Synopsis

Systems Integration Exam

Christoffer Drejer Mikkelsen

Cph-cm370@cphbusiness.dk

Introduction

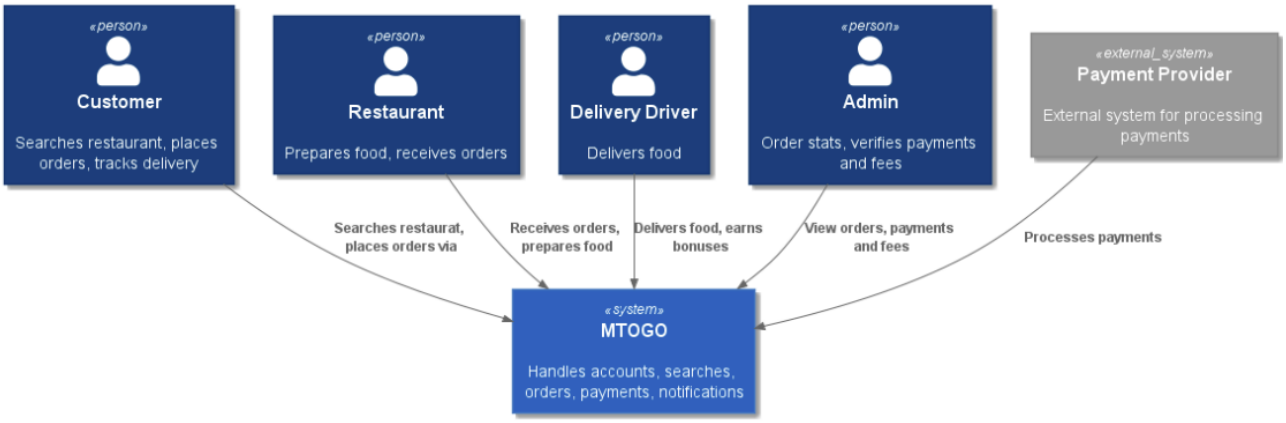
I have created a food delivery application MTOGO where customers can search for restaurants and place orders for food. Delivery drivers can create an account on the app and will be notified when an order is ready to be picked up at a restaurant so that it can be delivered. Restaurants can create an account as well, but it is not required to be able to receive orders. There's also an admin page where management can handle payments, fees and get an overview of orders.

During this project I have used an agile approach to the development. Using GitHub Projects I have planned out the sprints starting with the first iteration being focused on planning the project, event-storming, creating diagrams and user stories. During the second iteration I found out that some of the plans I had made had to change and expected this was going to happen which I why I chose agile since it focuses on responding to change over following a plan. I have also used practices for DevOps such CI/CD (Continuous Integration/Continuous Deployment) through GitHub Actions to automate building and testing the project whenever changes were pushed to GitHub.

Technology Stack

Layer	Tools
Version Control	Git, GitHub
IDE	Visual Studio Code
Programming Language	C#
Database	mssql
CI/CD	GitHub Actions
Testing	XUnit
Api	REST
Messaging	RabbitMQ
Project management	GitHub Projects

This is the technology stack I chose to use for this project. Even though Visual Studio is considered better for C# and .NET I have been using VS Code a lot and there's plenty of extensions that can help you in VS Code.



C4 System Context Diagram. Can be found in C4 folder on GitHub.

This is the system context diagram for the application showing the four types of users that will interact with the application customers, restaurants, delivery drivers and admins. Customers can search for restaurants and place orders. Restaurant will receive notifications when an order has been placed. Delivery driver will receive a notification when an order I ready to be picked up and delivered to a customer. Admins can view all the orders to make sure there are no problems like long delivery times and handle paying delivery drivers and sending fees to restaurants.

