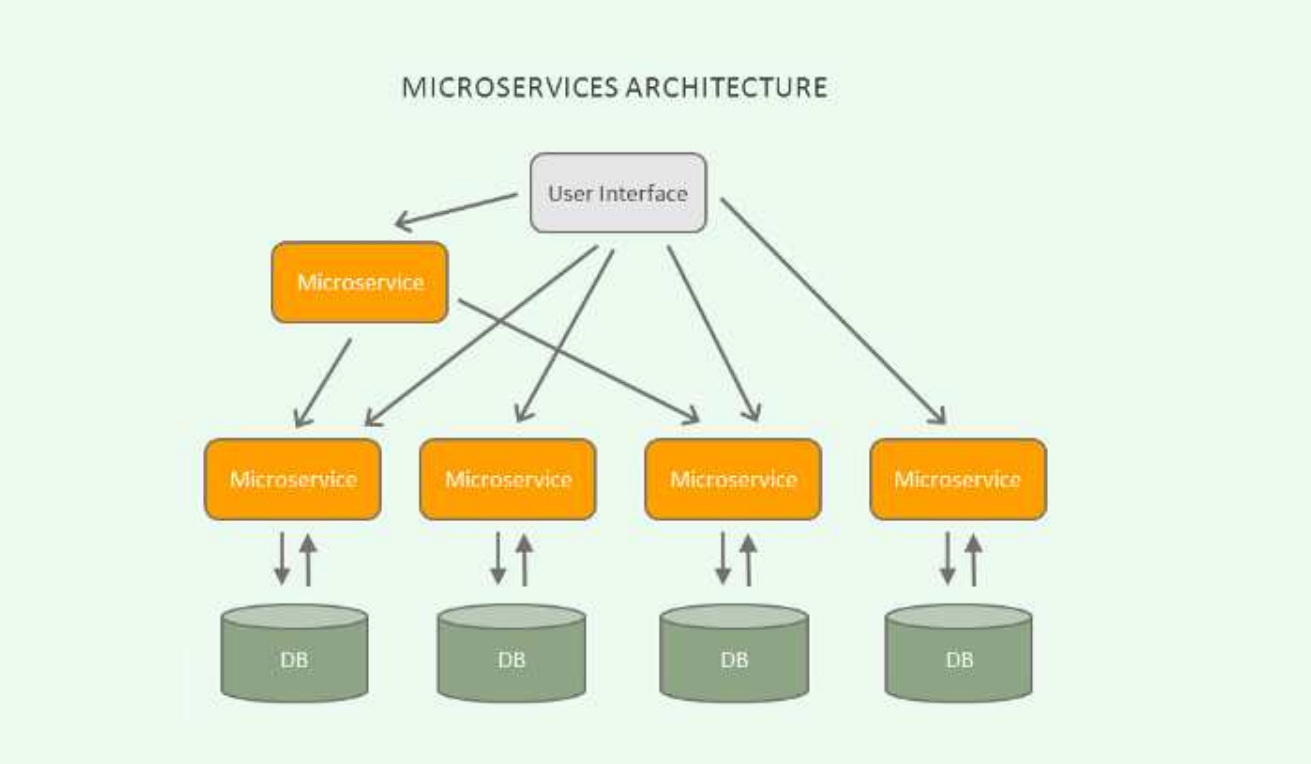
MicroServices

Introduction

Microservices are a form of service-oriented architecture style wherein applications are built as a collection of different smaller services rather than one whole app. Instead of a monolithic app, you have several independent applications that can run on their own and may be created using different coding or programming languages. Big and complicated applications can be made up of simpler and independent programs that are executable by themselves. These smaller programs are grouped together to deliver all the functionalities of the big, monolithic app.



Benefits

- These smaller applications are not dependent on the same coding language, the developers can use programming language that they are most familiar with.
- Increases development speed with lower costs and fewer bugs.
- The agility and low costs can also come from being able to reuse these smaller programs on other projects.

Microservices frameworks for Java

There are several microservices frameworks that you can use for developing for Java. Some of these are:

- Spring Boot
- Dropwizard
- Restlet
- Spark

A simple microservice example, which contains 3 web services

1. Pick a student details based on ID (Personal Details Service API)
2. Pick a student marks based on ID (Marks Details Service API)
3. Call these above two microservices (Student Details Service API = PDS + MDS)

steps for Developing the project

- Create 3 SpringBoot projects with "Web" dependency using STS
    - StudentService
    - MarksService
    - StudentMarksService
- First develop with all hardcoded values to return to Webservice calls.

