A microservices architecture model in the face of exponential growth

Author: Ezequiel H. Martinez

Date	Modification	Author
03/27/2020	Initial version.	Ezequiel

Table of contents

Introduction

Here we produced a proposal for a Movie database that could be fully customizable and scalable for the main public. The idea is to make it grow exponentially, if it may. For this, we have thought of an initial architecture that will entail a set of microservices that could be taken into a single Docker server and then move, progressively into a Kubernetes kind of arrangement.

Please, keep in mind that this is an “organic” kind of solution thought, initially, as a prototype that will test the market (an MVP) and then, progressively will be growing in a steady or explosive pace.

We have prepared the design with these regards in mind.

Assumptions

We suggest to complement this list with the one found in the README.md.

Schema less

We've assumed that this movie's database, could start like a "movie <-> actor" sort of relationship, with a simple set of attributes attached to each of those entities. However, we understand that this might evolve as the market and the users demands it. They could, as well, require us to add an infinite sort of attributes to describe the nodes or newer relationships between those.

Huge Scaling growth

We understand that this API will serve the general public; so, the potential lead in hits could be in several orders of magnitude very fast with respect to those that we could've find in a regular b-to-b kind of system.

No market willing knowledge

We want to allow the user's base to define how and what sort of data they might add to every entity; so we assume that the systems growth should be organic. For that matter, we've thought of a kind of architecture that starts with a simple MVP/prototype that could be scaled.

There will be no design for Services Orchestration neither Choreography

We understand that in our design, and given the present potential growth of those services, we will be dealing with a "cloud" of services interacting with each other. We leave this issue out of scope for this design; however, we'll be presenting a few words over the subject and a recommendation in our "**final words**" section.

First concept and initial MVP

We thought the first app to rollout as a singe “*standalone* java server REST API” running in a docker server as a docker’s image. The database will be Neo4J, a graph database that is schema less.

Figure 1 shows the distribution of the different API & internal DB into Docker images

