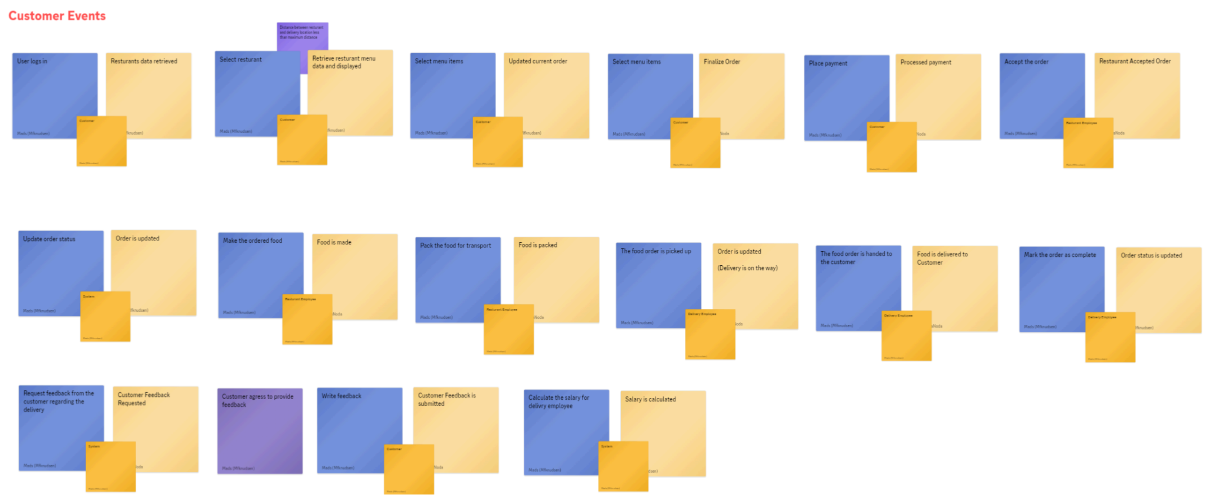
# Software Quality
# Exam Document

By Kristofer, Mads, Michael and Søren

## Event Storming:

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



## 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 can we see the need for our customer to login before placing an order and from that can we 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 rather or not if it is the application itself that needs to have the option to choose between the two options or if it's 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 giving 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. (?)
5. The customer should be able to proceed to checkout directly from the order summary.

**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.

**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. (?)

# Test Strategy Design:

## Scope and Overview:

For the first subject of our strategy we must create an overview of our project, as well as any information on who should be using this strategy. We should include information as to who will evaluate and approve strategy and lastly we should define the testing activities and phases that will be performed.

The project is about the development of a food order and delivery application wherein a user can select a restaurant and order food from their menu. They will then make the payment through the application and the food order will be sent to the restaurant. When the food is packed and ready for delivery then a delivery employee, from the company which we are meant to make the application for, will take the food, update the order and deliver the food order. When the order is delivered the employee will mark the order as complete. The customer will then be given the option to provide feedback for the delivery. Afterwards the system will calculate the bonus salary for the delivery employee.

The application will be made of three parts. The frontend where the customer will interact with our application, the backend which will run on a server, and lastly an external database.

The testing will primarily be done through the use of tools that will run on both our own computers during development as well as part of our continuous integration/continuous development pipeline. We will also be doing manual testing methods and analysis to ensure the quality of our product, and we will primarily do them as a group.

## Testing Methodology:

For the second subject of our strategy we specify the degrees of testing, testing procedures, roles and duties of our team members. We should define the testing process, testing levels, roles and the duties of each of our team members. We should also describe why each of the test types defined in the test plan should be performed, as well as any details such as when to begin, the test owner, the responsibilities, testing approach, and the details of automation strategy and any tools if any is used.

We have chosen a number of test methods for our project that being: Unit Tests, Integration Tests, Specification-based Tests, and within these we will be making use of things like: Simulation Testing, Equivalence Partitioning, Boundary Value Analysis, Acceptance Testing and Mocking.

The previously mentioned testing methods will primarily be used as part of our automated testing within the CI/CD pipeline, and will be created during the coding of the project. Other than these will we also be making use of tools for analyzing the code quality.
To keep the code standard up during the development will we primarily be using the built-in static code analysis tools that comes with our IDEA of choice, and for those who don't come with one, can we often find a plugin for a specific code language.

And will we also be making use of Code Coverage to analyze and ensure at least 65% of our code is covered in some way by tests.

Lastly we have the tools focused around group work, that being Peer Code Review and Software Review.
For the Peer Code Review will at least two members of our group come together to review the code written by another member of the group with the end goal being everyone being reviewed and having at least reviewed one other. It is also within here that we will perform taint analysis.
And for the Software Review will we primarily be going over the acceptance criteria and acceptance tests to ensure our application is meeting the requirement specified within the scenario.
Both the Peer Code Review and Software Review should be held at least once a week during programming to help keep the quality of the code consistent throughout the development process, and so that we can monitor the progress of our application.

- **Unit Tests:**
  These focus on individual components or functions in isolation to ensure that they perform as we expect. By testing each function in isolation we catch errors early in development.
  For unit testing, we'll create small isolated test cases to target individual functions, classes or methods within our codebase to test business logic. Each unit test will be written using Jest. Each function could have multiple test cases, covering different input types, functionality and cases.

- **Integration Tests:**
  Integration tests focus on how different modules of our system interact with each other. Using Supertest with Jest, we will be able to simulate real world api requests to ensure end-to-end data flow.

- **Specification-based Tests**
  Specification-based tests verifies that the application adheres to the requirements and design specifications. This helps confirm that all of the expected features and behaviors are implemented.For each requirement, we'll write test cases that directly target the expected behavior.

- **Simulation Testing:**
  Simulation testing will mimic real-life conditions that the user or app might face. Simulation tests will help our specification and acceptance testing, as they will be able to confirm whether the requirements and functionality work. Wil we can simulate the tests ourselves, we expect to use cyprus to mimic and simulate the steps the users go through.

- **Equivalence Partitioning and Boundary Value:**
  Equivalence partitioning involves partitioning input data into ranges and boundary values testing examines the test case. This helps in testing special cases, handling different inputs and such. In equivalence partitioning, we'll divide input data into partitions and create representative test data for each partition.

- **Acceptance Testing:**
  Acceptance test verifies that the entire application meets the defined requirements. By doing these tests, we will ensure that the app aligns with the stakeholders specifications.

- **Mocking:**
  Mocking will be used for any service that we depend on when we conduct our unit tests. In this way we will be able to strategically test logic independently and reliably, even for components that rely on third-party integrations.

- **Static Code Analysis:**
  Static code analysis helps identify potential errors or vulnerabilities in the codebase early. It is incorporated directly into our development environment to ensure good code standards and maintain quality during the development process. We will implement ESLint or Prettier into our project so that it automatically check code for syntax errors, potential vulnerabilities and adherence to standard.

- **Code Coverage:**
  To ensure that our tests cover at least 65% of our code, we will use Jest to analyze which parts of the code have been tested and covered. This metric will guide us in which parts need to me tested more, ensuring that all functions are tested properly-

- **Peer Code Review (Including Taint Analysis):**
  Peer code review allows team members to check each other's work, ensuring adherence to standards and identifying potential issues early. At Least 1 session will be held each week.
  Taint analysis is performed during these reviews to spot security risks, helping maintain a secure and high-quality codebase.

