UNIVERSITY OF STAVANGER, NORWAY

DAT240 Project Report (Group VyGoHuErOegKn): CampusEats

Simeon Vyizigiro, Filip B. Gotten, Hammad Hussain,
Torjus Knudsen, Rasmus Øglænd, Atle Ericson

November 19, 2022

CONTENTS

1 Introduction

In this project report we will talk about how we came up with certain designs, how we worked on dividing up the project, what technologies we used and how we communicated with each other while working on the project.
We will also discuss different struggles we had both with the technological aspect and with the communication as well as how we overcame them and improved overall.
We had already "started" this project in the Labs for the course, so we had some idea as to where we wanted to start, and what technologies we wanted to use.
The project however, was quite a bit more complex than the labs with the addition of users, user roles, a way to pay for the product, a way to become a courier and deliver the product and lastly the use of Docker as a means to build and run the application.

We also had to focus a lot on communication and working as a team. The project required the use of Github Actions for the CI. Since we were already using Github Actions, we decided to use the Github Issues for setting up issues and task for the team to pick and choose what to do. We will talk more about this in later sections of the report.

2 Organizing work

When we first started our project, we went through every section of the project requirements together. This was done intentionally so that everyone started on the same page. The importance of doing so, was that no one would misunderstand any of the important concepts and then get off-track as we got deeper into the project.
We started off trying to make every part of the foundation we got familiar in the labs from scratch, where everyone would work together on creating every small part of the base of the project. After some days of doing this, and realizing we just ended up with something very similar to what we already had from the labs, we decided to instead transfer and convert the labs foundation directly to fit our new application.
Throughout the project, we often split the tasks at hand between pairs of two. As we were a group of six members, everyone could get a partner and then they would work together with specific domains to both get a deeper understanding in that area.
Another benefit of the pair-programming was that no one alone would have the sole responsibility on big topics, with the result being we rarely had a situation where someone couldn't make it to school and then no one would have the knowledge needed for those specific parts of the code.
Using pair-Programming also allowed us to divide big tasks into smaller ones, and doing things one step at a time. We created GitHub issues and assigned them to the correct pair. After a while we stopped using GitHub issues, and instead took our assignments from Miro, as we saw that as a better way to distribute tasks.
In the last few weeks we had a specific meeting where we compared our finished functionality with the needed functionality from the task, and we realized that we had been focusing a lot more on some parts than others. This is when we decided to split the groups of two into

focusing each one on the different contexts; customer, courier and admin. This also resulted in using GitHub issues less, and instead deciding what next to do based on the sub tasks within the contexts on Miro more.

Link to Miro: `miro.com/app/board/uXjVPLNO9vo=/?share_link_id=31381295491`

2.1 The first meetings

We decided to use Miro as a way to showcase the plan and our work methods, as well as extra information regarding the project. Here we displayed what we wanted to achieve, how we wanted to achieve these goals as well as the coding structure we wanted to pursue. Miro was easy to use, user friendly as well as convenient for our project.

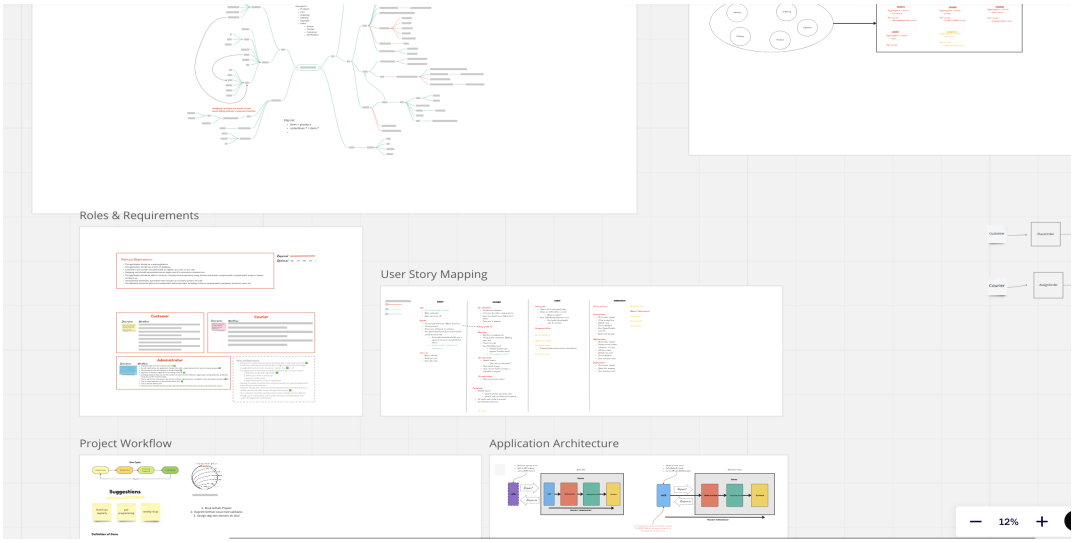

Figure 2.1: Here you can see the full map of Miro for our project.

