# Contents

# Books and Albums Search Service

# Design and Usage Introduction

## 1. Requirement Introduction

The service will accept a request with text parameters on input based on the Java framework or libraries. It will return a maximum of 5 books and a maximum of 5 albums that are related to the input term. The response elements will only contain the title, authors(/artists), and information whether it's a book or an album. For albums, it uses the iTunes API. For books, it uses Google Books API. The results are sorted by title alphabetically. The service needs to be production-ready from a resilience, stability, and performance point of view. The stability of the downstream service may not be affected by the stability of the upstream services. Results originating from one upstream service (and its stability /performance) may not affect the results originating from the other upstream service.

The service needs to:

- respond within a minute;
- be self-documenting;
- expose metrics on response times for upstream services;
- expose health check;
- limit results on upstream services that must be configurable per environment and preconfigured to 5;
- think about resilience;
- think about concurrency.

## 2. Service Introduction

### 2.1 Frameworks and Components

Table 1 The Frameworks and Components of the Service

| | |
|---|---|
| Frameworks | SpringBoot, SpringCloud |
| Service Registration and Discovery | Eureka |
| Circuit Breaker | Hystrix |
| Logging | Spring AOP, log4j |
| Health Check | Actuator |
| Remote Call | Feign |

### 2.2 Service Architecture

- A service provider;
- Two Eureka servers;
- A consumer.

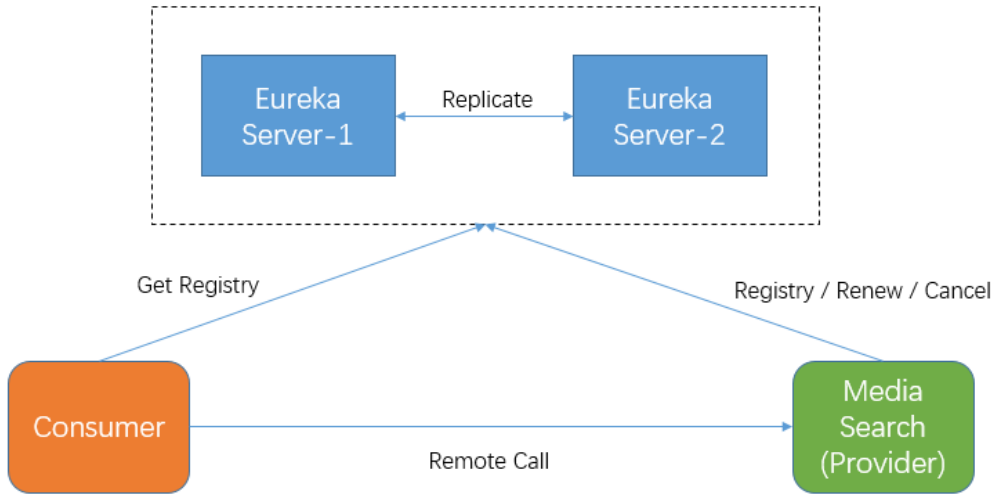


Figure 1. The Service Architecture

## 3. Technology / Mechanism Choice

The main function of the service is to query. Based on the design requirements, the service needs to have high availability. So it mainly implements the AP in the CAP principle.

### 3.1 Service registration and discovery

The registry supported by SpringCloud has Eureka, Zookeeper, Consul, and Nacos. As the service was implemented by Java, the main consideration is Eureka, Zookeeper, and Nacos. The comparisons of different registration and discovery components are in Table 2.

Table 2. The comparisons among Eureka, Zookeeper, Nacos

| Comparison | Eureka | Zookeeper | Nacos |
|---|---|---|---|
| CAP | AP | CP | AP / CP |
| Load Balancing | Ribbon | -- | Metadata/Weights/Selector |
| Health Check | Client Beat | Keep Alive | TCP/HTTP/MYSQL/Client Beat |
| Avalanche Protection | Yes | No | Yes |

Considering high availability and partition tolerance, Eureka and Nacos are both suitable to use. Furthermore, Eureka is more mature and stable than Nacos, as Nacos has been open-sourced since 2018.

## 3.2 Respond Time

The service needs to respond within a minute. It uses Java thread pool to reduce resource consumption, improve response speed, and improve thread manageability. Also, the thread pool is one of the suitable technology stacks to solve concurrency. Hystrix performs demotion logic when a request fails, is rejected, times out, or is blown, which supplies resilience to the service.

The comparisons of different circuit breakers are as Table 3.

