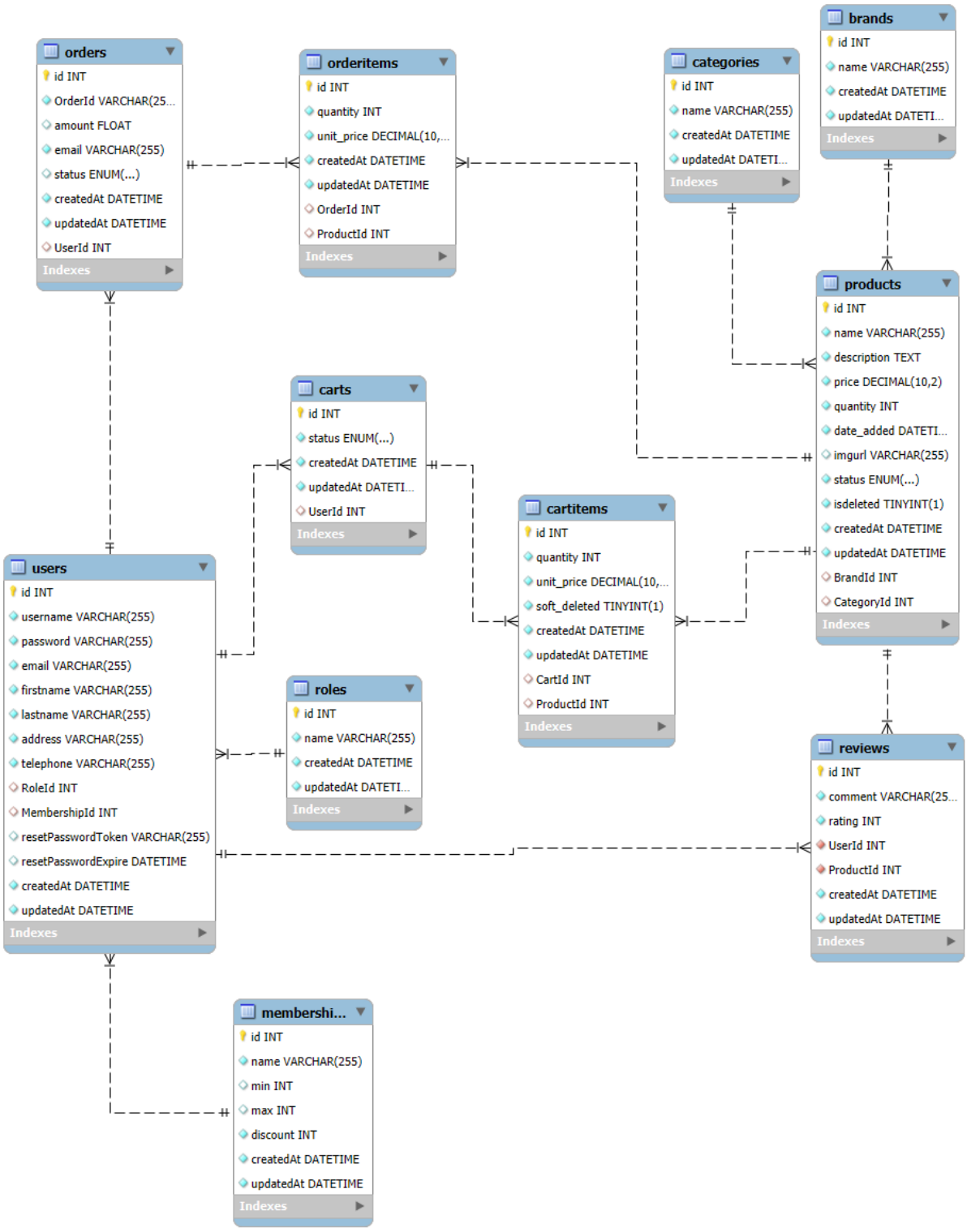
EXAM PROJECT 1 – REFLECTION REPORT

By Camilla Seeland JAN24FT

Table of Contents:

Screenshot of Database:



Tables and Their Relationships

1. users Table

Relationships:

- One-to-Many with orders: A user can place multiple orders (users.id -> orders.UserId).
- One-to-Many with carts: A user can have one or more carts (users.id -> carts.UserId).
- One-to-Many with reviews: A user can write multiple reviews (users.id -> reviews.UserId).
- Many-to-One with roles: Each user has a specific role (users.RoleId -> roles.id).
- Many-to-One with memberships: Each user can have a specific membership level (users.MembershipId -> memberships.id).

