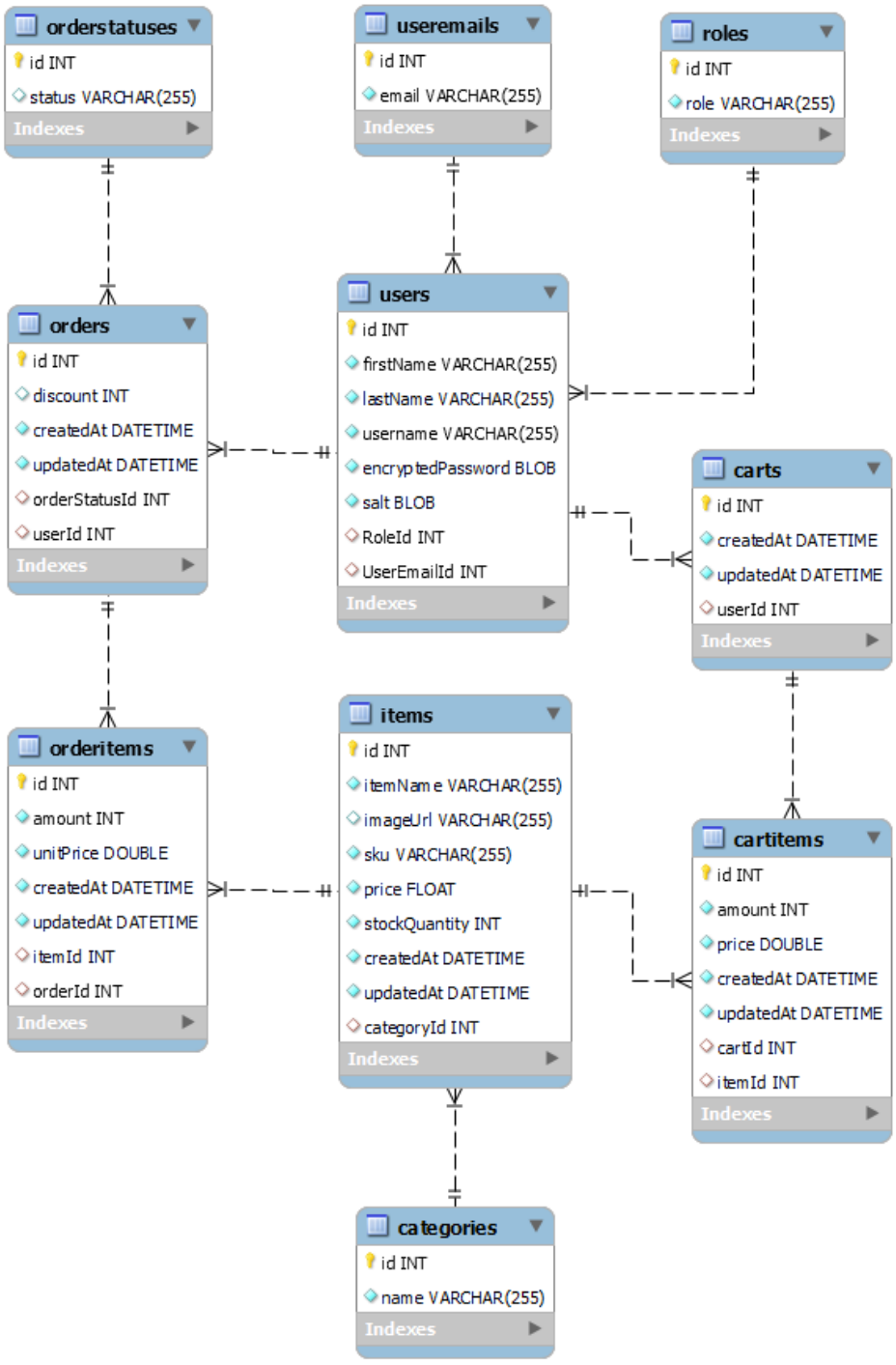
BED Exam Project

Documentation

Retro Perspective Report

1. Database

ERD extracted from MySQL workbench showing tables and relations.



The **users** data entity has one **role** and one **email**. Both role and email can be shared with several users and are stored in separate tables with a one to many relationship.