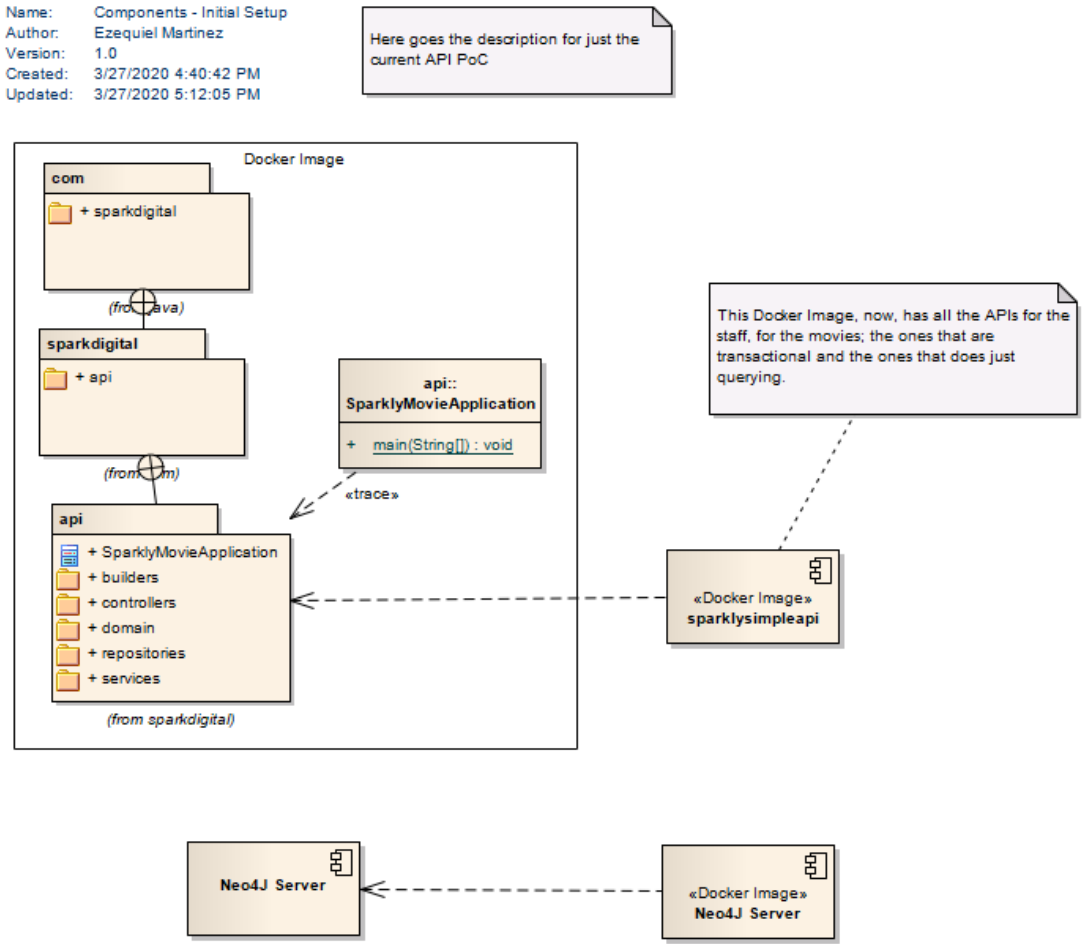

FIGURE 1

We will run the **Neo4J Server** in a separate Docker’s image, and the API in its own image as well. Both will be in separated containers, but will communicate between them (sockets will be exposed).

Figure 2 shows how the first deployment will be rolled out. It will work as an MVP.

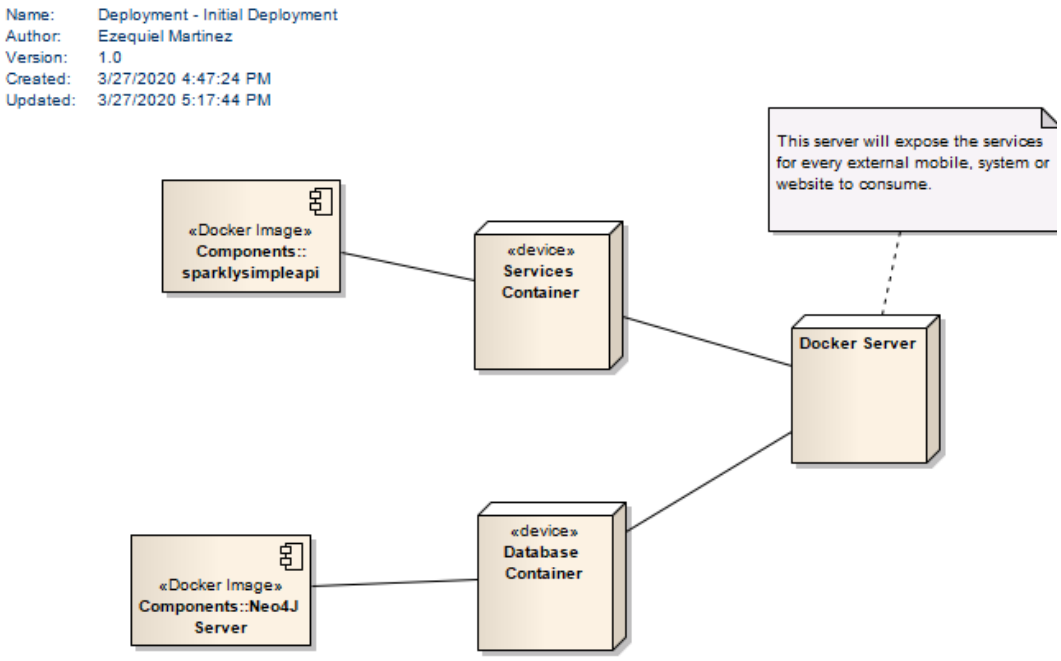

FIGURE 2

But this architecture, being sound in a “first steps service api” will not be possible if we do not take special care into its inner services design. We went for a microservices kind of architecture where every service will be as atomic as possible and they will be Restful (with all and the JSON return types).

We believe that those services should be exposed as standalone applications that will went into its own docker images.

