**Project**

Building a video streaming system that works with any camera and streams live events to your website:

1. Project Planning

* Objective: Understand the scope of the project.
* Steps:
  + Define project goals: streaming video from any camera to a website in real-time.
* Decide on the core technologies:
  + Java for backend API.
  + OpenCV, FFmpeg, or GStreamer for capturing video from any camera.
  + RTMP or WebRTC for real-time streaming.
  + Plan the architecture: camera → streaming server → CDN → website.
  + Set up a version control system (e.g., GitHub, GitLab) for collaboration and tracking progress.

1. Camera Integration

* Objective: Capture video from any camera.
* Steps:
  + Research supported cameras (USB webcams, IP cameras, mobile device cameras).
* Choose a library for camera integration:
  + OpenCV: for direct access to USB cameras.
  + FFmpeg or GStreamer: for advanced video processing and IP cameras using RTSP.
  + Test camera input with the chosen technology to ensure the video feed can be captured.
  + Set up a simple test environment where the video is captured and displayed on the local machine.

1. Set Up Streaming Server

* Objective: Stream video from the camera to your server.
* Steps:
  + Install and configure NGINX with the RTMP module (or use a service like Wowza).
  + Use FFmpeg to stream the camera feed to the NGINX server over RTMP.
  + Write a script to capture the video feed and push it to the RTMP server.
  + Test streaming locally by pushing the video from the camera to the RTMP server and confirming that it can be received by the server.
  + If using WebRTC, configure a WebRTC server (using tools like mediasoup or Janus).

1. Backend API Development

* Objective: Manage live streams and handle server-side logic.
* Steps:
  + Set up a Java Spring Boot project.
* Create RESTful APIs for:
  + Managing streams (start, stop).
  + Retrieving stream URLs for embedding on the website.
  + User authentication (JWT).
  + Set up a database (PostgreSQL, MySQL) for storing event information, stream metadata, and user data.
  + Integrate the backend with the streaming server so it can control stream actions (start/stop) and retrieve live stream URLs.

1. Transcoding and Encoding

* Objective: Ensure the video is in a web-friendly format.
* Step:
  + Use FFmpeg to transcode the video to H.264 (for video) and AAC (for audio).
  + Ensure that the stream is converted to HLS (HTTP Live Streaming) or WebRTC for browser compatibility.
  + Implement on-the-fly transcoding on the media server, especially if streaming in multiple quality formats (low, medium, high).
  + Test the transcoding process by ensuring the stream can be played in a web player.

1. Web Client for Video Playback

* Objective: Display the live stream on your website.
* Steps:
  + Build the front end using HTML5 and integrate an HTML5 video player such as Video.js or HLS.js.
  + Configure the video player to pull the live stream using HLS or WebRTC (whichever streaming protocol you've chosen).
  + Test the stream on the front end by embedding the video player in a basic HTML page and accessing the live stream from the media server.
  + Add UI elements for stream control (start, stop, volume control).

1. User Authentication and Access Control

* Objective: Secure access to the live stream.
* Steps:
  + Implement JWT (JSON Web Token) authentication in your backend API to control access to live streams.
  + Create user roles (admin, viewer) and implement role-based access controls to manage who can start, stop, or view streams.
  + Integrate JWT authentication with your video player, so only authenticated users can view the live stream.
  + Test access control by creating a sample user and ensuring they can access the stream based on their permissions.

1. CDN Integration for Scalability

* Objective: Ensure your stream can handle many viewers.
* Steps:
  + Integrate a Content Delivery Network (CDN) like AWS CloudFront, Cloudflare, or Akamai to handle global distribution.
  + Ensure the CDN supports HLS or WebRTC to ensure smooth playback across various devices.
  + Test the CDN setup by simulating multiple viewers and monitoring performance using the CDN's analytics.
  + Set up caching and buffering strategies to reduce server load and improve stream quality for users.

1. Monitoring and Analytics

* Objective: Track performance and user engagement.
* Steps:
  + Set up monitoring for the media server and backend using tools like Grafana and Prometheus.
  + Add real-time stream analytics (viewers, latency, bitrate) to track stream performance.
  + Use Google Analytics or a similar tool to track user engagement and stream viewership on your website.
  + Implement alerts (e.g., Slack, email) for issues like server downtime, high latency, or high CPU usage.

