

Java Microservices – S1

Manpreet Singh Bindra
Senior Manager





Do's and Don't

- Login to GTW session on Time
- Login with your Mphasis Email ID only
- Use the question window for asking any queries



Welcome

1. Skill - Proficiency Introduction
2. About Me - Introduction
3. Walkthrough the Skill on TalentNext
4. About Peer Learning



About Peer Learning Platform

Hi folks! I am preparing for the AWS Certified Developer – Associate Certification. It has been a wonderful experience, so far.

How I wish there was a forum where I could post questions, follow discussions, seek practical insights.



You are in luck, my friend! Say hello to "Peer Learning" feature on Talent Next! It's a collection of discussion forums on specific skills.

You can follow ongoing discussions, contribute to it by sharing your learnings and create new discussion threads to seek clarification. All these are bound to add to your overall learning experience.



Yes indeed! Peer Learning comes handy to get quick responses to your queries as a wide pool of subject matter experts interact actively





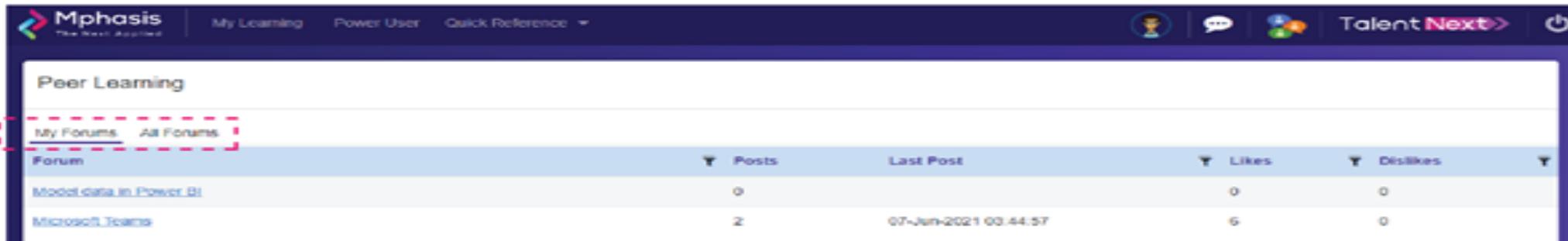
Where to find Peer Learning Platform

A quick way to get started:

Step 1: Click on the Peer Learning Icon  at the top right of the menu bar

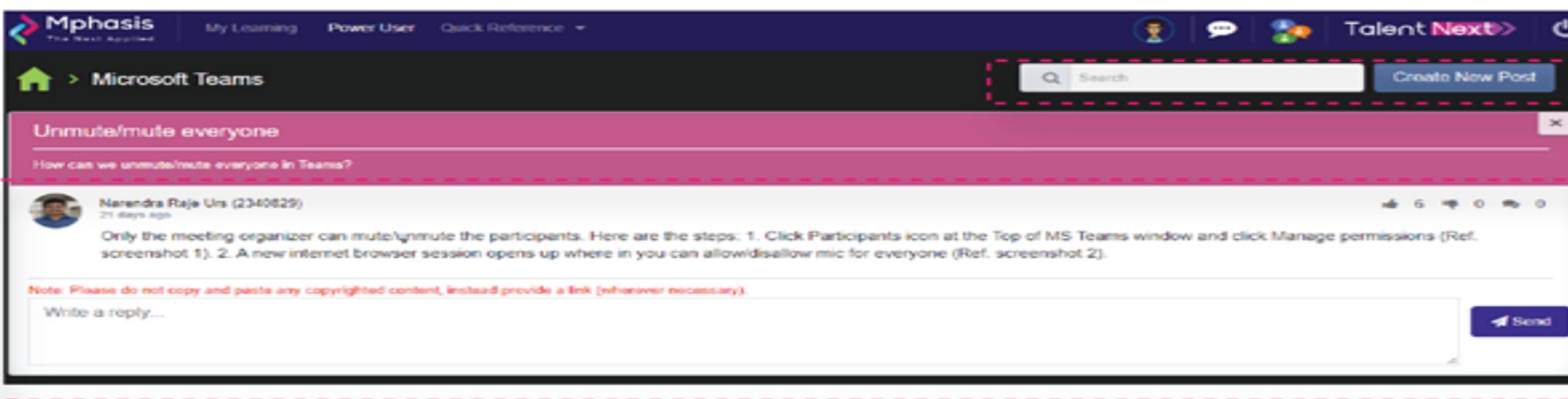


Step 2: Select the forum you want to post in from the list



Forum	Posts	Last Post	Likes	Dislikes
Model data in Power BI	0		0	0
Microsoft Teams	2	07-Jun-2021 03:44:57	6	0

Step 3: Use the “Create New Post” to create a new discussion thread



Unmute/mute everyone

How can we unmute/mute everyone in Teams?

Nanendra Raja Urs (2340629)
21 days ago

Only the meeting organizer can mute/unmute the participants. Here are the steps: 1. Click Participants icon at the Top of MS Teams window and click Manage permissions (Ref. screenshot 1). 2. A new internet browser session opens up where in you can allow/disallow mic for everyone (Ref. screenshot 2).

Note: Please do not copy and paste any copyrighted content, instead provide a link (whenever necessary).

Write a reply...

Send



Overall Agenda

- Understanding of SOA or similar design principles
- Understanding of MVC - Spring or any Framework
- Understanding of Micro Services architecture
- Compare Micro Services with other architectural styles (SOA/Monolithic applications)
- Understand 12 factor principles for micro service architecture
- Ability to develop a stand-alone micro service
- Understand inter-service communication
- Hands on experience with API (Spring boot, Data, REST web services)
- Deploy services that use Netflix Eureka, Hystrix, and Ribbon to create resilient and scalable services



Overall Agenda

- Distributed Tracing with Sleuth and Zipkin
- Understanding the replacement available in new Spring Cloud version
- Implementing Circuit Breaker using Resilience4J
- Client-Side Load-Balancing with Spring Cloud Load Balancer
- Monitoring Spring Boot with Micrometer and Prometheus
- Implementing Message-Driven Microservices using Spring Cloud Stream & RabbitMQ
- Publishing Custom Events with Spring Cloud Bus
- Understanding Stream Processing using Spring Cloud Data Flow
- Autoscaling microservices



Day - 1

- What Is Monolithic Architecture?
- Concerns With the Monolith
- The Microservice architecture
- Characteristics of a Microservice Architecture
- Principles of microservices
- Business demand as a catalyst for Microservices
- Technology as a catalyst for the microservices evolution
- Building microservices with spring boot
- The microservices capability model



Day - 1

Introduction to Microservices



What are Microservices?

- Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are
 - Highly maintainable and testable
 - Loosely coupled
 - Independently deployable services
 - Organized around business capabilities
 - Owned by a small team

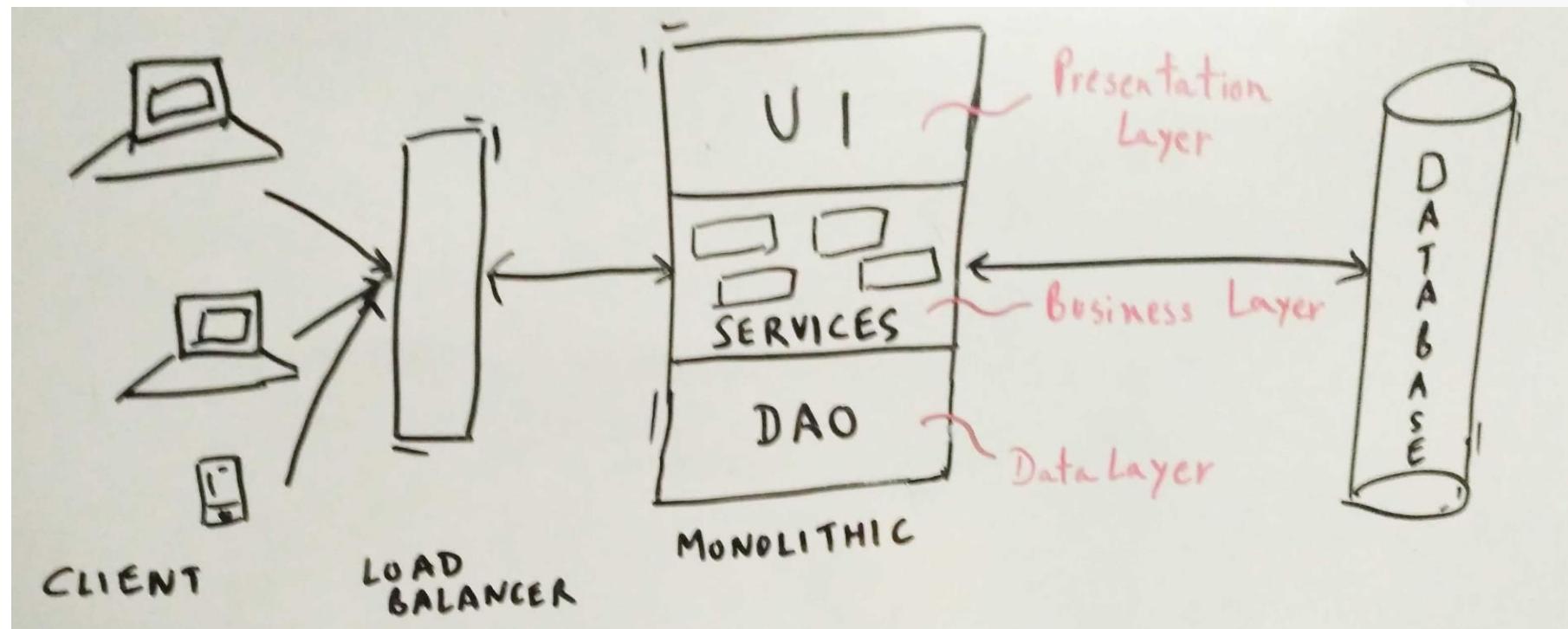


Need for Microservices

- To understand the need for microservices, we need to understand problems with our typical 3-tier monolithic architecture.

What Is Monolithic Architecture?

- Monolithic means composed all in one piece. A monolithic application is one which is self-contained. All components of the application must be present in order for the code to work.





Concerns With the Monolith

- The large monolithic code base
- Overloaded IDE
- Overloaded web container
- Continuous deployment is difficult
- Scaling the application can be difficult
- Obstacle to scaling development
- Requires a long-term commitment to a technology stack



The Microservice architecture

- The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack.
- While there is no precise definition of this architectural style, there are certain common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.



Characteristics of a Microservice Architecture

- Componentization via Services
- Organized around Business Capabilities
- Products not Projects (“you build, you run it”)
- Smart endpoints and dumb pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for failure



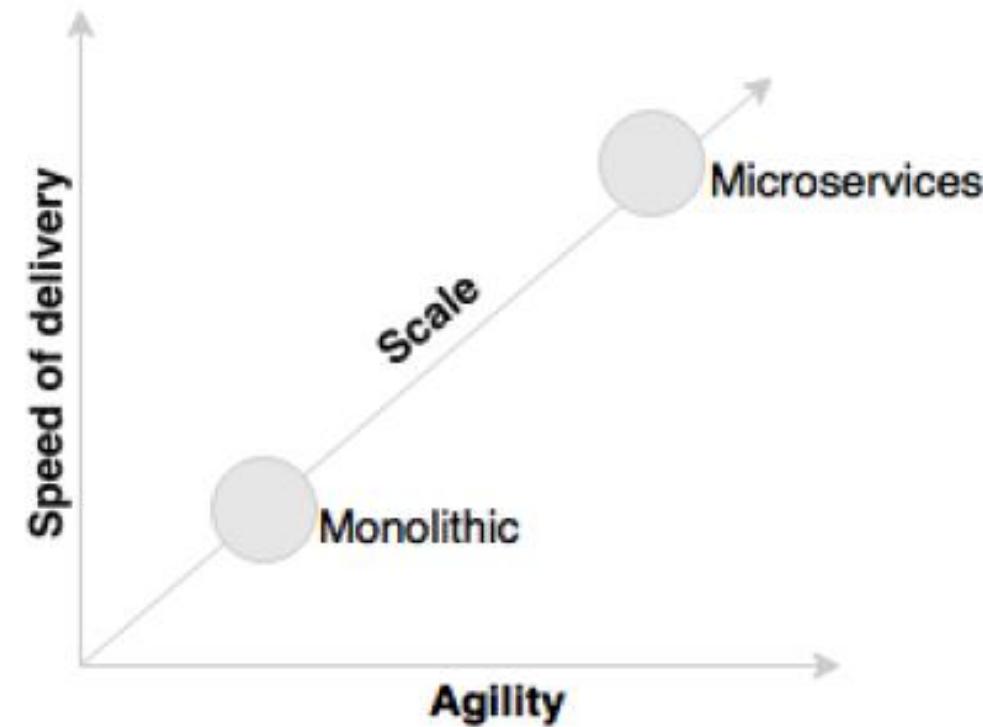
Business demand as a catalyst for Microservices

- In this era of digital transformation, enterprises increasingly adopt technologies as one of the key enablers for radically increasing their revenue and customer base.
- Enterprises primarily use social media, mobile, cloud, big data, and Internet of Things as vehicles to achieve the disruptive innovations. Using these technologies, enterprises find new ways to quickly penetrate the market, which severely pose challenges to the traditional IT delivery mechanisms.



Business demand as a catalyst for Microservices

- The following graph shows the state of traditional development and microservices against the new enterprise challenges such as agility, speed of delivery, and scale.





Technology as a catalyst for the microservices evolution

- Emerging technologies have also made us rethink the way we build software systems.
- The emergence of HTML 5 and CSS3 and the advancement of mobile applications repositioned user interfaces. Client-side JavaScript frameworks such as Angular, Ember, React, Backbone, and so on are immensely popular due to their client-side rendering and responsive designs.
- With cloud adoptions steamed into the mainstream, **Platform as a Services (PaaS) providers** such as Pivotal CF, AWS, Salesforce.com, IBMs Bluemix, RedHat OpenShift, and so on made us rethink the way we build middleware components.
- The container revolution created by **Docker** radically influenced the infrastructure space. These days, an infrastructure is treated as a commodity service.



Technology as a catalyst for the microservices evolution

- The integration landscape has also changed with **Integration Platform as a Service (iPaaS)**, which is emerging. Platforms such as Dell Boomi, Informatica, MuleSoft, and so on are examples of iPaaS.
- NoSQLs have revolutionized the databases space. A few years ago, we had only a few popular databases, all based on relational data modeling principles. We have a long list of databases today: Hadoop, Cassandra, CouchDB, and Neo 4j to name a few.



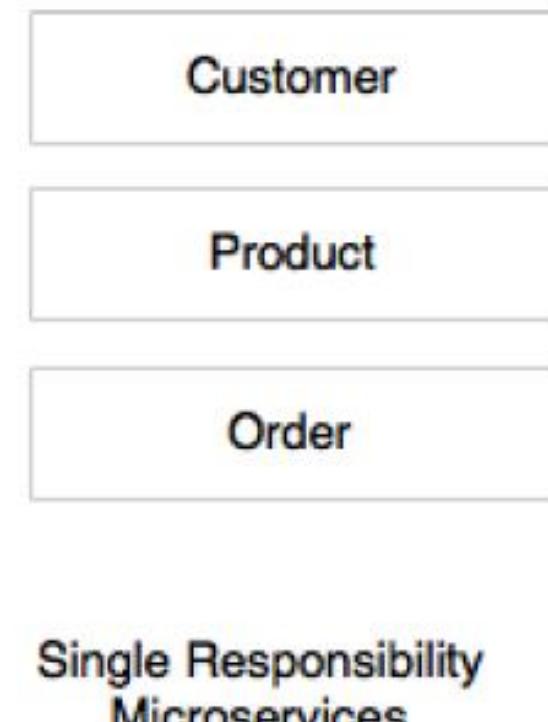
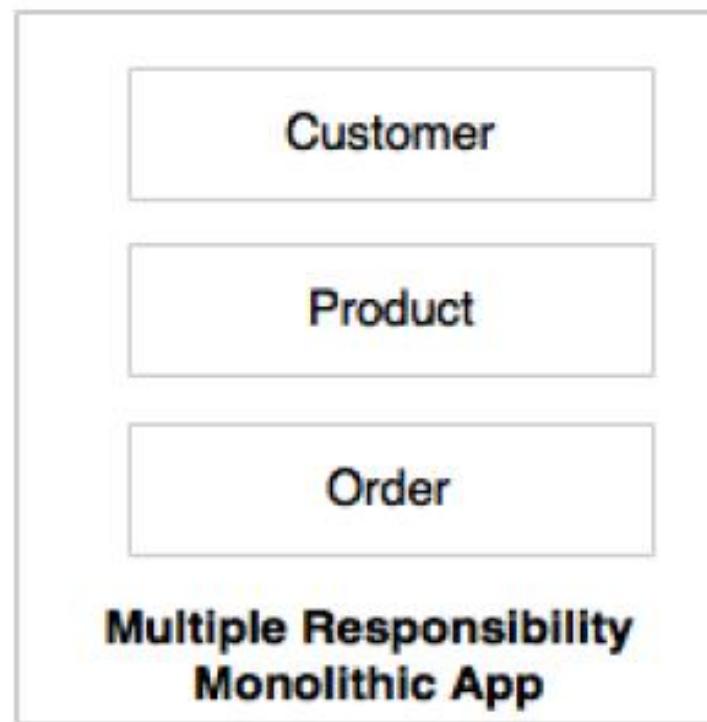
Principles of microservices

- These principles are a "must have" when designing and developing microservices.
 - Single responsibility per service
 - Microservices are autonomous



Single responsibility per service

- The single responsibility principle is one of the principles defined as part of the SOLID design pattern.
- It states that a unit should only have one responsibility.





