Microservices

[1 Microservice Architecture 3](#_Toc89714448)

[2 Microservices Level 1 – Communication and Service Discovery 3](#_Toc89714449)

[2.1 SOA vs Microservice 4](#_Toc89714450)

[2.2 Movie Catalog API Application 4](#_Toc89714451)

[2.3 Step 1: Create the 3 Services 5](#_Toc89714452)

[2.3.1 Using a shared library for the model class 7](#_Toc89714453)

[2.3.2 Creating a communication channel between our projects 7](#_Toc89714454)

[2.3.3 Avoid returning a List as your top level API response 8](#_Toc89714455)

[2.4 Service Discovery 8](#_Toc89714456)

[2.4.1 Why hard coded URLs are bad? 8](#_Toc89714457)

[2.4.2 Client-Side Discovery 8](#_Toc89714458)

[2.4.3 Server-Side Service Discovery 9](#_Toc89714459)

[2.5 Eureka 9](#_Toc89714460)

[2.6 Implementing Eureka Service Discovery 10](#_Toc89714461)

[2.6.1 Consuming the Discovery Service 12](#_Toc89714462)

[2.6.2 Is this a good approach? 12](#_Toc89714463)

[2.6.3 How Fault Tolerance Effects Discovery – “Heart Beats” 13](#_Toc89714464)

[3 Microservices Level 2 - Fault Tolerance and Resilience 14](#_Toc89714465)

[3.1 What can go wrong? 15](#_Toc89714466)

[3.1.1 Scenario 1: An instance goes down 15](#_Toc89714467)

[3.1.2 Scenario 2: A microservice instance is slow 15](#_Toc89714468)

[3.2 How Do We Solve the Problem of Microservices Being Slow? 16](#_Toc89714469)

[3.3 The Circuit Breaker Pattern 16](#_Toc89714470)

[3.3.1 When should a circuit break? 17](#_Toc89714471)

[3.3.2 Circuit Breaker Parameters 17](#_Toc89714472)

[3.3.3 What to do when a circuit breaks? 17](#_Toc89714473)

[3.3.4 Why Do We Need Circuit Breakers? 18](#_Toc89714474)

[3.4 Hystrix 18](#_Toc89714475)

[3.4.1 Adding hystrix to a Spring Boot application 18](#_Toc89714476)

[3.5 How does Hystrix work 19](#_Toc89714477)

[3.6 Parameters 21](#_Toc89714478)

[4 Microservices Level 3 – Configuration 22](#_Toc89714479)

[4.1 What and Why? 22](#_Toc89714480)

[4.2 Property files 23](#_Toc89714481)

[4.3 External Property Files 24](#_Toc89714482)

[4.4 @Value Annotation 24](#_Toc89714483)

[4.4.1 Property (key) is not defined 24](#_Toc89714484)

[4.4.2 Property key exist without value 24](#_Toc89714485)

[4.4.3 Giving a default value to @Value 24](#_Toc89714486)

[4.4.4 List or String[] 24](#_Toc89714487)

[4.4.5 System Property 25](#_Toc89714488)

[4.4.6 SpEL 25](#_Toc89714489)

[4.4.7 Map 25](#_Toc89714490)

[4.4.8 Constructor Injection and Setter Injection 26](#_Toc89714491)

[4.5 @ConfigurationProperties Annotation 26](#_Toc89714492)

[4.5.1 Nested Properties 27](#_Toc89714493)

[4.5.2 Using @ConfigurationProperties on a @Bean Method 27](#_Toc89714494)

[4.5.3 Property Conversion 27](#_Toc89714495)

[4.5.4 Property Validation 28](#_Toc89714496)

[4.5.5 Making it Immutable 28](#_Toc89714497)

[4.5.6 When to use which? 29](#_Toc89714498)

[4.6 Multiple Spring Profiles 29](#_Toc89714499)

[4.7 Using YAML vs Properties 30](#_Toc89714500)

[4.8 Spring Cloud Config Server 30](#_Toc89714501)

[4.8.1 How and where does this configuration service store the configs? 31](#_Toc89714502)

[4.8.2 Setting up a Spring Cloud Config Server 31](#_Toc89714503)

[4.8.3 Setting up the Spring Cloud Config Client 32](#_Toc89714504)

[4.8.4 Microservice Specific Configuration 32](#_Toc89714505)

Resources:

* Javabrains: Microservices 1 Playlist
* Javabrains: Microservices 2 Playlist
* Javabrains: Microservices 3 Playlist
* <https://www.baeldung.com/spring-value-annotation>
* <https://www.baeldung.com/configuration-properties-in-spring-boot>

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.