The **item** entities has one **category**, and one category may belong to several items, having a one to may relationship.

The **cart** entity can only belong to one **user** but the user can in principle have many carts, having a one to many relationships.

NOTE: For solution no user can have more than one cart. This is not constrained by database, but it is constrained by application.

The **cart item** entity can only belong to one **cart**, and only have one **item**, though the cart can hold several cart items, and items can belong to many cart items. These are one to many relationships.

The **order item** entities are similar to cart item entities have one to many relationships with **items** and **orders.**

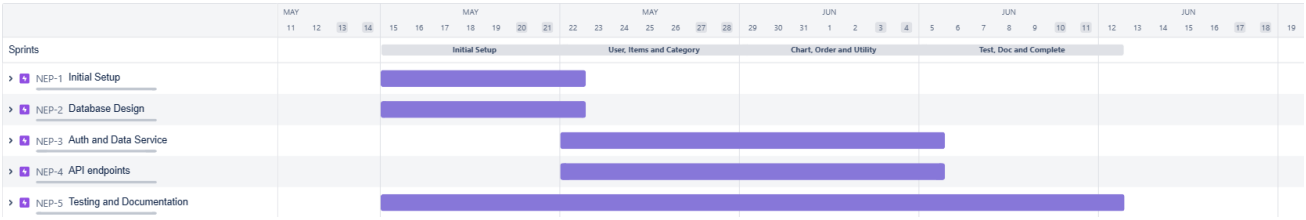
The **users** can have many **orders**, and one order can only belong to a single user, having a one to many relationship.

The **order** can have one predefined **order status**, and any order status can have several orders. Tables have a one to many relationship.

Timestamps are included for all entities where amount, quantity or similar values are expected to change.

Values that can be calculated for other data are not included in any tables as they are or at least should be calculated on query or processing.

Jira Roadmap



The Exam project was divided into 5 Epics and 4 Sprints. The defined epics are not kosher in accordance with agile, but represent a completed set of functionality inside the application. Each sprint are adding a set of features resulting in a functional application with extended features on completion. Tasks from two or more epics are concurrent in each sprint, including continuous updating of documentation.

Challenges / Retrospect

I wanted to keep the routes as clean as possible, so I chose to create an architecture where each each route was served by it's own designated data service. Any interaction with the data service not resulting in the desired outcome or dataset, would throw an error that would be handled by the route handler. The router handler would then relay the result with JSend as a success, fail or error depending on the request outcome.

Having JSend relaying error messages on **500 internal server error** is not a good solution, but left as is for development purposes.

I also created middleware for route handlers where validation of data in request body would require more than a few lines of code. Custom middleware was also created for securing endpoints.

Having one singular data service for each group of endpoints would result in some duplicate code, but seems to be limited to query methods only.

Not sure what should be strictly required in each entity, when posting data, I made everything strictly required, with the exception of updating items where only provided values are updated. I did not make item name required to be a unique value, but a warning is provided if an item with a similar name exists in the database.

When importing from Noroff API, the Sequelize **bulkCreate()** method was never a good option for a range of reasons. Instead I used loops to remap and insert the imported data. My first version of this method was created to insert any values existing in import missing in the database. But when reviewing the course assignment, I noted importing data should only execute if no data exists. I did a minor refactor to meet the requirement, but kept most of the code in the method as before for easier reversion to previous feature.

I never really been a huge fan of Object Relation Mappers, and honestly enjoy writing SQL code. In this solution I have used both Sequelize and raw queries through Sequelize for CRUD operations. Especially minor queries regarding a single value like ID have been my prime target for raw queries. There is a minor discrepancy in the data format when queried either by admin or user account. Since having clear definition of what data and in what format it is expected as return value, I left it as is.

The test part of this application had a real Catch 22:

post /search – search for all items with the category name "Laptop" (one item should be returned from the initial data).

My honest interpretation is that this test is supposed to query all times with the category name "Laptop", and expecting at least one item. But "Laptop" do not exist as a category in the dataset provided for the test. Not sure if I was supposed to write the right test to fail, or the wrong test to pass, I decided on the first alternative. So, this test will fail.