2. roles Table

Relationships:

- One-to-Many with users: A role can be assigned to many users (roles.id -> users.RoleId).

3. memberships Table

Relationships:

- One-to-Many with users: A membership level can be assigned to multiple users (memberships.id -> users.MembershipId).

4. orders Table

Relationships:

- One-to-Many with orderitems: An order can have many order items (orders.id -> orderitems.OrderId).
- Many-to-One with users: An order belongs to a user (orders.UserId -> users.id).

5. orderitems Table

Relationships:

- Many-to-One with orders: Order items belong to an order (orderitems.OrderId -> orders.id).
- Many-to-One with products: Each order item refers to a specific product (orderitems.ProductId -> products.id).

6. carts Table

Relationships:

- One-to-Many with cartitems: A cart can have multiple cart items (carts.id -> cartitems.CartId).
- Many-to-One with users: A cart belongs to a user (carts.UserId -> users.id).

7. cartitems Table

Relationships:

- Many-to-One with carts: Cart items belong to a cart (cartitems.CartId -> carts.id).
- Many-to-One with products: Each cart item refers to a specific product (cartitems.ProductId -> products.id).

8. categories Table

Relationships:

- One-to-Many with products: A category can have multiple products (categories.id -> products.CategoryId).

9. brands Table

Relationships:

- One-to-Many with products: A brand can have multiple products (brands.id -> products.BrandId).

10. products Table

Relationships:

- Many-to-One with brands: A product belongs to a specific brand (products.BrandId -> brands.id).
- Many-to-One with categories: A product belongs to a specific category (products.CategoryId -> categories.id). One-to-Many with orderitems:
- A product can appear in multiple order items (products.id -> orderitems.ProductId).
- One-to-Many with cartitems: A product can appear in multiple cart items (products.id -> cartitems.ProductId).
- One-to-Many with reviews: A product can have multiple reviews (products.id -> reviews.ProductId).

11. reviews Table

Relationships:

- Many-to-One with users: A review is written by a specific user (reviews.UserId -> users.id).
- Many-to-One with products: A review belongs to a specific product (reviews.ProductId -> products.id). Explanation of Relationships

One-to-Many:

Users and Orders: A user can place multiple orders, but each order is associated with one user.

Orders and OrderItems: An order can contain multiple items, but each order item belongs to one order.

Products and Categories: Each product belongs to one category, but a category can contain multiple products.

Many-to-One:

Users and Memberships: Each user has one membership, but a membership can be assigned to many users.

Products and Brands: A product belongs to a single brand, but a brand can have many products.

Many-to-Many:

Users and Products: Through orders and carts, users are indirectly related to products. Users can purchase or add multiple products to their cart.

Sceenshot of the Jira Roadmap

PROJECT REFLECTION

Discussion of the progression of the project.

Week 1: 25.11-1.12.24

The first week was focused on setting up the development environment and handling database models and relationships. I spent a considerable amount of time designing the models and figuring out their relationships, which led to several changes throughout the project. The first success was implementing a "GET" request to fetch all products. Later, I completed CRUD operations for the product model and added a search route. Authentication was also a priority, including user registration and JWT token generation for secure routes.

Week 2: 02.12.24-08.12.24

The second week was dedicated to CRUD operations for users, brands, and categories. A significant challenge was making the frontend and backend communicate properly. I misread the instructions and thought both apps had to run on the same port, leading to much confusion. After troubleshooting, I realized the frontend and backend needed to run on different ports (frontend on 3001, backend on 3000), as the instructions had stated. By the end of the week, I implemented cart and order functionality and began working on the review model.

