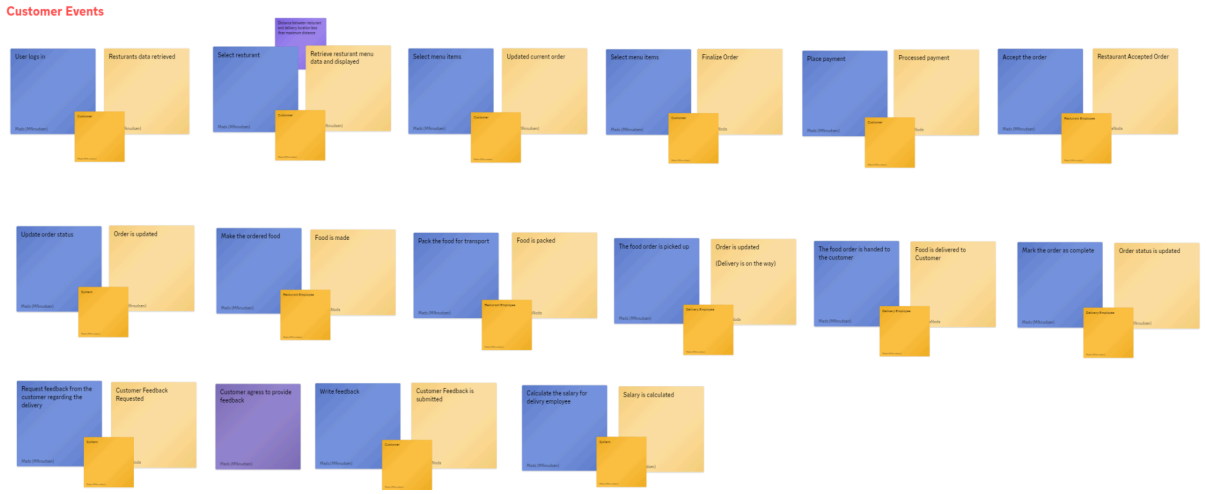
System Integration
Exam Document

By Kristofer, Mads, Michael and Søren

Event Storming:

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.

Scenario:

First we will go over the scenario and describe what we view as the requirements stated within. Then we will brainstorm the different roles that we believe need to interact differently with our application. When we have the different roles, what their requirements for their interactions are, will we be able to construct an initial idea for how we should make our application.

Overview, Description and Ambiguous Language:

Here we will go over the text scenario given to us and make a brief description about each part and how it could translate towards making our application. We will also go over any ambiguous language that we find doing this step.

From the description we can see the need for our customer to login before placing an order and from that we can also state that each role needs to be able to login and that their login needs to be assigned a role that provides them the correct flow.

The customer needs to be notified by the system when there is an update to their order. It is stated that the updates need to be either by SMS or via our application. It is not stated whether it is the application itself that needs to have the option to choose between the two options or if it is us, as the developers, that can decide to only implement one of the options.

The customer needs to be able to make a payment (mock) before the order is placed and sent to the restaurant for confirmation.

It is stated that management needs a dashboard of information regarding the current order statuses. Who management refers to is not clear and could both be referring to MTOGO management or to restaurant management. We would prefer to do both as we see both working and having a valid purpose for the application. However due to time constraints we will only be considering MTOGO to be management and develop based on that. If time allows, would we like to do both.

The delivery employee will need to be able to update an order when picking up the food order and when finalizing the delivery by handing over the food to the customer.

Our application also needs to be able to make room for new functionality. A complaint handling service, for both the customer and for the restaurant, is stated to not need to be implemented currently but that they would like to be able to implement it later. We see this as it needs the information from both restaurants, customers and orders. If this is all it needs then it would automatically be fulfilled by implementing the other services within our application.

We need to be able to make the application capable of maintaining growth via scalability and we are given a minimum and maximum for now and for the next five years. How we should accomplish this is not specified but it does infer that we need to code a way to handle a large number of messages to the backend. A message system would help ensure that request is

handled the right way but how many messages it would be capable of handling would also depend on the hardware of the server we deploy the instance on.

Brainstorm:

Within this section will we go over our initial ideas for our project and what we need to understand when we begin working on our application. Firstly we have, from our Overview and Description, determined the need to create a number of different roles that represent the different ways users will be interacting with our application. These roles will later on help us further understand how we need to develop the application.

Roles:

While all roles serve different purposes they all start at the same point by opening the application and logging in to their account which contains their role and determining how the frontend UI functions.

- **Customer:**

The customer is the primary role described within the scenario. They have the longest flow within our application.
After logging in they will be able to see their information, which they will not be able to change due to the information being a mock. They will also be able to select a restaurant from a list. The distance to each restaurant will be less than a maximum not currently set. When they have selected a restaurant will the UI change to only show the menu items from the restaurant in a list, and a list for what the current order contains, and a complete order button.
When clicking the complete order button will the UI change to give the user an option for payment and a way to complete the payment. When the payment is complete the user will be moved to view their orders.

