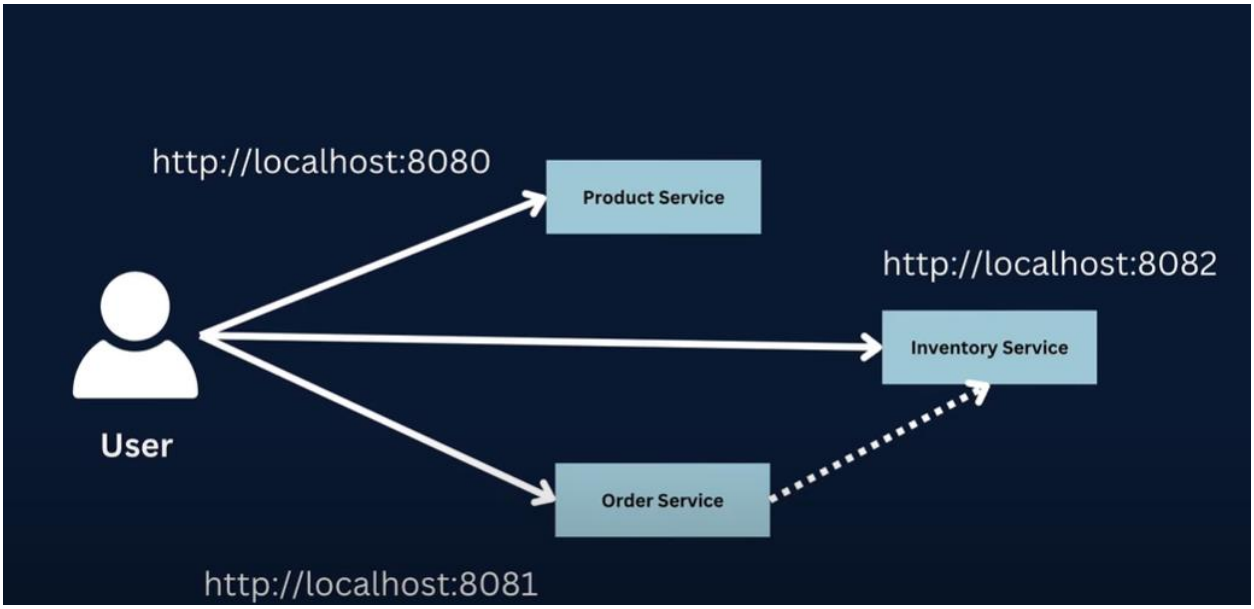
Spring Boot Microservices Tutorial - Part 3

In Part 3 of this **Spring Boot Microservices Tutorial** series, we will implement the API Gateway pattern using the Spring Cloud Gateway MVC library.

What is an API Gateway?

An API Gateway also called an Edge Server, acts as an entry point for our microservices, so that external clients can access the services easily. It also helps us to handle cross-cutting concerns like Monitoring, Security, etc. In some instances, API Gateway also acts as Load Balancers.



In the above microservices architecture, each microservice runs on a different port. If any changes are made to one microservice, the client-side also needs to be updated to reflect the service. This approach is not ideal. To resolve this issue, we can use an API Gateway.

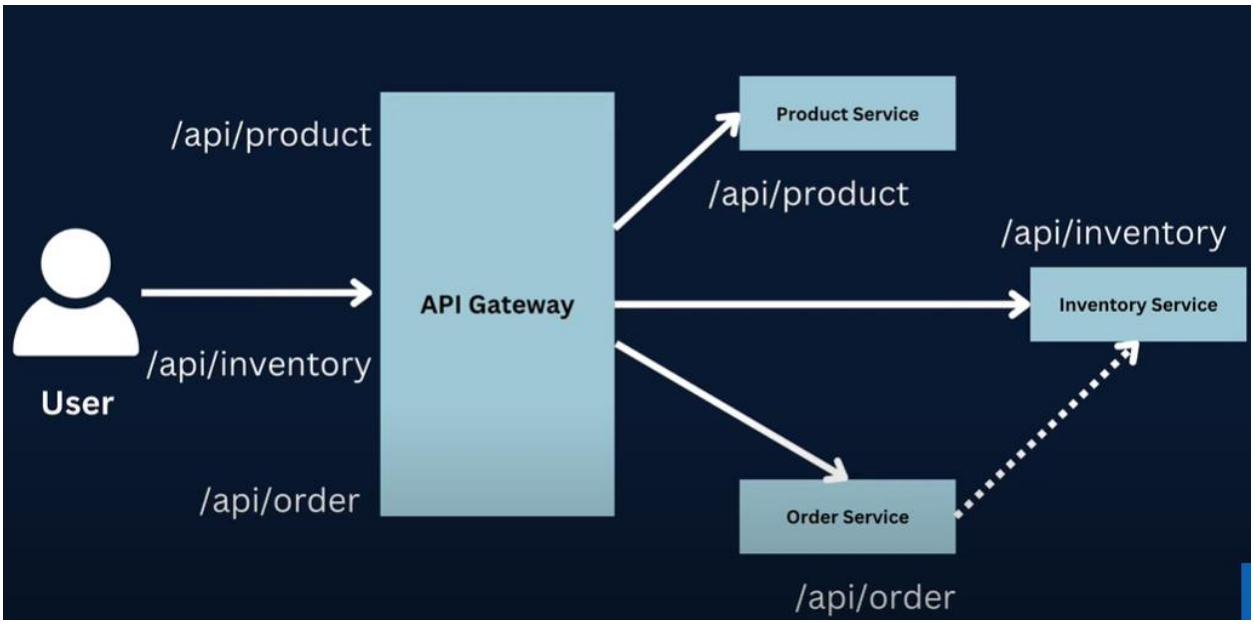
Why to use API Gateway?

