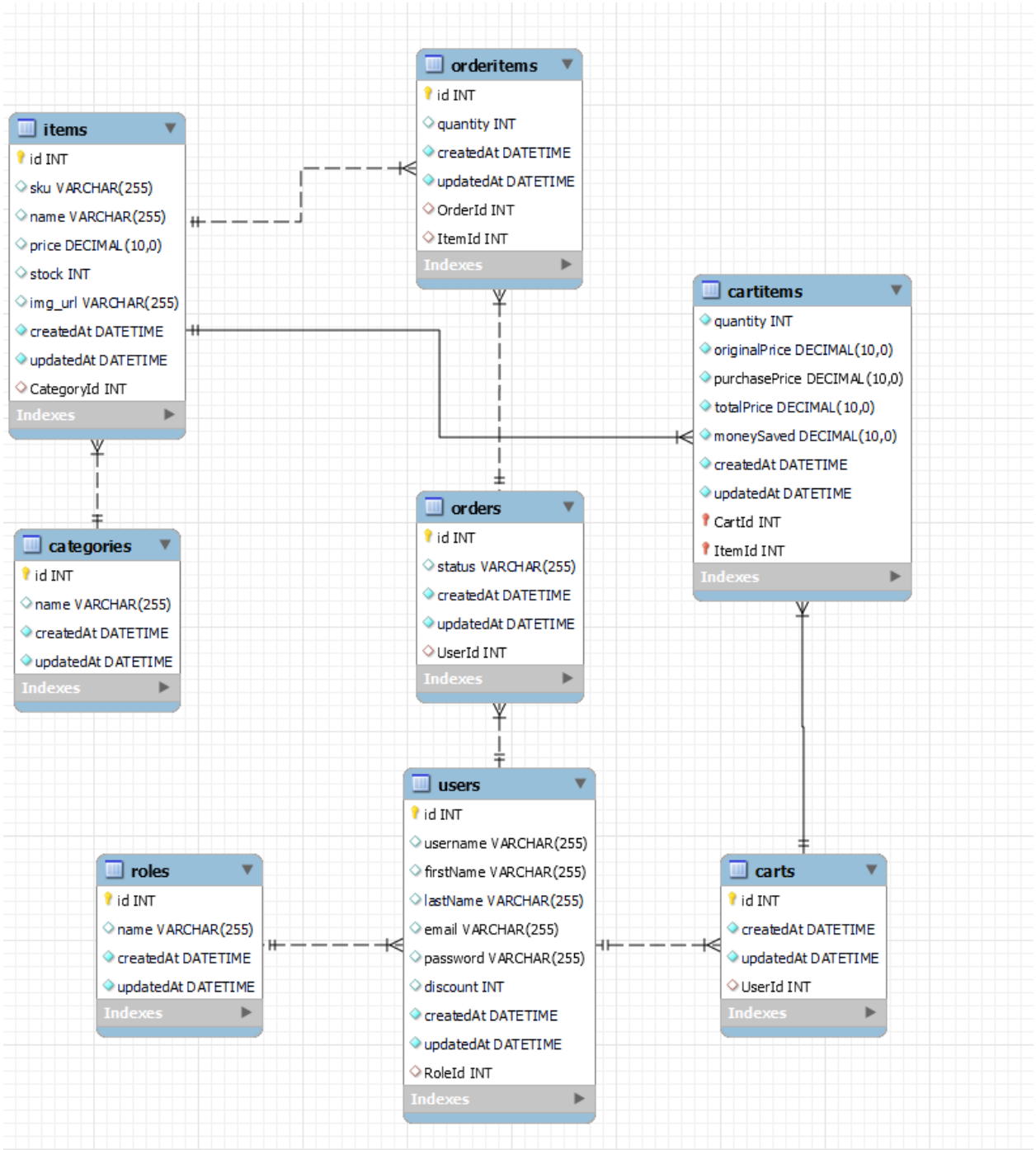
Retrospective Report

Screenshot of the complete Database ERD (Showing all tables, relationships, properties)



Role has many Users: This is a one-to-many relationship, where a Role can have multiple Users associated with it, but a User can only have one Role.

User has one Cart: This is a one-to-one relationship, where a User can have only one Cart associated with it, and a Cart belongs to a single User.

Category has many Items: This is a one-to-many relationship, where a Category can have multiple Items associated with it, but an Item can belong to only one Category.