Figure 3 shows how the “front end” controllers (the REST API itself) is intended to be implemented.

This idea will “pollenize” the system’s philosophy for scaling when it grows. We aim for, at the very end, having a bunch of services who are logically tied together into a single traceable docker image.

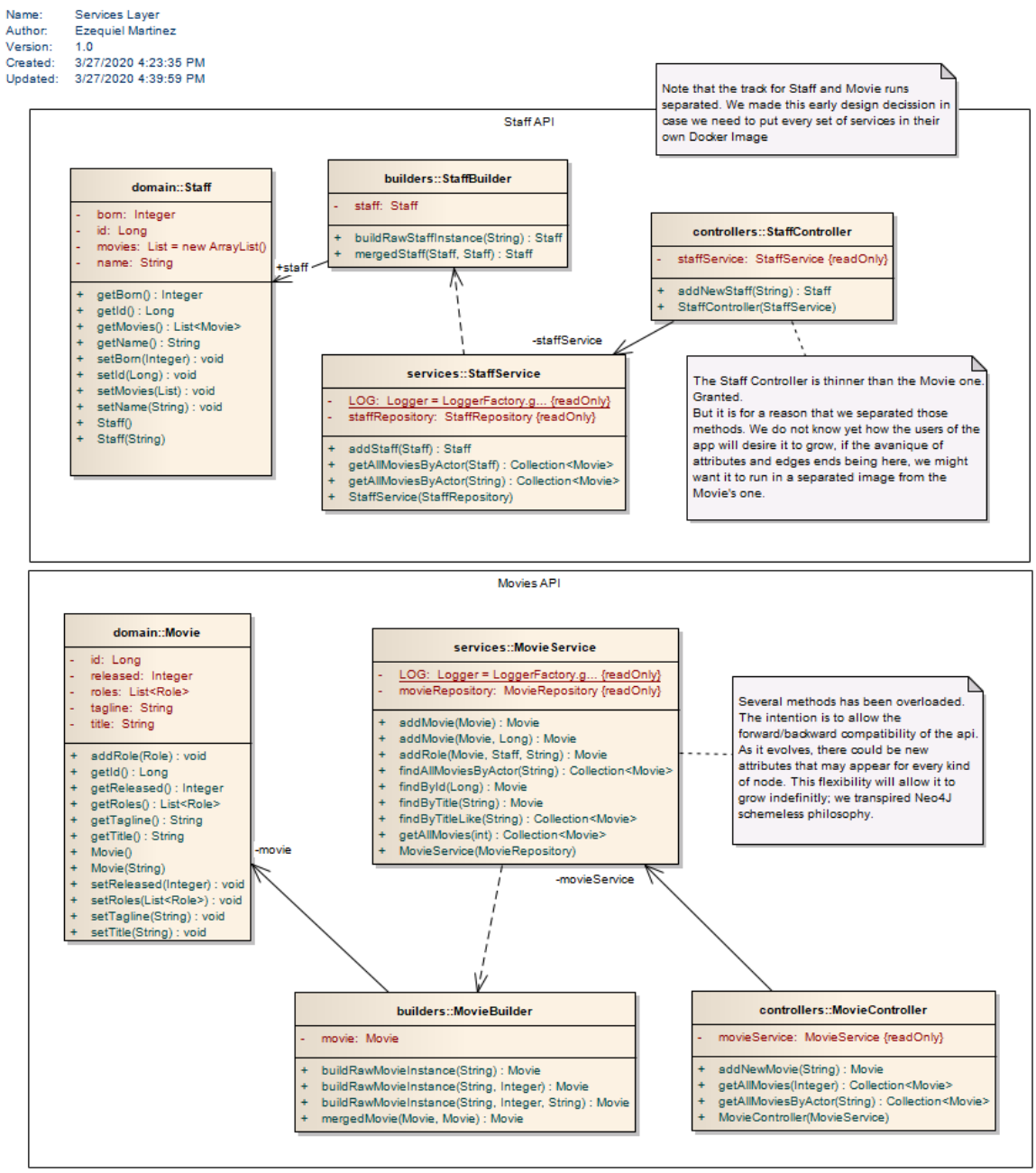

FIGURE 3

The key idea here, is that even if initially we have all the services in the same image, we'll design them to be as independent as we can. So, in a future, the architecture could support to have even "one image by service[1]". Given that, if we face a scenery at which we already have devices, web sites or even third party consumers linked to these defined methods, we will not have to

[1] Which at the end, will be a stand alone app. That's why the dockerization works so well here.

disconnect them from our API if the REST interface needs changes. We'll just have to add new containers for the new docker images, which are new services with the very same interfaces[2].