Table 3. The comparisons among Hystrix, Sentinel, Resilience4j

| Comparison | Hystrix | Sentinel | Resilience4j |
|---|---|---|---|
| Quarantine Policy | Semaphore isolation / Thread pool isolation | Semaphore isolation | Semaphore isolation |
| Circuit Breaker Downgrade Strategy | Based on the failure rate | Based on response time or failure rate | Based on response time or failure rate |
| Rate Limiting | Limited support | Based on QPS | Rate Limiter |
| Annotation-based support | Yes | Yes | Yes |
| Console | Simple monitoring and viewing | Out-of-the-box console | -- |

Resilience4j is a relatively lightweight circuit breaker, but it lacks production-level supporting facilities, such as a console that provides rule management and real-time monitoring capabilities. Sentinel is not as mature as Hystrix. So Hystrix is a better choice to choose.

## 3.3 Self-documenting

The service uses Spring AOP and log4j to log, including response time, request URL, parameters, etc. The log is mainly printed on the console, and the log folder is also generated locally.

Using the AOP method to record logs can make the code independent of other business logic codes. It is also conducive to later maintenance and minimizes performance impact. The configuration of log4j is configured in log4j.propertities,

which controls the printed log level, local folder location, the max file size of log doc, etc.

### 3.4 Exposes metrics on response times for upstream services

The response times for searching books and albums are recorded in the log, using Log4j.

### 3.5 Exposes health check

The service uses the Actuator health endpoint to detect the client's state.

By default, Eureka uses the client's heartbeat to determine whether a service is in the "UP" state. As long as the client registers the service successfully, the Eureka server will announce that the service is "UP". Therefore, if the entire service does not work, it is down. But for example, if a service provider can't connect to the database, the service instance is unavailable, but the server doesn't think so, because it has no way of knowing this service is problematic. So, Actuator is introduced here, which uses its /health endpoint for health detection.

It is convenient to check the health state by entering the link in the browser.
http://localhost:8080/actuator/health

The result of health check:


```
{
  "status": "UP",
  "components": {
    "discoveryComposite": {
      "status": "UP",
      "components": {
        "discoveryClient": {
          "status": "UP",
          "details": {
            "services": [
              "eureka-server",
              "mediasearch",
              "eureka-consumer"
            ]
          }
        },
        "eureka": {
          "description": "Remote status from Eureka server",
```

```
          "status": "UP",
          "details": {
            "applications": {
              "MEDIASEARCH": 1,
              "EUREKA-CONSUMER": 1,
              "EUREKA-SERVER": 2
            }
          }
        }
      }
    },
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 59501027328,
        "free": 736624640,
        "threshold": 10485760,
        "exists": true
      }
    },
    "hystrix": {
      "status": "UP"
    },
    "mediaSearch": {
      "status": "UP"
    },
    "ping": {
      "status": "UP"
    },
    "refreshScope": {
      "status": "UP"
    }
  }
}
```


## 3.6 Limit of results that is configurable per environment and preconfigured to 5

The limit of results was configured in the application.properties, which is preconfigured to 5. If the limitations are greater than 5, the service will assign the number to 5 in MediaSearchImpl.

# 4. Interface Description

- The interface returns a maximum of 5 books and a maximum of 5 albums by accepting the title of the book and album;
- If the number of books or albums is more than 5, it will be sorted by published date or released date in descending order and limited to 5;
- The final results are sorted by title alphabetically.

## 4.1 MediaSearch (Provider) Interface

### 4.1.1 HTTP Request Method

GET

### 4.1.2 Request Parameters

Table 4. The Request Parameters of MediaSearch Interface

| Name | Type | Required | Note |
|---|---|---|---|
| bookTitle | String | Y | Book Title |
| albumTitle | String | Y | Album Title |

### 4.1.3 Request Example

http://localhost:8080/media/getMediaByTitle?bookTitle=java&albumTitle=Heart On My Sleeve

### 4.1.4 Response Parameters

Table 5. The Response Parameters of MediaSearch Interface

| Name | Type | Note |
|---|---|---|
| title | String | Book or album title |
| author | String | Author or artist |
| publishedDate | String | Published or released date |
| type | String | Book / Album |

