Stock-control and sales system API documentation

API endpoints for a Back-end system of a Stock-control and sales system.

The global variable {{host}} is set to be http://localhost/3000/

Utility

Endpoints for the primary population of the database and for search functionalities that should be implemented in a search bar.

POST /setup

{{host}}/setup

Populates the initial data if items from noroff API aren't stored in the db.

POST /search

{{host}}search

Endpoint available to any type of user, allows for search of items or categories, based on criteria such as partial Items names, specific SKU, specific Category or a combination of partial items names and category specific name.

Signup / Login

Endpoints for signup and login of users.

Duplicate usernames are not allowed, a maximum of 4 duplicate emails are allowed (duplicate emails give relevant discounts upon checkout).
Credentilas are stored in the databse (passwords are hashed+salted and set as BLOB on the db for extra security)

POST /signup

{{host}}signup

Body raw (json)

```
json

{
    "FirstName": "Ciccio",
    "LastName": "Pasticcio",
    "Username": "chitemmuort",
    "Email": "test@test.com",
    "Password": "test123"
}
```

POST /login

{{host}}login

Logins will provide the user or the Admin with a token that expires after 2 h.

The token needs to be used in the headers to be able to access most parts of the endpoints.

Body raw (json)

```
json

{
    "Username": "test",
    "Password": "test123"
}
```

Categories

Endpoints that allows all type of users to see a list of available categories and allows only the Admin to add/update or delete a category.

GET /categories

EVeryone can access this endpoint and see a list of all categories.

POST /category

{{host}}category

A category can be added by the Admin only if its name doesn't exist already.

AUTHORIZATION Bearer Token

Token <token>

Body raw (json)

json

```
{
    "Name": "Basketball"
}
```

PUT /category/:id

{{host}}category/9

A category can be updated by the Admin to a new name only if such name isn't already in use.

AUTHORIZATION Bearer Token

Token <token>

Body raw (json)

json

```
{
    "Name": "Football",
}
```

DELETE /category/:id

{{host}}category/9

A category can be deleted by the Admin only if itsn't in use in any of the available items.

AUTHORIZATION Bearer Token

Token	<token>

Items

Endpoints that allows all type of users to see a list of available items and allows only the Admin to add/update or delete an item.

GET /items

As requested, guest users can only see in-stock items while registered users can see them all.

POST /item

{{host}}item

An item can be added by the Admin only if name and SKU does not exist, if SKU is in correct format, and if all attributes, apart from Image are in the request body.

AUTHORIZATION Bearer Token

Token	<token>

PUT /item/:id

{{host}}item/161

Admin can update whatever attribute of the item by adding it in the request body, could for example be only 1 or could be 3, and this endpoint will update the specific item that has been given as parameter :id with the specific attributes that were sent in the body.

Name can't be in use in other items.

SKU can't be in use in other items and has to be of valid format.

AUTHORIZATION Bearer Token

Token	<token>

DELETE /item/:id

{{host}}item/161

The Admin can Delete an item by giving its id as :id parameter, if the item exists, it will be deleted.

AUTHORIZATION Bearer Token

Token	<token>

Carts

Endpoints that allows users to see their specific cart and the cart items within it or completely empty their cart of all cart items.

Only the admin can see a list of each users cart and the cart items in them.

GET /cart

{{host}}cart

Registered users can see their own cart and the items in it.

AUTHORIZATION Bearer Token

Token	<token>

GET /allcarts

{{host}}allcarts

Admins only can access this endpoint and see all users carts and their cart items.

AUTHORIZATION Bearer Token

Token	<token>

DELETE /cart/:id

{{host}}cart/2

A cart can be emptied by its user, if a wrong cart id is given, an error message is sent.

AUTHORIZATION Bearer Token

Token	<token>

Cart Items

Endpoints that allow users to add items to their cart, update the quantity of a specific cart item, delete/remove a specific cart item from their cart.

POST /cart_item

{{host}}cart_item

An item can be added to a cart as cart item if it isn't already in the cart (in which case they should use the update endpoint) or if it is available amongst all items.

The cart item can be added by sending a request body with either id or Name of the specific item.

AUTHORIZATION Bearer Token

Token	<token>

Body raw (json)

```json
{
    "id": 129
}
```

}

PUT /cart_item/:id

{{host}}cart_item/130

An item id is sent as parameter :id and the endpoint allows a user to update a cart items' quantity, if that item is available in the cart and if the stock quantity of such item is enough to meet the desired quantity.

