Adding More Features to E-commerce Application

Rate Products

Implementing the functionality to rate products in our Node.js E-commerce API.

1. To enable rating functionality, we need to make changes in our product model. We already have three default products in our system, and we want to allow users to update or add ratings to these products.

2. Create a new function called `**rateProduct**` in the product model. This function will take three parameters: `userID` (the ID of the user who wants to rate the product), `productID` (the ID of the product to be rated), and `rating` (the rating out of five stars).

This code defines a static method `rateProduct` that is responsible for validating and updating the rating of a product by a user.

```
static rateProduct(userID, productID, rating) {
    // 1. Validate user and product
    const user = UserModel.getAll().find(
      (u) => u.id == userID
    );
    if (!user) {
      return 'User not found';
    }

    // Validate Product
    const product = products.find(
      (p) => p.id == productID
    );
    if (!product) {
      return 'Product not found';
    }
```

```
    // 2. Check if there are any ratings and if not then add ratings
array.
    if (!product.ratings) {
      product.ratings = [];
      product.ratings.push({
        userID: userID,
        rating: rating,
      });
    } else {
      // 3. check if user rating is already available.
      const existingRatingIndex = product.ratings.findIndex(
        (r) => r.userID == userID
      );
      if (existingRatingIndex >= 0) {
        product.ratings[existingRatingIndex] = {
          userID: userID,
          rating: rating,
        };
      } else {
        // 4. if no existing rating, then add new rating.
        product.ratings.push({
          userID: userID,
          rating: rating,
        });
      }
    }
  }
}
```

Explanation:
Let's go through the code step by step:

2.1) Validate user and product:

- The code first tries to find the user with the given `userID` in the `**UserModel**`. If the user is not found, it returns the string `''User not found''`.
- Next, it tries to find the product with the given `productID` in the `**products**` array. If the product is not found, it returns the string `''Product not found''`.

2.2) Check if there are any ratings and if not, add ratings array:

- If the `product` object does not have a `ratings` property (or if it exists but is `null` or `undefined`), a new empty array is assigned to `product.ratings`.
- Then, a new rating object is created with the `userID` and `rating` provided, and it is pushed to the `product.ratings` array.

2.3) Check if user rating is already available:

- If the `product.ratings` array exists and is not empty, the code searches for an existing rating in the array that matches the given `userID`. It does this by using the `findIndex` method, which returns the index of the first element that satisfies the provided condition.
- If an existing rating is found (i.e., `existingRatingIndex >= 0`), the code updates the rating for that user by replacing the existing rating object with a new rating object containing the updated `userID` and `rating`.

2.4) If no existing rating is found:

- If there is no existing rating for the user in the `product.ratings` array, a new rating object is created with the `userID` and `rating` provided, and it is pushed to the `product.ratings` array.

11. After implementing the rating functionality in the product model, we need to call this function from our controller and update the corresponding route.

This code defines a controller `rateProduct` in product.controller.js file:

```
rateProduct(req, res) {
    const userID = req.query.userID;
    const productID = req.query.productID;
    const rating = req.query.rating;
    const error = ProductModel.rateProduct(
      userID,
      productID,
      rating
    );
    if (error) {
      return res.status(400).send(error);
    } else {
      return res.status(200);
    }
  }
```

This code defines the route for **rate** api in product.routes.js:

```
productRouter.post(
  '/rate',
  productController.rateProduct
);
```

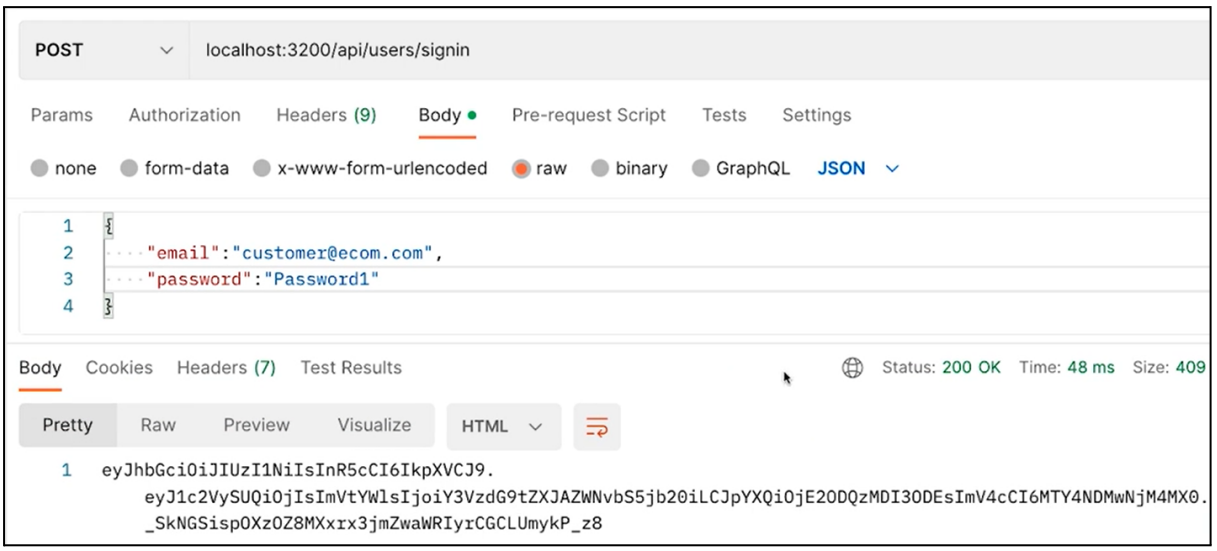
Testing Rate API

Testing the API we created to rate products in our e-commerce application. The rate product API takes three parameters: user ID, product ID, and rating. It applies the rating to the existing products in our application.