#### 4.1.5 Response Example


```
{
  "code": 200,
  "data": [
    {
      "title": "De tolk van Java",
      "author": "Alfred Birney",
      "publishedDate": "2016-03-14",
      "type": "Book"
    },
    {
      "title": "Geschiedenis van Java",
      "author": "Willemine Fruin-Mees",
      "publishedDate": "1919",
      "type": "Book"
    },
    {
      "title": "Heart On My Sleeve",
      "author": "Ella Mai",
      "publishedDate": "2022-05-06",
      "type": "Album"
    },
    {
      "title": "Heart On My Sleeve",
      "author": "Ella Mai",
      "publishedDate": "2022-05-06",
      "type": "Album"
    },
    {
      "title": "Heart On My Sleeve - Single",
      "author": "Traavonn",
      "publishedDate": "2022-04-30",
      "type": "Album"
    },
    {
      "title": "Heart On My Sleeve - Single",
      "author": "Sicco",
      "publishedDate": "2022-01-03",
      "type": "Album"
    },
    {
      "title": "Heart on My Sleeve",
      "author": "Ryda Ramone",
```

```
        "publishedDate": "2022-03-11",
        "type": "Album"
      },
      {
        "title": "Nacht over Java",
        "author": "Johan Fabricius",
        "publishedDate": "2013-03-14",
        "type": "Book"
      },
      {
        "title": "TCP/IP Sockets in Java",
        "author": "Kenneth L. Calvert,Michael J. Donahoo",
        "publishedDate": "2011-08-29",
        "type": "Book"
      },
      {
        "title": "West-Java-Koffij-Cultuur-Maatschappij",
        "author": "Gualtherus Suermondt,H. Hope Loudon",
        "publishedDate": "1865",
        "type": "Book"
      }
    ],
    "msg": null
}
```


## 4.2 Consumer Interface

### 4.2.1 HTTP Request Method

GET

### 4.2.2 Request Parameters

The same as MediaSearch Interface Chapter 4.1.2.

### 4.2.3 Request Example

http://localhost:8084/search/mediaResult?bookTitle=java&albumTitle=Heart On My Sleeve

#### 4.2.4 Response Parameters

The same as MediaSearch Interface Chapter 4.1.4.

#### 4.2.5 Response Example

The same as MediaSearch Interface Chapter 4.1.5.

# 5. Operation Instructions

## 5.1 Local Environment

The consumer example is listed in the package “eureka-consumer”.

#### 5.1.1 First Step

The first step is to create a SpringBoot project and update it into a SpringCloud project by adding a dependency in the pom.xml.

```
<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>
```

At the same time, it also needs to specify the version of SpringCloud that is matched with SpringBoot. The website lists the compatibility of the two as Figure 2.

**Adding Spring Cloud To An Existing Spring Boot Application**

If you an existing Spring Boot app you want to add Spring Cloud to that app, the first step is to determine the version of Spring Cloud you should use. The version you use in your app will depend on the version of Spring Boot you are using.

The table below outlines which version of Spring Cloud maps to which version of Spring Boot.

**Table 1. Release train Spring Boot compatibility**

| Release Train | Boot Version |
|---|---|
| 2021.0.x aka Jubilee | 2.6.x |
| 2020.0.x aka Ilford | 2.4.x, 2.5.x (Starting with 2020.0.3) |
| Hoxton | 2.2.x, 2.3.x (Starting with SR5) |
| Greenwich | 2.1.x |
| Finchley | 2.0.x |
| Edgware | 1.5.x |
| Dalston | 1.5.x |

Figure 2. The Compatibility of SpringBoot and SpringCloud

As the version of SpringBoot is 2.6.7, it is suitable to add the SpringCloud version of 2021.0.1.

```
1. <properties>
2.     <java.version>1.8</java.version>
3.     <spring-cloud.version>2021.0.1</spring-cloud.version>
4. </properties>
```

### 5.1.2 Second Step

The second step is to add dependency of Eureka Client to pom.xml.

```
1. <dependency>
2.     <groupId>org.springframework.cloud</groupId>
3.     <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
4. </dependency>
```

### 5.1.3 Third Step

The third step is to register to the Eureka Server. Add the following code to the configuration file named application.properties.

```
1. eureka.client.service-
   url.defaultZone=http://localhost:8082/eureka/,http://localhost:8083/eureka
   /
```

#### 5.1.4 Fourth Step

The fourth step is to add Feign dependency in the pom.xml. The dependency is as follows.

```
1. <dependency>
2.     <groupId>org.springframework.cloud</groupId>
3.     <artifactId>spring-cloud-starter-openfeign</artifactId>
4. </dependency>
```

#### 5.1.5 Fifth Step

The fifth step is to write an interface using Restful and use Feign to make remote calls.

### 5.2 Production Environment

The difference between the local environment and the production environment is the configuration doc “application.properties”. The port and hostname could be replaced from “localhost:8080” to the server of the production environment as follows.

```
1. server.port=8081
2. eureka.instance.hostname=192.168.178.17
3. eureka.client.registerWithEureka=true
4. eureka.client.fetchRegistry=true
5. eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${se
   rver.port}/eureka/
```