- **Admin:**

Admin refers to the MTOGO management. When they have logged in they will be able to view the current status of all orders involving all customers and all restaurants.
The admins purpose is to have an overview and possible to be able to add or remove restaurants from the application, though are we thinking that adding and removing is not needed for a high fidelity application.

- **Restaurant Employee:**

When logged in, the restaurant employee will be able to see all orders and mark accept or reject any new order where after the orders will be updated to show the interaction.
The restaurant employee will not update the order further and any interactions between accepting or rejecting the food order is not part of our system and will therefore not be part of our application.

- **Delivery Employee:**

After logging in, the delivery employee will be able to select an order and change its status to be on the way. Later they will be able again to change the order's status to be complete

when they finish the delivery. They can also see the bonuses earned for each individual order when viewing the selection list.

Initial idea for Application construction:

We will go over how we can split up the application into different microservices based on our event storming session. From there we will have a better understanding of what data we need to store in a database. Then we can split the work between the frontend and the backend. For all of this will we also need to construct our reasoning for our decisions and how they can affect our project and our finished application.

Microservices:

Since our application will focus on microservice architecture we believe it would be best to create an initial overview of how we will build our application by defining the different services our application requires to pass the criterias set in the scenario.

- **Database Service:**

Firstly we have the database service which will handle the data. We initially wanted to have a separate database for the individual microservices but because of monetary constraints we decided to only have one running database for all the services.

- **Login Service/User Service**

Now when a user first navigates to our website they will first need to login. We originally considered the use of Google OAuth for a login microservice, but it, like most we could initially find, are all frontend oriented while we want our service to exist mostly within our backend.
Then there is the question about security and storage within the database. Therefore have we decided that we will forego the sign up option and have a static set of usernames and passwords for each of the individual roles we previously set up.
We feel that this will not impact our final product negatively as the application is considered a high fidelity application.

- **Restaurant Service:**

The restaurant service handles all functionality and information directly related to the restaurants. The information stored for an individual restaurant would be the name of the restaurant, its physical address, its entire menu (including any items that have ever been on the menu), and possibly a copy of all orders ever made.
While the orders are contained within their own service and the restaurants themselves would likely keep their own database of orders made, we should still consider the need to store a copy of the order without the user's personal information. We may need to store it in case that the order service is down and the restaurants will want the information regardless.

- **Order Service:**

The order service will handle the orders made by the customers and sent to the restaurants. The orders themselves will contain the customers user id, the restaurant id, a list of menu item ids, a payment method id, and a status on the order.

Because the list of menu items is never meant to change after it has been stored, then we can reduce the storage side by changing the menu list to an id to another list containing previously ordered items. This would reduce the amount of strings stored.

- **Employee Service:**

The employee service will be what the delivery employees will be making use of to check their bonus salary.. Again, like the restaurant service, should we consider the addition of storing a copy of the orders they handle in case the order service goes down.

- **Payment Service:**

We intend to make use of Swipe as part of our payment service which is meant to mock the action with different payment options.

Hardware:

For hardware will we most likely be creating a droplet using the services of DigitalOcean due to our previous experience with running a backend from a droplet using a Docker environment. This is the cheapest option that we can set up and that we know will reliably function as a server for our project.

Something to consider is how our current decision is made because of our current project scope and our own private resources, and that this would most likely be different were this to be an actual work project as part of a company. We should therefore also consider the possible hardware set up as if this were an actual product.
Firstly the restaurants would most likely have their own systems for on-site ordering that would be running before MTOGO would have a working product. In this sense we may be able to provide an API service between a restaurant's current systems and MTOGOs system. This could reduce the hardware requirements needed by the application specific to the restaurants.
In the same way could MTOGO have a separate system for their employees paychecks as it would be reasonable for that to not be part of the food ordering application. This would again reduce the hardware requirements needed to run the application.
This would of course mean an increase in code complexity within our application while reducing the overall cost of the project and application in a real world scenario.

But the project we are working on is not part of a real company and so we must work based on what we realistically can manage with our resources.

Database:

With the backend running from the cheapest option of droplet, means that we have a limited long term and short term memory space for running both the backend and the needed database(s).
With our group agreeing to not spend a large sum of money means the droplet will be taking most, if not all, of the money, and therefore have decided to select a database based primarily on the capability to run on its own from the providers own servers and that it would not cost anything.

Based on the project scenario considered we first the use of an sql based database, with MySQL being the one we are most familiar with, but because there isn't a provider that allows us to run it own their servers for free, means it's not the one we will be using for our project. We originally discussed using a sql database because of its structured setup that we feel would work well with the data of the project.