Configuration of Tomcat

Out of these above three services can only one can run at a time. You need to make sure the remaining two should shutdown. The reason behind behavior is Tomcat server embedded in all services uses the same port i.e. 8080, which is the default configuration. But in business, those three services should be running at the same time, without the impact of the other one.

To achieve this, we need to add a property in `application.properties` file :

```
server.port=8181
```

in each of the service and give different port number.

Verify the below URLs:

- Student details of mentioned ID : http://localhost:8181/student/101
- Marks details of the mentioned student id : http://localhost:8282/marks/101
- Student details and marks details of the mentioned student id : http://localhost:8383/studentmarks/101

Source code :

These below three projects were created using STS 4.4.4.x. Select Spring Web is the only dependency has choose, since the objective is only to observe the execution flow and basic behavior.

StudentService – Project

Just to demonstrate to connect the Webservice to database data, this only project has Spring JPA and MySQL dependency along with Spring Web dependency.

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.3.3.RELEASE</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.dxc</groupId>
	<artifactId>StudentService</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>StudentService</name>
	<description>Demo project for Spring Boot</description>

	<properties>
		<java.version>1.8</java.version>
	</properties>

	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-jpa</artifactId>
		</dependency>
		<dependency>
			<groupId>mysql</groupId>
			<artifactId>mysql-connector-java</artifactId>
			<scope>runtime</scope>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
			<exclusions>
				<exclusion>
```

```
                                <groupId>org.junit.vintage</groupId>
                                <artifactId>junit-vintage-engine</artifactId>
                        </exclusion>
                </exclusions>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/dxcmicro1
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update
spring.jpa.database-platform=org.hibernate.dialect.MySQL57Dialect
spring.jpa.generate-ddl=true
spring.jpa.show-sql=true

server.port=8181
```

StudentServiceApplication.java

```
package com.dxc.student;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class StudentServiceApplication {

	public static void main(String[] args) {
		SpringApplication.run(StudentServiceApplication.class, args);
	}

}
```

Student.java

```
package com.dxc.student.model;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Student {
	@Id
	private int id;
	private String name;
	public Student() {
		super();
		// TODO Auto-generated constructor stub
	}
	public Student(int id, String name) {
		super();
		this.id = id;
		this.name = name;
	}
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	@Override
	public String toString() {
		return "Student [id=" + id + ", name=" + name + "]";
	}


}
```

StudentResource.java

```
package com.dxc.student.resource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.dxc.student.model.Student;
import com.dxc.student.repository.StudentRepository;

@RestController
```

```
@RequestMapping("/student")
public class StudentResource {

	@Autowired
	StudentRepository studentRepository;

	@RequestMapping("/{id}")
	public Student getStudent(@PathVariable("id") int studid) {
		Student student = (Student)studentRepository.findById(studid).orElse(new
Student());
		return student;

	}

}
```

StudentRepository.java

```
package com.dxc.student.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.dxc.student.model.Student;

public interface StudentRepository extends JpaRepository<Student, Integer> {

}
```

MarksService – Project

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.3.3.RELEASE</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.dxc</groupId>
	<artifactId>MarksService</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>MarksService</name>
	<description>Marks Service</description>

	<properties>
		<java.version>1.8</java.version>
```

```
	</properties>

	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
			<exclusions>
				<exclusion>
					<groupId>org.junit.vintage</groupId>
					<artifactId>junit-vintage-engine</artifactId>
				</exclusion>
			</exclusions>
		</dependency>
	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>

</project>
```

application.properties

```
server.port=8282
```

MarksServiceApplication.java

```
package com.dxc.marks;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MarksServiceApplication {

	public static void main(String[] args) {
		SpringApplication.run(MarksServiceApplication.class, args);
	}

}
```

Marks.java

```
package com.dxc.marks.model;

public class Marks {
	private int studid;
	private int examid;
	private int sub1;
	private int sub2;
	public Marks() {
		super();
		// TODO Auto-generated constructor stub
	}
	public Marks(int studid, int examid, int sub1, int sub2) {
		super();
		this.studid = studid;
		this.examid = examid;
		this.sub1 = sub1;
		this.sub2 = sub2;
	}
	public int getStudid() {
		return studid;
	}
	public void setStudid(int studid) {
		this.studid = studid;
	}
	public int getExamid() {
		return examid;
	}
	public void setExamid(int examid) {
		this.examid = examid;
	}
	public int getSub1() {
		return sub1;
	}
	public void setSub1(int sub1) {
		this.sub1 = sub1;
	}
	public int getSub2() {
		return sub2;
	}
	public void setSub2(int sub2) {
		this.sub2 = sub2;
	}
	@Override
	public String toString() {
		return "Marks [studid=" + studid + ", examid=" + examid + ", sub1=" +
sub1 + ", sub2=" + sub2 + "]";
	}


}
```

