System Integration Exam Synopsis

By Kristofer, Mads, Michael and Søren

Introduction

As part of our exam projects, we were given a single work assignment document which provides a scenario about a food ordering service which we were then tasked to create an application for.
We had about six weeks for the System Integration part, where in we spent the first two weeks preparing with documentation and planning.
We then spent the remaining four weeks coding the application by creating two projects, a frontend and a backend, which we then saved in a GitHub repository, as well as preparing the required delivery material for the exam.

We are tasked with creating an application that meets the requirements of the scenario of MTOGO (Meals TO GO), as well as various requirements specific to System Integration, through the creation of a high fidelity prototype wherein we have both a frontend and a backend.

We started by doing event storming, first for the customers flow and then MTOGO management flow, restaurant employee flow and lastly the delivery employee flow.
We would then use these event storming sessions to create our user stories which then helped us create our architecture.

Architecture

Event Storming:

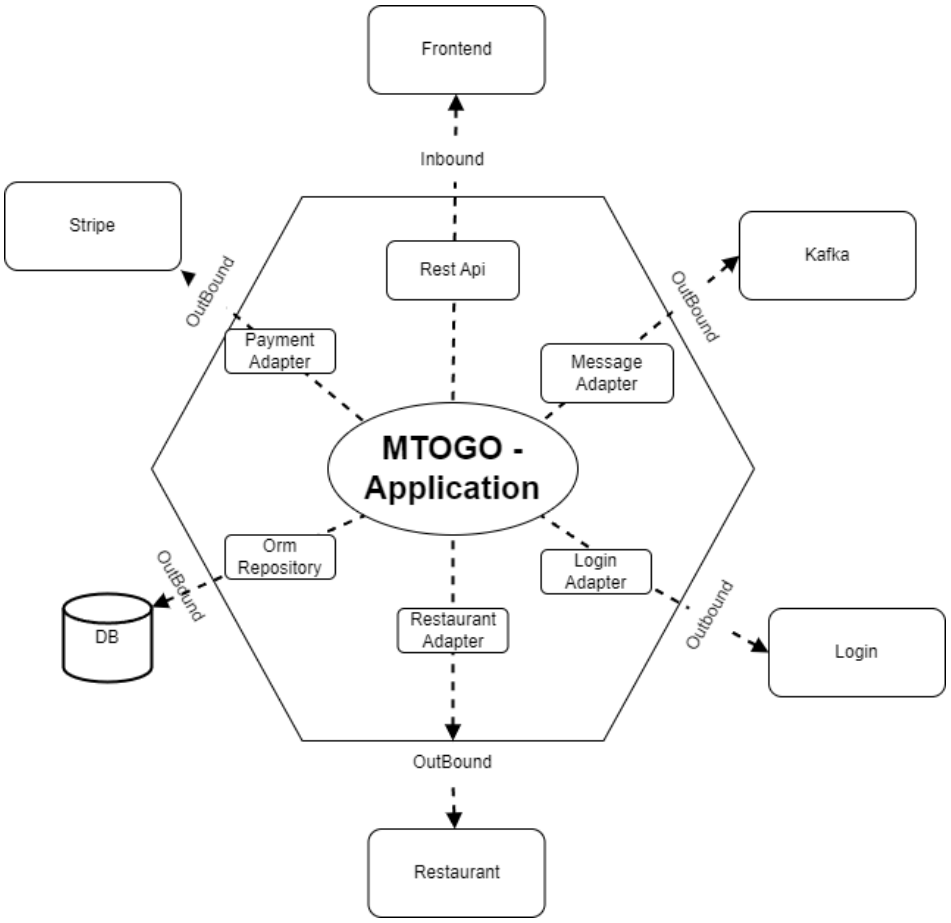
Before we started coding or making the diagrams, the event storming sessions provided a more clear baseline which, alongside the user stories, made it clear what kind of services we would need to make and what information would be needed at each step.
The main advantage we gained by doing the event storming sessions was a more clearly defined project by taking the scenario and separating it into different user roles and steps, with the steps being something we could connect to one or more user story that clearly defined the acceptance criterias and therefore defined what we would need to make.
Also by following the events from start to end could we easily see our progress for completing the application. With the time constraints we were at first unsure if we could fully implement all of the services so we choses to focus on the customer flow and admin flow which would at least meet the requirements of the work assignment.