With MySQL not being a working option have we instead looked at a Document database, MongoDB, as a possible solution. MongoDB allows us to have a running instance that we can remotely connect to. It doesn't have the structured set up that we originally wanted but instead we could handle that through our backend services.
With a microservice architecture project setup, each service wherein we would need to store and/or read information, would it be best to have a separate database for each service to ensure that should one database not be working properly then it would not affect the other services which is the entire reason behind using a microservice architecture. But because of monetary constraints will we only be using a single database for the entire project.
However we still intend to code the project as if there is a separate database of each of the services that they all will access through the use of the database service.

Frontend:

With the frontend we intend to make it available to retrieve from our droplet using Nginx because of our familiarity to it and because we know it will work on the droplet. However we will most likely not be purchasing a domain address for it due to our monetary constraints, so if a user would like to retrieve the frontend from the droplet, they would require the ip address.

Because the project is more about the backend setup and planning of the project as a whole, will we not focus on making the frontend visually appealing and instead we will only focus on the minimum requirements for the project which is setting up the functionality so that it can fully work with the backend.

Backend:

With the backend we plan on it mostly consisting of the microservices. It will be a Rest API with a message system to handle inbound requests. One of the requirements of the work assignment is that we implement a monolith aspect to our backend, meaning that one of the microservices should be dissolved and be made part of the core of the backend.

User Stories:

User flow:

User story 1:
As a user, I want to be able to log into the system with my username and password so that I can access features based on my assigned role (Customer, restaurant employee, delivery employee, or admin).

Acceptance Criteria

- The user can enter a username and password on the login page.
- Upon submitting valid credentials, the system authenticates the user by verifying that they exist in the database.
- Once logged in, the user is directed to the appropriate interface based on their role:
    - Customer: Access to ordering and viewing menus.
    - Restaurant employee: Access to manage incoming orders.
    - Delivery employee: Access to view and manage delivery tasks.
    - Admin: Access to system-wide management features.
- If login credentials are invalid, the user receives an error message without any further details.
- If the user is not found in the database, they are informed that the credentials are incorrect.

Customer flow:

User story 1:
As a customer, I want to be able to see a list of available restaurants, so that I can choose where to order from.

Acceptance criteria:

1. The system should display a list of all available restaurants to the customer.
2. Each restaurant in the list should display the following information:
    - Name of the restaurant
    - Location
    - ...
3. The list of available restaurants should update in real-time based on availability.

User story 2:
As a customer, I want to be able to select a restaurant, so that I can see their information.

Acceptance criteria:

1. The customer should be able to click on a restaurant from the list of available restaurants to view its menu.
2. Once a restaurant is selected, the system should display:
    - The restaurant's name and location.
3. The customer should be able to navigate back to the list of available restaurants.

User story 3:
As a customer, I want to be able to see the full menu of a selected restaurant as a list of menu items, so that I can get an overview of what they have to offer.

Acceptance criteria:

1. The system should display the restaurant's full menu as a list of items.
2. Each menu item should include:
    - Name of the dish
    - Price
    - Customer reviews (?)

User story 4:
As a customer, I want to be able to add menu items to my order, so that I can put together my preferred order.

Acceptance criteria:

1. The customer should be able to click an "Add to menu" button, and add a menu item from the restaurant's menu to their order.
2. After adding an item, the system should update the order summary (showing total items and the current total price).
3. If an item is out of stock or unavailable, the system should notify the customer before they attempt to add it to the order. (?)

User story 5:
As a customer, I want to be able to see an order summary (showing total items and the current total price), so that I can review my selections before finalizing my order.

Acceptance Criteria:

1. The customer should be able to access the order summary at any point during the ordering process.
2. The order summary should display:
    - A list of all items in the order, including their name, quantity(?), and individual price.
    - The total number of items in the order. (?).
    - The current total price.
    - A dropdown menu for selecting the payment method (e.g., credit card, PayPal).
3. The customer should be able to edit their order from the summary by:
    - Changing the quantity of any item. (?)
    - Removing items from the order. (?)
4. The system should automatically update the total price whenever an item is added, removed, or edited. (?)

User story 6:
As a customer, I want to be able to checkout/pay for my order, so that I can complete the transaction.

Acceptance Criteria:

1. The customer should be able to pick a payment option from a dropdown menu.
2. Upon clicking the "Checkout/Pay" button, the system should:
    - Validate the payment information.
    - Process the payment automatically.
3. If the payment is successful, the system should display an order confirmation message.
4. If the payment is successful, the system should create and store the order.
5. If payment fails, the system should display an appropriate error message and allow the customer to correct any issues with their payment information.
6. The customer should have the option to save payment information for future orders (?).

User story 7:
As a customer, I want to be notified when my order is on the way, so that I can prepare to receive it.

Acceptance Criteria:

1. The system should send a notification to the customer when the order has been dispatched from the restaurant.
2. The notification should include:
    - Estimated delivery time. (?)
    - A brief message confirming that the order is on the way.
3. The notification should be delivered via push notification or SMS.
4. The system should log notification events for future reference and troubleshooting. (?)

User story 8:
As a customer, I want to be able to write feedback to the restaurant/delivery employee, so that I can share my experience and help others make informed decisions.

