Microservices

Table of Contents

[1 Microservice Architecture 2](#_Toc87788303)

[2 Microservices Level 1 - Communication 2](#_Toc87788304)

[2.1 SOA vs Microservice 3](#_Toc87788305)

[2.2 Movie Catalog API Application 3](#_Toc87788306)

[2.3 Step 1: Create the 3 Services 4](#_Toc87788307)

[2.3.1 Using a shared library for the model class 6](#_Toc87788308)

[2.3.2 Creating a communication channel between our projects 6](#_Toc87788309)

[2.3.3 Avoid returning a List as your top level API response 7](#_Toc87788310)

[2.4 Service Discovery 7](#_Toc87788311)

[2.4.1 Why hard coded URLs are bad? 7](#_Toc87788312)

[2.4.2 Client-Side Discovery 7](#_Toc87788313)

[2.4.3 Server-Side Service Discovery 8](#_Toc87788314)

[2.5 Eureka 8](#_Toc87788315)

[2.6 Implementing Eureka Service Discovery 9](#_Toc87788316)

[2.6.1 Consuming the Discovery Service 11](#_Toc87788317)

[2.6.2 Is this a good approach? 11](#_Toc87788318)

[2.6.3 How Fault Tolerance Works – “Heart Beats” 12](#_Toc87788319)

[3 Microservices Level 2 - Fault Tolerance and Resilience 13](#_Toc87788320)

Resources:

* Javabrains: Microservices 1 Playlist
* Javabrains: Microservices 2 Playlist

1. Microservice Architecture

TODO

1. Microservices Level 1 – Communication and Service Discovery

Traditionally we built monolithic apps. There may be one or more code bases but at the end of the day what gets deployed to the server will be one app. It runs on one server, and it scales as a single entity. On the coding side you can have many projects, you can have all the modularity but you will still end up one monolith deployed on the server.

With microservices this will change. You can still follow the same patterns with coding but what gets deployed and what happens at runtime is very different. This effects the way that we write code and deployment, also need to think what other challenges you might have.

In this course we will create a few microservices and have them communicate with each other. You can either hard code in the microservices the other microservices which it can talk or you can be more elaborate and use service discovery and have microservices discover each other. You can do a whole lot of other things with microservices as well.

Some of the challenges:

* Where to start? Spring Cloud is just one way to create microservices, one set of technologies. Just in Spring Cloud you have a lot of buzzwords, Greek names etc.
* Lots of patterns
* Interdependent concepts and we need to use a lot of those

**Why?** Why there is this complexity, why do we have so many technologies involved. Because we have one big chuck of code which forms your application and you break them into smaller pieces.

You are solving scalability, modularity of deployment, meaning you can make changes to one portion of the application without having to redeploy the whole thing. So, these are some of the advantages. But then you have a whole lot of new challenges you need to solve. Because you have this modularity you have to make sure your release process is working. Because of the scalability you have to make sure your microservices can scale and can have multiple copies and it will still work.

So, you solve some problems but with those solutions comes new problems.

One difference is that with monolith apps you have specific problems about your domain. For example if you are building an e-commerce app, your problem set would be: “How do I make sure my shopping card service/logic is called from my catalogue service?”. It is very specific to your domain which means you have to solve those problems for each of the monolith you build.

But if you build it with microservices it becomes a more generic problem. Load balancing is a generic problem. No matter what problem set you have, as long as you broke them down as microservices, it becomes a common problem across all domains/problem sets. If it’s a common problem it means that you can use frameworks, patterns and conventions to solve them more easily. You will have proven patterns and proven technologies no matter what application you are building.

Diagram

Description automatically generated

* Complexity hidden withing monolith vs complexity between microservices

**Text, letter

Description automatically generatedManaging this complexity**: Service discovery is a pattern which solves the question “how should small parts of your app communicate with each other?”, “How do you have them discover whom to communicate with?”

And you have technologies which allows to work with those patterns like **EUREKA**.