MarksResource.java

```
package com.dxc.marks.resource;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.dxc.marks.model.Marks;

@RestController
@RequestMapping("/marks")
public class MarksResource {

	@RequestMapping("/{id}")
	public List<Marks> getMarks(@PathVariable("id") int studid) {
		List<Marks> marks = Arrays.asList(
				new Marks(101, 801, 89, 89),
				new Marks(101, 802, 78, 78),
				new Marks(102, 801, 99, 88),
				new Marks(102, 802, 77, 99)

				);
		return marks.stream()
				.filter( mark -> mark.getStudid() == studid)
				.collect(Collectors.toList()) ;
	}

}
```

StudentMarksService – Project

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.3.3.RELEASE</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.dxc.studentmarks</groupId>
	<artifactId>StudentMarksService</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>StudentMarksService</name>
```

```
	<description>Student Marks Service</description>

	<properties>
		<java.version>1.8</java.version>
	</properties>

	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
			<exclusions>
				<exclusion>
					<groupId>org.junit.vintage</groupId>
					<artifactId>junit-vintage-engine</artifactId>
				</exclusion>
			</exclusions>
		</dependency>
	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>

</project>
```

application.properties

```
server.port=8383
```

StudentMarksServiceApplication.java

```
package com.dxc.studentmarks;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class StudentMarksServiceApplication {

	public static void main(String[] args) {
		SpringApplication.run(StudentMarksServiceApplication.class, args);
```

```
	}

}
```

StudentMarks.java

```
package com.dxc.studentmarks.model;

import java.util.List;

public class StudentMakrs {

	// One set of record from Student model
	// many sets of record from Marks model
	private String name;
	private List<Marks> marks;
	public StudentMakrs() {
		super();
		// TODO Auto-generated constructor stub
	}
	public StudentMakrs(String name, List<Marks> marks) {
		super();
		this.name = name;
		this.marks = marks;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public List<Marks> getMarks() {
		return marks;
	}
	public void setMarks(List<Marks> marks) {
		this.marks = marks;
	}
	@Override
	public String toString() {
		return "StudentMakrs [name=" + name + ", marks=" + marks + "]";
	}


}
```

Student.java

Copy from StudentService project

Marks.java

Copy from MarksService project

StudentMarksResources.java

```
package com.dxc.studentmarks.resource;

import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import com.dxc.studentmarks.model.Marks;
import com.dxc.studentmarks.model.Student;
import com.dxc.studentmarks.model.StudentMakrs;

@RestController
@RequestMapping("/studentmarks")
public class StudentMarksResource {

	@RequestMapping("/{id}")
	public StudentMakrs getStudentMarks(@PathVariable("id") int studid) {

		// Obtain Student data from it's webservice :
http://localhost:8181/student/{id}

		RestTemplate restTemplate = new RestTemplate();
		Student student =
restTemplate.getForObject("http://localhost:8181/student/"+studid, Student.class);

		Marks[] marks =
restTemplate.getForObject("http://localhost:8282/marks/"+studid, Marks[].class);

		// Bring student data and marks data and combine
		return new StudentMakrs(student.getName(),
				Arrays.asList(marks)
				);
	}
}
```

Enhancements in the code :

Optimize RestTemplate in StudentMarksService Project

Hard coding the URL is not good practice. Since, it dynamic, it may be changed any time. And we initialized RestTemplate in the Resource class. This is second overhead, since we instantiated and initialized within getStudentMarks() method, which will be called as many time we may refresh the page. Instead of instantiating if we use DI feature and make RestTemplate a bean, we can overcome many instances of RestTemp, since the scope of spring bean is Singleton (other scopes are Prototype, WebApplicationContext scope etc).

- Update in StudentMarksServiceApplication : (main class)
    - Define a member method as below :

```
@Bean
public RestTemplate getRestTemplate(){
	return new RestTemplate();
}
```

- Update StudentMarksResource :
    - Declare a class member variable of RestTempate as below :