1. Testing and Deployment

* Objective: Ensure everything works seamlessly in production.
* Steps:
  + Perform stress testing using tools like JMeter or Locust to simulate many viewers accessing the stream.
  + Test the entire flow: camera → streaming server → backend → frontend.
  + Set up a CI/CD pipeline for continuous integration and deployment to easily push updates to the server (e.g., GitHub Actions, Jenkins).
  + Deploy the backend and frontend on cloud platforms like AWS, Google Cloud, or DigitalOcea.

1. Go Live and Maintenance

* Objective: Launch the live streaming system and maintain it.
* Steps:
  + Deploy your backend and frontend on production servers.
  + Announce your live event platform and perform a soft launch to monitor performance.
  + Continue monitoring the system for performance issues or bugs.
  + Regularly update your backend and media server to keep up with new features or security patches.
  + Add new features like chat integration, recording streams, or multiple streams.

High-Level Summary:

1. Camera Integration (OpenCV, FFmpeg, or IP Camera RTSP)
2. Streaming Setup (NGINX RTMP/ WebRTC Server)
3. Backend API Development (Java Spring Boot for stream management)
4. Transcoding (FFmpeg for H.264 + HLS/WebRTC)
5. Web Frontend (HTML5 video player)
6. Security (JWT for user access)
7. CDN (Cloudflare, AWS CloudFront for scalability)
8. Monitoring (Grafana, Prometheus)
9. Testing (Stress testing with multiple viewers)
10. Deployment (AWS, GCP for production hosting)

**Streaming Live Video and Storing Videos with NGINX Open Source**

### **Installing the Build Tools**

The NGINX configurations presented in this blog do not include security measures to restrict who can watch your video stream. There are a variety of ways to secure your streams with the front‑end application your viewers use to watch the video, such as allowing access only from certain IP addresses or requiring viewers to authenticate.

### Installing the Build Tools

Before compiling NGINX, you need to have some basic build tools installed: autoconf, gcc, git, and make. To download and install them, run the command for your operating system

**Ubuntu**

$ **sudo yum install git**

#### Installing Dependencies with a Package Manager

$ **sudo apt install libpcre3-dev libssl-dev zlib1g-dev**

### Compiling NGINX with the RTMP Module

$ **cd /path/to/build/dir -> go to a directory to have the file**

$ **git clone** <https://github.com/arut/nginx-rtmp-module.git>

$ **git clone** <https://github.com/nginx/nginx.git>

$ **cd nginx**

$ **./auto/configure --add-module=../nginx-rtmp-module**

$ **make**

$ **sudo make install**

### **Configuring NGINX**

You can configure NGINX to stream video using one or both of the HTTP Live Streaming (HLS) and Dynamic Adaptive Streaming over HTTP (DASH) protocols. The protocols provide the same functionality, so choosing between them is really a matter of preference. If you’re not familiar with them, see **[HLS vs DASH](https://www.vidbeo.com/blog/hls-vs-dash" \t "_blank)**

Create Nginx.conf for either HLS or DASH

For HLS, the configuration is as follows.

rtmp {

server {

listen 1935;

application live {

live on;

interleave on;

hls on;

hls\_path /tmp/hls;

hls\_fragment 15s;

}

}

}

http {

default\_type application/octet-stream;

server {

listen 80;

location /tv {

root /tmp/hls;

}

}

types {

application/vnd.apple.mpegurl m3u8;

video/mp2t ts;

text/html html;

}

}

For DASH, the configuration is as follows

rtmp { server {

listen 1935;

application live {

live on;

dash on;

dash\_path /tmp/dash;

dash\_fragment 15s;

}

}

}

http {

server {

listen 80;

location /tv {

root /tmp/dash;

}

}

types {

text/html html;

application/dash+xml mpd;

}

}

**BOTH CAN BE COMBINED INTO ON CONFIGURATION**

### Validating the Configuration and Starting NGINX

It’s always a good idea to validate your NGINX configuration to make sure there are no syntactic errors. Run this command:

$ **cd nginx** sudo /usr/local/nginx/sbin/nginx

$ **sudo /usr/local/nginx/sbin/nginx -t** nginx: the configuration file filename syntax is ok

nginx: configuration file filename test is successful

Then run this command to start NGINX:

$ **sudo /usr/local/nginx/sbin/nginx**

**Testing the Playback Methods**

**Using OBS Studio**

**[OBS Studio](https://obsproject.com/" \t "_blank)** is a commonly used open source tool that allows you to livestream from your workstation to your NGINX server by configuring a custom RTMP server. Configure OBS to stream to **rtmp://***NGINX\_server***/tv/tv2**, where *NGINX\_server* is the IP address or hostname of your NGINX server. No stream key is required.

