LAB 3: DOMAIN DESIGN

The goal of this lab session is to teach you how to translate a collection of user stories pertaining to a specific problem domain into a set of micro services that can realize these stories.

You will learn to:

- Identify services, and how they interact with each other through events and commands.
- To design a micro-service architecture, including domain services, sagas, aggregates…

1 User stories

Consider a factory that makes weaving machines to be sold all around the world. They take orders from old or new textile factories (weaving mills), each order often consisting of hundreds or even dozens of machines. The company manufactures them and ships them to their destinations. Machines have many different options and hence come in many different configurations.

1.1 Placing an order

The sales department of the company handles all sales. A client wishes to place an order and negotiates the specifics with the sales department. Before the sale is approved it must first be checked if the company has the available production capacity to meet the requested deadline and if all items can be shipped to the country in question. Machines with much less strict safety equipment are legal (and sadly preferred) in countries like Pakistan or China but are illegal in the EU due to strict workplace safety laws. If everything checks out the sale is approved.

1.2 Keeping stock

Keeping stock is expensive. The less stock you can keep while keeping production running the cheaper your operational overhead costs. The stock department tries to optimize this process. Based on the current production queue, historic data as well as data regarding the typical delivery times of the various suppliers the department tries to optimize its desired stock and places orders accordingly. Since larger grouped orders tend to get larger discounts the stock service also takes this into account.

A second responsibility of the stock department is trying to optimize the inventory storage. An optimized storage keeps the used space to a minimum (so more items can be stored in the same amount of space) while also optimizing the operational cost of adding and removing items (i.e. store ordered parts when they arrive and fetch parts required for production)

1.3 Production & QA

Machines that need to be produced are placed in a queue. At a steady rate machines are produced and when they are, they are handed over to QA. The unique code on the machine is scanned by an employee, which signals the QA department. QA will also add the machines to be checked to a queue. When the machine is checked it is either accepted or rejected, based on whether or not it passed. If it is accepted the shipping department will add it to the waiting shipment, if not it goes back to production.

1.4 Shipping

When a sale is being made the sales department tries to create a new shipment as part of the ordering process. The shipping department will only allow this if the delivery is legal (i.e. safety ok). The shipping department then creates a corresponding delivery for the order. The delivery is then split up into multiple shipments. A shipments is a transportation of one or multiple machines from one location to another. A shipment can be made by truck, ship, or plane. These services are booked with partnered transportation companies. If multiple deliveries are to be made to the same region the shipping department may choose to delay shipping a bit to group the orders and use a ship rather than many trucks.

As machines are accepted by QA they are added to their respective deliveries. If a delivery is complete and does not need to be grouped it is sent out.

2 Assignment

Design an architecture for the user stories of section 3. Make a service diagram of your architecture using the three step process from the theory course (a recap is in the appendix). If you want you can collaborate with your neighbor. After finishing your service diagram show your supervisor before making detailed service diagrams for each of your services. After finishing these, again show your work to your supervisor.

3 Appendix: Theory recap

First we will give a short recap of the relevant domain design aspects shown in the theory course.

Aggregates and their invariants

A DDD aggregate is a cluster of domain objects that can be treated as a single unit. Only the aggregate will be exposed to the outside so the aggregate can maintain its invariants. Invariants are conditions that must always be true for the aggregate. An example could be a plumber's work schedule, which contains various tasks, but the total load of the schedule should never exceed 40 hours per week. Another example is a coupon that gives you 5% discount on your order total if you ordered at least $200. If items are removed from the basked the coupon must also be removed if the total drops below $200.

Domain services

Domain services are units of pure business logic. They should be kept in separate classes in the micro service in order to isolate changes to business rules. That way, if the business rule changes it is localized to a single class which is responsible for it. This also means that a single set of tests needs to change. An example of a Domain Service is a service that calculates housing tax based on (current) regulations. Note that a domain service resides inside a micro service.

Sagas

Sagas model processes within the business logic of a service. If a strict order of events needs to be upheld or a cross service transaction is required then you should use a saga. An example can be found in our hospital application. When booking a visit, the ward operations is contacted to reserve the bed and an invoice is opened. If either of these fail, the other is rolled back.

Separation of adapters and controllers from the domain model

Make sure to keep controllers and adapters separated from your domain model and keep the boundaries strict. Also separate domain service logic into services and sagas, each contained within a different class. Keeping these separate will improve the understandability of your design and isolate changes to the controllers and adapters from your model.

Events and commands

Micro services can communicate with each other either by using commands, queries or events. Both can be implemented with either REST or messaging. Since REST is typically synchronous it lends well for queries and short commands (with a return value). Messaging on the other hand, being asynchronous, lends itself better for events or long executing commands. Messaging also allows easy one-to-many communication. However, both methods can be implemented either synchronously or asynchronously.

REST leads to tighter coupling since it invokes another service directly whereas messaging does not. This means that when using messaging the service sending the event does not need to know of the existence of the service(s) consuming it nor how these consumer(s) handle the event. REST does provide more security options typically while messaging is easier to scale. As a rule of thumb, REST is used for public facing API's whereas messaging is used internally as much as possible.