```
@Autowired
RestTemplate restTemplate;
```

- Now remove instantiation of RestTemplate within StudentMarksResource class.
- @Autowire : consume
- @Bean : produce

WebClient is an alternative for RestTemplate, may be in upcoming versions.

- Reactive style of programming approach.
- Update pom.xml, add another dependency.

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

- Update StudentMarksResource :

```
WebClient.Builder webClientBuilder = WebClient.builder();
```

- Below updations…

```
Student student = webClientBuilder.build()
	.get()
	.uri(uri)
	.retrieve()
	.bodyToMono(Student.class)
	.block();
```

Return a wrapper instead of collection

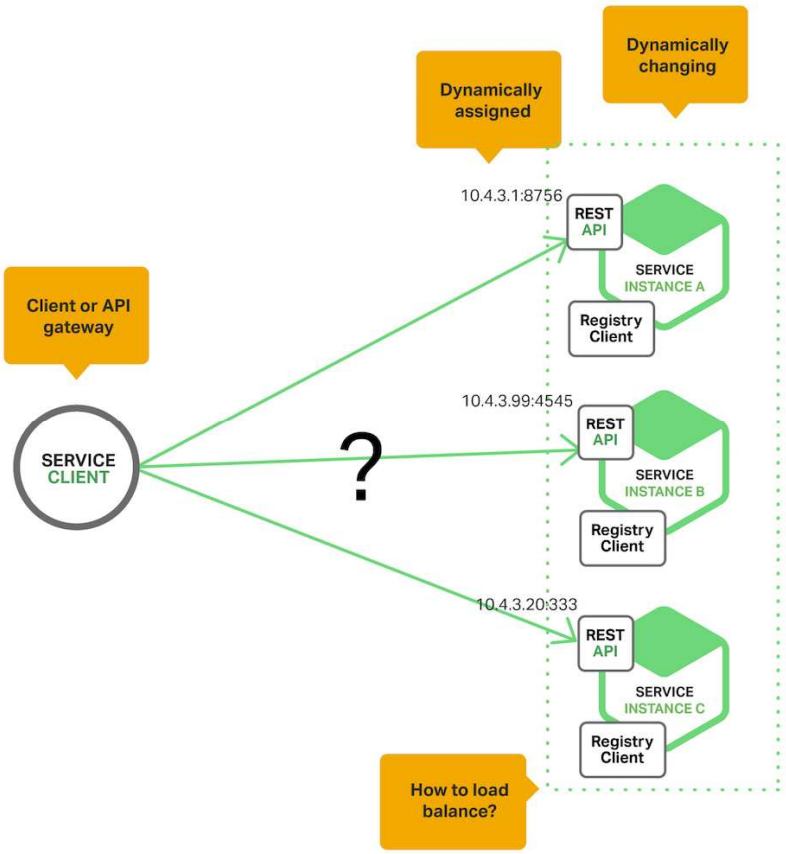
MarksResource method is returning List<Marks>, but instead of returning List, define another class MarksList and have a private member variable List<Marks> + setters + getters + constructors. And return this object from getMarks().

Service Discovery

Let's imagine that you are writing some code that invokes a service that has a REST API. In order to make a request, your code needs to know the network location (IP address and port) of a service instance. In a traditional application running on physical hardware, the network locations of service instances are

relatively static. For example, your code can read the network locations from a configuration file that is occasionally updated.

In a modern, cloud-based microservices application, however, this is a much more difficult problem to solve as shown in the following diagram.



Service instances have dynamically assigned network locations. Moreover, the set of service instances changes dynamically because of autoscaling, failures, and upgrades. Consequently, your client code needs to use a more elaborate service discovery mechanism.

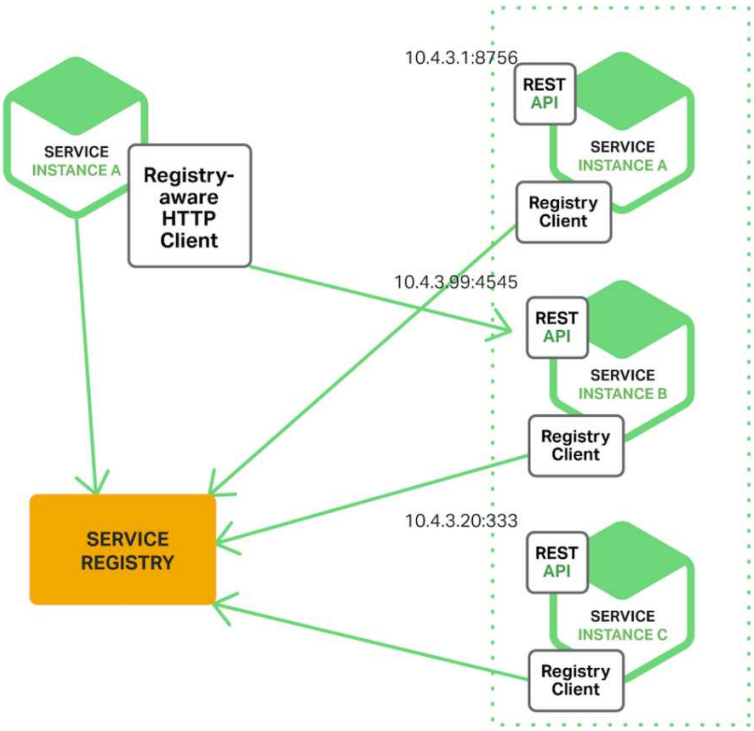
There are two main service discovery patterns: client-side discovery and server-side discovery. Let's first look at client-side discovery.

The Client-Side Discovery Pattern

When using client-side discovery, the client is responsible for determining the network locations of available service instances and load balancing requests across them. The client queries a service registry,

which is a database of available service instances. The client then uses a load-balancing algorithm to select one of the available service instances and makes a request.

The following diagram shows the structure of this pattern.

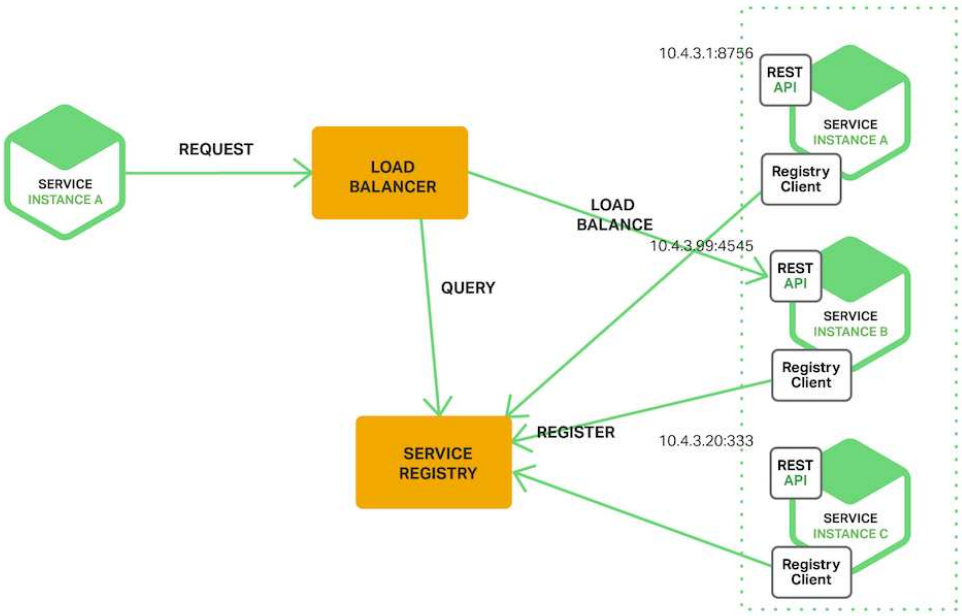


The network location of a service instance is registered with the service registry when it starts up. It is removed from the service registry when the instance terminates. The service instance's registration is typically refreshed periodically using a heartbeat mechanism.

