UNIVERSITY OF STAVANGER, NORWAY

DAT240 Project Report (Group Mor_Qui_Sun_Tis_Lau): Campus Eats

João Lucas Moraes, Jesus Sanchez, Joachim Tisøv,
Anna Sund, Martin Sævareid Lauritsen

December 6, 2024

Contents

1 Introduction

Person in Charge; all
We decided to go for the project Campus eats as it looked like an interesting challenge with its normal and advanced requirements. Campus eats is an application similar to apps like Foodora or Wolt with a slightly dialed down scope. Through Campus eats a student should be able to login, order food from the school cafeteria and have it shipped to their specified building and room. They should also be given live updates on the status of their orders and see a detailed view over their past orders. Any user should also be able to register as a courier then fulfill placed orders. By delivering order these couriers should recieve a pre determined cut of 80% of the delivery fee and any tip the user might leave after order delivery. The courier should be able to look at all available orders, all orders they are or have been responsible for aswell as a detailed view over their earnings. The admins should be able to apply simple CRUD features on the items available for ordering, invite new admins, see both a filtered and unfiltered list of all orders, an earnings overview and a tool to edit the global delivery fee for future orders. This application implements several features needed in a full scale implementation such as payment handling through Stripe and login / authentication through 3rd parties like google and facebook using the industry standard OAuth2. The aim for this project is to get as close to a deployment ready product on the software side as possible.

2 Design & Organization

2.1 Block Diagram

The related document is in the relations.md file inside the /Mor_Qui_Sun_Tis_Lau folder

The block diagram is a contiued version of lab3, we have added an admin context which is placed outside of the main application since an admin won't interact like a customer and can only preform product CRUD-operations, responding to courier requests, change the delivery fee and invite new users to the admin board. The process of ordering is connected together with the product, cart and ordering context. The fulfillment and invoicing contexts are handling the status changes of an order, and bound with a foreignkey to the order class. The notification context is also a new addition which only handles general events happening in the application, it acts like an portal between all the context and users. Therefore we a visual direct link from notification context to the user context. The stripe context is the last new addition to the block diagram and has a strong connection with ordering/invoicing for paying tip or order cost.

2.2 Class Diagram

C# is an object oriented language and our design naturally became well structured and separeted by classes. The contexts were made as independent as possible and mostly react to events, we unfortunately use foreign services/repositories inside of some contexts to get related information, mainly since we only send ids through events. The user context is the most central and important context since a user is required in every context. The application wouldn't be functional without having a user context, theres room for improvement of indenpency, but at the end of the day will the user context always be difficult to replace.

The stripe context is unfortunately dependent on the foreign key in the product, cart item and order class, which could be resolved by creating a table mapping the product ids to the stripe product ids. The current solution simplfies the implementation of stripe and saves us from creating new products and prices for each checkout session.

2.3 Organization

Person in Charge; Martin
Throughout the development of the application it became more appeարent who had the most general knowledge with the code base and who assisted with the most code reviews, true to industry standards they were labeled Software Architects. Most of the organization was through Github's own tools such as Issues, Project and Actions. General talks and discussions were carried out on a discord server for the group members. Most of the channels were for discussing the different contexts but informative channels for posting resources and similar notes was made. We realized two key factors where the group communication was flawed during the project. One was that we didnt have a dedicated space with instructions on how to setup the project. This meant that when a new environment variable was added and not pushed to main alot of our developers struggled with solving the seemingly complicated error. This was solved by adding a 'dev-environments-notes' channel to collect all the info needed from downloading and then running the project flawlessly. The other problem was the long code review times, parts of the team was not aware they had been asked to code review, by adding another channel 'you-have-been-requested-to-review' this problem was quickly fixed, a simple channel where we tagged the member to code review and a link to the pull-request.

Github Issues was used to track all tasks that needed to be completed and bugs or problems arising when developing. Issues with lists were used to accurately track progress towards the project description that was currently implemented on main. Projects was used to give the group a visual representation of deadlines for each issue and also what stage each issue was currently in. Actions was more used as a control tool to make sure that the code passed all dotnet tests, and that the project was able to run the docker container and get an OK status code back when accessing the website, on all pull-requests and pushes.

We also held weekly meetings each thursday either in-person on campus or on discord where we discussed our individual progress and the architects summarized the total progress towards the final product. Through these meetings we also had discussions about high level desicions and some cooperative coding sessions.

3 Backend

3.1 Repository

Person in Charge; Joachim / "Martin"
Choosing to go with repositories over pipelines was in short terms a design decision to reduce the amount of reused code and generalize intercation with the database. Through the implementation of the generic database repository we as a group had a structured way to build database interactions that was not dependent on any specific table, centralized all those functions within one singular file which improved the efficiency of locating certain database functions.

3.2 UrlProvider

Person in Charge; Joachim
The redirect to page method in asp.net is easily used by providing a url which is determined by where the page is in the "Pages" folder. We started with just a few hardcorded strings which escalated further into alot more. This caused an issue when we wanted to sort the pages into folders, manually going through each hard coded was a tedious task and while doing it I realized that we can create a url provider.