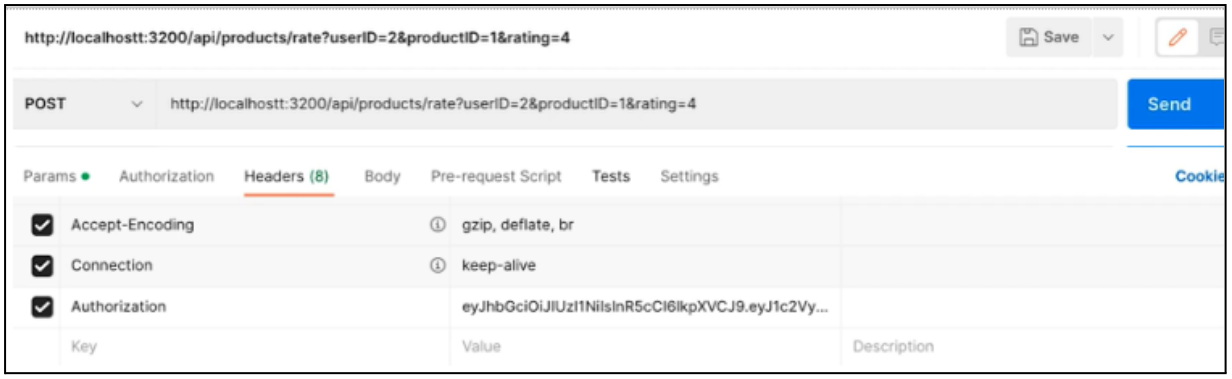
Testing Process

To test the API, we will use Postman. The following steps outline the testing process:

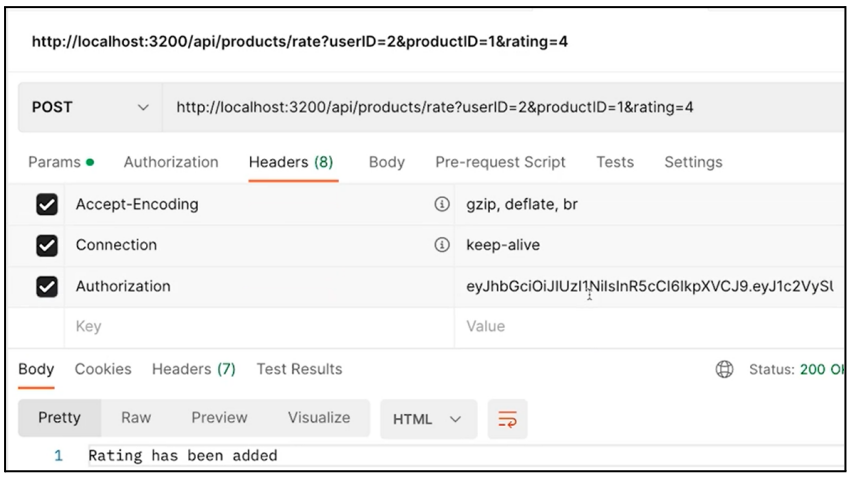
1. Run the API using the Node Server command.
2. Sign in as a customer user since customers should be rating the products.
3. Obtain the authentication token from the sign-in response.



4. In Postman, create a new request and set the request method to POST. Add the required query parameters: `userID`, `productID`, and `rating`. Include the authentication token in the request header using the "Authorization" key.



8. Send the request and observe the response.



Adding Cart Feature

Implementing an important feature required for an e-commerce application: managing the cart. The cart feature allows users to add items to their cart, which is a fundamental functionality in e-commerce applications. We will create an API to add items to the cart.

Creating the Cart Items Model

1. Create a new folder named "cartItems" to organize the cart-related code.
2. In the "cartItems" folder, create a model file named "cartItems.model.js" to define the
   Cart Item model.
3. The Cart Item model will have the following properties:
   - **productID**: Represents the ID of the product added to the cart.
   - **userID**: Represents the ID of the user who added the product to the cart.
   - **quantity**: Represents the quantity of the product in the cart.
4. Export a class named CartItemModel as the default export.

5. Define a constructor in the CartItemModel class to initialize the productID, userID, and quantity properties.
6. Optionally, you can add some existing cart items as examples by instantiating the CartItemModel class with predefined values.
7. Implement an add function in the CartItemModel class to add new items to the cart.
    - The add function takes productID, userID, and quantity as parameters.
    - Instantiate the CartItemModel class with the provided values and push it into an array of cart items.
    - Return the newly added cart item.

cartItems.model.js file:

```
// productID, userID, quantity
export default class CartItemModel {
  constructor(productID, userID, quantity, id) {
    this.productID = productID;
    this.userID = userID;
    this.quantity = quantity;
    this.id = id;
  }

  static add(productID, userID, quantity) {
    const cartItem = new CartItemModel(
      productID,
      userID,
      quantity
    );
    cartItem.id = cartItems.length + 1;
    cartItems.push(cartItem);
    return cartItem;
  }
}

var cartItems = [new CartItemModel(1, 2, 1, 1)];
```

