APIs for web, mobile, and other clients (such as single-page applications built with React, Angular, or Vue.js).
- **Microservices:** Express is ideal for microservice architecture, where different parts of the system are broken into small, independent services.
- **Real-time Applications:** Combined with technologies like WebSockets, Express.js can also be used for real-time applications like chat apps or collaborative tools.

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

Middleware in Express.js refers to functions that have access to the request (req), the response (res), and the next function in the application's request-response cycle. They allow for handling various aspects of HTTP requests, such as logging, authentication, or modifying the request/response objects before passing them along the chain of handlers.

Middleware functions can be used to:

- Execute any code before passing the request to the next handler.
- Modify the request or response objects.
- End the request-response cycle (e.g., sending a response).
- Call the next() function to pass control to the next middleware function in the stack.

Middleware can be applied globally (for all routes) or to specific routes.

Types of Middleware in Express.js:

1. **Application-level Middleware:** These are functions that run during the application's request-response cycle and are registered globally or to specific routes.
2. **Router-level Middleware:** Middleware applied to specific routes using the Express router.
3. **Built-in Middleware:** Express provides built-in middleware like `express.json()` and `express.static()` to handle JSON payloads and serve static files, respectively.
4. **Third-party Middleware:** Express can utilize third-party middleware like `morgan` for logging, `helmet` for securing HTTP headers, or `cors` to enable Cross-Origin Resource Sharing.

How Middleware Works in Express.js

1. **Request Flow:** When a client sends a request to the server, the request will first pass through any global middlewares (e.g., `app.use()` middlewares).
2. **Execution Order:** Middleware is executed in the order it's defined. The request passes through each middleware in sequence.
3. **Route Handlers:** After going through all relevant middlewares, the request reaches the route handler (e.g., `app.get()` or `app.post()`).
4. **Response Flow:** Once the request handler completes, any subsequent middlewares (if any) will handle the response before sending it back to the client.

Creating a Simple CRUD Application

Initialize a Node.js Project

```
PS C:\Users\HP\Desktop\node\tp_express> npm init -y
Wrote to C:\Users\HP\Desktop\node\tp_express\package.json:

{
  "name": "tp_express",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

Install Express

```
PS C:\Users\HP\Desktop\node\tp_express> npm install express
>>

added 65 packages, and audited 66 packages in 10s

13 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\HP\Desktop\node\tp_express> |
```

Set Up Express: Run the Server Using app.listen

In this snippet, we set up an Express server that listens on port 3000 and uses a JSON parser middleware to handle incoming requests with JSON bodies.

```
JS index.js ×
JS index.js > ...
1  const express = require('express');
2  const app = express();
3  app.use(express.json()); // Middleware to parse JSON bodies
4
5  // Start the server
6  const PORT = 3000;
7  app.listen(PORT, () => {
8    console.log(`Server is running on port ${PORT}`);
9  });
10 |
```

Create a POST Endpoint to Add Items

This POST endpoint listens at /items, receives an item in the request body, adds it to the items array, and responds with the newly added item.

```
11 // add a post endpoint to add items
12 let items = [];
13
14 app.post('/items', (req, res) => {
15   const newItem = req.body;
16   items.push(newItem);
17   res.status(201).json(newItem);
18 });
```

Create a GET Endpoint to Retrieve All Items

This GET endpoint returns all items stored in the items array when a request is made to /items.

```
20 //Create a GET Endpoint to Retrieve All Items
21 app.get('/items', (req, res) => {
22   res.json(items);
23 });
```

Create a GET Endpoint by ID to Get a Specific Item

This GET endpoint fetches an item by its id from the array. If found, it returns the item; otherwise, it responds with a 404 error.

```
26 //Create a GET Endpoint by ID to Get a Specific Item
27 app.get('/items/:id', (req, res) => {
28   const itemId = parseInt(req.params.id);
29   const item = items.find(i => i.id === itemId);
30   if (item) {
31     res.json(item);
32   } else {
33     res.status(404).send('Item not found');
34   }
35 });
```

Create a PUT Endpoint to Update an Existing Item

This PUT endpoint allows updating an existing item by ID. It searches for the item and updates it with the request body if found.