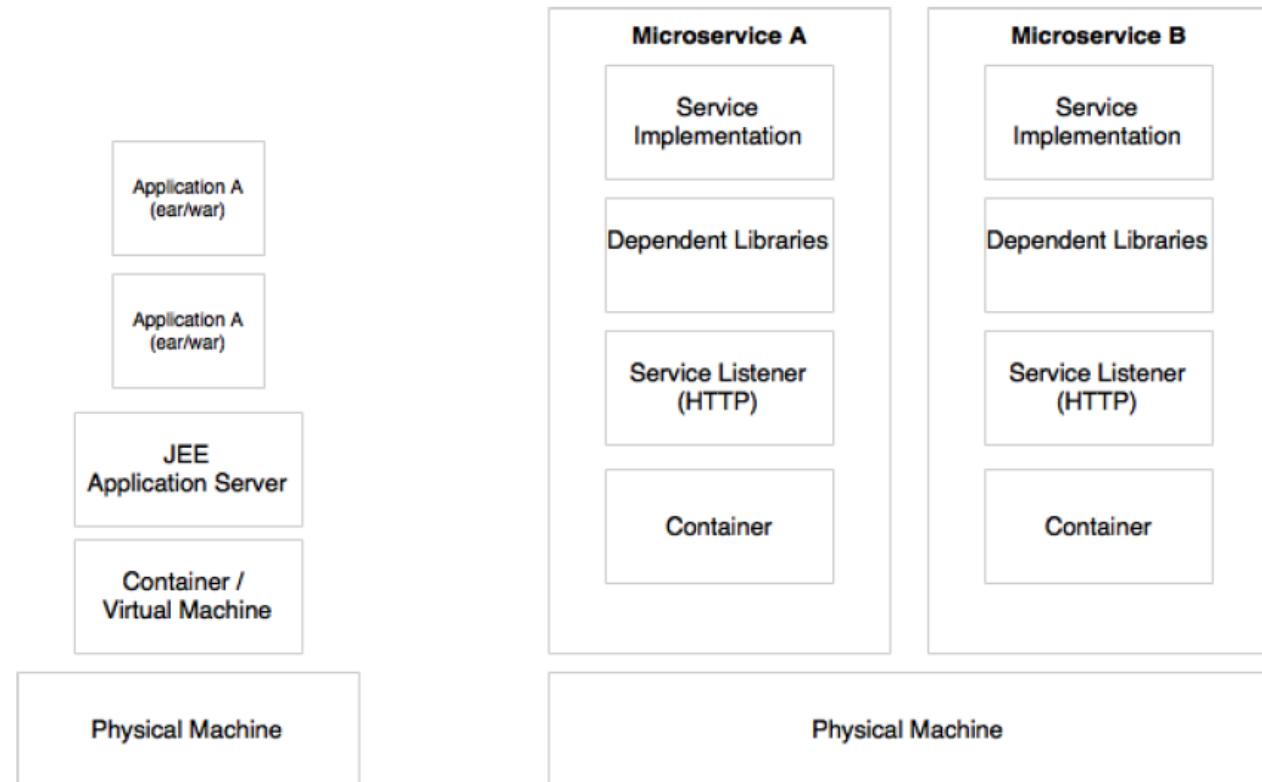
Microservices are autonomous

- Microservices are self-contained, independently deployable, and autonomous services that take full responsibility of a business capability and its execution.
- They bundle all dependencies, including library dependencies, and execution environments such as web servers and containers or virtual machines that abstract physical resources.



Major differences between Microservices and SOA

- One of the major differences between microservices and SOA is in their level of autonomy. While most SOA implementations provide service-level abstraction, microservices go further and abstract the realization and execution environment.





Microservices are lightweight

- Well-designed microservices are aligned to a **single business capability**, so they perform only one function. As a result, one of the common characteristics we see in most of the implementations are microservices with smaller footprints.
- When selecting **supporting technologies**, such as web containers, we will have to ensure that they are also lightweight so that the overall footprint remains manageable. For example, Jetty or Tomcat are better choices as application containers for microservices compared to more complex traditional application servers such as WebLogic or WebSphere.
- **Container technologies such as Docker** also help us keep the infrastructure footprint as minimal as possible compared to hypervisors such as VMWare or Hyper-V.

Microservices are lightweight

Physical Machine

Virtual Machine

Application Server

A B C

All in one App EAR

Physical Machine

Docker Container

A libs + Listener

Microservice A

Docker Container

B libs + Listener

Microservice B

Docker Container

C libs + Listener

Microservice C

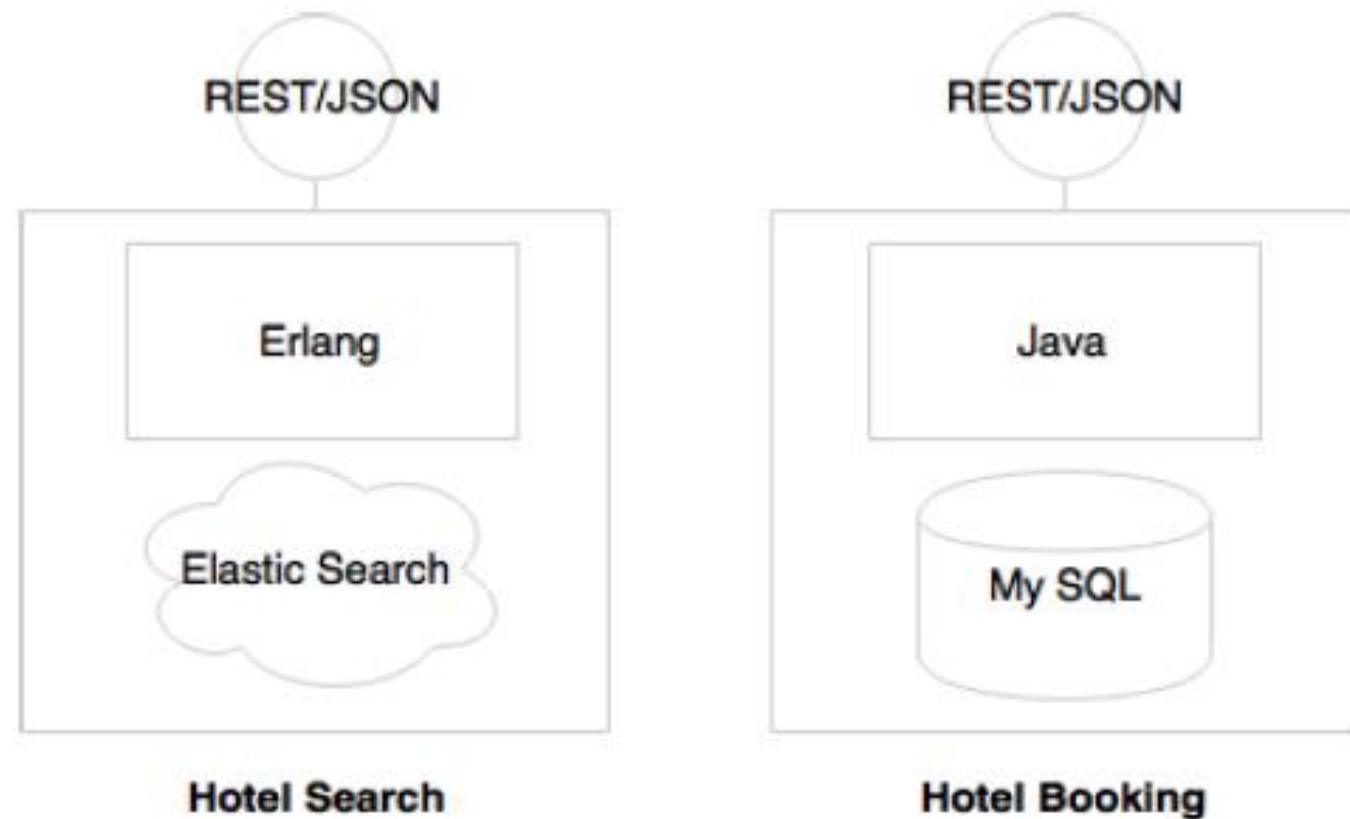
Traditional Deployment

Microservices Deployment

- As microservices are autonomous and abstract everything behind service APIs, it is possible to have different architectures for different microservices.
- A few common characteristics that we see in microservices implementations are:
 - Different services use **different versions** of the same technologies. One microservice may be written on Java 1.7, and another one could be on Java 1.8.
 - **Different languages** are used to develop different microservices, such as one microservice is developed in Java and another one in Scala.
 - **Different architectures** are used, such as one microservice using the Redis cache to serve data, while another microservice could use MySQL as a persistent data store.



Microservices with polyglot architecture





Day - 1

Building Microservices with Spring Boot



Building microservices with spring boot

- There is no "one size fits all" approach when implementing microservices.
- Examining the simple microservices version of this application, we can immediately note a few things in this architecture:
 - Each subsystem has now become an independent system by itself, a microservice.
 - Each service encapsulates its own database as well as its own HTTP listener.
 - Each microservice exposes a REST service to manipulate the resources/entity that belong to this service.

- Supports Polygot Architecture
- Enabling experimentation and innovation
- Elastically and selectively scalable
- Allowing substitution
- Enabling to build organic systems
- Helping reducing technology debt
- Allowing the coexistence of different versions
- Supporting the building of self-organizing systems
- Supporting event-driven architecture
- Enabling DevOps



Relationship with other architecture styles

- We will explore the relationship of microservices with other closely related architecture styles such as SOA and Twelve-Factor Apps

- The definition of SOA from The Open Group consortium is as follows:

"Service-Oriented Architecture (SOA) is an architectural style that supports service orientation. Service orientation is a way of thinking in terms of services and service-based development and the outcomes of services.
- A service:
 - Is a logical representation of a repeatable business activity that has a specified outcome
 - It is self-contained.
 - It may be composed of other services.
 - It is a "black box" to consumers of the service."

- Twelve-Factor App, forwarded by Heroku, is a methodology describing the characteristics expected from modern cloud-ready applications.
- Twelve-Factor App is equally applicable for microservices as well. Hence, it is important to understand Twelve-Factor App.
- See 12factor.net/config

III. Config

Store config in the environment

An app's config is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:

- Resource handles to the database, Memcached, and other backing services
- Credentials to external services such as Amazon S3 or Twitter
- Per-deploy values such as the canonical hostname for the deploy



Microservices early adopters

- Netflix
- Uber
- Airbnb
- Orbitz
- eBay
- Amazon
- Gilt
- Twitter
- Nike



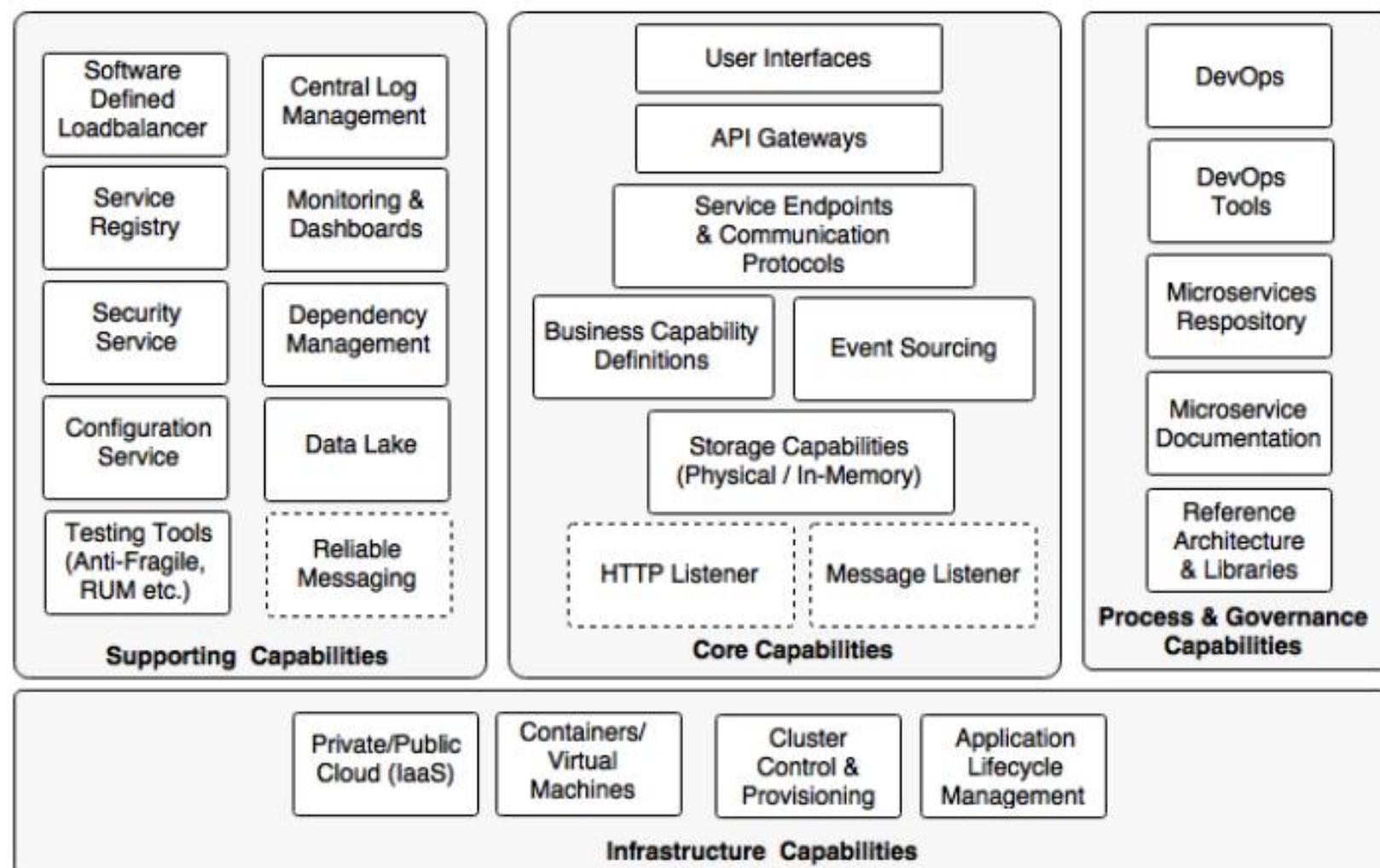
Day - 1

The Microservices Capability Model



The microservices capability model

- The following diagram depicts the microservices capability model:





The microservices capability model

The capability model is broadly classified in to four areas:

- **Core capabilities:** These are part of the microservices themselves
- **Supporting capabilities:** These are software solutions supporting core microservice implementations
- **Infrastructure capabilities:** These are infrastructure level expectations for a successful microservices implementation
- **Governance capabilities:** These are more of process, people, and reference information

The core capabilities are explained as follows:

- Service listeners (HTTP/messaging)
- Storage capability
- Business capability definition
- Event sourcing
- Service endpoints and communication protocols
- API gateway
- User interfaces

The Infrastructure capabilities are explained as follows:

- Cloud
- Containers or virtual machines
- Cluster control and provisioning
- Application lifecycle management

The Supporting capabilities are explained as follows:

- Software defined Load Balancer
- Central log management
- Service registry
- Security service
- Service configuration
- Testing tools (anti-fragile, RUM and so on)
- Monitoring and dashboards
- Dependency and CI management
- Reliable Messaging

