# Sem assignment 1

## 1. Task 1: Software architecture

We have identified seven different bounded contexts. We have chosen these boundaries based on a logical subdivision of the system focusing on individual component functionality. We will describe each bounded context below. The order in which we will describe these contexts is as follows: user, authentication, order, store, coupon, food, and ingredient.

We have identified *user* as being a discrete entity of our system that exhibits unique functionality and requires separate storage. That's why we have decided to create a bounded context for handling users and their interactions with the rest of the system. Users need to have a unique identifier coupled with the authentication system. Furthermore, users have different roles that we need to handle as they have different permissions and can thus execute different actions. Said permissions are also handled by the authentication component. The user bounded context maps directly to our user microservice which we will describe in detail later.

It is a well-known best practice to have *authentication* as a separate, generic subdomain. The authentication bounded context manages the authorities of all users based on their roles. This bounded context corresponds directly to the authentication microservice.

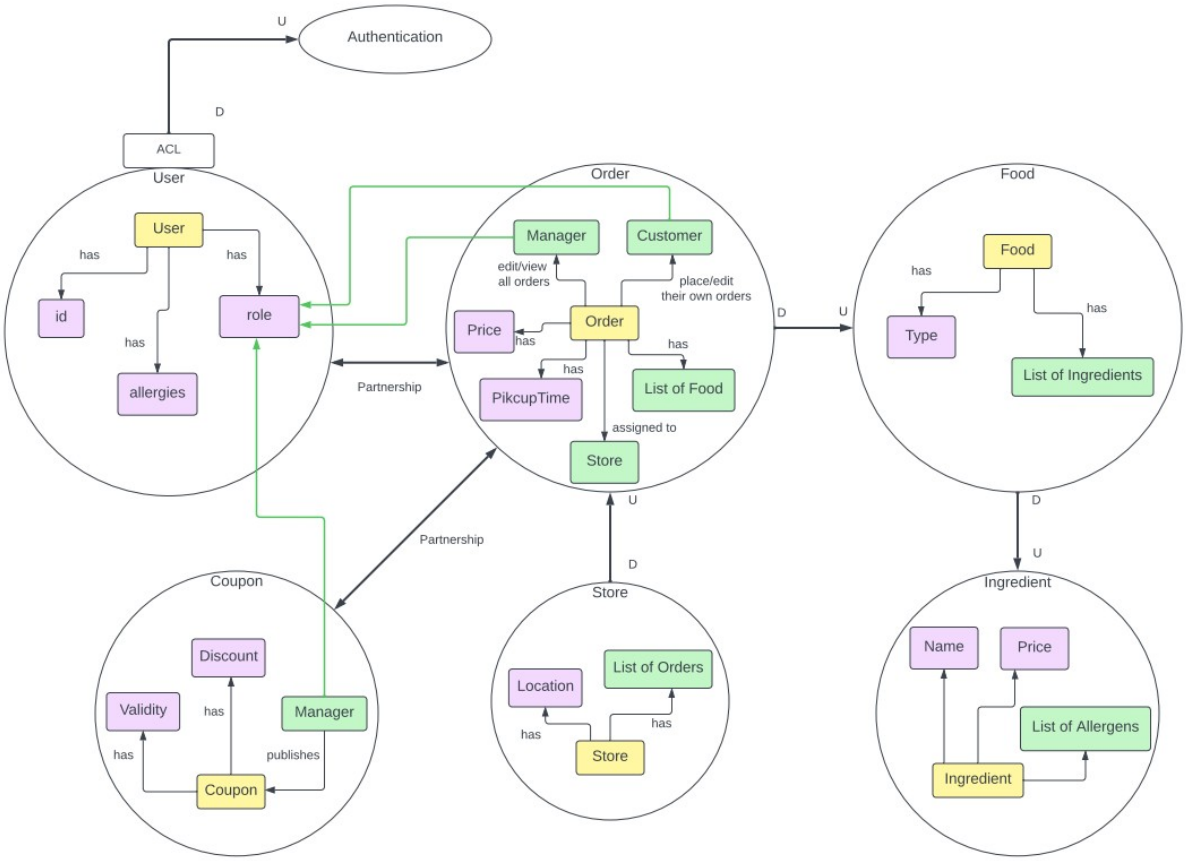
Orders are the central point of our system. They require separate storage and a hefty amount of business logic much of which relies on communication with other components. In particular, an *order* consists of a pickup time, a store, a list of foods, and the user who placed them. We will go into more detail when we talk about the order microservice.