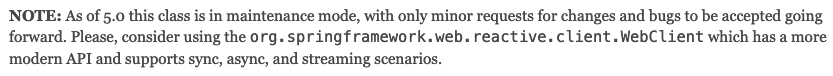
But it is on its way of being **deprecated**! There is another way of doing it called **WebClient**.

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 Effects Discovery – “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? These terms are used interchangeably sometimes but they are different things.

In the context of microservices:

* **Fault Tolerance** means, given an application, if there is a fault, what is the impact of that fault? How much tolerance does the system have for a specific fault? If one of your microservices goes down, does all your other services go down or won’t work anymore? Or will only the functionalty/part which that service provide go down? Or in the best case, is there a way where the impact will be handled so that there is no perceived impact?
* **Resilience** means, how many faults can a system tolerate? So, in the first case we are looking at a single fault, how the system will tolerate it. In resilience the question is how many faults can a system tolerate before it is brought down to its knees? A part of resilience is also how good a system can bounce back from a fault. If a service goes down for 10 minutes and then comes back up again, does everything work again as before?

You can have technically a system which is very fault tolerant but not relisient at all but that’s difficult to do. Most of the time these 2 terms go hand in hand.

**One change we will make**

Graphical user interface, application

Description automatically generatedWe want our movie-info-service to make an external call to **MovieDB** to get real information about a movie. It's a good example of showing how there are points of failure that can be introduced into the system by making external calls. The more external interactions you have the more failures you have. We want to simulate that.

I’ve created an account on MovieDB with a temp mail, **unuttum11**, with the standard easy and short passw and entered the v3 api key as a VM option in intellij.

Text

Description automatically generated with medium confidence

Text

Description automatically generated

This works when I call localhost:8081/catalog/66

A picture containing graphical user interface

Description automatically generated

Jackson is already on the path, it comes from one of the spring dependencies.

**IntelliJ** has an amazing feature where you can see a dependency graph and you can search through it with **CMD + F**.

Diagram

Description automatically generated**Is this microservice architecture resilient?**

This is what we have now and NOPE, our architecture is not resilient. If a microservice goes down, the whole app is not usable.

* 1. What can go wrong?
     1. Scenario 1: An instance goes down

It can happen that an instance goes down.

The most straight forward solution to avoid that our whole application goes down is to run **multiple duplicate instances** of the same service and run them on different machines.

* Running multiple duplicate instances on the same machine under different ports is **not recommended**.

Now even one were to go down, the other ones can handle the request easily using service discovery.

* + 1. Scenario 2: A microservice instance is slow

A microservice being slow is a **much much bigger problem** than if an instance will go down.

Let's see that external service **MovieDB** is slow. We shouldn't have picked it, that thing is slow. What ends up happening is as a result your **movie catalog service** becomes slow to process requests. But you might expect that other, non related requests will work fine and won’t be effected by the slow MovieDB.

* But that’s not the case! The MovieDB response being slow can cause for instance the Request **catalog -> ratings-data-service** to fail or to be slow. How can an unrelated service be slow because some other service, in some other corner of your microservice architecture is slow?

The answer is **THREADS**!

* With each new request to the Web server a new Thread is created to process that request.
* The thread does the work, processes the request, returns a response and goes away.

Diagram

Description automatically generatedSo what happens when a thread takes a little longer to finish and meanwhile other requests comes in? If the requests come at a faster pase than the threads can process them, then you will end up with a bunch of threads that are busy with processing all the requests.

What happens is that all the available threads will be used. You can configure the **maximum number of concurrent threads** in java servers. If you don’t configure it, the server will keep opening new threads and you will run out of system resources and **crash the system**.

Let’s say A and B are requests to other microservices and let’s also say that B is really slow and A is fast to respond. Since the requests to B will take much longer to finish and free up the thread, eventually with more requests coming in for B, all the threads will be used up by those requests. Now when a requests comes in for A, since there is no available threads to process that request, it has to wait for a thread to be freed.

* So, event though the request to A is super fast and no dependencies to B, it will still can be negatively affected by B being super slow.
  1. How Do We Solve the Problem of Microservices Being Slow?

Increasing the number of threads won’t solve the problem. What do web app users do when things are slow? They constantly refresh the page. So, when an application is slow because of thread limitaions, a bunch of users will hit refresh multiple times and it will make it even worse.

The solution to this is **Timeouts**! When something takes too long to process, you can cancel and free up that resource after a certain time. That’s why you see TimeoutExceptions with “Maximum timeout limit exceeded”.

