

NTNU

Exam Project

IDATG2204

MAY 2024

Member names, student ID & candidate ID

Knut Fineid	(588802)	(10030)
Torbjørn Halland	(588797)	(10147)
Steffen Martinsen	(758526)	(10209)
Gisli Nielsen	(588793)	(10112)
Ådne Olsen Mauno	(588813)	(10002)

Table of Contents

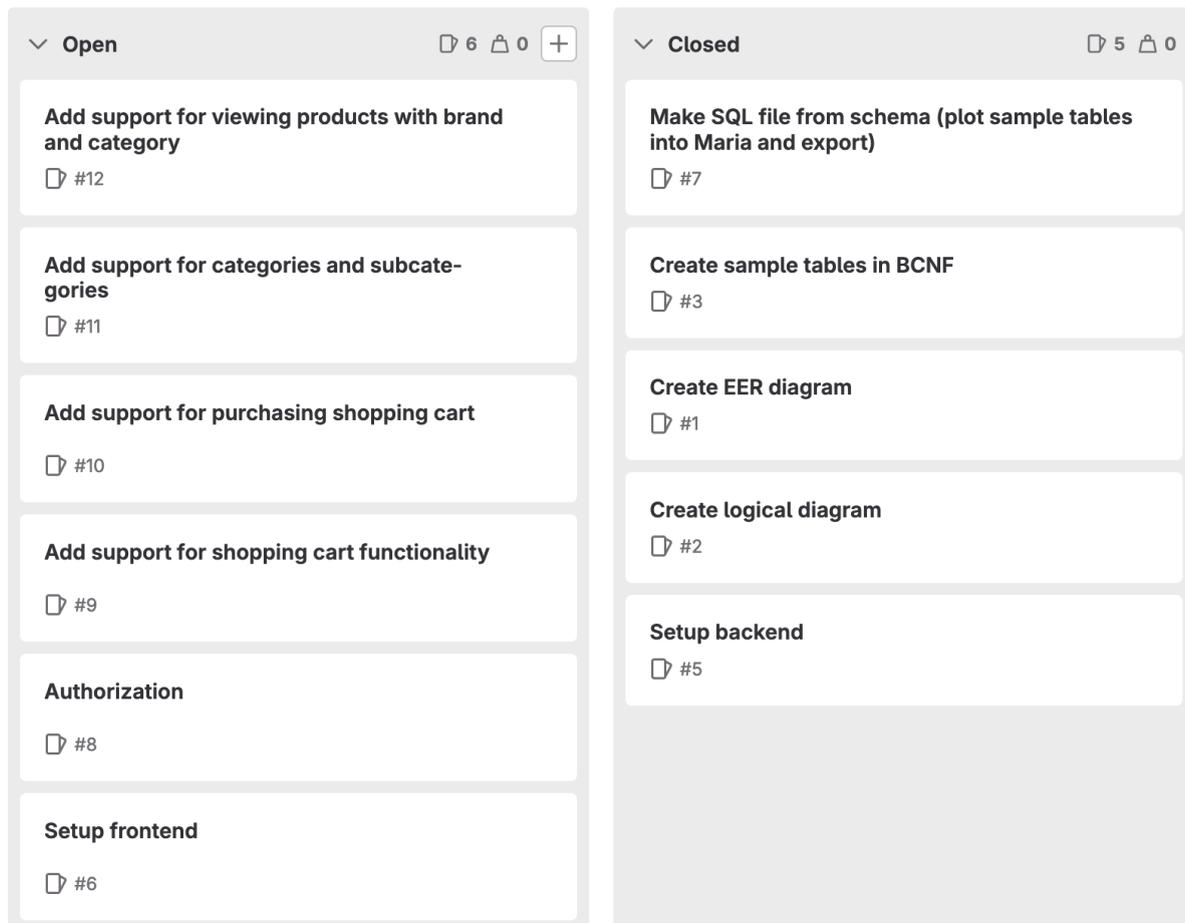
Member names, student ID & candidate ID	1
Collaboration	3
Sharing of workload	3
README	4
Modelling	4
EER Model	4
Logical Model	4
Normalization	4
Frontend	5
Database	6
Backend	7
Authentication	7
Categories and products	7
Order	8
Screenshots of the running API	8
Register user	9
Log in	9
Get all categories	10
Get one category	11
Get all products	12
Get one product	12
Product search	13
Place order	13
Get all orders from the logged in user	14
Pay order	14
Log out	15