If we do not do this, we will permeate the forward compatibility and the backward compatibility for the API in the face of changes. Like the ones we might find if we are adding new kinds of entities (nodes in graphs), relationships (edges) or simple attributes.

A few other things to take into consideration:

- The interaction with the API will be token based; we didn't model this, because it's trivial to resolve it in a RESTful sort of API.
- There should be logging in place and a proper storage for it.
- We will deliver the API version within a part of the web services response.

In **Figure 4** we describe how we propose to interact with the backend repository for data access; that's Neo4j's graph structure with which we will be dealing with. We propose to handle several abstractions and do full usage of Spring Data drivers.

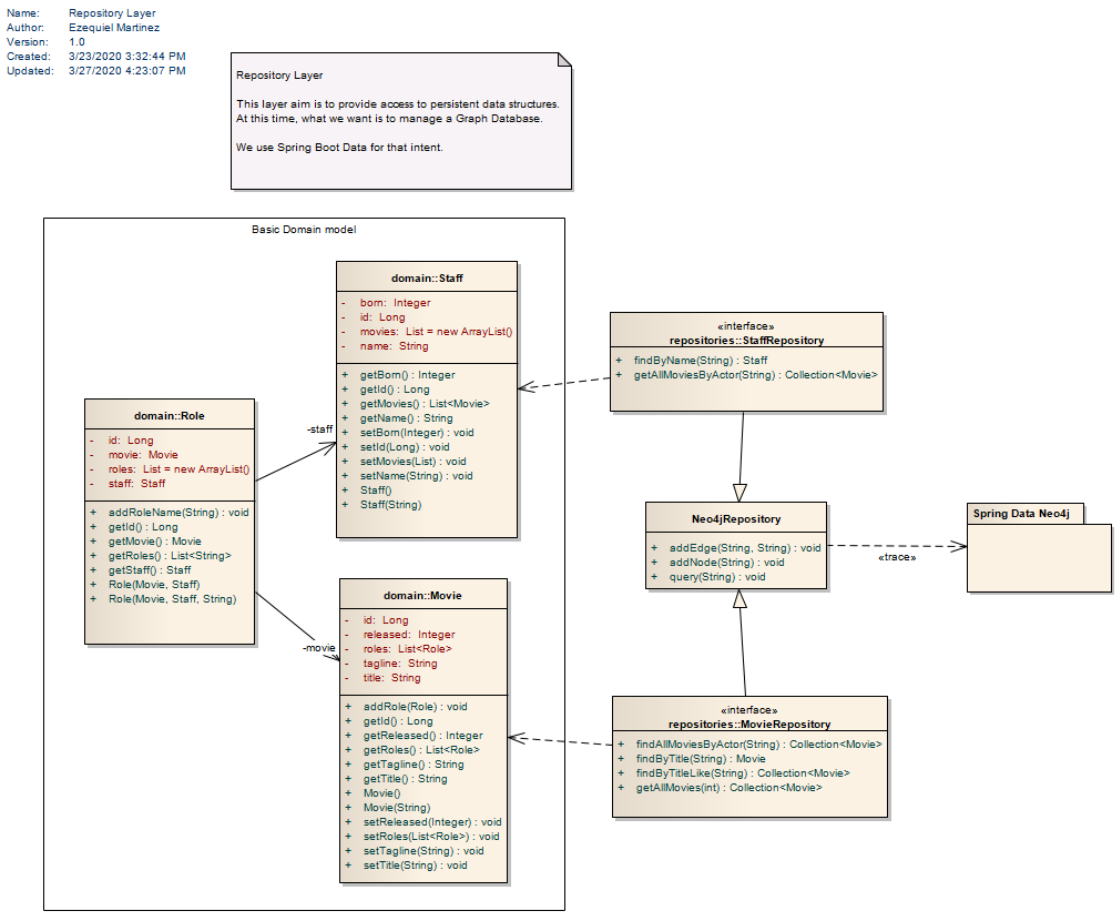

FIGURE 4

There are many key concepts here, the main one is to use domain classes that maps (OGM) to the original graph. This structure should mirror the kind of queries that the final users might require. We could have a different API for every sort of data structure required by every sort of

[2] We well need some sort of orchestration there, true. But we leaved that out of scope; read a small note in the **final words**.

specific consumer. However, we recommend to sustain the same sort of data exposure consistently and to not customise any external interface[3].

This model will enforce the service to take a hand on the match in front of it and resolve it, while it aims for the possible changes in the environment and business. Our main concerns has been:

- Ensuring the schema changes.
- Allowing the services to be fully manageable from the outside and fully recoverable.
- Atomic services that resolves just one specific business item, when possible.

[3] If we can. Sometimes this is simply not possible.

“Goto” architecture and “why”s

The next step is to think big.

What would we do in the scenario of an explosive grow in users, functionality and requests? In this section we will explain how to deal with “may be too many users”, “maybe too many new features” and “multi devices”.

Figure 5 may look too small for it to be explored, but the image is such that can be zoomed.