Creating the Cart Items Controller

1. In the "cartItems" folder, create a controller file named "cartItemsController.js" to handle cart-related operations.

2. Export a class named `**CartItemsController**` as the default export.
3. Implement an `add` method in the `CartItemsController` class to handle the addition of items to the cart.
    - The `**add**` method takes a `request` and `response` as parameters.
    - Extract the `**productID**` and `**quantity**` from the request's query parameters.
    - Retrieve the `**userID**` from the request object's `**userID**` property (accessed from the token).
    - Import the `CartItemModel` and call its `add` function with the `productID`, `userID`, and `quantity` parameters.
    - Return a response with a status code of 201 (Created) and a message indicating that the cart is updated.

cartItems.controller.js file:

```
import CartIemModel from './cartItems.model.js';

export class CartItemsController {
  add(req, res) {
    const { productID, quantity } = req.query;
    const userID = req.userID;
    CartIemModel.add(productID, userID, quantity);
    res.status(201).send('Cart is updated');
  }
}
```

Creating the Cart Items Routes

1. In the "cartItems" folder, create a routes file named "**cartItems.routes.js**" to define the routes related to cart items.
2. Import the necessary dependencies, such as Express and the `CartItemsController`.
3. Create an instance of the Express Router and assign it to a variable named `cartRouter`.
4. Instantiate the `CartItemsController`.
5. Set up the route for adding a new item to the cart using the POST method.
    - Specify the route path as "/add".
    - Call the `add` method of the `CartItemsController` for this route.
6. Export the `cartRouter` as the default export.

cartItems.routes.js file:

```
// Manage routes/paths to ProductController
```

```
// 1. Import express.
import express from 'express';
import { CartItemsController } from './cartItems.controller.js';

// 2. Initialize Express router.
const cartRouter = express.Router();

const cartController = new CartItemsController();

cartRouter.post('/', cartController.add);

export default cartRouter;
```

Setting Up Server Configuration

1. In the main server file, import the `cartRouter` from the cart items routes file.
2. Add a middleware for the cart-related APIs to use the `cartRouter`.
3. Ensure that the cart-related APIs are secured by adding a JWT authentication middleware.
4. Changes in **server.js** file:

```
import cartRouter from
'./src/features/cartItems/cartItems.routes.js';
server.use('/api/cartItems', jwtAuth, cartRouter);
```

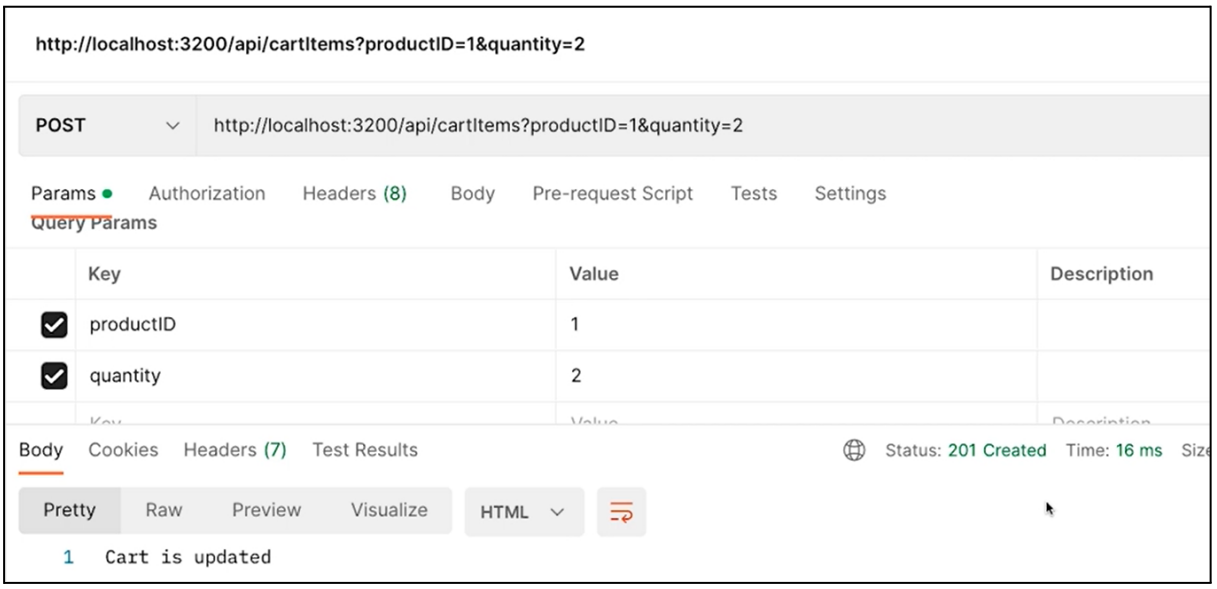
Testing Cart Feature

We will test this API and proceed to create another API to retrieve the items in the user's cart.

Testing the Add Item to Cart API

1. Start by running the server to test the API's functionality.
2. Open Postman and log in with the user credentials.
3. Create a new request for adding an item to the cart:
    - Set the request method to POST.
    - Add the authorization header with the token.
    - Set the request URL as "/API/cart-items" (or the appropriate endpoint).
    - Add the required query parameters: "productID" and "quantity".
    - Example: productID=1, quantity=2.

4. Send the request and verify that the response shows a "Cart is updated" message.



5. At this point, there is no direct way to confirm the update, so we will proceed to implement an API for retrieving the cart items.

Implementing the Get Cart Items API

1. Modify the model to include a function that returns all cart items for a specific user.
    - Create a function called `get` that takes the `userID` as a parameter.
    - Filter the cart items array based on the provided `userID`.
    - Return the filtered cart items.

