Report TP Express.JS

Computer Science and Data Engineering

Prepared by:

ARI Chaymaa

Part 1

1. Definition of Express JS

Express is a web application framework for Node.js that provides a wide range of features for building both web and mobile applications. It supports the development of single-page, multi-page, and hybrid applications, serving as an abstraction layer over Node.js to simplify server and route management.

→ Key advantages of Express:

- Designed to facilitate the creation of APIs and web applications with ease.
- Significantly reduces development time, often by half, while maintaining efficiency.
- Built using JavaScript, making it accessible for beginners with little to no prior programming experience.
- Helps many new developers break into web development due to JavaScript's simplicity.

→ Express was developed for Node.js to offer:

- Time efficiency
- High speed
- Cost-effectiveness
- Ease of learning
- Support for asynchronous operations

→ What Can We Make with Express.js?

- **RESTful APIs**: Express.js is widely used to create RESTful APIs for handling CRUD (Create, Read, Update, Delete) operations.
- **Web Applications**: You can build dynamic websites by handling routing, middleware, and templating engines (e.g., Pug, EJS).
- **Real-time Applications**: By combining it with other tools (like Socket.io), you can create real-time applications such as chat applications.
- **Microservices**: Express.js is suitable for building small, modular, and independent services that communicate over HTTP.
- **Single-Page Applications (SPAs)**: It is often used as a backend to serve API endpoints for SPAs built with front-end frameworks like React, Angular, or Vue.js.

2. Definition of MiddleWares

Middleware is a request handler that allows you to intercept and manipulate requests and responses before they reach route handlers. They are the functions that are invoked by the Express.js routing layer.

It is a flexible tool that helps add functionalities like logging, authentication, error handling, and more to Express applications. To learn how to build efficient middleware in full-stack applications, the Full Stack Development with Node JS course covers middleware integration and request handling in Node.js

→ How we use Middlewares in Express JS:

Middleware functions can be used for various purposes, such as logging, authentication, error handling, and serving static files. They are defined using the app.use() method or as route-specific middleware.

- **Example 1: Logging Middleware**

The logging middleware in Express.js operates on the principle of intercepting incoming requests to the server. It captures key information about each request, such as the HTTP method (GET, POST, etc.) and the requested URL. This information is then logged (usually to the console or a log file) for monitoring and debugging purposes.

- **Example 2: Authentication Middleware**

Authentication middleware is designed to protect certain routes in an Express.js application by verifying whether a user is authorized to access specific resources. The principle behind this middleware involves checking the incoming request for valid authentication credentials, such as tokens or session identifiers.

Part 2: Simple CRUD Application

1. Create a project directory named ExpressTP :
2. Initialize a Node.js Project

Run **npm init -y** in the project directory to generate a **package.json** file that holds important project information and dependencies.

```
PS E:\IID3\TechnologieJS_ProgramMobile\TP_Express\Expr
essTp> npm init -y
Wrote to E:\IID3\TechnologieJS_ProgramMobile\TP_Expres
s\ExpressTp\package.json:

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

3. Install Express using **npm Install express**
4. create a file app.js to set up Express

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

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

// Start the server at the port
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

- The first part shows us the imports of the Express.js library (express) and initializes the application (app).

- The line **app.use(express.json()) adds middleware** to the app that automatically parses incoming requests with JSON payloads. This allows the server to read and process JSON data from the request body, making it accessible via **req.body** in routes like POST or PUT. This middleware is essential when working with APIs that expect to receive data in JSON format.

- The second part sets up a basic server using Express.js in a Node.js environment. It begins by defining a constant PORT set to **3000**, which determines the port number on which the server will listen for incoming requests. **The app.listen(PORT, () => {...});** method starts the server and allows it to accept connections on port 3000. Once the server is successfully running, the callback function is triggered, logging the message "Server is running on port 3000" to the console. This confirms that the server has been launched and is operational

5. Create a Post Endpoint allow us to add items to a local variable:

```
let items = [];

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

The code defines a POST endpoint in an Express.js application that allows users to add new items to a local array called "**items**". When a request is made to the **"/add"** endpoint, the item sent in the request body is extracted, added to the **"items"** array, and a response with status code **201** (indicating successful creation) is sent back along with the newly added item. This provides a simple way to store and manage items locally in the server's memory during runtime.