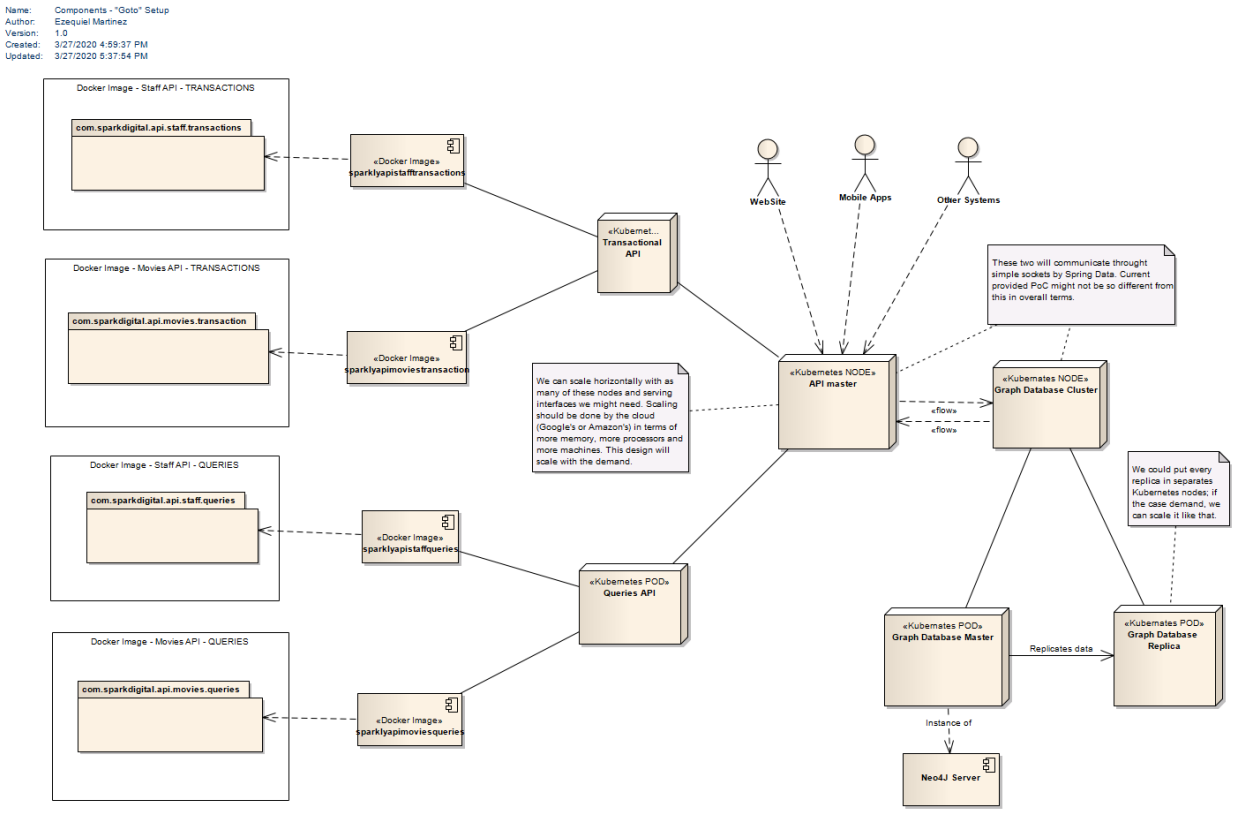

FIGURE 5

The aim is to solve most of the API growth is to design it as a Kubernetes bouquet of services; understanding the neo4j servers as a set of services intended for our inner usage.

To explain the final architecture, we need to imagine every service like a responsibility that we need to fulfil. Those responsibilities, could be tied together into one single API, if their functional and non-functional requirements binds them. In this design we identified these two:

- Transactional APIs vs Querying APIS.
- By kind of entity.

This division is sensible when we go for the party of how much querying we could have and how many new data we might face. And we could have different load balancing there. So, if we could

dockerize them as atomically as we can; maintaining the same URLs, JSON responses and parameters for the clients, we going to be able to grow almost indefinitely[4].

The main idea here is to have a docker image for one single web service API[5] (with a sizeable amount of REST interfaces, of course) if the load requires it; and get those dockerized images into kubernetes pods that could be monitored, scaled and replicated in different sets of clusters. We'll initially go for pods arranged into nodes that are just for services. Those nodes will communicate with other nodes which sole purpose is to handle the database servers that will be consumed internally.

Having this into account, we see that we could add new nodes and scale horizontally or vertically as much as we need if we go for an "on demand" kind of deployment like the ones provided by Amazon or Google.

---

[4] Of course, this comes with a cost. Allowing over-customization, has always taken a toll over systems performance. So, we'll have to supply it with the due infrastructure; as we will see.

[5] We'll try to avoid having one single service in one single docker image; but if the matters comes to worse, the designs might allow it through a small amount of refactoring.

Final words

On Golang

We understand that as a part of this exercise it was required to provide with a Golang kind of PoC.