AUTHORIZATION Bearer Token

Token	<token>

Body raw (json)

json

```
{
    "Quantity": 5
}
```

DELETE /cart_item/:id

{{host}}cart_item/129

Allows a user to deleete a specific cart item from the cart, if the item id parameter sent matches with any item in the users cart.

AUTHORIZATION Bearer Token

Token	<token>

Orders

Users are allowed to see only their completed orders (as per requirement) and checkout one cartitem at the time from their carts. (customer has specified that to check them all out a loop can be made on the frontend).

Only admins can access the endpoint showing all orders (with all statuses) and the endpoint to update a specific orders' status.

GET /orders

{{host}}orders

Users will see a list of all their completed orders and admins will see a list of all users orders.

AUTHORIZATION Bearer Token

Token	<token>

GET /allorders

{{host}}allorders

Only Admins can access this endpoint and will be able to see a list of all users orders and all their related orderitems

AUTHORIZATION Bearer Token

Token	<token>

POST /order/:id

{{host}}order/145

With this requested endpoint, if an item id that is set as parameter :id matches a cartitem, an in-progress order is created (if it doesn't exist) and consequentially an orderitem too is created. The cart item is removed from the cart and (if there are no other cart items) a checkout message is sent with relevant order id, total price, discounts and final cost of the whole order.

Else, a message will tell the customer that the orderitem is added to the order.

Availability checks and adjustments of the stock quanitites are also done within this endpoint.

AUTHORIZATION Bearer Token

Token	<token>

PUT order/:id

{{host}}order/3

An admin can update an order status from this endpoint.
If complete, the order can ba viewed by the customer in the previous mentioned get orders endpoint.

If cancelled, the endpoint adjusts the quantity of stock that was removed during checkout procedures through POST/order/:id (or POST/cart/checkout)

AUTHORIZATION Bearer Token

Token <token>

Body raw (json)

json

```
{
    "Status": "complete"
}
```

A better checkout

An endpoint that automatically creates an order and orderitems out of all the cart items available in the cart.

It is a better and more reliable solution than the POST/order/:id as it doesn't need extra complexity of having only one "in-progress" order available per each user, and doesn't require too many unneded extra steps for the same result to be achieved.

Also, It accomplishes what earlier is stated by using the POST/order/:id, just so that this extra checkout endpoint that I added can better please and meet the requirements the "customer" had for this project.

Please check my readme for further descriptions on it all works.