- **Software Review:**
  Our weekly software reviews will assess our current progress against the acceptance criteria and ensure we’re on track with project requirements. During these meetings, we’ll review completed acceptance tests, identify any gaps and ensure our application meets the project’s goals. This consistent review process helps us catch issues early, providing time to address them before the final release.

## Testing Environment Specifications:

The specification of the testing environment we have to define the test data requirements, with clear instructions on how to prejudice the test data. Overall this section should contain information on the number of environments and the required setup, the strategies for backup and restoration as well. Some of the data we will define for our test strategy is the number of environments and their individual required configuration, as well as the number of supported users within each individual environment, including the user’s access roles, software and hardware requirements including their operating system, ram, free disc and the number of systems. We also need to define the data needed for each individual test with specific instructions on how to create the test data, using either generated data or using production data by masking fields for privacy, and define a backup and restoration strategy as well. Due to unhandled circumstances in the code, the test environment database may encounter issues, so the backup and restoration method should state who will take backups when backups should be taken, what should be included in the backups, when the database should be restored, who will restore it, and what data masking procedures should be implemented if the database is restored.

For the testing environment we have three environments to consider. Firstly there is the workspaces of each individual group member, we then have the server where our backend instance will be running as well as where the users can retrieve the frontend. Lastly we have the GitHub action environment where our continuous integration/continuous development pipeline will be running which will be containing automated tests.

For both our CI/CD pipeline it would be best that they match to ensure all the tests within the pipeline behave exactly as they will on the server docker environment. For servers the general recommendation is to use Linux as opposed to Windows or MacOS due to it being more bare bone. Then we looked at what GitHub provides as options as well as DigitalOcean, which we intend to make use of as our server.

| DigitalOcean Server Droplet with Docker container |
|---|
| Ram: 512 MiB<br>Memory: 10 GiB SSD<br>CPU: Single thread<br>OS: (Linux) Ubuntu latest version |

| GitHub-hosted runner for public repository |
|---|
| Ram: 16 GB<br>Memory: 14 GB SSD<br>CPU: Multi thread (4)<br>OS: (Linux) Ubuntu latest version |

For the specifications of the testing environment for our group members, it would be too much to include every single environment that we make use of as we are four members who each have a minimum of two computers each, so instead we set up a minimum that each setup will adhere to.

| Group members workspace (Minimums) |
|---|
| Ram: 16 GB<br>Memory: 500 GB<br>CPU: Multi thread (At least 2)<br>OS: Windows 10 or Windows 11 |

## Testing tools:

When it comes to the tools we may use for our testing, we must define what tools for test management and automation that we will use for the tests, describe the test approach and the tools needed for performance, load and security testing, as well as mention whether the product is open-source or commercial, and the number of individuals it can accommodate.

For our project, will we be making use of GitHub and GitHub Actions for our continuous integration/continuous development pipeline which means we need tools that are capable of working within that environment during the automated process. We will also need tools that work in a manual sense where we run the test ourselves. The tools we have chosen are based on previous experience and based on what we view as needed to cover everything.

**Jest:**
Jest will be used for our unit test and integration tests. It's a testing framework which focuses on simplicity and works with typescript, which we use to make our application. It's capable of generating code coverage by the addition of flags and makes use of a custom resolver for imports in our tests which makes it simple to mock up.
We will also be using it to handle the code coverage analysis. Through the code coverage can we ensure that our project is properly covered by the tests we have created.

**Cypress:**
Cypress will be used for our single specification-based test. This testing framework makes it possible to simulate the users experience with simulated input for our final running application version. It allows us to create a set of actions that follows the user stories we create and the test is considered successful if all actions can be completed and if the simulated application successfully updates to reflect the actions.

**JMeter:**
JMeter is an open source software that is 100% made using Java and is designed to load test functional behavior and measure performance. It's used to test performance on both static and dynamic resources. It works on both Linux and Windows, and has the ability to perform load and performance test many different applications/server/protocol types: Web - HTTP, HTTPS (Java, NodeJS, PHP, ASP.NET, …), SOAP / REST Web Services, FTP, Database via JDBC, LDAP, Message-oriented middleware (MOM) via JMS, Mail - SMTP(S), POP3(S) and IMAP(S), Native commands or shell scripts, TCP, Java Object.
With that we can analyze the results and provide an update plan for scalability that is firstly based on our initial plan and later refactored based on the new information we will gather.

**Postman:**
Postman is an API platform for building and using APIs. Postman can simplify each step of the API lifecycle and streamlines collaboration so that we can more easily and faster create better APIs. With it we can also test our API calls once we have set up a running instance of our backend.

## Release Control:

The release control is used to make sure that the test execution and release management strategies are established in a systematic way. To do that we need to specify different software versions in the test and UAT environments that can occur from unplanned release cycles. And that all adjustments in that release will be tested using the release management strategy, which includes a proper version history. And finally set up a build management process that answers any questions like where the new build should be made available, where it should be deployed, when to receive the new build, where to acquire the production build, who will give the go signal for the production release.

Our release control we have planned to make it relatively loose. For our build process will we be primarily be using a continuous integration/continuous development pipeline that will test, build and deploy to the droplet when our source control tool detects a push to a specific branch, either the main branch or one made and name “deployment” with the specific purpose of only being pushed to when we want to update the build. When coding the application will we also be constructing test and those test are then meant to automatically run during the CI/CD pipeline and in the event where at least one fails then the entire process will end and an error message will be stored within our source control tool, GitHub and GitHub Actions.

While our deployment of the newest build of the application to the droplet should make use of a way to first deny any new inbound request to the message system, and then when the message system is empty, end the running instance and deploy the newest, starting it and allowing new inbound request to the newly running backend instance.
We have instead decided to not spend time on this due to time constraints and because there won’t actually be any steady stream of new inbound requests when deploying the newest build version.

Because both the developers of our group and the CI/CD pipeline should make use of the same version of dependencies for both the testing tools and for any libraries used within the code project. Because we believe setting a specific version for both Linux and Windows would be too much work for us to maintain throughout the entirety of the project, we will instead be making use of the latest version of both Linux, Windows, testing tools and any libraries used within our code project.

## Risk Analysis:

In this section we need to go over the potential hazards that are associated with our project that can become an issue during the test execution. We need to make a list of the potential dangers, provide a detailed plan to manage the risks and a backup in case the potential dangers become reality.

We will create a list of potential risks that we have considered which will follow with a management strategy about minimizing the chance of it happening and finally a backup on how to handle the risk should it occur.

| | Risk Description | Risk Management: | Backup Plan |
|---|---|---|---|
| **1** | File system command differences between Linux and Windows. | If statements based on the OS version the instance is running on with a response to both Linux and Windows.<br><br>Using Docker with a specific OS helps determine usage of command language. | Avoid using the file system. |
| **2** | External database is down due to errors outside our control. | We can't reduce the risk of the database shutting down due to errors outside our control. | Create error handles to ensure no tests based on interaction with a running database succeed. |
| **3** | Dependency updates breaking functionality | Lock dependencies to a specific version in package.json. | Maintain backup branches with the last working version, so we can revert if an update introduces critical issues. |
| **4** | Lack of environment parity with dev, test, production (differences between | Standardized testing and production environments by using docker containers or create ARM or Terraform templates to recreate dev, test and prod environments | Perform smoke test regularly |

# Test Plan:

## Unit Tests:

Firstly we want to, using unit tests, create mock functions for the different functionalities of our application. It is generally considered good practice to develop using Test Driven Development where we first create unit tests with mock functions before creating the final functions and it is therefore why we first will set up unit tests.