**Using local and ffmpeg**

We start by running the **stream.sh** script, which has these contents:

Sample script for **stream.sh**

ffmpeg -re -I bbb\_sunflower\_1080p\_60fps\_normal.mp4 -vcodec copy -loop -1 -c:a aac -b:a 160k -ar 44100 -strict -2 -f flv rtmp:192.168.1.138/live/bbb

Command Breakdown

1. `ffmpeg`

This is the command-line tool used to process multimedia files, including video and audio. In this case, it's being used for streaming.

2.`-re`

This flag tells FFmpeg to read the input file in real-time. This is important for streaming as it simulates the input's natural speed instead of processing it as fast as possible. It ensures the stream goes out at a steady, realistic rate.

3. `-i bbb\_sunflower\_1080p\_60fps\_normal.mp4`

This specifies the input file. Here, the input is `bbb\_sunflower\_1080p\_60fps\_normal.mp4`, which is a video file in MP4 format.

4. `-vcodec copy`

This flag tells FFmpeg to copy the video codec without re-encoding. It means that the video will be streamed in its original codec, which saves processing time and resources. In this case, the video codec in the input file will remain unchanged during the stream.

5. `-loop -1`

This option makes the video file loop indefinitely, streaming it continuously until you stop the process manually. If you don't want the video to loop, you can omit this flag.

6. `-c:a aac`

This specifies the audio codec to be used. Here, the command tells FFmpeg to use the AAC audio codec (`aac`), which is a popular format for streaming because of its efficiency and quality.

7. `-b:a 160k`

This sets the bitrate for the audio stream to 160 kbps. The higher the bitrate, the better the audio quality, but it also requires more bandwidth. 160 kbps is a good balance between quality and bandwidth usage for streaming.

8.`-ar 44100`

This sets the audio sample rate to 44.1 kHz (44100 Hz), which is the standard for most audio, including music and video streaming. It's commonly used for most consumer devices.

9. `-strict -2`

This is a legacy option to use experimental features. It's used here because some versions of FFmpeg require this flag when using the AAC codec. The AAC codec was considered experimental in older versions of FFmpeg, so `-strict -2` allows you to use it. You can remove this if using a newer FFmpeg version where AAC is no longer experimental.

10. `-f flv`

This flag sets the output format to FLV (Flash Video). The RTMP (Real-Time Messaging Protocol) streaming protocol often requires FLV as the container format. Although FLV is an older format, it is still widely used for live streaming with RTMP.

11. `rtmp://192.168.1.138/live/bbb`

This is the RTMP server URL where the stream will be sent.

- `rtmp://192.168.1.138`: The IP address of the machine running the RTMP server.

- `/live/bbb`: This part specifies the RTMP application (`live`) and the stream key or name (`bbb`). The stream key identifies the specific stream to the RTMP server, allowing multiple streams under different keys.

**API STRUTURE PART – JAVA(BACKEND)**

**Using the Micro Service Approach to this.**

**A diagram of a software application

Description automatically generated**

**START WITH THE SERVER REGISTRY- USING EUREKA SERVER**

**What is Eureka Server:**

Eureka Server is a service discovery tool provided by Netflix as part of the Spring Cloud Netflix ecosystem. It plays a crucial role in microservice architectures by allowing services to discover and communicate with each other without hard coding their locations. Essentially, it helps manage dynamic IP addresses in cloud environments where services can start, stop, and scale frequently.

**Key Concepts of Eureka Server:**

1. Service Registry:

Eureka Server maintains a registry of all the microservices (clients) running in the system. Each service registers itself with the Eureka Server when it starts, providing its hostname, port, and other metadata. It also sends periodic heartbeats to notify the server that it’s still active.

2. Service Discovery:

When one microservice (say `MovieService`) needs to communicate with another microservice (say `ExternalApiService`), it doesn’t need to know the exact IP or port of that service. Instead, it can query Eureka to find the location of `ExternalApiService`. Eureka provides the address dynamically, allowing for service scalability and failover handling.

3. High Availability and Load Balancing:

Eureka supports high availability and replication. Multiple Eureka Servers can be run in a cluster, replicating data between them. It also supports client-side load balancing using **Ribbon**, meaning it can balance traffic across multiple instances of a microservice registered with it.

