

Spring Boot Microservices - Spring Cloud API Gateway

author: Ramesh Fadataré

MICROSERVICES

SPRING BOOT

SPRING CLOUD

In a previous couple of tutorials, we have seen:

[Spring Boot Microservices Communication Example using RestTemplate](#)

[Spring Boot Microservices Communication Example using WebClient](#)

[Spring Boot Microservices Communication Example using Spring Cloud Open Feign](#)

[Spring Boot Microservices - Spring Cloud Config Server](#)

[Spring Boot Microservices - Spring Cloud Netflix Eureka-based Service Registry](#)

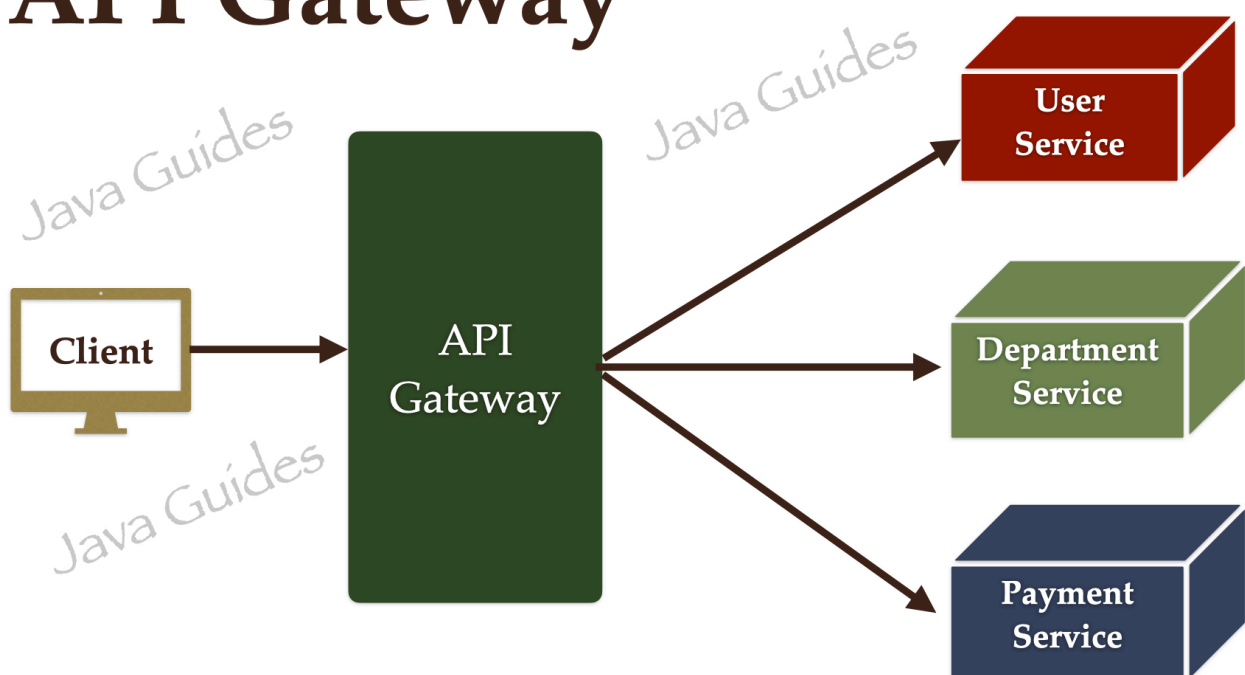
In this tutorial, we will learn how to set up an API gateway into our microservices project using the Spring Cloud Gateway library.

YouTube Video

Spring Cloud Gateway Overview

Spring Cloud Gateway provides a library for building an API Gateway on top of Spring WebFlux. Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross-cutting concerns to them such as security, monitoring/metrics, and resiliency.

API Gateway



The Spring Cloud Gateway has three important parts to it:

Route – These are the building blocks of the gateway which contain the URL to which the request is to be forwarded to and the predicates and filters that are applied to the incoming requests.

Predicate – These are the set of criteria that should match for the incoming requests to be forwarded to internal microservices. For example, a path predicate will forward the request only if the incoming URL contains that path.