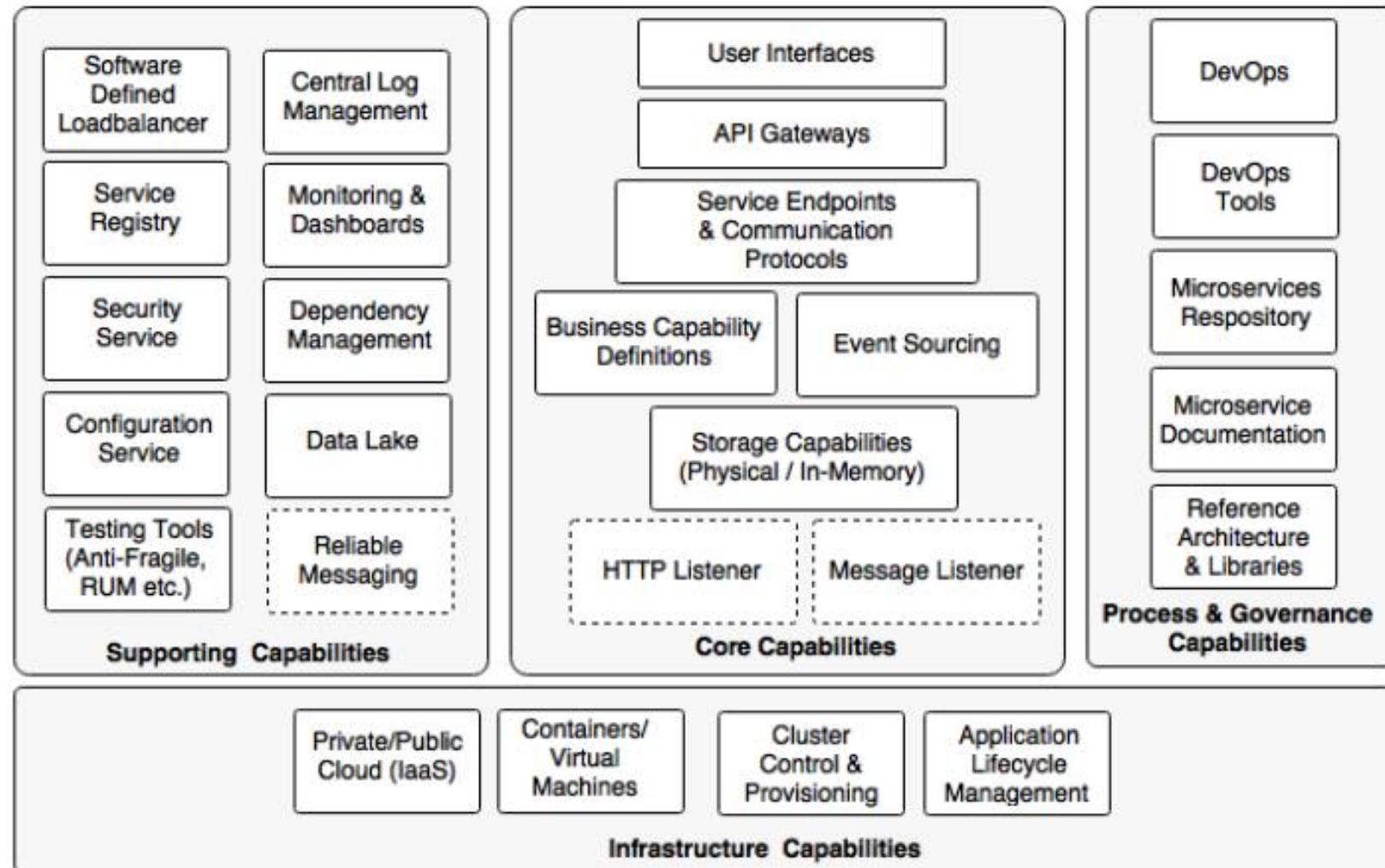
The Process and governance capabilities are explained as follows:

- DevOps
- DevOps tools
- Microservices repository
- Microservices documentation
- Reference architecture and libraries

We will explore the following microservices capabilities from the microservices capability model:

- Software Defined Load Balancer
- Service Registry
- Configuration Service
- Reliable Cloud Messaging
- API Gateways

Reviewing microservices capabilities



- What Is Monolithic Architecture?
- Concerns With the Monolith
- The Microservice architecture
- Characteristics of a Microservice Architecture
- Principles of microservices
- Business demand as a catalyst for Microservices
- Technology as a catalyst for the microservices evolution
- Building microservices with spring boot
- The microservices capability model



Day - 2



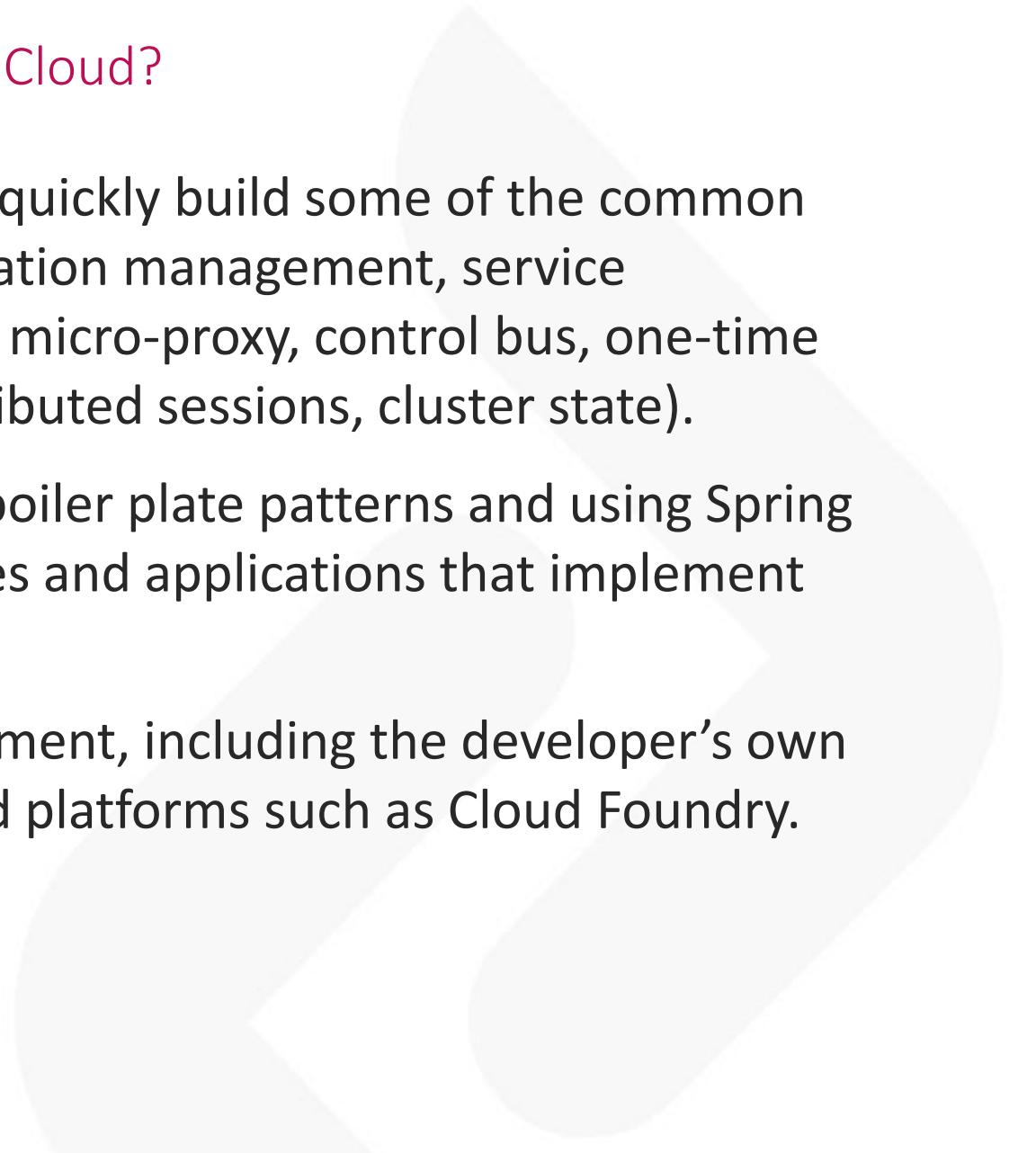
Day – 2 Agenda

- What is Spring Cloud?
- Components of Spring Cloud
- Spring Cloud Configuration – Centralized, Versioned Configuration
- Set up a Git repository
- Setting up the Config Server
- Accessing the Config Server from Clients
- Spring Cloud Config: How to use multiple configs



Day - 2

Spring Cloud



What is Spring Cloud?

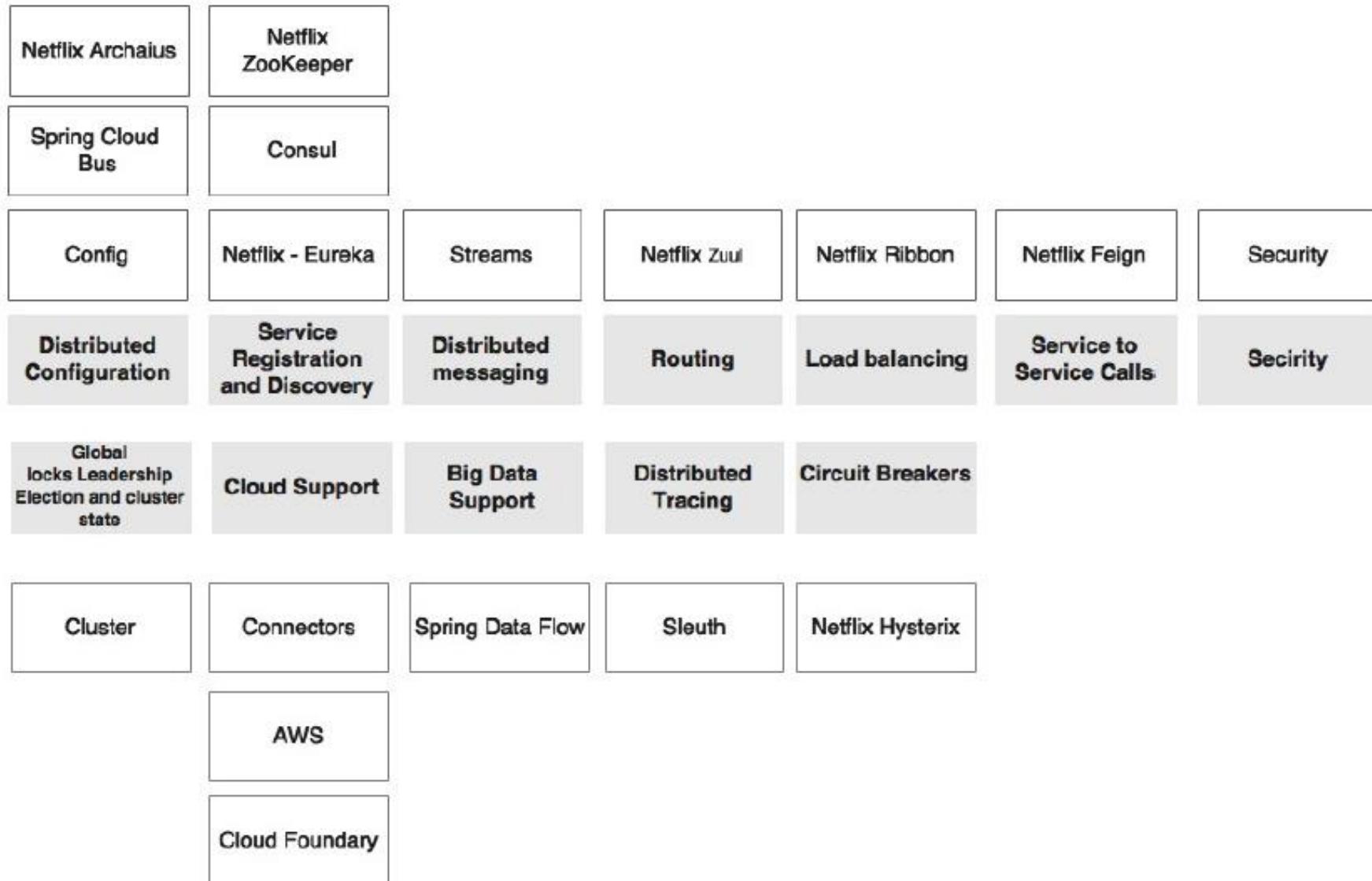
- Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g., configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state).
- Coordination of distributed systems leads to boiler plate patterns and using Spring Cloud developers can quickly stand-up services and applications that implement those patterns.
- They will work well in any distributed environment, including the developer's own laptop, bare metal data centers, and managed platforms such as Cloud Foundry.

Spring Cloud focuses on providing good out of box experience for typical use cases and extensibility mechanism to cover others:

- Distributed/versioned configuration
- Service registration and discovery
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers
- Global locks
- Leadership election and cluster state
- Distributed messaging



Components of Spring Cloud



- Many of the Spring Cloud components which are critical for microservices deployment came from the **Netflix Open-Source Software (Netflix OSS)** center.
- Netflix is one of the pioneers and early adaptors in the microservices space. In order to manage large scale microservices, engineers at Netflix produced several homegrown tools and techniques for managing their microservices.
- Later, Netflix open-sourced these components, and made them available under the **Netflix OSS platform** for public use.
- These components are extensively used in production systems and are battle-tested with large scale microservice deployments at Netflix.
- Spring Cloud offers higher levels of abstraction for these Netflix OSS components, making it more Spring developer friendly. It also provides a declarative mechanism well-integrated and aligned with Spring Boot and the Spring framework.