6. Create a GET Endpoint allow us to retrieve all items.

```
//Get Endpoint to fetch the items = it means to  get all the elements
app.get('/items', (req, res) => {
    res.status(200).send(items);
  });
```

The code defines a GET route at the **/items** endpoint that allows users to fetch all the elements stored in the items array. When the endpoint is accessed, the server sends an HTTP response with a status code of **200** (indicating success), along with the current

contents of the items array. This provides a way to retrieve all previously added items in the application.

7. Create a GET Endpoint by ID allow us to get a specific item.

```
//Get Endpoint to find the item with a specific id
app.get('/items/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const item = items.find(i => i.id === id);
    if (item) {
      res.status(200).send(item);
    } else {
      res.status(404).send({ message: "Item not found" });
    }
  });
```

The code sets up a GET route at the **/items/:id** endpoint, where **:id** represents a dynamic URL parameter corresponding to the id of an item. When the endpoint is accessed, the id parameter is extracted from the request, converted to an integer, and used to search for an item in the items array. If an item with the matching id is found, it is returned with a **200 status code**. If no such item exists, the server responds with a **404 status** code and a message indicating that the item was not found. This allows users to fetch specific items by their unique id.

8. Create a PUT Endpoint allow us to update an existing item.

```
//Put Endpoin to modify an item with a specific id
app.put('/items/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const index = items.findIndex(i => i.id === id);
    if (index !== -1) {
      items[index] = req.body;
      res.status(200).send(items[index]);
    } else {
      res.status(404).send({ message: "Item not found" });
    }
  });
```

The code creates a PUT route at the **/items/:id** endpoint to update an existing item with a specific **id**. When a request is made, the **id** from the URL is extracted and converted to an integer. The server then searches the **items** array for an item with the corresponding **id**. If the item is found, it is replaced with the data sent in the request body, and the updated item is returned with a **200 status** code. If the item is not found, a **404 status** code is returned along with a "**Item not found**" message. This enables users to modify the details of an item based on its unique **id**.

9. Create a DELETE Endpoint allow us to delete an item

```
//Delete an item by id
app.delete('/items/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const index = items.findIndex(i => i.id === id);
    if (index !== -1) {
      items.splice(index, 1);
      res.status(200).send({ message: "Item deleted" });
    } else {
      res.status(404).send({ message: "Item not found" });
    }
  });
```

The code sets up a DELETE route at the **/items/:id** endpoint, allowing users to delete an item by its **unique id**. The id is extracted from the request parameters and used to find the item's index in the items array. If the item is found, it is removed from the array using the splice method, and a success message with a **200 status** code is returned. If the item is not found, a **404 status** code is sent along with a message indicating that the item could not be located. This allows users to delete specific items by **id**.

10. Start the Server In the terminal:

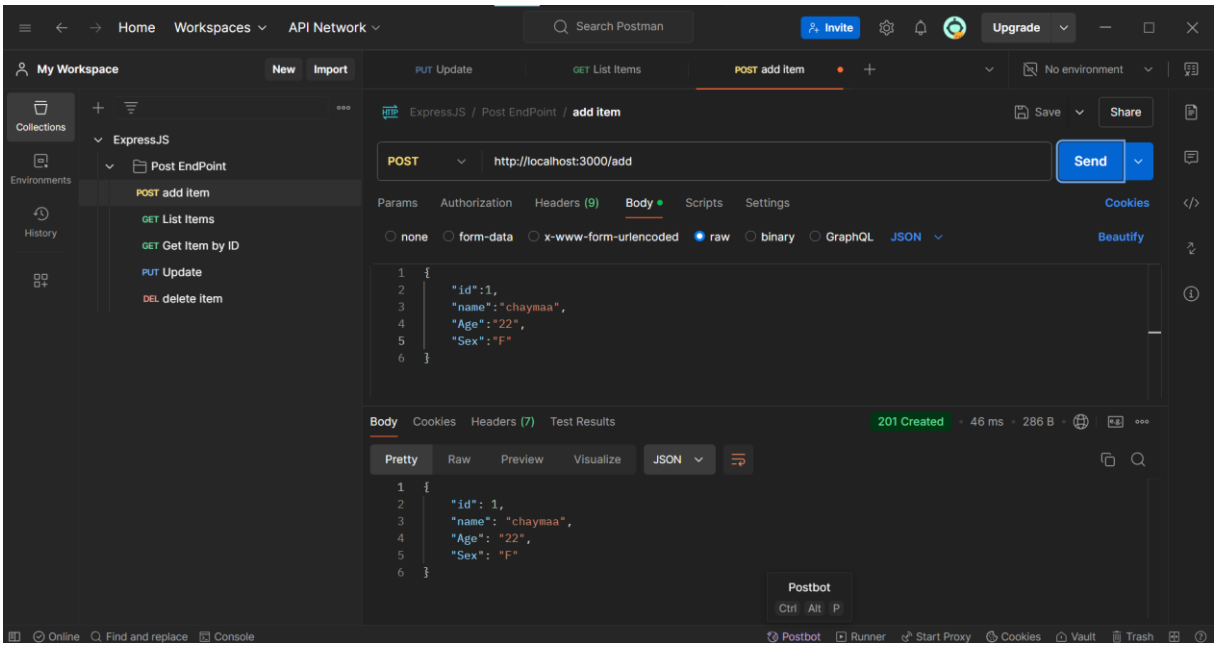
```
PS E:\IID3\TechnologieJS_ProgramMobile\TP_Express\ExpressTp> node app.js
>>
Server is running on port 3000
```