SQL vs NoSQL

Since each micro-service has its own data and hence its own database it can freely choose which database technology to use. Typically the choice is between an SQL database vs a Mongo document store (but things like graph databases etc exist as well). One of the reasons one might favor Mongo over SQL is if the fields of the data vary. Think of an object with many subclasses. Each subclass has different fields but with mongo they can easily be stored together. Another reason you might prefer mongo over SQL is if the entity you are storing contains collections of other entities. In Mongo they are simply stored as a JSON array in the document, whereas in SQL separate tables need to be made and every query needs to join the data again.

4 Appendix: Tips for identifying relevant information

In the theory sessions you have seen the three step process needed to develop your architecture. First identify the system operations, i.e. consider your system as being a single block and define the operations happening on your system. Next try to identify the services within your system. Typically it is a good idea to split services on bounded context boundaries or department boundaries if they exist. (E.g. ward operations, reception …). Finally draw events, commands and queries between your services that make the high level operations possible. Doing so you get your service decomposition, an example of which is shown in figure 1.

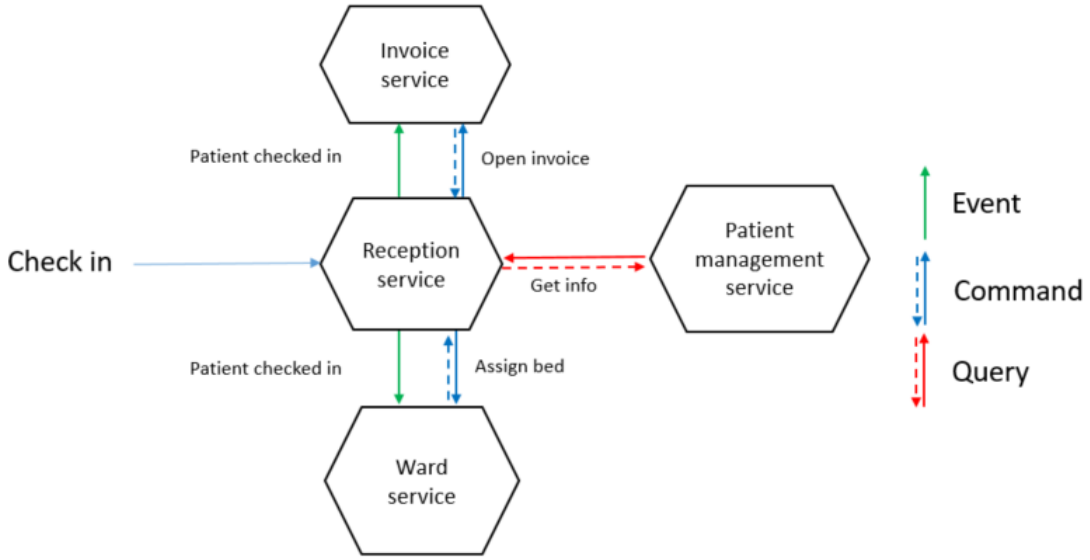

Figure 1: Service decomposition example. 'Check in' as only system operation.

Once you have your service decomposition you will want to make a detailed service diagram of each of your services showing the internal structure. An example can be seen in figure 2. Note however that a service can be very simple and must not necessarily be this complex.

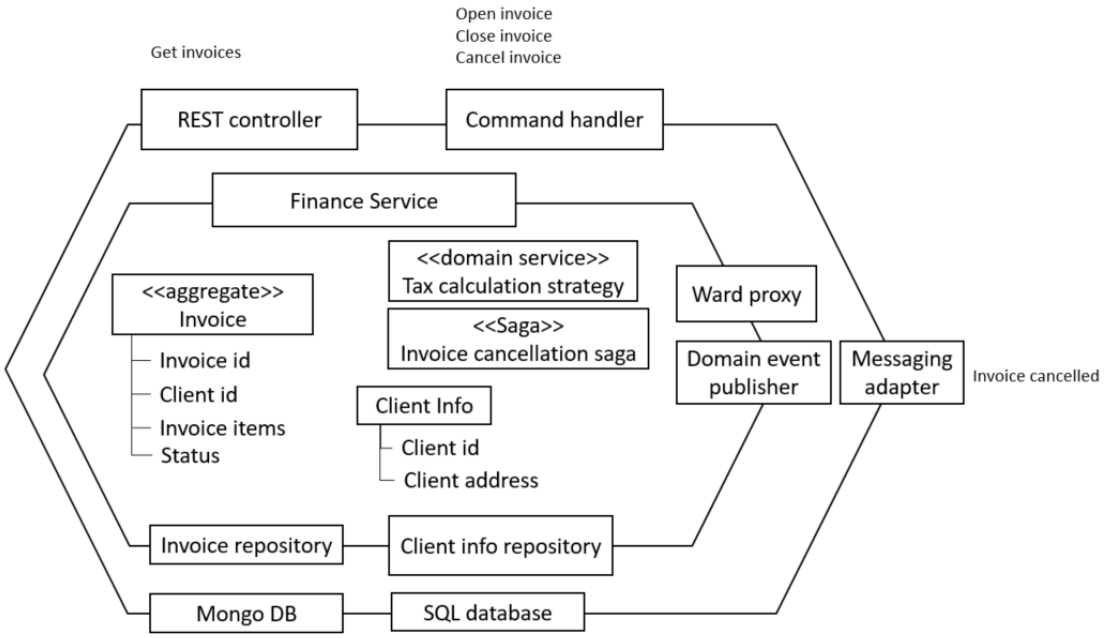

Figure 2: Detailed service diagram