Day - 2

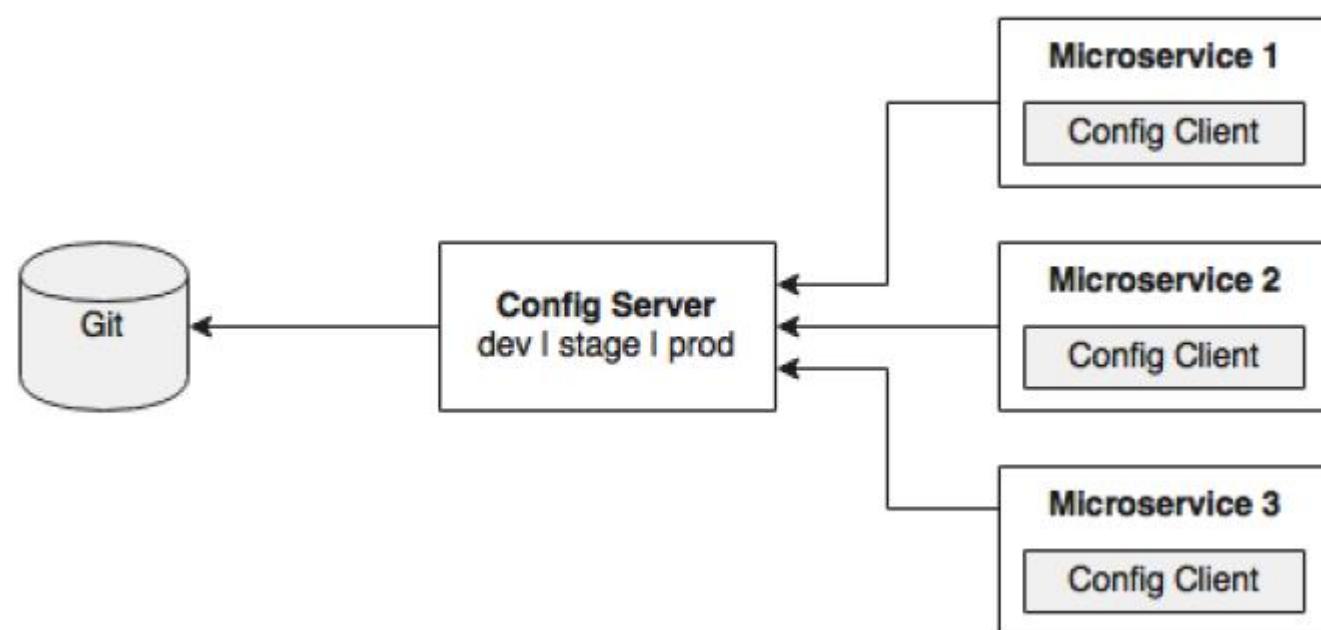
Spring Cloud Config

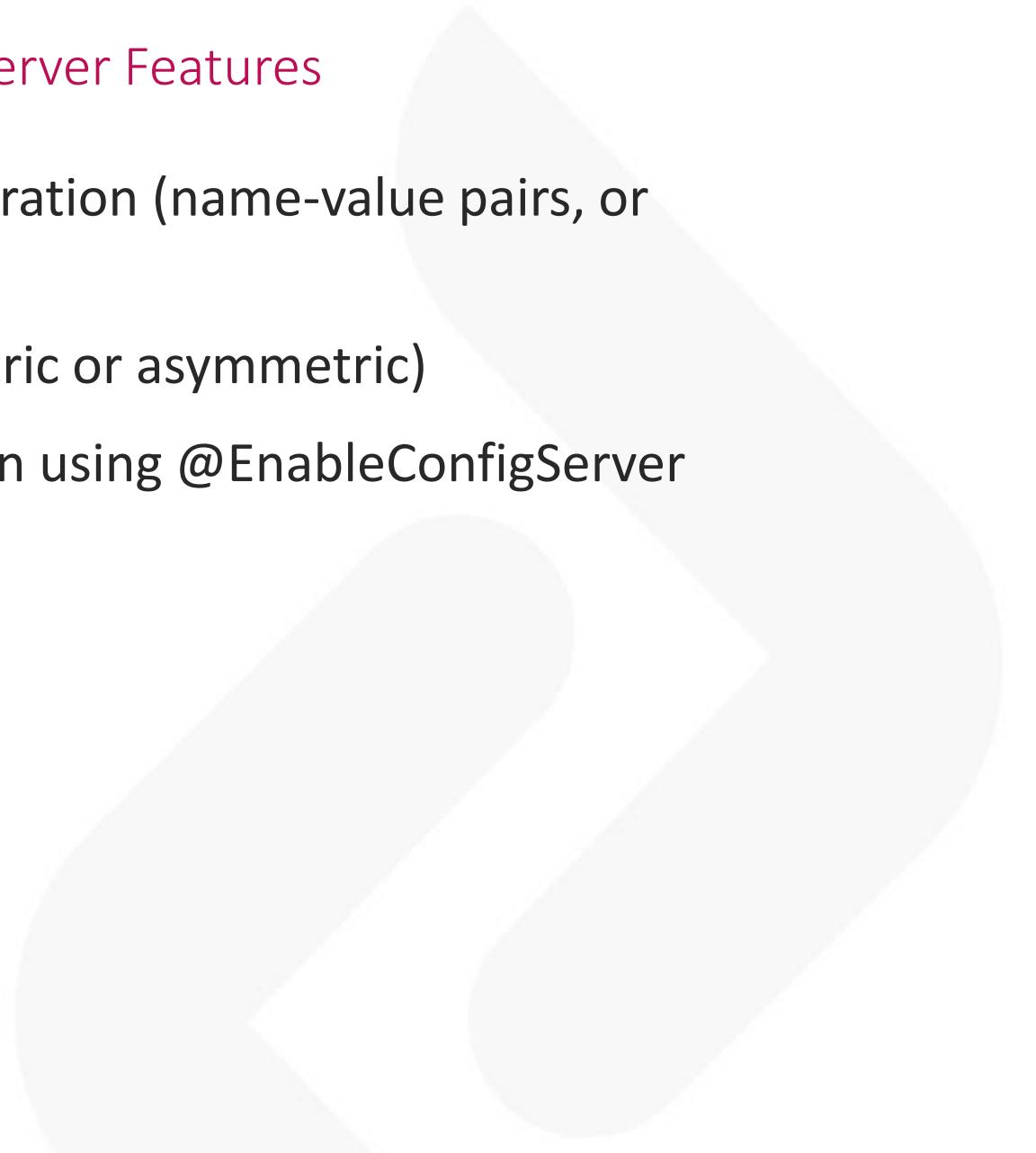
- The Spring Cloud Config server is an externalized configuration server in which applications and services can deposit, access, and manage all runtime configuration properties.
- The Spring Config server also supports version control of the configuration properties.



Spring Cloud Config Server

- The Spring Config server stores properties in a version-controlled repository such as Git or SVN. The Git repository can be local or remote. A highly available remote Git server is preferred for large scale distributed microservice deployments.
- The Spring Cloud Config server architecture is shown in the following diagram:





Spring Cloud Config Server Features

- HTTP, resource-based API for external configuration (name-value pairs, or equivalent YAML content)
- Encrypt and decrypt property values (symmetric or asymmetric)
- Embeddable easily in a Spring Boot application using `@EnableConfigServer`



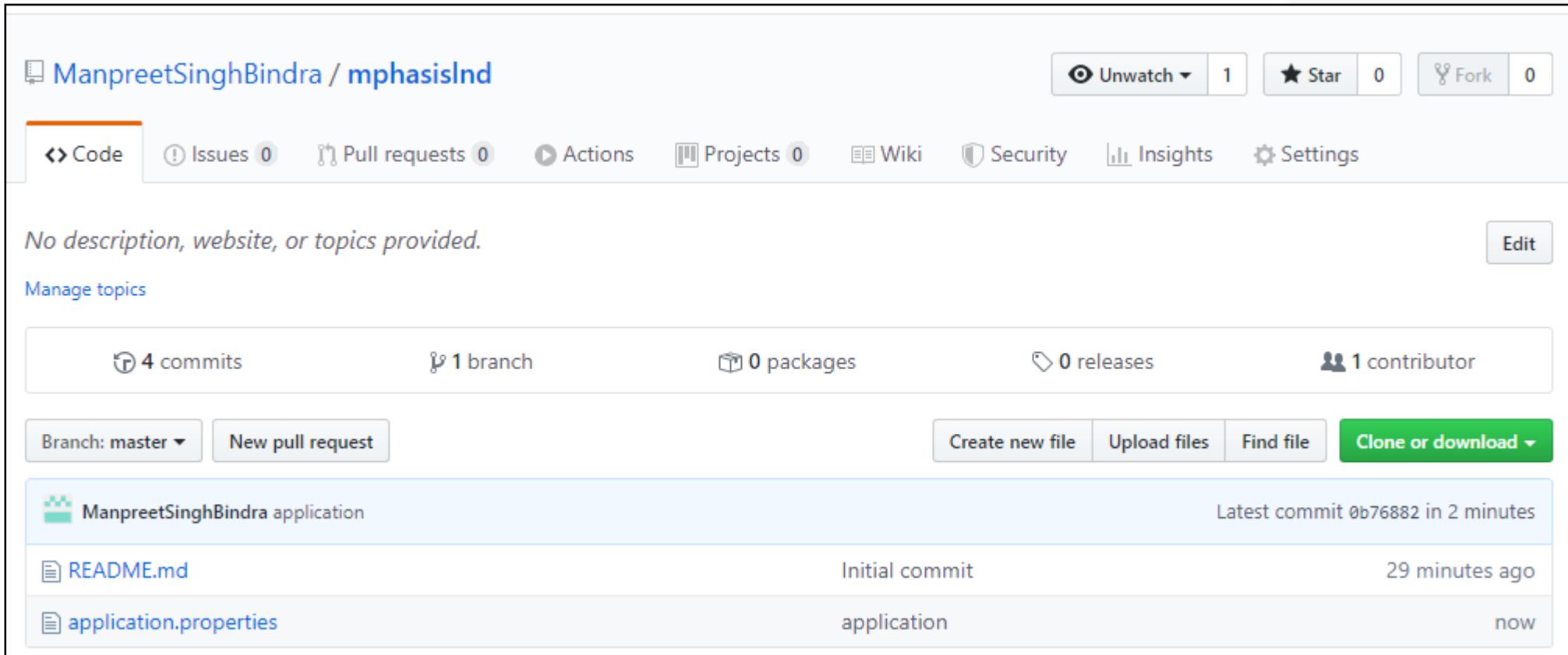
Config Client Features

- Bind to the Config Server and initialize Spring Environment with remote property sources
- Encrypt and decrypt property values (symmetric or asymmetric)



Setting up the Config Server

1. Set up a Git repository and upload the application.properties file



The screenshot shows a GitHub repository page for the user 'ManpreetSinghBindra' with the repository name 'mphasisInd'. The repository has 4 commits, 1 branch, 0 packages, 0 releases, and 1 contributor. The latest commit was made 29 minutes ago. The repository contains files like 'README.md' and 'application.properties'.

ManpreetSinghBindra / mphasisInd

Code Issues 0 Pull requests 0 Actions Projects 0 Wiki Security Insights Settings

No description, website, or topics provided. Edit

Manage topics

4 commits 1 branch 0 packages 0 releases 1 contributor

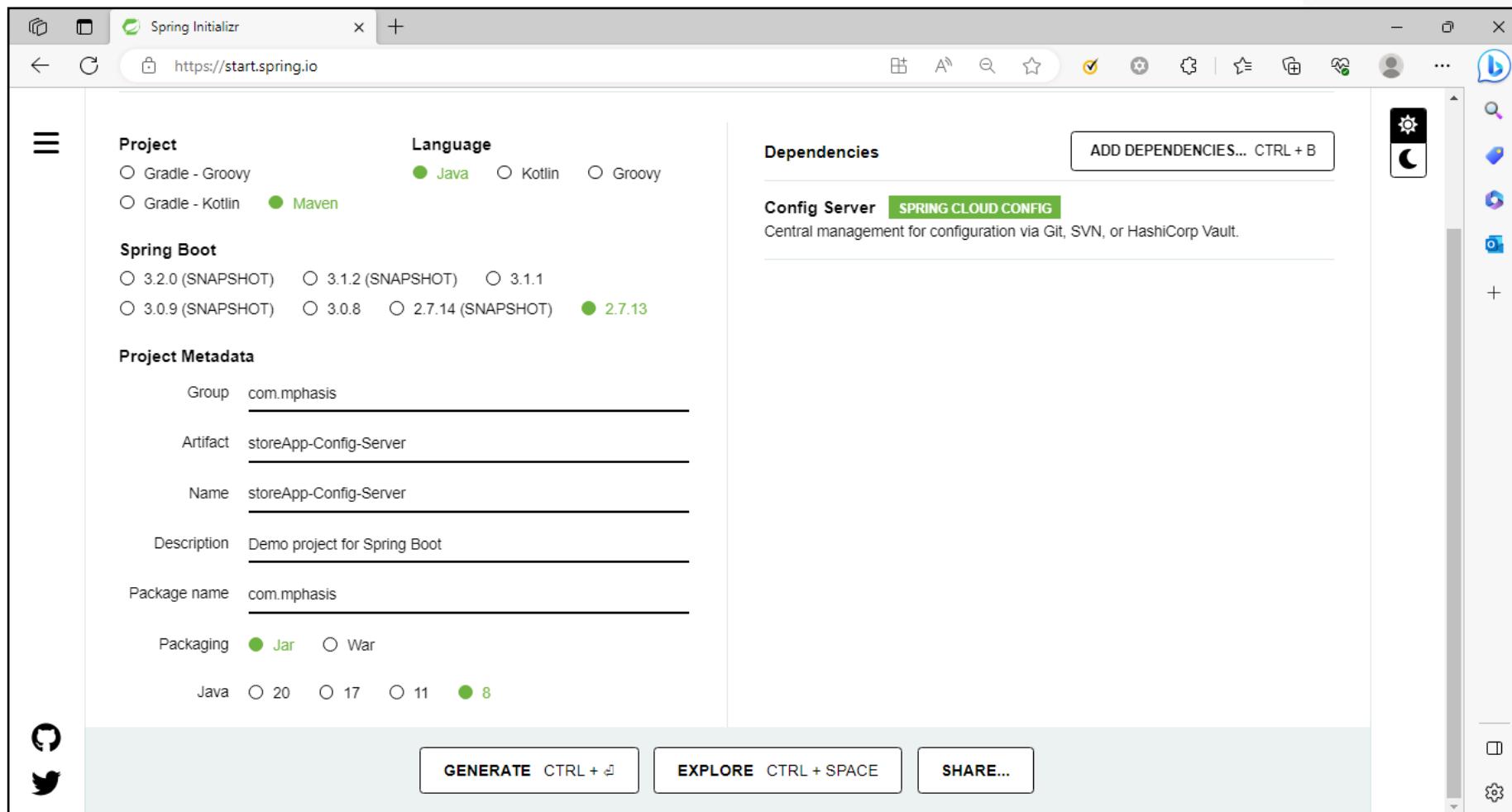
Branch: master New pull request Create new file Upload files Find file Clone or download

ManpreetSinghBindra application	Initial commit	Latest commit 0b76882 in 2 minutes
README.md		29 minutes ago
application.properties	application	now



Setting up the Config Server

2. Create a new Spring Starter Project, and select Config Server and Actuator as shown in the following diagram:





Setting up the Config Server

3. Add @EnableConfigServer in Application.java:

```
@EnableConfigServer
```

```
@SpringBootApplication
```

```
public class ConfigserverApplication { ... }
```



Setting up the Config Server

4. Edit the contents of the new application.properties file to match the following:

```
spring.port=8888
```

```
spring.cloud.config.server.git.uri=https://github.com/ManpreetSinghBindra/talentnext
```

```
#spring.cloud.config.server.git.username=
```

```
#spring.cloud.config.server.git.password=
```

```
#spring.cloud.config.server.git.clone-on-start=true
```

```
#spring.cloud.config.server.git.basedir=file://${user.dir}/cloned_configurations
```



Setting up the Config Server

5. Run the Config server by right-clicking on the project, and running it as a Spring Boot app.
6. Use the syntax listed below to test the eBookStore-Config-Server application:

```
http://localhost:8888/{name}/{profile}/{label}
```

7. Check **http://localhost:8888/application/default/master** to see the properties specific to application.properties



Accessing the Config Server from clients

- In the previous section, a Config server is set up and accessed using a web browser.
- In this section, the Product microservice will be modified to use the Config server. The Product microservice will act as a Config client.



Accessing the Config Server from clients

1. Add the Spring Cloud Config and actuator dependency.

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

2. The new application.properties/bootstrap.properties file will look as follows:

server.port=8080

#Before - Spring Boot 2.4 Version

#spring.cloud.config.uri=http://localhost:8888

#After - Spring Boot 2.4 Version

spring.config.import=optional:configserver:http://localhost:8888

spring.cloud.config.label=master

spring.application.name=application

3. Start the Config server.

4. Then start the Product microservice.



Working with Multiple Profiles

- Now the team planned to deploy the application in the multiple stages with multiple profiles, using the H2 and MySQL databases.
- Here will have multiple layers into the application:
 1. No changes in the source code of the application.
 2. Create the below properties files:
 - a) application.properties file for setting the profile if required or use command line.
 - b) application-dev.properties file with the H2 configuration.
 - c) application-prod.properties file with the MySQL configuration.

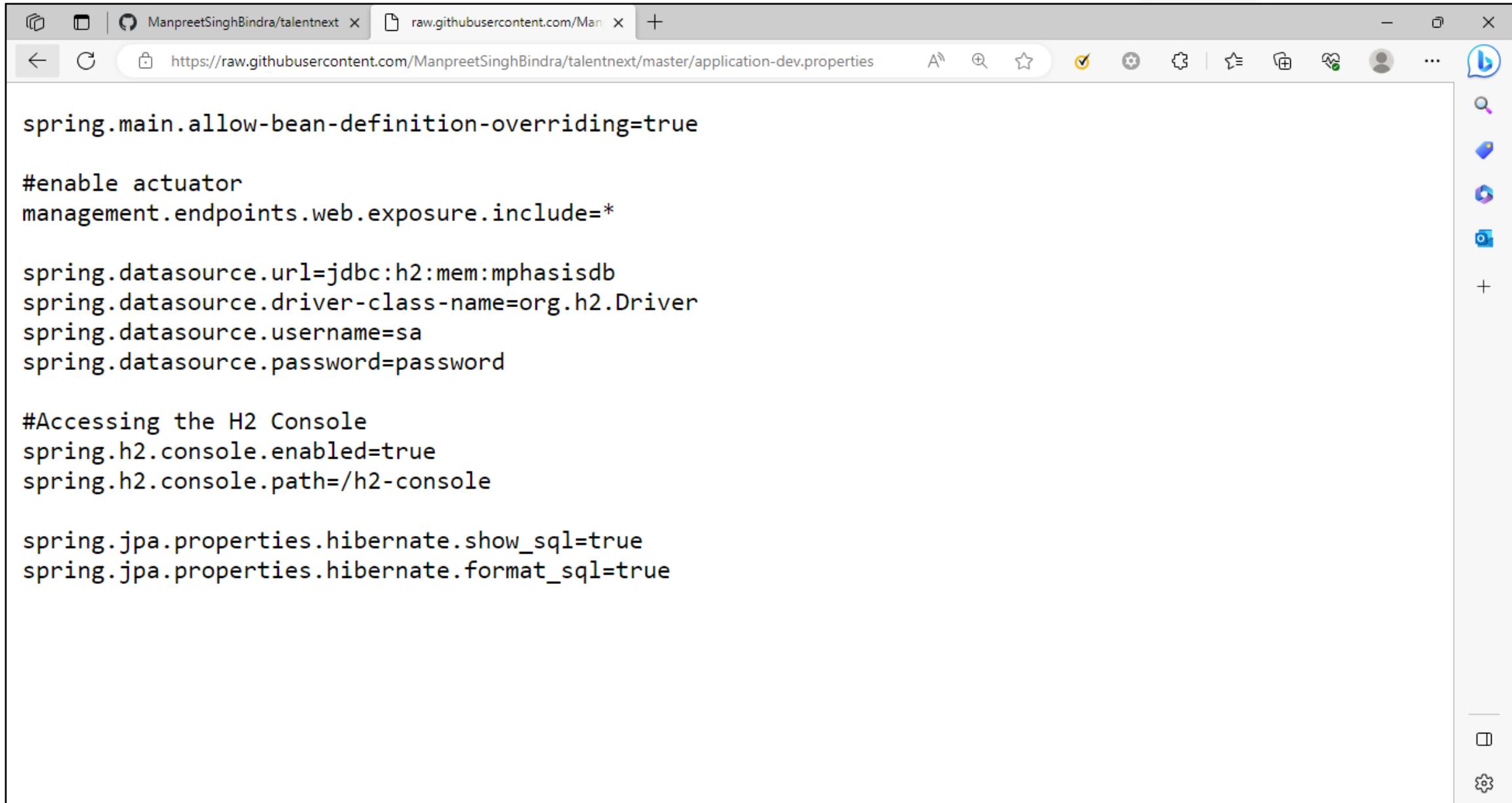


Working with Multiple Profiles

```
application.properties ✘
1
2 server.port=8080
3
4 #Before-Spring Boot 2.4 Version
5 #spring.cloud.config.uri=http://localhost:8888
6
7 #After-Spring Boot 2.4 Version
8 spring.config.import=optional:configserver:http://localhost:8888
9
10 spring.cloud.config.label=master
11 spring.application.name=application
12
```