DDD:

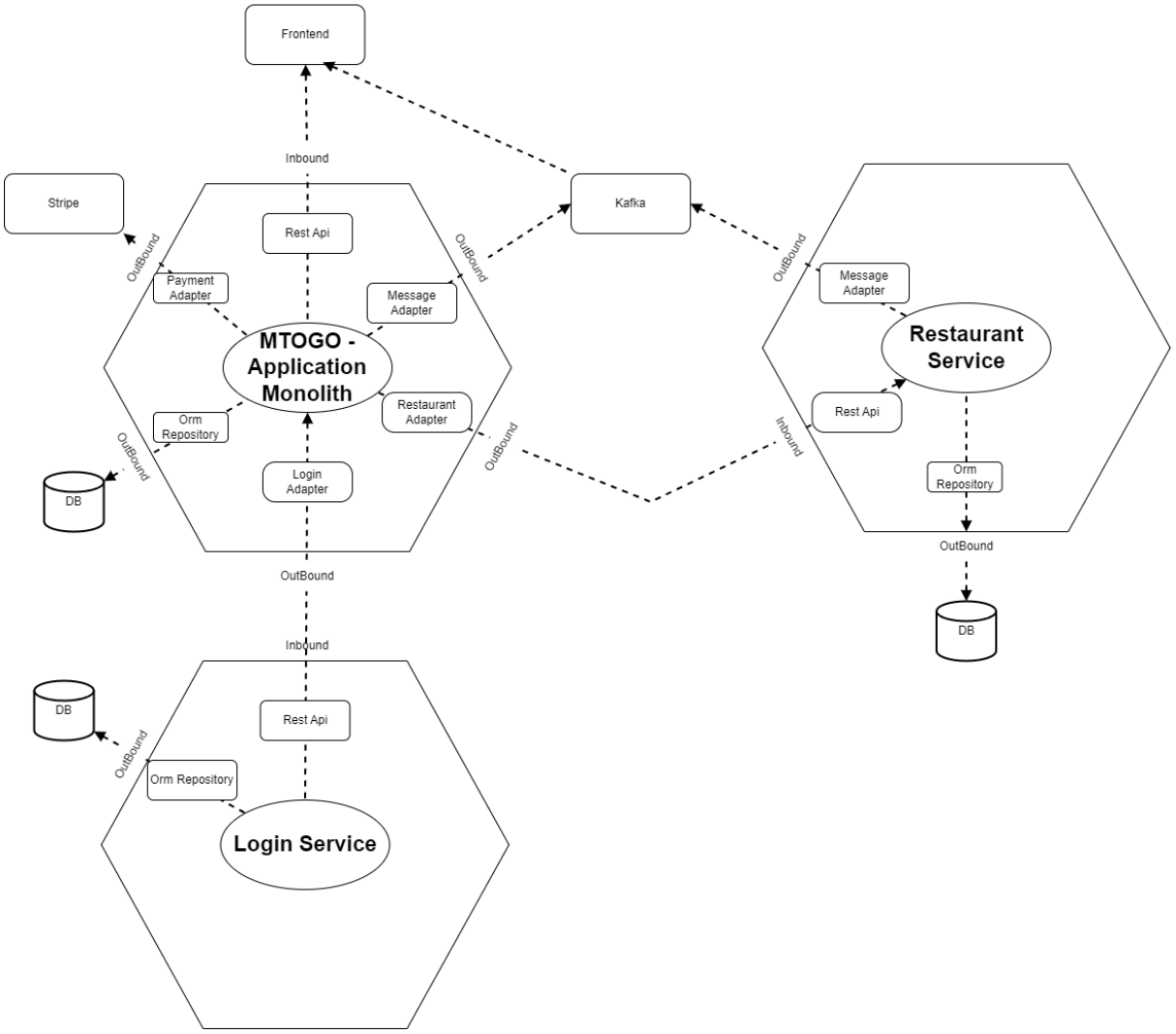
The main advantage of using Domain Driven Development is that we would be able to take our large system and divide it into smaller bounded contexts which would then have their own individual model.
With DDD focusing on using the business language instead of more technical language, we can more easily bridge the gap between business and product. As well it would also make the system diagram more readable to everybody as technical text can be hard to follow even for developers when viewing diagrams.
It does have negative aspects as when developing you cannot look at the diagram and expect every possible scenario to be covered. When developing you will most likely encounter something that would warrant a change to the diagram, but during development we don’t always have the time to do so, which would leave the diagram to be outdated. It also has the negative aspect that it doesn’t show value types which can create conflicts.
.