There are 2 common ways to set timeouts:

* TODO name: (simple way)
* TODO name: (recommended way)

**Graphical user interface, text

Description automatically generatedThe simple way:**

We can set the timeout values in during the creation of our restTemplate Bean. Spring Boot will also pass a RestTemplateBuilder instance from which a RestTemplate can be created. See also the documentation: <https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.resttemplate>

From now on, every request done with this restTemplate will have a timeout value.

* But does this solve our Problem? **No, it does not solve it fully**.

This definitely helps and is good to do but if the rate of incoming requests are faster than thread rate which will be cleared because of the timeout, we will still run into the same issue of the thread pool exceeding.

* 1. The Circuit Breaker Pattern

One solution for this: the **movie-catalog-service** should be a little smart about sending the requests. The catalog service will say “I'm sending so many requests to this guy but this guy is not responding on time, he's taking a lot of time”. So it will detect the slow responding service (in our case the **movie-info-service**) and it will stop sending requests for a period of time.

The better, more scalable solution is, rather than only having the timeouts, also to not call a microservice which is responding slower than expected. So, we are kind of deactivating the problem component so it won’t affect other components. This is called the **circuit breaker pattern**.

A picture containing text

Description automatically generatedYou see this in electrical systems all the time. When there is a spike for example, the circuit will be broken to keep the safety of the other components. When it is better, after a while, either manually or automatically the detached component will be joined again.

A **circuit breaker** is an automatically operated electrical switch designed to protect an electrical circuit from damage caused by excess current from an overload or short circuit. Its basic function is to interrupt current flow after a fault is detected. Unlike a **fuse**, which operates once and then must be replaced, a circuit breaker can be reset (either manually or automatically) to resume normal operation.

**Where can we introduce a circuit breaker in our architecture?**

Technically in every component which is calling somebody else because when there is a call, it can technically lead to the exhaustion of resources, if the call will be slow.

* It is especially important where there is a microservice which is calling multiple microservices. We don’t want one of the slow connections to affect all other fast connections. In our case we will add a circuit breaker to **movie-catalog**.
  + 1. When should a circuit break?

In the case of an electrical circuit, the circuit breaker (sigorta) is triggered by a variance in the power that a particular component is getting. There is a limit, when it spikes and crosses a particular limit, the circuit breaker will kick in.

What is that parameter, that limit for a microservice?

Diagram

Description automatically generatedLet’s sat **request 1** -> success, **request 2** -> timeout. What do we do? Do we just break the circuit because we got a timeout? Well, that would be a bit harsh. Timeouts can happen. A single timeout is fine.

What would be the trigger? We could say that at any time, when the last 3 requests failed (either timeout or an error), the circuit should break. But then we won’t cover the case when 1 request is a success, the next 2 times out, the next success, the next 2 times out and so on.

This logic hast to be smart enough to cover all the possible cases

* + 1. Circuit Breaker Parameters

These are the parameters we need to set when using a circuit breaker pattern. There are more but these are the important.

**When does the circuit trip?**

* **Last n requests to consider for the decision**: When a request fails, look at the last n requests to make a decision
* **How many of those should fail?** Let’s say we are looking at the last 5 requests. If 2 Failed, do we break it?
* **Timeout duration:** At which point we will consider a request to be failed?

**When does the circuit un-trip?**

* **How long after a circuit trip to try again?**

It is a tricky question how many requests should be considered. It depends on how many requests your app is getting, how many threads are available in your thread pool and so on. It’s more like a trial an error thing. Load testing can be helpful.

* + 1. What to do when a circuit breaks?

So, if **movie-catalog** detects that another microservice is slow, then it has to do something else. It cannot just wait because then we will have the same problem of thread pool exhaustion of the server. It has to return something and free up the thread.

We need to have a **fallback mechanism**. So, we have code which executes for requests, instead of the normal request flow.

1. **We can throw an error (not recommended)**: It is bad for a microservice to throw an error message. Why? Because then something else must handle it/catch it. If not handled, it can bubble up in the request stack and somewhere it eventually must be handled. Imagine a system with a lot of microservices, if all of them could throw an error message, all other microservices will have all the handling logic for every call they make, which is messy and a lot of work.
2. **Return a fallback “default” response (a little better)**: Should the client know that it’s not a “normal” response? It is recommended to have the client not be aware of it at all. Same reason why we don’t want to throw an error.
3. **Save previous response (cache) and use that when possible (the optimal solution, if it is possible to do).** With mission critical systems not possible because you might return an old response while the client was expecting a different response. But for standard web applications it is fine.
   * 1. Why Do We Need Circuit Breakers?