Acceptance Criteria:

1. The customer should have access to a feedback form after receiving their order.
2. The feedback form should include the following fields:
    - A rating system (1 to 5 stars) for overall experience.
    - A rating system (1 to 5 stars) for the food.
    - A rating system (1 to 5 stars) for the delivery worker.
3. The feedback for the delivery worker should default to 3 stars if the customer does not fill it out. (?)
4. The customer should be able to submit the feedback with a "Submit" button.
5. Upon submission, the system should display a confirmation message indicating that the feedback was successfully submitted.
6. The feedback should be linked to the specific order and visible to the restaurant or delivery employee for review.
7. The feedback should be aggregated and displayed on the restaurant's profile page to inform future customers. (?)

Restaurant employee flow:

User story 1:
As a restaurant employee, I want to be able to see a list of orders processed by my restaurant, so that I can efficiently track, manage, and fulfill customer orders.

Acceptance Criteria:

1. The restaurant employee can view a list of all orders processed by their restaurant in a clear, organized format.
2. Each order in the list displays key details, including:
    - Order ID
    - Customer name or identifier
    - Order items with quantities (?)
    - Order status (e.g., “Pending,” “In Preparation,” “Ready for Pickup/Delivery,” etc.) (?)
    - Timestamp for when the order was placed. (?)
3. The list updates in real-time to reflect any new or modified orders.
4. The restaurant employee can filter orders based on status (e.g., show only “Pending” or “Ready for Pickup”). (?)
5. Orders that have been completed or fulfilled are archived in a separate view for easy reference if needed. (?)

User story 2:
As a restaurant employee, I want to be able to select an order, so that I can see the specific order information.

Acceptance Criteria:

1. The restaurant employee can click on or select any order from the list of orders to view its details.
2. Upon selection, a detailed view of the order displays, showing:
    - Full list of items in the order with quantities. (?)
    - Customer name and contact information.
    - Delivery or pickup instructions and address.
    - Current status of the order (e.g., “In Preparation,” “Ready for Pickup”). (?)
    - Timestamp for when the order was placed and expected preparation time. (?)
3. The detailed order view includes an option to return to the main orders list. (?)
4. Any updates to the order (such as status changes or additional notes) are reflected in real-time on the detailed view. (?)

User story 3:
As a restaurant employee, I want to be able to accept or reject new orders, so that I can manage order flow and ensure that only valid orders are processed.

Acceptance Criteria:

1. The restaurant employee can view a list of new, unprocessed orders awaiting acceptance.

2. Each new order includes details such as:
    - Order ID
    - Customer name (or identifier)
    - Order items with quantities (?)
    - Delivery or pickup details
3. The restaurant employee can select an option to either **Accept** or **Reject** each new order.
4. **When an order is accepted:**
    - The order status updates to “Accepted” and moves to the list of active or in-progress orders.
    - The customer is automatically notified that their order has been accepted and is being prepared.
5. **When an order is rejected:**
    - The system prompts the employee to provide a reason for rejection. (?)
    - The order status updates to “Rejected,” and the customer receives a notification with the rejection reason, if provided.
6. The list of new orders updates in real-time, removing any orders that have been accepted or rejected.
7. A confirmation message or alert appears after each action to confirm that the order was successfully accepted or rejected. (?)

Technical Requirement:
When an order is accepted, the system should automatically send a notification to the customer, informing them that their order has been accepted and is now being prepared.

Technical Requirement:
When an order is rejected, the system should automatically send a notification to the customer, informing them that their order has been rejected. If a reason for rejection is provided, it should be included in the notification.

Delivery employee flow:

User story 1:
As a delivery employee, I want to be able to see a list of accepted orders, so that I can get an overview of available work.

Acceptance Criteria:

1. The delivery employee can view a list of all accepted orders.
2. Each order in the list displays essential details, including:
    - Order ID
    - Customer name or identifier
    - Delivery location
    - Order acceptance time (?)
    - Restaurant name/location (?)
3. The list only displays orders that have been accepted but not yet delivered or completed.
4. The list is updated in real-time or refreshes periodically to reflect any new orders. (?)

User story 2:
As a delivery employee, I want to be able to select an order, so that I can see the specific order information.

Acceptance Criteria:

1. The delivery employee can select any order from the list of accepted orders.
2. Upon selection, a detailed view of the order is displayed, showing:
    - Full customer name and contact details
    - Delivery address
    - List of items in the order
    - Order acceptance time (?)
    - Restaurant name/location (?)
3. The detailed order view includes a “back” or “close” option to return to the list of orders. (?)

User story 3:
As a delivery employee, I want to be able to accept/pick up an order, so that I can take responsibility for delivering it to the customer.