Architecture Diagram

To illustrate our architecture we have drawn hexagonal architecture diagrams.

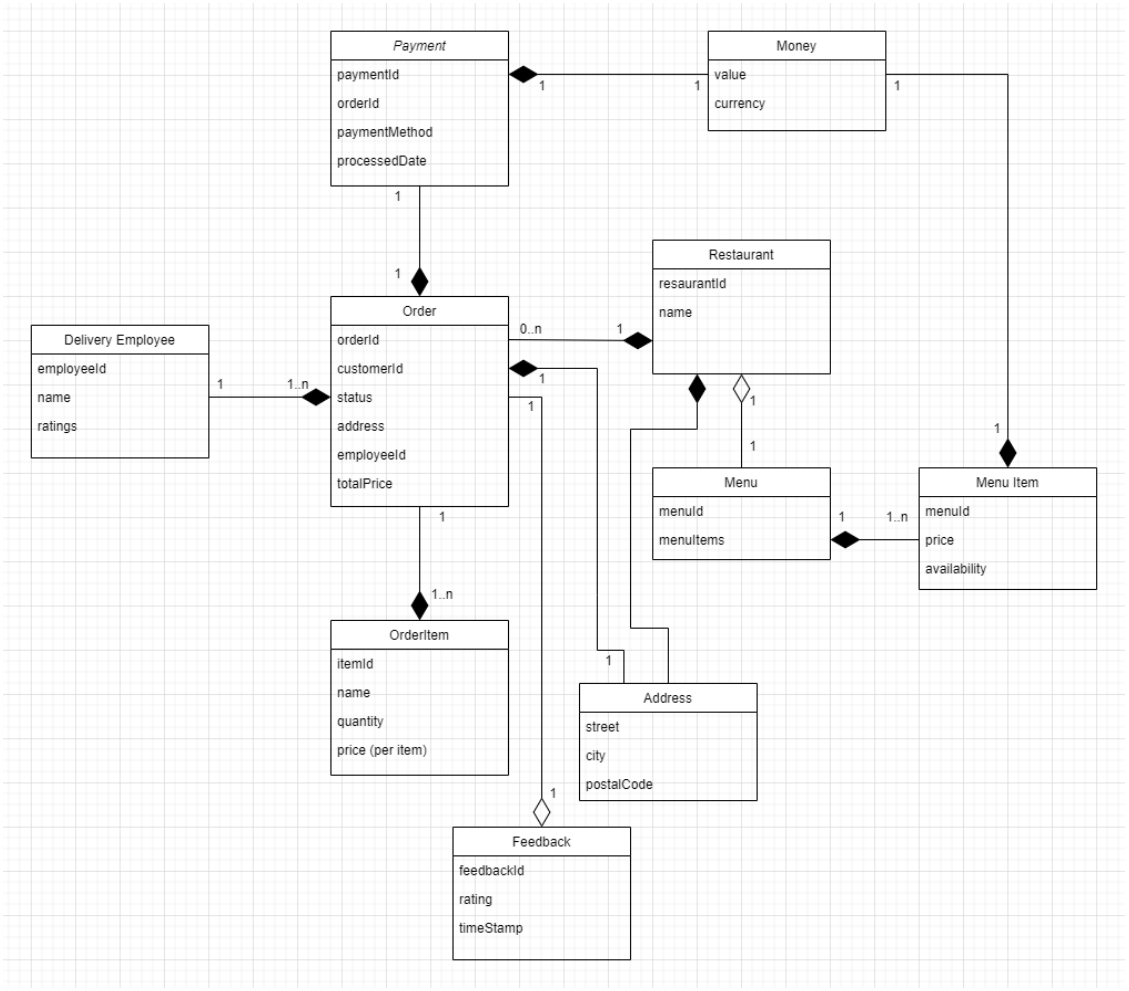


With our current instance of our application, we are using a monolithic architecture diagram, the architecture represented by the diagram above. We have a core application (monolithic code base) that interacts with different services. To come to production faster we are implementing some prebuilt services. To interact with each service, an adapter pattern was implemented. For our database we use mongoDB and use an ORM repository.