* **Failing fast**: We want to fail as soon as possible if something is wrong. This is also said in a lot of microservice talks.
* **Fallback Functionality**: we can have our own fallback logic
* Diagram

  Description automatically generated with medium confidence**Automatic recovery**

So, do we need to implement our own circuit breaker pattern in our applications to solve these questions? Of course NOT.

* 1. Hystrix

An open-source library created by Netflix which implements the circuit breaker pattern, so you don’t have to do all the network programming and thread programming. We just need to give it the configuration parameters we talked about. Works very well with Spring Boot.

* **Hystrix** is **no longer in active development**, and is currently in maintenance mode. But this is because the project implemented all of their goals.
* Hystrix is no longer recommended to be used in new projects (The @EnableCircuitBreaker annotation is deprecated). TODO: See other implementations like **Resilience4j** or **Spring Cloud Circuit Breaker** (an abstraction layer).
* There are also other patterns being evaluated. One of them is **adaptive fault tolerance**. What we have is now static, we give hystrix config parameters and it does the job. But things will change and these parameters must be updated manually. With adaptive fault tolerance these parameters will change depending on the statistics of your we application. They write “Meanwhile, our focus has shifted towards more adaptive implementations that react to an application’s real time performance rather than pre-configured settings”.
  + 1. Adding hystrix to a Spring Boot application

1. **Add the dependency**. We have to add the version manually because it doesn’t come from spring cloud dependencies, because it is not actively developed?
2. Add **@EnableCircuitBreaker** to your main class. (deprecated annotation)
3. Add **@HystrixCommand** to methods that need circuit breakers. We are adding it to our movie catalog service because it is making multiple calls to other microservices.
4. Text

   Description automatically generatedConfigure Hystrix behaviour

Graphical user interface, text

Description automatically generated

Graphical user interface

Description automatically generated with low confidence

Now when we don’t start the movie-info-service, we get our fallback response.

* 1. How does Hystrix work

Text, whiteboard

Description automatically generatedHow does Hystrix do this? All that we did was to annotate our method. The answer is simple. It is how most of the things work in the Spring Environment (?).

* Hystrix wraps a **proxy class** around your API class.

When a component asks to get an instance of your API class, it will get this proxy class instead and this proxy class contains this circuit breaker logic. Hystrix is constantly monitoring what it gives back. It is proxying, passing and returning request to and from your class. When things fail, then depending on the configuration parameters we’ve given, it can take action, redirecting the request to the fallback method for example.

* This proxying behaviour can lead to some problems, so keep this in mind.

**Improving our Circuit Breaker logic**

The way we implemented our circuit breaker is **not ideal**. We added the annotation to the main method in movie-catalog which calls 2 API’s, so if any one of them fail, the fallback response will be shown. We want to be more granular. It is better to have a fallback for each of the API calls, so if one of them fail, the other one still will deliver the correct results.

Graphical user interface, text

Description automatically generatedI’ve extracted the 2 API call methods:

Graphical user interface, text, application, email

Description automatically generated

This code won’t work. The reason is the Proxy Class around this class.

In the previous example, our getCatalog method (annotated with **@HystrixCommand**) was called from the outside, from some other component which had the proxy class instance. That’s why the request could be tracked by hystrix.

But this time, the call to the methods which are annotate with **@HystrixCommand** are done from inside the method, the proxy class is out of the picture. Hystrix does not have an opportunity to intercept, 2 methods calling eachother inside the instance. This is why this code won’t work.

* This is a problem because of the way that Histrix was implemented. The only way we can solve this problem is to extract those methods to another class. So once you Autowire the new class and call the method on it, it will get the proxied method and only then the circuit breaker logic can kick in.

Text

Description automatically generated **Extracting our methods to classes**

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Now we can see, if the **ratings-data-service** is down, we get a fallback rating of 0, but still get the correct results from the **movie-info-service**.

* I’m not so sure about if this fallback solution is a good idea in **PROD**. It probably depends on the use case but if the response is 200 but you get a weird response, it might be a problem. Well, at least to the outside clients it will be. Maybe in internal communication from microservice to microservice it is more suitable?
  1. Parameters

commandProperties is an array of Hystrix property annotations.