3.3 3rd Party Login

Person in Charge; Martin
We decided to implement two options for 3rd party authentications, Google and Facebook [1]. Early in development a slight error was made leading to the full convertion towards HTTPS to accomodate for this functionality. Later it was discovered that running HTTPS was not strictly necessary. Running the application on HTTPS also caused some issues with the certificate when creating and running it from a docker container. In reality this is only a problem since we cannot get an issued trusted certificate for this project and with the requirement for a very limited setup, it was simply not feasable to authorize a dotnet certificate. This is why we opted to run the application on HTTP instead. This did however seem to cause some issues on the safari browser, but given the scope of this project, that safari did work with HTTPS and that for any official buissness deployment one would use HTTPS with a valid certificate, we did not explore this issue further.

3.4 Integration Tests

Person in Charge; Joachim
We planned on implementing multiple integration tests to confirm the functionality of our

application, but regrettably could'nt carry through with our goal. We have a few integratoin tests covering simple get and post requests which is succesful, these tests especially saved us while we developed the application. The mocking of the authentication in identity made us face a challenging issue, and we therefore could'nt create complete workflow tests for the customer, courier and admin. The unit tests on the other hand covers most of the functionality which we have around 350 of.

3.5 Stripe

The library[4] stripe is mainly used for the checkout session, and this session requires products, we therefore integrated the whole process of creating and maintaning products with stripe. We have two main handlers create and edit which reacts to changes in the product repository in the product context.

The checkout session consist of multiple types of payments which is of course in a testing environment where you can simulate failure and success. We handle the result of the transaction by updating the invoice depending on the checkout session's outcome.

The transaction through a session contains both a success and fail return url which we discoverd had a vulnerability since we reveal the order id when a user views an order. The user could then attack with "Parameter Tampering", which means that the user can easily recreate the return url and bypass the payment of tipping. The invoice id is never revealed to the user, but having an extra guid or unique string for each session increases the difficulty of recreating the return url.

3.6 SignalR

SignalR[3] is used to send real time notifications to users. We have implemented one main method for alerting a user, when for example the status of their order is updated. We encountered an issue with reloading and alerting the user simultaneously since signalR only operate per http request, and reloading the page would override the alert functionality. We solved this by storing the alerts in the database, then after forcefully reloading the user's page, the OnConnected method in signalR is executed and runs the notify user method for each notifications related to given user.

4 Frontend

4.1 Courier

Person in Charge; Martin
For the courier dashboard we went with a three split view separated into partials with:
OrdersOverview, OrderHistory and Earnings. OrdersOverview displays a view over all active and placed orders, OrderHistory shows a list of every order handled by any given courier. Earnings shows the total earned salary calculated based on delivery fee cut and tips. Through

a pie chart we show a visual to the courier displaying their earnings split into delivery fee cut and tips, this gives the courier a deeper insight into their earnings profile. The earnings view is also able to be filtered for any months since the couriers first registered order.

4.2 Canteen

Person in Charge; Anna
For the canteen, we implemented a split-view interface. The left side displays the food items available at the university canteen. When a user clicks on an item, its details are dynamically displayed on the right, along with controls to adjust the item count. We utilized JavaScript to manage the item count on the client side, storing the value until the user adds the item to the cart. To enhance the user experience we used AJAX to load item details, eliminating the need for full page reloads when viewing different items. Additionally, we disabled the “Add to Cart” button when the item count is set to 0. This simple solution was a quick and easy fix to a problem that would otherwise require more code and validation.

4.3 Orders

Person in Charge; Anna
The orders page provides users with a view of their order history. Orders are sorted by date, with the newest orders being displayed first, making them quick to locate. Each order includes details such as the order Id, total price, and payment status. Buttons for actions like "Finish Order", "Invoice", "Cancel Order," and "Tip Courier" are dynamically rendered based on backend logic, ensuring users can only perform valid actions for a given order. An example of this is the "Finish Order" button which appears for incomplete orders and is replaced with "Invoice" once the order is completed/placed. It is essentially the same button, with different text being rendered. The button routes users to the same ordering page, where content is dynamically rendered based on the state of the individual order.

New orders are marked with a specific status “New” and saved, allowing users to return and complete them later. This feature provides flexibility, ensuring users don’t lose their order if they want to finish it later. The payment status indicator was added after we realized that its absence made the interface less intuitive. This addition improved transparency and usability, enabling users to quickly understand their order status.

5 Contexts

5.1 User

Person in Charge; Joachim / "Martin"
The user class plays a crucial role in the application and is highly integrated with the Identity framework[2]. The user can have three different roles: customer, courier and admin, we debated removing the customer role since its almost self-explanatory that user who registers

would be a customer, but quickly realized we needed to keep the role to separate the admin from the customer role. Both the user service and repository is implemented with signin- and user manager which eliminates the need to implement external login, authorization and other complex functionality. Therefore is the user the only entity in our codebase which does not inherit from base entity.