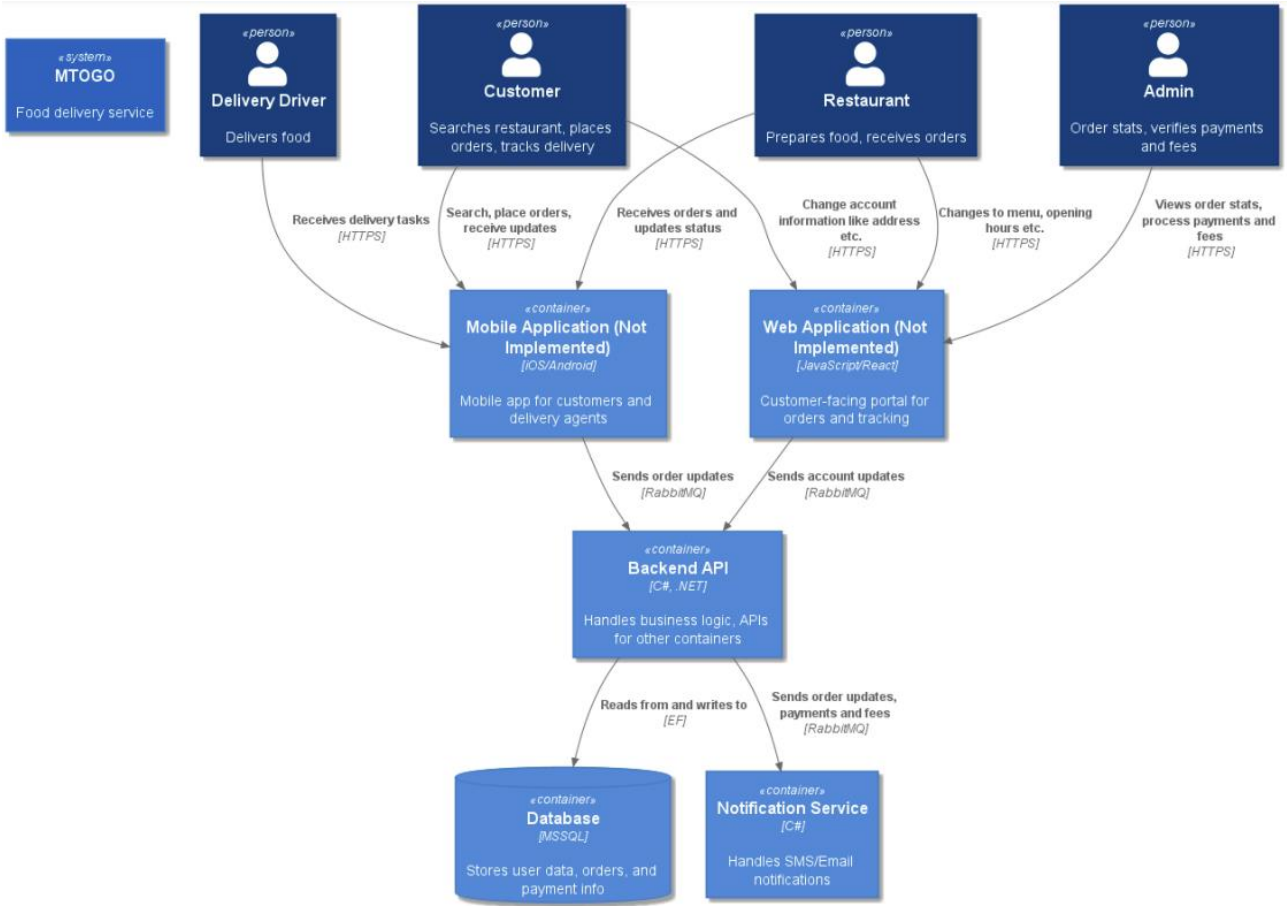
Architecture

Working with Domain-Driven Design makes a lot of sense when it comes to building microservices. Figuring out the bounded contexts for a system makes it easy to then create a microservice for the bounded context. I didn't quite end up with the services I created being the same as the bounded contexts that I found during the event storming process (Event storming diagrams can be found in Event Storming folder on GitHub). During coding I realized that some of the services I had planned on creating like a delivery service weren't needed and could just be combined with the order service. I also wanted to create a restaurant management service where the menu could be edited, and you could calculate the restaurant fee based on orders but instead that became a part of the account service and management service. Domain-Driven Design is open to change, and iterative collaboration is an important part. Software requirements change and the domain can change and therefore it is important to be open to this change and ensure that the system can adapt to new business needs. By using microservices it is easy to add new features like handling complaints.

Not all parts of the system benefit from Domain-Driven Design. A simple service for creating user accounts with simple CRUD operations doesn't benefit much from fully modeling the domain. It is also a complex design and might lead to problems with over-segmenting the domain into too many contexts. Another part of Domain-Driven Design that I haven't really

been able to take advantage of during this project is ubiquitous language which helps ensure that developers, domain experts and stakeholders have a shared language to try and avoid miscommunication.

Event storming is a great way to figure out the domain events that happen in a system and plan out the timeline of these events. This way you find out when you will need to interact with the database and when you need to send notifications. The bounded contexts work like a kind of guide to help you figure out which microservices you need and what they should be able to do and which external systems they should interact with. One of the key principles of event storming is the collaboration between software developers and domain experts during the process. While I couldn't do this due to working alone on this project, I still found event storming incredibly valuable. It helped me explore the domain, clarify complex processes, and design a better system by uncovering critical events and their relationships.



C4 Container Diagram. Can be found in C4 folder on GitHub.