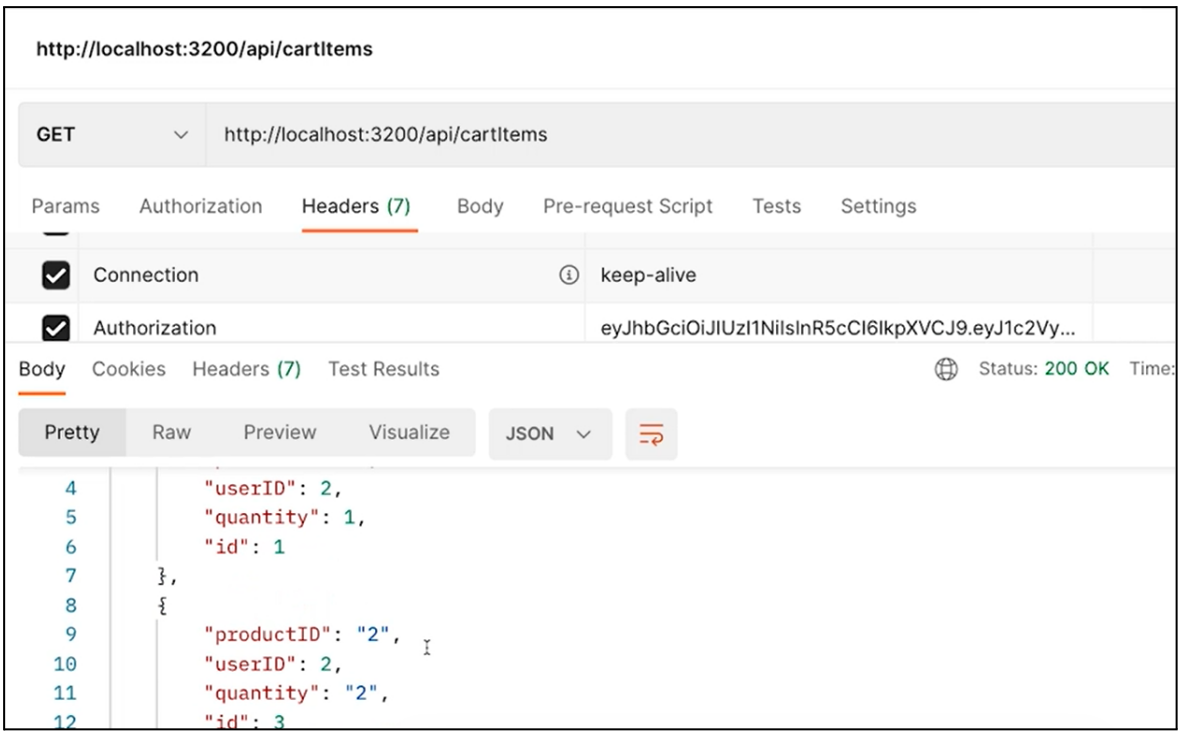
```
static get(userID) {
   return cartItems.filter(
     (i) => i.userID == userID
   );
 }
```

2. In the controller, create a method called `get` to handle retrieving cart items.
    - The method should take a `request` and `response` as parameters.
    - Retrieve the `userID` from the token stored in the request object.
    - Call the `get` function from the model, passing the `userID`.
    - Return a response with a status code of 200 and send the cart items as the response body.

```
get(req, res) {
    const userID = req.userID;
    const items = CartIemModel.get(userID);
    return res.status(200).send(items);
  }
```

Testing the Get Cart Items API

1. In Postman, create a new request to test the Get Cart Items API.
2. Set the request method to GET.
3. Add the authorization header with the token.
4. Set the request URL to the appropriate endpoint ("/api/cart-items" or similar).
5. Send the request and verify that the response shows the cart items specific to the logged-in user.
6. The response should only include the cart items belonging to the user with the associated user ID.



Deleting Cart Item

We will add another API related to the cart feature, allowing users to delete existing cart items.

Implementing the Delete Cart Item API

1. Create a `delete` function in the model that takes the `cartItemID` as a parameter.
   - Use the `find` method to search for the cart item with the matching ID.
   - If the cart item is not found, return an error message.
   - If the cart item is found, find its index using `findIndex`.
   - Use the `splice` function to remove the cart item from the `cartItems` array.

```
static delete(cartItemID, userID) {
    const cartItemIndex = cartItems.findIndex(
```

```
      (i) =>
        i.id == cartItemID && i.userID == userID
    );
    if (cartItemIndex == -1) {
      return 'Item not found';
    } else {
      cartItems.splice(cartItemIndex, 1);
    }
  }
```