* 1. SOA vs Microservice

A lot of service-oriented architecture (SOA) concepts are around creating utilities, programs which the creator does not know exactly where it will be used. Like an IP discovery service. A soap endpoint where you pass an IP and it returns the location of that IP/server. So the service does not know where it’s gonna be used, whoever wants to use it can use it. Similar concept -> web services

Microservices is not like that. You have an idea about what the application is and who will use the smaller parts. If you want to build an e-commerce app and you split your app into microservices, you know very well which part will use which microservice. Sometimes even a microservice can be used only by one other entity and that is completely fine. You are not intending it to be reused. Can it be reused? Yes, but not a requirement.

* SOA/SOAP services had a very strict contract because you had to provide the same interface for all the clients and you did not know who were using it but microservices does not have/need that.
  1. Movie Catalog API Application

In this course we will use **Spring Cloud** (one of the most robust way for building microservices). But there are a hundred different ways how you can build microservices.

This will be a **Good Reads** clone but for movies. We will build 3 microservices. 2 microservices which provides different kinds of data, the third will call those other 2 to get the data processes it and sends back a response. Generally, you will see this kind of behaviour. Use lots of parts and then consolidate/aggregate all the data and return back one response.

Text, letter

Description automatically generatedLet’s imagine a JS developer which wants an API from you. He is building a UI where you say **example.com/userid** and it will pull up all the movies this person watched and rated, movie name, description and rating. We will give back a response with a list with each element being the movie name, movie description and how the user rated it.

No database, no **PUT**, **POST** and **DELETE**. We will hardcode a lot of thigs and just focus on the microservice logic and not the spring boot logic.

* movie-catalog-service
* ratings-data-service
* Diagram, schematic

  Description automatically generatedmovie-info-service

So first our movie-catalog-service will call the ratings-data-service with the userId to figure out what movies this guy watched and rated.

After that, for each movie in the response it will make a call to the movie-info-service to get more information about the movie

1. Create 3 Spring Boot projects. Each of these microservices will be a separate project.
2. First, we will build the movie-catalog-service API. After that the movie-info-service API and lastly ratings-data-service API.
3. Have movie-catalog-service call the other two services (the naive way / hard coded)
4. Implement a better way (Service discovery)

It is also possible for the clients to call individual microservices directly.

Not all microservices need to be Spring Boot applications. Since they will be communicating with REST, each microservice can be implemented with a different technology as long as it speaks REST. There are other technologies we could use. It a bit of preference but Spring makes it also easy.

* 1. Step 1: Create the 3 Services

I’ve created a parent pom project and created the 3 microservices as modules and one model module which contains the model classes which are shared across modules.

Text

Description automatically generated with medium confidenceGraphical user interface, text, application

Description automatically generatedA picture containing table

Description automatically generated

We have the Spring Web dependency which comes with **tomcat**. This way when we start our applications they will continue running and keep listening on their defined server.port. Without Spring Web, the Spring Boot applications would exit after they started.

Graphical user interface, text, chat or text message

Description automatically generatedText

Description automatically generatedGraphical user interface, text

Description automatically generatedText

Description automatically generated

A screenshot of a computer

Description automatically generated with low confidenceText

Description automatically generated

Graphical user interface, text

Description automatically generatedGraphical user interface, text, application

Description automatically generated

Text

Description automatically generatedGraphical user interface, text, application

Description automatically generated

Text

Description automatically generated

We use the ports 8081, 8082 and 8083 for our apps.

* **In production** you rarely have microservices running on the same machine, so the port for all apps could be 8080 since they will be deployed to different machines. Each spring project is running its own instance of tomcat.
* **In production** we would also run our application like this. We would start the jar file. If we need to configure tomcat for example, we can do that with the application.properties file as we just saw. We changed the port of the app which means configured tomcat.

A good practice is to not run any application on 8080. Pick other ports, so if for some reason you will need 8080, you won’t need to configure anything.

* + 1. Using a shared library for the model class
* Having multiple copies of the model classes are fine in microservices because you want the microservices independent. If you have a shared library for your model classes, then you can’t really change it independently and have to coordinate it with the other teams etc.
* If one microservice has more information which it needs in that model, it can just put it there without the fear of breaking anything.
* If you add a new field to a model for example, it is fine since it won’t affect the consumers. But if you change an existing field or an endpoint then things will break and this is where **versioning your microservices** comes into play. You either let people/other teams know of that change or you create a new version /api/v1/… and /api/v2/…
  + 1. Creating a communication channel between our projects

We are making the catalog and info services talk to each other. We are using a REST/HTTP client library. Spring Boot comes with a client built in; it’s called **RestTemplate**. This is what we are using to make REST calls.