* **timeoutInMilliseconds**: this is the timeout value. Hystrix will wait up to 2 seconds for the request to be finished, if not it will consider it as failed
* **requestVolumeThreshold**: looking at the last n request, in our example we saw it was 5. requestVolumeThreshold is a minimum threshold for the *volume* (number) of calls through the circuit that must be met (within the rolling window), before the circuit calculates a percentage failure rate at all. Only when this minimum volume (in each time window) has been met, will the circuit compare the failure proportion of your calls against the errorThresholdPercentage you have configured.
* **errorTresholdPercentage**: trip the circuit breaker if the this X percentage of the last n requests failed.
* **sleepWindowInMilliseconds:** How long the circuit breaker will sleep before it picks up again.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

* Add **actuator** and **hystrix-dashboard** dependencies
* **@EnableHystrixDashboard** in your main class
* Add properties
* Go to: <http://localhost:8081/actuator/hystrix.stream>

Graphical user interface, text

Description automatically generated

A picture containing text

Description automatically generated

This will show the request statistics.

I stopped **ratings-data-service**, the first request (first failure) showed the fallback response (not sure why).

Hystrix now correctly shows ‘Circuit Closed’ for getRatingsFromApi but it didn’t take up 10 seconds to it be “Closed” again.

I need to test this and read more about how it works.

1. Microservices Level 3 – Configuration
2. Understanding microservice configuration goals
3. Configuration features in Spring Book
   1. Value
   2. Config props
   3. Actuator
   4. Spring Profiles
   5. Environment
4. Spring Cloud Config Server
5. Dynamic configuration
6. Best practices and patterns
   1. What and Why?

Configuration is essential for any production grade application. Any decent size production application requires settings which can be adjusted, some kind of putting all the right gears and the switches in the right place so that the application can do its job.

* The best practice is to separate code and config, meaning your configuration should not be mixed within your code. This is expressed as “Don’t hardcode”, meaning to keep it dynamic.
* Why? Because those configuration values should be easily changeable, without having to change and re-deploy your code. Preferably you should be able to change them while your app is still running, without even the need to restart your application.

Example configuration types:

* **Database connections**: the URL to the DB and DB-Credentials
* **Credentials**: other kind of credentials
* **Feature flags**: flags which indicates if some features should be active or not, or even the time frame, to which users this feature should be visible and all kind of possible options.
* **Business logic configuration parameters**: For example, an E-Commerce platform where you want to have a discount percentage. Depending on the feedback, you can increase or decrease such values on the fly.
* **Scenario testing**: A/B testing, you don’t want to enable a feature or a scenario for every user, you instead split the users. For instance, 10% will see A, the rest will see B. Then you can monitor the usage and gradually increase or decrease the percentage.
* Diagram

  Description automatically generated**Spring Boot configuration**: Spring Boot has a lot of config parameters and there are also all the other Spring projects which heavily rely on configuration parameters.

A decent configuration management system should allow you to change these types of config parameters whenever you want, preferably real time, without re-testing and re-deploying your application.

Most of the time you will have a CI/CD system and you want your config to be separate. In most cases, a config parameter should not require you to build and deploy your application again.

Types of Configurations

* **XML Files**: The earlier versions of Java EE, Hibernate, Spring you needed a lot of XML files. Thankfully we don’t use them anymore because they were so verbose.
* **Property Files, YAML, JSON:** these provide a better developer experience.

Goals:

* **Externalized**: In the case of microservices, you might have hundreds or thousands of different microservices, or multiple of the same microservice instances. Therefore, we want our config to be externalized, meaning separate from our business logic.
* **Environment specific**: You want your application to run on different environments/servers,   
  LOCAL, DEV, TEST, STAGING, PROD, meaning each environment has to have have it’s specific configuration.
* **Consistent**: all of the instances should have the same config values.
* **Version history:** you want to be able to track your changes.
* **Real-time management:** you want the changes to happen in real-time, without restarting your services.

We will start minimally and build up from there.

* 1. Property files

The first step to externalize our configuration is the put it in a property file and read it from there.

Graphical user interface, text, application, Word

Description automatically generatedYou can put all your config values in your application.properties file as **key value pairs** and read these values in your application.

You can use the **@Value** annotation to inject property values into fields.

* See the Spring Boot document for more detailed examples of the @Value annotation.

You can also reference another property in your property file:

Timeline

Description automatically generated

Have we achieved any of the goals we want?

* **NO**, we didn’t event achieve externalizing configs fully because when you build your application, this property file will be inside your JAR. You can still benefit from this, since these property values can be overridden but it is not fully externalized yet.
  1. External Property Files

How can we change the property files inside a JAR after we’ve built it?