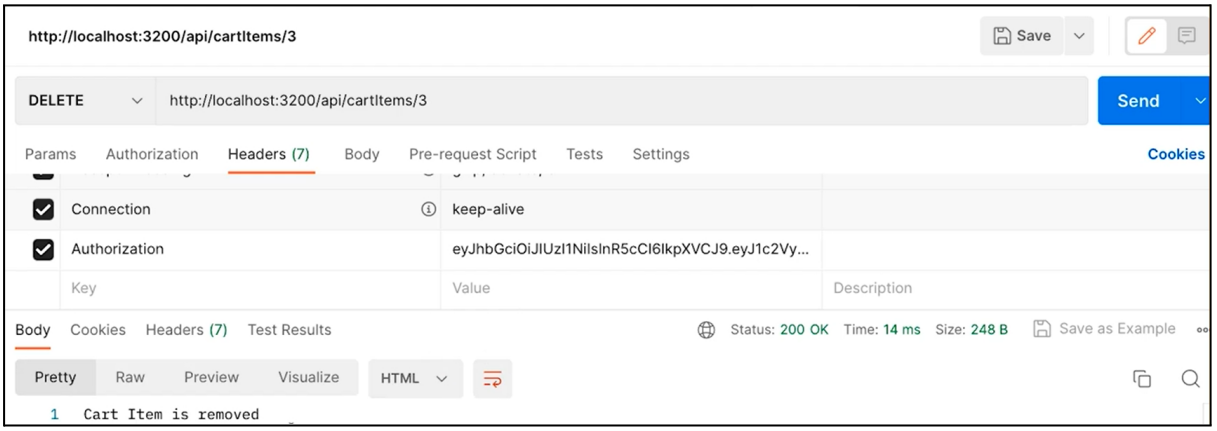
2. In the controller, create a `delete` method to handle deleting a cart item.
    - Accept the `request` and `response` parameters.
    - Retrieve the `userID` from the token stored in the request object.
    - Retrieve the `cartItemID` from the request parameters.
    - Call the `delete` function from the model, passing the `cartItemID`.
    - Implement a validation to ensure that the user is deleting their own cart items.
    - Return an error message if the validation fails.
    - Return a response with a status code of 404 (Resource Not Found) if the cart item is not found.
    - Return a response with a status code of 200 (Success) and a message stating that the cart item has been removed.

```
delete(req, res) {
    const userID = req.userID;
    const cartItemID = req.params.id;
    const error = CartIemModel.delete(
      cartItemID,
      userID
    );
    if (error) {
      return res.status(404).send(error);
    }
    return res
      .status(200)
      .send('Cart Item is removed');
  }
```

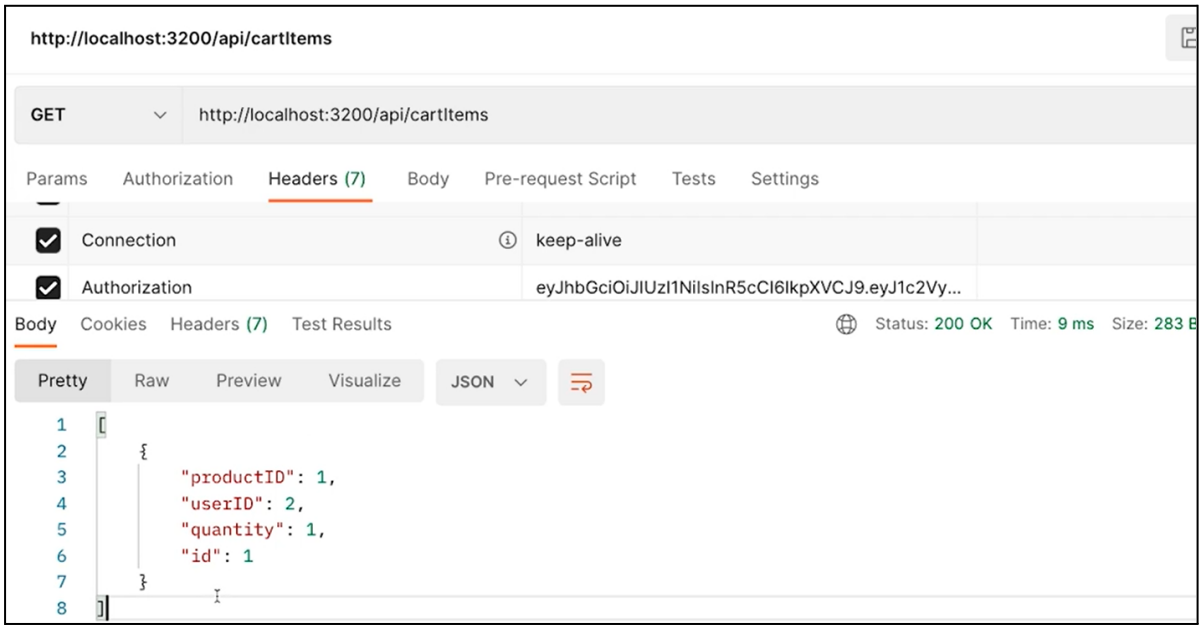
Testing the Delete Cart Item API

1. Run the server and ensure that the cart items are present in the application.
2. In Postman, create a new request for deleting a cart item.

3. Set the request method to DELETE.
4. Add the authorization header with the token.
5. Set the request URL to the appropriate endpoint ("/API/cart-items/:id" or similar).
6. Replace `:id` with the ID of the cart item you want to delete.
7. Send the request and verify that the response shows a "cart item is removed" message.



8. To verify the deletion, send a GET request to retrieve the cart items and confirm that the deleted item is no longer present.



API Documentation

We will discuss the importance of API documentation and how it helps clients understand and use our APIs effectively. We will explore the OpenAPI specification,

which provides a standardized means to define and document APIs. Additionally, we will introduce Swagger as a popular tool for implementing API documentation.

The Need for API Documentation

- Clients need clear instructions on how to use APIs.
- As developers, we may not know who our clients will be.
- API documentation makes it easier for clients to understand and use our APIs.

