Microservices with Spring Boot — Intro to Microservices (Part 1)

A gentle Intro to Microservices

OMAR ELGABRY Follow
Jun 11, 2018 · 4 min read

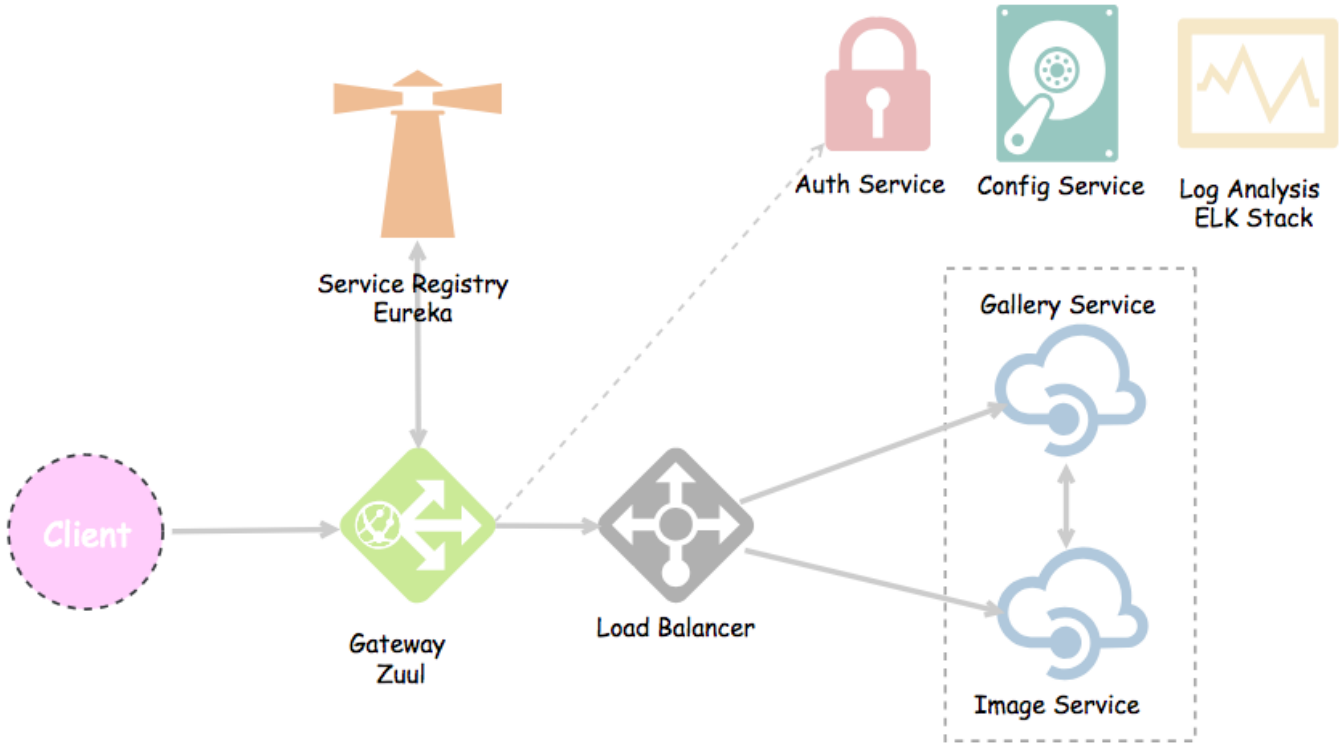
“Microservices, in a nutshell, allows us to break our large system into a number of independent collaborating components.”

Spring Cloud — *which builds on top of Spring Boot*, provides a set of features to quickly build microservices. It’s very smart to know how to get them working together, can quickly setup services, with minimal configurations.

Things like service registration and discovery, circuit breakers, proxies, logging and log tracking, monitoring, authentication, etc.

We'll walk through the code on how to integrate the common features of Spring Cloud, and we'll explain each of them. Excited?! So, let's get started.