Acceptance Criteria:

1. The delivery employee can select an order from the list and choose an option to “Accept” or “Pick Up.”
2. Once accepted, the order status updates to “In Progress” or similar.
3. Accepted orders are moved from the list of available orders to the list of active orders assigned to the delivery employee. (?)
4. The system prevents other employees from picking up an order that has already been accepted by someone else.
5. A confirmation message or alert displays to confirm the order has been successfully accepted.

User story 4:
As a delivery employee, I want to be able to update the order status to “on its way”, so that the customer can be notified of its arrival.

Acceptance Criteria:

1. The delivery employee can select an accepted order assigned to them and see an option to update the status to “On Its Way.”
2. When the status is updated to “On Its Way,” the system automatically sends a notification to the customer with an estimated delivery time.
3. The updated status is reflected in real-time for both the customer and delivery employee views.
4. The “On Its Way” status can only be set after the order has been accepted or picked up.
5. A confirmation message or alert displays to confirm the status has been successfully updated.
6. The system logs the status update time for tracking purposes. (?)

Technical Requirement: When the order status is updated to ‘on its way,’ the system should automatically send a notification to the customer to inform them.

User story 5:
As a delivery employee, I want to be able to mark an order as complete, so that I can indicate the delivery is finished and the customer has received their order.

Acceptance Criteria:

1. The delivery employee can select an order that is currently “On Its Way” and see an option to mark it as “Complete.”
2. Once marked as complete, the order status updates to “Delivered” or “Completed” in the system.
3. The customer receives an automatic notification confirming the delivery as complete and requesting feedback.
4. Completed orders are moved from the list of active orders to a history or completed orders list. (?)
5. The option to mark as “Complete” is only available for orders that are in an “On Its Way” status.
6. A confirmation message or alert displays to confirm the order has been successfully marked as complete.
7. The system logs the completion time for record-keeping and tracking purposes. (?)

Technical Requirement: When the order status is updated to ‘Completed,’ the system should automatically send a feedback request to the customer.

User story 6:
As a delivery employee, I want my bonus salary to be calculated based on the relevant business rules, so that I can receive the correct salary for each delivery.

Acceptance Criteria

1. The system calculates a bonus for each delivery based on the order value, applying a set percentage or amount for each delivery completed.
2. The system applies an additional bonus for deliveries completed outside standard working hours (early morning or late evening), following predefined time ranges and bonus rates.
3. Bonus Based on Customer Reviews
    - For deliveries with positive customer reviews (above a specified rating threshold), the system adds a bonus to the delivery employee's pay.
    - The system does not add a bonus for ratings below the threshold or for unreviewed orders. (?)
4. Bonus Calculation and Display Accuracy
    - The system recalculates bonuses accurately whenever there are updates to relevant factors (e.g., new reviews or order adjustments).
    - The bonus amounts displayed to the delivery employee match the calculated totals for each period.

User story 7:
As a delivery employee, I want to be able to see my bonus salary for a completed order, so that I can understand my earnings for each delivery.

Acceptance Criteria:

1. The delivery employee can view a completed order and see the associated bonus salary or earnings displayed clearly.
2. The bonus salary information includes:
    - Base delivery bonus or earnings amount.
    - Any additional bonuses or adjustments (e.g., for rating or high-demand times).
3. The bonus salary is only displayed for orders that have been marked as “Complete.”
4. The information is accurate and matches the company's bonus policy and payment records.
5. The system ensures that the bonus salary displayed updates in real-time if there are adjustments after the order is marked complete. (?)
6. A history or list of completed orders with associated earnings is available for the delivery employee to review past bonuses and track earnings. (?)

Admin flow:

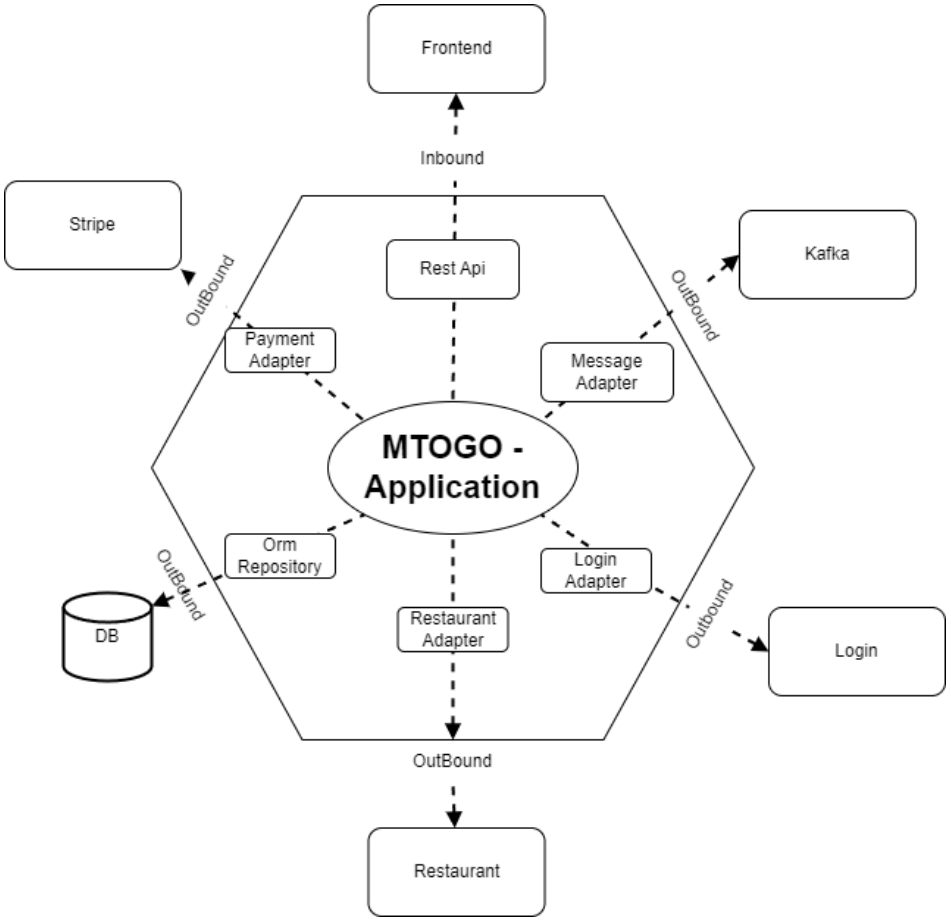
User story 1:
As an admin, I want to view a dashboard displaying key order metrics across the system so that I can monitor operational efficiency and track order processing trends across all restaurants.

Acceptance Criteria

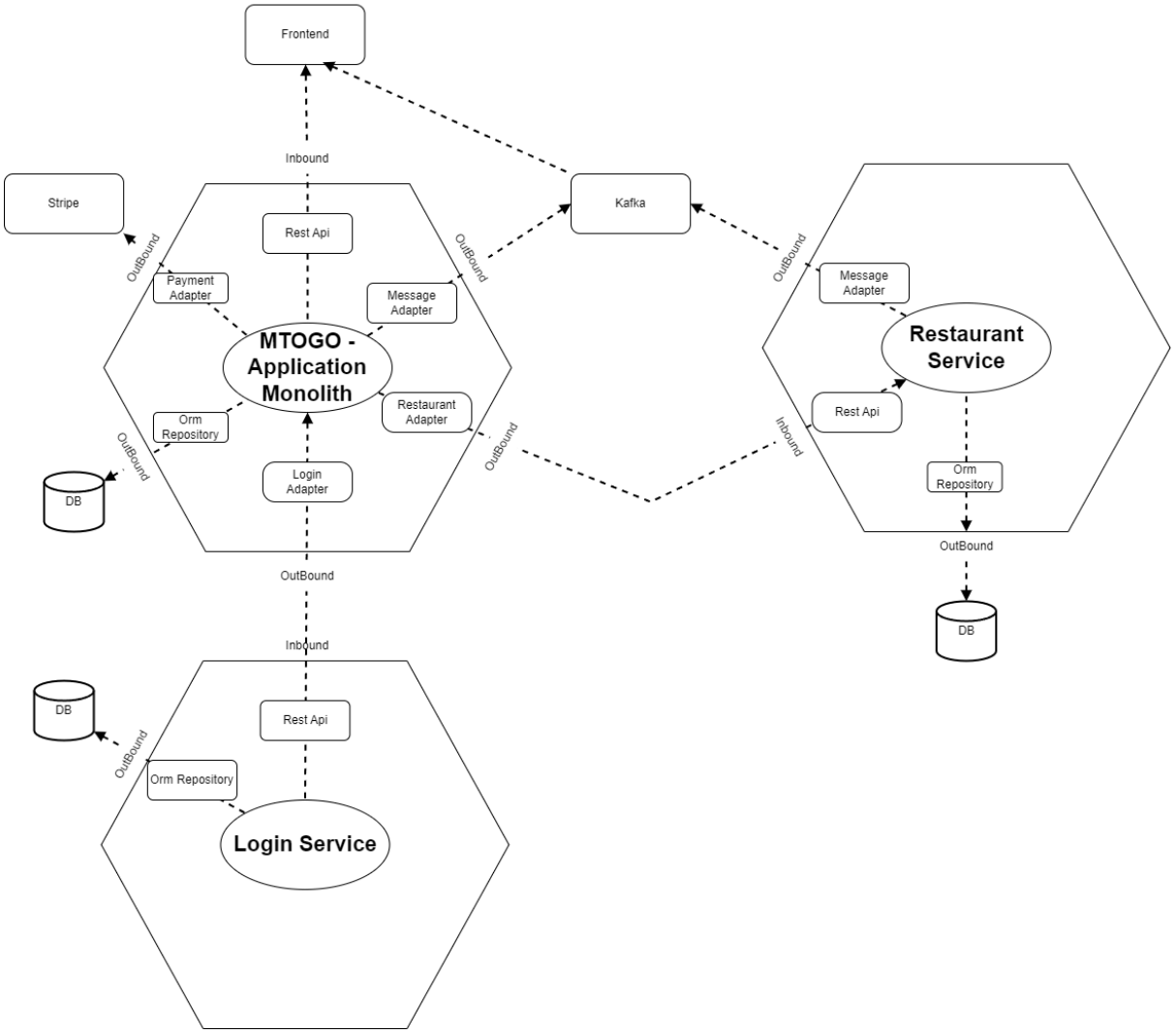
- The admin can access a dashboard showing:
    - The current number of open orders.
    - The average number of orders received per 24-hour period over the past week.
    - The average time taken to process orders (from placement to completion). (?)
    - ...
- Each metric is displayed in real-time and updates to reflect the latest order data.
- The admin can view a breakdown of metrics by restaurant for comparison. (?)