The C4 container diagram zooms in compared to the system context diagram and shows which part of the application the different users will be communicating with. The delivery driver will receive deliveries through the mobile application as well as email and SMS. They will also receive their paycheck through email and SMS notifications which will show things like total deliveries completed and amount. Customers will be able to search for restaurants and place orders through the mobile app and they can use the web app to change their

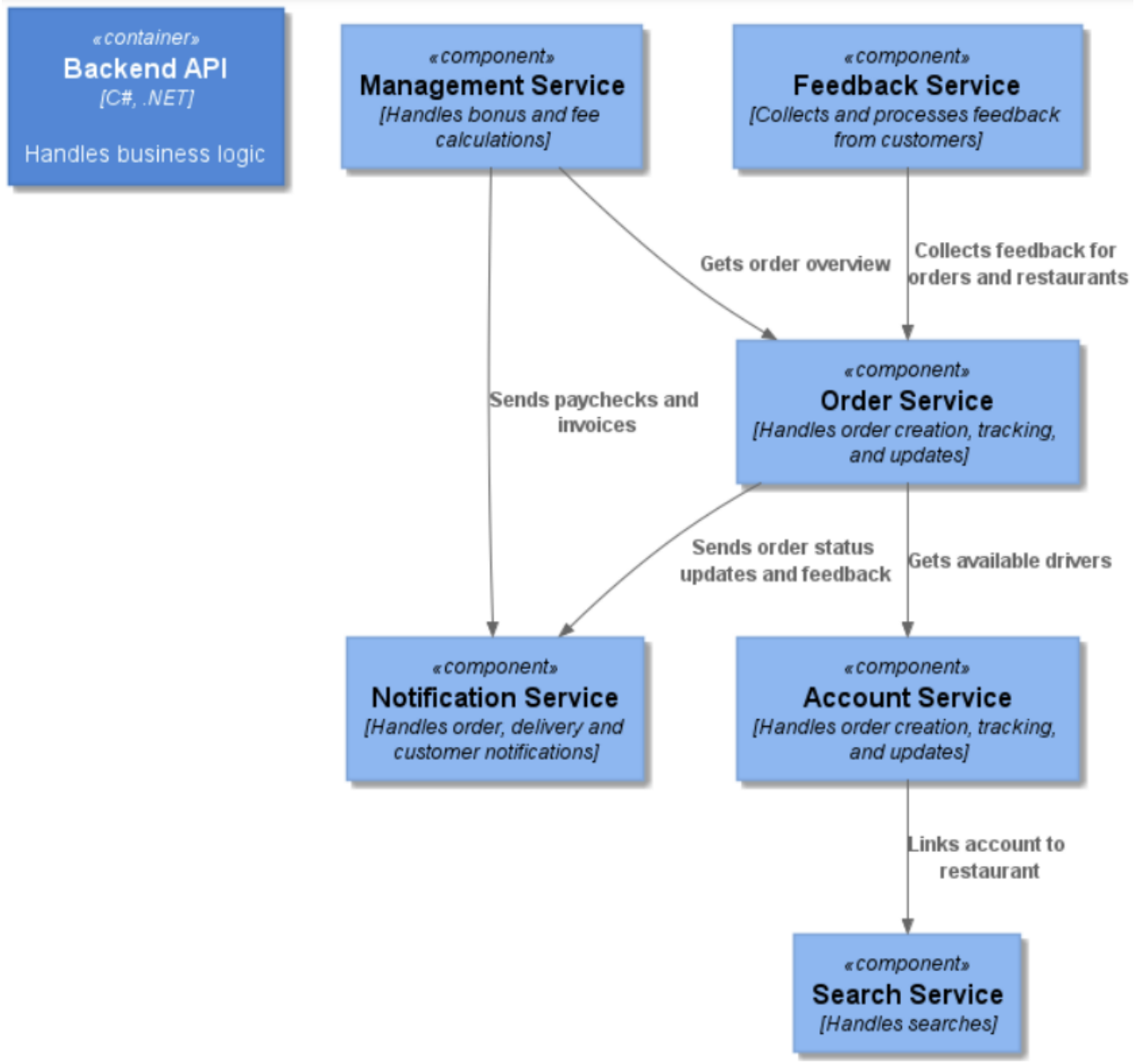
account information like address and phone number. They will receive notifications via email and SMS. Restaurants can receive orders and updates through email and SMS notifications. They can create an account on the web app to change their information like opening hours and menu. Admins will have access to order stats and be able to process payments and fees. This will all be saved in the individual databases for each service.

Code

When going from the planning phase to creating the services I used the event storming diagram to figure out the services I wanted to create. I started with the search service since I felt this was an important feature that was needed, and I didn't need the account service to work. It also wasn't the most complicated or biggest service which meant I could setup GitHub actions and create some tests so that I had an idea of how it would work for the other services. My domain model helped when it came to creating the databases for the different services. I could use it to figure out what sort of data I needed for orders, restaurant and feedback etc. I often looked back on the event storming diagram when I started working on a new service to make sure I included all the different domains and events.