Week 3: 09.12 - 15.12.24

In the third week, I focused on Swagger integration for API documentation and using Jest for testing. I encountered issues logging in as an admin, which blocked progress on running tests. The main problem was the shared middleware for app.js and testing, causing unnecessary complexity. I created a new testManagement.js middleware specifically for testing. Additionally, I developed a new userManagement.js middleware for the frontend, using role-based authentication to differentiate between users, guests, and admins. I eventually resolved the token and authentication issue by modifying the backend routes and using RoleId = 1 for admin authentication. I also started working on the Admin Dashboard, with a few CRUD operations working by the end of the week.

Week 4: 16.12 - 22.12.24

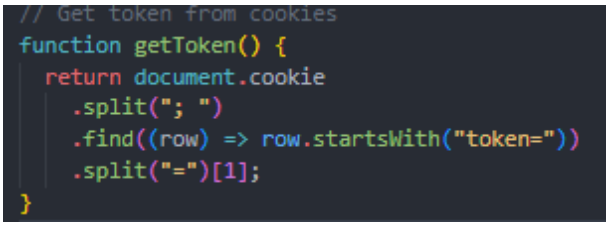
The final week was focused on bug fixing, testing, and styling the frontend. I identified several bugs with authentication, token injection, typos in aliases, and data not being fetched correctly. These issues were resolved by fixing token handling and ensuring routes were correctly protected. The frontend-backend communication continued to be a challenge, but most issues were resolved. Although many frontend functions still had issues, the backend was sending the correct data in JSON format.

Challenges Faced During the Development of the Project

Token and Authentication Issues

Managing authentication and token handling between the frontend and backend was one of the main challenges. Initially, I incorrectly injected the token directly into the frontend template, causing malfunctions. After debugging, I switched to using cookies to store and pass the token in headers, resolving the issue.

Changed from getToken():



```
// Get token from cookies
function getToken() {
  return document.cookie
    .split("; ")
    .find((row) => row.startsWith("token="))
    .split("=")[1];
}
```

To adding the token in the request with cookies:

```
    // Pass the token and categories to the frontend
    res.render("admin/categories", {
      title: "Manage Categories",
      categories,
      token, // Pass the token to the template
      user: req.user,
    });
  } catch (error) {
    console.error("Error fetching categories:", error.message);

    // Render the page with an empty array and token
    res.render("admin/categories", {
      title: "Manage Categories",
      categories: [],
      token: req.cookies.token,
      user: req.user,
    });
  }
});
```

```
try {
  const response = await fetch(url, {
    method: method,
    headers: {
      "Content-Type": "application/json",
      Authorization: `Bearer ${token}`, //
    },
    body: JSON.stringify(categoryData),
  });
```

Middleware and Admin authentication:

I was originally adding the authentication to my routes like this: *authorize([admin]),*

But for some reason it did not see username = admin as Admin, as well as user.role.name as Admin. Solution: Switching to **RoleId** for authentication solved the dilemma. Ensuring that RoleId = 1 is always Admin in the InitializeDatabase.js file. When changing to: **authorize([1]),** It finally recognized the user as Admin.

Port problems:

I initially tried to run both the frontend and backend on the same port, which led to several errors. I eventually realized I needed to use different ports for both applications

(3001 for the frontend and 3000 for the backend), which the instructions had clearly stated but I misinterpreted.

```
GET /categories 404 8.083 ms - 1426
Error fetching categories: Request failed with status code 404
GET / 200 41.786 ms - 6513
GET /stylesheets/style.css 304 1.799 ms - -
GET /path/to/global-styles.css 404 3.728 ms - 1426
GET /images/shopit_logo.png 304 0.535 ms - -
GET /images/banner-bg.jpg 404 1.106 ms - 1426
PS C:\Users\iCami\OneDrive\Skrivebord\examp_v1\ep-ca-1-SheepyCami\front-end> ^C
PS C:\Users\iCami\OneDrive\Skrivebord\examp_v1\ep-ca-1-SheepyCami\front-end> npm start

> front-end@0.0.0 start
> node ./bin/www

Port 3000 is already in use
PS C:\Users\iCami\OneDrive\Skrivebord\examp_v1\ep-ca-1-SheepyCami\front-end> npm start

> front-end@0.0.0 start
> node ./bin/www
```

.env file:

I struggled with connecting the frontend and backend middleware and authentication. Initially, I thought a .env file in the frontend alone would work. However, I later realized that both the frontend and backend needed identical secret keys in their .env files to make token handling work correctly.

```
]
GET / 304 25.308 ms - -
Invalid token: secret or public key must be provided
Invalid token: secret or public key must be provided
GET /stylesheets/style.css 304 1.729 ms - -
Invalid token: secret or public key must be provided
GET /path/to/global-styles.css 404 2.662 ms - 1426
Invalid token: secret or public key must be provided
GET /images/shopit_logo.png 304 1.130 ms - -
Invalid token: secret or public key must be provided
Invalid token: secret or public key must be provided
GET /images/banner-bg.jpg 404 2.033 ms - 1426
```

```
23
PROBLEMS  OUTPUT  DEBUG CONSOLE  PORTS  GITLENS
TERMINAL
> node ./bin/www

Token verification failed: secret or public key must be provided
GET /admin/dashboard 404 25.536 ms - 1426
Token verification failed: secret or public key must be provided
GET /admin/login?identifier=admin%40noroff.no&password=P4ssword2023&login-button= 200 4.805 ms - 3714
Token verification failed: secret or public key must be provided
GET /path/to/global-styles.css 404 10.073 ms - 1426
Token verification failed: secret or public key must be provided
GET /admin/dashboard 404 4.416 ms - 1426
```

Jest and Supertest

For a day, all my tests failed, which was frustrating. I reviewed the testing setup and discovered that I needed separate test files, including one specifically for testing login. I also had issues with the middleware not functioning correctly in test files, so I created a new middleware and a test.js route just for testing purposes. Once the login test was successful, I was able to complete other tests.

Cart and Orders

I misunderstood the brief and thought only admins could change the membership status of users. This led to issues with order items, which were received as empty objects. Additionally, I didn't account for how quantities purchased by users would affect membership and discounts. I also misunderstood the cart deletion process, thinking it had to be soft deleted, and later realized only the products required that functionality.

Frontend

Even though the tokens and middleware were extremely exhausting, which will haunt my dreams for the rest of my life, nothing can compare to the frontend part of this project. I wish I could say I filled the frontend folder with my proudest work, but sadly it is extremely messy, illogical and probably way more complicated and hard-coded than it had to be. Whatever worked in the end, I left that way, to not waste time trying to fix it again. Because most of my time went to solving communication between the 2 folders, I did not dare to touch it once something finally worked, and it is sort of left in a graveyard-state, swimming with other code that was created at a heart-breaking attempt of loading data for an Admin table. I can safely say that frontend is not for me. The backend was fun, challenging, taught me a lot of things, but the frontend was tears blood and sweat, and I think looking at my frontend code you can see the frustration in every line.

Features that were planned, but not finished developing due to time constraints

"resetPasswordToken": null, "resetPasswordExpire": null: You will see this in the Model and database, but I did not have enough time to implement a reset password feature as planned to use Nodemailer.

Reviews feature and model: I did not have time to finish setting up reviews for the frontend, but you can test the endpoints with the backend! Admin was also supposed to have access to reviews to delete any inappropriate reviews from the website, but I decided to remove the reviewManagement.js file since I was running out of time.

Payment with Stripe: As you can see in the Admin Dashboard, there is total sales section there, this is because I was going to try to learn Stripe for this project!

Admin Dashboard: some functions in the dashboard are not working properly in the frontend version, but in the backend all endpoints should not leave errors except for cart items.

Search bar in frontend: I could not, for the death of me, get this to work. It is working in backend but getting it to fetch the data from backend seemed almost impossible. As much as I'd love for it to work, I had to give up this battle.

Cart count: The cart in frontend was supposed to display the number of items added. I could not get this to work properly and decided to delete it.