```
38 //Create a PUT Endpoint to Update an Existing Item
39 app.put('/items/:id', (req, res) => {
40   const itemId = parseInt(req.params.id);
41   const itemIndex = items.findIndex(i => i.id === itemId);
42
43   if (itemIndex !== -1) {
44     items[itemIndex] = { ...items[itemIndex], ...req.body };
45     res.json(items[itemIndex]);
46   } else {
47     res.status(404).send('Item not found');
48   }
49 });
```

Create a DELETE Endpoint to Delete an Item

This DELETE endpoint removes an item by its ID. If the item is found, it deletes the item and responds with a 204 status code indicating successful deletion.

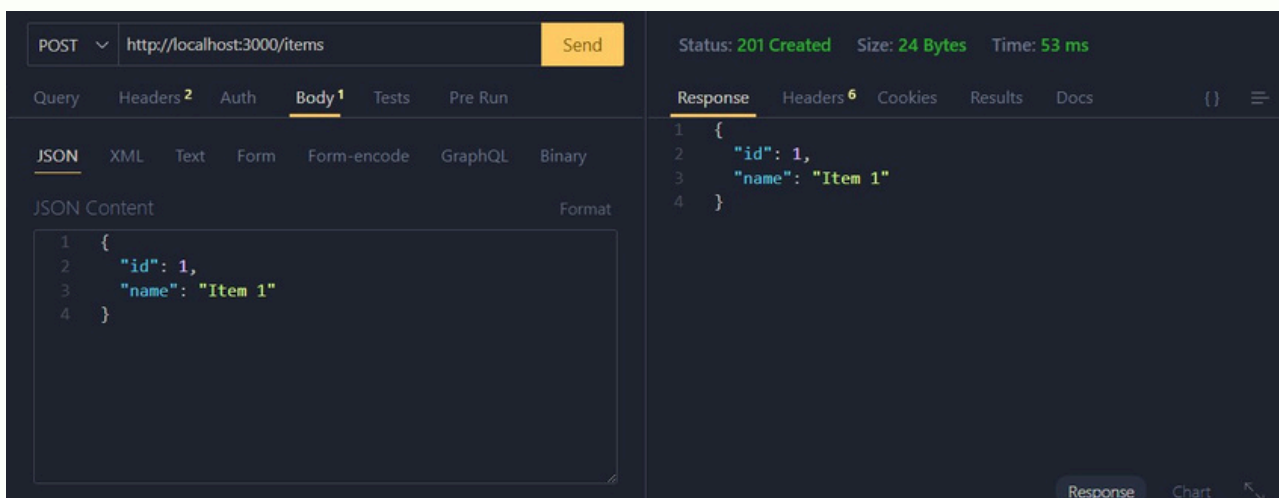
```
52 //Create a DELETE Endpoint to Delete an Item
53 app.delete('/items/:id', (req, res) => {
54   const itemId = parseInt(req.params.id);
55   const itemIndex = items.findIndex(i => i.id === itemId);
56
57   if (itemIndex !== -1) {
58     items.splice(itemIndex, 1);
59     res.status(204).send(); // 204 No Content response
60   } else {
61     res.status(404).send('Item not found');
62   }
63 });
```

Start the Server

```
PS C:\Users\HP\Desktop\node\tp_express> node index.js
Server is running on port 3000
```

Test the Endpoints Using Postman

POST /items:

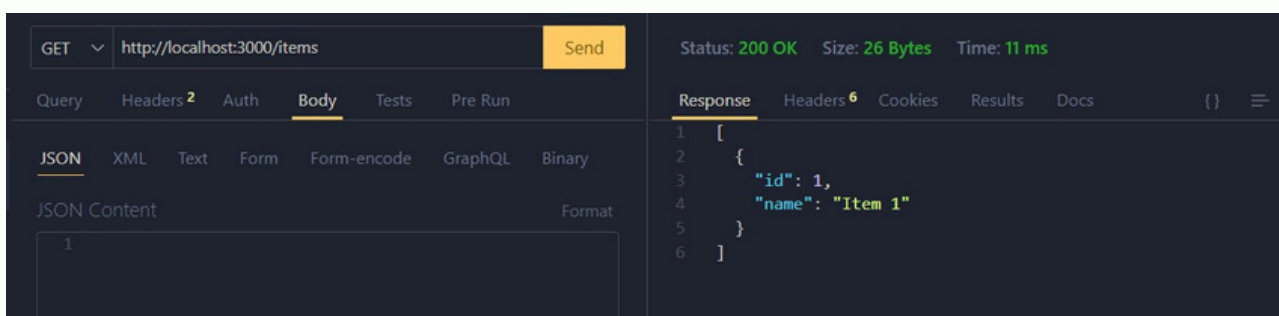


A screenshot of the Postman application showing a POST request to `http://localhost:3000/items`. The request body is a JSON object: `{ "id": 1, "name": "Item 1" }`. The response status is `201 Created`, with a size of `24 Bytes` and a time of `53 ms`. The response body is a JSON object: `{ "id": 1, "name": "Item 1" }`.

Method	URL	Status	Size	Time
POST	http://localhost:3000/items	201 Created	24 Bytes	53 ms

```
{
  "id": 1,
  "name": "Item 1"
}
```

GET /items:



A screenshot of the Postman application showing a GET request to `http://localhost:3000/items`. The response status is `200 OK`, with a size of `26 Bytes` and a time of `11 ms`. The response body is a JSON array containing one object: `[{ "id": 1, "name": "Item 1" }]`.

Method	URL	Status	Size	Time
GET	http://localhost:3000/items	200 OK	26 Bytes	11 ms

```
[
  {
    "id": 1,
    "name": "Item 1"
  }
]
```


GET /items/

GET

http://localhost:3000/items/1

Send

Query

Headers2

Auth

Body

Tests

Pre Run

JSON

XML

Text

Form

Form-encode

GraphQL

Binary

JSON Content

Format

1

Status: 200 OK

Size: 24 Bytes

Time: 9 ms

Response

Headers6

Cookies

Results

Docs

{}

≡

1

{

2

"id": 1,

3

"name": "Item 1"

4

}

PUT /items/

PUT

http://localhost:3000/items/1

Send

Query

Headers2

Auth

Body1

Tests

Pre Run

JSON

XML

Text

Form

Form-encode

GraphQL

Binary

JSON Content

Format

1

{

2

"name": "Updated Item 1"

3

}

Status: 200 OK

Size: 32 Bytes

Time: 9 ms

Response

Headers6

Cookies

Results

Docs

{}

≡

1

{

2

"id": 1,

3

"name": "Updated Item 1"

4

}

DELETE /items/

DELETE

http://localhost:3000/items/1

Send

Query

Headers2

Auth

Body

Tests

Pre Run

JSON

XML

Text

Form

Form-encode

GraphQL

Binary

JSON Content

Format

1

Status: 204 No Content

Size: 0 Bytes

Time: 6 ms

Response

Headers3

Cookies

Results

Docs

{}

≡

1