Legacy applications are applications that are no longer being actively worked on but just being maintained so that they can keep running. Monolithic applications are single centralized units which means that when you have to test or run the system you have to run the entire application. This means that adding new features or making changes can be very time-consuming. Most legacy applications are built as monoliths and often businesses are relying on these system for important parts of their workflows. This is where the Strangler Fig pattern can be used. This pattern explains how to replace small parts of the legacy monolithic application with new services. The business can focus on replacing the most important parts of the legacy system and slowly replace the entire system with a microservice application.

Microservices are independently deployable services usually with their own database. The service can be developed, deployed and tested by itself and it makes deploying and testing easier and faster compared to monoliths. It also means that the one service failing doesn't break the entire system. It does also make it more complex and can be harder to plan the project. When the different services need some data form another service that service first has to get the data from the database and then send it to the other service which is an extra step compared to monoliths. Even though it might be complex to plan and create a microservice once a monolith becomes big enough it becomes very complex and there's a lot of parts and steps to go through when it comes to debugging large monoliths. This is where microservices work the best for large systems that need to be open to new features and changes. There is also the advantage of being able to use services for other systems. If you built a notification service that can send out emails this might be something that other systems, the business is on working also needs.



C4 Component Diagram. Can be found in C4 folder on GitHub.

This is a component diagram of all my services and how they interact with each other. The notification service sends out notifications for order status updates, notifies restaurants when an order has been placed and notifies delivery drivers that the restaurant has finished the order, and it is ready to be delivered. It also sends out paychecks to delivery drivers and fees to restaurants.

It does this using a publish subscribe messaging pattern. When starting the notification service, it listens to incoming message and when the order service or management service publish a message it receives that message and sends out the email or SMS (mocked) to the restaurants, customer or drivers depending on the message published.

When the services communicate, this is done using a request response messaging pattern. The service sends a post, get or some sort of http request and they then get a response with the information they need.

```
2 references
public async Task<List<Account>> GetAvailableDrivers()
{
    return await _context.Account
        .Where(a => a.AccountType == AccountType.DeliveryDriver && a.Status == "Available")
        .ToListAsync();
}
```

In this function we get a list of all accounts with the type Delivery Driver who are currently set as available from the database.

```
4 references
public async Task<int> GetAvailableDriverWithLongestWaitTime()
{
    var drivers = await _accountRepository.GetAvailableDrivers();

    if (drivers == null || !drivers.Any())
    {
        throw new InvalidOperationException("Delivery driver list empty");
    }

    var driverWithLongestWait = drivers
        .OrderBy(driver => driver.StatusChanged)
        .FirstOrDefault();

    if (driverWithLongestWait == null)
    {
        throw new InvalidOperationException("No driver found");
    }

    return driverWithLongestWait.AccountID;
}
```

We call the repository function to get the list of available delivery drivers and then we find the driver who has the earliest status changed datetime (the one who has waited the longest). We then return this so that we can call this function when we need to attach a delivery driver to an order.

```
5 references
public async Task<List<Restaurant>> SearchRestaurantAsync(string? name = null, string? address = null,
string? category = null)
{
    IQueryable<Restaurant> query = _context.Restaurant
        .Include(r => r.Menu)
        .Include(r => r.Categories);

    if (!string.IsNullOrEmpty(name))
    {
        query = query.Where(r => r.Name.Contains(name));
    }

    if (!string.IsNullOrEmpty(address))
    {
        query = query.Where(r => r.Address.Contains(address));
    }

    if (!string.IsNullOrEmpty(category))
    {
        query = query.Where(r => r.Categories.Any(c => c.Category.Contains(category)));
    }

    return await query.ToListAsync();
}
```

This function is the search function when a customer needs to find a restaurant. A customer can search for a name, address and/or category. You can search for any combination of the three options or none (will return all restaurant if no search values specified so that a customer can just browse all available restaurants). Because I use the C# Contains method you can search for partial strings like if you search for the name “king” it would return “Burger King” and other restaurants with king in the name. The search function returns all information on the restaurant like opening hours, menu and categories.

Working with microservices makes a lot of sense when it comes to services like the notification service. Being able to handle all the logic of sending notifications in one service works well. For other services like order and account it would be easier to have it as one service for a project of this size. For other services like the order service and account service having to make get request to the account service to get customer information for an order is just an extra step. This makes functions that need data from multiple services quite complex and it can be easy to make mistakes or overlook something.

Conclusion

The results of this project align well with the original objectives. Going from planning to development did have a few problems and I realized that some of the things I had planned weren’t needed and that I didn’t need as many services as I had planned. Using microservices architecture definitely led to some extra work when it came to some of the services, but it made a lot of sense for the notification service. Microservice architecture has enhanced the overall flexibility and maintainability of the system. Working with microservices on this project has been a valuable experience, offering significant insights into their advantages.