Software Engineering Methods: Assignment 1

Group 04a: Bogdan Iordache, Kyongsu Kim, Matyáš Kollert, Anna Lantink,
Kristóf Lévay, Javier Páez Franco

December 2nd, 2022

1 Software Architecture

Our software project is creating an implementation for the ordering system in Anni's Pizza Scenario. Below, we will describe how we have implemented the scenario as a microservice-based architecture.

1.1 Bounded Contexts

The scenario states that different users should be able to do different things. So, we have identified three main types of users - customer, store manager and regional manager. We have merged these into the User's bounded context, which stores the login information along with a role, which determines what the user can do. The user also needs to be authenticated, which is why we have an Authentication bounded context which handles that.

Since the whole scenario is centered around ordering, our main bounded context is called Orders. It connects to the customer who created the order, the store at which the order was created at, the products connected to it and more information as specified in Figure 1. It made sense for us to group these things into an order, since an order is always linked with a store and products. The scenario states that a regional manager gets access to all the orders and is able to cancel them if they decide.

Another bounded context we identified was Store. Each customer orders a product from a certain store, and there is a store manager connected to each store. Coupons are also connected to stores. A store can exist outside of an order, so it is its own bounded context.

Based on the Scenario we decided that a product, such as a pizza, should be another bounded context. A product is grouped with its toppings, since toppings must always go on a product. Each product stores its name, price, allergens, and a list of toppings. The toppings themselves store their own name, price, and allergens.

The last bounded context we derived from the Scenario description is Coupons. We believe it should be on its own since there are many aspects to a coupon that the scenario asks for, like an expiration date, an activation code, as well as different types of coupons. The Regional Manager and Store Owners are allowed to create, modify or delete coupons. Each coupon also stores its unique code and id.

After considering all the options, we have decided to group Order, Coupon and Store into one microservice as we felt that these bounded contexts were closely related and would be communicating with each other often. Grouping these frequently communicating contexts will reduce overhead. We have also combined Authentication and User into one microservice, as most data used in one is also used in the other and separating them would create duplication. Menu is quite self-contained so we left it as its own microservice, which left us with three separate microservices.

