# SprinJava Spring Boot Microservices – Develop API Gateway Using Spring Cloud Gateway

Last Updated : 01 Aug, 2023

The API Gateway Pattern in some cases stands for “Backend for frontend”. It is basically the entry gate for taking entry into any application by an external source. The pattern is going on in a programmer’s mind while they are making the client’s application. It acts as a medium between the client applications and microservices. For example-Netflix is the most famous example of an API gateway. To know more about API Gateway refer to this article [What is API Gateway Pattern?](https://www.geeksforgeeks.org/what-is-the-api-gateway-pattern/)

**Why Implement API Gateway in Microservices?**

An API gateway simplifies the communication between a client and a service, whether that be between a user’s web browser and a server, or between a frontend application and the backend application that it relies on. **The main purpose of integrating the API gateway in microservice communication is, API Gateway acts as a single entry point to access services**. We will see the whole implementation in the example below. As of now please refer to the below image to get an idea of how the API gateway works.

**What is Spring Cloud Gateway?**

Spring Cloud Gateway provides a library for making API gateways on top of Spring and Java. It provides a flexible way of routing requests based on a number of criteria, as well as focuses on cross-cutting problems like security, resiliency, and monitoring. Some of the important features of Spring Cloud Gateway are:

* It is Built on Spring Framework 5, Project Reactor, and Spring Boot 2.0
* You can integrate Circuit Breaker with Spring Cloud Gateway
* You can integrate Spring Cloud DiscoveryClient
* Predicates and filters are specific to routes
* Path Rewriting
* It is able to match routes on any request attribute, etc.

***Note****: Please refer to this article to know more about*[*Spring Cloud Gateway*](https://www.geeksforgeeks.org/spring-cloud-gateway/)*.*

**How to Include Spring Cloud Gateway:**

**For Maven:**

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-gateway</artifactId>  
</dependency>

**For Gradle:**

implementation("org.springframework.cloud:spring-cloud-starter-gateway")

***Note****: If you include the starter, but you do not want the gateway to be enabled, set*

***spring.cloud.gateway.enabled=false***

**Developing API Gateway Using Spring Cloud Gateway**

**Step 1: Create a New Spring Boot Project in Spring Initializr**

To create a new Spring Boot project, please refer to [How to Create a Spring Boot Project in Spring Initializr and Run it in IntelliJ IDEA](https://www.geeksforgeeks.org/how-to-create-a-spring-boot-project-in-spring-initializr-and-run-it-in-intellij-idea/). For this project choose the following things

* Project: Maven
* Language: Java
* Packaging: Jar
* Java: 17

 Please choose the following dependencies while creating the project.

* Gateway (SPRING CLOUD ROUTING)

Refer to the below image

Below is the complete pom.xml file. Please cross-verify if you have missed some dependencies

* 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**>3.1.1</**version**>          <**relativePath**/> <!-- lookup parent from repository -->      </**parent**>      <**groupId**>com.gfg</**groupId**>      <**artifactId**>spring-cloud-gateway</**artifactId**>      <**version**>0.0.1-SNAPSHOT</**version**>      <**name**>Spring Cloud Gateway</**name**>      <**description**>Spring Cloud Gateway</**description**>      <**properties**>          <**java.version**>17</**java.version**>          <**spring-cloud.version**>2022.0.3</**spring-cloud.version**>      </**properties**>      <**dependencies**>          <**dependency**>              <**groupId**>org.springframework.cloud</**groupId**>              <**artifactId**>spring-cloud-starter-gateway</**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**> |

**Step 2: Make Changes in Your application.yaml file**

Now make the following changes in your [application.yaml](https://www.geeksforgeeks.org/spring-boot-application-yml-application-yaml-file/) file.

server:  
 port: 8085  
spring:  
 application:  
 name: API-GATEWAY-SERVICE  
 cloud:  
 gateway:  
 routes:  
 - id: DEMO-SERVICE  
 uri: http://localhost:9090  
 predicates:  
 - Path=/demo/\*\*

Here,

* **id**: You can give any id as of now.
* **uri**: Here you have to provide the port in which your microservice is running
* **predicates (- Path)**: Here we have provided the path “/demo/\*\*”, which means any request starting with path “/demo/\*\*” rout it to DEMO-SERVICE.

**Step 3: Develop the DEMO-SERVICE**

To create a new service, please refer to [How to Create a Spring Boot Project in Spring Initializr and Run it in IntelliJ IDEA](https://www.geeksforgeeks.org/how-to-create-a-spring-boot-project-in-spring-initializr-and-run-it-in-intellij-idea/). For this project choose the following things

* Project: Maven
* Language: Java
* Packaging: Jar
* Java: 17

 Please choose the following dependencies while creating the project.