1. **Extract the JAR, change the properties file and zip it again**
2. **Runtime arguments**: You can override properties during runtime. Make sure you pass them before the name of the jar, because if you pass them after, they will be considered **program arguments** (available in the **args** array of your main method) and not **VM-arguments** (control the heap size, etc.), I think they are also called **system properties**.  
   System properties have priority over normal properties, therefore we can override normal properties by passing system props with the exact same name like this:  
     
   java -jar **-Dapp.name=Billy** spring-boot-configuration-0.0.1-SNAPSHOT.jar  
     
   So basically, the arguments after -jar and before the .jar name, can be either VM-arguments or system properties.
3. **External property file**: You can also create an **application.properties** file in the same folder as the jar you are executing and the values in this external jar will override the ones which is inside the jar.   
   It will ONLY override the existing/overlapping properties and not the whole file. So if you have 5 properties in your jar, you can only override 1 with an external jar for instance.
4. **Spring Cloud Config: TODO** ?
   1. @Value Annotation



* + 1. Property (key) is not defined

When the given property does not exist, the application won’t start because the bean cannot be created:

* org.springframework.beans.factory.BeanCreationException: Error creating bean with name **'greetingController'**: Injection of autowired dependencies failed; nested exception is java.lang.IllegalArgumentException: Could not resolve placeholder '**app.fullName**' in value "**${app.fullName}**"
  + 1. Property key exist without value
* If the property is a **String**, then it will be an empty string (not null)
* If the property is a **List<String>**, then it will be an empty list (not null)
* If the property is a **String[]**, then it will be an empty array (not null)
  + 1. Giving a default value to @Value

1. **Property does not exist:** then the app will start and will   
   inject “default value” as the value to the field
2. **Property exists but is empty**: then the empty string will be injected.
   * 1. List or String[]

Text

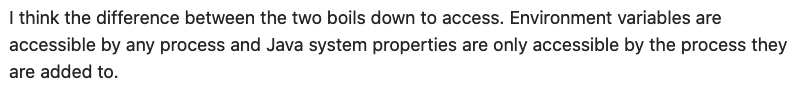
Description automatically generated



* + 1. System Property

Graphical user interface, text, application, email

Description automatically generated



System properties can also be accessed the same way with the @Value annotation, like normal properties. But they have higher priority and will “override” the normal property if they have the same name

To inject all current system properties:

Text

Description automatically generated with medium confidence

* + 1. SpEL

The Spring Expression Language (SpEL) is a powerful expression language that supports querying and manipulating an object graph at runtime. It can be used with XML or annotation-based Spring configurations.

These expressions can also be used inside @Value annotations in order to do many dynamic different operations

* For more info: <https://www.baeldung.com/spring-expression-language>
  + 1. Map

Using SpEL, we can also assign the value of a property as a map to a field.





Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated with medium confidence

Text

Description automatically generated

Text

Description automatically generatedYou can also do other cool things like

* access an entry in a map
* Text

  Description automatically generatedIf you are not sure whether the Map contains a certain key, we should choose a **safer expression** that will not throw an exception but set the value to null when the key is not found:
* You can also set default values for the properties or keys that might not exist
* Map entries can also be filtered before injection
  + 1. Constructor Injection and Setter Injection

Graphical user interface, text, application, email

Description automatically generated

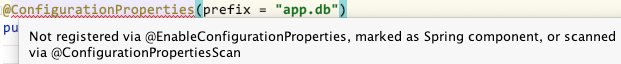
* 1. @ConfigurationProperties Annotation

Rather than injecting each property value one by one, we can inject a group of values to a given bean.

Spring will automatically bind any property defined in our property file that has the prefix app.db and the same name as one of the fields in the ConfigProperties class.

* **Text

  Description automatically generatedImportant**: use the **immutable properties** (see 4.5.5) in a production system to make sure properties cannot be changed once they have been initialized.
* Spring uses some relaxed rules for binding properties.   
  As a result, the following variations are all bound to the property **hostName**:



Text

Description automatically generatedTo be able to use this class we need to do one of the following:

* Register it with **@EnableConfigurationProperties**
* ****Mark it as a Spring Component. Use **@Configuration** so that Spring creates a Spring bean in the application context.
* Use **@ConfigurationPropertiesScan**: As of Spring Boot 2.2, Spring finds and registers **@ConfigurationProperties** classes via classpath scanning. Therefore, there is no need to annotate such classes with @Component (and other meta-annotations like @Configuration), or even use the @EnableConfigurationProperties. We can also scan only custom locations, like in almost every type of annotation which scans the class path



* + 1. Nested Properties

Text

Description automatically generatedUse immutable properties in a production system (see 4.5.5)

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

* + 1. Using @ConfigurationProperties on a @Bean Method

We can also use the **@ConfigurationProperties** annotation on @Bean-annotated methods.