Diagrams:

Application Architecture:

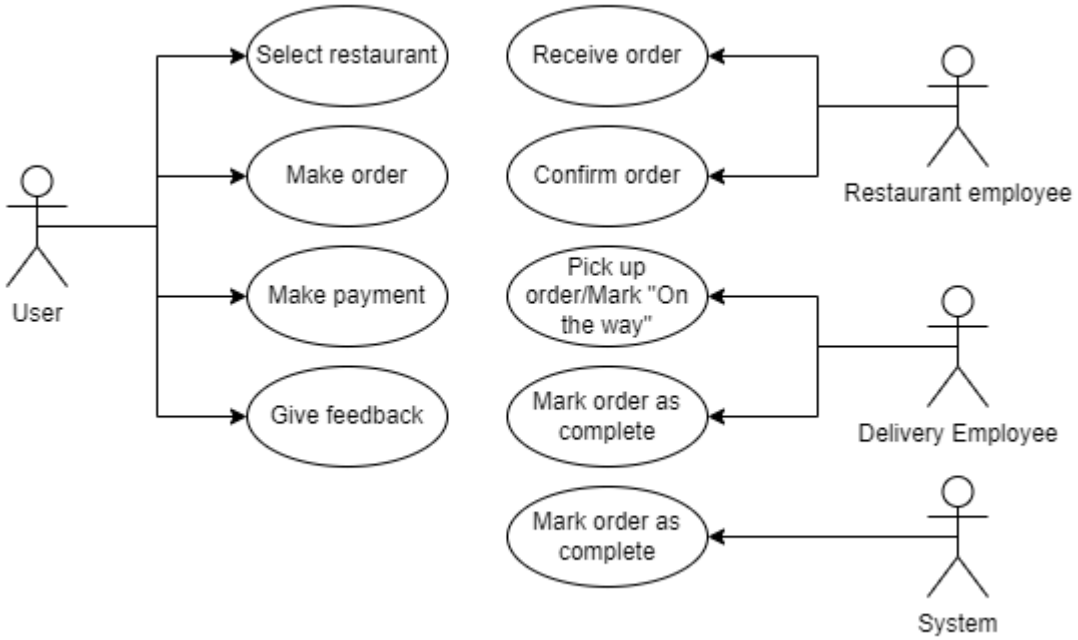


With our current instance of our application, we are using a monolithic architecture diagram, the architecture represented by the diagram above. We have a core application (monolithic code base) that interacts with different services. To come to production faster we are implementing some prebuilt services.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