OpenAPI Specification

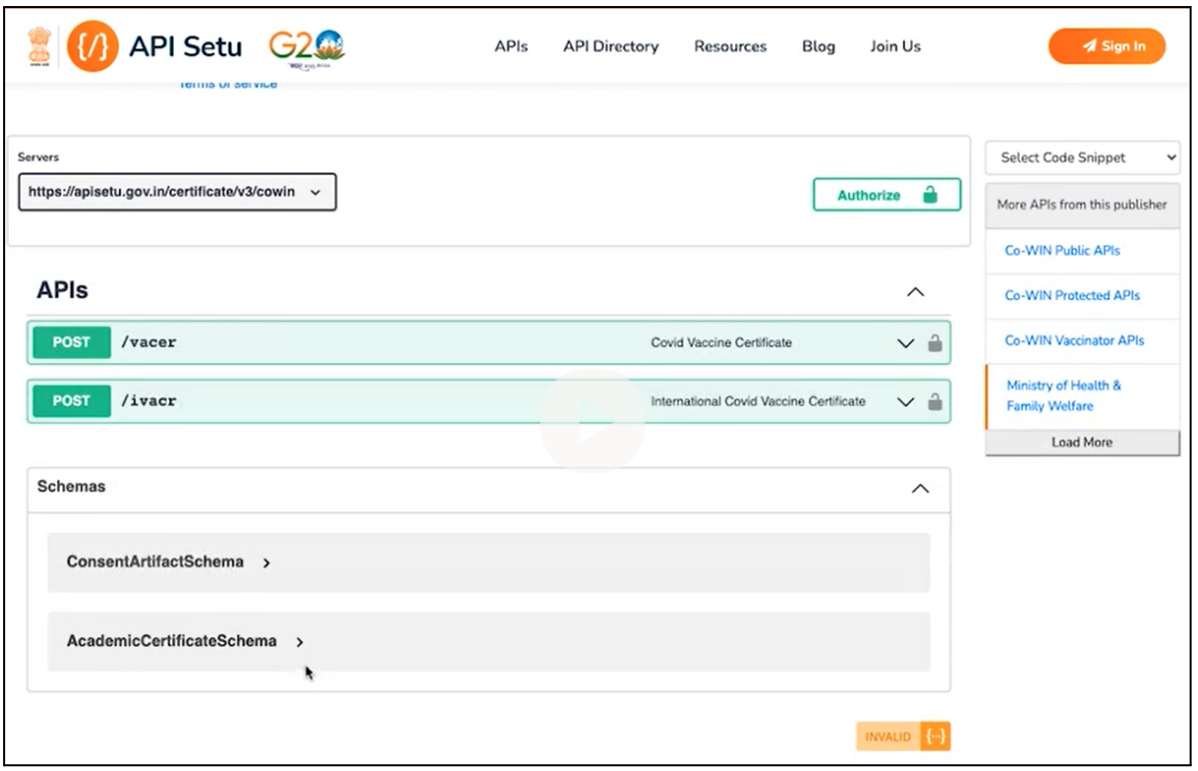
- OpenAPI is a widely adopted specification for documenting APIs.
- It provides a standardized way to define APIs and communicate their functionality to clients.
- OpenAPI allows clients to quickly understand how an API works, configure infrastructure, generate client code, and create test cases.

Swagger for API Documentation

- Swagger is an implementation of the OpenAPI specification.
- It helps developers create interactive API documentation for their APIs.
- Swagger provides a user-friendly interface that allows clients to explore and understand the API's endpoints, request methods, parameters, and responses.

Example: COVID Vaccination API Documentation

- Government of India provided an API for booking COVID vaccination appointments.
- Applications like Paytm and Arogya Setu implemented this API to enable appointment bookings.
- The API documentation followed the Swagger format.
- Clients could understand the API endpoints, request methods, parameters, request/response formats, and error codes through the Swagger documentation.



Using Swagger

Implementing Swagger in our Node.js API application. Swagger is a tool that helps us generate API documentation and provides a user-friendly interface for clients to understand and interact with our APIs.

Installing Swagger UI Express

- We will use the `swagger-ui-express` package to implement Swagger in our application. (Link)
- Install the package using the command: `npm install swagger-ui-express`

Creating a Swagger JSON File

- Create a JSON file called `swagger.json` to define the API documentation.
- Start by specifying the Swagger version as 2.0.
- Provide basic information about your API, including the version, description, and title.
- Define the `host` where your API is hosted (e.g., `localhost:3200`).
- Specify the paths for your API endpoints (e.g., `/api/products`, `/api/users/signin`).
- For each path, define the request methods (e.g., GET, POST) and include a summary and description.

- Define the parameters for each request, such as query parameters or request body.
- Specify the expected responses for each request, including status codes and descriptions.
- ***swagger.json file:***

```
{  "swagger": "2.0",
  "info": {
    "version": "1.0.0",
    "description": "API for E-commerce application",
    "title": "E-commerce API"
  },
  "host": "localhost:3200",
  "securityDefinitions": {
    "JWT": {
      "in": "header",
      "name": "Authorization",
      "type": "apiKey"
    }
  },
  "paths": {
    "/api/products": {
      "get": {
        "tags": ["Products"],
        "summary": "Get Products",
        "description": "User will get all products",
        "security": [{ "JWT": {} }],
        "responses": {
          "200": {
            "description": "OK"
          },
          "401": {
            "description": "Unauthorized"
          }
        }
      }
    },
    "/api/users/signin": {
      "post": {
        "tags": ["Users"],
        "summary": "Login",
```

```
        "description": "User login to get token",
        "parameters": [
          {
            "in": "body",
            "name": "body",
            "description": "User Credentials",
            "schema": {
              "type": "object",
              "properties": {
                "email": {
                  "type": "string"
                },
                "password": {
                  "type": "string"
                }
              }
            }
          }
        ],
        "responses": {
          "200": {
            "description": "OK"
          },
          "400": {
            "description": "Incorrect Credentials"
          }
        }
      }
    }
  }
}
```