11. Test the Endpoint using Postman:
    - Add item:

In Postman, to add a new item using the POST method, I start by entering the appropriate URL for the API endpoint (**http://localhost:3000/add**) in the request field. Then, select the POST method from the dropdown next to the URL field, since I'm sending data to the server. Next, navigate to the "Body" tab, choose the "raw" option, and set the format to "JSON" from the dropdown menu on the right. In the body section, I wrote the JSON object representing the item I want to add:

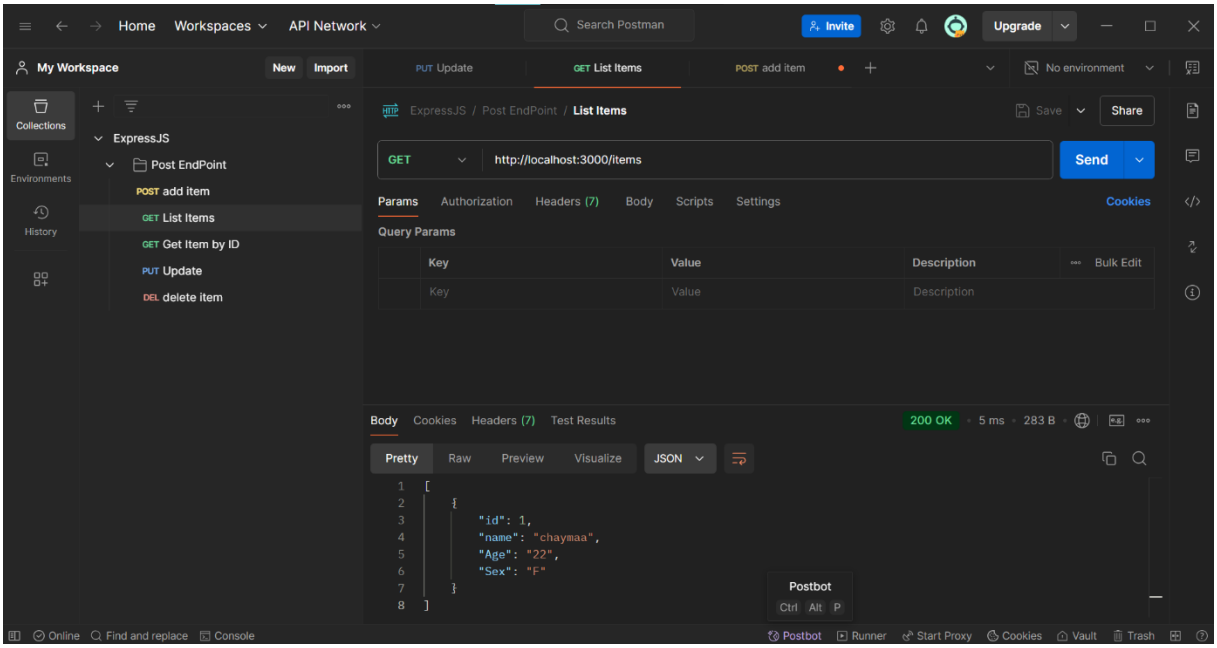
{"id":1, "name":"chaymaa", "Age":"22", "Sex":"F" }

Once the data is ready, I click "Send" to submit the request. If successful, I receive the item data I submitted.