Collaboration

We utilized a Gitlab repository for our collaborative efforts in the project. We met up physically approx. once a week, with other virtual meetings added in. We facilitated solo work on tasks by keeping the Issue Board on Gitlab updated.

Gitlab repository:

<https://git.gvk.idi.ntnu.no/steffemar/idadg2204-project>



An arbitrary screenshot of our issue board at the beginning of development.

Sharing of workload

We collaborated as a team on the core aspects of the assignment, including the EER and logical model, as well as the design and implementation of the database.

As for the rest, Torbjørn Halland and Ådne Olsen Mauno focused on the frontend, while Gisli Nielsen, Steffen Martinsen and Knut Fineid were in charge of the backend.

README

The README is included on the front page of the repository containing instructions on how to invoke the service.

The README can be found [here](#).

Modelling

EER Model

The EER model is the first task we completed. We completed this together to ensure that every group member had the same understanding of the implementation. We created the EER model before we normalized the tables.

It shows the different main entities the database is based on. It also shows the relationships between them, as well as the cardinality.

As a note the category entity is recursive because a category can hold a sub_category.

Logical Model

The logical model shows the relationship between the different tables in the database. The model also clearly shows the different primary keys and foreign keys used in the database. Most have a typical setup with a single primary key, and foreign keys to related tables. However, there are some special cases we would like to elaborate on.

The entity **order_item** is a weak entity type which is uniquely identified through the primary keys of the **order** and **product** entity. In the diagram the partial primary keys for the **order_item** entity are included in the entity as primary keys.

This is also true for **payment** which is uniquely identified through **order's** *order_id*, and this attribute is included as **payment's** primary key.

Lastly, we also want to clarify that "order" is a reserved word in SQL, so we decided to change the table name from order to user_order.

Normalization

In the normalization process, we removed the *user_id* attribute from the **user** relation. We assume that the *email* is enough for the user to log in and create an account. We split the **user** relation into **user** and **user_details**, to split the transitive dependencies.

The normalized model is included in the repository, and is created with the same tool as the logical model. Based on the task description it was a little unclear if we should make individual models that complied to either 1NF, 2NF, 3NF or BCNF respectively, or make a single model that complied to all of them. We assumed the latter, and thus the model is fully normalized.

The table is in 1NF because no single cell has more than one value. The table is also in 2NF because every non-prime attribute is fully functionally dependent on the primary key, meaning that all attributes can be uniquely identified through the primary key. Furthermore, it's in 3NF because there are no transitive dependencies relations, meaning every attribute is dependent on the primary key and nothing else. Lastly, it's also in BCNF because every candidate key can uniquely identify every non-prime attribute.

Frontend

While we initially started on the frontend and made a homepage for the ElectroMart website we got notified that the frontend part got ruled out of the evaluation of the database project. We decided as a team to include the work we have made for the frontend part, however it is not a part of the implementation of our solution. Instead, we put greater focus on the backend and database aspects. Hence, this section only documents the work we have done on the frontend part prior to our decision not to include it.

For the start of the frontend implementation we started out by having a team discussion on how the layout would look like. After getting a rough idea of how the frontend part would visually look like and how both the front- and backend part should be integrated with each other, we made an initial sketch of the homepage, checkout and profile page. These sketches can be found inside the "Assets" directory in our repository, under the name "WebApp-Sketch.pdf".