The picture above gives a closer look at our project overview in Miro, where we decided to set up the workflow. We tried to include everything that was related to the project: All the tasks, an overview of the project, how we wanted to work, as well as the logic behind how we were going to do things.

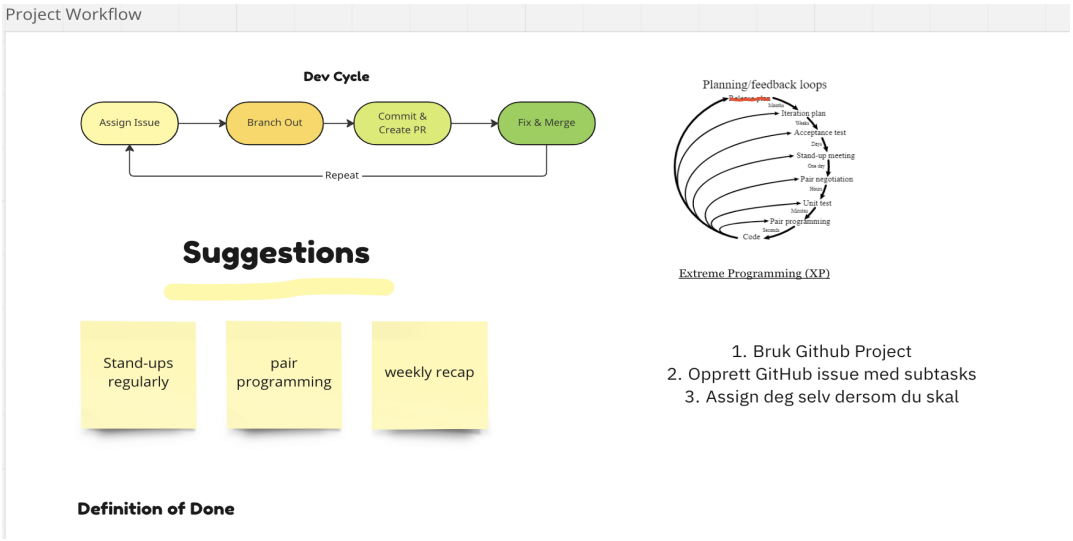

Figure 2.2: This is a description of how we planned to work each week with the project.

We also decided to use GitHub issues to make the flow of making tasks, assigning yourself or another person on the team. Committing code in small chunks and then creating a pull request. After that a pull request was created, where everyone in the group would review and give feedback. After the necessary changes were made, the pull request would be merged into main. Branching out, adding new code and pull requesting was a strategy that was used throughout the whole timeline of us working with the project.