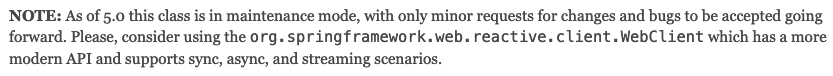
But it is on its way of being **deprecated**! There is another way of doing it called **WebClient**.

RestTemplate is easier compared to WebClient. Because **WebClient** uses reactive programming. An asynchronous way of programming in java (RxJava), providing callback functions meaning “call this function when it is ready, I’m not going to wait for you” and you provide a lambda or function.

* RestTemplate is **synchronous** and **blocking** i.e. when you do a rest call you need to wait till the response comes back to proceed further.
* WebClient is complete opposite of this, **asynchronous**. The caller need not wait till response comes back. Instead he will be notified when there is a response.
* RestTemplate is still widely used but WebClient is the future.

Text

Description automatically generated



* + 1. Avoid returning a List as your top level API response

Returning an object as the top-level node in your API response is generally better because you can avoid these:

1. **Breaking the API Contract**: If in the future you want to add another field to the top level API response, then you must break the contract to do so. But if you return an wrapper object which just contains a list of items, you can then add more fields to it in the future without breaking anything. The consumers (in most cases) won’t need to change anything and their code will work
2. Diagram

   Description automatically generatedRestTemplate **Un-marshalling**: We need to pass a class to RestTemplate so the response can be cast to that object but it is not easy to pass a list.
   1. You can either use something like ParameterizedTypeReference, which is not pretty, or,
   2. Un-marshalling to an array of that type and then converting that array to a list.
   3. Service Discovery
      1. Why hard coded URLs are bad?

* **Changes requires code updates / new deployments**
* **Dynamic URLs in the cloud**: When you deploy something to AWS for instance, the URLs will be dynamic and will change often.
* **Load Balancing**: When the demand is high and we run multiple more instances of the same microservice, how can we do load balancing if we have hard coded URLs which point to only one of the instances?
* **Multiple Environments**: URLs for Local development environment, test and prod
  + 1. Client-Side Discovery

Because of these reasons we need to use service discovery. This is another pattern for building microservices.

So let's say you have one client which is our movie catalogue service and you have three other services that it needs to consume. One of those three it needs to consume. How does it locate/discover something? So what's the minimum thing you need to have to enable the service discovery? What would be the first step? Let's say you're doing this yourself.

You have to provide **a layer of abstraction** in between incoming requests and the servers, this layer will be in charge of forward those requests to your services. The client will first call that **discovery server** and ask for a URL for a particular service to then send the request to the best available service. Now the question is, again, how does the client discover the discovery server? **TODO:** lets assume somehow it knows. (17 - 3:45). **Note** that the client in these examples is a Spring Boot client.

What happens is,

1. you start your discovery server and
2. each of those services which wants to be discovered registers on the discovery server.

A picture containing text, businesscard

Description automatically generatedSo once every server it's registered, it's all good to go. That discovery server and knows where those services are and the client talks to the discovery server and it tells what service it needs (for example the name of the service). The discovery server says “sure, here it is” and it provides the address, the link to where that service is and then the client can call it.

Diagram

Description automatically generatedDiagram

Description automatically generated

The **disadvantage** here should be obvious:

* it's a little bit chatty: rather than having the client make the call directly to a service, there is an additional step that needs to happen to discover the service.

So, there is a hop, response and then the actual API call. There are ways to mitigate that we'll be covering that in a bit.

* Who's doing all the work to discover the server? it's the client. This is called **client-side service discovery**. This is the first of the two models for service discovery.
  + 1. Server-Side Service Discovery

The other model is doing all the work on the server side.

You still have the phonebook it has a registry and everybody registers to that discovery server

Diagram

Description automatically generatedDiagram

Description automatically generatedA picture containing text, businesscard

Description automatically generated

The client, instead of asking the address of a service, it tells the discovery server “Can you send my request to Service 2?”. It doesn’t contact Service 2 directly. Technically this is not a discovery server anymore. It’s more like a message passing service. It's a service in charge of passing the message to the right server.