* Spring Web

In this MIcrosevice we have created a simple REST API in our controller class.

* Java

|  |
| --- |
| **package** com.gfg.demo.controller;    **import** org.springframework.http.ResponseEntity;  **import** org.springframework.web.bind.annotation.GetMapping;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;    @RestController  @RequestMapping("/demo")  **public** **class** DemoController {        @GetMapping("/gfg")  **public** ResponseEntity<String> getAnonymous() {  **return** ResponseEntity.ok("Welcome to GeeksforGeeks");      }    } |

Now make the following changes in your application.properties file.

server.port=9090

Now run your application and test it out.

**Step 4: Testing in Postman**

Now let’s test our API. Hit the following URL

http://localhost:9090/demo/gfg

And you are going to get a response like this

Now we can get the same response by using our API gateway port which is 8085. Now hit the following URL

http://localhost:8085/demo/gfg

And you are going to get a response like this

So this is how the API gateway works. If you have hundreds of microservices then you don’t need to remember the port of all microservices. You can just configure them in your API Gateway and you can access all your API by using only one port.

**g Cloud GatewaySs**

Last Updated : 07 Jun, 2023----------------------

If you are aware of a [microservices architecture](https://www.geeksforgeeks.org/what-is-microservice-architecture-and-why-to-use-them/), then there we have several spring boot applications (microservices) running on different ports or routes. An API gateway acts as a single point of entry for a collection of microservices. In simple words, all microservices can be accessed through a single port or route. It is a non-blocking and reactive gateway that provides several features like routing, filtering, load balancing, circuit breaking, and more. In this article first, we’ll look at the spring cloud gateway architecture and then implement it.

**Spring Cloud Gateway Architecture**

The main components of the spring cloud gateway are:

1. **Route:** It is an important component in the spring cloud gateway. I consist of ID, destination URI, predicates, and filters.
2. **Predicates:**It is the same as Java 8 function predicate. A Predicate simply evaluates the input against a condition defined by the Predicate and returns true if the condition is satisfied. Here the Predicate is used to match the HTTP requests. A route is matched if the Predicate returns true.
3. **Filter Chain:** It is a series of filters applied to incoming requests and responses. It can be used for various purposes like authentication, requests or response transformation, and many more.

*Spring Cloud Gateway Architecture*

The process begins with the client sending a request to the API gateway. The request first goes to the Gateway mapping handler. It uses Predicate to check whether a request matches a route. The request is then transferred to Gateway Web Handler. It passes the request through the Filter Chain specific to the request. Here the filters can be considered in two categories. When requests arrive all the pre-filter logic is executed. After the request is made all the post-filter logic is executed.

**Project Configuration**

Each spring boot application that we’ll create will be the Maven project with the following configurations

* Language: Java
* Spring Boot Version: 3.0.6
* Java Version: 17
* Packaging: Jar

**First Microservice Implementation**

Below is what the project structure looks like

*Project structure of Microservice 1*

Include the below dependencies in the pom.xml.

* 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](http://maven.apache.org/POM/4.0.0%C2%A0)  <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**>3.0.6</**version**>          <**relativePath**/> <!-- lookup parent from repository -->      </**parent**>      <**groupId**>com.microservice</**groupId**>      <**artifactId**>Microservice1</**artifactId**>      <**version**>0.0.1-SNAPSHOT</**version**>      <**name**>Microservice1</**name**>      <**description**>Demo project for Spring Boot</**description**>      <**properties**>          <**java.version**>17</**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**>          </**dependency**>      </**dependencies**>        <**build**>          <**plugins**>              <**plugin**>                  <**groupId**>org.springframework.boot</**groupId**>                  <**artifactId**>spring-boot-maven-plugin</**artifactId**>              </**plugin**>          </**plugins**>      </**build**>    </**project**> |

Now, provide a name for the spring application and configure the port number. To do so, make the following changes to the application.properties file

spring.application.name=MicroService1

server.port=8081

The above configuration ensures that our first microservice runs on port 8081. Now let’s create the Controller class.