Only the homepage was partially implemented out of the three pages we planned to create. The checkout and the profile page has no implementation other than the sketches we have made for them. For the actual code we decided to use React as a framework to create the web pages. The inspiration to use React came from one of our team members who have worked with it before. After hearing how easy it made integrating HTML, CSS and JavaScript, the ones on our team who had primary responsibility for the frontend started out by learning how React worked before actually starting to code.

For the functionality of our frontend web application, we wanted to have these key features.

Homepage

- Navigation tab where the user has easy access to all products and categories.
- See a set of products under a defined category in. (E.g., see all phones under smartphones category)
- Get quick access to checkout tab and user profile page.
- See number of items in the checkout tab.
- Add products to cart.

Checkout Page

- See all products in cart.
- See quantity of a product in cart.
- See the cost of a product
- See total cost of the products in cart.

Profile Page

- Navigation field for account information, wishlist, orders and account settings.
- Quick display of basic account details and registered address for user.

The code we have written for all our frontend work can be found under the “Frontend” directory in our repository. Most of the code we have written is under “Frontend/src/” directory. All of the code has been modularized and split up into their own files with name that fits the code’s functionality/use case. CSS is under the “Frontend/css/” directory and images used for homepage are under “Frontend/Assets/”.

Database

We used MariaDB to implement our database. The SQL file containing the database can be located under the “database” directory in the project repository. This includes pre-populated data for all relations in the database.

The database is based on the normalized logical diagram, which can be found in the “Assets” directory of the repository. All objects in this diagram is implemented as its own relation, and all attributes are included as well. The primary and foreign keys are also applied.

The logical model shows the connections between the different relations in our database. Additionally we have an EER model, named "EMART-EER-Model", which is what the logical model is based on. This model can be located in the same directory as the logical model, and act as a supplemental guide to the underlying logic the database foundation is based on.

The naming of the relations is corresponding to the objects in the logical model, with the exception of order, which is a reserved word in SQL, and has thus been renamed to user_order. We realized that ‘user’ is a reserved word in SQL as well, and the user relation should have been changed.

Security measures

- Least privileged concept, only give the database users the lowest possible permissions.
- Hashing, salt and pepper. When storing the password, we make sure to use hash and salt the password. To make it even more secure, we could have added pepper to the password as well.
- Only returning the needed data. We make sure to only return the relevant data to the end user. This gives any potential threats less knowledge about the database.
- Confidentiality. Using authentication, we can make sure that no other users can access each other's data.

We have also reflected around the status attribute, which is relevant for both the user_order relation as well as the payment relation. In the current implementation this is set to be of type string. However we realize that it would perhaps be better suited to use an enum type here, as this would further ensure that the status is at all times set to one of a few selected values.

Backend

Authentication

To ensure that we are able to replicate the stored password in their hashed form, we saw the need to store the generated salt in a persistent way. We used the **bcrypt** library in Python, generated salt with a built-in function and stored the salt in the database, together with the user. This way, we are able to extract the salt for user authentication when the user logs back in.

In our implementation the salt and the hashed password is stored together in the database. To further secure the storage of the password and general security of user information. We could introduce “pepper” and multiple hashing iterations. For ease of use and simplicity we decided to not do this, because the implementation of the backend is not the main part of the assignment.

We store a cookie for authentication, which lives for 24 hours. Once a user is logged in, the cookie ensures authentication for the duration of the cookie. The cookie holds the value of the user id and we use this to be able to distinguish which user is logged in.

When a user is created the password is hashed with salt and both are stored in the user table along with the user id and email. In the user details table we store all of the other information like personal details.

When logging in we query the user table on the supplied email address and check whether the email and password match the ones stored in the database.

When logging out we delete the cookie that holds the user id of the logged in user.

Categories and products

The category and the product endpoints have two common features, one where we can view all of the entities and one where we can view one specific entity.