**Components:**

- Eureka Server: This is the main registry where all the microservices register themselves. You can run it as a standalone application.

- Eureka Client: Each microservice in your architecture is typically a Eureka client. It registers itself with the Eureka server and can also discover other services via Eureka.

**How Eureka Works:**

1.Registration: When a microservice starts, it registers itself with Eureka by sending its metadata (like its hostname, port, etc.). The service will also send heartbeats to the Eureka Server to indicate that it’s alive.

2. Discovery: When a service needs to call another service, it queries the Eureka Server to get the location of the target service. The client can then make direct calls to the instance.

3.Load Balancing: If multiple instances of a service are running, Eureka can return multiple addresses. The client can use a load-balancing strategy (like round-robin) to distribute requests across instances.

**Example Workflow:**

1.Service Registration:

`movie-service` starts up and registers with the Eureka Server by sending its address and metadata.

2. Service Discovery:

`movie-service` needs data from `external-api-service`. It asks the Eureka Server for a list of available instances.

3. Call the Service:

Eureka returns the available instances, and `movie-service` calls one of the instances of `external-api-service`.

**Eureka Configuration:**

1. Setting up a Eureka Server:

You can set up a Eureka Server using Spring Boot with just a few annotations.

**Add dependencies in your `pom.xml` (for Maven):**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

**Enable Eureka Server in your Spring Boot application:**

@SpringBootApplication  
@EnableEurekaServer  
*public class* ServiceRegistryApplication {  
  
 *public static void* main(String[] args) {  
 SpringApplication.*run*(ServiceRegistryApplication.*class*, args);  
 }  
  
}

**Configure `application.properties**

*server.port*=8761  
  
*# Eureka Server Configuration  
eureka.instance.hostname*=localhost  
  
*# because it is a server, it should not register itself  
eureka.client.register-with-eureka*=*false  
  
# because it is a server, it should not fetch registry  
eureka.client.fetch-registry*=*false*

Advantages of Eureka Server:

-Dynamic Discovery: Services don’t need to know the addresses of other services, reducing tight coupling.

- Fault Tolerance: If one service instance goes down, Eureka helps route traffic to healthy instances.

- Scalability: Eureka handles multiple service instances and balances load across them.

Alternatives:

- Consul and Zookeeper are also popular tools for service discovery and registration, but Eureka integrates deeply with the Spring ecosystem, making it a preferred choice in Spring Boot projects.

By using Eureka Server, you gain the ability to manage a dynamic and scalable microservice architecture with ease

**Movie Catalogue Service**

Here we are using just 2 dependencies.

1 Spring web

2 Eureka Client server -

**Setting up Eureka Client (Microservices):**

In your microservices, add the Eureka Client dependency:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

Annotate your application with `@EnableEurekaClient` and configure the Eureka server URL in `application.properties `:

*spring.application.name*=movie-catalogs  
  
*server.port*=8090  
  
*# Eureka Configuration  
eureka.client.service-url.default-zone*=http://localhost:8761/eureka

*spring.application.name*=movie-streaming  
  
*server.port*=8091  
  
*# Eureka Configuration  
eureka.client.service-url.default-zone*=http://localhost:8761/eureka  
  
*logging.level.org.springframework.web*=DEBUG

**Enable Eureka Client in your Spring Boot application:**

@SpringBootApplication  
@EnableDiscoveryClient  
*public class* MovieCatalogsApplication {  
  
 *public static void* main(String[] args) {  
 SpringApplication.*run*(MovieCatalogsApplication.*class*, args);  
 }  
  
}

@EnableDiscoveryClient  
@SpringBootApplication  
*public class* MovieStreamingApplication {  
  
 *public static void* main(String[] args) {  
 SpringApplication.*run*(MovieStreamingApplication.*class*, args);  
 }  
  
}

The @EnableDiscoveryClient will allow out movie catalogs service to connect to the service registry server via eureka.

**Overview**

1. Movie Catalogue Service: Manages movie information such as titles, descriptions, and file paths.

2. Movie Stream Service: Streams video content to clients, handling both full and partial content requests.

**Movie Catalogue Service**

The Movie Catalogue Service is responsible for storing and managing movie metadata. It interacts with a database to perform CRUD (Create, Read, Update, Delete) operations on movie records.