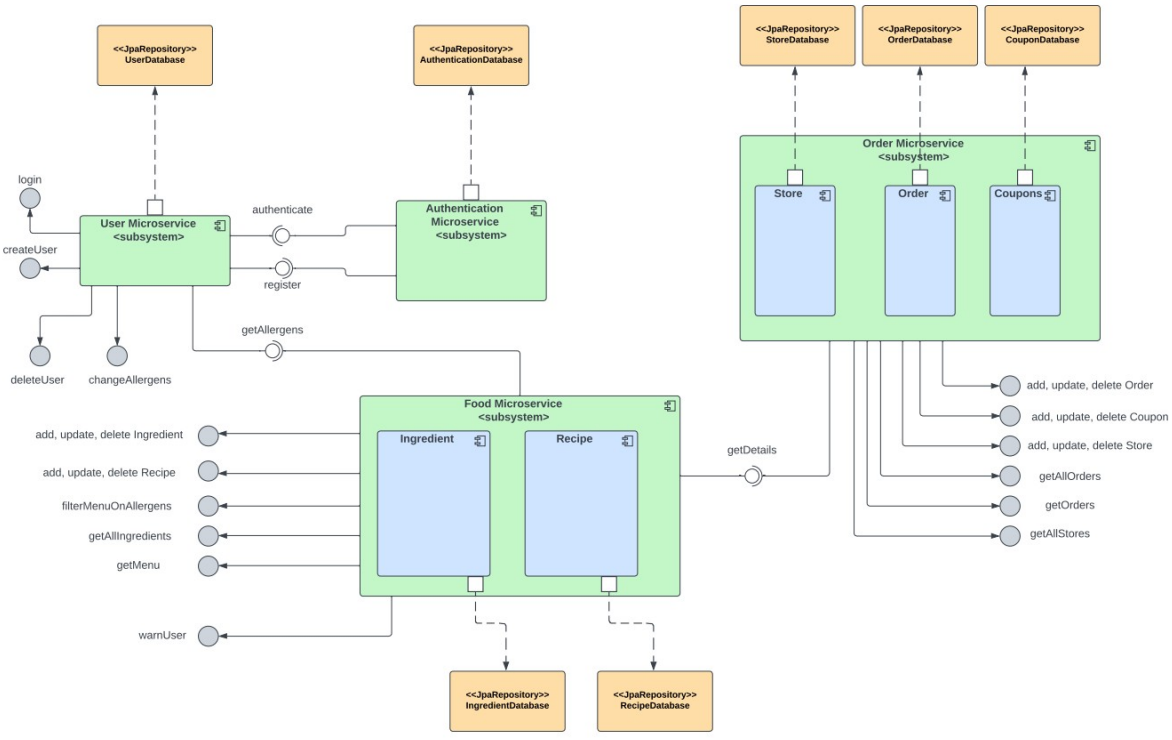
At the current state of our solution, *store* is a relatively small context. We decided to make it a separate component to increase scalability options when it comes to implementing new store business logic. We have decided to map this context into the order microservice since it has very few responsibilities and is highly coupled with orders.

Coupons have a unique role in the business strategy of the pizza chain. Even though they are only coupled with the order component, we still decided to make it a separate bounded context because of the real-world difference between a *coupon* and an order. That being said, this logical division would impede an actual implementation and that's why we decided to implement the coupon functionality inside the order microservice.

Thinking about *food* as a collection of *ingredients* is a more modular way of dealing with allergens in different recipes. This is why we decided to have two separate bounded contexts for food (recipes) and ingredients. Since we are creating a pizza store management system it makes sense to have a well-designed component to handle the functionality involving food. Due to their high coupling, we decided to map them in a single microservice called food.

Below you will find the context map, illustrating the bounded contexts. You will also find the UML component diagram.