In the product endpoint we can view all information available for all of the products that exist in the database by querying the entire table. We can also view one product by setting the product id in the path and querying with a “WHERE id = id” to get the product. This will act as a product page for the item in the front-end part of the store. In addition to these two we have implemented a search feature. This is to showcase that the use of a “LIKE” statement in the query can be used as a search function for products.

In the category endpoint we have the same functionality, though a little different. The main endpoint displays all the different categories in the database by querying the category table in the database and joining this with the subcategory table to see all the subcategories of a category. To further see the products of a category one has to have a category in the path, this will then display all the products belonging to that category and its subcategories. To do this we query the products table on the category to get the correct products then create an array with all the products in the category object.

Order

In our implementation of the order and payment system we make the assumption that in the complete application the front end will handle the “cart” for the customers, so the backend handles a POST request that contains all the products a user has in the cart when an order is made.

The order will then be created with the status of “Pending” until the payment endpoint is called with that order id, it will then be paid in full, and the status will be changed to “Paid”.

When an order is created, all its info along with all the items for that order are inserted into the “user_order” and “order_items” tables. We use a mutex lock to avoid a race condition when we get and set the stock quantity in the order. This is so we are certain that two users can’t empty the stock of one product at the same time, resulting in one of the users buying a product that is not in stock.

In the payment endpoint, we do not actually handle the payment. There is a dummy function that returns “True” every time to indicate that a successful payment has been made. We query to make sure the user logged in is the user with that order, and then update the status of the order to indicate that the order has been paid by the user.

If we were to make any further order processing it would be to change the status to shipped and delivered. This would not show any different querying other than what was done in the payment part, so it is not implemented.

Screenshots of the running API

To run the application, an empty database has to be created in MariaDB called “ElectroMart” and import the sql file to get the tables and some basic information. Then, have it running while running the python application.

To run the python file one needs to run the file located at “/Backend/main.py” locally and request on port “8080”.

All screenshots are from the postman application where one can see the method, path, body of the request and body of the response.

When we have a GET request the request body is empty and when registering a user there is only a “201” status code with no content in the response body.

When logging out the request body is left empty.

Register user

POST http://localhost:8080/register/

Params Authorization Headers (8) **Body** Scripts Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL **JSON** ▾

```
1 {
2   ... "email": "user@user.com",
3   ... "password": "password",
4   ... "firstname": "John",
5   ... "lastname": "Doe",
6   ... "address": "Something 783, 4587 Somethingsville"
7 }
```

Body Cookies Headers (5) Test Results 🌐 Status: 201 CREATED

user_id	1	hash	salt	email
5	\$2b\$12\$REOfSHwzYEJJ/7GBvBXJJO3EvFoMwcNSUXMD0dyETdr...	\$2b\$12\$REOfSHwzYEJJ/7GBvBXJJO	user@user.com	

email	first_name	last_name	address
user@user.com	John	Doe	Something 783, 4587 Somethingsville

Log in

POST http://localhost:8080/login/

Params Authorization Headers (9) **Body** Scripts Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL **JSON** ▾

```
1 {
2   ... "email": "user@user.com",
3   ... "password": "password"
4 }
```