5.2 Admin

Person in Charge; Jesus & João
While the Admin class extends from the User class, we decided to keep all the logic related to administrators separate from that of other types of users. Within this context, we have two distinct repositories.

Additionally, we implemented an Admin Service that consolidates the core functionalities, such as handling admin logins, creating admin accounts, accepting courier requests, and more. One of the most critical components in this context is the Configuration Manager. This class is responsible for modifying the JSON file where the delivery fee is defined.

Why did we choose a JSON file instead of a database table? This was a challenging decision. Ultimately, we concluded that there were very few configuration values to store —essentially just the delivery fee— making a dedicated database table somewhat wasteful, as it would only contain a single row. However, this approach is not without its drawbacks. Using a database table would have offered greater security and integrity, as the configuration values would be protected within a controlled schema. This trade-off highlights why this decision was particularly difficult.

Finally, the system includes Events and Handlers, which enable the admin context to share and receive information from other contexts. There are two specific handlers: one for courier request submissions and another for admin invitation responses. Additionally, several events notify other contexts when actions such as accepting courier requests and posting admin invitations.

5.3 Product & Cart

Person in Charge; Anna
Both the product and the cart contexts were initially implemented similarly to what we had in the labs, using individual pipeline files. However, after some code reviews and discussions, we replaced all the pipelines with repositories.

Admittedly, there were some disagreements and misunderstandings around this choice in the beginning. Some group members were more experience programers than others, which allowed them to plan ahead and visualize the code a bit better. Less experienced members originally wanted to write code similar to the labs, as it felt more familiar and safe. However, as the project progressed, we collectively agreed that using repositories provided a faster and more manageable solution.

A solution we did keep from the labs however, was combining session and database to handle cart. We used session to store the cartId for fast and efficient access, free from excessive database queries. The actual cart data is stored in the database, providing data integrity and concurrent access if needed. This solution creates a clear separation of concern between our database and a temporary user session.

5.4 Order

Person in Charge; Martin
Ordering was a pretty straight forward context handling the creation of an order taking in an a shipping address and invoice address and send these values on when placing the order. As with the other implementations the ordering context uses generic repositories for streamlined database interactions. We decided to implement functionality directly in the order class to dynamically calculate the different TotalCosts to have them accomodate the Tip value that was not being set at creation but rather later after the order had been delivered.

5.5 Invoicing

Person in Charge; Joachim
The invoice context handles the event of order status changed. An Invoice is created when an order is placed, and updated when the order is either canceled or the user completes a stripe checkout session for paying an invoice. An invoice will be credited if its canceled before the order is picked by a courier and is updated to just the delivery fee when a user cancels an order which has status picked.

5.6 Fulfillment

Person in Charge; Joachim
Fulfillment also handles the event of order status changed. It tracks wether an order can be refunded or not, which is determined by both the payment and the status of an order. The offer is only refunded if the user has paid the invoice, and this is done by handling the event of the related invoice being paid. The offer will then be refunded if the order is either canceled or goes missing.

5.7 Notification

Person in Charge; Joachim
The notification context consist of the email service, standard notification and alert notifications. SignalR is used to alert the user in real time, while we also add notifications to the database which persist until the database is deleted. The implementation of the notification context has made notifying the user simple, and makes the task of adding a new notification straightforward. The context's handlers primarly use the different tools like alert, add notification to database and send email to alert users. The notification context can viewed as checkpoint for messages transfered from the different contexts.

6 Summary and Contributions

6.1 Summary of the project

The final project ended up including most requirements both normal and advanced features. There were some problems and issues along the way but these were addressed accordingly.

Several of the projects Advanced requirements were implemented including: Self-hosted passwords, 3rd party logins, Sending emails at certain events, Testing environments for Payment through Stripe, and Push-Notifications.

There was some slight problems that surfaced both related to implementation but also related to communication within the group, these problems were handled through teamwork and setting up structured ways to communicate certain important information. All in all these issues were quickly resolved once they revealed themselves.

6.2 Contributions

Distrobtion of work for each person in the group, was given in table 6.1 .

Name	**Role**	**Tasks**
João	Developer	Admin
Jesus	Developer	Admin
Joachim	Architect	Full-stack, Advanced-Requirments
Anna	Designer & Developer	Cart, Product, Full-Stack
Martin	Architect	Full-Stack, Docker, Organization

Table 6.1: The work distrobution

References

[1] O. Source. Facebook, google, and external provider authentication in asp.net core. *ASP.NET Core*, 09 2024.

[2] O. source. Introduction to identity on asp.net core. *Microsoft Learn Documentation*, 2024.

[3] O. Source. Overview of asp.net core signalr. *Microsoft Learn Documentation*, 2024.

[4] Stripe. Stripe api reference. *Stripe Docs*, 2024.