

# SPRING MICROSERVICES

---

Pre-reqs:

- Java
- Spring Boot

# Microservices introduction

- Microservices is an architecture wherein all the components of the system are put into individual components, which can be built, deployed, and scaled individually.
  - a particular way of designing software applications as suites of independently deployable services.
- Microservices are loosely coupled services which are combined within a software development architecture to create a structured application.
  - A microservices architecture allows the individual services to be deployed and scaled independently (typically via containers), worked on in parallel by different teams, built in different programming languages, and have their own continuous delivery and deployment stream.

# Microservices

- The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an [HTTP resource API](#).
  - Each of these services is responsible for discrete task and can communicate with other services through simple APIs to solve a larger complex business problem.
  - These services are built around business capabilities and independently deployable by fully automated deployment machinery.
  - There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.
- Microservices provide an approach for developing quick and agile applications, resulting in less overall cost.

# Characteristics of microservices

- Single responsibility per service : states that a unit should only have one responsibility.
- Microservices are autonomous : they are self-contained, independently deployable, and autonomous services that take full responsibility of a business capability and its execution.
- Services are first-class citizens : Microservices expose service endpoints as APIs and abstract all their realization details.
  - Internals such as the data structure, technologies, business logic, and so on are hidden
  - Access is restricted through the service endpoints or APIs.
  - For instance, Customer Profile microservices may expose Register Customer and Get Customer as two APIs for others to interact with
- Microservices are lightweight
  - Well-designed microservices are aligned to a single business capability, so they perform only one function; resulting in microservices with smaller footprints
- Microservices communicate based on a few basic principles
  - They employ lightweight communication protocols such as HTTP and JSON for exchanging data between the service consumer and service provider.

## First Demo (using Spring Boot)

- Create three Spring Boot projects
  - Build Movie Catalog service API (port 8081)
  - Build Movie Info service API (port 8082)
  - Build Ratings Data Service API (port 8083)
  - Have the Movie Catalog service call the other two (the naïve way)



## movie catalog service

- Lets focus on movie catalog service: This is the API we want
- Given the userid, I want a payload.
- So, add a API to movie-catalog-service at /catalog/{userid} that returns a hard-coded list of movie + rating info.

```
package com.trg.moviecatalogservice.model;

public class CatalogItem {
    String name; // title of movie
    String desc; // movie description
    int rating;

    public CatalogItem(String name, String desc, int rating) {}

    public String getName() {}

    public void setName(String name) {}

    public String getDesc() {}

    public void setDesc(String desc) {}

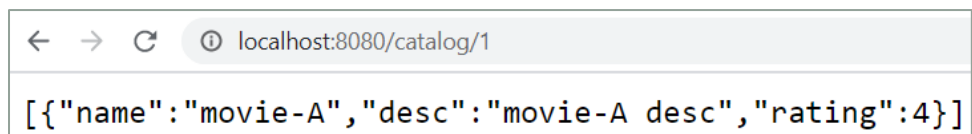
    public int getRating() {}

    public void setRating(int rating) {}
}
```

```
@RestController
@RequestMapping("catalog")
public class MovieCatalogResource {

    @RequestMapping("/{userid}")
    public List<CatalogItem> getCatalog(@PathVariable("userid") String userid){
        return Collections.singletonList(
            new CatalogItem("movie-A", "movie-A desc", 4)
        );
    }
}
```

Now run the main app, which starts the embedded server. Test the app by calling the URL :  
<http://localhost:8080/catalog/1>



```
localhost:8080/catalog/1

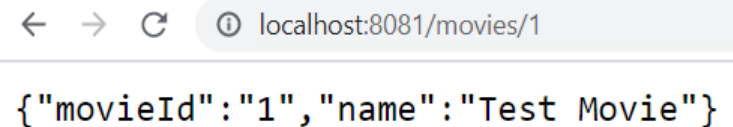
[{"name":"movie-A","desc":"movie-A desc","rating":4}]
```

# Movie Info Service

- Given the movieid, I want a payload. So, add a API to movie-info-service at /movies/{movieid} that returns a hard-coded movie info.
- Create 2 classes:

```
public class Movie {  
    String movieId;  
    String name;  
  
    public String getMovieId() {  
        return movieId;  
    }  
    public void setMovieId(String movieId) {  
        this.movieId = movieId;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public Movie(String movieId, String name) {  
        super();  
        this.movieId = movieId;  
        this.name = name;  
    }  
}
```

```
@RestController  
@RequestMapping("movies")  
public class MovieResource {  
  
    @RequestMapping("/{movieId}")  
    public Movie getMovieInfo(@PathVariable("movieId") String movieId){  
        return new Movie(movieId, "Test Movie");  
    }  
}
```



```
localhost:8081/movies/1  
  
{"movieId":"1","name":"Test Movie"}
```

- To run this, once again run the main application for this project. Change port to 8081

## rating-data-service

- Add a API to rating-data-service at /ratingsdata/{movieid} that returns a rating given the movieid.

```
public class Rating {
    String movieId;
    int rating;

    public String getMovieId() {
        return movieId;
    }

    public void setMovieId(String movieId) {
        this.movieId = movieId;
    }

    public int getRating() {
        return rating;
    }

    public void setRating(int rating) {
        this.rating = rating;
    }

    public Rating(String movieId, int rating) {
        super();
        this.movieId = movieId;
        this.rating = rating;
    }
}
```

```
@RestController
@RequestMapping("ratingdata")
public class RatingResource {

    @RequestMapping("/{movieId}")
    public Rating getRatingInfo(@PathVariable("movieId") String movieId){
        return new Rating(movieId, 5);
    }
}
```