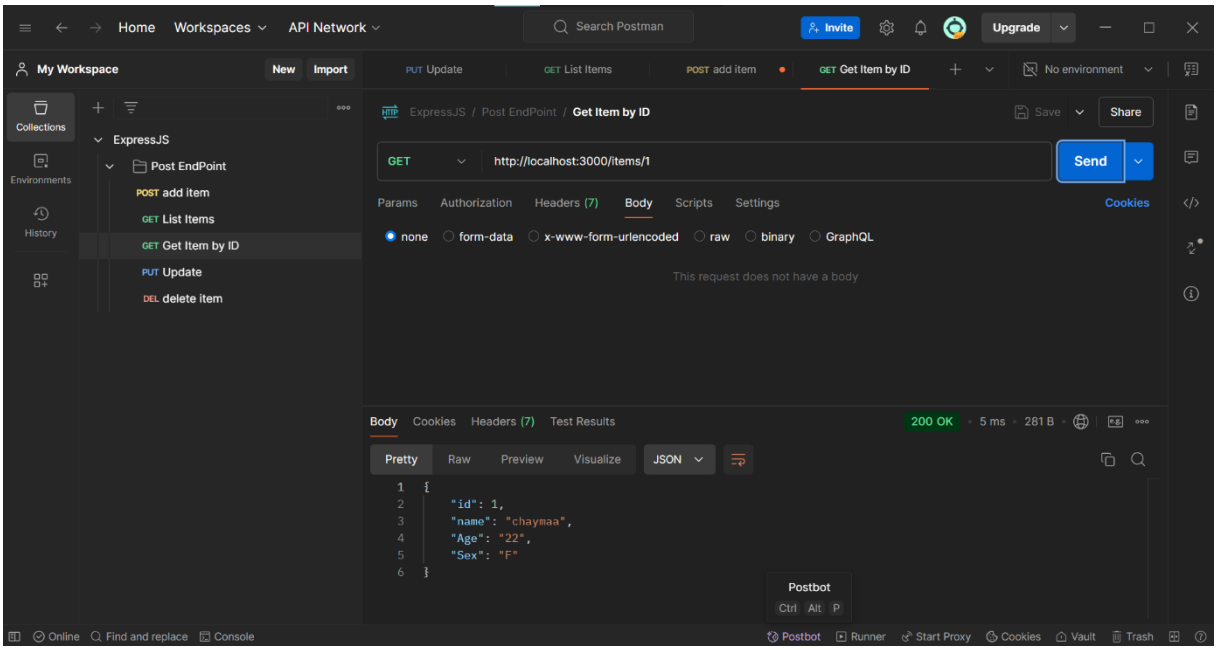
- Get items:

To fetch the list of items using the GET endpoint in Postman, I enter the URL for the API endpoint (`**http://localhost:3000/items**`) and select the GET method. Then, I click "Send" to submit the request. If successful, I will receive a 200 status code along with a response containing the list of all items stored in the array. This allows me to see all the items added to the application.