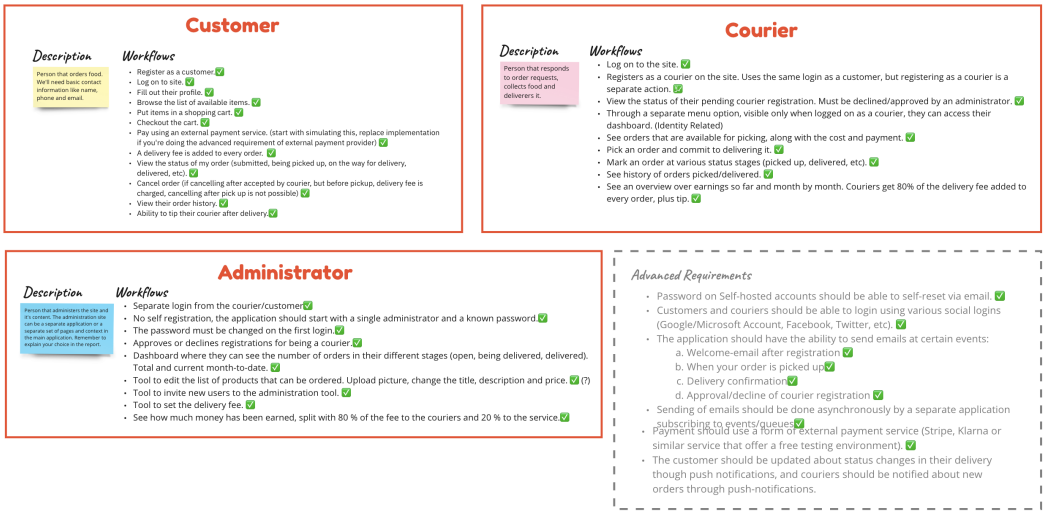

Figure 2.3: This is what we started using when we stopped with GitHub issues. The points marked with a check mark, are the tasks that are currently fulfilled.

2.2 Design

At the beginning of the project we tried to map out the domains within our problem space. As we didn't have any domain experts to refer to, the team itself became the domain experts. Our ubiquitous language revolved around how the project requirements were formulated. Since the project was very similar to lab 3, a lot of our design choices were influenced by the lab.

By looking at the requirements we decided that the core problem the application ought to solve was for customers to be able to order products from the campus cafeteria and have it delivered. We can divide the problem into three main components:

- Customers who can order a product
- Someone who can deliver the order
- An inventory of products that can be ordered

Therefore, we decided that the problem space could be split into three core domains: Products, Ordering and Delivering. But the requirements also specified some important functionalities such as administering the products, paying for orders and more. By looking at the requirements we identified some supporting and generic subdomains.

The end result resulted in a domain model that was split up into the following subdomains:

- Admin
- Cart
- Delivering
- Identity
- Notifications
- Ordering
- Products

Among these subdomains we consider Identity, Cart and Notifications as generic subdomains. These subdomains all provide functionality needed in order to solve the core problem, but they are not special to our business. Payment is also a generic subdomain within our problem space, but we decided to outsource this subdomain by utilizing Stripe's services. And lastly we consider the Admin subdomain as a supporting subdomain as it is directly tied to our problem space, however it is not part of the main components of the core problem that we are trying to solve.

Whilst structuring the project solution, we mainly took inspiration from the lecture Todo application. Our main goal was to create a maintainable code-base with loosely coupled bounded contexts. We also focused on encapsulating business critical behavior to achieve consistency boundaries around them e.g the order status should only be updated in incremental steps or a delivery can only be assigned a courier once.

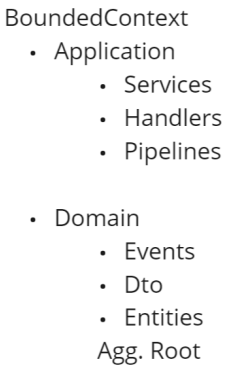

Figure 2.4: This template displays how we tried to structure our code.

The team agreed upon a solution structure used to structure each context. We did this in order to make the codebase easier to navigate such that everyone had a common understanding of how to structure each context. The structure itself was also chosen to make it easier to conform to the domain driven design principles. This made it easier, but the codebase did diverge from time to time and had to be pulled back to it's ideal form.

2.3 How we could improve

Organizing the work in our group went very well. At the start we separated into pairs to do pair-programming where each pair focused on one part of the project. One pair was working on Customer, one pair on Admin and the last pair on the Courier part. We also helped each other out where we could, so if one pair or a person was feeling stuck they got the help they needed. As mentioned earlier, using GitHub issues went out of date for us, as we found more effective ways of distributing tasks. Using 2.3, the "issues" were already made and set up, so we could self-assign from there. This was done with good communication between the different pairs.

In order to improve, we believe that we could check in with each others work more often, as we noticed that there were a couple of times when someone weren't clear on what they had to do next. In the figure 2.3 there are certain tasks in one role context, that were tasks that required that certain tasks from other roles would have been completed. For example, to make a page for total earnings as admin, you would first have to have added a delivery fee added to an order. Therefore, the ones who were working on admin, would have to wait on the ones working on orders to add the delivery fee methods. To solve this would require better planning, communication and collaboration between the different pairs. What we did, was that one of the pairs would have to wait for the other pair to implement something, in order to implement something themselves. This was not optimal at all. We should have been better at noticing the overlapping tasks, and found a way to combine our knowledge on the different contexts to solve the task. This is something that should have been made clear early on during the planning process.