Body Cookies (1) Headers (6) Test Results 🌐 Status: 200 OK

Pretty Raw Preview Visualize **JSON** ▾ ⇅

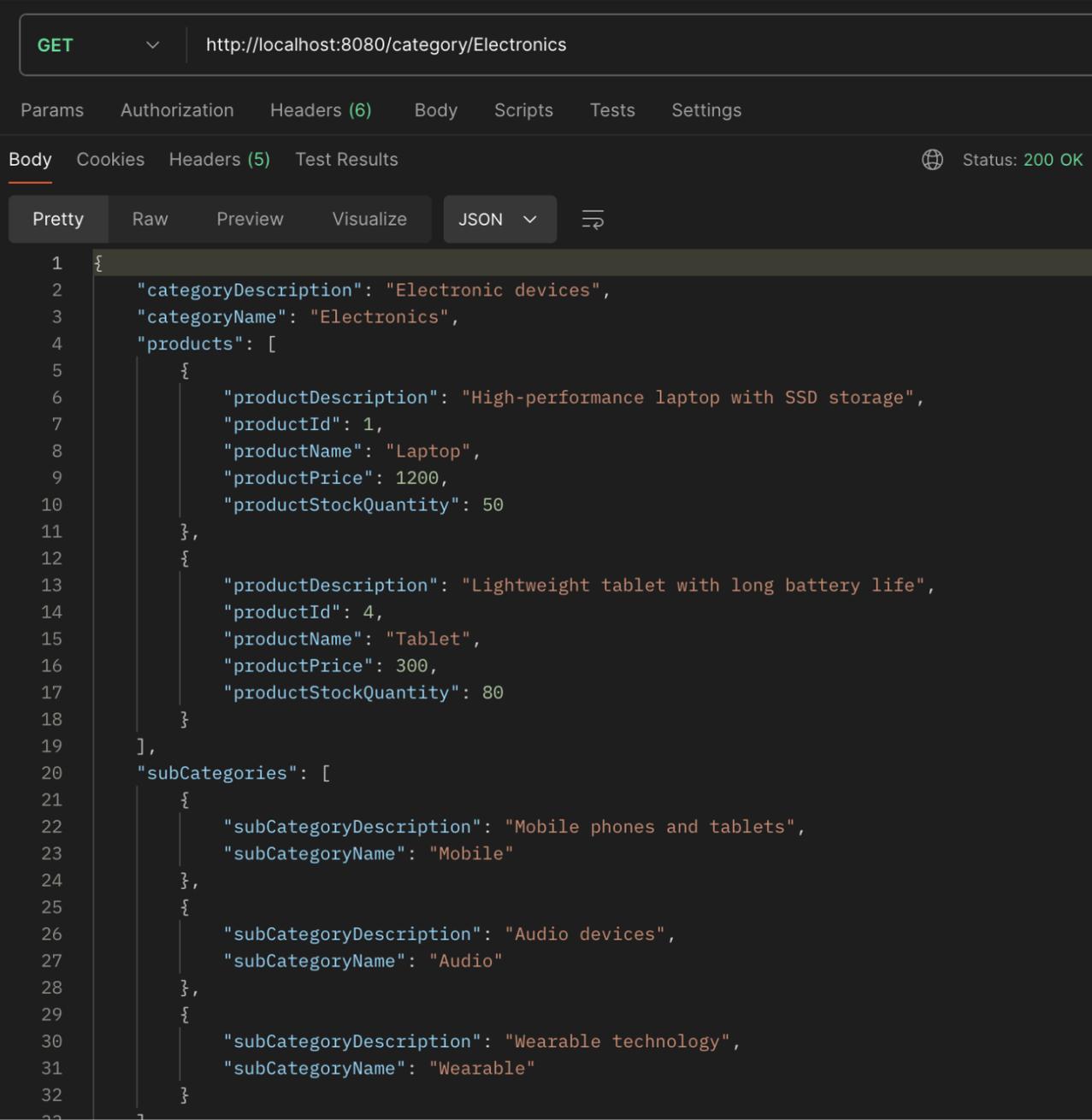
```
1 {
2   "message": "Login successful"
3 }
```

Get all categories

The screenshot shows a REST client interface with a GET request to `http://localhost:8080/category/`. The response is displayed in JSON format, showing a list of categories and their sub-categories.

```
1  [
2    {
3      "categoryDescription": "Electronic devices",
4      "categoryName": "Electronics",
5      "subCategories": [
6        {
7          "subCategoryDescription": "Mobile phones and tablets",
8          "subCategoryName": "Mobile"
9        },
10       {
11         "subCategoryDescription": "Audio devices",
12         "subCategoryName": "Audio"
13       },
14       {
15         "subCategoryDescription": "Wearable technology",
16         "subCategoryName": "Wearable"
17       }
18     ]
19   },
20   {
21     "categoryDescription": "Mobile phones and tablets",
22     "categoryName": "Mobile",
23     "subCategories": []
24   },
25   {
26     "categoryDescription": "Audio devices",
27     "categoryName": "Audio",
28     "subCategories": []
29   },
30   {
31     "categoryDescription": "Wearable technology",
32     "categoryName": "Wearable",
33     "subCategories": []
```