The client-side discovery pattern has a variety of benefits and drawbacks. This pattern is relatively straightforward and, except for the service registry, there are no other moving parts. Also, since the client knows about the available services instances, it can make intelligent, application-specific load-balancing decisions such as using hashing consistently. One significant drawback of this pattern is that it couples the client with the service registry. You must implement client-side service discovery logic for each programming language and framework used by your service clients.

The Server-Side Discovery Pattern

The other approach to service discovery is the server-side discovery pattern. The following diagram shows the structure of this pattern.



The client makes a request to a service via a load balancer. The load balancer queries the service registry and routes each request to an available service instance. As with client-side discovery, service instances are registered and deregistered with the service registry.

The Service Registry

The service registry is a key part of service discovery. It is a database containing the network locations of service instances. A service registry needs to be highly available and up to date. Clients can cache network locations obtained from the service registry. However, that information eventually becomes out of date and clients become unable to discover service instances. Consequently, a service registry consists of a cluster of servers that use a replication protocol to maintain consistency.

Introduction to Eureka

The Eureka server is nothing but a service discovery pattern implementation, where every microservice is registered and a client microservice looks up the Eureka server to get a dependent microservice to get the job done.

The Eureka Server is a Netflix OSS product, and Spring Cloud offers a declarative way to register and invoke services by Java annotation. And few other products from different vendors are Ribbon, Hysterix and Zuul.

Eureka Server / Client Communication

Every microservice registers itself in the Eureka server when bootstrapped, generally using the {ServiceId} it registers into the Eureka server, or it can use the hostname or any public IP (if those are fixed). After registering, every 30 seconds, it pings the Eureka server to notify it that the service itself is available. If the Eureka server not getting any pings from a service for a quite long time, this service is

unregistered from the Eureka server automatically and the Eureka server notifies the new state of the registry to all other services.

Now one question may pop up our mind: what is the Eureka server itself?

The Eureka server is nothing but another microservice which treats itself as a Eureka client.