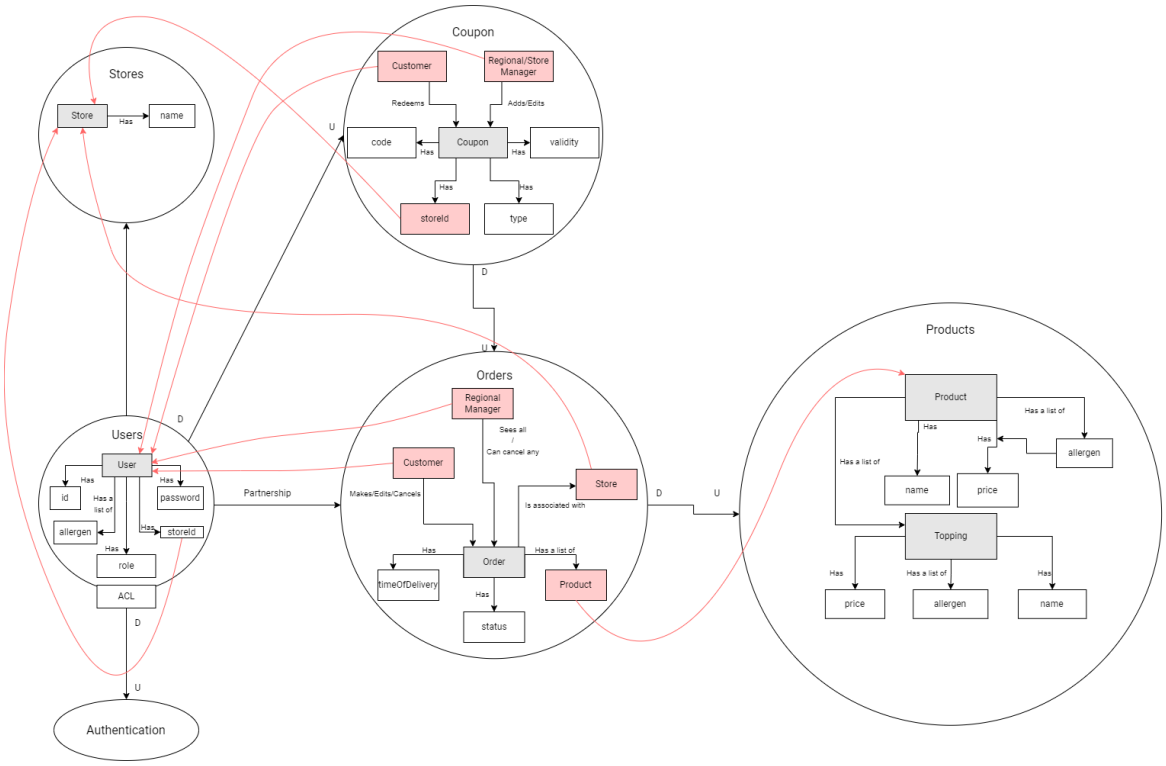

Figure 1: Bounded Contexts diagram of the Ordering System

1.2 Overall Architecture

As stated in the Bounded Contexts subsection, our software has three microservices. The first microservice is the User microservice, which is in charge of registering and authenticating users that access the system. All users must be authenticated to access the system. The user microservice also stores the roles of the users, and the allergens of customers. The second is the Order microservice, which allows users to create, edit, view, cancel, and pay for their pizza orders. Finally, there is the Menu/Product microservice, which essentially manages the menu of the stores, allowing food products to be created and viewed. For more detail about each component and their motivations, refer to subsections 1.3, 1.4, and 1.5. All of these microservices access persistence infrastructure, which in this case is the Java Persistence API, and the User component accesses security infrastructure, in this case, Spring Security. For a diagram of the software architecture, refer to Figure 2.
In Figure. 2 the subsystems and components of our system our shown. In our program, the subsystems correspond to microservices in our ordering system. As shown in the diagram, each of these microservices are their own self-contained system, that communicate with each other to produce our ordering system. The components in the diagram are packages within the microservice. Each component is somewhat self-contained, and communicates more frequently with other packages in the microservice than it does with external microservices. The JPA and Spring Security components are part of the infrastructure that our system uses for persistence and authentication, respectively.

Every user that accesses the system has a role assigned to them. They can be either a customer, a manager, or a store owner. The role that a user has determines which endpoints they can access. The microservices also communicate with each other through these endpoints, and they do that through REST requests.