- **Function being tested:** `getAvailableRestaurants()`
  - **Description:** Fetches a list of restaurants for ordering.
  - **Setup for Test:** Mock the API call within `getAvailableRestaurants()` to return a test double (e.g., a list of test double restaurant objects).
  - **Expected Result:** The function returns the test double list of restaurant objects without throwing errors. Each object in the list should contain fields such as `name` and `location`.

- **Function being tested:** `getRestaurantById(restaurantId: string)`
  - **Description:** Retrieves details of a selected restaurant.
  - **Setup for Test:** Create a test double restaurant object to be returned by the API based on the provided `restaurantId`.
  - **Expected Result:** The function returns the mocked restaurant object with fields such as `name` and `price`. If the `restaurantId` is invalid,the function should throw an error.

- **Function being tested:** `addItemToOrder(menuItemId: string)`
  - **Description:** Adds a specified item to the current order.
  - **Setup for Test:** Mock the inventory check for the menu item. Create a test double for an initial order state.
  - **Expected Result:** The function adds the mock item to the order object, updating the item count and total price accordingly. If the item is unavailable, it returns an error message, and the order remains unchanged.

- **Function being tested:** `processPayment(paymentDetails: PaymentDetails,customer: Customer)`
  - **Description:** Processes payment based on provided payment details.
  - **Setup for Test:** Mock the payment service to return success for valid data and failure for invalid data.
  - **Expected Result:** For valid payment details, returns a success confirmation. For invalid details, throws an error or returns an error message without proceeding.

- **Function being tested:** `sendOrderNotification(orderId: string)`
    - **Description:** Prepares and sends a dispatch notification.
    - **Setup for Test:** Mock the notification service to confirm message sending.
    - **Expected Result:** Returns a success confirmation if the order ID is valid. For an invalid order ID, returns an error message without triggering a notification.

- **Function being tested:** `submitFeedback(orderId: string, feedback: FeedbackDetails)`
    - **Description:** Submits customer feedback on a completed order.
    - **Setup for Test:** Mock the feedback service or database to accept the feedback details.
    - **Expected Result:** Successfully returns a confirmation message if feedback submission succeeds. For invalid `orderId`, throws an error or returns an error message.

- **Function being tested**: `getRestaurantOrderList()`
    - **Description**: Retrieves a list of orders for a restaurant employee to track and manage.
    - **Setup for Test**:Create a test double list of orders with fields like `orderId`, `customerName` and `items`. Mock the data source call within `getRestaurantOrderList()` to return this test double list.
    - **Desired Successful Result**: Returns an array of order objects, each containing fields like `orderId`, `customerName` and `items`.

- **Function being tested**: `manageNewOrderStatus()`
    - **Description**: Allows restaurant employees to accept or reject new orders.
    - **Setup for Test**: Create a test double order marked as "new" with fields like `orderId`, `customerName`, and `status`. Use a mock to confirm that status update functions are called based on the chosen action (accept or reject).
    - **Desired Successful Result**: Successfully updates the order's status to either "Accepted" or "Rejected" and triggers a notification to the customer. The mock confirms the update method was correctly called based on the action.

- **Function being tested**: `viewOrderDetails()`
    - **Description**: Allows delivery employees to view details of a selected order.

- **Setup for Test**: Create a test double order with fields such as `orderId`, `customerDetails`, `deliveryAddress`, and `orderItems`. Mock the function that retrieves this order's details to return the test double.
- **Desired Successful Result**: Returns an order object with expected details, confirming it matches the test double and includes fields like `deliveryAddress` and `customerDetails`.

- **Function being tested**: `acceptDeliveryOrder()`
  - **Description**: Allows delivery employees to accept/pick up an order.
  - **Setup for Test**: Create a test double for an order with status set to "accepted." Mock the update function to verify that it only updates the status if the order has not been accepted by another employee.
  - **Desired Successful Result**: Updates the order's status to "In Progress" upon acceptance, and prevents other employees from picking it up. The mock confirms the correct update function is invoked.

- **Function being tested**: `updateOrderStatusOnItsWay()`
  - **Description**: Updates the order status to "On Its Way" to notify the customer of delivery.
  - **Setup for Test**: Create a test double order in "accepted" status with fields like `orderId`, `status`, and `notification`. Mock the status update function to confirm it changes to "On Its Way" and that a notification is triggered.
  - **Desired Successful Result**: Successfully changes the status to "On Its Way" and triggers a customer notification. The mock verifies the update function and notification were both called with correct arguments.

- **Function being tested**: `markOrderAsComplete()`
  - **Description**: Marks an order as complete upon delivery.
  - **Setup for Test**: Create a test double order with "On Its Way" status. Mock the status update function to verify it's only marked as "Complete" if previously "On Its Way."
  - **Desired Successful Result**: Updates the order's status to "Complete". The mock ensures the update was triggered with the expected final status.

- **Function being tested**: `calculateDeliveryBonus()`
  - **Description**: Displays bonus salary information for a completed order.
  - **Setup for Test**: Create a test double completed order with field `bonus`. Verify the test double accurately calculates the bonus based on policies.
  - **Desired Successful Result**: Correctly calculates and returns the total bonus, ensuring it aligns with company policy.

- **Function being tested**: `getAllOrders()`
    - **Description**: Retrieves a system-wide list of all orders for admin monitoring.
    - **Setup for Test**: Create a test double list of orders, each containing fields like `orderId`, `customerName`, `restaurant`, `status`, and `timestamp`. Mock the data retrieval function to return this test double list.
    - **Desired Successful Result**: Returns a comprehensive list of all orders across restaurants, verifying it includes expected fields like `restaurant` and `timestamp`.

## Integration Tests:

- **Function being tested**: `createDatabaseConnection()`
    - **Description**: Create A connection to the database.
    - **Setup for Test**: Function read environment variables correctly.
    - **Expected Result**: The database connection has been created without any errors.

- **Function being tested**: `getAvailableRestaurants()`
    - **Description**: Fetches a list of restaurants for ordering.
    - **Setup for Test**: Setup a connection with the database, create test doubles in the database so that restaurant entities exist in the database.
    - **Expected Result**: The function returns the test double list of restaurant objects without throwing errors. Each object in the list should contain fields such as name and `location`.

- **Function being tested**: `getRestaurantById(restaurantId: string)`
    - **Description**: Retrieves details of a selected restaurant.
    - **Setup for Test**: Create a test double restaurant object in the database to be returned by the API based on the provided `restaurantId`.
    - **Expected Result**: The function returns the mocked restaurant object with fields such as name and `price`. If the `restaurantId` is invalid,the function should throw an error.