> *This tutorial also assumes you're already familiar with the basics of Spring Boot.*



Eureka — JWT Auth — Zuul — ELK

In this series of tutorials, we are going cover:

- Introduction to Microservices
- Creating our Microservices & Gateway— *Eureka and Zuul*
- Authentication — *JSON Web Token (JWT)*
- Circuit Breaker & Log Tracing — *Hystrix & Sleuth*
- Managing, Searching, and Visualizing Logs — *ELK (Elasticsearch, Logstash, Kibana)*

> *The last part "ELK (Elasticsearch, Logstash, Kibana)" won't be covered here. I added a link to another simple step-by-step guide. It's worth adding it as part of this series of tutorials.*

UPDATE:

🤼‍♂️🍕🍕💀 The GitHub repository for the demo application: https://github.com/OmarElGabry/microservices-spring-boot

. . .

Before getting on how to start building the microservices, let's just have a gentle introduction to microservices.

Microservices

Well, there are many definitions out there, and probably you'll get distracted. The next definition combines the most common concepts of Microservices.

> *A microservice is an engineering approach focused on **decomposing** applications into **single-function** modules with **well-defined interfaces** which are **independent**ly deployed and operated by **small teams** who own the **entire lifecycle** of the service.*
>
> *Microservices accelerate delivery by **minimizing communication** and coordination between people while reducing **the scope and risk of change**.*

Decomposing

So, instead of having one large application, we decompose it into separate, different, mini-applications (services).

Each service handles a specific business domain (logging, auth, orders, customers) and provides the implementation for the user interface, business logic, and connection to the database.

```
class UserApp {
  User getUser() {
     // 1. auth user
     // 2. get user data
     // 3. log user actions
   }
}
```

```
class UserApp {
  void authUser(User user) { ... }
  User getUserData() { ... }
  void logUserActions() { ... }
}
```

Single-function

Each and every service has a specific function or responsibility. And yes, a service can do many tasks, but all of them are nevertheless relevant to this single function.

Well-defined interfaces

Services must provide an interface that defines how can we communicate with it. This basically defines a list of methods, and their inputs and outputs.

Independent

Independent means services don't know about each other implementation. They can get tested, deployed, and maintained independently.

It might be the case where services are implemented using different language stacks and communicate with different databases.

But that doesn't mean they don't work together. They do, in order to complete there required operation.

```
class UserApp {
  void authUser(User user) {
    // log user login action (success or failure)
    // using logUserActions
  }
  User getUserData() { ... }
  void logUserActions() { ... }
}
```

Small Teams

We split the work up and team across the services. Each team focuses on a specific service, they don't need to know about the internal workings of other teams.

Those teams are can work efficiently, communicate easily, and each service can be deployed rapidly as soon as it's ready.

Entire Lifecycle

The team is responsible for the entire lifecycle of the service; from coding, testing, staging, deploying, debugging, maintaining.

In a traditional application, we may have a team for coding, and another one for deployment. In microservices, that's not the case.

Minimizing Communication

Minimizing communication doesn't mean that the team members should ignore each other. It means the only essential cross-team communication should be through the interface that each service provides.

They all need to agree on the external interface so that communication between services is clearly defined.

The scope and risk of change

Services should be changed without breaking other services. And so long as we don't change the external interface there will be no problem for other services.

As a result of changes, the versions of services are updating individually, and there is no relationship between them.

. . .

Interested about characteristics of a microservices? Here's a talk by Martin Fowler, and a text version of a video. He explains the most common characteristics should have.

. . .

Monolithic Vs Microservices

The choice between the two approaches depends on the context, and complexity of the application.

Indeed microservices solves problems occurs in the large application from scaling, managing, but it's not always the way to go.

It is important to remember that microservices might be used in contexts where they are not meant to be used, resulting in extra effort and cost, project failures.

Most of the problems in Microservices are inherited as a result of having separate components.

For example, communication between methods in monolithic is much faster when compared to services asycnrousns communications, which slower, harder to debug, and must be secured.

And for sure, there will be an extra effort for operations, deployment, scaling, configuration, monitoring and testing as each service is separate.

For that being said, we need to have a skilled DevOps team to handle the complexity involved in deployment and monitoring automation.

Ok. So, we're done with the Intro to microservices. Next, we'll start implementing our microservices.

. . .

Thank you for reading! If you enjoyed it, please clap 👏 for it.

Microservices Java Software Development Programming Other