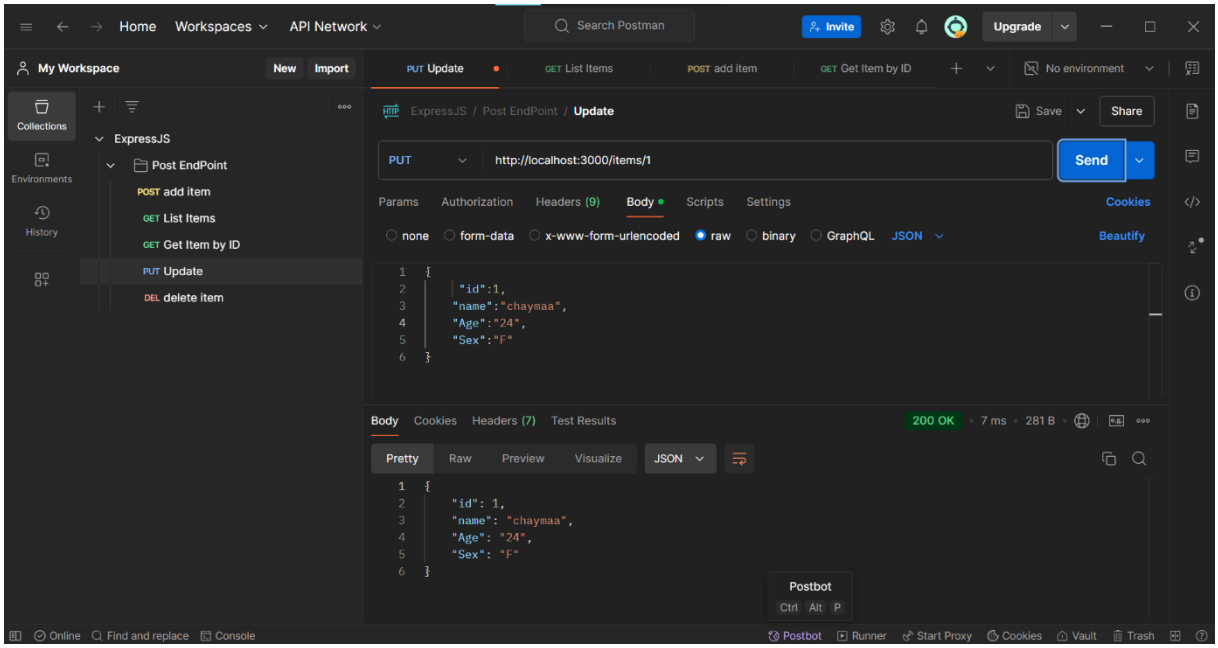


- Get item by id :

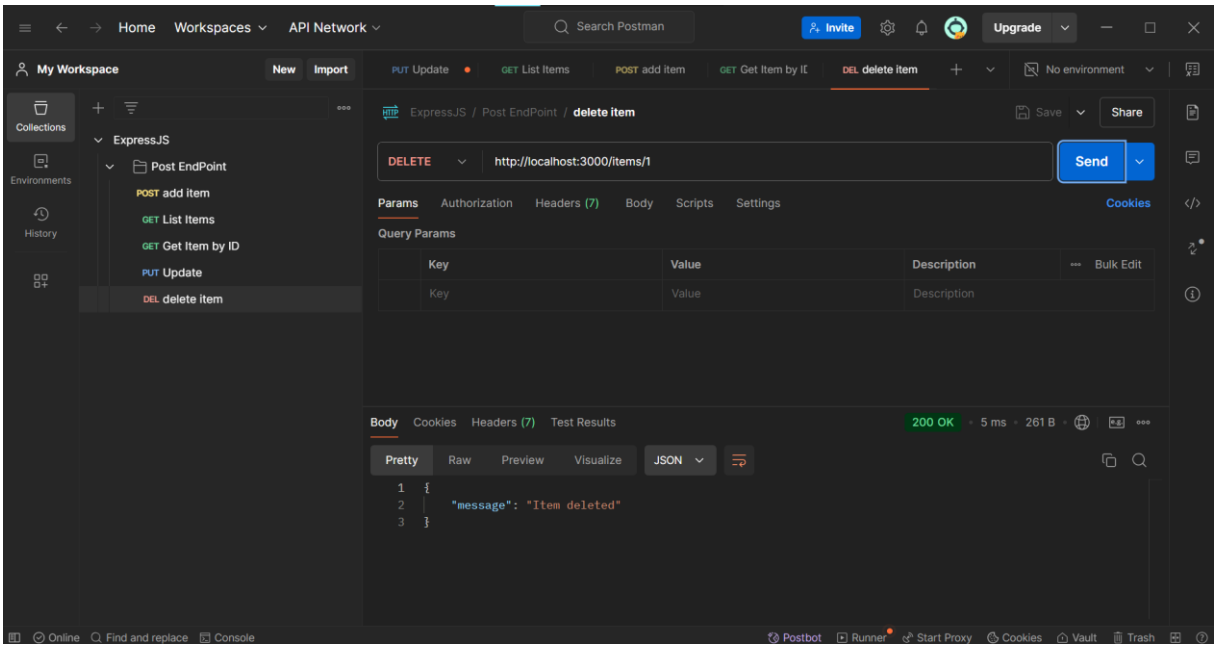
If I want to fetch a specific item, I enter the URL for the API endpoint with the specific item ID (**http://localhost:3000/items/1**) and follow the same steps.



- Update item



- Delete Item


Home
Workspaces
API Network
Search Postman
Invite
Upgrade
My Workspace
New
Import
Collections
Environments
History
PUT Update
GET List Items
POST add item
GET Get Item by ID
DEL delete item
No environment
ExpressJS
Post EndPoint
POST add item
GET List Items
GET Get Item by ID
PUT Update
DEL delete item
ExpressJS / Post EndPoint / delete item
Save
Share
DELETE
http://localhost:3000/items/1
Send
Params
Authorization
Headers (7)
Body
Scripts
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Body
Cookies
Headers (7)
Test Results
200 OK
5 ms
261 B
Pretty
Raw
Preview
Visualize
JSON
{
"message": "Item deleted"
}
Online
Find and replace
Console
Postbot
Runner
Start Proxy
Cookies
Vault
Trash