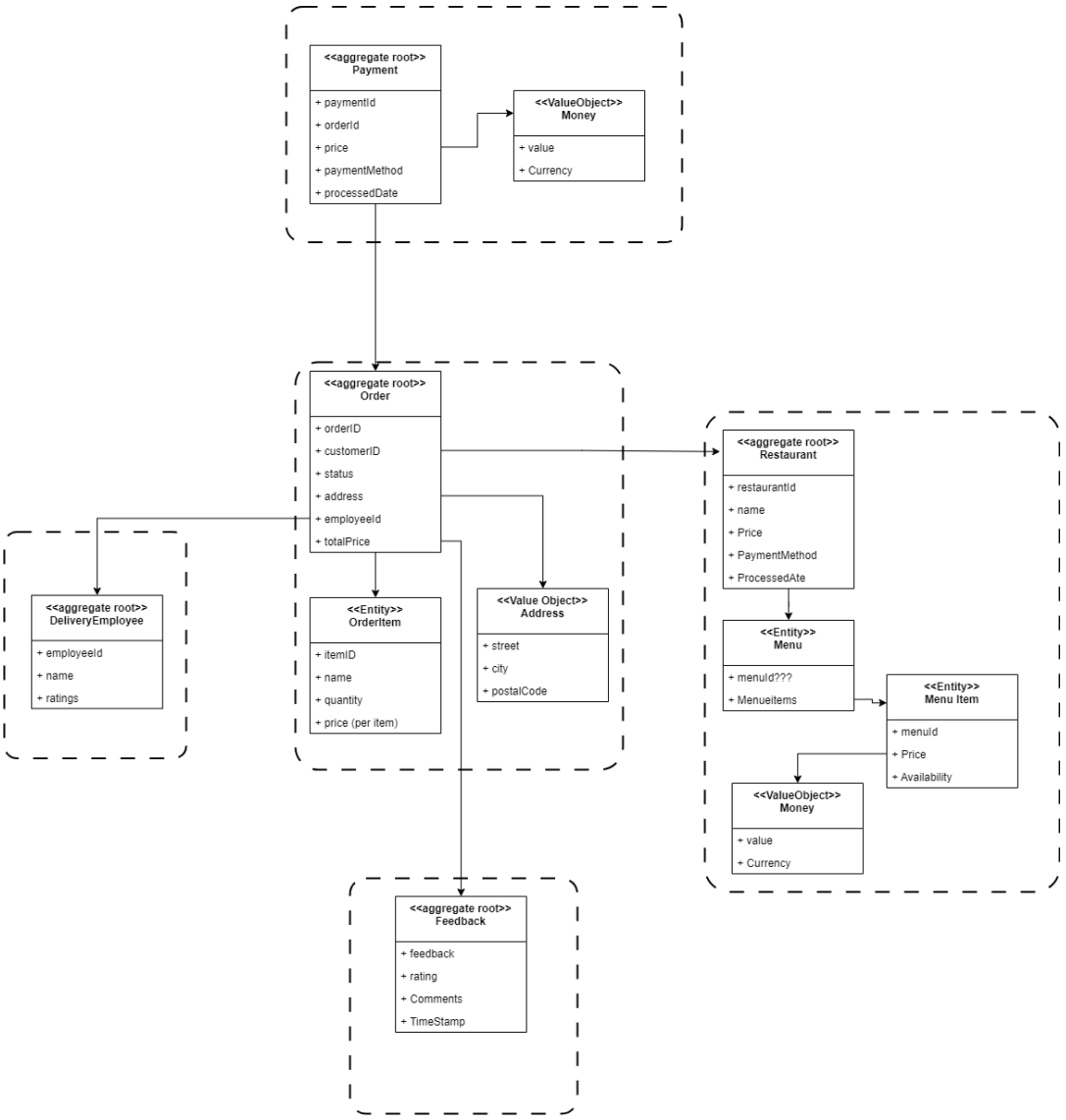


In this setup, we plan on completely decoupling the Login Service and Restaurant service, deploying each service as their own service. Each service will have their own dedicated Database and will interact. To simulate this architecture in code, we have attempted to decouple each service in our code to represent this.

Domain Diagram:



Domain diagram made before coding started.



Updated diagram based on changes made during coding.

We started by creating different entities including what information they should include. This helped us to determine which services would handle what and also what kind of information we would need to provide each service to function.
By creating the domain diagram above, we each could start on our own implementations without the need to constantly discuss content, but because we were more focused on implementing the individual services we also required multiple refactors due to value typing conflicts between our version control branches.

Code:

When we started to code we first took our hexagonal architecture diagram, which shows the different services, and used it to create overall implementation issues (tickets) for both the frontend and the backend. Some parts are also given their own issues due to size or complexity or requirements of other services.
We mostly started by implementing the backend services first as the implementation of the frontend was made easier by having the backend running locally.