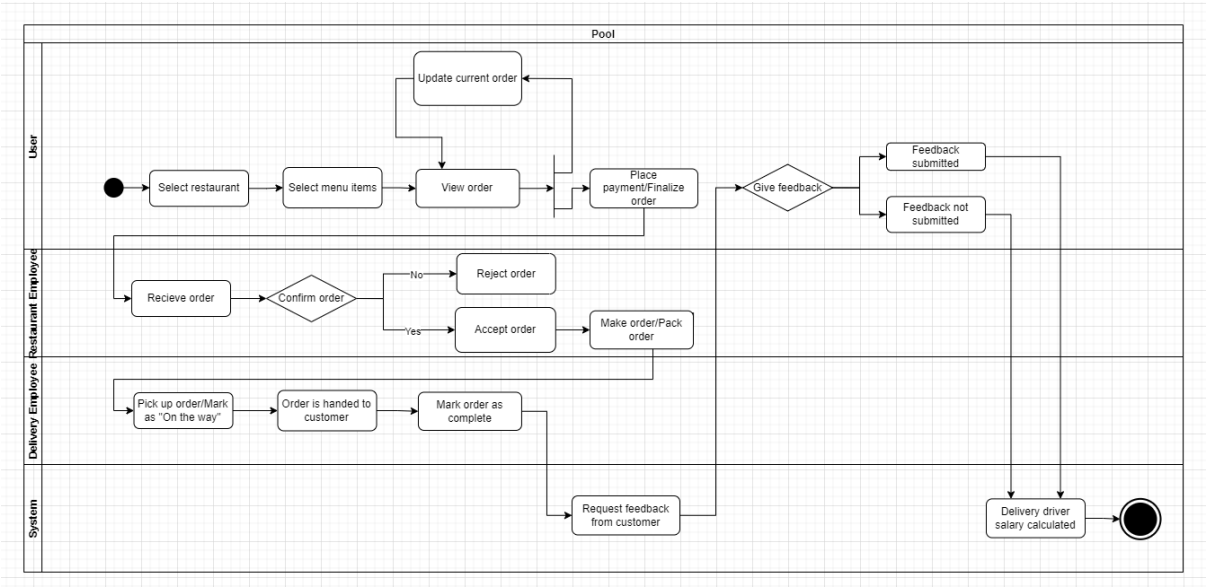


In this setup, we plan on completely decoupling the Login Service and Restaurant service, deploying each service as their own service. Each service will have their own dedicated Database and will interact. To simulate this architecture in code, we have attempted to decouple each service in our code to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

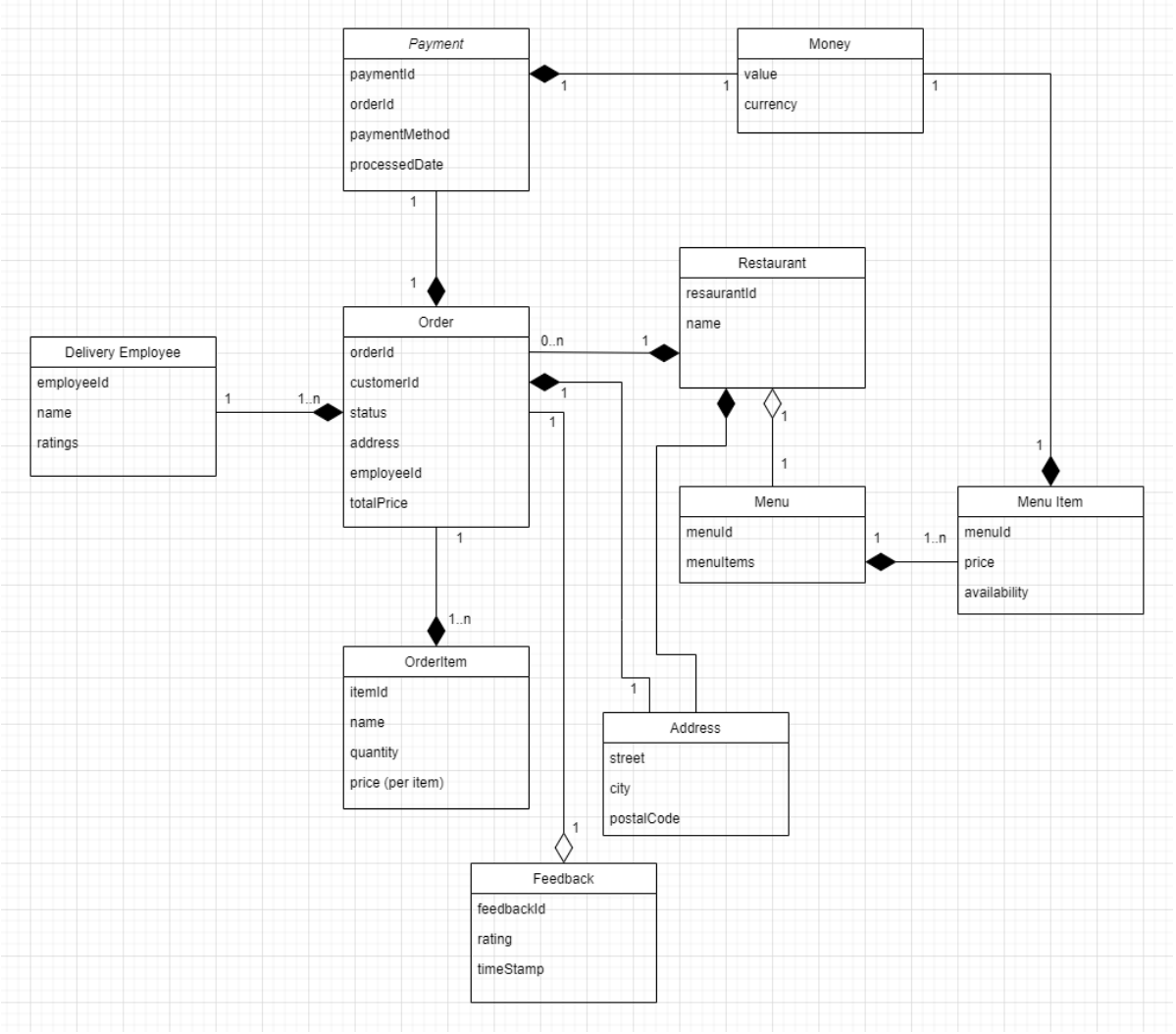
Use Case:



Sequence:



Domain:



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