- **Function being tested:** `addItemToOrder(menuItemId: string)`
    - **Description:** Adds a specified item to the current order.
    - **Setup for Test:** Create database connection. Create a test double for an initial order state.

    - **Expected Result:** The function adds the mock item to the order object, updating the item count and total price accordingly. If the item is unavailable, it returns an error message, and the order remains unchanged.

- **Function being tested**: `processPayment(paymentDetails: PaymentDetails, customer: Customer)`
    - **Description:** Processes payment based on provided payment details.
    - **Setup for Test:** Create a payment and customer test double. Use the test payment information for stripes.
    - **Expected Result:** For valid payment details, returns a success confirmation. For invalid details, throws an error or returns an error message without proceeding.

- **Function being tested:** `submitFeedback(orderId: string, feedback: FeedbackDetails)`
    - **Description:** Submits customer feedback on a completed order.
    - **Setup for Test:** Create database connection to accept the feedback details.
    - **Expected Result:** Successfully returns a confirmation message if feedback submission succeeds. For invalid `orderId`, throws an error or returns an error message.

- **Function being tested**: `getRestaurantOrderList()`
    - **Description**: Retrieves a list of orders for a restaurant employee to track and manage.
    - **Setup for Test**:Create a test double list of orders with fields like `orderId`, `customerName` and `items`. Create a database connection to return this test double list.
    - **Desired Successful Result**: Returns an array of order objects, each containing fields like `orderId`, `customerName` and `items`.

- **Function being tested**: `manageNewOrderStatus()`
    - **Description**: Allows restaurant employees to accept or reject new orders.
    - **Setup for Test**: Create a test double order marked as "new" with fields like `orderId`, `customerName`, and `status`. Setup connection to the database.
    - **Desired Successful Result**: Successfully updates the order’s status to either “Accepted” or “Rejected” and triggers a notification to the customer. The mock confirms the update method was correctly called based on the action.

- **Function being tested**: `viewOrderDetails()`
    - **Description**: Allows delivery employees to view details of a selected order.
    - **Setup for Test**: Create a connection to the database. Create a test double order with fields such as `orderId`, `customerDetails`, `deliveryAddress`, and `orderItems` and save them in the database.
    - **Desired Successful Result**: Returns an order object with expected details, confirming it matches the test double and includes fields like `deliveryAddress` and `customerDetails`.

- **Function being tested**: `getAllOrders()`
    - **Description**: Retrieves a system-wide list of all orders for admin monitoring.
    - **Setup for Test**: Create a database connection then create a test double list of orders, each containing fields like `orderId`, `customerName`, `restaurant`, `status`, and `timestamp and store them`.
    - **Desired Successful Result**: Returns a comprehensive list of all orders across restaurants, verifying it includes expected fields like `restaurant` and `timestamp`.

## Specification-based tests:

- **Function being tested**: placeOrder()
    - **Description**: Ensures the order placement feature meets requirements.
    - **Setup for Test**: Create test restaurant, customer, order, menuitems,
    - **Expected Result**: Validates that all specifications (e.g., item availability, correct pricing, and confirmation messages) align with user requirements.

## Mutation Testing:

Mutation testing will be employed during development to assess the robustness of the test suite. Mutation testing will be done either when a test is made when a test is made or during reviews. As mutation testing will be automated it will be easy to test like this and all potential tests will be tested.

## Usage of Test doubles:

Primarily throughout our tests we make use of dummy objects, stubs and mocks. For all of our unit tests we start by creating a dummy object that is solely used in equal statements to determine rather or not the test was successful. Together with our dummy objects we make use of mock functions as they only provide answers during the test and are pre-programmed with expectations of what they will receive.

With integration testing we again make use of dummy objects for containing the values to be used with equal statements and we make use of stubs by creating local methods within the tests themselves. We also make use of mock functions in the same way as the unit tests where we use the dummy objects as values for them.

Lastly we have our specification-based test which aren’t made the same way, that being through programming, and instead is created within the tool Cypress. As it simulates the final application, it doesn't use any extra code and because of that we don’t see how any of the points made by Martin Fowler fits this test.

### Usage of Mocking:

In our unit tests, mocking is used to isolate functions and simulate specific behaviors or responses. We create mocks for functions interacting with external services or dependencies, allowing the test to focus solely on the behavior of the function under test. Mocks are configured with expected inputs and return values to simulate various scenarios, helping us verify that methods handle both success and error cases as expected.
For integration tests, we rely on mocks sparingly to simulate services or dependencies that fall outside the scope of the components being tested. This keeps our focus on ensuring that components interact correctly, reserving mocks for parts of the system where real data or processing might not be feasible or practical for the test environment.

## Usage of Equivalence Partitioning and Boundary Value Analysis:

By using Equivalence Partitioning and Boundary Values as part of our tests, we can better understand how our code is affected by values that can change in size.
When using a set of business rules for how we want behavior to be and for how we want information to be set up, we can more easily ensure that the code meant to enforce those rules behave as expect, which also help us to ensure information receive via input from the user, or in other way outside the system, is within expected size.
Usage of Equivalence Partitioning and Boundary Values will primarily be used as part of the unit tests and the integration tests.

## Reviews:

With how the deliveries of the exam project is being handled, have we only a total of six weeks from beginning to our first delivery hand-in, with the last two being roughly one month later.
We have therefore allocated three weeks of the first six to develop a working high fidelity prototype.
We then have to consider how many reviews we intend to have during these three weeks, as they each can take a considerable amount of time from us, which could be spent on actually coding the application.
With that in mind will we attempt to hold a review session each friday with the entire day being allocated to this as well as going over any changes to the test plan that we need to make based on the weeks work.

## Code Review:

For the code reviews we will go over what tasks were completed and by who. We will split into two groups of two group members where we will go over the other groups work. We will primarily go over code standard, test coverage, tests, and if the code is based on one or more user stories then how well it meets the acceptance criteria.
The reviewers will not be making actual changes to the code they review after the review is over, but will instead create documentation of their review, from where the group member responsible for the code can read and make their own changes to accommodate.

## Software Review:

We will primarily be doing an overall software review of our entire application in a group with all group members present but should all members not be available during the review day then we need to at least have two members present.
While it would be best for all group members to be present during the review, we believe that a group of two is still enough as long as the reviewers provide a detailed enough documentation of the review to ensure that any missing members have a full understanding of the finding of the review and the reasoning behind the results.
With the software review also being weekly it will also help provide our group of the current progress of our application and any documentation made during the weeks, as well as a better understanding of how the other group members have worked and any changes to their individual time table.

## Code Coverage:

We will, as part of the Code Review and Software Review, also be going over how the code is covered by tests. The coverage will primarily be from the unit tests and integration tests as the tools we used include code coverage functionality while the other tests don't.
We will however still document how those other tools have covered the code of our application.

## Taint Analysis:

Remember to take notes and evidence for taint analysis, has it improved the code.

## Design Patterns:

In this section will we go over some design patterns and how we have considered making use of them when implementing features into our application, but it is important to note that just because we have written the patterns herer and where we could use them, does not mean that it’s a guarantee.
Later on will we also be documenting how we have implemented design patterns during our actual code development process.