API Endpoints

1. Create a Movie

- Endpoint: `POST /movies`

- Description: Adds a new movie to the catalogue.

- Request Body:

{

"title": "Movie Title",

"description": "Movie Description",

"path": "/path/to/movie.mp4"

}

- Response:

- 201 Created: On successful creation.

- 400 Bad Request: If required fields are missing or invalid.

2. Retrieve All Movies

- Endpoint: `GET /movies`

- Description: Retrieves a list of all movies.

- Response:

[

{

"movieId": 1,

"title": "Movie Title",

"description": "Movie Description",

"path": "/path/to/movie.mp4"

}

]

3. Retrieve a Movie by ID\*\*

- Endpoint: `GET /movies/{id}`

- Description: Retrieves details of a specific movie by ID.

- Response:

{

"movieId": 1,

"title": "Movie Title",

"description": "Movie Description",

"path": "/path/to/movie.mp4"

}

4. Update a Movie

- Endpoint: `PUT /movies/{id}`

- Description: Updates details of an existing movie.

- Request Body:

{

"title": "Updated Title",

"description": "Updated Description",

"path": "/new/path/to/movie.mp4"

}

- Response:

- 200 OK: On successful update.

- 404 Not Found: If the movie does not exist.

5. Delete a Movie

- Endpoint: `DELETE /movies/{id}`

- Description: Deletes a movie from the catalogue.

- Response:

- 200 OK: On successful deletion.

- 404 Not Found: If the movie does not exist.

**Movie Stream Service**

The Movie Stream Service provides video streaming functionality. It supports full content delivery as well as partial content (range requests) for efficient video playback.

API Endpoint

1. Stream a Video\*\*

- Endpoint: `GET /streams/{videoPath}`

- Description: Streams video content based on the requested file path.

- Request Headers:

- `Range`: Specifies the byte range of the content to be streamed (e.g., `bytes=0-499`).

- Response:

- Full Content (HTTP 200):

HTTP/1.1 200 OK

Content-Type: video/mp4

Content-Length: {fileLength}

- Body: Full video content.

- Partial Content (HTTP 206)

HTTP/1.1 206 Partial Content

Content-Type: video/mp4

Content-Length: {contentLength}

Content-Range: bytes {start}-{end}/{fileLength}

`

- Body: Partial video content based on the range request.

- Example Request:

GET /streams/A\_Quiet\_Place.mp4 HTTP/1.1

Host: localhost:8091

Range: bytes=0-499

-Response Example:

HTTP/1.1 206 Partial Content

Content-Type: video/mp4

Content-Length: 500

Content-Range: bytes 0-499/5000

- Body: Partial video data.

**Additional Considerations**

1. Error Handling:

- 404 Not Found: If a movie or video file is not found.

- 400 Bad Request: For malformed requests or invalid parameters.

- 500 Internal Server Error: For unexpected server errors.

2. File Management\*\*:

- Ensure proper handling of file paths and permissions to avoid file not found errors.

- Use proper logging to track file access issues.

3. Testing:

- Test with various file sizes and formats to ensure the streaming service handles all cases correctly.

- Verify that range requests are properly handled, and the correct byte ranges are served.

**USING GIT FOR THIS MODULARIZED APPLICATION.**

Main Project Folder with git

Streamflatform – project name

service-registry

cd service-registry/

git add .

git commit -m “initial commit”

git remote add origin <url-of-service-registry-repository>

git push -u origin main

cd ..

git submodule add <url-of-service-registry-repository> service-registry

cd movie-catalogue/

git add .

git commit -m “initial commit”

git remote add origin <url-of- movie-catalogue -repository>

git push -u origin main

cd ..

git submodule add <url-of-movie-catalogue-repository> movie-catalogue

**API GATEWAY(Spring Cloud Routing)**

An API Gateway is a crucial component in a microservices architecture, acting as a single entry point for all client requests. It handles routing, authentication, and various cross-cutting concerns like logging and monitoring. In a Spring-based microservices setup, Spring Cloud Gateway is a popular choice for implementing an API Gateway.

### Spring Cloud Gateway

\*\*Spring Cloud Gateway\*\* is a lightweight, flexible, and high-performance API Gateway built on Spring WebFlux. It provides a way to route requests, apply filters, and aggregate responses.

### Key Features

1. \*\*Routing\*\*: Route requests to different microservices based on URL patterns.