Working with Multiple Profiles



The screenshot shows a browser window displaying a GitHub raw file at <https://raw.githubusercontent.com/ManpreetSinghBindra/talentnext/master/application-dev.properties>. The file contains the following Spring properties:

```
spring.main.allow-bean-definition-overriding=true

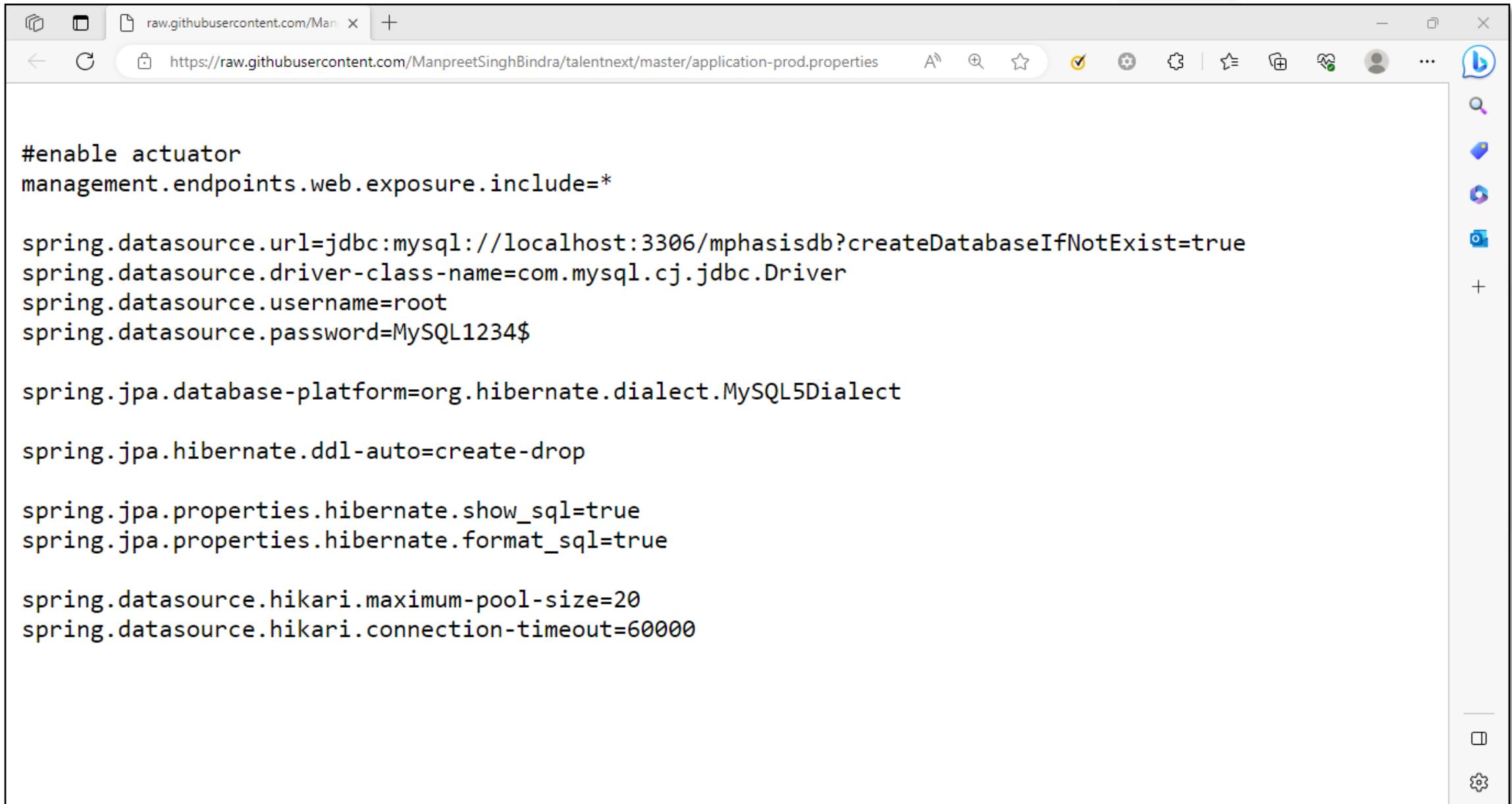
#enable actuator
management.endpoints.web.exposure.include=*

spring.datasource.url=jdbc:h2:mem:mphasisdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password

#Accessing the H2 Console
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Working with Multiple Profiles



The screenshot shows a web browser window displaying a configuration file from GitHub. The URL in the address bar is <https://raw.githubusercontent.com/ManpreetSinghBindra/talentnext/master/application-prod.properties>. The browser interface includes a back button, forward button, search bar, and various extension icons. The configuration file content is as follows:

```
#enable actuator
management.endpoints.web.exposure.include=*

spring.datasource.url=jdbc:mysql://localhost:3306/mphasisdb?createDatabaseIfNotExist=true
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=MySQL1234$

spring.jpa.database-platform=org.hibernate.dialect.MySQL5Dialect

spring.jpa.hibernate.ddl-auto=create-drop

spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true

spring.datasource.hikari.maximum-pool-size=20
spring.datasource.hikari.connection-timeout=60000
```



Working with Multiple Profiles

- Execute the application by switching the profile to Prod/Dev using command line/application.properties files.

```
java -Dspring.profiles.active=dev -jar storeapp-0.0.1-SNAPSHOT.jar  
/  
java -Dspring.profiles.active=prod -jar storeapp-0.0.1-SNAPSHOT.jar
```



Recap of Day – 2

- What is Spring Cloud?
- Components of Spring Cloud
- Spring Cloud Configuration – Centralized, Versioned Configuration
- Set up a Git repository
- Setting up the Config Server
- Accessing the Config Server from Clients
- Spring Cloud Config: How to use multiple configs



Day - 3



Day – 3 Agenda

- Changing the application name for the Spring Cloud Config Client
- Spring Cloud Config Client Refresh Strategies
- Enabling Auto-Refresh of Configuration
- Dynamically manage and refreshable configuration with Spring Cloud Bus
- Spring Cloud Netflix
- Understanding Dynamic Service Registration and Discovery
- Understanding Eureka
- Setting up the Eureka server
- Enable dynamic registration and discovery to our microservice



Changing the application name for the Spring Cloud Config Client

1. Rename the properties files as follows on the GitHub:
 - a. application-dev.properties to product-service-dev.properties
 - b. application-prod.properties to product-service-prod.properties
2. In storeapp - application.properties files, add the following properties:

```
application.properties ✎  
1  
2 server.port=8080  
3  
4 #Before-Spring Boot 2.4 Version  
5 #spring.cloud.config.uri=http://localhost:8888  
6  
7 #After-Spring Boot 2.4 Version  
8 spring.config.import=optional:configserver:http://localhost:8888  
9  
10 spring.cloud.config.label=master  
11 spring.application.name=product-service  
12 |
```



Changing the application name for the Spring Cloud Config Client

3. Start the storeApp-Config-Server.
4. Test the storeApp-Config-Server application with the following Profiles:

```
-----  
product-service-dev.properties  
http://localhost:8888/product-service/dev/master  
  
product-service-prod.properties  
http://localhost:8888/product-service/prod/master  
-----
```

5. After that, launch the storeapp application using VM arguments to configure the profiles.

```
-----  
VM Arguments:  
-Dspring.profiles.active=dev  
-Dspring.profiles.active=prod  
-----
```



Spring Cloud Config Client Refresh Strategies

- The **/actuator/refresh** endpoint is used for the first way of property refreshment. This endpoint is exposed in configuration clients, a call to it just refreshes the client to which the request is made.
- Assume that we have changed the maximum-pool-size in properties file in the git repository to 20 and want to reflect this change in our application at runtime.

```
"spring.datasource.hikari.maximum-pool-size": "20",
```

- Next, we make a rest call to the **/actuator/refresh** endpoint exposed by the config client application.



Spring Cloud Config Client Refresh Strategies

- Either use curl or Postman to send the rest call to the **/actuator/refresh** endpoint.
- Using the Postman tool:

The screenshot shows the Postman application interface. A POST request is being made to `http://localhost:8080/actuator/refresh`. The response body is displayed in Pretty JSON format, showing the following configuration keys:

```
1 [ "config.client.version", "spring.datasource.hikari.maximum-pool-size" ]
```



Enabling Auto-Refresh of Configuration

- **@RefreshScope** helps in providing a new scope for defining the bean. It refreshes the property sources as well as the bean.
- The attributes bounded with **@Value** in beans with the annotation **@RefreshScope** are refreshed immediately after reloading property sources via the **/actuator/refresh** endpoint.



Steps for Enabling Auto-Refresh of Configuration

1. Create a new **ValueRefreshConfigBean** class for apply the theme to your application.

```
@Component
@RefreshScope
public class ValueRefreshConfigBean {

    private String color;

    public ValueRefreshConfigBean(
        @Value("${application.theme.color}") String color) {

        this.color = color;
    }

    public String getColor() {
        return color;
    }
}
```

2. Include the following custom property to product-service-prod.properties file:

```
"application.theme.color": "dark"
```



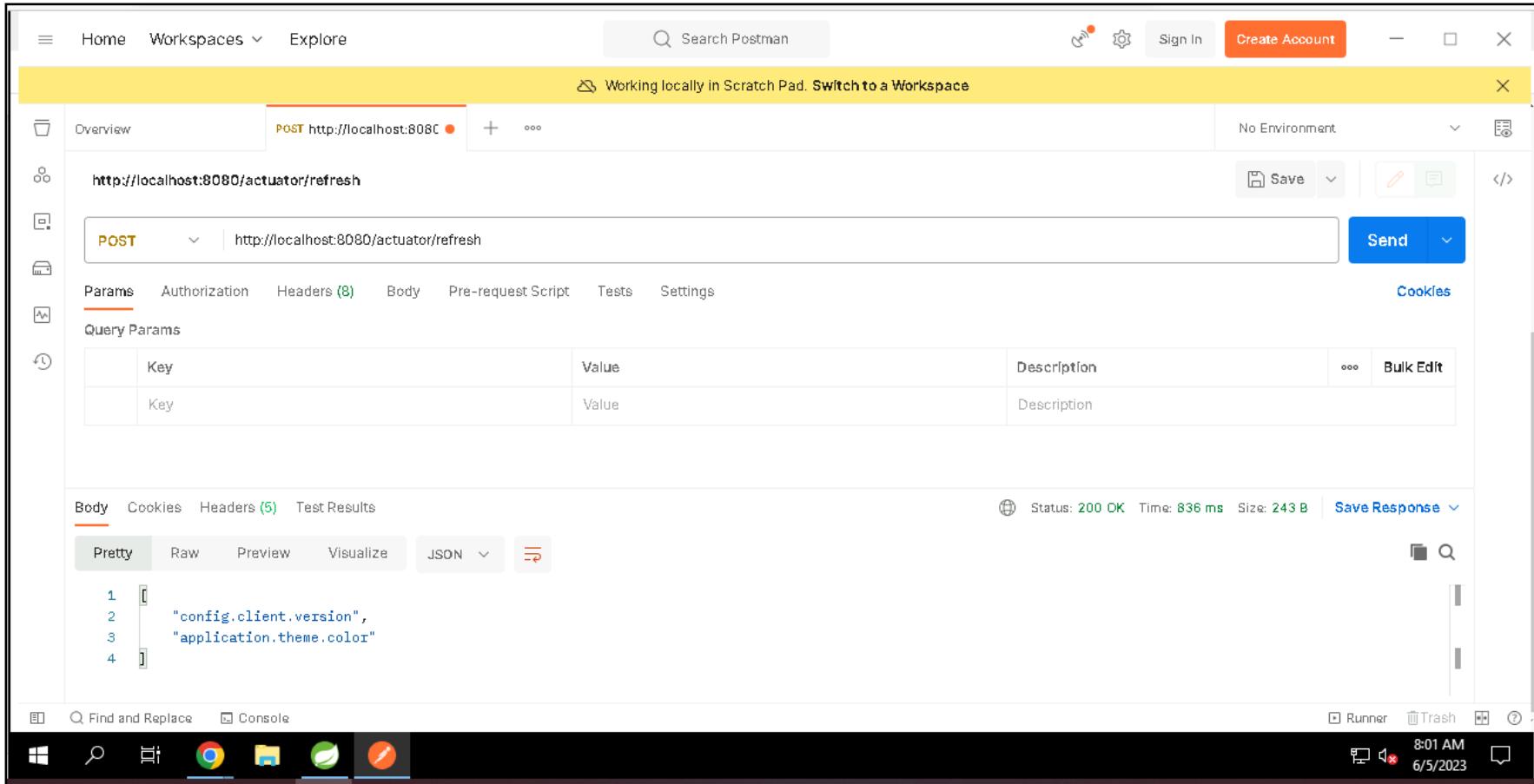
Steps for Enabling Auto-Refresh of Configuration

3. Start the storeApp-Config-server and storeApp application.
4. Verify the current environment variables.
5. Assume that we have changed the **application theme** in properties files in the git repository and want to reflect these changes in our application at runtime.
6. Either use curl or Postman to send the rest call to the /actuator/refresh endpoint.



Steps for Enabling Auto-Refresh of Configuration

7. Using the Postman tool:



The screenshot shows the Postman application interface. At the top, there is a navigation bar with Home, Workspaces, Explore, a search bar, and account options. A yellow banner at the top center says "Working locally in Scratch Pad. Switch to a Workspace". Below the banner, the main workspace shows a single collection named "Overview" with one item: "http://localhost:8080/actuator/refresh". The item details show a POST method and the URL. The "Params" tab is selected, showing a table for "Query Params" with two rows: "Key" and "Value". The "Body" tab is selected, showing a JSON response with four lines of code. The status bar at the bottom indicates a 200 OK response with a size of 243 B.