Configuring Swagger in the Express Server

- Import the `swagger-ui-express` and `swagger.json` files into your Express server.
- Create a route (e.g., `/api/docs`) to expose the Swagger UI.
- Use the `swagger.serve` middleware to create the Swagger UI.
- Configure the Swagger UI using `swagger.setup` and pass the imported `swagger.json` object.

```
import apiDocs from './swagger.json' assert {type:'json'};
server.use(
  '/api-docs',
  swagger.serve,
  swagger.setup(apiDocs)
);
```

Testing the Swagger UI

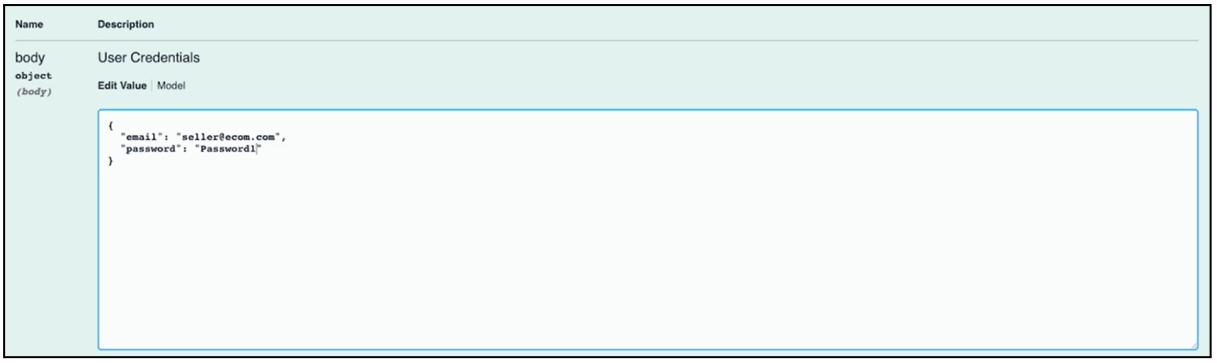
- Start the server and navigate to the Swagger UI route (e.g., `http://localhost:3200/api/docs`).
- Verify that the Swagger UI displays your API documentation.
- Clients can now use the Swagger UI to explore and test your API directly from the browser.

Testing Swagger

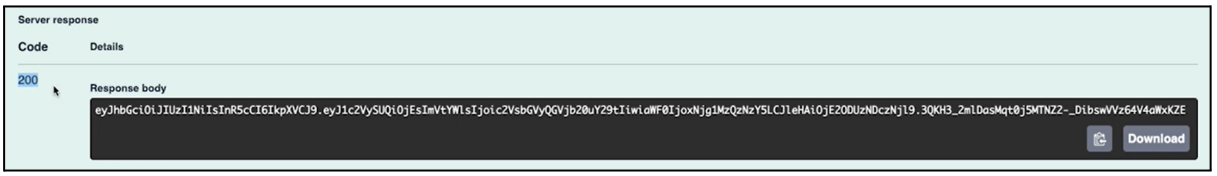
We have implemented Swagger in our Node.js Express application and created documentation for the sign-in API. Now, we will explore how to test our API using the Swagger UI.

Testing the Sign-In API

- In the Swagger UI, locate the sign-in API documentation.
- Click on "Try it out" and enter the predefined credentials.



- Execute the request and verify the response.
- The response should include a token that can be used for subsequent API calls.



Handling 404

Here we will focus on handling an important response, the 404 error, when a user requests an API that does not exist in our application.

Understanding the Issue

- When a user sends a request to a non-existent API, the default response provided by the server and client (browser) may not be informative or helpful.
- The default response displays "Cannot GET /api/users/get" which is not user-friendly.

Providing a Custom Response

- To address this issue, we need to send a separate message that indicates the requested resource does not exist or is not found.
- We can configure a middleware at the end of all our routes to handle 404 requests.
- This middleware will handle requests that do not match any existing paths in our application.

Implementing the 404 Middleware

- Add the 404 middleware after all existing routes.

- Use `server.use` to configure the middleware to handle all request methods (`use` is used instead of `get`, `post`, etc.).
- In the middleware, end the response with the appropriate error message and set the status code to 404 (Not Found).

```
// 4. Middleware to handle 404 requests.
server.use((req, res)=>{
  res.status(404).send("API not found. Please check our
documentation for more information at localhost:3200/api-docs")
})
```

