WEBTEAM: Mini-Project 2

RESTFUL API - CURRENTME

rywi, lukj, jato, jvia, jskb

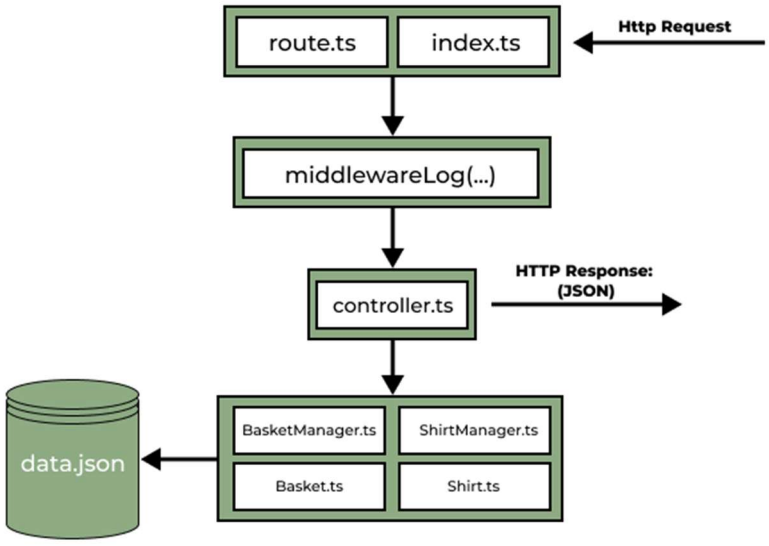
The Meme Team

Table of Contents

Introduction

For the second iteration of our project, we implemented a RESTful API to handle the data that a customer would interact with, such as filtering their search based on their favorite cryptocurrency. On the server side, we implemented a RESTful API that would allow us to manage our items within a shop product line. We used a three-step process to determine what resources would need to be implemented. First, we took the time to identify our resources and their respective URI's or uniform resource identifiers. For our case, we have a web shop meaning that our primary entities are customer baskets and our product line. After establishing our resources, we then mapped out what potential resource operations could be. And lastly, we defined the details of each operation, including their parameters, status codes and data structures. However, before going into more detail about this process, we want to take the time to explain our system architecture and the non-functional components of our project.

System architecture



Our system is built using the three-layered architecture together with the separations of concerns design pattern. Our view is the request and response object received and returned by the system by the router. For our Restful API, every route is passed through a single middleware. The view talks only to the controller which in our case consists of a single file,

controller.ts. The controller calls the model which represents the data in our web shop, and these services are contained within the following files, *shirts.json*, *baskets.json*, *Shirt.ts*, *ShirtManager.ts*, *BasketManager.ts*.

Non-Functional Requirements

It is important to note that for this project we developed our RESTful API in TypeScript, a superscript of JavaScript. Our decision to switch to TypeScript was based on the idea that having type declaration, especially for server-side API's, make our ability to check for errors and debug potential issues easier. The project also used Node.js, which has a built-in web server implantation that can handle HTTP requests. Furthermore, Node.js is applicable for our project because our project is not CPU intensive. We are also using Express to create the middleware necessary for our API to parse JSON requests into responses and to use the router. Lastly, all data is stored in JSON format. This can be viewed in the sections *baskets.json* and *shirts.json.*

Functional Requirements

Now that the non-functional requirements have been explained, we will describe the three-step process of our functional requirements in detail as mentioned in our introduction. After identifying the resources or our web shop we determined that our resources required the following operations. First, a customer should be able to GET a list of categories available for our product lines. Thus, our resource path is */categories* and our documentation for this task can be found in the GET: /categories section found in our appendix. In the ShirtManager.ts file of our project, the method *getEveryCategory* realizes this request by showing the set of values associated important categories such as shirt color, top deals, and currency attributes. Second, we identified that a customer should also be able to GET a list of products sorted by a specific category. The documentation for this task can be found in the *GET: /products/category/{categoryName}*.

The customer should be able to GET the full product line, they are looking to buy products after all. Therefore, we have implemented this request, and the documentation for this resource can be found in *GET: /products*. In ShirtManager.ts, the method *getProducts*

realizes this request by returning the entire product line selection. A customer should also be able to search for an individual product and GET that product. Thus, the documentationfor this resource path can be found in *GET: /products/{productId}*. This functionality is found again in the ShirtManager.ts as *getProductsByCategory* and returns a specific product based on its ID as a criterion.