In our microservice project landscape, we have 3 services accessible to the user:

- Product Service
- Order Service
- Inventory service

For example, imagine that external clients like Web and Mobile applications consume these three independent services through the exposed endpoints. If the internal implementation of these services changes, then also the clients need to update the Endpoints on their side.

To work around this issue, we use an API Gateway as the facade that provides an abstraction over the internal microservices.



Benefits of using API gateway:

Handles cross cutting concerns like

- security
- Monitoring
- Rate Limiting
- SSL Termination

Drawbacks of using API Gateway:

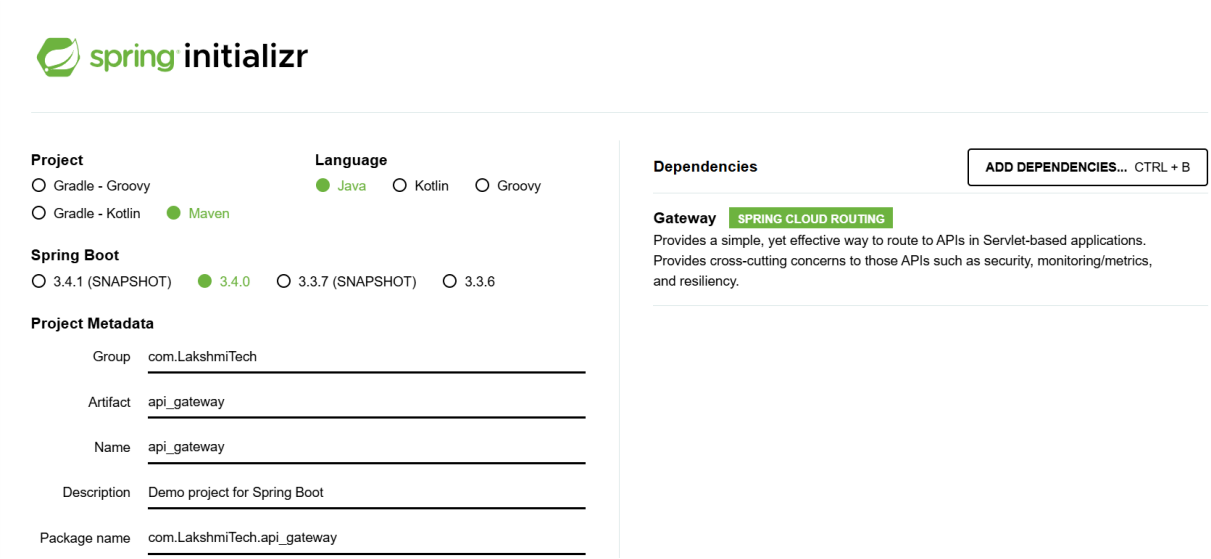
- Increases complexity as we have to maintain yet another component in our system landscape
- Needs lot of effort to maintain it, as it will be a single point of failure.

single point of failure → means, if API gateway is not working, then we cant able to access our application completely.

- Increased latency for the client requests. → because if we call API, it will call again internal microservice.

Spring Cloud Gateway MVC

Spring Cloud Gateway MVC is a library under the Spring Cloud project, that provides the API Gateway functionality. Let's go ahead and create the API Gateway for our project, as usual, we use the start.spring.io website to create the project.



Make sure you use the above configuration and click on Generate Project to download the source code to your machine.

As we learned before, an API Gateway acts as an abstraction over the microservices, and it forwards the request from the client to the relevant microservices.

To implement this feature, Spring Cloud Gateway uses the below building blocks:

- Routes
- Predicates
- Filters

Routes

A Route is the basic building block of the gateway, it can be defined using a uniqueId, a destination URI, and a collection of predicates and filters

Predicates

A Predicate is nothing but a criteria or a condition that you define to match against the incoming HTTP Request, for example, you can create a routing rule where you want to route the requests that have a specific Header and Request Parameter to Service A, then you can consider the headers and request parameters you want to match against the request as predicates.

Filters

Filters are components that allow you to modify the requests and responses before they are sent to the destination.

Let's see how we can implement the API Gateway in our project using Spring Cloud Gateway MVC.

Note that we will be using Spring Cloud Gateway MVC, but not Spring Cloud Gateway which is based on reactive stack backed by Spring Webflux.