Customizing the Error Message

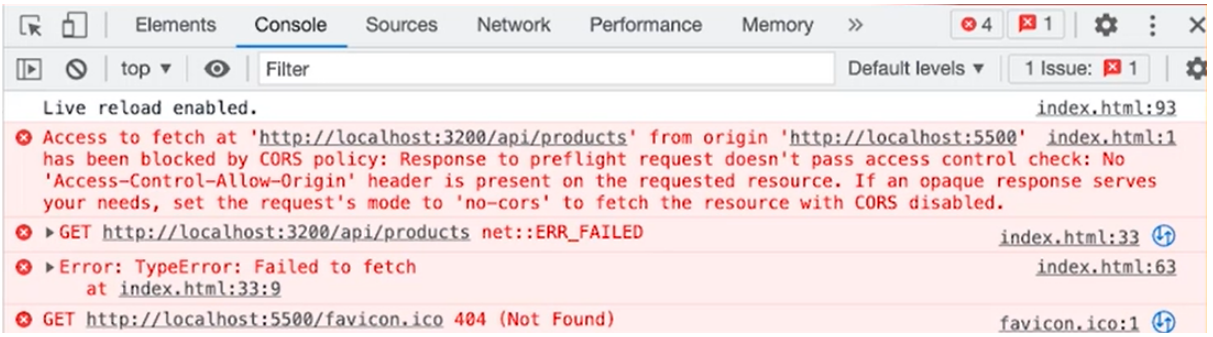
- Instead of the default "cannot get" message, send a more descriptive error message, such as "API not found."
- Optionally, provide additional information or guidance to the user, such as pointing them to the API documentation for more information.
- You can specify a link to the documentation path, where users can find all available APIs.

CORS

We will focus on an important feature for improving client experience while consuming our APIs: Cross-Origin Resource Sharing (CORS).

Understanding the Issue

- When clients, particularly web-based UI clients, try to consume our APIs from a different origin (e.g., different port or domain), they may encounter a CORS error.
- CORS stands for Cross-Origin Resource Sharing and is a security feature implemented by browsers and servers to prevent cross-origin requests by default.
- Cross-origin requests occur when the server and client are on different origins (e.g., different ports or domains).



CORS Policy and Restrictions

- The default CORS policy restricts cross-origin requests to ensure API access is limited to specific clients for security reasons.
- APIs may need to be accessed only by specific authorized clients, such as a bank's web UI or mobile application, rather than allowing anonymous access.

Configuring CORS Policy

- To allow cross-origin requests from specific clients while maintaining security, we need to configure the CORS policy on our server.
- In our case, we want to allow requests from the client running on `5500` port (origin) to access our server running on `3200` port (origin).

CORS using headers

We will configure the CORS policy in our server application to allow access to our client application.

Configuring CORS Policy

- CORS policy configuration is done using middlewares.
- We will create a middleware to handle CORS policy configuration using `server.use`.
- The middleware will take the request, response, and next object as parameters.
- To allow access to our client application, we need to set the `access-control-allow-origin` header in the response.
- The value of the header should be the URL of our client application (e.g., `http://localhost:5500`).
- To allow access from all web browser clients, we can set the value to `*`.

Handling Pre-flight Requests

- Pre-flight requests are verification requests sent by the browser before making the actual request.
- The browser checks if the server responds with an HTTP OK status (200) to the pre-flight request.
- To handle pre-flight requests, we can check if `request.method` is `OPTIONS`.
- If it is an OPTIONS request, we need to send a response with an OK status using `response.sendStatus(200)`.

Allowing Request Headers

- If the client application includes specific headers in the request, such as the Authorization header, we need to allow those headers in the CORS policy.
- We can specify the allowed headers using the `access-control-allow-headers` header in the response.
- To allow all headers, we can set the value to `*`. Alternatively, we can specify specific headers like `Content-Type` or `Authorization`.

```
// CORS policy configuration
server.use((req, res, next)=>{
  res.header('Access-Control-Allow-Origin',
'http://localhost:5500');
  res.header('Access-Control-Allow-Headers', '*');
  res.header('Access-Control-Allow-Methods', '*');
  // return ok for preflight request.
  if(req.method=="OPTIONS"){
    return res.sendStatus(200);
  }
  next();
})
```

CORS using library

Previously we configured the CORS policy in our server application manually using response headers. However, there is a popular library available in Node.js that simplifies the CORS configuration process. In this video, we will explore how to use the CORS library in our application.

Using the CORS Library

- The CORS library is a widely-used Node.js middleware for handling CORS policy.
- We can install the library using npm by running `npm install cors`.
- To use the library, we need to import it in our server application using `import cors from 'cors'`.
- Instead of manually configuring the CORS policy, we can use the `cors()` function provided by the library as middleware.
- This simplifies the configuration process and takes care of handling the necessary response headers.

Default Configuration

- By default, the CORS library allows access from all origins and includes all headers.
- If we don't specify any options, the library assumes an open CORS policy.

Configuring Specific Options

- If we want to configure specific options for the CORS policy, we can pass an options object to the `cors()` function.
- The options object can include properties like `origin` and `allowedHeaders`.
- We can specify the allowed origins by setting the `origin` property to the desired URLs.
- To allow specific headers, we can set the `allowedHeaders` property to an array of header names or use the `*` wildcart for all headers.

Summarising it

Let’s summarise what we have learned in this module:

- Learned about the process of rating products, which involves allowing users to assign ratings.
- Implement and tested a shopping cart feature enabling users to add items they wish to purchase.
- Implemented delete cart item feature that allows users to remove items from their shopping cart, improving the overall user experience.

- Learned how to create clear and comprehensive documentation for an API, which helps other developers understand how to use it effectively.
- Learned how to utilize Swagger to create interactive and machine-readable API documentation.
- Followed the process of testing the Swagger documentation and ensuring it accurately reflects the API's functionality.
- Learned how to handle HTTP 404 errors, which occur when a requested resource is not found on the server.
- Learned how to set specific HTTP headers to control CORS behavior and permissions.
- Learned how to use a library or framework to simplify the implementation of CORS in your web application.

Some Additional Resources:

- **Documenting your Express API with Swagger**
- **Cross-Origin Resource Sharing (CORS)**
- **A Guide to CORS in Node.js with Express**