**Adapter:**
Out of all the design patterns it is only the adapter pattern that is placed as a must implement in our system, as it is a crucial part of the microservice architecture design of which our application is made from.
The adapter pattern is made to act as a middle man between two incompatible interfaces. It converts one class's interface to another interface that the client expects.
It helps when an already existing system doesn’t match what we need, as well as when we want the option to change the system that the adapter converts to without having to change the underlying code used by the rest of the system.

**Strategy:**
For the payment service, Swipe will itselves make use of the pattern as it includes the options to, among other, pay with different types of cards. However we can extend our own payment options by letting the customer have stored payment methods where after we can implement the functionality to either use a pre-saved payment method or to manually type it in on the front. The API endpoint that receives the result from the frontend can then be made to handle both saved and manual.
The strategy pattern is meant to allow the choice between multiple different algorithms or methods during runtime without changing the source code. Different strategies would implement the same interface so that the client can choose between or change the algorithm dynamically based on their current need.

**Singleton:**
Considering that we are limited by our resources so that we only will have one database running for all the services, even though they should have their own for fault tolerance, means we could create our database service into a singleton from where it would then use the adapter pattern as described above. This way all the services have easy access to the database service.
The singleton pattern ensures that only one instance of a class exists within the whole of the system and provides global access within the system to the class. For a singleton class the constructor is set to private to ensure no other instances can be created and the only instance that will exist will be created within the singleton itself.

**Mediator:**
Considering we are planning on implementing a message broker system in our application that handles the incoming messages from the frontend, it could also be expanded upon to also handle messages between our services in the same way,

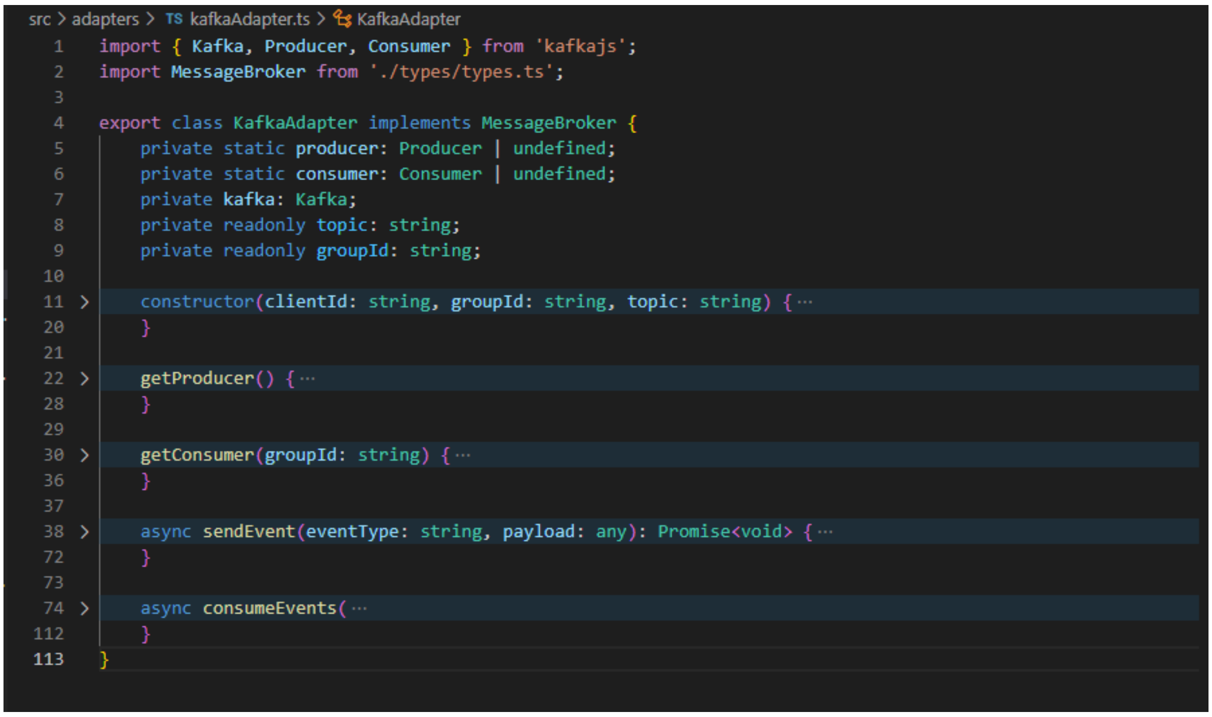
The mediator pattern introduces an object that controls communication between different classes that would normally interact directly between each other. This makes them less dependent on each other and reduces complexity since all interactions go through the mediator.

**Factory:**
Creating the orders from input from the frontend?
When receiving an order from the frontend we need to create a new order object within our system that also makes the correct updates to the different services, we will therefore make a factory that handles it all. We only need to make sure that whenever we create a new order object that it's done through the factory.
The factory pattern is used to abstract the instancing of new objects where the concrete implementation is hidden for the client. It provides a method for creating objects based on specific parameters which makes the system more flexible and less dependent on concrete classes.

## Implementation During Development:

Here we will describe each implementation of a design pattern we have within our code base.

### kafka adapter class:



src > adapters > kafkaAdapter.ts > KafkaAdapter

```typescript
import { Kafka, Producer, Consumer } from 'kafkajs';
import MessageBroker from './types/types.ts';

export class KafkaAdapter implements MessageBroker {
    private static producer: Producer | undefined;
    private static consumer: Consumer | undefined;
    private kafka: Kafka;
    private readonly topic: string;
    private readonly groupId: string;

    constructor(clientId: string, groupId: string, topic: string) {…
    }

    getProducer() {…
    }

    getConsumer(groupId: string) {…
    }

    async sendEvent(eventType: string, payload: any): Promise<void> {…
    }

    async consumeEvents(…
    }
}
```