```
1 [ "config.client.version", "application.theme.color" ]
```



Day - 3

Spring Cloud Bus



Spring Cloud Bus

- Spring Cloud Bus links nodes of a distributed system with a lightweight message broker.
- This can then be used to broadcast state changes (e.g., configuration changes) or other management instructions.
- AMQP and Kafka broker implementations are included with the project.
- Alternatively, any Spring Cloud Stream binder found on the classpath will work out of the box as a transport.



Dynamically manage and refreshable configuration with Spring Cloud Bus

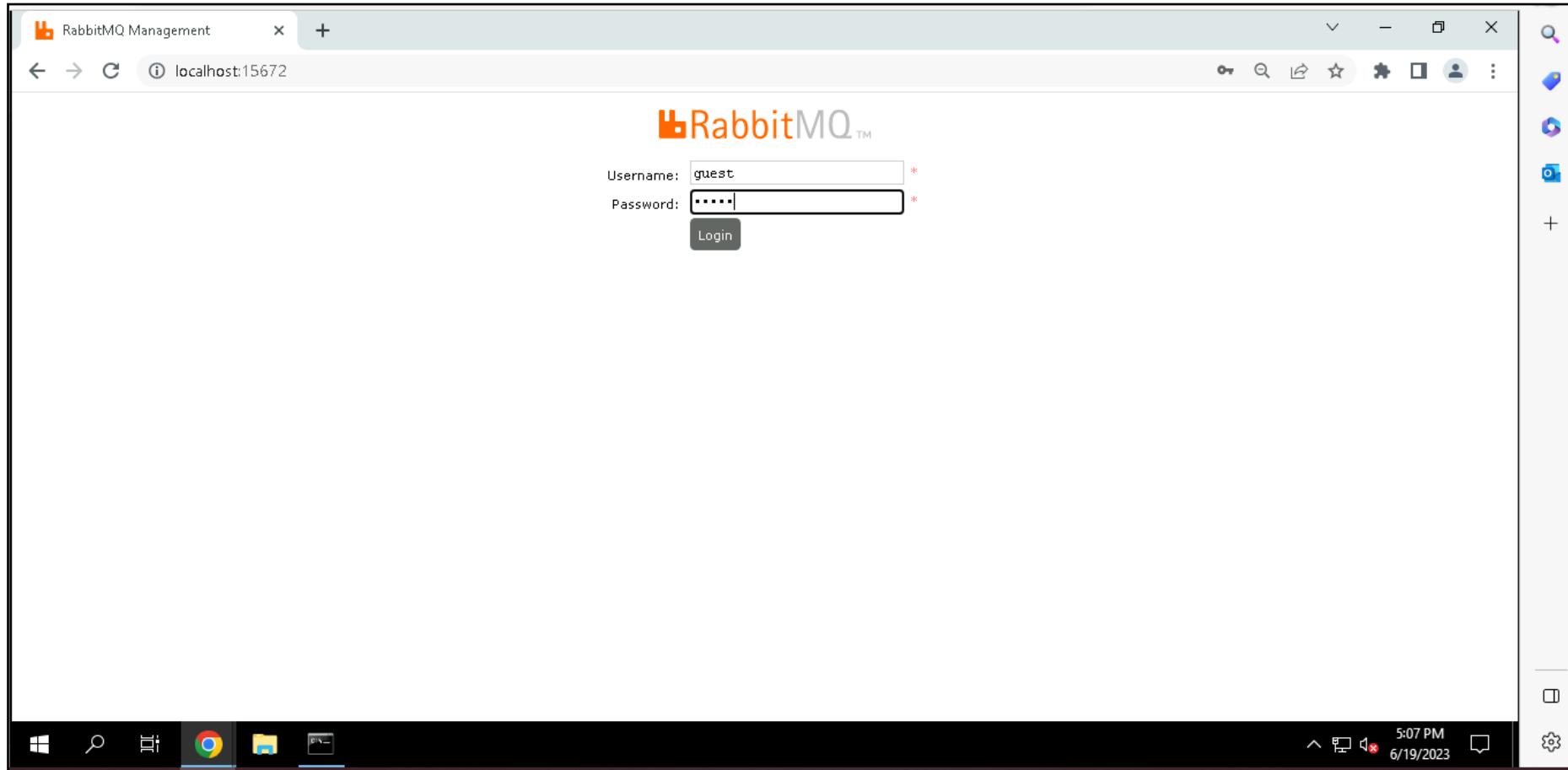
1. To Download and Run Rabbit MQ, go to <https://rabbitmq.com>
2. Click on Download + Installation button.
3. Open the command prompt and execute the below docker command:

```
# Latest RabbitMQ 3.12
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.12-management
```



Dynamically manage and refreshable configuration with Spring Cloud Bus

4. Access the RabbitMQ dashboard via <http://localhost:15672>



5. The default username and password of RabbitMQ is guest.



Dynamically manage and refreshable configuration with Spring Cloud Bus

6. Enhance the **storeApp-Config-Server** application.
7. Add the Spring Cloud Bus and Actuator dependencies to pom.xml:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```



Dynamically manage and refreshable configuration with Spring Cloud Bus

8. To enable the **/busrefresh** URL endpoint, add the below properties to application.properties:

```
application.properties ✎
1
2 server.port=8888
3
4 spring.cloud.config.server.git.uri=https://github.com/ManpreetsinghBindra/talentnext
5 #spring.cloud.config.server.git.username=
6 #spring.cloud.config.server.git.password=
7 #spring.cloud.config.server.git.clone-on-start=true
8 #spring.cloud.config.server.git.basedir=file://${user.dir}/cloned_configurations
9
10 management.endpoints.web.exposure.include=busrefresh
11
12 spring.rabbitmq.host=localhost
13 spring.rabbitmq.port=5672
14 spring.rabbitmq.username=guest
15 spring.rabbitmq.password=guest
16
```



Dynamically manage and refreshable configuration with Spring Cloud Bus

9. Enhance the **storeApp** application.
10. Add the Spring Cloud Bus dependency to pom.xml:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter/bus-amqp</artifactId>
</dependency>
```



Dynamically manage and refreshable configuration with Spring Cloud Bus

11. Add the following properties in application.properties file:

```
application.properties
1
2 server.port=8080
3
4 #Before-Spring Boot 2.4 Version
5 #spring.cloud.config.uri=http://localhost:8888
6
7 #After-Spring Boot 2.4 Version
8 spring.config.import=optional:configserver:http://localhost:8888
9
10 spring.cloud.config.label=master
11
12 #spring.application.name=application
13 spring.application.name=product-service
14
15 spring.rabbitmq.host=localhost
16 spring.rabbitmq.port=5672
17 spring.rabbitmq.username=guest
18 spring.rabbitmq.password=guest
19 |
```



Dynamically manage and refreshable configuration with Spring Cloud Bus

12. Start the storeApp-Config-server and storeApp application.
13. Observe the value of “**spring-datasource-hikari.maximum-pool-size**” attribute which is remotely fetched during the startup of the application using the actuator env endpoint.
14. Now go to remote git repository and update the value in the “**product-service-prod.properties**”.



Dynamically manage and refreshable configuration with Spring Cloud Bus

15. Let's use the **/actuator/busrefresh** endpoint at **Config Server** to push the configuration while they are running. The microservices to get these updates do not need to be restart.

The screenshot shows the Postman application interface. On the left, the 'History' panel lists a single entry: a POST request to `http://localhost:8888/actuator/busrefresh`. The main workspace displays a POST request to the same URL. The 'Params' tab is selected, showing a table with two rows:

	Key	Value
	Key	Value

Below the table, the 'Body' tab is selected, showing the response body which contains the number '1'. The status bar at the bottom right indicates a 204 No Content response with a duration of 2.03s and a size of 112 B.



Day - 3

Spring Cloud Netflix



Spring Cloud Netflix

- Spring Cloud Netflix provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.
- With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components.
- The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client-Side Load Balancing (Ribbon)



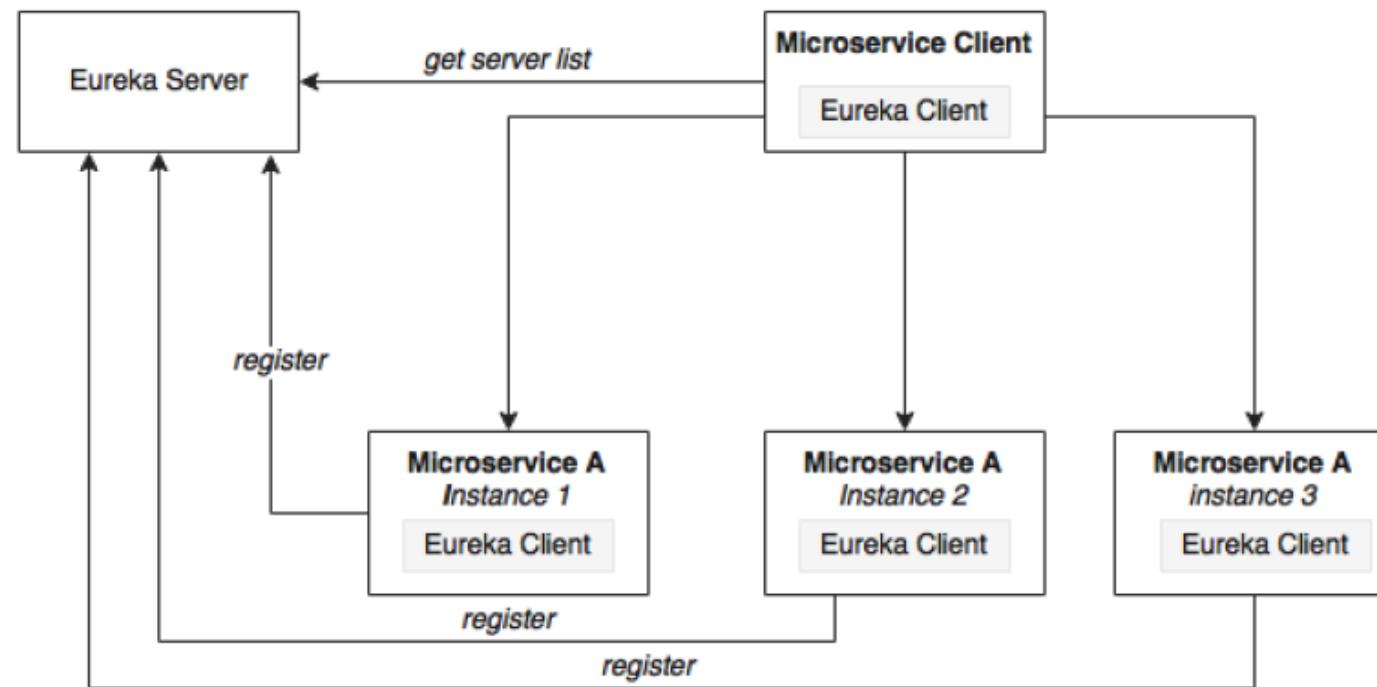
Understanding Dynamic Service Registration and Discovery

- **Dynamic registration** is primarily from the service provider's point of view. With dynamic registration, when a new service is started, it automatically enlists its availability in a central service registry. Similarly, when a service goes out of service, it is automatically delisted from the service registry. The registry always keeps up-to-date information of the services available, as well as their metadata.
- **Dynamic discovery** is applicable from the service consumer's point of view. Dynamic discovery is where clients look for the service registry to get the current state of the services topology, and then invoke the services accordingly. In this approach, instead of statically configuring the service URLs, the URLs are picked up from the service registry.
- There are several options available for dynamic service registration and discovery.
- *Netflix Eureka, ZooKeeper, and Consul* are available as part of Spring Cloud.



Understanding Eureka

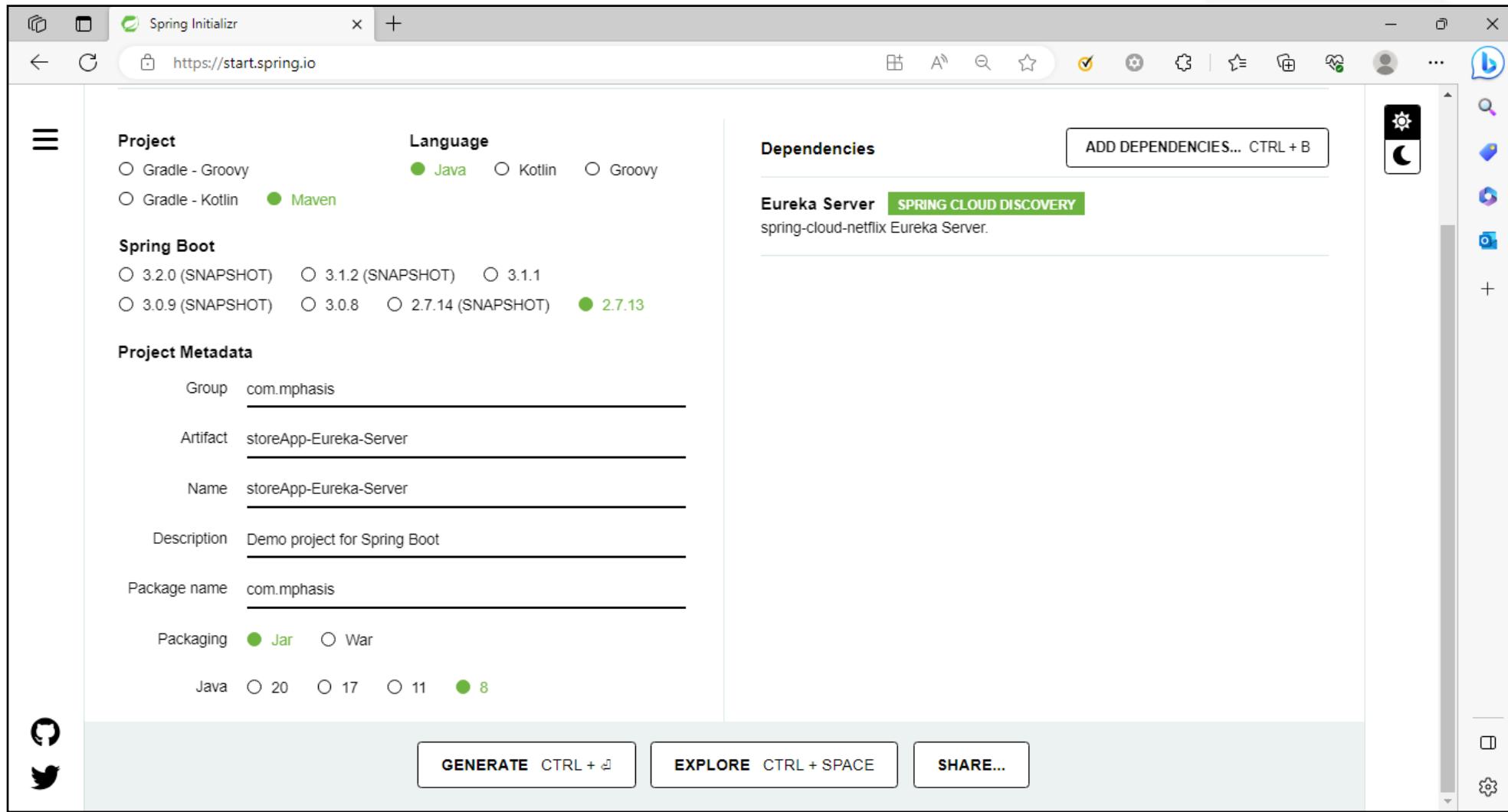
- Spring Cloud Eureka also comes from Netflix OSS. The Spring Cloud project provides a Spring-friendly declarative approach for integrating Eureka with Spring-based applications.
- Eureka is primarily used for self-registration, dynamic discovery, and load balancing.





Setting up the Eureka server

1. Start a new Spring Starter project, and select Eureka Server starter:



2. Create a application.properties/bootstrap.properties

```
server.port=8761  
eureka.client.register-with-eureka=false  
eureka.client.fetch-registry=false
```

3. Add @EnableEurekaServer in Application.java:

```
@EnableEurekaServer  
  
@SpringBootApplication  
  
public class EurekaServerApplication { ... }
```

4. We are now ready to start the Eureka server. Once the application is started, open <http://localhost:8761> in a browser to see the Eureka console.



Enable dynamic registration and discovery to our microservice

1. In storeapp application, add the Eureka dependencies to the pom.xml file.

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



Enable dynamic registration and discovery to our microservice

2. Add the following properties to application.properties file:

```
application.properties ✘

2 server.port=0
3
4 #Before-Spring Boot 2.4 Version
5 #spring.cloud.config.uri=http://localhost:8888
6
7 #After-Spring Boot 2.4 Version
8 spring.config.import=optional:configserver:http://localhost:8888
9
10 spring.cloud.config.label=master
11
12 #spring.application.name=application
13 spring.application.name=product-service
14
15 spring.rabbitmq.host=localhost
16 spring.rabbitmq.port=5672
17 spring.rabbitmq.username=guest
18 spring.rabbitmq.password=guest
19
20 eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
21 eureka.instance.preferIpAddress=true
22 |
```



Enable dynamic registration and discovery to our microservice

3. Add @EnableEurekaClient in Application.java:

@EnableEurekaClient

@SpringBootApplication

public class EurekaclientApplication { ... }

4. Ensure the RabbitMQ docker container, Config Server, Eureka Server and storeapp microservice is running.



Recap of Day – 3

- Changing the application name for the Spring Cloud Config Client
- Spring Cloud Config Client Refresh Strategies
- Enabling Auto-Refresh of Configuration
- Dynamically manage and refreshable configuration with Spring Cloud Bus
- Spring Cloud Netflix
- Understanding Dynamic Service Registration and Discovery
- Understanding Eureka
- Setting up the Eureka server
- Enable dynamic registration and discovery to our microservice



Day - 4



Day – 4 Agenda

- Client-Side Load Balancer: Ribbon
- Access from a client service
- Feign as a declarative REST client
- Create a ProductServiceProxy interface
- Have Service-to-Service calls
- Feign clients to be scanned and discovered



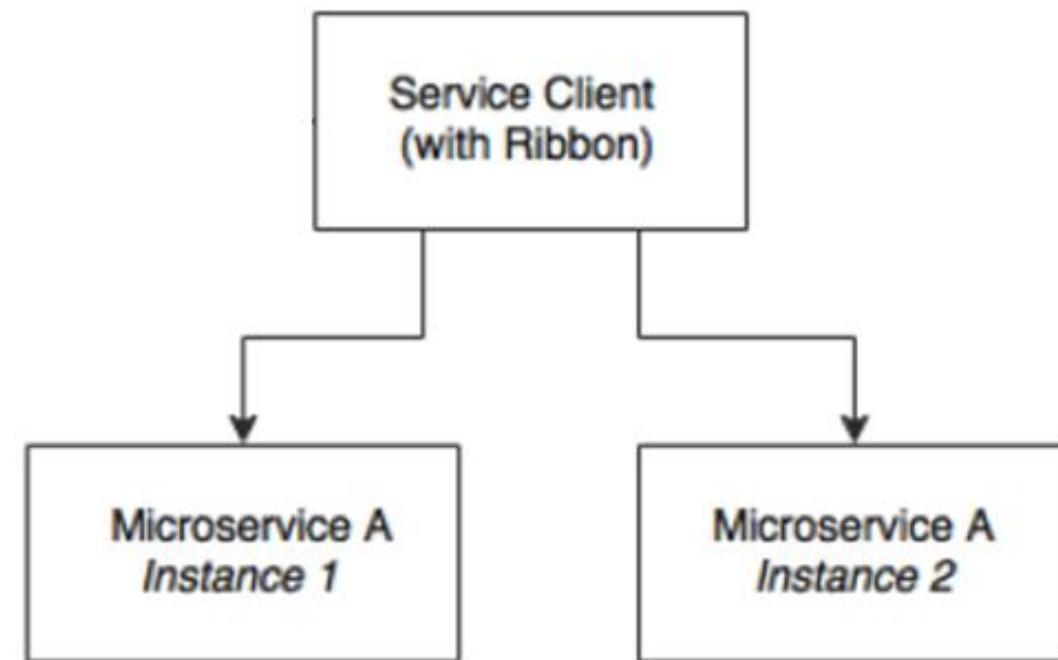
Day - 4

Client-Side Load Balancer: Ribbon



Ribbon for load balancing

- Netflix Ribbon is an Inter Process Communication (IPC) cloud library.
- Ribbon primarily provides client-side load balancing algorithms.