2. \*\*Filters\*\*: Modify requests and responses. Supports pre and post filters.

3. \*\*Load Balancing\*\*: Distribute requests across multiple instances of a microservice.

4. \*\*Rate Limiting\*\*: Control the number of requests a client can make.

5. \*\*Security\*\*: Apply security policies, such as authentication and authorization.

6. \*\*Monitoring\*\*: Integrate with tools like Spring Boot Actuator for health checks and metrics.

### Setting Up Spring Cloud Gateway

Here’s a step-by-step guide to set up an API Gateway using Spring Cloud Gateway:

#### 1. Add Dependencies

Add the following dependencies to your `pom.xml` (for Maven) or `build.gradle` (for Gradle):

\*\*Maven:\*\*

```xml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-gateway</artifactId>

</dependency>

```

\*\*Gradle:\*\*

```groovy

implementation 'org.springframework.cloud:spring-cloud-starter-gateway'

```

Add the Spring Cloud dependency management to your `pom.xml` or `build.gradle` as well:

\*\*Maven:\*\*

```xml

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>2023.0.0</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

```

\*\*Gradle:\*\*

```groovy

dependencyManagement {

imports {

mavenBom "org.springframework.cloud:spring-cloud-dependencies:2023.0.0"

}

}

```

#### 2. Configure Routing

Create a configuration class or update your `application.yml` file to define routes.

\*\*Using `application.yml`:\*\*

```yaml

spring:

cloud:

gateway:

routes:

- id: movies-service

uri: http://localhost:8081

predicates:

- Path=/movies/\*\*

filters:

- RewritePath=/movies/(?<segment>.\*), /${segment}

- id: streams-service

uri: http://localhost:8082

predicates:

- Path=/streams/\*\*

filters:

- RewritePath=/streams/(?<segment>.\*), /${segment}

```

In this configuration:

- Requests to `/movies/\*\*` are routed to `http://localhost:8081`.

- Requests to `/streams/\*\*` are routed to `http://localhost:8082`.

\*\*Using Java Configuration:\*\*

```java

import org.springframework.cloud.gateway.route.RouteLocator;

import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class GatewayConfig {

@Bean

public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {

return builder.routes()

.route("movies-service", r -> r.path("/movies/\*\*")

.uri("http://localhost:8081")

.filters(f -> f.rewritePath("/movies/(?<segment>.\*)", "/${segment}")))

.route("streams-service", r -> r.path("/streams/\*\*")

.uri("http://localhost:8082")

.filters(f -> f.rewritePath("/streams/(?<segment>.\*)", "/${segment}")))

.build();

}

}

```

#### 3. Configure Security (Optional)

If you need to secure your gateway, you can configure Spring Security:

```yaml

spring:

security:

oauth2:

client:

registration:

google:

client-id: YOUR\_CLIENT\_ID

client-secret: YOUR\_CLIENT\_SECRET

authorization-grant-type: authorization\_code

redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"

scope: profile, email

provider:

google:

authorization-uri: https://accounts.google.com/o/oauth2/auth

token-uri: https://oauth2.googleapis.com/token

user-info-uri: https://www.googleapis.com/oauth2/v3/userinfo

user-name-attribute: sub

```

#### 4. Testing the API Gateway

1. \*\*Run Your Services\*\*: Start the services (e.g., Movie Service and Stream Service) on their respective ports.

2. \*\*Start the Gateway\*\*: Run the Spring Cloud Gateway application.

3. \*\*Test Routes\*\*: Use tools like Postman to send requests to `http://localhost:8080/movies` and `http://localhost:8080/streams` to verify routing.

### Conclusion

Spring Cloud Gateway simplifies the implementation of an API Gateway in a microservices architecture. It provides powerful routing capabilities, supports various filters, and integrates well with other Spring components.

This setup can be extended with more advanced features such as custom filters, load balancing, and integration with monitoring tools. For detailed information, refer to the [official Spring Cloud Gateway documentation](https://spring.io/projects/spring-cloud-gateway).

---

Feel free to adjust the details to fit your project’s requirements and configuration.

**Service to Service Communication**

In a microservices architecture, \*\*Service-to-Service Communication\*\* is essential for microservices to interact with each other. There are several methods to enable this communication, each with its own use cases, benefits, and trade-offs. Here’s an overview of the most common approaches and how to implement them using Spring.