A customer will also want to add, update, and delete things from their individual basket. To implement these features, we have created *data.json* that tracks all customers' individual baskets. When a new costumer adds a shirt to their basket a POST request is made to create their basket. The documentation for this interaction can be found in the *POST: customer/{customerId}/basket*. The functionality is found in BasketManager.ts as *updateShirt* and creates a basket for a customer. After the shopping basket is created, a customer will want to view the products within the basket. Therefore, a customer will be able to GET the basket from out basket.json. The documentation for this path is explained in *GET: customer/{customerId}*, and its functionality is found in BasketManager.ts as *findBasket.*

It was also necessary to modify the items in the basket. Therefore, we implemented a function to update existing item established quantity, give the customer the ability to remove items from a basket, and add a new product to the basket. First, we can update the quantity of an existing item with a PUT request. The documentation can be found in the *PUT: Customers/{customerId}/basket/{itemId} PUT.* The functionality of this request can be found in the BasketManager.ts as *updateShirt*. Second, removing an item is found in the same file path, however removing an item is a DELETE request. The documentation can be found in the *DELETE: Customers/{customerId}/basket/{itemId}*. Finally, a customer can add a new item to the basket with a POST request. The functionality of the request can be found BasketManager.ts as *UpdateItem.* The documentation can be found in POST: /customers/{customerId}/basket/{itemId}.

Conclusion

In conclusion, our group set out to successfully create a RESTful API for our existing Web shop. We believe with our implementation accounts for the following requirements. We have functions that get the most important information about our products, products by categories, important information about products for a specific category. Furthermore, we can create a shopping basket based on a user. Products can be selected by a user and can be placed in their basket. There is an ability for a user to remove a product from their basket. And the user can view the shopping basket content at. With these requirements fulfilled we hope to have successfully created a RESTful API based on the requirements of this mini project that demonstrates what we have learned throughout the course.

GitHub Repository

https://github.itu.dk/lukj/FW-Project-Group3.git

Pathway Specification

Code	A	B	C	D	E
1	**Resource path**	**POST**	**GET**	**PUT**	**DELETE**
2	/categories	Error	Return the list of categories	Error	Error
3	/products/category/{categoryName}	Error	Return list of products under that category	Error	Error
4	/products	Error	Return all products	Error	Error
5	/products/{productId}	Error	Return a specific product	Error	Error
6	/customers/{customerId} /basket	Create a shopping basket for a specific customer	Return the shopping basket and its content for and to a specific customer	Error	Error
7	/customers/{customerId} /basket/{itemId}	Add new item to basket	Error	Update the quantity of the item in the basket	Remove the item from the basket

GET: /categories

Get List of Products Under Category

Path: /categories

Method: GET
Summary: Returns a list of categories

URL Params: -
Body: -

Success Response:
Code: 200 OK

Body Content:

{
categories: ["White", "Black", "Blue", "BitConnect", "Shiba", "Terra Luna", "Bitcoin", "true", " Solana", "Ripple"]
}

Error Response:
Code: 500 Server Error
Body Content:

{error: “Request message not understood by server”}

Sample Call:

```
let response = await fetch('/categories’,{
            method: GET,
            headers: {'Content-Type': 'application/json;charset=utf-8'},
            body: JSON.stringify(user)
});
```

GET: /categories/products

Get List of Products Under Category

Path: /products/category/{categoryName}

Method: GET
Summary: Return a list of products under that category

URL Params: categoryName: string, required in path

Body: -

Success Response:
Code: 200 OK

Body Content:
```
{
        categories: "Bitcoin",
              products:
           [{
             name: "dogeShirt", gender: "Male", color: "White",
               Coin: "Bitcoin", size: 10, img: "image.jpg",
               topDeal: false},...
           ]}
```

Error Response:

Code: 404 Not Found
Body Content:
```
{error: “No such category exist”}
```

Sample Call:

```
let categoryName: string = “Bitcoin”;
let response: Response = await fetch(`/products/${categoryName}`, {
            method: 'GET',
            headers: {'Content-Type': 'application/json;charset=utf-8'},
});
```

GET: /products

Finds all Products

Path: /products

Method: GET
Summary: Get all products

URL Params: -
Body: -

Success Response:

Code: 200 OK
Body Content:

```
products: [
        {name: "dogeShirt", gender: "Male", color: "White",
            Coin: "Bitcoin", size: 10, img: "image.jpg",
            topDeal: false},
        {name: "dogeShirt", gender: "Male", color: "White",
            Coin:"Bitcoin", size: 10, img: "image.jpg",
            topDeal: false}...]
```

Error Response:

Code: 500 Server Error
Body Content:
{error: “Request message not understood by server”}

Sample Call:

```
let response = await fetch('/products,{
            method: GET,
            headers: {'Content-Type': 'application/json;charset=utf-8'},
            body: JSON.stringify(user)
});
```

GET: /products/{productId}

Gets Information on Specific Products

Path: /products/{productId}

Method: GET
Summary: Retrieve information on a specific product.