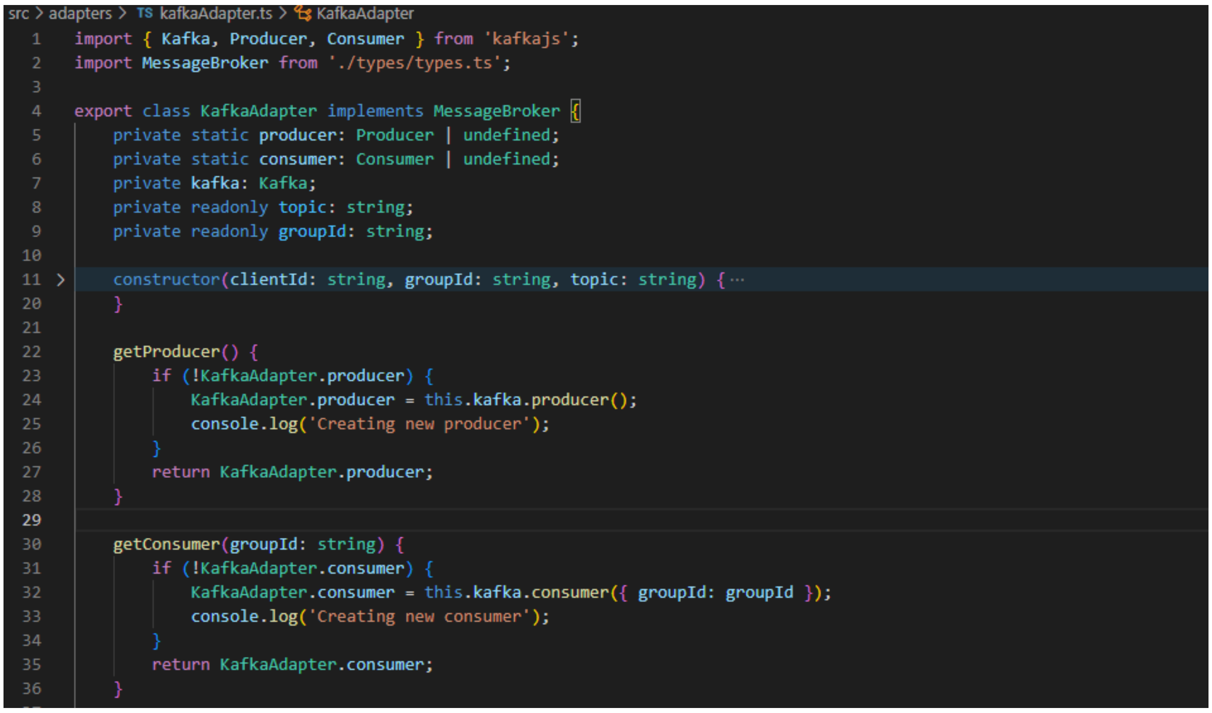
The KafkaAdapter class uses the Adapter design pattern to integrate the Kafka client library (kafkajs) with a custom interface (MessageBroker). This pattern helps the code by providing a consistent and simplified way to interact with Kafka, regardless of the underlying implementation details.

This pattern allows the KafkaAdapter to act as an intermediary, adapting the Kafka client's methods to match the MessageBroker interface. In the constructor (lines 11-19), the Kafka client is initialized with the provided clientId, groupId, and topic, and the Kafka brokers are read from environment variables. The getProducer method (lines 21-27) ensures that only one instance of the Kafka producer is created and returned, adapting the Kafka producer creation to the MessageBroker interface. Similarly, the getConsumer method (lines 29-35) ensures that only one instance of the Kafka consumer is created and returned, adapting the Kafka consumer creation to the MessageBroker interface. The sendEvent method (lines 37-65) sends an event to the Kafka topic, adapting the Kafka producer's send method to the MessageBroker interface. Finally, the consumeEvents method (lines 67-113) consumes events from the Kafka topic and processes them using a provided handler function, adapting the Kafka consumer's run method to the MessageBroker interface. This implementation allows the rest of the application to interact with Kafka in a consistent and simplified manner.

By implementing the MessageBroker interface, the KafkaAdapter class ensures that the rest of the application can use a standard set of methods (getProducer, getConsumer, sendEvent, and consumeEvents) to interact with Kafka. This makes the code easier to understand, maintain, and extend.

The Adapter pattern was chosen because it allows the KafkaAdapter to act as an intermediary, adapting the Kafka client's methods to match the MessageBroker interface. This means that any changes to the Kafka client library or the way Kafka is used can be isolated within the KafkaAdapter class, without affecting the rest of the application. This leads to better code organization and separation of concerns.

## Kafka adapter singleton



src > adapters > kafkaAdapter.ts > KafkaAdapter

```typescript
import { Kafka, Producer, Consumer } from 'kafkajs';
import MessageBroker from './types/types.ts';

export class KafkaAdapter implements MessageBroker {
    private static producer: Producer | undefined;
    private static consumer: Consumer | undefined;
    private kafka: Kafka;
    private readonly topic: string;
    private readonly groupId: string;

    constructor(clientId: string, groupId: string, topic: string) {⋯
    }

    getProducer() {
        if (!KafkaAdapter.producer) {
            KafkaAdapter.producer = this.kafka.producer();
            console.log('Creating new producer');
        }
        return KafkaAdapter.producer;
    }

    getConsumer(groupId: string) {
        if (!KafkaAdapter.consumer) {
            KafkaAdapter.consumer = this.kafka.consumer({ groupId: groupId });
            console.log('Creating new consumer');
        }
        return KafkaAdapter.consumer;
    }
```

In the provided KafkaAdapter class, the Singleton pattern is applied to the producer and consumer instances. This ensures that only one instance of each is created and shared across all instances of the KafkaAdapter class. Here's how it works:
The Singleton pattern for the producer and consumer in the KafkaAdapter class helps in several ways:

**Resource Management:**
Only one instance of the producer and consumer is created, saving resources and avoiding performance issues.

**Consistency:**
All parts of the application use the same producer and consumer instances, ensuring consistent behavior.

**Simplified Connection Management:**
The code handles connecting and disconnecting the producer and consumer only once.

**Reduced Overhead:**
Reusing a single instance reduces the overhead of creating and managing multiple instances. This was noticeable during testing. Our tests speeds drastically improved as they ran asynchronously.

**Easier Debugging and Maintenance:**

A single instance makes it easier to track and debug issues. Maintenance is simplified as changes are made in one place.

## Refactoring:

### Cases:

# Code Reviews:

## Review 1:

### Kristofer:

**Message Broker System #9:**

- **Code readability and maintainability:**

With microservices we want to make use of the adapter pattern to ensure decoupling of systems, but for the messaging system have you called the adapter: kafkaAdapter.ts, which breaks the design pattern. The adapter should be its own class which should bridge the gap between our system and the kafka system while maintaining the ability to switch out the kafka system. Changing the kafkaAdapter.ts to kafkaMessaging.ts and making an adapter called messagingAdapter which should be called from messaging.ts and through the adapter then call kafkaMessaging.ts.

Within kafkaAdapter.ts both the sendEvent() and runConsumer() functions do not have proper error handling.

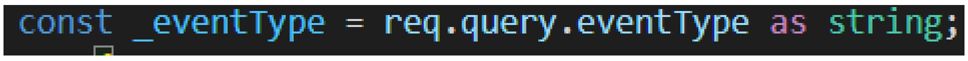
- **Adherence to coding standards:**

There are no tests written for the code.

- **Correct use of mocks in testing:**

There are no tests written for the code.

- **Identification of potential bugs:**



```
const _eventType = req.query.eventType as string;
```

Typescript doesn't verify types of query parameters, so using a type assertion may not produce the desired result, and introduce runtime errors.

- **Constructive feedback:**

With test driven development we should always start out with making tests for anything we code within our project.

### Mads:

**Test, build and deploy pipeline (GitHub Actions):**

- **Identification of potential bugs:**

The use of || true in the cleanup ensures that the pipeline continues even if the command fails. This can be useful if the command is not critical for the process, however it hides any potential errors that can be resulted from the command, as any errors will not be reported as errors.

- **Constructive feedback:**

Very nice

**- Backend create order and get all orders:**

- **Adherence to coding standards:**

Everything looks correct.

- **Correct use of mocks in testing:**

All database functions are mocked as they should in the unit test.

- **Identification of potential bugs:**

Our domain drawing model may have been drawn wrongly which meant the entities created for this service aren't correct. An Order should consist of a list of orderItems( Or orderlines). Currently in the code the entity consists of a list of menu items, while the idea is fine, we are missing the quantity.