Filters – These act as the place where you can modify the incoming requests before sending the requests to the internal microservices or before responding back to the client.

To know more read [Spring Cloud Gateway documentation](#).

Prerequisites

Refer to the below tutorial to create `department-service` and `user-service` microservices and configure Netflix Eureka Service Registry:

[Spring Boot Microservices - Spring Cloud Netflix Eureka-based Service Registry](#)

1. Create and Setup Spring Boot Project in IntelliJ IDEA

Let's create a Spring boot project using the [spring initializr](#).

Refer to the below screenshot to enter details while creating the spring boot application using the [spring initializr](#):

The screenshot shows the Spring Initializr web application interface. The browser address bar displays 'start.spring.io'. The page features a sidebar with a hamburger menu and a settings icon. The main content area is divided into several sections:

- Project:** Includes radio buttons for 'Maven Project' (selected) and 'Gradle Project'.
- Language:** Includes radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'.
- Spring Boot:** Includes radio buttons for various versions: '3.0.0 (SNAPSHOT)', '3.0.0 (M5)', '2.7.5 (SNAPSHOT)', '2.7.4' (selected), '2.6.13 (SNAPSHOT)', and '2.6.12'.
- Project Metadata:** Includes input fields for 'Group' (net.javaguides), 'Artifact' (api-gateway), 'Name' (api-gateway), 'Description' (Demo project for Spring Boot), and 'Package name' (net.javaguides.api-gateway).
- Packaging:** Includes radio buttons for 'Jar' (selected) and 'War'.
- Java:** Includes radio buttons for versions '19', '17' (selected), '11', and '8'.
- Dependencies:** A list of dependencies with checkboxes and descriptions:
 - Spring Boot Actuator** (OPS): Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc. (checked with a red checkmark).
 - Eureka Discovery Client** (SPRING CLOUD DISCOVERY): A REST based service for locating services for the purpose of load balancing and failover of middle-tier servers. (checked with a red checkmark).
 - Gateway** (SPRING CLOUD ROUTING): Provides a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as security, monitoring/metrics, and resiliency. (checked with a red checkmark).

At the bottom, there are three buttons: 'GENERATE' (with a keyboard shortcut), 'EXPLORE' (with a keyboard shortcut), and 'SHARE...'. A red checkmark is also visible next to the 'ADD DEPENDENCIES...' button.

Click on Generate button to download the Spring boot project as a zip file. Unzip the zip file and import the Spring boot project in IntelliJ IDEA.

Here is the `pom.xml` file for your reference:

```
<?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>2.7.4</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>nt.javaguides</groupId>
    <artifactId>api-gateway</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>api-gateway</name>
    <description>api-gateway</description>
    <properties>
        <java.version>17</java.version>
        <spring-cloud.version>2021.0.4</spring-cloud.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-gateway</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Now we have all the dependencies that we need to have in our API gateway application. So now let's configure the Routes and other API gateway-specific configurations to use in our project.

2. Enable Eureka Client using @EnableEurekaClient

The `@EnableEurekaClient` annotation makes your Spring Boot application act as a Eureka client.

```
package nt.javaguides.apigateway;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication
@EnableEurekaClient
public class ApiGatewayApplication {

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

}
```

Don't miss this step, you have to configure the API gateway as a Eureka Client for this project. Otherwise, you couldn't use discovery functions to identify the correct API from the service registry.

3. Configure Eureka Server URL

To register the Spring Boot application into Eureka Server we need to add the following configuration in our `application.properties` file and specify the Eureka Server URL in our configuration.