Authentication
U
D
ACL
User
User
has
has
id
role
has
allergies
Partnership
Order
Manager
Customer
edit/view
all orders
place/edit
their own orders
Price
has
Order
has
has
PikcupTime
List of Food
assigned to
Store
D
U
Food
Food
has
has
Type
List of Ingredients
D
U
Partnership
U
D
Coupon
Discount
Validity
has
Manager
has
publishes
Coupon
Store
Location
List of Orders
has
has
Store
Ingredient
Name
Price
List of Allergens
Ingredient


<<JpaRepository>>
UserDatabase
<<JpaRepository>>
AuthenticationDatabase
<<JpaRepository>>
StoreDatabase
<<JpaRepository>>
OrderDatabase
<<JpaRepository>>
CouponDatabase
login
createUser
User Microservice
<subsystem>
authenticate
register
Authentication
Microservice
<subsystem>
Order Microservice
<subsystem>
Store
Order
Coupons
deleteUser
changeAllergens
getAllergens
Food Microservice
<subsystem>
Ingredient
Recipe
add, update, delete Ingredient
add, update, delete Recipe
filterMenuOnAllergens
getAllIngredients
getMenu
warnUser
getDetails
add, update, delete Order
add, update, delete Coupon
add, update, delete Store
getAllOrders
getOrders
getAllStores
<<JpaRepository>>
IngredientDatabase
<<JpaRepository>>
RecipeDatabase

Regarding the number of microservices that the application requires, we concluded that four would be enough. Authentication, User, Order, and Food. Each microservice was designed in the best way possible so that it provides efficiency and room for future scalabilities that could occur. In addition to that, each microservice was analysed about how it would interact with the other microservices efficiently, leading us to change our initial design by merging two microservices into one and having four instead of six microservices in total.

To begin with, authentication is a crucial microservice to maintain the correct functionality in the application. The main reason that authentication is a separate microservice, is that it simplifies implementation and management of secure authentication for the application. It also allows the authentication microservice to be updated or replaced without affecting the rest of the application. Moreover, using a separate microservice for authentication helps to ensure the security and scalability of the application, and allows for more flexibility and control over the authentication process.

The authentication microservice has numerous responsibilities. First of all, it defines the role of each user which defines its permissions. Our system defines the following two roles: Manager and Customer. Additionally, this microservice is responsible for registering new users. This works as follows. A user makes a request to the register endpoint by providing a unique identifier and a password. The role of this user is automatically set to customer and the id and the password are safely stored in the authentication database. Since every user is assigned the role of customer, we need a way of changing the roles in a safe manner. We do this by only allowing a manager to change roles. Furthermore, we have an authentication endpoint that is for already existing users. The user must provide the correct combination of an existing id and password. If that is the case we return a JWT (JSON Web Token) which is generated via Spring Security. For security reasons, we decided to not include the password in the JWT token since it will be used in all the other microservices for authentication in an efficient manner. Thus it only includes the id and the role. Finally, by passing around the JWT token, the other microservices can access the role of the user and give the corresponding permissions.

The user micro-service’s functionality is directly related to storing and extracting information about users. We can create a new account, login into an already created account, delete an already existing one, get the allergies related to a user and also change them.

Firstly, we need to be able to create an account. We have the createUser endpoint, which takes an object as a parameter that contains user-email, password, user-name and user-allergies, . We save the current user into the repository and generate a unique string id corresponding to the current user. Note that we don’t store the user password in this database. After that, we send a request to the authentication micro-service containing the user id we generated and also the password we had put in the object we sent.

The second endpoint we have handles the login part. We send an object containing user-email and password, we query by the email in order to get the user-id, and then we send a request to the authentication micro-service’s endpoint(authenticate) containing user-id and password. If we successfully

authenticate we receive a JWT Token which we will use for the future authentications. We then put that token into the response body of our original request.

Another endpoint we have handles requesting allergies for a user(getAllergies). It requires no parameters in the body, but requires you to put your JWT Token in the authorization header. If it gets successfully validated, the id is extracted from it and we query the repository for the user record corresponding to the id we have. We then extract the allergies from it and we put them into the response's body.

Furthermore we allow users to change their allergies(changeAllergies endpoint), if they provide their JWT Token in the authorization header and also a list of allergies in the body. The JWT Token is first validated again, the id is extracted from it, and lastly we update the record with the new allergies.

