SEM Assignment 1

Task 1: Software Architecture

Users:
We need to handle users in our application. Since they are such an important component and have separate functionality, they should have a separate context. As per the scenario description, the user needs to be able to create an account with a subscription type. So, within the context of the user we have a subscription type. On top of this, the users need to have a unique ID coupled to the authentication system. Additionally, we have different types of users that can be on the system. We have a customer, who places and edits orders, an employee, who can handle and process an order and a regional manager, who may edit and cancel any given order at any given time, a sort of omnipotent entity. To this aim, we will have roles within the user context. Finally, the user also needs to be able to specify if they have any allergies, so this is also part of the user context.

The user microservice will handle user account creation and data storage, and user functionality (what an employee can do is different from what a customer can do). It will also handle adding and editing allergens. Furthermore, it may receive requests from the pizza+ingredient microservice in order to filter out pizzas that contain the user allergens.

Orders:
In our application, we should allow users to create new orders, edit or remove pizzas that are currently in their order, or even cancel already made ones that are within half an hour before the given time. The local manager also wants to be able to see all currently placed orders and also cancel some specific ones. We should also be able to calculate the prices for each order and successfully apply coupons to get the new price for the current order. Since all this functionality is directly related to orders, we decided to separate them into a bounded context that handles all the aforementioned requirements.

The microservices will handle order creating, editing and removing pizzas in orders and also cancellation if in time. We store the orders in a database and after finalising one a price is calculated with the appropriate coupons which are used. A list of all the current orders can be returned when requested.

Stores:
Due to the pizza chain having multiple separate physical locations, stores should have a separate context. When an order is completed or when it is cancelled the respective store should be notified, as stated in the scenario description, this will be handled by the store context. Each store will have a unique location combined with a list of orders. The location of the store is used to provide the users with an option of stores they can pick up their pizzas from. Stores will also be responsible for storing the orders assigned to them whenever an order has been placed. Since stores are quite distinctly separate from other contexts, we

decided to keep it as its own context. It would also allow for more scalability, such as having pizza delivery for certain stores or each store providing a range of different food ordering options (e.g. different toppings available per store or special pizzas for certain stores).

We chose to map the store context into its own microservice since, as stated before, its functionality should be independent of other contexts. It would store all of the existing Anni's Pizza stores,
The microservice would communicate with the Orders microservice to fetch placed orders for each store. The Orders microservice would also

Pizzas:
We have a lot of operations involving pizzas. The regional manager should be able to make a list of pizzas that are available for purchase. Users should be able to choose and customise their pizzas with various toppings. Additionally, the client requested some functionality involving allergens. Since all these are a substantial part of the application and that's why we decided to make pizzas a bounded context. Furthermore, we thought it would be a good idea to make pizzas a separate bounded context for scalability reasons. If the store would want to include different types of food, we are able to just add these in our bounded context without having to duplicate any code.

This bounded context will be mapped in the microservice called food. This microservice consists of four main classes: Food, Ingredients, FoodRepository and IngredientRepository. The FoodRepository will handle the storing, updating and retrieving of the menu, while the IngredientRepository will handle the storing, updating and retrieving of the default set of extra toppings. The Ingredient class will just be value objects we use to calculate the final price of a pizza. The food class will be an abstract class from which we will extend and create the Pizza class. The motivation for this choice is scalability. If the store would like to offer extra food types, it would be trivial to implement this by just implementing another child class. The food class will take care of filtering the menu with respect to the user's allergens and warning the user in case they select a pizza that contains one of their allergens. This means our microservice will interact with the user microservice as it should retrieve the allergens. The final feature our food microservice will implement is the price calculation of a pizza. We could implement this feature anywhere but we decided to put it here to reduce coupling. The order microservice will use this feature to calculate the final price of the order.

Authentication:
The authentication microservice is crucial for this application. We thought about security and authentication for this application considering various roles that users can have (Manager, Employee, Customer). Managers are users who are admins of the application. They can order, view other orders, and furthermore edit and cancel other orders. Employees are users that can order and also view other orders. Customers are users that can only order. For users that are already in the system, they will need to be authenticated through this microservice when ordering. For customers that are not in the system, the authentication microservice will be responsible for registering them. If everything goes as planned and the authentication through Spring Security is successful, then the Authentication microservice will provide a JWT (Json Web Token) which can be used to bypass Security in the other microservices.

Coupons:

Coupons represent tokens that a store can issue and a user can subsequently use to affect the price calculation for an order. Coupons can have different effects, which can be implemented on demand (e.g: buy one get one free).
Coupons depend on stores, pizzas and ingredients.
Users (e.g: buyers, regional managers) and orders depend on the coupons.
Since the coupon system is substantial and interconnected enough with other contexts, we decided to make it a context of its own.

The coupon microservice consists of two main classes. The interface Coupon and the CouponRepository. The CouponRepository allows the regional manager to add and update coupons. The Coupon interface defines the behaviour of the coupons. It prescribes that coupons have two methods: validate and calculateDiscount. The validate method will check if the coupon is valid when the user applies it. The calculateDiscount method will calculate the discount of the selected coupon. We decided to make it an interface since it allows extra coupons to be added that each have their own validation and calculateDiscount method.