We also should have been better at planning the solution of different tasks earlier in the project. By this we mean a pseudo-code solution to most of the tasks, so that we knew what we had to implement in order to have it all make sense. To optimize this, we think that if each pair would go through their given context, and discuss together how they would plan to solve each given task, would result in code that would be easy to modify the different tasks if they would need to work together with tasks from other contexts.

3 Communication

The communication in the group was good. Everyone from the group knew each other before the project started and we have worked on different projects together before.
During the weekdays we usually worked together at campus. This made it possible to discuss problems while working, and gave everyone an idea of what was getting done.

Whenever we faced any challenges, our go-to method in finding a solution would be to ask other members of the group. This was the most effective way in resolving a problem, because one would spare all the internet researching time. If no-one in the group knew a technology or a solution to a problem, then researching on the internet would be done.

Our experience in regards to the communication was very positive. We had good synergy all the way around. Everyone contributed well in the group and were focused on achieving the best possible result. We always kept out head high when faced with challenges, and always sought to find the best solutions to every problem. We made sure that everyone knew what to do next to avoid someone being left with nothing to do.

3.1 Stand-up

We decided early on to have stand-up meetings at least once a week, sometimes more, to make sure people were getting things done and have a place to discuss.
We started each stand-up going around and telling what we had done since last time and if we had any problems during this time.
Then we went over what still needed to be done and prioritized what tasks to work on until next stand-up.
This was a good method to see what the others were working with and also see how much work we had left.

3.2 Discord

We already had a Discord server for our group before the project started, and continued to use this throughout the project. If someone was working from home, they could simply ask a question in the chat and other members of the team would reply as soon as they had the chance to. We also had some evenings where we everyone joined the voice chat and shared our screen if we had problems. This all worked out very well.

4 Technological decisions, problems and how we fixed them

Our web application consists of a static web page that accesses a database. The application is entirely implemented in C# by the use of the .NET Core 6 framework. This choice was made as this was the technology that the entire team was familiar with.

4.1 Database

We chose to use a Postgres database which we deployed using docker. This was chosen because of the database's current popularity and reputable reputation. Lastly we figured it would be more robust and easier to implement in the end when we built our own application as a docker container. The different services could then all talk together within their shared docker network.

In order to access and manage the database we used an object-relational mapper (ORM) Entity Framework Core (EF Core) as our . This choice was natural as EF Core is the ORM created by Microsoft themselves. The documentation and support for EF Core is therefore very extensive.

By using an ORM we could architect our project in a Code-First approach. This meant that we could create our desired domain model without being restricted by the database model.

EF Core simplifies the data access logic, but because of it's different details, it sometimes made it difficult to understand and debug why some errors occurred. Especially working with database migrations proved to be rather difficult. Both managing and understanding how all the generated migrations files worked was challenging.

4.2 Docker

As we previously stated we used docker on multiple occasions and for different use cases throughout the project. We got the database Postgres and the application adminer up and running straight away with adminer being a database access web tool where we could easily view the database data and even edit certain values and tables. This made it easy to verify that our entities and our functions using those entities worked as expected.
Having our database as a docker container also meant it was quite easy to destroy the database information and start from scratch with new migrations and new tables. This was quite handy in the beginning when there were constant updates making migration updates hard to do. After having our tables and database properly configured, it still was handy for everyone to have their own separate version of the database, which docker made easy to do.
At the end when we were ready to create our own docker container built from our project, we also included the use of nginx as our reverse proxy to serve HTTPS instead of using the dotnet kestral integrated https service together with a development certificate.

4.3 ASP.NET Core Identity

For our authentication and authorization needs, we chose to use the Identity package from ASP.NET core as our base because it was recommended in the dat240 discord and it seemed to be quite popular and often recommended.
What followed was a lot of documentation reading and trial and error to get it setup correctly the first time, but after some time we cracked the code and figured out the basics on how it would work together with our goals. After getting it working it made the adding of new authentication features easier with the use of its predefined tables, pages and built in functionality.

Throughout the project however it did cause some issues where we wanted to do one thing, but Identity insisted on only doing it its own way. This was for example the case where we wanted to add more fields to the IdentityUser class, which could only be done by creating a new AppUser class which inherited from IdentityUser and telling our IdentityContext to refer to the AppUser class instead of the built in IdentityUser class. We also had to specifiy the addition of some more database columns in the model builder to refer to our new AppUser fields.