The advantage here is that there are **no extra hops**.

Both models are very much valid. Each one has its own advantages and disadvantages. The model that **Spring Cloud** uses is **client-side service discovery**. When you create an application using **Spring Boot** and **Spring Cloud** and then you say “I want to use service discovery”, the libraries that facilitate service discovery will reside on the client. That's not to say that there isn't something in middle. You need that level of abstraction, you need a discovery server that's a given.

In fact the way it works with spring cloud is, you don't have to do any of this stuff. All the work that the client has to do, calling the discovery server, getting the services and their endpoints, parsing through and then sending the request to the right server. You just have to put configuration and then everything happens behind the scenes. It's almost scary how much of this works automagically.

* 1. Eureka

The technology which implements **client-side discovery** is **Eureka**. Spring cloud uses it and integrates it well in its system and it is very often used. Eureka was made open-source, thanks to **Netflix** OSS. Netflix is one of the leaders in micro service libraries and they made some open source software that works well with spring boot. Some of the project they created are **Eureka, Ribbon, Hysterix, Zuul** etc. The projects started because Netflix needed such solutions and then they made them open-source. They are pioneers when it comes to microservice architectures.

These technologies are then integrated in Spring. This is what Spring does very well. Spring has layers of abstraction over technologies that work well together but spring builds and abstraction so they you don't have to worry too much about the technology. Kind of how it works with **JDBC**. The **spring data** project, there is an API which allows you to use data interactions that are provided by spring but you don't mess with JDBC you use the spring abstractions. This makes the interface, the API much more developer friendly and another advantage is that you can possibly change the underlying technology in the future if your needs change. Let's say you're using hibernate, you can probably change that with some other JPA provider. The spring code is gonna be the same, you just change the configuration and then spring is going to manage the rest. Spring provides a layer so that you're not aware of, you don't have to be aware about what the technologies is used under the hood. Spring the spring community saw these open source technologies and said “hey this works well in the micro service context”, so they built wrappers around this.

Currently we have two services, we have the **ratings data service** and the **movie catalog service** and we are hard-coding the URLs. So, we're goingt use **Spring Cloud** and **Eureka** to discover those URLs instead of hard-coding them.

* We are going to start up the **Eureka Server** and our individual micro services will be **Eureka Clients.** Eureka clients register with the eureka server.
* The clients which consumes, meaning which want to send requests to particular services with the help of the discovery server also has to be a **Eureka Client**. Because it needs to ask the Eureka Server about the address of other services.

This one has to be eureka client as well by the way the thing that's consuming has to be recommend to this eureka clan is doing two things one it's telling the eureka server hey I'm here let me know if somebody needs me and then eureka client can also talk to the eureka server if it needs something like I need this service so it serves two purposes the rocket line serves two purposes the steps involved in making this work is to start eureka server obviously have micro services register using v-ray cut line and then have micro service located using the same array cut line talking to the same you reckon server.

Our goals are:

1. Start up a Eureka server
2. Have microservices register (publishing) using Eureka client
3. Have microservices locate (consume) other microservices using Eureka client
   1. Implementing Eureka Service Discovery

* Application which will be the Eureka Discovery Server -> needs the **Eureka Server** dependency
* Application which uses/consumes -> needs the **Eureka Client** dependency

Text, letter

Description automatically generatedText

Description automatically generatedI’m downloading a new spring boot project from Spring Initialzr. This is a simple spring boot project. With only the dependency on the classpath.

You might get some errors when starting this application if you are running it with Java 11. Since Jaxb was removed with Java 10, it can be possible that it might throw some errors.

Graphical user interface, text

Description automatically generatedWe need to do 2 things:

1. The @EnableEurekaServer annotation is used to make your Spring Boot application acts as a Eureka Server
2. Text

   Description automatically generatedBy default, the Eureka Server tries to register also as a client. Tell the application to not act also as a Eureka client. (and change the port)

Graphical user interface

Description automatically generated with medium confidenceAnd the application started under **localhost:8761**

Every **Eureka Server** is also a **Eureka Client**. When the server runs, not only it provides a registry, it also tries to register with other Eureka Servers since you can also have multiple instances of Eureka Servers. So that they can register with each other and in case of a failure, the other servers can process the requests.