Cart belongs to User: This is a one-to-one relationship, where a Cart belongs to a single User, and a User can have only one associated Cart.

Item belongs to Category: This is a many-to-one relationship, where an Item belongs to a single Category, and a Category can have multiple associated Items.

Cart belongs to many Items (through CartItem): This is a many-to-many relationship, where a Cart can have multiple Items associated with it, and an Item can be associated with multiple Carts. This relationship is facilitated through the CartItem join table.

User has many Orders: This is a one-to-many relationship, where a User can have multiple Orders associated with it, but an Order belongs to a single User.

Order belongs to User: This is a many-to-one relationship, where an Order belongs to a single User, and a User can have multiple associated Orders.

OrderItem belongs to Order and Item: This is a many-to-many relationship, where an OrderItem is associated with a single Order and a single Item. An Order can have multiple OrderItems, and an Item can be associated with multiple OrderItems.

Cart has many CartItems: This is a one-to-many relationship, where a Cart can have multiple CartItems associated with it, but a CartItem belongs to a single Cart.

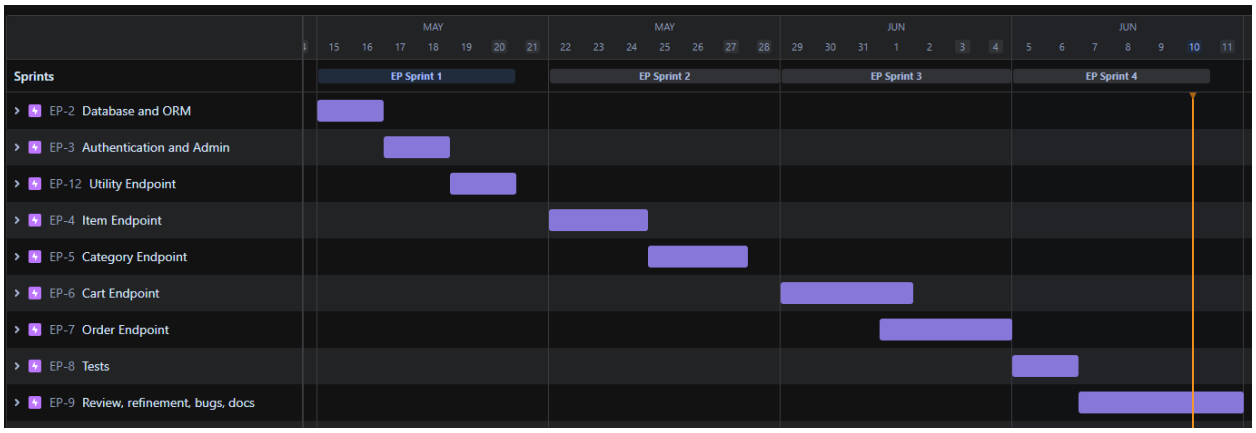
CartItem belongs to Cart: This is a many-to-one relationship, where a CartItem belongs to a single Cart, and a Cart can have multiple associated CartItems.

Item has many CartItems: This is a one-to-many relationship, where an Item can have multiple CartItems associated with it, but a CartItem belongs to a single Item.

CartItem belongs to Item: This is a many-to-one relationship, where a CartItem belongs to a single Item, and an Item can have multiple associated CartItems.

These relationships define how the tables are related to each other in the database schema, allowing for querying and retrieving data in a smooth way.

Screenshot of only the Jira Roadmap (Showing Epics and Sprints)



Summary

In this assignment the task was to create a stocksalesdb based off of guidelines from the school. The project allowed for some free-thinking, i.e. creativity, withing the specific requirements.

The first challenge was to create the Sequilize model, which is a crucial part of the project. In retrospect I really should have tested the API call first to know all the data being returned, instead of creating mock values and then adjust later on.
The model was created and after some modifications to align with the required functionality, such as “hasMany”, “belongsTo”, “belongsToMany” and some FK, it all seemed to work as intended.

During the coding process for the endpoints it became apparent that the model needed some further adjustments to better meet the criteria for each endpoint.

I had to do several drop table and modifications, which in hindsight could have been avoided.

After finishing all the code I went back because I wanted a slightly different response body from my POST order:id.
This led to even further modifications on the model, and some late changes to the code as well. Even though it did lead to extra work, I believe the response body looks better now and was worth the extra effort.