### 1. \*\*RESTful Communication\*\*

\*\*REST (Representational State Transfer)\*\* is one of the most common methods for service-to-service communication. It uses HTTP requests to perform CRUD operations and is widely supported.

#### Implementation with Spring Boot:

\*\*Service A\*\*: Making HTTP requests to another service.

```java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.GetMapping;

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

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

import org.springframework.web.client.RestTemplate;

@RestController

@RequestMapping("/serviceA")

public class ServiceAController {

@Autowired

private RestTemplate restTemplate;

@GetMapping("/callServiceB")

public String callServiceB() {

String url = "http://localhost:8082/serviceB/endpoint";

return restTemplate.getForObject(url, String.class);

}

}

```

\*\*Service Configuration\*\*:

```java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.client.RestTemplate;

@Configuration

public class AppConfig {

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

```

### 2. \*\*FeinClient\*\*

\*\*Feign\*\* is a declarative HTTP client that simplifies service-to-service communication. It abstracts the complexity of creating HTTP requests and integrates well with Spring Cloud.

#### Implementation with Spring Cloud OpenFeign:

\*\*Service A\*\*: Define a Feign client interface.

```java

import org.springframework.cloud.openfeign.FeignClient;

import org.springframework.web.bind.annotation.GetMapping;

@FeignClient(name = "serviceB", url = "http://localhost:8082")

public interface ServiceBClient {

@GetMapping("/endpoint")

String getEndpoint();

}

```

\*\*Service A\*\*: Use the Feign client.

```java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.GetMapping;

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

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

@RestController

@RequestMapping("/serviceA")

public class ServiceAController {

@Autowired

private ServiceBClient serviceBClient;

@GetMapping("/callServiceB")

public String callServiceB() {

return serviceBClient.getEndpoint();

}

}

```

\*\*Dependencies (Maven)\*\*:

```xml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

```

\*\*Enable Feign Clients\*\*:

```java

import org.springframework.cloud.openfeign.EnableFeignClients;

import org.springframework.context.annotation.Configuration;

@Configuration

@EnableFeignClients

public class AppConfig {

}

```

### 3. \*\*gRPC\*\*

\*\*gRPC\*\* is a high-performance RPC framework developed by Google that uses HTTP/2 for transport and Protocol Buffers for serialization.

#### Implementation with Spring Boot:

\*\*Service A\*\*: Define a gRPC client.

```java

import io.grpc.stub.StreamObserver;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class ServiceAClient {

@Autowired

private ServiceBGrpc.ServiceBBlockingStub serviceBBlockingStub;

public String callServiceB() {

Request request = Request.newBuilder().build();

Response response = serviceBBlockingStub.callService(request);

return response.getMessage();

}

}

```

\*\*Service B\*\*: Define the gRPC service implementation.

```java

import io.grpc.stub.StreamObserver;

import org.springframework.stereotype.Service;

@Service

public class ServiceBImpl extends ServiceBGrpc.ServiceBImplBase {

@Override

public void callService(Request request, StreamObserver<Response> responseObserver) {

Response response = Response.newBuilder().setMessage("Hello from Service B").build();

responseObserver.onNext(response);

responseObserver.onCompleted();

}

}

```

\*\*Dependencies (Maven)\*\*:

```xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

<dependency>

<groupId>io.grpc</groupId>

<artifactId>grpc-spring-boot-starter</artifactId>

</dependency>

```

### 4. \*\*Message Brokers\*\*

\*\*Message Brokers\*\* (like RabbitMQ or Kafka) are used for asynchronous communication. They allow microservices to communicate by publishing and subscribing to events.

#### Implementation with Spring Boot and RabbitMQ:

\*\*Service A\*\*: Publish a message.

```java

import org.springframework.amqp.rabbit.core.RabbitTemplate;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.GetMapping;

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

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

@RestController

@RequestMapping("/serviceA")

public class ServiceAController {

@Autowired

private RabbitTemplate rabbitTemplate;

@GetMapping("/sendMessage")

public String sendMessage() {

rabbitTemplate.convertAndSend("exchange", "routing.key", "Hello, Service B");

return "Message sent";

}

}

```

\*\*Service B\*\*: Consume the message.

```java

import org.springframework.amqp.rabbit.annotation.RabbitListener;

import org.springframework.stereotype.Service;

@Service

public class ServiceBService {

@RabbitListener(queues = "queue")

public void receiveMessage(String message) {

System.out.println("Received: " + message);

}

}

```

\*\*Dependencies (Maven)\*\*:

```xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-amqp</artifactId>

</dependency>

```

### Conclusion

Service-to-Service communication can be implemented in various ways depending on your use case, whether you need synchronous or asynchronous communication, or you need to ensure high performance. Spring provides multiple tools to facilitate this, from simple REST calls with `RestTemplate` to advanced solutions like gRPC and message brokers.