Here are the routing rules we will implement:

- If a request matches the path - /api/product, then forward it to Product Service
- If a request matches the path - /api/order, then forward it to Order Service
- If a request matches the path - /api/inventory, then forward it to Inventory Service

Coding

Let's start developing our API Gateway, once you open the project downloaded from start.spring.io, you should see the below **pom.xml** file

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
```

```xml
<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>3.2.4</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.programming.techie</groupId>
    <artifactId>api-gateway</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>api-gateway</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>21</java.version>
        <spring-cloud.version>2023.0.1</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-mvc</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</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>
```

And the main Spring Boot application class, **ApiGatewayApplication.java**

```
package com.programming.techie;

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

@SpringBootApplication
public class ApiGatewayApplication {

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

}
```

Now it's time to create the routing rules defined above, for that we can follow 2 approaches

- Using Java API
- Using Property files

We will go with the approach of using Java API in this tutorial, for that let's create a class called Routes.java

Routes.java

```java
package com.programming.techie.routes;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.function.RequestPredicates;
import org.springframework.web.servlet.function.RouterFunction;
import org.springframework.web.servlet.function.ServerResponse;

import java.net.URI;

import static
org.springframework.cloud.gateway.server.mvc.filter.CircuitBreakerFilterFunctions.circuit
Breaker;
import static
org.springframework.cloud.gateway.server.mvc.handler.GatewayRouterFunctions.route;
import static org.springframework.cloud.gateway.server.mvc.handler.HandlerFunctions.http;

@Configuration
public class Routes {

    @Bean
    public RouterFunction<ServerResponse> productServiceRoute(){
        return GatewayRouterFunctions.route("product_service")
                .route(RequestPredicates.path("/api/product"),
HandlerFunctions.http("http://localhost:8080"))
                .build();
    }
}
```

The code snippet defines a route in **Spring Cloud Gateway** using the **functional programming style**.

`@Bean:` Registers this method's return value as a Spring-managed bean.

`RouterFunction<ServerResponse>`: Represents a route that handles requests and returns server responses.

`route("order_service"):` Starts defining a route with the ID `order_service`. This ID is used for identifying and debugging routes.

`RequestPredicates.path("/api/order")`: Matches incoming requests with the path `/api/product`. When a client sends a request to `/api/order`, this route processes it.

`http("http://localhost:8081"):` For requests matching `/api/order`, the Gateway forwards them to `http://localhost:8081`.

`.build();` → Finalizes the route definition.

The above code defines a route to the product service, the **route()** method takes in two arguments one for the path which is the predicate we want to match in this case (/api/product), and the second argument is `http("<target-destination-url>")` which points to the target destination ie. product service that is running at **http://localhost:8080**.

We will see how to use Filters in the upcoming section when we implement Circuit Breakers for resiliency.

Let's add also the remaining routes for the order service and inventory service

```
package com.programming.techie.routes;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpStatus;
import org.springframework.web.servlet.function.RequestPredicates;
import org.springframework.web.servlet.function.RouterFunction;
import org.springframework.web.servlet.function.ServerResponse;

import java.net.URI;

import static
org.springframework.cloud.gateway.server.mvc.filter.CircuitBreakerFilterFunctions.circuit
Breaker;
import static
org.springframework.cloud.gateway.server.mvc.handler.GatewayRouterFunctions.route;
import static org.springframework.cloud.gateway.server.mvc.handler.HandlerFunctions.http;

@Configuration
public class Routes {
    @Bean
    public RouterFunction<ServerResponse> productServiceRoute(){
        return GatewayRouterFunctions.route("product_service")
                .route(RequestPredicates.path("/api/product"),
HandlerFunctions.http("http://localhost:8080"))
                .build();
    }
```

```
    @Bean
    public RouterFunction<ServerResponse> orderServiceRoute(){
        return GatewayRouterFunctions.route("order_service")
                .route(RequestPredicates.path("/api/order"),
HandlerFunctions.http("http://localhost:8085"))
                .build();
    }

    @Bean
    public RouterFunction<ServerResponse> inventoryServiceRoute(){
        return GatewayRouterFunctions.route("inventory_service")
                .route(RequestPredicates.path("/api/inventory"),
HandlerFunctions.http("http://localhost:8084"))
                .build();
    }
}
```

You can observe that the other routes have very similar code, but the only differences are obvious, with the path being /api/order, routed to the Order Service, and the path /api/inventory to the Inventory Service.

Finally, let's add a property in the application.properties file to make sure that the api-gateway service runs on port 9000 as 8080 is already taken by the product service.

```
server.port=9000
```

Now if you make an HTTP GET request to the URL:

http://localhost:9000/api/product then you should see the below response

```
[
{
"id": "661b5c40ad645e4a98d0f623",
"name": "iPhone 15",
"description": "iPhone 15 is a smartphone from Apple",
"price": 1000
}
]
```

That's it for this part, in the next part we will discuss how to implement security in our project by integrating OAuth2 using Keycloak.