- **Constructive feedback:**

We need to fix our domain diagram to incorporate the issue in potential bugs, from there we can properly fix feedback.

## Michael:

### Get all restaurants:

- **Code readability and maintainability:**

There has not been made use of the adapter design pattern for the restaurant service.

In the restaurant service folder there is a file called dbFunctions.ts which can be unclear when coding. While it is in the proper folder it should have a name that makes it more clear for what the database functions resolve around. It could for example be renamed to restaurantDatabaseFunctions or to make it adhere to other scripts with the same purpose but in different services: restaurantRepository.

In the test for “get restaurants”, undefined is used to represent an empty parameter list. This is unnecessary and reduces code readability.

- **Adherence to coding standards:**

The test made for the api call is not enough as we test for both successful and unsuccessful results. In this case there needs to be a test where the mocking returns an error or a wrong result, so that we know our api call will function no matter the end result.

- **Correct use of mocks in testing:**

The mocked list only contains one element, which potentially introduces some ambiguity as to what the test is doing.

- **Identification of potential bugs:**

Missing menu entity which we have drawn in the domain diagram. It might not be a huge problem that it's missing but we either have to remove it from the diagram or add it to the code.

## Søren:

**Backend login service:**

- **Code readability and maintainability:**

Perfect.

- **Adherence to coding standards:**

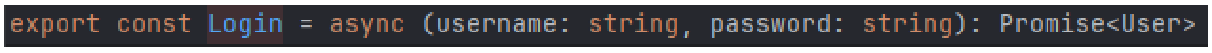
Perfect.

- **Correct use of mocks in testing:**

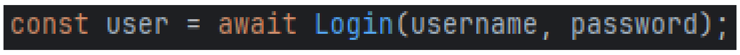
Perfect.

### Frontend login service:

- **Code readability and maintainability:**



```
export const Login = async (username: string, password: string): Promise<User>
```



```
const user = await Login(username, password);
```

Within the folder containing the api handling logic there is the file login.ts that exports a function called Login which is used within loginPage.tsx to make an api call with a username and a password, but the name of the function, Login, hides the method of how it logins, and changing the name to something like LoginAPI or LoginUsingAPI could help with readability for new developers by making it clear that it would make a call to our backend.

The mocking in the login test method looks correct, the mocking is unclear to us on how it works. Looking at the mock/handler.ts we understand the outcome and the general idea that it mocks the api call to the backend. More comments could solve this issue.

- **Adherence to coding standards:**

Perfect.

- **Correct use of mocks in testing:**

Perfect.

- **Identification of potential bugs:**

The username and password input from the user that is used when handling the login submit is not checked or sanitized in any way before it is sent to the backend. While the frontend handles the expected result and has error handlers for anything else, the frontend does not know how the backend will be affected.

### Backend submit feedback:

- **Code readability and maintainability:**

While the api app post call contains an error handling, the function createFeedbackAndLinkOrder does not contain any error handling which is a problem considering it interacts directly with our database repositories which can throw errors due to a number of reasons outside our control.

- **Adherence to coding standards:**

Perfect.

- **Correct use of mocks in testing:**

Perfect.

- **Constructive feedback:**

It would be nice to have a more clear oversight of the tests, how they work and why, through the use of code **comments**. Especially when the test script makes no reference to the mock handling script, so we don't need to look through it to find a specific mock for a specific test.

## Review 2:

### Michael:

**Display list of all restaurants - Frontend**

- **Code readability and maintainability:**

Almost perfect. - 1 console.log - customerPage line 11

- **Adherence to coding standards:**

Perfect.

- **Correct use of mocks in testing:**

No function to test.

- **Identification of potential bugs:**

Nothing.

- **Constructive feedback:**

Would like some code comments in the test regarding the mocking for easier readability.

**Get accepted orders - Backend**

- **Code readability and maintainability:**

In GetAllAcceptedOrders, if the database returns an empty list, we could make use of an early return statement, to prevent unnecessary code to run, and improve readability.

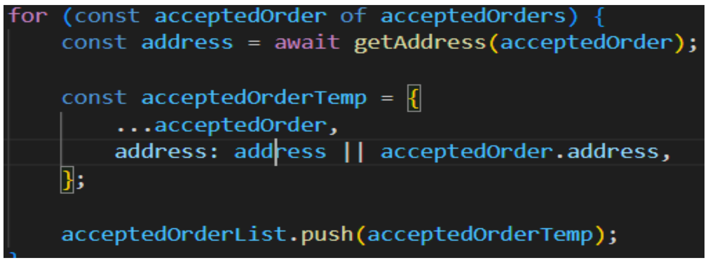
- **Adherence to coding standards:**

The tests //Get all accepted orders in order.test.ts is located under describe('Post /create') when it should be under a describe for 'Get /acceptedOrders'.

- **Correct use of mocks in testing:**

The test only mocks for successful responses. It should also mock for unsuccessful responses to ensure correct error handling.

- **Identification of potential bugs:**



```
for (const acceptedOrder of acceptedOrders) {
    const address = await getAddress(acceptedOrder);

    const acceptedOrderTemp = {
        ...acceptedOrder,
        address: address || acceptedOrder.address,
    };

    acceptedOrderList.push(acceptedOrderTemp);
```

The address associated with an order should always exist in the database, so if getAddress doesn't return an address, we have encountered an error, therefore we shouldn't just ignore it and set the acceptedOrders address to the id, but instead handle this error.

- **Constructive feedback:**

Separate tests into their own api call specific describe.

**Get accepted orders - Frontend**

- **Code readability and maintainability:**

GetAcceptedOrdersAPI returns an error with "Failed to login".
Function to call GetAcceptedOrdersAPI has the same generic name (fetchOrders) as the function to call GetOrdersAPI. May help to reduce ambiguity to give more declarative function names.

- **Adherence to coding standards:**

The values used in the tests are decoupled from the values used in the mock handler. If a value in the test file or handler file is changed the other file will not react and this could break a test.

- **Correct use of mocks in testing:**

The test only mocks for successful responses. It should also mock for unsuccessful responses to ensure correct error handling.

- **Identification of potential bugs:**

Nothing.

- **Constructive feedback:**

Separate tests into their own api call specific describe.

**Order details page - Frontend**

- **Code readability and maintainability:**

The restaurantPage is somewhat difficult to read and could benefit from being divided into smaller components.
handleOrderClick has an unnecessary try catch block and error message. Since all the logic contained in the function is a setState, it might be more concise to just call the setState in the onClick function itself.

- **Adherence to coding standards:**

Exported function is misspelled restuarantPage.

- **Correct use of mocks in testing:**

No function to test.

- **Identification of potential bugs:**

The top of the order details card is cut off, hiding some of the information.

- **Constructive feedback:**

Nothing

**Get orders by restaurant - Backend**

- **Code readability and maintainability:**

In GetAllOrdersById, if the database returns an empty list, we could make use of an early return statement, to prevent unnecessary code to run, and improve readability.

- **Adherence to coding standards:**

The descriptive text in the it() before the test for unsuccessful retrieval of orders, is wrong.