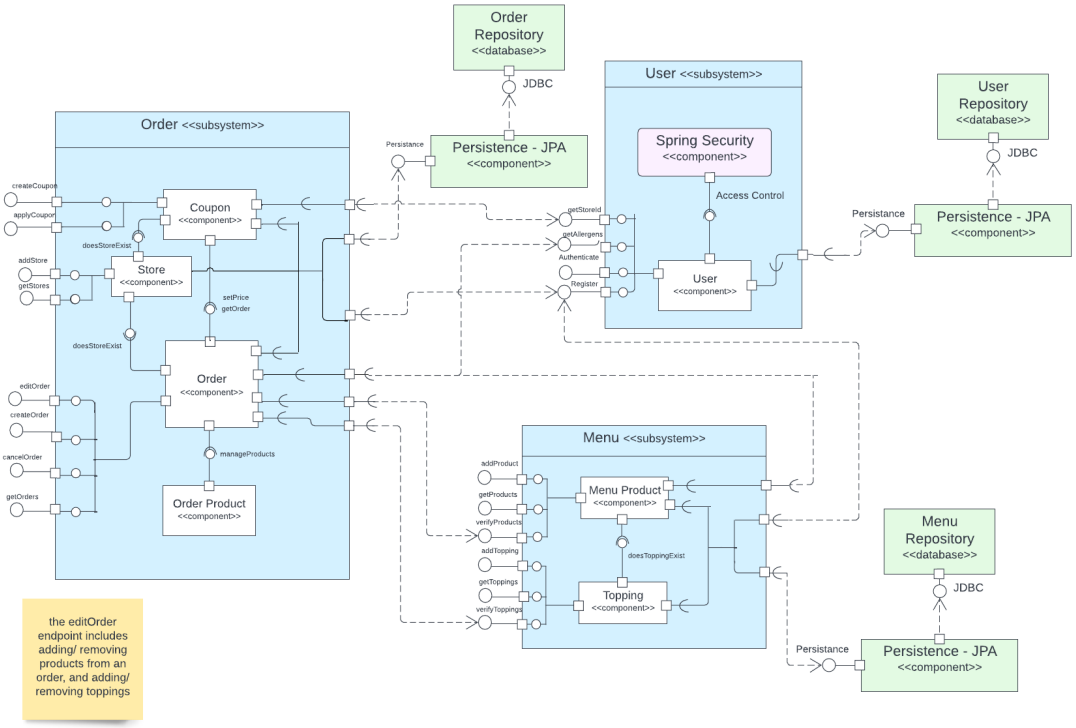

Figure 2: UML Component diagram of the Ordering System

The most significant role in the system is that of the customer. A customer interacts primarily with the Order component. A request is sent to Order if the customer wants to make, edit, view, or cancel an order using the corresponding endpoint. A request is also sent to Order once the customer wants to pay for their order, apply a coupon, or view the stores that they can order from. When an order is created or edited, the Order component interacts with the Menu component to get check if a product or topping is valid, and to get any other necessary information. The customer can also directly access the Menu/Product microservice to view the full menu that they can order from.

Another role in the system is that of the regional manager. Through the Order microservice, a manager can request to view all orders that are associated with a certain store, as well as being able to cancel any order. On top of that, they can make a coupon through the Order microservice, and this coupon will apply to all stores. Through the Menu microservice the manager can create toppings and products that go on the menu of stores.

The last role in the system is that of the store owner. The store owner can create coupons through the designated endpoint in the Order microservice, and these coupons will only work for the store that the store owner belongs to. They can also view orders associated to their store.

1.3 Microservice: User

The User microservice is responsible for handling authentication and security as well as handling all user information like roles and allergens. Within this microservice we have contained the bounded contexts of both users and authentication. Login information is recieved through an API call and if valid it returns a JWT token. The login information consists of an email, password, role, and store id (which is only relevant for store owners). The token must be sent to all other endpoints, so they can locally authenticate the user. Within this token we have encoded the email and the role of said user. Only users with certain roles can access certain endpoints. The possibles roles are 'ADMIN', which corresponds to regional managers, 'STORE_OWNER', and 'USER', which is the regular customer. The data we store for every user is a special id, a hash of the password and email, allergens, storeId (only relevant for store owners) and roles. As previously mentioned, different roles can do different actions. If an unauthorized user tries to access an endpoint, the action will fail and returns Unauthorized error. This is done through checking the role sent in the JWT token.

Registering works by checking if the email is not already in use and then saving the user in the database.The password saves is a hash, not the actual password. This is done for security purposes. A user can change their password, and the database will update accordingly. The authentication works by checking if the hash of the password that attempted to login is the same as the hash of the password stored internally. If so, the token is returned.

All endpoints besides registering and authenticating require a correct bearer token to be accessed. For allergens control we have endpoints for adding / removing an allergen. We store allergens by ids where each id refers to an allergen type. Each allergen is saved one by one, and the allergen id must be valid. There are 14 allergens according to the EU, and our allergen integers go from 1-14 according to this system. Additionally, a user can view all of their allergens.

Only admin users can get all users, and delete users. We added this because we believe admins should be able to view all their customers, and edit this list if necessary. Store owners, and only store owners, can access their store id (since other roles don't require a store id). We added this because store owners can add coupons and list orders according to their store id.