Ribbon for load balancing

- Ribbon provides also other features:
 - Service Discovery Integration
 - Fault Tolerance
 - Configurable load-balancing rules

- In order to use the Ribbon client, we will have to add the following dependency to the pom.xml file:

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

- In case of development from ground up, this can be selected from the Spring Starter libraries, or from <http://start.spring.io/>.

- Ribbon is available under **Cloud Routing**:

Cloud Routing

- Zuul**

Intelligent and programmable routing with `spring-cloud-netflix Zuul`

- Ribbon**

Client side load balancing with `spring-cloud-netflix` and **Ribbon**

- Feign**

Declarative REST clients with `spring-cloud-netflix Feign`



Client-side vs server-side load balancing

- The multiple instances of the same microservice is run on different computers for high reliability and availability.
- Server-side load balancing is distributing the incoming requests towards multiple instances of the service.
- Client-side load balancing is distributing the outgoing request from the client itself.

- Spring RestTemplate can be used for client-side load balancing
- Spring Netflix Eureka has a built-in client-side load balancer called Ribbon.
- Ribbon can automatically be configured by registering RestTemplate as a bean and annotating it with @LoadBalanced.

```
@Configuration
public class Config {

    @LoadBalanced
    @Bean
    public RestTemplate getRestTemplate() {
        return new RestTemplate();
    }
}
```

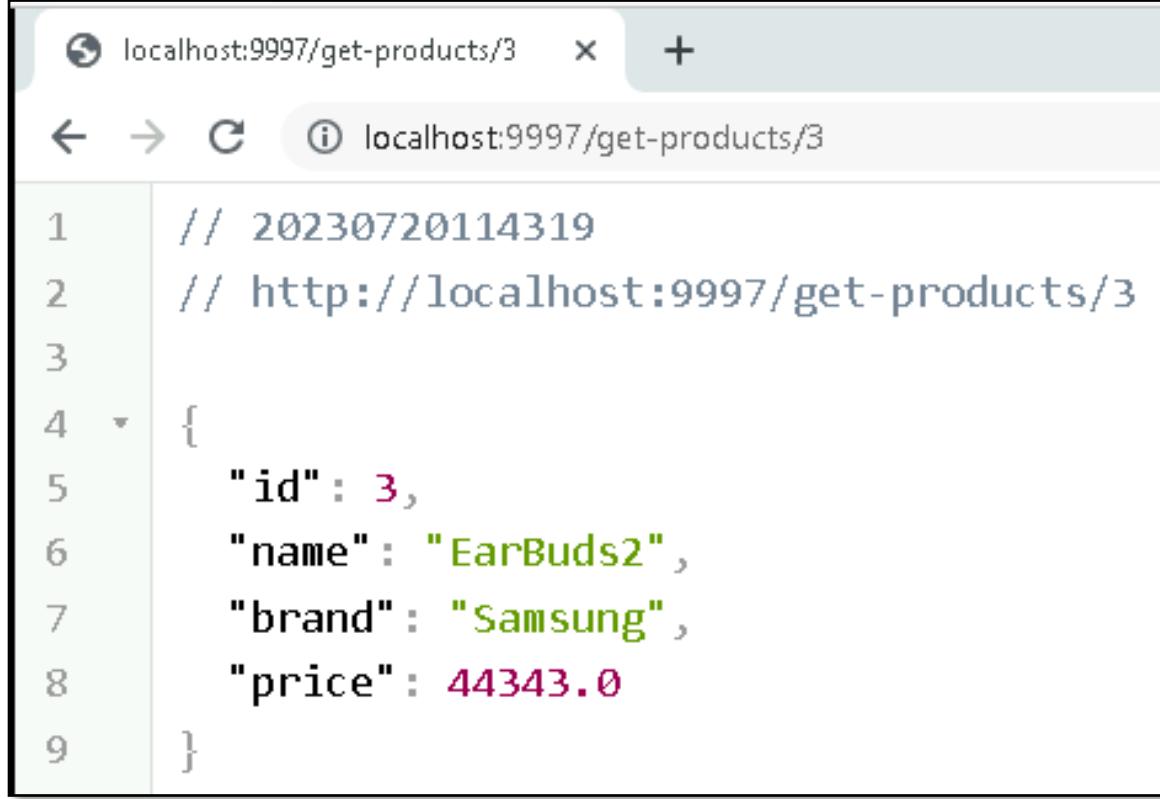


Access from a client service

```
@RestController  
@Scope("request")  
public class ProductClientController {  
  
    @Autowired  
    private RestTemplate restTemplate;  
  
    @GetMapping("/get-products/{id}")  
    public Product getProductId(@PathVariable("id") int id) {  
  
        Product product = restTemplate.getForObject(  
            "http://product-service/products/" + id,  
            Product.class );  
  
        return product;  
    }  
}
```



Test your Consumer



A screenshot of a web browser window. The address bar shows the URL `localhost:9997/get-products/3`. The page content displays a JSON object with the following structure:

```
1 // 20230720114319
2 // http://localhost:9997/get-products/3
3
4 {
5   "id": 3,
6   "name": "EarBuds2",
7   "brand": "Samsung",
8   "price": 44343.0
9 }
```



Day - 4

Spring Cloud OpenFeign



Feign as a declarative REST client

- Feign is a Spring Cloud Netflix library for providing a higher level of abstraction over REST-based service calls.
- Spring Cloud Feign offers a declarative approach for making RESTful service-to-service call in a synchronous way.
- When using Feign, we write declarative REST service interfaces at the client, and use those interfaces to program the client.
- The developer need not worry about the implementation of this interface.
- This will be dynamically provisioned by Spring at runtime.
- With this declarative approach, developers need not get into the details of the HTTP level APIs provided by RestTemplate.



Feign as a declarative REST client

- In order to use Feign, first we need to change the pom.xml file to include the Feign dependency as follows:

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

- For a new Spring Starter project, Feign can be selected from the starter library selection screen, or from <http://start.spring.io/>.
- This is available under **Cloud Routing** as shown in the following screenshot:

Cloud Routing

- Zuul**
Intelligent and programmable routing with spring-cloud-netflix Zuul
- Ribbon**
Client side load balancing with spring-cloud-netflix and Ribbon
- Feign**
Declarative REST clients with spring-cloud-netflix Feign



Create a ProductServiceProxy interface

- The next step is to create a new ProductServiceProxy interface.
- This will act as a proxy interface of the actual Product service:

```
@FeignClient("product-service")
public interface ProductServiceProxy {

    @GetMapping(value = "/products/{id}", produces = {
        MediaType.APPLICATION_JSON_VALUE})
    public Product getProductById(@PathVariable("id") int id);

    @GetMapping(value = "/products", produces = {
        MediaType.APPLICATION_JSON_VALUE})
    public ArrayList<Product> getAllProducts();
}
```



Have Service-to-Service calls

- The next step is to create a new ProductClientController class

```
@RestController
```

```
@Scope("request")
```

```
public class ProductClientController {
```

```
    @Autowired
```

```
    private ProductServiceProxy productServiceProxy;
```

```
    @GetMapping("/get-products/{id}")
```

```
    public Product getProductId(@PathVariable("id") int id) {
```

```
        Product product = productServiceProxy.getProductById(id);
```

```
        return product;
```

```
}
```

```
}
```



Feign clients to be scanned and discovered

- Add @EnableFeignClients in Application.java:

@EnableFeignClients

@SpringBootApplication

```
public class FeignclientApplication { ... }
```



Recap of Day – 4

- Client-Side Load Balancer: Ribbon
- Access from a client service
- Feign as a declarative REST client
- Create a ProductServiceProxy interface
- Have Service-to-Service calls
- Feign clients to be scanned and discovered



Day - 5



Day – 5 Agenda

- Isolating from failures
- Spring Cloud Hystrix for fault-tolerant microservices



Day - 5

Isolating from failures

- The circuit breaker subproject implements the circuit breaker pattern.
- The circuit breaker breaks the circuit when it encounters failures in the primary service by diverting traffic to another temporary fallback service.
- It also automatically reconnects back to the primary service when the service is back to normal.
- It finally provides a monitoring dashboard for monitoring the service state changes.
- The Spring Cloud Hystrix project and Hystrix Dashboard implement the circuit breaker and the dashboard, respectively.



Spring Cloud Hystrix for fault-tolerant microservices

- Spring Cloud Hystrix as a library for a fault-tolerant and latency-tolerant microservice implementation.
- Hystrix is based on the fail-fast and rapid recovery principles. If there is an issue with a service, Hystrix helps isolate it. It helps to recover quickly by falling back to another preconfigured fallback service.
- Hystrix is another battle-tested library from Netflix. Hystrix is based on the circuit breaker pattern.



Day - 5

Build a Circuit Breaker with Spring Cloud Hystrix with Ribbon



Build a circuit breaker with Spring Cloud Hystrix with Ribbon

1. Add the Hystrix dependency to the service.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

2. Change the version - Spring Boot and Spring Cloud:

- Spring Boot: 2.3.10.RELEASE
- Spring Cloud: Hoxton.SR11



Build a circuit breaker with Spring Cloud Hystrix with Ribbon

3. Add the following property to application.properties file:

```
application.properties ✘  
1  
2 server.port=9995  
3 eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka  
4 eureka.client.register-with-eureka=false  
5  
6 #Enable Actuator  
7 management.endpoints.web.exposure.include=*
```



Build a circuit breaker with Spring Cloud Hystrix with Ribbon

4. Create a ProductService class with **@HystrixCommand** annotation:

```
@Service
public class ProductService {

    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "fallbackMethodForGetProductById")
    public Product getProductId(int id) {

        Product product = restTemplate.getForObject("http://product-service/products/" + id,
            Product.class);
        return product;
    }

    public Product fallbackMethodForGetProductById(int id) {

        return new Product(id, "Monitor", "Jio", 34343.0);
    }
}
```



Build a circuit breaker with Spring Cloud Hystrix with Ribbon

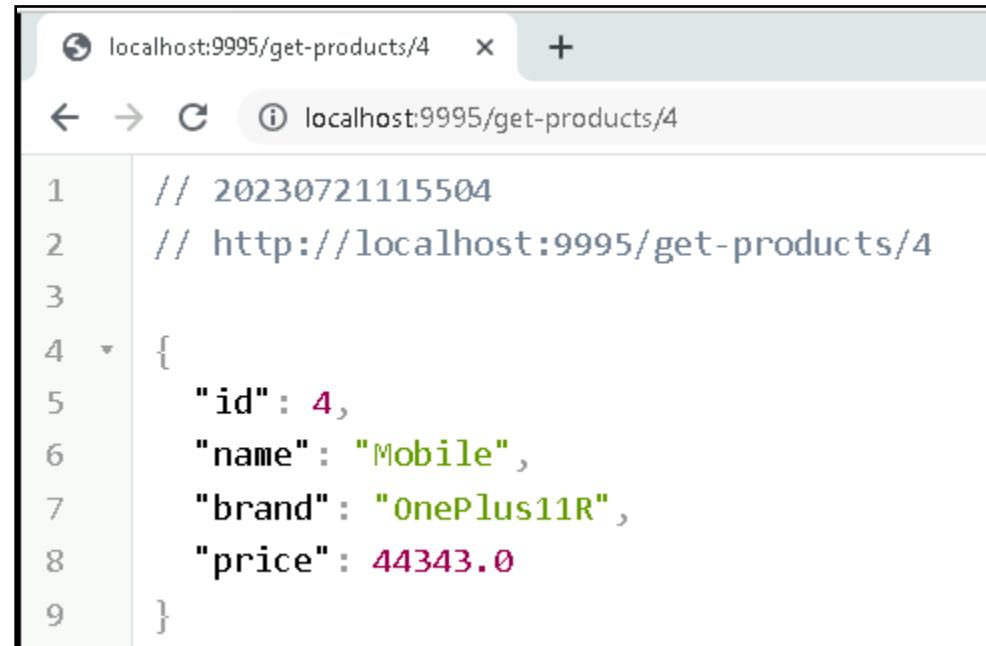
4. Update the ProductClientController class:

```
ProductClientController.java ✘
1 package com.mphasis.controller;
2
3+import org.springframework.beans.factory.annotation.Autowired;□
11
12 @RestController
13 @Scope("request")
14 public class ProductClientController {
15
16@Autowired
17 private ProductService productService;
18
19@GetMapping("/get-products/{id}")
20 public Product getProductById(@PathVariable("id") int id) {
21
22     return productService.getProductById(id);
23 }
24 }
```



Build a circuit breaker with Spring Cloud Hystrix with Ribbon

5. Add **@EnableCircuitBreaker** in the Application main class.
6. Ensure the RabbitMQ docker container, Config Server, Eureka Server and storeapp microservice is running.
7. Test the Consumer Microservices:



A screenshot of a web browser window displaying a JSON response. The URL in the address bar is `localhost:9995/get-products/4`. The response body contains the following JSON data:

```
1 // 20230721115504
2 // http://localhost:9995/get-products/4
3
4 {
5   "id": 4,
6   "name": "Mobile",
7   "brand": "OnePlus11R",
8   "price": 44343.0
9 }
```



Day - 5

Build a Circuit Breaker with Spring Cloud Hystrix with Feign



Build a circuit breaker with Spring Cloud Hystrix with Feign

1. Add the Hystrix dependency to the service.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

2. Change the version - Spring Boot and Spring Cloud:

- Spring Boot: 2.3.10.RELEASE
- Spring Cloud: Hoxton.SR11



Build a circuit breaker with Spring Cloud Hystrix with Feign

3. Add the following property to application.properties file:

```
application.properties ✘
1
2server.port=9994
3eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
4eureka.client.register-with-eureka=false
5
6#Enable Actuator
7management.endpoints.web.exposure.include=*
8
9#Imp
10feign.hystrix.enabled=true
11
```



Build a circuit breaker with Spring Cloud Hystrix with Feign

4. Create a ProductServiceFallback class with **@Component** annotation:

```
ProductServiceFallback.java ✘
1 package com.mphasis.fallback;
2
3 import java.util.ArrayList;
4
5
6 @Component
7 public class ProductServiceFallback implements ProductServiceProxy {
8
9
10    @Override
11    public Product getProductId(Integer id) {
12        return new Product(id, "Monitor", "Jio", 34343.0);
13    }
14
15    @Override
16    public List<Product> getAllProducts() {
17        return new ArrayList<Product>();
18    }
19
20    @Override
21    public void updateProduct(Integer id, String name, String brand, double price) {
22    }
23
24}
```



Build a circuit breaker with Spring Cloud Hystrix with Feign

5. Update the ProductServiceProxy class with **fallback** attribute:

```
ProductServiceProxy.java
1 package com.mphasis.proxy;
2
3+import java.util.List;
12
13 @FeignClient(name = "product-service",
14     fallback = ProductserviceFallback.class)
15 public interface ProductServiceProxy {
16
17@  @GetMapping(value = "/products/{id}", produces = {MediaType.APPLICATION_JSON_VALUE})
18  public Product getProductById(@PathVariable("id") Integer id) ;
19
20@  @GetMapping(value = "/products", produces = {MediaType.APPLICATION_JSON_VALUE})
21  public List<Product> getAllProducts() ;
22 }
23
```