- **Correct use of mocks in testing:**

When we use “wrong id” to represent an incorrect restaurant id, it is a little bit confusing to use an actual object which may or may not be directly from the database to represent a correct id, would be more clear to just use something like “correct id”.

- **Identification of potential bugs:**

The address and customer associated with an order should always exist in the database, so if getAddress or getCustomer doesn’t return a valid object, we have encountered an error, therefore we shouldn’t just ignore it and set the ordersTemp address/customer to the id, but instead handle this error.

- **Constructive feedback:**

Nothing.

**Display order list (for restaurant) - Frontend**

- **Code readability and maintainability:**

The name GetOrdersAPI does not clearly represent what the api is fetching (orders by restaurant id - not all orders).
The function GetOrdersAPI which were first made to get all orders, have been changed to get orders by a restaurant id. The api call should have been giving its own api call GetOrdersByRestaurantIDAPI instead of overriding an already used function.

- **Adherence to coding standards:**

Order ID is not being displayed in the list.

- **Correct use of mocks in testing:**

The test uses a string for the id parameter of GetOrdersAPI but that id is disconnected from the mock response that is taken from /mocks/orders. Retrieving the id value from the mocks file would also ensure better readability.

- **Identification of potential bugs:**

Nothing.

- **Constructive feedback:**

The returned html from RestuarantPage() is hard to read, so adding code comments to separate parts of the code into sections with a description would make it easier to read.

# Software Reviews:

## Review 1:

- **Functional correctness:**

The purpose of the project is to make a food ordering application where a customer can view restaurants, select a restaurant, make and pay for a food order, and get notifications about its status. The backend functionality works but mostly are still separated with the lack of frontend functionality.

The development environment has also been deployed successfully. As discussed we are using digital ocean droplets to host, and are containerizing our services. We have several containers, backend, frontend, nginx, zookeeper and Kafka. For simplifying the prototyping, the containers are all deployed on the same droplet. The database is hosted on a mongodb cluster.

- **Non-functional aspects:**

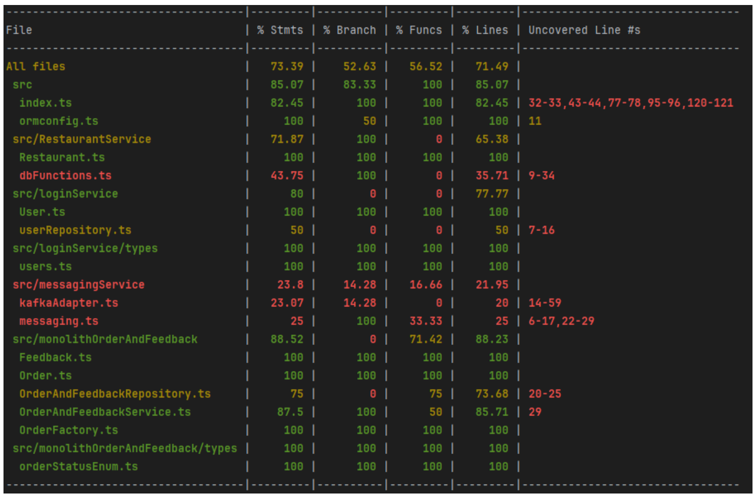
Tests have been made across the project to help ensure the code continues to work as intended in the future.

We have created a CI/CD pipeline which runs on every push to a specific branch of our GitHub repository called “deploy”. When pushed the pipeline will first connect to our remote server and clean it for the previous deployed versions. It then first tests, builds the application and then a docker image, and then saves the image to a tarball file which is transferred to the remote server. Afterwards the image is loaded from the tarball file and runs with a shared folder alongside the running instance of Nginx.
After the frontend is deployed then the backend goes through the same steps as the frontend but since it doesn’t interact with Nginx then it will not share a folder.

- **Adherence to best practices:**

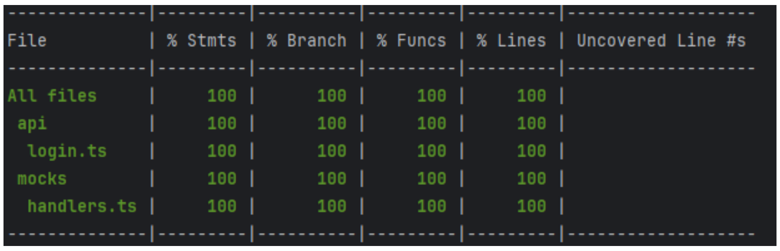
Both for our frontend and backend applications we meet the criteria set for percentage of code covered by tests, which is 65%.

Backend code coverage:



```
------------------------------------|---------|----------|---------|---------|---------------------------------
File                                | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
------------------------------------|---------|----------|---------|---------|---------------------------------
All files                           |   73.39 |    52.63 |   56.52 |   71.49 |
 src                                |   85.07 |    83.33 |     100 |   85.07 |
  index.ts                          |   82.45 |      100 |     100 |   82.45 | 32-33,43-44,77-78,95-96,120-121
  ormconfig.ts                      |     100 |       50 |     100 |     100 | 11
 src/RestaurantService              |   71.87 |      100 |       0 |   65.38 |
  Restaurant.ts                     |     100 |      100 |     100 |     100 |
  dbFunctions.ts                    |   43.75 |      100 |       0 |   35.71 | 9-34
 src/loginService                   |      80 |        0 |       0 |   77.77 |
  User.ts                           |     100 |      100 |     100 |     100 |
  userRepository.ts                 |      50 |        0 |       0 |      50 | 7-16
 src/loginService/types             |     100 |      100 |     100 |     100 |
  users.ts                          |     100 |      100 |     100 |     100 |
 src/messagingService               |    23.8 |    14.28 |   16.66 |   21.95 |
  kafkaAdapter.ts                   |   23.07 |    14.28 |       0 |      20 | 14-59
  messaging.ts                      |      25 |      100 |   33.33 |      25 | 6-17,22-29
 src/monolithOrderAndFeedback       |   88.52 |        0 |   71.42 |   88.23 |
  Feedback.ts                       |     100 |      100 |     100 |     100 |
  Order.ts                          |     100 |      100 |     100 |     100 |
  OrderAndFeedbackRepository.ts     |      75 |        0 |      75 |   73.68 | 20-25
  OrderAndFeedbackService.ts        |    87.5 |      100 |      50 |   85.71 | 29
  OrderFactory.ts                   |     100 |      100 |     100 |     100 |
 src/monolithOrderAndFeedback/types |     100 |      100 |     100 |     100 |
  orderStatusEnum.ts                |     100 |      100 |     100 |     100 |
------------------------------------|---------|----------|---------|---------|---------------------------------
```

Frontend code coverage:



```
--------------|---------|----------|---------|---------|-------------------
File          | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
--------------|---------|----------|---------|---------|-------------------
All files     |     100 |      100 |     100 |     100 |
 api          |     100 |      100 |     100 |     100 |
  login.ts    |     100 |      100 |     100 |     100 |
 mocks        |     100 |      100 |     100 |     100 |
  handlers.ts |     100 |      100 |     100 |     100 |
--------------|---------|----------|---------|---------|-------------------
```

- **After review improvements:**

While our test covers the needed percentage, it would still be a positive to create tests which would push each individual file to a coverage percentage of at least 65%.