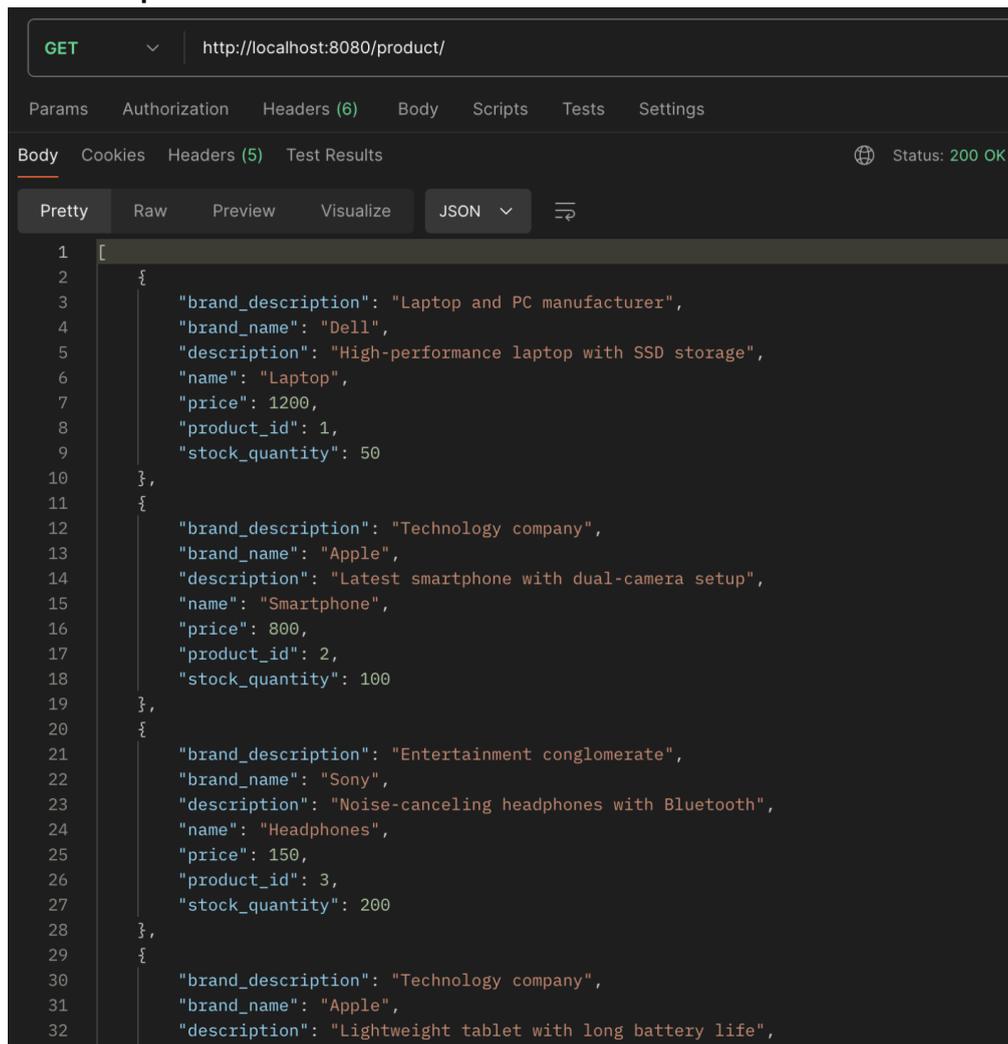
Get one category



The screenshot shows a REST client interface with a GET request to `http://localhost:8080/category/Electronics`. The response is a JSON object with the following structure:

```
1 {
2   "categoryDescription": "Electronic devices",
3   "categoryName": "Electronics",
4   "products": [
5     {
6       "productDescription": "High-performance laptop with SSD storage",
7       "productId": 1,
8       "productName": "Laptop",
9       "productPrice": 1200,
10      "productStockQuantity": 50
11    },
12    {
13      "productDescription": "Lightweight tablet with long battery life",
14      "productId": 4,
15      "productName": "Tablet",
16      "productPrice": 300,
17      "productStockQuantity": 80
18    }
19  ],
20  "subCategories": [
21    {
22      "subCategoryDescription": "Mobile phones and tablets",
23      "subCategoryName": "Mobile"
24    },
25    {
26      "subCategoryDescription": "Audio devices",
27      "subCategoryName": "Audio"
28    },
29    {
30      "subCategoryDescription": "Wearable technology",
31      "subCategoryName": "Wearable"
32    }
33  ]
34 }
```

Get all products