Lastly, we allow users to delete their accounts(deleteUser endpoint). In order to delete an account you need to provide your JWT Token in the request's authorization header. After that we extract the id of the user we want to delete from the token, and finally delete the record from the database.

Another important part of the application is the food microservice. In essence, it provides all the functionality a user would require before placing an order such as querying the menu (collection of recipes), filtering the menu according to the user's allergens, providing the functionality for the chain to update its menu, and communicating with the order microservice so it can calculate the price of an order.

As mentioned above, the food microservice provides the functionality so that any user can query the menu and the list of extra toppings. This is done by making a request to the menu and extra toppings endpoints. These endpoints both work by retrieving all the elements in the recipe and ingredient database respectively. Additionally, the food microservice allows managers to save, update and delete the collection of recipes and ingredients. The role of the user making the request will be checked by acquiring the information from the provided JWT. It is important to note that recipes store the ids of the ingredients they contain and not ingredient objects. Furthermore, the user can filter out the menu so that all the recipes do not contain any of the user's allergens or the user will be warned if he picks a pizza that contains one of its allergens. The filtering of the menu works by making a request to the allergens endpoint. This endpoint sends the JWT that the user passed to the user microservice that returns the list of allergens. Then the menu will be filtered recipe by recipe to check that none of the ingredients of each recipe contain any of the allergens. On the other hand, the warning of the user happens by receiving a recipe id and a JWT token on the warning endpoint. This endpoint will also send the JWT to the user microservice that returns the list of allergens. Afterwards all the ingredients of the specified recipe will be checked that they don't contain any of the returned allergens. If it does we respond by sending a Boolean that informs the UI whether it should show a warning or not.

The food microservice interacts with the user microservice as described above. Furthermore, we accept requests from the order microservice in which we return the names and the prices of the specified recipes and ingredients.

The order microservice encapsulates the functionality related to orders, coupons, and stores. We can create, edit and delete said entities. Additionally, the microservice provides viewing functionality for the regional manager to see the stores and the orders.

To start with, we have an addOrder endpoint, which takes in an order object (that comes from the UI or client side). An important part in this process is the verification of the user id. Through authentication, we make sure the person who places the order matches the id provided and/or is a valid user of the system. We also validate the price of the order by recalculating it, which includes requesting price values from the food microservice and applying (if any have been selected) the coupon that reduces the total price the most. We check that the products are valid and available, since the price request also returns an error if the requested items have problems. The interaction with the food microservice also includes getting names of the food and ingredients we request price for, so we can send that to the store in the notification, display orders more clearly, etc. Finally, we validate the pickup time is at least 30 minutes in the future. If all validations pass, we store the order in the DB, return the completed order to the user and notify (have a service send an email) the store of a new order.

Once an order is placed, a user can editOrder if they are still 30 minutes or more before the pickup time. We receive an edited order and execute the same validations as for adding it, resulting in returning the new order to the user, updating the DB entry and notifying the store of the change. A user can also cancelOrder in which case we only verify time and user id, before sending confirmation to the user, deleting our entry from the database and notifying the store. It is important to note that if the user has the role of manager, they can cancel any order they wish.

Furthermore, the microservice allows adding, editing, and deleting coupons through three respective endpoints addCoupon, editCoupon, deleteCoupon. To satisfy these requests, we verify the user has the role of manager (and thereby the permission to execute these operations). Coupons are additionally interacting together with orders (inside the microservice) to make sure the coupons that have been redeemed for the order (if any) are valid and can be used for the price calculation.

Within the microservice, we also take care of stores and have endpoints addStore, editStore, deleteStore to add, edit and remove stores from the list of stores. Once more, in order to execute these operations, the user must have the role of manager, which is verified. Stores interact together with orders when it comes to assigning an order to a store and notifying stores that there has been an order placed, edited or deleted. Once an order is serviced by the store or a user cancels it, this order is moved to another table in the order DB, containing orders that are done.

Finally, we provide functionality to view one’s order for the user through getOrders. Here, we verify the id and return all orders from this user, currently active ones, and also past ones (completed and cancelled). Also, a user with the role of manager can getAllOrders and getAllStores to view all the order history from the whole chain and all the currently placed orders, and to view all the current stores in the chain respectively.