POST /cart/checkout

{{host}}cart/checkout

AUTHORIZATION Bearer Token

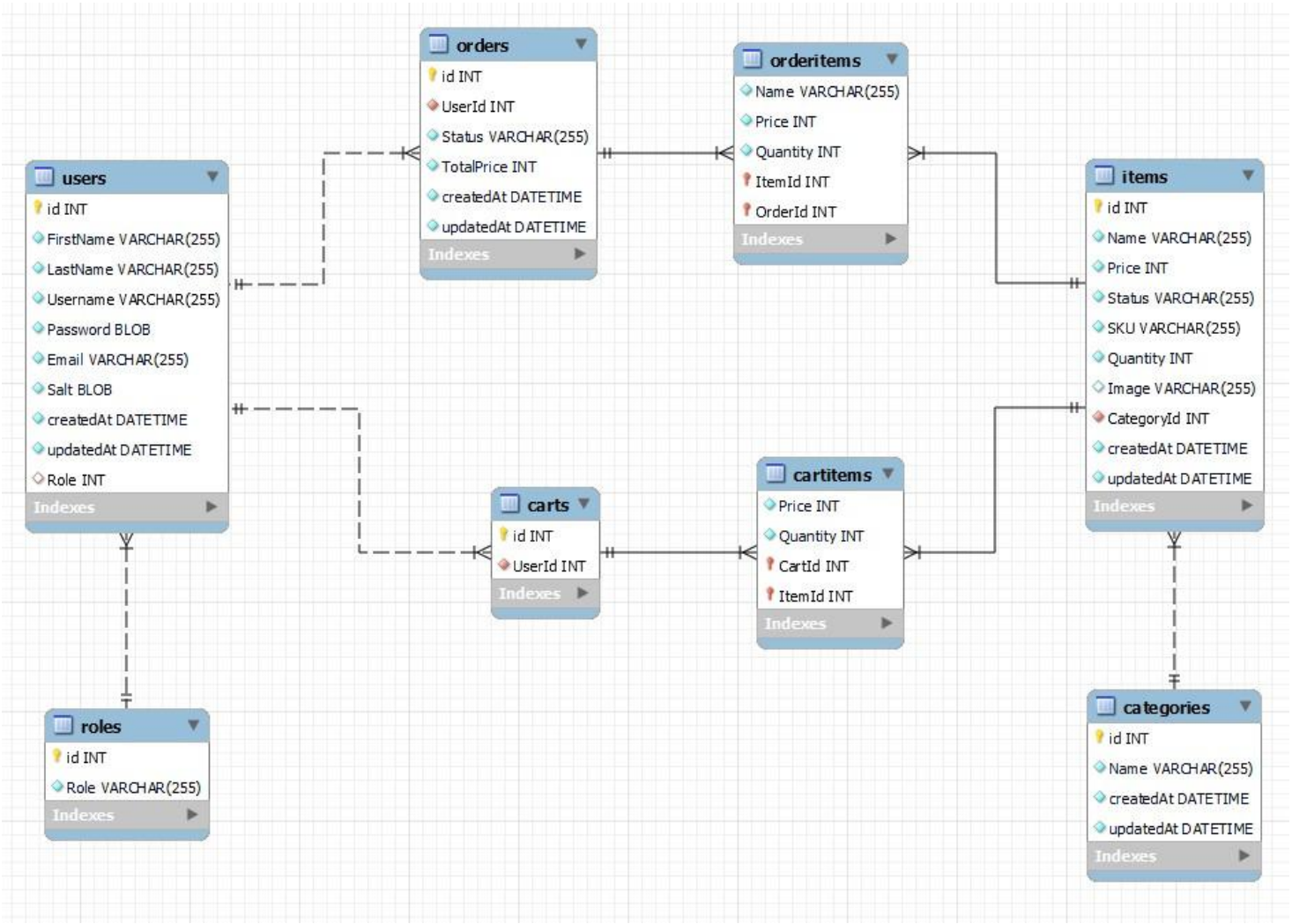
Token <token>

Postman documentation

https://documenter.getpostman.com/view/25181873/2s93mBwyw9

The pdf format isn't showing all the examples (apparently they can't be printed as pdf), but is included as requested.

EER of tables and their relationships



Role has many User (One-to-many);
Category has many Item, Item belongs to Category (One-to-many);
Cart belongs to User (One-to-one)
User has many Order, Order belongs to one User (One-to-many);
Super many-to-many relationships:
1- Item/Cart:

- Item belongs to many Cart (through CartItem) , Cart belongs to many Item (through CartItem); (many-to-many)
- Cart has many CartItem, CartItem belongs to Cart; (One-to-many)
- Item has many CartItem, CartItem belongs to Item; (One-to-many)

2- Item/Order:

- Item belongs to many Order(through OrderItem), Order belongs to many Item(through OrderItem);
- Item has many OrderItem, OrderItem belongs to Item; (One-to-many)
- Order has many OrderItem, OrderItem belongs to Order; (One-to-many)

As the Sequelize documentation explains, a super many to many relationship is the best of two worlds and allows for any kind of eager loading and deeply nested includes.
(Don't worry, I've used the raw queries where requested).
Plus, it kind of was set to be like this,as the project document requests orderitems/cartitems itemId to be referencing the items ids from the item table.

ROADMAP with Epics and stories

Sprint

- ESSP-1 As the owner of the stock-control system I want a relational based database, so that data can be stored and easily retrhieved
  - ESSP-2 Create main backend fundaments of the application such as tables and their relationships (sequelize ORM)
  - ESSP-3 Implement utility /setup endpoint for the first population of roles, admin user, categories and items from Noroff API
- ESSP-4 As a user I want to be able to interact with the data, so that I can rethrieve or modify it
  - ESSP-5 Implement authorization with JWT and signup login endpoints
  - ESSP-6 Implement category and item endpoints
  - ESSP-7 Implement Cart and cartItem endpoints
  - ESSP-8 Implement Order and orderItems endpoints
- ESSP-9 As the developer of the stock-control system I want to implement testing, so that I can make sure that all works as intended
  - ESSP-10 Do some testing of all endpoints manually, triple check that all requirements are met
  - ESSP-11 Implement unit testing with Jest and Supertest as per requirement
- ESSP-12 As a developer of the stock-control system I want to document the system and its api endpoints, so that it easily understandable how it works
  - ESSP-13 Add detailed postman documentation with examples for each endpoint available + a detailed readme for installation and usage
  - ESSP-14 Write a retrospective report of progression and challenges met with the project, how the relationships of the tables are set and work and ss of the roadmap

Retrospective on progression

My planning through scrum sprints was in the beginning a bit messy, as I attempted to make it a structured plan with all the requirements from the project documentation as tasks belonging to stories. But after some thorough thinking I landed on a simpler set up which was easier to follow, and as per the requirements, I just went back reading the project documentation thoroughly to spot every single requirement that has been spread around various parts of the document.
The progression went well, the second sprint was started earlier than it was set for, but took a little longer, in general though I managed to respect my 3 weeks time limit that I set with 1 week timeframe per sprint.