Legacy monolith:

One part of the work assignment requirements was that we had “legacy” code as part of our application which meant to take several services, which would be separate under a microservice architecture, and make them into a monolith.
It was meant to mimic as if we are developing an application based on already existing code but because we of course were making the application from scratch meant that we didn’t have any already existing code.
We did this with the order, delivery and the feedback services. These services did not make use of an adapter which means we cannot easily change the logic without changing the core of our application. In this way the core application interacts with the different services.

Microservices:

Using adapter to both make a monolith microservice application and to show how we would handle separating each service into their own instance to run multiple versions of in a scalable way.
Because we also made considerations for scalability, we decided to have the adapters for the login and payment services run through localhost ports so the application could make use of multiple individual instances of a standalone service in a docker environment.
Our application currently only does it to showcase how we would handle it, and currently the application runs as a single instance.
Due to limited time and monetary resources have we chosen to only make use of a single database for all services. We chose to use MongoDB because we could run remotely through their own services for free.

Strangler Pattern:

Considering that we have legacy code in our project it is also worth mentioning how we would consider refactoring it into our microservice architecture. With the strangler pattern we aim to replace one part at a time to ensure that any changes don't have any negative effect on the rest of the system, mainly during production.
For our application nearly all events start through our API which means we could start by creating an adapter to whatever service we intend to make from the legacy code and have it still refer to the existing logic. With all of the events now going through the adapter to the existing code we can ensure that everything still works as intended, we then start by selecting one of the adapters calls and code a replacement within the new service logic. We

should then see over time that the adapter stops referring to the legacy code as we replace it with the new service.

Layered VS Microservices:

When considering using microservices architecture over a layered architecture, one of the biggest strengths of microservices is that each service is completely decoupled from each other and should be able to work even if all other services are down. This aligns with our requirements for having large amounts of users (traffic) and high Uptime. This also acts as separation of responsibility within the system. Each service is also able to be scaled individually which helps us manage traffic.

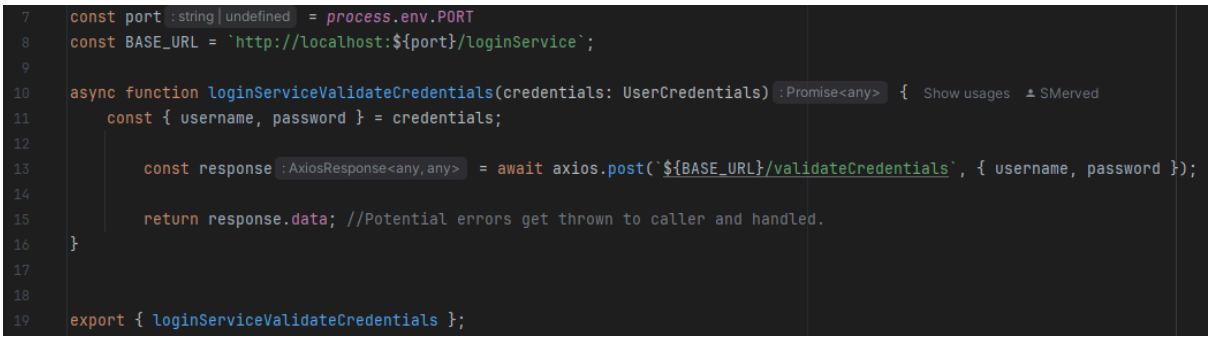
Because of the use of the adapter pattern when constructing an application using the microservices architecture also means we can easily replace a service, from mocking/development version to a production version, or if one service are dependent on a third party system that are then deprecated or completely remove then we can easily switch it without changing any of the core programing of the application.

Early in the development process, we chose Kafka as the message broker for our messaging service. Later, we encountered issues running it on our remote droplet server. Fortunately, using a microservice architecture allows us to easily adjust the service logic. Despite the challenges, we decided to keep Kafka due to time constraints.

When working with microservice architecture we rarely had any conflicts because what we were working on was completely separated from each other. And because we had already decided what information would be stored, where and how, meant we could start on nearly any implementation issue and complete it without any delay.