References and source code used

References

- **Udemy Courses:**
  - ***Natours app*** *- Master Node by building a real-world RESTful API and web app (with authentication, Node.js security, payments & more)*, Course: Node.js, Express, MongoDB & More: The Complete Bootcamp | Udemy
  - ***ShopIT e-commerce*** *- Build Powerful E-commerce Project using React, Redux Toolkit, Node.js, Express, MongoDB, Stripe from DEV to DEPLOYMENT*, Course: MERN Stack Full Ecommerce Site - Using React, Redux, Node.js | Udemy
  - ***Lebaba e-commerce*** *- This is a full-featured e-commerce project using the MERN stack (MongoDB, Express.js, React, Node.js)*, Course: Building a Full-Stack MERN E-commerce Website | Udemy
- **Noroff Code:**
  - *Srv m3: RV - Module 3 - TDD lesson*. Retrieved from https://github.com/noroff-bed1/srv_m3_l4_tdd
  - ***User login test***, sourced from Noroff's module 3.4. Test Driven Development (TDD) - User login test.
- **AI Used:**
  - **ChatGPT. (2024).**
- **Tutorials and Videos:**
  - ***Node and Express*** *- Connect to MySQL Database with Sequelize*. Retrieved from

https://www.youtube.com/watch?v=pKhdNPN4b1g&list=PL2HhFRI8JWYGr93nOp19qCCSr9Rj3VTcy&index=19&t=726s
    - ***Sequelize Tutorial: Episode 4*** - *Model Querying*. Retrieved from https://www.youtube.com/watch?v=jWdVy265Q-A&list=PL2HhFRI8JWYGr93nOp19qCCSr9Rj3VTcy&index=21
    - ***How to Create a Stunning Portfolio Website with Nextjs, Tailwind CSS and Framer-motion*.** Retrieved from**:** https://www.youtube.com/watch?v=Yw7yWHigGKI&list=PL2HhFRI8JWYGr93nOp19qCCSr9Rj3VTcy&index=13
    - ***Responsive Portfolio Website From Scratch***. Retrieved from: https://www.youtube.com/watch?v=ldwlOzRvYOU&list=PL2HhFRI8JWYGr93nOp19qCCSr9Rj3VTcy&index=14
- **Source Code References:**
    - ***Codes for Token, sendToken, JWT***: Inspired from the Udemy course *Master Node by building a real-world RESTful API and web app (with authentication, Node.js security, payments & more)*, Section 10: Authentication, Authorization, and Security.
    - ***Authentication/Middleware Code*:** Sourced from Noroff's own project files in the server section: srv m3. Available on GitHub.
    - ***API's and API Design*:** Inspired by *Master Node by building a real-world RESTful API and web app (with authentication, Node.js security, payments & more)*, Section 6: Express: Let's Start Building the Natours API!
    - ***Web Design and Layout***: Main inspiration for the webpage design and layout was Komplett (Komplett.no).
    - ***CSS, Styling, Headers, Footer, Components***: Heavily inspired by the ShopIT app from Udemy: *Build Powerful E-commerce Project using React, Redux Toolkit, Node.js, Express, MongoDB, Stripe from DEV to DEPLOYMENT*.
    - ***Database Connection (Models/Index.js)*:** Direct copy with minimal changes from the index.js file in Noroff's srv m3 folder.
    - ***Models***: Inspired from the following Udemy tutorials: *Build Powerful E-commerce Project using React, Redux Toolkit, Node.js, Express, MongoDB, Stripe from DEV to DEPLOYMENT* and *This is a full-featured e-commerce project using the MERN stack (MongoDB, Express.js, React, Node.js)*.
    - ***Frontend Javascript Management Files*:** Heavily sourced from AI (ChatGPT). Retrieved from https://chatgpt.com/.
    - ***Frontend Tables, Modals, EJS Files***: Heavily sourced from AI (ChatGPT). Retrieved from https://chatgpt.com/.
    - ***Backend Controller and Service Files*:** Functions and file structure inspired by the ShopIT app. *Build Powerful E-commerce Project using*

React, Redux Toolkit, Node.js, Express, MongoDB, Stripe from DEV to DEPLOYMENT.

- ***Routes***: Heavily inspired by the ShopIT app. *Build Powerful E-commerce Project using React, Redux Toolkit, Node.js, Express, MongoDB, Stripe from DEV to DEPLOYMENT*.
- ***Swagger and Jest*:** Assistance from ChatGPT, as well as Noroff's tutorial on Test Driven Development (TDD), Section 3.4: *Setting up for test driven development*. Retrieved from https://noroff.bravais.com/s/4sjBtVQSL2YDXmkDOaVB