Each approach has its strengths:

- \*\*RESTful Communication\*\*: Simple and widely used.

- \*\*FeignClient\*\*: Simplifies HTTP communication with a declarative approach.

- \*\*gRPC\*\*: High-performance RPC communication.

- \*\*Message Brokers\*\*: Asynchronous and decoupled communication.

Choose the method that best fits your architecture and requirements.

**third party api that uses a key for it request**

To communicate with a third-party API that requires an API key for authentication, you typically need to include the API key in the request headers or as a query parameter. Here’s a step-by-step guide on how to do this in Spring Boot using \*\*`RestTemplate`\*\* or \*\*Feign Client\*\*.

### Using `RestTemplate`

#### 1. \*\*Configure the RestTemplate Bean\*\*

First, define a `RestTemplate` bean in your Spring configuration class:

```java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.client.RestTemplate;

@Configuration

public class AppConfig {

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

```

#### 2. \*\*Create a Service to Call the Third-Party API\*\*

You will need to create a service that uses the `RestTemplate` to send a request to the third-party API.

```java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpEntity;

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpMethod;

import org.springframework.http.ResponseEntity;

import org.springframework.stereotype.Service;

import org.springframework.web.client.RestTemplate;

@Service

public class ThirdPartyApiService {

@Autowired

private RestTemplate restTemplate;

private final String API\_KEY = "your-api-key-here";

private final String API\_URL = "https://api.example.com/data";

public String getDataFromApi() {

// Set up headers

HttpHeaders headers = new HttpHeaders();

headers.set("Authorization", "Bearer " + API\_KEY); // Some APIs use 'Authorization: Bearer'

// Or headers.set("API-Key", API\_KEY); for other cases

HttpEntity<String> entity = new HttpEntity<>(headers);

// Make the request

ResponseEntity<String> response = restTemplate.exchange(

API\_URL, HttpMethod.GET, entity, String.class);

// Return the response body

return response.getBody();

}

}

```

In this example, the API key is passed in the `Authorization` header. Some APIs may require the key as a query parameter instead, like this:

```java

String API\_URL = "https://api.example.com/data?api\_key=" + API\_KEY;

```

### Using Feign Client

Spring Cloud OpenFeign simplifies calling external REST APIs by allowing you to define an interface for the external service.

#### 1. \*\*Add the Feign Dependency\*\*

Make sure you have the Feign dependency in your `pom.xml`:

```xml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

```

#### 2. \*\*Create the Feign Client Interface\*\*

Define an interface that represents the third-party API.

```java

import org.springframework.cloud.openfeign.FeignClient;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestHeader;

@FeignClient(name = "thirdPartyApi", url = "https://api.example.com")

public interface ThirdPartyApiClient {

@GetMapping("/data")

String getData(@RequestHeader("Authorization") String apiKey);

}

```

#### 3. \*\*Use the Feign Client in a Service\*\*

Now, you can use the Feign client in a service to call the API.

```java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class ThirdPartyApiService {

@Autowired

private ThirdPartyApiClient thirdPartyApiClient;

private final String API\_KEY = "Bearer your-api-key-here";

public String getDataFromApi() {

return thirdPartyApiClient.getData(API\_KEY);

}

}

```

In this case, the API key is passed in the `Authorization` header. You can adjust this based on how the API you're interacting with expects the key to be included (headers, query parameters, etc.).

### Summary

1. \*\*RestTemplate\*\* is a lower-level solution where you manage request headers manually.

2. \*\*Feign Client\*\* is a declarative HTTP client that abstracts the HTTP request creation and simplifies communication with third-party services.

Both methods work well for calling APIs with API keys, and you should choose based on your preference for simplicity (Feign) or flexibility (RestTemplate).

Help videos -> <https://www.youtube.com/watch?v=MrSECdSIaOg>

<https://www.youtube.com/watch?v=c1MzBXjedFg>

<https://www.themoviedb.org/login?to=read_me&redirect=%2Freference%2Fmovie-top-rated-list>

http://localhost:8761