Challenges faced during the development of the project

One first challenge was as earlier described laying properly out a progression plan.
Other challenges met thereafter were fully understanding, once again, the 3N form and how to not overthink it at the same time. I found a good and simple article describing in detail how to achieve the 3n form (it is linked underneath this section in Sources used for this project).

Another fun but a bit challenging part, as I hadn't dealt too often with such an issue previously, was the use of the reduce method to rearrange the array results retrieved from the raw queries. Of allcarts and allorders. Once understood and achieved the desired result though it felt easy and straightforward to use such a method. (sources for this part are also linked underneath).

Not a challenge, but parts that I went researching (refreshing my memory) on, in order to fully understand the best practices of them, were the use of associations, hooks and operators in sequelize. (also linked in the sources part underneath).

To conclude, my main biggest challenge was probably having to interpret the document, its wording and all its entangled requirements.
Some requirements are contradictory to one another, others are specific in some parts and vague in others parts.
There are multiple incongruences between what is stated in the whole documentation with what is required through the API requirements.

To make an example of what I mean with those comments, I could reference for example the users should see complete orders only (and orders history), in 2 of the sentences you refer to this very specifically but then in the last sentence, you state that Users should see all their orders. I decided to interpret that last sentence as, users should be shown order status, but only admin (a type of user as well) should see all types of orders and their relative status.

Another good example of vaguely set specifications was the POST order/:id which didn't really give any hint on what that parameter was meant to relate to, it actually took several students asking about this on chat channels to actually have this outlined and explained.
For instance, it couldn't possibly be an order id, as this isn't created anywhere else prior to that endpoint. Furthermore, as I explain in my readme and documentation, I find it kind of weird that this should be set up as specified by the teacher in the chat channels.
Why set up a backend that requires another developer working on the frontend of the system to set a loop on an endpoint call, when the same results could be achieved by having the backend developer setting the endpoint to automatically do a complete checkout/order of all the items in a cart, with one simple call.

I understand this, as it has been echoed by the teachers several times, is an attempt at creating a

“real-world” type of scenario for us students, where a customer might or might not be a developer and might or might not be completely clear with the requests in the project specifications.
But, as this exam document both gives very strict rules on how some parts should be built and at the same time is vague and at times even has requirements contradicting each other, it doesn’t really fit into one nor the other type of “customer” one might meet under professional work circumstances.
If the customer hasn’t got development experience, there probably wouldn’t be such strict requirements. On the other hand, if the customer is actually a developer, I believe the requirements wouldn’t have the prior mentioned flaws.

Please look at this as constructive feedback, I mean no disrespect with what I’m stating here.
I, as a student, was looking forward to having the opportunity to showcase how much I’ve learned during this year of studies, hoping to be given some main project requirements, such as the ones that an internship project would have had, and then build a personal project based on those main requirements, build it to be as efficient and well structured as I possibly could, while maintaining the main requirements and using concepts and methods/technologies that were thought during this year.

Instead I’m left with a feeling of being given a project that wasn’t fully quality assured, restricts my possibilities to showcase, at full, my acquired knowledge (since most requirements points straight to how and what to do) and rather sets too much weight on the students capability of interpreting the requirements given and understand what the “customer” really meant/desired with them.

I understand that this is a vital skill to have as a developer, but I believe that in an exam like this one, the evaluation of my developer knowledge and skills, gathered through the year, should be prioritized above all other types of soft skills that a developer should have and obtain through work experience and practice.

Sources used for the project

- Various module code examples from BED courses
- For proper understanding of the 3N form: https://simplesqltutorials.com/third-normal-form/
- Associations and hooks:
    - https://sequelize.org/docs/v6/core-concepts/assocs/
    - https://sequelize.org/docs/v6/other-topics/hooks/
- Use of operators:
  https://sequelize.org/docs/v6/core-concepts/model-querying-basics/#examples-with-opand-and-opor
- Understanding the buffer for deciphering passwords on login:
  https://stackoverflow.com/questions/66226092/how-to-use-buffer-from-with-crypto-timingsafeequal
- Supertest basics: https://github.com/ladjs/supertest
- Reducing results from raw queries:
    - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce
    - https://stackoverflow.com/questions/47840445/js-reduce-array-to-nested-objects
- My brain and a lot of trial and error.