← → ↻ ⓘ localhost:8083/ratingdata/1

{"movieId":"1","rating":5}



# How to make a REST call from your code?

- Call REST APIs programmatically
  - Spring Boot already comes with a client in your classpath – the RestTemplate
  - Why use RestTemplate? Because response is in the form of a JSON object
- Lets revisit MovieCatalogResource.java. We had hardcoded a list previously. Now we need to do several steps:
  - Get all rated movie ids
  - For each MovieId, call movie-info-service and get movie details.
  - Put them all together
- **I want movie info from movie-info-service! I will make a REST call using RestTemplate**

```
@RestController
@RequestMapping("/catalog")
public class MovieCatalogResource {

    @RequestMapping("/{userid}")
    public List<CatalogItem> getCatalog(@PathVariable("userid") String userid) {

        List<Rating> ratings = Arrays.asList(new Rating("1", 1), new Rating("2", 2), new Rating("3", 3));
        RestTemplate restTemplate = new RestTemplate();

        return ratings.stream().map(rating -> {
            Movie movie = restTemplate
                .getForObject("http://localhost:8082/movies/" + rating.getMovieId(), Movie.class);
            return new CatalogItem(movie.getName(), "Hardcoded Desc", rating.getRating());
        }).collect(Collectors.toList());
    }
}
```

# Understanding Service Discovery

- Spring Cloud
  - Spring Cloud is a [new project](#) in the [spring.io family](#) with a set of components
  - Spring Cloud framework provides tools for developers to build a robust cloud application quickly.
  - We can also build the microservice-based applications, for example, configuration management, **service discovery**, circuit breakers, intelligent routing, cluster state, micro-proxy, a control bus, one time tokens, etc.
  - To a large extent Spring Cloud 1.0 is based on components from [Netflix OSS](#). Spring Cloud integrates the Netflix components in the Spring environment in a very nice way using auto configuration and convention over configuration similar to how [Spring Boot](#) works.

# Netflix OSS

Netflix Component Name	Functionality
Eureka	Service Registration and Discovery
Ribbon	Client Side Load Balancing
Hystrix	Circuit Breaker
Zuul	Edge Server, Intelligent Routing

- Eureka Server is an application that holds the information about all client-service applications.
- Every Micro service will register into the Eureka server and Eureka server knows all the client applications running on each port and IP address.
- Eureka Server is also known as **Discovery Server**.
- Eureka also comes with a Java-based client component, the Eureka Client, which makes interactions with the service much easier.
- **Eureka Server comes with the bundle of Spring Cloud.**
- Eureka is a open source project from Netflix

# Understanding Service Discovery

- Spring Cloud uses Client side service discovery
  - ***Client-side service discovery*** allows services to find and communicate with each other without hard-coding hostname and port.
  - The only 'fixed point' in such an architecture consists of a *service registry* with which each service has to register.
- The Technology that implements Service discovery in Spring Cloud is **Eureka**.
- There are lot of other technologies that Spring integrates with, but Eureka is mostly the technology of choice

# Spring Cloud Netflix

- In Spring Initializer download the Spring Boot project with Eureka server dependency

**Project**  
☒ Maven Project  
☐ Gradle Project

**Language**  
☒ Java ☐ Kotlin  
☐ Groovy

**Spring Boot**  
☐ 2.4.0 (SNAPSHOT) ☐ 2.4.0 (M3)  
☐ 2.3.5 (SNAPSHOT) ☒ 2.3.4 ☐ 2.2.11 (SNAPSHOT)  
☐ 2.2.10 ☐ 2.1.18 (SNAPSHOT) ☐ 2.1.17

**Project Metadata**  

Group

com.trg

Artifact

discovery-server

Name

discovery-server

Description

Demo project for Spring Boot

Package name

com.trg.discovery-server

Packaging

☒ Jar ☐ War

Java

☐ 15 ☐ 11 ☒ 8

**Dependencies**

ADD DEPENDENCIES... CTRL + B

**Eureka Server** **SPRING CLOUD DISCOVERY**  
spring-cloud-netflix Eureka Server.

# Building a Eureka Server

- Add `@EnableEurekaServer` annotation to Spring Boot Application class
  - The `@EnableEurekaServer` annotation is used to make your Spring Boot application acts as a Eureka Server.

```
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
...
@SpringBootApplication
@EnableEurekaServer
public class EurekaserverApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaserverApplication.class, args);
    }
}
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

# Building a Eureka Server

- By default, the Eureka Server registers itself into the discovery.
- You should add the below given configuration into your application.properties file or application.yml file.

```
#application.properties
server.port=8761
# so that Eureka doesnt register with itself
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

```
#application.yml file
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
  server:
    port: 8761
```

- eureka.client.register-with-eureka = false - Makes it so the server does not attempt to register itself.
- eureka.client.fetch-registry = false - With this, we inform customers that they must not store the data of the available instances in their local cache. This means that customers must consult the Eureka server whenever they need to access a service. In production, this is often set to true to expedite requests. This cache is updated every 30 seconds by default.