```
GET http://localhost:8080/product/

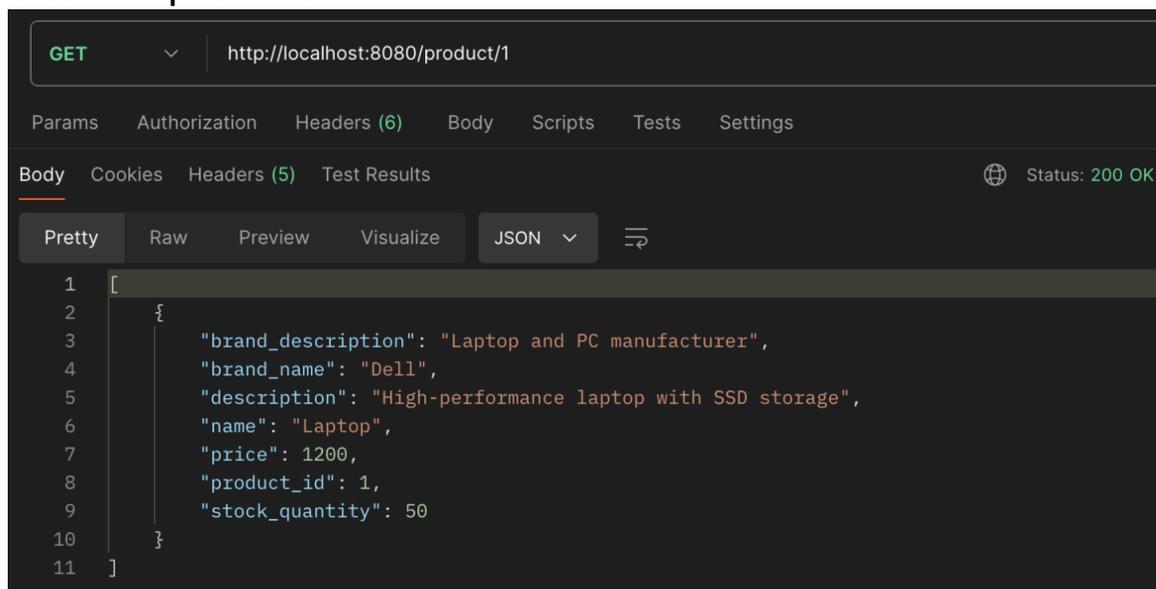
Params Authorization Headers (6) Body Scripts Tests Settings

Body Cookies Headers (5) Test Results Status: 200 OK

Pretty Raw Preview Visualize JSON ↕

1 [
2   {
3     "brand_description": "Laptop and PC manufacturer",
4     "brand_name": "Dell",
5     "description": "High-performance laptop with SSD storage",
6     "name": "Laptop",
7     "price": 1200,
8     "product_id": 1,
9     "stock_quantity": 50
10  },
11  {
12    "brand_description": "Technology company",
13    "brand_name": "Apple",
14    "description": "Latest smartphone with dual-camera setup",
15    "name": "Smartphone",
16    "price": 800,
17    "product_id": 2,
18    "stock_quantity": 100
19  },
20  {
21    "brand_description": "Entertainment conglomerate",
22    "brand_name": "Sony",
23    "description": "Noise-canceling headphones with Bluetooth",
24    "name": "Headphones",
25    "price": 150,
26    "product_id": 3,
27    "stock_quantity": 200
28  },
29  {
30    "brand_description": "Technology company",
31    "brand_name": "Apple",
32    "description": "Lightweight tablet with long battery life",
```