What we did with Java and Spring Boot, was to provide a stand alone application; which will allow us to dockerize it and expose the services alone. This is great for trying to keep everything in boxes and to ease the way in which we deploy the services and make their environment grow. The idea, is to have one api, in one docker image, and for it to run as stand alone as possible. Ideally, we could go for the minimal, most atomic set of services.

In this regard, the main concern that we face is how we do this in Golang:

- Give a stand alone server less kind of architecture to a Golang web app.
- Generate a REST API in Golang
- Make the connectors to Neo4J as an ORM (Data Mapper), Active Record or even Record Set.

If we could do this, all the rest of the architecture remains *as-is*[6]. Allowing us to deliver the same amount of flexibility and scalability that Kubernetes and docker might deliver in a microservices kind of architecture.

Dealing with growth complexities and multiple services at once

The main idea will be to go with **Kafka** and services choreography. I'll not get deeper into the details, but we could land for several **topics** and queue them in streamlines of the like:

- "Adding new movies"
- "Adding new studios"
- "Modifying actors and roles"
- "Retrieve a cloud of actors for a movie plus their twitter's current status"
- etc.

And other "business" kind of matters that might require the combination of several services and perhaps some external ones. Given that the vast majority of the services we are going to dealt with are in "our side", we foresee that using queues could be the way to go. However, in the face that some of the workflows might become too complex or too pervasive over the services status quo, we will go with an Orchestration solution incidentally (Like JBPM). But we do not believe that might be required given the present requirements.

[6] Being the heart of the subject, this might not be trivial. But the good news is that we could end having part of the services in Golang, part in Java or C# if they provide a way to build a standalone app which exposes services within the same interfaces.