1.4 Microservice: Order

The Order microservice handles creating, editing, and cancelling orders, as well as viewing orders.

Users can view all stores and their ids at the *listAllStores* endpoint. Customers can then easily create an order by accessing the *sendOrder* endpoint, specifying the store they want to buy from and choosing a set of products, and the delivery time. The delivery time of an order must be at least 30 minutes to maximum 12 hours in the future. They can also specify the toppings of each pizza. Customers can change their minds after making an order, so orders are saved in a sort of "inactive" state, before the customer pays. While an order is inactive, the customer can still modify the order with the *editOrder* endpoint. Because of the ability to edit an order, we introduced a finalization step in the ordering process. A customer completes the order by paying for it at the *payOrder* endpoint, which simply flags an order as 'PAID', and they can not longer modify the order. We did this because it doesn't make sense for a pizza to be prepared by the store that can still be changed. The store is notified that an order has been paid, since then is when the order should start being made.

Users can request to cancel orders with the *cancelOrder* endpoint, and have it notify the store of the cancellation. Customers can cancel their orders, and managers can cancel any order. Users can also view orders. Customers are limited to only getting information about their orders, while managers can get information about any order in the database, and store owners can only get the orders of their store.

Another main functionality of the Order microservice is pricing and coupons. The set of prices corresponding to the products that the user selected is received from a Product endpoint. Once an order is made, the total price of the order is calculated and added to the order database. A coupon can be applied to an existing order. In this case, the activation code is checked through the coupon database to see whether there exist a corresponding coupon. Afterwards, it examines whether the coupon is valid using the expiry date, and also checks if the store that the order is associated with matches the store the coupon is associated with. If the coupon is considered valid, it finds the type of the coupon, which is stored using an enumerator. An enumerator was used because it would allow for other coupons in the future to be added. The coupon type will influence how the price is modified using a strategy design pattern. Our service contains two coupons for now, which is 'Buy One Get One Free' (BOGO) and discount. When there are several coupons inserted, the coupon that returns the cheapest total price will be selected.

As stated in 1.2, a store owner or regional manager will be able to create new coupons. The microservice will receive the coupon type, discount percent, expiry date and activation code through the endpoint. It will check that the type and discount percent (if discount type) is valid, and that the expiration date is formatted correctly and after the current date. It will also check that the activation code is unique and is formatted as 4 letters followed by 2 numbers. The activation code was chosen to be unique because it makes interactions with the database less complex, and a unique activation code for each coupon also makes coupon application less confusing for the user. The store ID of a coupon is determined by the user role. If they are a store owner, the store ID corresponds to the store of the owner. If the user is a manager, the coupon applies to all stores. If valid, it will create a new coupon with unique ID.

For all this functionality, we merged the bounded contexts of Orders, Stores and Coupons, making the Order microservice the largest. Our motivation was that the concept of stores and prices is so intertwined with orders, as can be seen in Figure 2 that the price and the store to buy from can be viewed as properties of each order. Therefore, it is best to keep the info about prices and stores directly as part of Orders to avoid heavy coupling and overhead.

1.5 Microservice: Menu

The menu microservice handles all the requests connected to the default menu of each store, which includes adding and retrieving default products and toppings along with verifying a product/topping that a customer wants to add to their order. It also stores the default products and toppings in its own database.

Users can request the current products and toppings available at the *getProducts* and *getToppings* endpoints respectively. The regional manager can also add new products and toppings through the *addProduct* and *addTopping* endpoints. Once a user selects items for their order, their names need to be verified in the product microservice first which the order microservice does through the *verifyProducts* and *verifyToppings* endpoints. If the items are valid, the microservice checks the user's allergens and adds a warning to the response if necessary. Afterwards the items can be added to an order.

For scalability, we decided to allow multiple types of products, not only pizzas. This is also why Product was chosen to be its own microservice, because that way the end user can directly interact with the menu rather than having to go through order. This made sense to us because sometimes a customer might want to view the menu without necessarily making an order.