Bloom Project Documentation

System Analysis

Requirements

“A hospital keeps the records of its patients including the registration details (name, id, age, abd DoB (date of birth)) of the patient and the fee payments. The entry of patients is determined whether he/she has arrived in emergency, OPD or for a routine check-up. The patient who gets admitted is provided with a room, specified by type and unique number, according to his/her choice. The fee payments include information such as amount, date, and payment number. The patient is allotted a doctor according to his illness. The hospital also keeps the record (id, name, salary, and date of birth) of the doctors visiting the hospital and other permanent employees of the hospital. Each doctor has a few days associated with his/her visit to the hospital and also the timings when she/he is available in the hospital. Nurses are assigned to a specific shift (A, B, or C) and ranked from 1 to 5. The hospital maintains the record and amount of the inventory of the hospital such as the equipment, the medicines, and blood bank. The inventory must be managed by a nurse with rank 1.”

Description

Every hospital needs an easy to use, highly reliable, resilient, and efficient system to make sure they can get things done in a fast paced environment, operate any kind of data whenever they want securely and efficiently. Also, hospital systems need to grow as the hospital grows, sometimes it needs to grow faster to make sure everything can be handled smoothly in the future, so, the system also need to be elastically scalable to make sure the system holds in the long run, when the peek is high or low, and operate at the same level, in every step of growing of the system.In conclusion, we are going to build a hospital system that has those specifications, and features that will make things easier for the people who use it.

Goals

- Provide an easy to use website for both patients and hospital employees.
- Report, when needed, to the people in charge for better decision making.
- Highly Scalable, efficient, fault tolerant, and resilient system.

System Design and Architecture

Back-End

Description

TL;DR: The Back-End will be built with the Domain Driven Design using microservices and CQRS pattern, message broker, and the reactive programming paradigm.

Our back-end will handle how our system behaves, whether it's a user operation or what will happen when the system fails or the load is high, so it is important to make sure everything is going to happen as it should, provide consistency among every client, and build a system that will hold in the long run.

We will decouple our system into microservices, for many reasons, firstly, the development cycle will be faster since we have a small application to develop and debug, Loose coupling, so we focus on a small branch of the problem instead of a big monolithic application. Secondly, microservies provide a better abstract layer for the system as a whole, in addition to that, it also provides scalability; because of the fact that each microservice is fully independent from other microservices, and take ownership in all its operations and data models, so it can be scaled alone without worrying about others, and many more advantages that will play nice in our case. There are many problems in microservices, it's not perfect for all use cases, some of the problems are the same problems that are in distributed systems, which is networking, consistency, and more, CAP Theorem, but in our case we will use it for the advantages that I talked about before, and solve the disadvantages with very neat solutions.

We will need something to make the most use of system resources, which includes but not limited to, Threads, Memory, Networking and others; to make the system efficient, reliable, and resilient. How are we going to do that?, you might ask, well, Reactive Programming actually does what I just said, it provides a more efficient way to communicate over a network, more precisely, it is a programming paradigm that deals with asynchronous sequences of events. Events in our case can be a user request, a database transaction, or even just a single log message from one of the microservices, this way we can have non blocking I/O, Input and Output, which leads to a high throughput with a low memory footprint. Besides that, it will give the concept of flow control, Back Pressure, over the network, which will help both sides of the connection manage how much events that can produce or consume and control them in a very efficient way, to make sure the system holds the load.

So, I have been saying a lot of things, to conclude this, we are going to follow the Domain Driven Design: Domains are first class citizens in our use case. And we will do that using microservices, each will have its domain ownership so it will be fully independent from the others, the microservices will be communicating using a message broker to make the system more decoupled, and the Command–query separation (CQRS) pattern to separate the reads

from the writes in each microservice, and the reactive programming paradigm to provide both non-blocking I/O and back pressure to the system.

API

Our systems will handle client events, eventually, so we need a way to communicate. So, we'll provide a Representational state transfer (REST) API from every microservice, but this way the front-end will have many APIs to connect to, to solve this we will have an API gateway in front of all the APIs, and just route requests from the gateway to the appropriate microservice, and when the response is ready it will go from the microservice to the gateway and return to its client. But that is not the only thing that we'll take advantage of from the gateway, we will also have our authorization and security in it as well.

Message Broker

Why use a message broker? you might ask, the answer is easy, to provide loose coupling between our microservices, which solves some distributed systems problems, which are networking problems, data redundancy and consistency issues, as well as, joining data records on the network is quite expensive in every context, whether it's network bandwidth, latency, or consistency, so it will affect the whole system with the load that it cause just to join two or more domains. So the message broker will have the events that happened, and the microservices which need an event will consume it, in an asynchronous way of course, and process it in it's own resources. And I don't think there is, or at least right now, a better Message Broker than Apache Kafka, which also have other properties and features such as stream processing, which will help in our CQRS pattern. In addition, Apache Kafka has a lot of the specifications that our systems have, efficiency, scalability, and reliability.

Database

As I said each microservice will have it's own database, to follow the data ownership standard, and the domain of that microservice will be persisted in that database. So we will use the amazing open source database, PostgreSQL, which will play nicely with our specifications and microservices standards.

Domain

Domains, as I said, are first class citizens in our use case, so we want it to be as perfect as possible in order to make our system work as it should, and provide the insights that the clients need. To make sure we follow our microservices standards, each microservice will handle a specific type of a domain only, separation of concerns and data ownership, using its own database, which also provides a layer of abstraction over the system.

- ERR Diagram
- Relational Diagram

To be continued つづく