* This approach may be particularly useful when we want to bind properties to a **third-party component** that's outside of our control.

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated with medium confidence

* + 1. Property Conversion

@ConfigurationProperties supports conversion for multiple types of binding the properties to their corresponding beans.

Graphical user interface, text

Description automatically generatedSome examples are:

* **Duration** (nanoseconds, microseconds, milliseconds, seconds, minutes, hours, and days),
* **DataSize** (B, KB, MB, GB, and TB)
* **Custom Converter:** We can also add our own custom Converter to support converting a property to a specific class type.

A picture containing text

Description automatically generated

* + 1. Property Validation

@ConfigurationProperties provides validation of properties using   
the **JSR-303 format**. This allows all sorts of neat things. We need a dependency to enable it.

* The class must be annotated with **@Validated** for the validations to take effect.
* Graphical user interface, text, application

  Description automatically generatedGraphical user interface, text, application, chat or text message

  Description automatically generatedJava provides out of the box simple type checking. So if you define an **int** field, your app will fail if you assign a string to that property. With these validations, you can have a fine-grained control over how properties are validated. If any of these validations fail, then the main application would fail to start with an **IllegalStateException**.
  + 1. Making it Immutable

This is **really important** to make sure nobody can accidentally change the properties defined in a bean.

* **@ConstructorBinding** is needed.
* Furthermore, it's important to emphasize that to use the constructor binding, we need to explicitly enable our configuration class either with @EnableConfigurationProperties or with @ConfigurationPropertiesScan.

Table

Description automatically generated with medium confidence

Graphical user interface, text, application

Description automatically generated

* + 1. When to use which?
* Use **@Value** when you have just one or two properties you want to access in just some of the places in your app.
* Use **@ConfigurationProperties** when you have multiple properties which can be categorized under a term/class. It is better to use this when you want to access the same properties in multiple places in your application. Since this creates a bean, you just can inject it at the places you want to use it, instead of defining the same @Value fields each time.
  1. Multiple Spring Profiles

We haven’t made our configuration environment specific yet. To achieve this, a good way is to use **spring profiles**. With spring profiles, you can have different configuration values for different profiles and you can specify the profile to be used in each environment.

* When you don’t specify any active profile, the **default profile** is used by all the beans. All the configuration you put in your application.properties file is used by default because the default profile is active.
* Graphical user interface, application

  Description automatically generatedThe default profile is always active.
* When you specify an **active** profile (for example test), it will sit on top of the default profile, which means test will override the values defined in default
* You can have multiple profiles active at the same time. But **be careful** if you want to use multiple profiles (other than default) and you define the same values in multiple profiles. The order is: The last definition wins and there are some other precedence rules.
* Text

  Description automatically generated with low confidenceKeep in mind, you can customize both the **active** and the **default** profile.
* You can also configure beans for a specific environment with the **@Profile(“test”)** annotation. But again, be careful with this.   
  By default all the beans use **@Profile(“default”)**
* Maven profiles and spring profiles are similar concepts.

Graphical user interface, text, application

Description automatically generatedUsing this we can create different config sets for each profile. We can put all the values inside application.properties which does not change depending on the profile.

There are 2 ways to define multiple profiles:

1. Define everything in application.properties. You can use   
   the **#--** separator so spring will understand that they are different profiles.
2. Text

   Description automatically generated with medium confidenceDefine one file for each profile with the name:  
   **application-<profile>.yml/properties**



* 1. Using YAML vs Properties

As well as Java properties files, we can also use YAML-based configuration files in our Spring Boot application.

* YAML is a convenient format for specifying **hierarchical configuration data**.

Text

Description automatically generatedText

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidenceTable

Description automatically generated

* 1. Graphical user interface, text, application, chat or text message

     Description automatically generatedSpring Cloud Config Server

We have reached the following goals. We still don’t really have consistency, version history and real time management.

**Consistent**: changes in the config should be reflected in all instances of your app

**Version history**: we kind of have this one but we cannot be sure if the last changes were actually deployed or not.

**Real-time management**: we have to re-build and re-deploy our app each time when the config changes.

Consistency is really important since we need to make sure every instance of your app gets the same config values when a change happens. Each instance holds its own config values. We have to take that out and put it into a configuration service, making it the single source of truth and making all the instances talk with that service.

* This is called config as a microservice. This is a common pattern. A service which is responsible for providing your configuration and all the other microservices fetches the values needed from that service. This way we can solve the consistency problem.
* The time when the values are being fetched can depend. Can be on app startup, can be real-time.