Another issue we had to overcome was the introduction of our nginx reverse proxy together with the oauth2 external authentication, which we used to implement Google and Github login. The issue was that during the direct to Google or Github to authenticate our self and the following callback, somehow we lost the https information in our header, and we got a redirect match error.
When inspecting the url calls, and the headers we found that callback information in the url was carrying a http link instead of a https one. We found out this was because our dotnet application was oblivious to the nginx reverse proxy and its https conversion, so it thought it was only running http.
To fix this, we had to spend a lot of time debugging and searching the internet for answers. Finally we added some more fields to our program.cs file and some nginx.conf configuration lines, then it all started working again.

4.4 MediatR

A common technique to achieve loosely coupled code is by inversion of control (IoC) through a message broker. MediatR became our choice as the message broker in our code, and was therefore important for us to be able to follow the DDD principles. This choice was made because MediatR provides a simple API for handling requests as well as publishing and subscribing to domain events.

MediatR is also widely used in the dotnet community meaning there is a lot of documentation for the library, an important factor as it was the first time working with a message broker for the team. MediatR was something everyone was somewhat familiar with from the lectures and labs, which made the cooperation between different group members who may have worked on different contexts easier.

By using a message broker it not only allowed for more decoupled domains, but also more

isolated application logic which in turn provides better mobility of the code. Each MediatR request, often 1:1 relation to HTTP request, gets it's own handler which made the code and it's dependecies easily discoverable.

Even though the concept of using a message broker to achieve inversion of control it sometimes lead to some confusion of when certain pieces of code were executed. But as time went on and the cooperation between team members became better the understanding of the application control flow became clearer for everyone.

4.5 External Services

In our domain model we have some generic domains, one for sending out notifications and one for handling the payment of an order. In the project we have chosen to use external services for both of these domains. We have chosen to use the SendGrid and Stripe SDKs, respectively, for the two generic domains.

4.5.1 SendGrid

At certain events our application sends email to the customer notifying them with status updates about their order and when request are approved or declined etc. After exploring different technologies we chose the email API provided by SendGrid. SendGrids software as a service (SaaS) is responsible for sending the emails asynchronously to the costumers. Implementing the API was not complicated and it works well.

4.5.2 Stripe

As our external payment service we chose to use Stripe, which offers a free testing environment. The stripe testing mode makes it possible to simulate real life transactions. The reason for using Stripe rather than Klarna was that Stripe had much better integration with .NET using the Stripe.net library as opposed to the deprecated Klarna .NET Core SDK.

When using stripe, a Stripe session is created at our end with the desired details. This session contains a URL that can be used to redirect to the Stripe checkout. One of the difficulties we faced was how we should store an order. We could of course store the order as the session was created, but that would mean that the order would be stored in our database before the payment was received.
To solve this we created an endpoint where Stripe would redirect to if the payment succeeded with a URL containing the Stripe session-id. This endpoint used the OnGet method to emit an event saying that the payment was successful. The session-id could then be used to extract all the data needed about the transaction for us to store the order.

4.6 Testing Frameworks

In our project we ended up down prioritizing tests, but we were able to create some tests. We focused primarily on integration tests as we felt that they provided the best insight into what was working and not. To perform our integration tests we use multiple testing libraries. The most prominent ones were:

- xUnit
- Testcontainers
- FluentAssertions

FluentAssertions was chosen to create more human-readable assertions in our tests. We used xUnit as the main testing framework to execute all our tests. The Testcontainers library was used to create testing databases with Docker. By using Testcontainers we could use Postgres, the same database provider as in our application, which gives us more reliable integration tests. The decision for choosing these two libraries were made in order to follow a technique of using a single database instance when testing inspired by Nick Chapsas' video.

Additionally we used Moq to mock some of the services in the DI container, such as the CurrentUser service.

5 Summary

This has been a difficult and stressful, but educational and fun project. First of all we have developed a better understanding of how software development should be done, and how important good communication is. Secondly, through this project we have followed the principles and patterns of Domain Driven Design when designing the software. This particular design is used a lot in real life work environments, so it was necessary for us to learn it. Understanding domain driven design was hard for us at the beginning, but as we progressed, the understanding and knowledge surrounding it increased. We can honestly say that we are proud of these last few weeks of hard work and the learning experience we got.