Get one product



```
GET http://localhost:8080/product/1

Params Authorization Headers (6) Body Scripts Tests Settings

Body Cookies Headers (5) Test Results Status: 200 OK

Pretty Raw Preview Visualize JSON ↕

1 [
2   {
3     "brand_description": "Laptop and PC manufacturer",
4     "brand_name": "Dell",
5     "description": "High-performance laptop with SSD storage",
6     "name": "Laptop",
7     "price": 1200,
8     "product_id": 1,
9     "stock_quantity": 50
10  }
11 ]
```

Product search

GET `http://localhost:8080/product/search/smart`

Params Authorization Headers (6) Body Scripts Tests Settings

Body Cookies Headers (5) Test Results Status: 200 OK

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "brand_description": "Technology company",
4     "brand_name": "Apple",
5     "description": "Latest smartphone with dual-camera setup",
6     "name": "Smartphone",
7     "price": 800,
8     "product_id": 2,
9     "stock_quantity": 100
10  },
11  {
12    "brand_description": "Electronics conglomerate",
13    "brand_name": "Samsung",
14    "description": "Fitness tracker with heart rate monitoring",
15    "name": "Smartwatch",
16    "price": 200,
17    "product_id": 5,
18    "stock_quantity": 150
19  }
20 ]
```

Place order

POST `http://localhost:8080/order/`

Params Authorization Headers (9) Body Scripts Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "products": [
3     {
4       "product_id": 1,
5       "quantity": 2
6     },
7     {
8       "product_id": 3,
9       "quantity": 1
10    }
11  ]
12 }
```

Body Cookies (1) Headers (5) Test Results Status: 201 CREATED

Pretty Raw Preview Visualize JSON

```
1 {
2   "message": "Order placed"
3 }
```

order_id	order_date	total_amount	status	user_id
6	2024-05-08 10:52:04	2550	Pending	5

Get all orders from the logged in user

The screenshot shows a REST client interface with a GET request to `localhost:8080/order/`. The response is a JSON array of three order objects:

```
1 [
2   {
3     "order_date": "2024-05-07T15:55:49",
4     "order_id": 7,
5     "status": "Paid",
6     "total_amount": 2150
7   },
8   {
9     "order_date": "2024-05-07T15:57:05",
10    "order_id": 8,
11    "status": "Paid",
12    "total_amount": 2150
13  },
14  {
15    "order_date": "2024-05-07T15:58:45",
16    "order_id": 9,
17    "status": "Pending",
18    "total_amount": 2150
19  },
20 ]
```

Pay order

The screenshot shows a REST client interface with a POST request to `http://localhost:8080/order/payment/`. The request body is a JSON object:

```
1 {
2   "order_id": 6,
3   "payment_method": "Vipps"
4 }
```

The response is a JSON object:

```
1 {
2   "message": "Order paid"
3 }
```

order_id	order_date	total_amount	status	user_id
6	2024-05-08 10:52:04	2550	Paid	5

Log out

The screenshot shows a REST client interface for a POST request to `http://localhost:8080/logout/`. The status is `200 OK`. The response body is displayed in JSON format:

```
1 {  
2   "message": "Logout successful"  
3 }
```