Endpoints, rewrites and refining

The first endpoint Authentication and Admin, which went rather smoothly. At the end of the project I realized I could have added a separate middleware to check for admin, instead of repeating myself throughout the code. So, even though I thought this went smoothly, I could have spent a bit more time to think this through as to stick to the DRY principle in the rest of the code. It led to a bit more extra work, but a valuable lesson was learned.

The setup endpoint went off fairly smooth. After watching some tutorials on youtube I decided I would try node-fetch instead of axios, which we have used previously. The setup endpoint went well and populated the database as intended.

On to the item endpoint. That too went rather well. After a little trial and error I was happy with the result and moved on to category.

The category endpoint seemed to go well too. Not to much trouble there. Some minor adjustments to the model, but otherwise it went rather swimmingly.

Now, the cart endpoint is where things started to get a bit more tricky. After quite a few adjustments to the model, I was quite happy with the endpoint. But then I decided to start fiddling with the stock quantity. I found a solution to updating the stock, and checking the stock, when adding items to cart. I even made a function that returned items to stock when items were deleted from cart.

When I got to the orders endpoint I realized that it was here the update to stock should happen. Luckily, it was pretty easy to remove the logic from cart and move it over to order instead. I assumed, since stock were to be removed from this part, this was were the discount should be applied. So I made it so. Only to realize it would make little sense to show the customer the final price so late in the process. If I were a customer I

would have liked to have known the price when putting the item in the cart. So I moved that logic back to cart instead. There had to be some minor changes to the model when dealing with this endpoint too.

When it came to put order:id I was faced with a choice; since the items are removed from stock when a user does a post, should it be added back in if an Admin changes the status to “Cancelled”?

Response bodies

During this project I was making a backend for a fictional database with no knowledge of what data the front-end would like to show to the customer.

Of course, I could just send all the data on each response (unless something else was specified in the assignment). But if we picture this on a larger scale that would lead to a lot of data being sent and possibly a slow and frustrating experience for the user.

Ultimately, I decided on the various response bodies based off of what I presumed the front-end, and customer, would like to see.
If this were a real life scenario the communication between front-end, back-end, research etc would have been key to make the experience as good as possible for the customer. If some responses have too much or too little information that could have been addressed pretty easily and handled quickly as the project was moving along.

Testing

In regards to the tests, I added an extra endpoint of delete user:id, because I thought this would look more uniform and concise in regards to the rest of the tests.
For some reason, I could not get the /setup test to work the way I had the setup route set up. After reading a lot of documentation and consulting my trusted rubber ducky, I rewrote the setup route to use Axios instead of node-fetch.
After that the rest of the tests went pretty good. Why node-fetch works with postman but not with my tests I do not know. There could have been an oversight on my part on how to setup the test, variables etc, when using node-fetch instead of Axios.

I added a check in the setup test to see if the database already was populated, even though that was not required, I thought it looked cleaner.

Last minute changes

As I was making all the documentation and checking every single possible outcome of each endpoint and checking everything up against the specs again, I realized I had made some errors and where I could do some improvements.

For example, I had missed the fact that allcarts should be a raw sql query.
And I made some improvements to response body under order, to make it more user friendly (here I for example added a money saved so the user would see how much they saved and would be encouraged to do more shopping).
This created a bit of a rabbit hole where I had to change several things in order to make it work like I would like it to.

Ultimately, I believe it was the right decision as now there is better functionality.

I had a great idea of reserving a username for testing, but when I ran the test it gave me an error because the username was reserved. I could have done a work-around, but decided the solution that would cause less possible future errors, in regards to this project, was to add an instruction in the readme on what username to avoid during testing.

I had implemented a regexvalidation check to make sure the password was strong, with prompts on what the password should contain (lowercase, uppercase, special sign, number) but ultimately I decided against this as it could potentially be irritating for testing purposes. If this were ever to go live I would recommend adding that back in.

Documentation

I went through all the endpoints in Postman, checking that everything worked as intended, that the admin endpoints returned a 403 forbidden, etc.
Wrote up what each endpoint should do, what response could be expected, put examples for the request body and response body. I did not rearrange the code in order

to force a 500 error, as I did not deem that necessary.
There could have been something that I missed in testing where I should have added more information, but as per now I believe I did a thorough check of each endpoint and added adequate information and examples.

Postman Link

https://documenter.getpostman.com/view/25739749/2s93sZ6u76

Acknowledgements