**Controller Class**

* Java

|  |
| --- |
| **import** org.springframework.http.ResponseEntity;  **import** org.springframework.web.bind.annotation.GetMapping;  **import** org.springframework.web.bind.annotation.RestController;    @RestController  @RequestMapping("/serviceA")  **public** **class** Controller {      @GetMapping("/displayMessage")  **public** ResponseEntity<String> showMessage(){  **return** ResponseEntity.ok("Microservice 1 controller executed");      }  } |

The above controller class only contains one API “displayMessage” which is accessible via GET request. We’ll fetch this API through Spring Cloud Gateway soon.

**Second Microservice Implementation**

The second microservice has the same project structure and dependencies as that of the first microservice. Here also provide a name to the application and set the port number in the application.properties file.

spring.application.name=MicroService2

server.port=8082

**Controller Class**

Here also, We create a GET API inside the controller and then access it through Spring Cloud Gateway.

* Java

|  |
| --- |
| **import** org.springframework.http.ResponseEntity;  **import** org.springframework.web.bind.annotation.GetMapping;  **import** org.springframework.web.bind.annotation.RestController;    @RestController  @RequestMapping("/serviceB")  **public** **class** Controller {      @GetMapping("/displayMessage")  **public** ResponseEntity<String> showMessage(){  **return** ResponseEntity.ok("Microservice 2 controller executed");      }  } |

Now, we have both of our microservices reading and running on port numbers 8081 and 8082 respectively. Now let’s create a Spring Cloud Gateway running at port 8083 and then we’ll see whether both microservices can be accessed from port 8083 or not.

**Spring Cloud Gateway Implementation**

There are two ways to create an API gateway.

* **Programmatic configuration:** Here we create Spring Cloud Gateway as Java bean. The routes, predicates, and all are created as a traditional Java program.
* **Property configuration:** Here, we create components of Spring Cloud Gateway as properties in the application.properties or application.yml file.

**Spring Cloud Gateway Implementation using Properties**

Create a Separate Spring boot application to create a gateway. Include the following dependencies in the pom.xml file.

* 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**>3.0.6</**version**>          <**relativePath**/> <!-- lookup parent from repository -->      </**parent**>      <**groupId**>com.gateway</**groupId**>      <**artifactId**>Gateway</**artifactId**>      <**version**>0.0.1-SNAPSHOT</**version**>      <**name**>Gateway</**name**>      <**description**>Demo project for Spring Boot</**description**>      <**properties**>          <**java.version**>17</**java.version**>          <**spring-cloud.version**>2022.0.2</**spring-cloud.version**>      </**properties**>      <**dependencies**>          <**dependency**>              <**groupId**>org.springframework.boot</**groupId**>              <**artifactId**>spring-boot-starter-webflux</**artifactId**>          </**dependency**>          <**dependency**>              <**groupId**>org.springframework.cloud</**groupId**>              <**artifactId**>spring-cloud-starter-gateway</**artifactId**>          </**dependency**>            <**dependency**>              <**groupId**>org.springframework.boot</**groupId**>              <**artifactId**>spring-boot-starter-test</**artifactId**>              <**scope**>test</**scope**>          </**dependency**>          <**dependency**>              <**groupId**>io.projectreactor</**groupId**>              <**artifactId**>reactor-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**> |

The project structure looks like

*Project Structure*

Now create a file with the name application.yml inside the resources folder and populate it with the following:

server:

port: 8083

spring:

cloud:

gateway:

routes:

- id: Microservice1

uri: http://localhost:8081/

Predicates:

- Path=/serviceA/\*\*

- id: Microservice2

uri: http://localhost:8082/

Predicates:

- Path=/serviceB/\*\*

**Spring Cloud Gateway Programmatic Implementation**

here also, the project structure and dependencies remain the same. Also apart from port configuration, remove everything from the YML file. Now create a bean inside GatewayApplication.java class as follows:

* Java

|  |
| --- |
| **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  **import** org.springframework.cloud.gateway.route.RouteLocator;  **import** org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;  **import** org.springframework.context.annotation.Bean;    @SpringBootApplication  **public** **class** GatewayApplication {    **public** **static** **void** main(String[] args) {          SpringApplication.run(GatewayApplication.**class**, args);      }        @Bean  **public** RouteLocator routerBuilder(RouteLocatorBuilder routeLocatorBuilder){  **return** routeLocatorBuilder.routes()                          .route("Microservice1",r->r.path("/serviceA/\*\*")                                  .uri("<http://localhost:8081/>"))                          .route("Microservice2",r->r.path("/serviceB/\*\*")                                  .uri("<http://localhost:8082/>")).build();      }  } |

Now, on running the gateway application using any of the above methods, we can see that both microservices can be accessed with a single port as below

*Accessing the first microservice*

*Accessing the second microservice*