Options:

* Apache Zookeeper
* A picture containing text, clock, watch

  Description automatically generatedETCD – distributed key-value store
* Consul
* Spring Cloud Config Server
  + 1. How and where does this configuration service store the configs?

We want to push the changes to the config without a re-build/re-deploy. The spring cloud config server can connect to various different data sources but there is one data source which is extremely popular because it provides also versioning. That is a **Git repository**. (It can also use **SVN** or **HashiCorp Vault**)

* Spring cloud config server can directly connect to a Git repo hosted online and look up the values from there.
* So, changing the config values for production is as easy as creating a new commit and submit it to that Git repo.
  + 1. Graphical user interface, text, application, email

       Description automatically generatedSetting up a Spring Cloud Config Server

I’ve created a new spring boot project with only one dependency

* Added the **@EnableConfigServer** annotation
* http://localhost:8888/**<file-name>**/**<profile>**

Text

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

* Calling http://localhost:8888**/application/default** will read the application.yml file on the remote repo and will display the following results

Graphical user interface

Description automatically generatedChart

Description automatically generated with medium confidence

Graphical user interface

Description automatically generated with medium confidence

Graphical user interface, application

Description automatically generated

* A picture containing logo

  Description automatically generatedFetching properties for DEV:   
   http://localhost:8888**/application-dev/default**
* Logo

  Description automatically generatedFetching properties for PROD:   
   http://localhost:8888**/application-prod/default**
  + 1. Setting up the Spring Cloud Config Client

Adding the following snippets to my already existing project:

Graphical user interface, text, application, email

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

The properties are successfully read from the spring cloud config server:

Graphical user interface, text, application, Word

Description automatically generatedText

Description automatically generated with low confidence

A picture containing logo

Description automatically generatedCalling http://localhost:8077/remote:

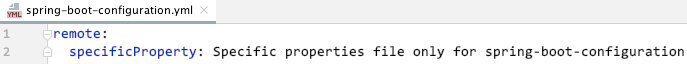
* A picture containing text

  Description automatically generatedWhen **no profile (default)** is active
* When **dev** is active
* When **prod** is active

Which means the client correctly calls the right Path depending on which profile is currently active.

* + 1. Text

       Description automatically generatedMicroservice Specific Configuration

We can also create property files specific for a certain microservice. I’ve created a spring-boot-config.yml file and pushed it.

Calling http://localhost:8888**/spring-boot-configuration/default** displays:  
(Other microservices which consume this config server would not be able to get these values)

A picture containing graphical user interface

Description automatically generated

* + 1. Dynamic Configuration

By default, when you push a new version of the config to the config repo:

* The new config is displayed at the **Spring Cloud Server endpoint** automatically. Because it checks for changes each time the endpoint is called (?)
* But the running **clients** will still have/use the old value. This is by design. When you have a bunch of microservices, you don’t want them constantly check if there are any updates, it is resource intensive. Since the properties are not changed that often, it is not feasible to check for it periodically. Therefore by default, the only way is to restart the client so it would read the updated value from the config server.

But we can make this process dynamic.

1. Add the spring-boot-starter-actuator dependency and enable the **refresh** endpoint (an endpoint / a hook for you to call to tell the application to refresh its config.   
   Quick and dirty way: management.endpoints.web.exposure.include=\*)
2. Mark the spring components that should be refreshed when the config changes.   
   The **@RefreshScope** is an annotation that tells the spring framework to refresh the config used in the bean.  
   “*Convenience annotation to put a @Bean definition in refresh scope. Beans annotated this way can be refreshed at runtime and any components that are using them will get a new instance on the next method call, fully initialized and injected with all dependencies*”
3. Send a **POST** request to http://localhost:8077**/actuator/refresh**

Graphical user interface, text, application

Description automatically generatedThe response displays the names of properties where the value have changes



* + 1. Best Practices

**TODO** – saving passwords as plain text in github does not seem great - what about encryption or hashiCorp vault?

https://www.youtube.com/watch?v=AiGCx0raQfs

**TODO** – the first invocation of the properties takes a while. Does spring cloud clone the remote repo first?

**TODO** – can yaml be used always as property files, any configuration needed? no problems with it?

**TODO**: property file for each profile like we have. Do I have this in Spring Boot Document already?

**TODO**: 3 ways to run Spring Boot apps from command line – maybe put it in Spring Boot Document? <https://www.youtube.com/watch?v=Le5YjYNYtZg>

**TODO**: put the **4.2 External properties file** section also in the Spring Boot document

**TODO**: this is also interesting - <https://www.baeldung.com/properties-with-spring>

**TODO**: Also document here how to use the **java-dotenv** and read environment variables easily.