What I mean is that the Eureka server has to be highly available as every service communicates it to discover other services. So it is recommended that it should not be a single point of failure. To overcome it, we need multiple Eureka server instances running behind a load balancer. Now when there are multiple Eureka servers, each server needs to have synchronized registry details so that every server knows about the current state of every microservice registered in the Eureka server registry.

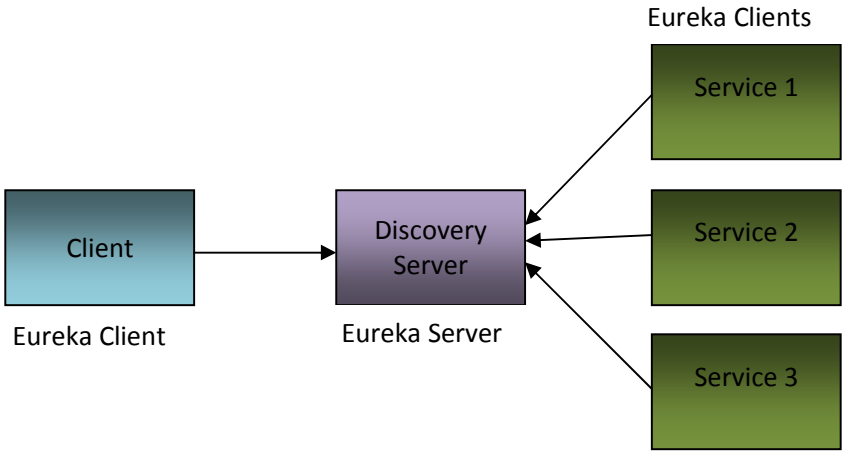
The Eureka server communicates its peer server as a client and clones the updated state of the registry, so the Eureka server itself acts as a client. We can perform this by just configuring the eureka.client.serviceUrl.defaultZone property.

The Eureka server works in two modes:

Standalone: in local, we configure a stand-alone mode where we have only one Eureka server (localhost) and the same cloning property from itself.

Clustered: we have multiple Eureka servers, each cloning its states from its peer.

The Eureka server can have a properties file and communicate with a config server as other microservices do.



Eureka Server is not a software to download. Instead it is a spring boot application. Choose **Eureka Server** dependency and another **Eureka Discovery**, which client. Use the **Eureka Discovery** for the applications to be published or applications search for web services.

Steps for Eureka Server

- Create a Spring Boot application using STS with Eureka Discovery dependency
- Make sure the application program has annotation @EnableEurekaServer

- Run the main application, with the port number **8761** by default.

Steps for Eureka Clients

- Update all 3 clients (StudentService, MarksService, StudentMarksService)
- pom.xml
    - Add spring cloud eureka client dependency
    - Spring cloud version property
- And start all three projects, are visible in eureka server web interface page.
- Update all application.properties files with appropriate application name.
    - Spring-application-name=student-service [change the name as per the application]
    - eureka.client.instance.preferIpAddress = true
- Add @EnableEurekaClient (Optional)

Consuming the eureka services

- After the services registered, we need to update the code to discover a service from eureka, instead of hard coding.
- RestTemplate getter method to be annotated as :
    - @LoadBalanced alongwith @Bean
- Update the localhost:8181 part with it's lookup name
    - http://student-service/student...
- Similarly update StudentMarksService

Client Side Load Balancing configuration

- Run the jar file from command line, with –Dserver.prort=8686 <<jarfile_name>>
    - This uses port number 8686, which overrides the default port number which may be 8282 based on the configuration in application.properties file.
- And verify the Eureka web UI. Now it is maintaining balancing at the client side.