URL Params: productId: number, required in path

Body: -

Success Response:
Code: 200 OK

Body Content:


```
      { "id": 4, "itemName": "dogeShirt", "color": "White", "currency":
"Bitcoin", "image": "/Project1/Images/dogetshirt.jpg", "price": 10, "topDeal":
false }
```


Error Response:

Code: 404 NOT FOUND
Body Content:
{error: "Item with ID: doesn't exist"}

Code: 500 Server Error
Body Content:
{error: "Request message not understood by server"}

Sample Call:

```
let productId = 1

let response = await fetch(´/products/{$productId}´,{

      method: 'GET',

      headers: {'Content-Type': 'application/json;charset=utf-8'},

});
```

GET: /customers/{customerId}

Finds All Products in a Customers Basket

Path: /customers/{customerId}

Method: GET
Summary: Get all products

URL Params: customerId: number, required in path

Body: Basket JSON data

Success Response:
Code: 200 OK

Body Content:
```
{ "customerId": 3, "contents": [ { "itemId": 5, "amount": 2 }, { "itemId": 7, "amount": 1 } ] }
```

Code: 204 No Content
Body Content: - { }

Error Response:

Code: 400 BAD REQUEST

Body Content:
```
{error: "Customer with id: `{$customerId}` already has a basket"}
```

Sample Call:

```
let user = {customerId: 12, customerName: 'Boris Johnson'};

let response = await fetch(``/Customer/{$customerId}/basket`,
{
      method: GET,

      headers: {'Content-Type': 'application/json;charset=utf-8'},

      body: JSON.stringify(user)
});

let user = await fetch(`/Customer/{$customerId}/basket`, options)
```

POST: /customers/{customerId}/basket

Make a Customers Shopping Bag

Path: /customers/{customerId}/basket

Method: POST
Summary: Create a shopping basket for a specific customer

URL Params: None

Body: Basket JSON data
```
{“customerId” : id, “basketItems”: []}
```

Success Response:
Code: 201 CREATED

Body Content:
```
message: {“Successfully created basket”}
```

Error Response:

Code: 400 BAD REQUEST

Body Content:
```
{error: “Bad Request”}
```

Sample Call:

```
let customerId = 12

let response = await fetch('/customers/{customerId}/basket,{

      method: POST,

      headers: {'Content-Type': 'application/json; charset=utf-8'},

});
```

PUT: /customers/{customerId}/basket/{itemId}

Update Customer Items Amount

Path: /customers/{customerId}/basket/{itemId}
|
Method: PUT
Summary: Update the quantity of the item in the basket

URL Params:

- customerId: number, for identifying the customer
- itemID: number, for identifying the item

Body:

```
{“customerId”: 1, “itemId”: 43, “quantity”: 4}
```

Success Response:

Code: 201 CREATED
Body Content: -

Error Response:

Code: 400 BAD REQUEST
Body Content:
```
{error: “Bad request”}
```

Sample Call:

```
let customerId = 1;
let itemId = 1;
let item = {“customerID”: 1, “itemID”: 1, “quantity”: 4}
let response = await fetch(`Customers/${customerId}/basket/${itemId`),
      {
      method: 'PUT',
      headers: {'Content-Type': 'application/json; charset=utf-8'},
      body: JSON.stringify(item)
});
```

DELETE: /customers/{customerId}/basket/{itemId}

Customer can Delete Items they put in Basket

Method: DELETE
Summary: Remove an item from the basket

URL Params:

- customerID: number, for identifying the customer
- itemID: number, for identifying the item

Body: Customer JSON data

```
{ “customerId”: 1, “itemId”: 43}
```

Success Response:

Code: 200 OK
Body Content:
{message: “Successfully removed item.” }

Error Response:

Code: 400 BAD REQUEST
Body Content:

```
{error: “Item with id: itemId does not exist in basket”}
```

Code: 401 BAD REQUEST

Body Content:

```
{error: “Customer with id: customerId does not exist in basket”}
```

Sample Call:

```
let item = { “customerId”: 2, “itemId” : 43}
let customerId = 1;
let itemId = 2;
let response = await fetch(`Customers/${customerId}/basket/${itemId}`,
      {
      method: 'DELETE',
      headers: {'Content-Type': 'application/json;charset=utf-8'},
      body: JSON.stringify(item)
});
```

POST: /customers/{customerId}/basket/{itemId}

Customer can add Item to Basket Path

Path: /customers/{customerId}/products/{itemId}

Method: POST
Summary: Summary: Add an item to the basket URL

URL Params:

- customerId: number, for identifying the
- customer itemId: number, for identifying the item

Body: Basket JSON data

```
{“customerId” : itemId, “quantity”: 1}
```

Success Response:
Code: 200 OK

Body Content:

```
{“customerId” : itemId, “quantity”: 1}
```

Error Response:
Code: 400 BAD REQUEST

Body Content:

```
{error: “Bad request”}
```

Sample Call:

```
let item = {“customerId” : itemId, “quantity”: 1}

let response = await fetch('/customers/{customerId}/basket/{$itemId},{
      method: POST,
      headers: {'Content-Type': 'application/json; charset=utf-8'},
      body: JSON.stringify(item)

});
```