Quality:

For our application we originally wanted to create a microservice architecture by running individual instances of each service through clusters or Docker Swarm, but due to resource and time constraints we decided not to. We still wanted to show how we would handle this and so our adapters for the login service, restaurant service and payment service, all make calls via localhost. While our current solution does handle the calls from the same application, we could easily change the ports.



```
7   const port : string | undefined  = process.env.PORT
8   const BASE_URL = `http://localhost:${port}/loginService`;
9
10  async function loginServiceValidateCredentials(credentials: UserCredentials) : Promise<any>  {  Show usages  SMerved
11      const { username, password } = credentials;
12
13          const response : AxiosResponse<any, any>  = await axios.post(`${BASE_URL}/validateCredentials`, { username, password });
14
15          return response.data; //Potential errors get thrown to caller and handled.
16  }
17
18
19  export { loginServiceValidateCredentials };
```

LoginServiceAdapter

```
const port : string | undefined = process.env.PORT;
const BASE_URL = `http://localhost:${port}/paymentService`;

async function paymentServiceValidatePayment( Show usages  Mads Knudsen
    price: number,
    customerId: string,
    cardNumber: number
): Promise<boolean> {
    try {
        const response : AxiosResponse<any, any> = await axios.post(`${BASE_URL}/validatePayment`, {
            price,
            customerId,
            cardNumber,
        });

        return response.status == 200;
    } catch (error) {
        console.error(error);
        return false;
    }
}

export { paymentServiceValidatePayment };
```

PaymentServiceAdapter

```
const port : string | undefined = process.env.PORT
const BASE_URL = `http://localhost:${port}/restaurantService`;

async function restaurantServiceGetAllRestaurants() : Promise<AxiosResponse<any, any>> {    Show usages
        const response : AxiosResponse<any, any> = await axios.get(`${BASE_URL}/getAllRestaurants`);

        return response; //Potential errors get thrown to caller and handled.
}


export { restaurantServiceGetAllRestaurants };
```

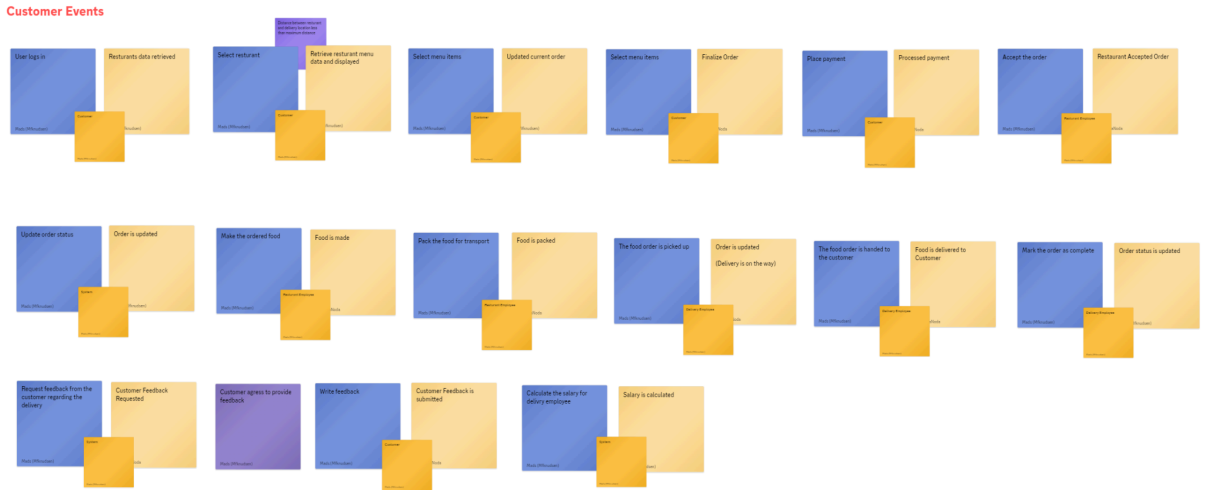
RestaurantServiceAdapter

Conclusion

Over the course of the project we have created event storming sessions based on the scenario provided to us and from there have we created user stories and diagrams to help us construct our application by laying a baseline for the services our application would need. We then developed our high fidelity application to run both locally and remotely using a microservice architecture but including a monolith legacy code. And while the project may not meet our own initial vision, that being having to cut having individual instances of our service due to resource and time constraint, do we believe that what we made is of good quality.

Images:

Event storming session based on the user experience for the Customer.



Event storming session based on the user experience for the MTOGO management.

Event storming session based on the user experience for the Restaurant Employee.

Event storming session based on the user experience for the Delivery Employee.