Microservices

Table of Contents

[1 Microservice Architecture 2](#_Toc81133781)

[2 Javabrains Microservices 1 2](#_Toc81133782)

[2.1 SOA vs Microservice 3](#_Toc81133783)

[2.2 Movie Catalog API Application 3](#_Toc81133784)

[2.3 Step 1: Create the 3 Services 4](#_Toc81133785)

[2.3.1 Using a shared library for the model class 6](#_Toc81133786)

[2.3.2 Creating a communication channel between our projects 6](#_Toc81133787)

Resources:

* Javabrains: Microservices 1 Playlist

1. Microservice Architecture

TODO

1. Javabrains Microservices - Level 1

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. Note that 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.

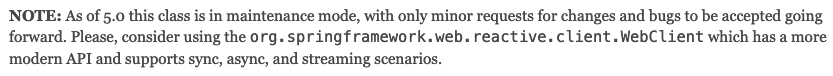
But it is on its way of being **deprecated**! There is another way of doing it called **WebClient**.

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 we run multiple 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

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