```
spring.application.name=API-GATEWAY
server.port=9191
eureka.instance.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
management.endpoints.web.exposure.include=*
```

4. Configuring API Gateway Routes With Spring Cloud Gateway

Now, you might be wondering how API Gateway knows the hostname or IP and port of microservices right.

When a client sends a request to the API gateway, It will discover the correct service IP and PORT using the service registry to communicate and route the request.

Let's configure Routes using properties:

```
spring.application.name=API-GATEWAY
server.port=9191
eureka.instance.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
management.endpoints.web.exposure.include=*

spring.cloud.gateway.routes[0].id=USER-SERVICE
spring.cloud.gateway.routes[0].uri=lb://USER-SERVICE
spring.cloud.gateway.routes[0].predicates[0]=Path=/api/users/**

spring.cloud.gateway.routes[1].id=DEPARTMENT-SERVICE
spring.cloud.gateway.routes[1].uri=lb://DEPARTMENT-SERVICE
spring.cloud.gateway.routes[1].predicates[0]=Path=/api/departments/**

spring.cloud.gateway.routes[2].id=DEPARTMENT-SERVICE
spring.cloud.gateway.routes[2].uri=lb://DEPARTMENT-SERVICE
spring.cloud.gateway.routes[2].predicates[0]=Path=/message/**
```

What are the properties that we set for API gateway routes?

- id – This is just an identification of the routes.
- URI – Here we can use either URL <http://localhost:8080> or lb://DEPARTMENT-SERVICE. But if we need to use the inbuilt load balancer on the Netflix Eureka server, we should use lb://DEPARTMENT-SERVICE, then the API registry will take over the request and show a load-balanced request destination to the API gateway.
- predicates – In here we can set multiple paths to identify a correct routing destination. Eg:- If the API gateway gets and request like <http://localhost:9191/api/users/1> then it will be routed into <http://localhost:8081/api/users/1>.

5. Run All the Microservices

Department Service running on port: <http://localhost:8080>

User Service running on port: <http://localhost:8081>

API Gateway service running on port: <http://localhost:9191>

Service Registry service running on port: <http://localhost:8761>

6. Verify Registered Instances in Service Registry

Go to the browser and hit this link in a new tab: <http://localhost:8761>

System Status

Environment	test	Current time	2022-10-02T20:43:34 +0530
Data center	default	Uptime	01:20
		Lease expiration enabled	true
		Renews threshold	8
		Renews (last min)	16

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - 192.168.1.11:API-GATEWAY:9191
CONFIG-SERVER	n/a (1)	(1)	UP (1) - 192.168.1.11:config-server:8888
DEPARTMENT-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.11:department-service
USER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.11:USER-SERVICE:8081

General Info

Name	Value
total-avail-memory	80mb
num-of-cpus	8
current-memory-usage	32mb (40%)

7. Testing API Gateway using Postman Client

Get Department REST API:

Note that we are using API-Gateway service port (9191) to call department-service API (port 8080)

Department Service running on port: <http://localhost:8080>

API Gateway service running on port: <http://localhost:9191>

API Gateway route the request from <http://localhost:9191/api/departments/1> to <http://localhost:8080/api/departments/1>

http://localhost:9191/api/departments/1

Save

GET

http://localhost:9191/api/departments/1

Send

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
	Key	Value	Description		

Body

Cookies

Headers (3)

Test Results

200 OK

279 ms

198 B

Save Response

Pretty

Raw

Preview

Visualize

JSON

```
1 {
2   "id": 1,
3   "departmentName": "IT",
4   "departmentAddress": "Pune",
5   "departmentCode": "IT001"
6 }
```

Get User REST API:

Note that we are using API-Gateway service port (9191) to call the user-service API port (8081).

User Service running on port: <http://localhost:8081>

API Gateway service running on port: <http://localhost:9191>

API Gateway route the request from <http://localhost:9191/api/users/1> to <http://localhost:8081/api/users/1>

http://localhost:9191/api/users/1

Save

GET

http://localhost:9191/api/users/1

Send

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
	Key	Value	Description		

Body

Cookies

Headers (3)

Test Results

200 OK

184 ms

302 B

Save Response

Pretty

Raw

Preview

Visualize

JSON

```
1 {
2   "user": {
3     "id": 1,
4     "firstName": "Ramesh",
5     "lastName": "Fadatare",
6     "email": "ramesh123@gmail.com"
7   },
8   "department": {
9     "id": 1,
10    "departmentName": "IT",
11    "departmentAddress": "Pune",
12    "departmentCode": "IT001"
13  }
14 }
```

8. Conclusion

In this tutorial, we learned how to set up an API gateway into our microservices project using the Spring Cloud Gateway library.