Now we have to add **Eureka Client** dependency to our microservices and make them register their selves with the discovery server.

Putting the spring cloud version property to our parent pom and annotating our microservices. Note that I also had to add a dependency under dependencyManagement because mvn couldn’t find the spring cloud eureka client version.

Graphical user interface, text, application, email

Description automatically generatedGraphical user interface, text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Table

Description automatically generated

When we have an spring.application.name, we can now see the microservices registering in the discovery server.

* Up to a certain version the @EnableEurekaClient was mandatory but now it is not. It is enough if the client dependency is in the classpath. But the annotation makes it explicit that this is a eureka client.

**How did the Client find the Server?**

The Eureka clients look by default for **localhost:8761**. Since we specified the default port for **Eureka Server** in our **discovery-service** application, this setup works. If your eureka server is on a different port/server, you need to specify that in the configuration.

**TODO:** we then have to hardcode the discovery server URL’s or is there another way?

* + 1. Consuming the Discovery Service

The consumption will happen in our movie catalog service since there we hard-coded all the required endpoints.

* Graphical user interface, text

  Description automatically generatedWe can now tell RestTemplate to use service discovery and it will make all the necessary steps behind the scenes each time we want to send a request to a service. We just have to provide the name of the service.

So, in this case, the movie catalog service is our “Client” from our previous examples.

* We have to add the **@LoadBalanced** annotation.

Text

Description automatically generatedNow we can give the name of the service we want to call:

* **Important**: Now we **cannot** use the load balanced rest template to make normal requests.   
  A request to localhost:8082 will throw an exception: “No instances available for localhost”
  + 1. Is this a good approach?

With **@LoadBalanced** annotation 2 things happen:

1. Client-Side Load balancing
2. Service discovery

In this method, the client does the load balancing. If we had 5 of the same service, each time, with each request from the client to this service, the client gets the list of the available instances and has to pick one according to a load balancing algorithm. In this case it is (I guess) **round-robin** (an arrangement of choosing all elements in a group equally in some rational order). So, with each new request it picks another service.

**The problem** is: this is not effective load balancing. There might be another 100 clients wanting to send requests to the services. Each has to do load balancing. Since they are not aware of each other, there is technically a chance that most clients can send a request to the same service at the same time.

I’ve started another instance of ratings-dataservice from the terminal and passed some properties to see the different instances in the logs. You can see that it registered with Eureka.

Calling localhost:8081/catalog/66 twice showed that the ratings-dataservice is of each instance is **called ONCE** while the movie-info-service (which had only one instance running) was **called twice**. So, this looks like a **round-robin** logic.



A picture containing text

Description automatically generated





**Programmatic Access**

DiscoveryClient is available in the classpath. You can autowire it in your class and get for example all the instances if you pass a service id and has other methods. But it is not recommended, restTemplate should handle it. Only do it if you really know what you are doing.

* + 1. Diagram

       Description automatically generatedHow Fault Tolerance Works – “Heart Beats”

What happens if a service goes down? Let’s say our service 2 goes down. The client asks to get service 2, the discovery service says “Sure, here is the address”. But since the service 2 is down, the client won’t be able to access it. This problem occurs because the discovery service has no mechanism where it can know if the registered instances are still up.

* **The solution** is, sending “**heart beats**” to the service registry.

What a **Eureka Client** does by default is to ping the **Eureka Server** on a regular basis and send out heartbeats, sating “hey, I’m still here and alive”. The Eureka Server has then a logic where it expects these pings every so often, else it will register the service as “down” and removes it from the registry. So, it is not a one-time discovery. The clients need to keep sending pings in order to stay in the registry, which is handled by default.

**What if the discovery service goes down?**

That’s where the cache comes into play. The client which does the call sees that it gets no response and then as a fallback, it takes the address which it received the last time it called it.

1. Microservices Level 2 - Fault Tolerance and Resilience

What is Fault Tolerance and resilience?