Build a circuit breaker with Spring Cloud Hystrix with Feign

6. Update the ProductClientController class:

```
ProductClientController.java ✘
14 @RestController
15 @Scope("request")
16 public class ProductClientController {
17
18    @Autowired
19    private ProductserviceProxy productServiceProxy;
20
21    @GetMapping("/get-products/{id}")
22    public Product getProductById(@PathVariable("id") int id) {
23
24        Product product = productServiceProxy.getProductById(id);
25        return product;
26    }
27
28    @GetMapping("/get-products")
29    public List<Product> getAllProducts() {
30
31        List<Product> products = productServiceProxy.getAllProducts();
32        return products;
33    }
34 }
```



Build a circuit breaker with Spring Cloud Hystrix with Feign

7. Add **@EnableCircuitBreaker** in the Application main class.
8. Ensure the RabbitMQ docker container, Config Server, Eureka Server and storeapp microservice is running.
9. Test the Consumer Microservices:

A screenshot of a browser window displaying a JSON response. The URL in the address bar is `localhost:9994/get-products/4`. The JSON data is as follows:

```
1 // 20230721120532
2 // http://localhost:9994/get-products/4
3
4 {
5     "id": 4,
6     "name": "Mobile",
7     "brand": "OnePlus11R",
8     "price": 44343.0
9 }
```



Day - 5

Enable Hystrix Dashboard

1. Add the Hystrix, Hystrix Dashboard, and Actuator dependency to the application.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

2. In the Spring Boot Application class, add the **@EnableHystrixDashboard** annotation.
3. Add the below property to application.properties:
`hystrix.dashboard.proxy-stream-allow-list=*`
4. Ensure the RabbitMQ docker container, Config Server, Eureka Server and storeapp microservice is running.
5. Ensure the Consumer (Hystrix Dashboard) application is running.
6. The Hystrix Dashboard is started on application port. (<http://localhost:9994/hystrix>)
7. The Hystrix Stream Endpoint: (<http://localhost:9994/actuator/hystrix.stream>)

Hystrix Dashboard



Hystrix Dashboard

`https://hostname:port/turbine/turbine.stream`

Cluster via Turbine (default cluster): https://turbine-hostname:port/turbine.stream
Cluster via Turbine (custom cluster): https://turbine-hostname:port/turbine.stream?cluster=[clusterName]
Single Hystrix App: https://hystrix-app:port/actuator/hystrix.stream

Delay: ms Title:

Monitor Stream

Hystrix Dashboard



Hystrix Stream: http://localhost:9996/actuator/hystrix.stream



HYSTRIX DEFEND YOUR APP

Circuit Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)
[Success](#) | [Short-Circuited](#) | [Bad Request](#) | [Timeout](#) | [Rejected](#) | [Failure](#) | [Error %](#)

Method	Description	Host: Rate	Cluster: Rate	Circuit Status
Prod...#getProductById(int)	0.0 % 0 0 0.0 % 0 0 0.0 % 0 0 0.0 % Host: 0.0/s Cluster: 0.0/s Circuit Closed	Host: 0.0/s Cluster: 0.0/s Circuit Closed		
Prod...oxy#getAllProducts()	0.0 % 0 0 0.0 % 0 0 0.0 % 0 0 0.0 % Host: 0.0/s Cluster: 0.0/s Circuit Closed	Host: 0.0/s Cluster: 0.0/s Circuit Closed		

Thread Pools Sort: [Alphabetical](#) | [Volume](#) |

Service	Description	Host: Rate	Cluster: Rate
product-service	Host: 0.0/s Cluster: 0.0/s Active: 0 Queued: 0 Pool Size: 10 Max Active: 0 Executions: 0 Queue Size: 5	Host: 0.0/s Cluster: 0.0/s	

Hystrix Dashboard

Hybris Monitor x + localhost:9996/hystrix/monitor?stream=http%3A%2F%2Flocalhost%3A9996%2Factuator%2Fhystrix.stream - □ X

Hybris Stream: http://localhost:9996/actuator/hystrix.stream

Circuit Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)

[Success](#) | [Short-Circuited](#) | [Bad Request](#) | [Timeout](#) | [Rejected](#) | [Failure](#) | [Error %](#)

Prod...oxy#getAllProducts()
Host: 0.5/s
Cluster: 0.5/s
Circuit **Closed**
Hosts 1
Median 90ms
Mean 188ms
90th 1001ms
99th 1015ms
99.5th 1015ms

Prod...#getProductById(int)
Host: 0.0/s
Cluster: 0.0/s
Circuit **Closed**
Hosts 1
Median 9ms
Mean 9ms
90th 9ms
99th 9ms
99.5th 9ms

Thread Pools

Sort: [Alphabetical](#) | [Volume](#) |

product-service			
	Host: 0.5/s	Cluster: 0.5/s	
Active 0	Max Active 1	Executions 5	
Queued 0	Queue Size 5		
Pool Size 10			



Recap of Day – 5

- Isolating from failures
- Spring Cloud Hystrix for fault-tolerant microservices



Day - 6



Day – 6 Agenda

- What is API Gateway?
- Spring cloud routing – Zuul
- Setting up Zuul

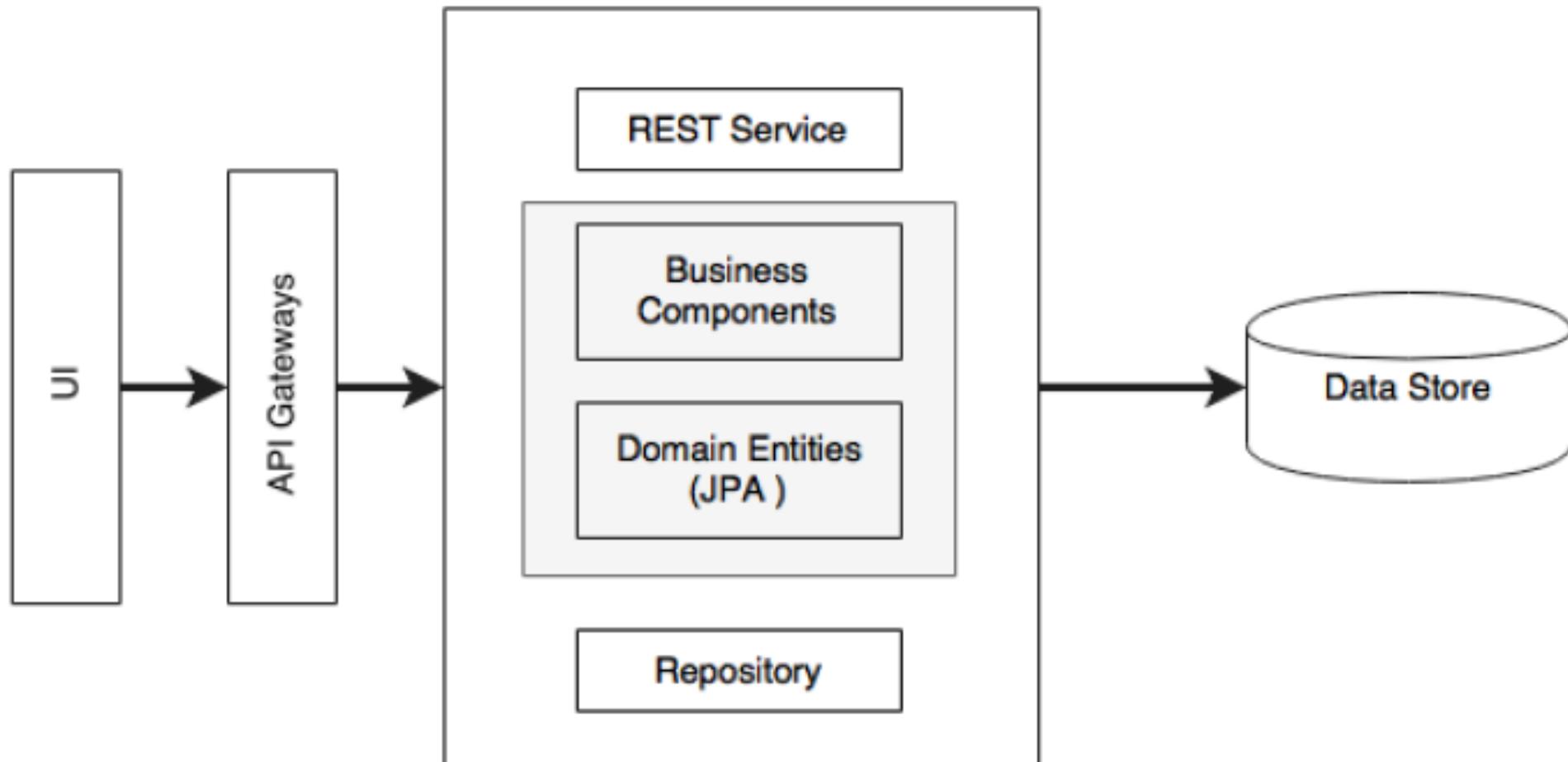


Day - 6

Spring cloud routing - Zuul

- An **API Gateways**, aka **Edge Service**, provides a unified interface for a set of microservices so that client no need to know about all the details of microservices internals.
- The API gateway provides a level of indirection by either proxying service endpoints or composing multiple service endpoints.
- The API gateway is also useful for policy enforcements.
- It may also provide real time load balancing capabilities.
- There are many API gateways available in the market.
- Spring Cloud Zuul, Mashery, Apigee, and 3scale are some examples of the API gateway providers.

API Gateway





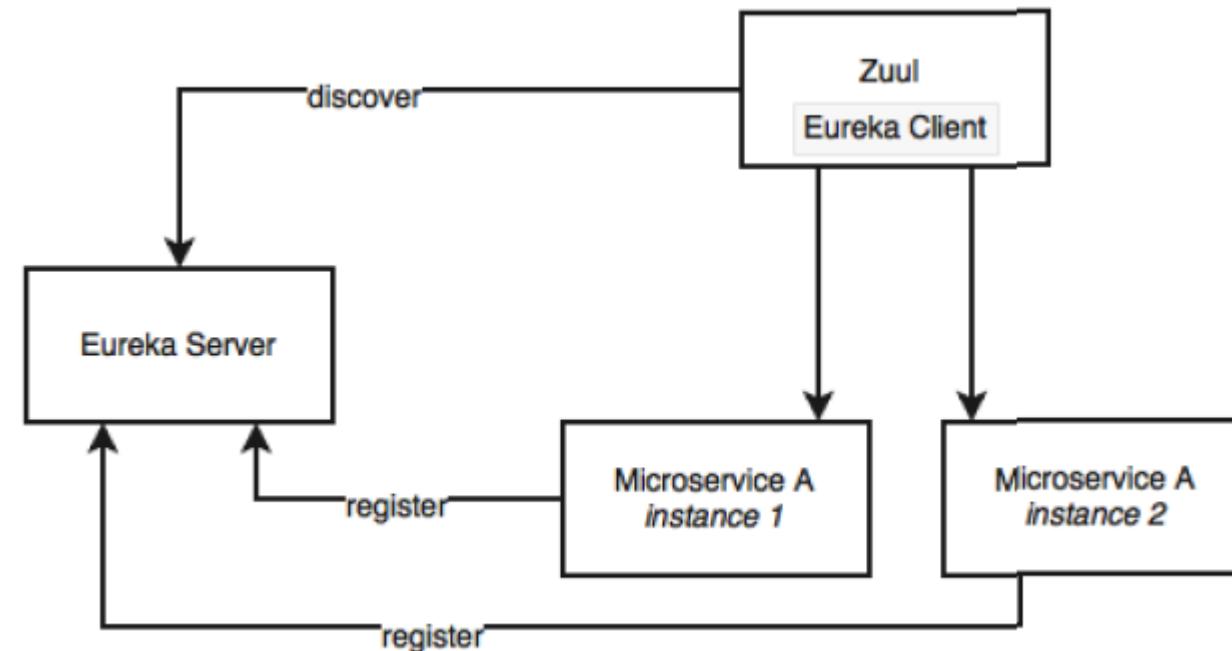
Spring Cloud Routing

- Routing is an API gateway component, primarily used like a reverse proxy that forwards requests from consumers to service providers.
- The gateway component can also perform software-based routing and filtering.
- Zuul is a lightweight API gateway solution that offers fine-grained controls to developers for traffic shaping and request/response transformations.



Zuul proxy as the API Gateway

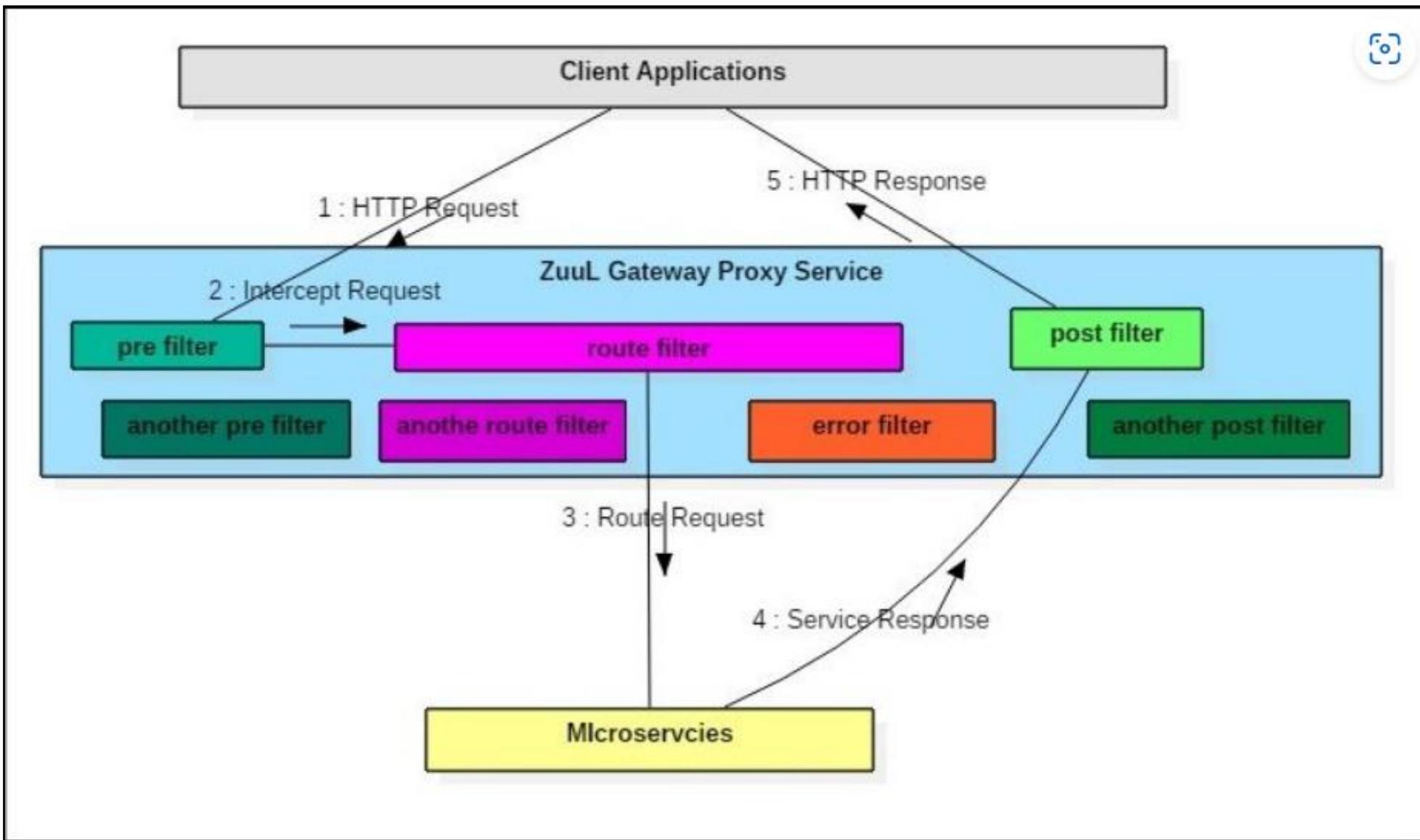
- Zuul is a simple gateway service or edge service that suits these situations well.
- Zuul also comes from the Netflix family of microservice products.
- Unlike many enterprise API gateway products, Zuul provides complete control for the developers to configure, or program based on specific requirements:



- The Zuul proxy internally uses the Eureka server for service discovery, and Ribbon for load balancing between service instances.
- The Zuul proxy is also capable of routing, monitoring, managing resiliency, security, and so on.
- In simple terms, we can consider Zuul a reverse proxy service. With Zuul, we can even change the behaviors of the underlying services by overriding them at the API layer.

- Zuul has mainly four types of filters that enable us to intercept the traffic in different timeline of the request processing for any particular transaction.
- We can add any number of filters for a URL pattern.
 - **pre filters** – are invoked before the request is routed.
 - **post filters** – are invoked after the request has been routed.
 - **route filters** – are used to route the request.
 - **error filters** – are invoked when an error occurs while handling the request.

Zuul Components



1. Add the Zuul dependency to the application

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
```

2. Change the version - Spring Boot and Spring Cloud:

- Spring Boot: 2.3.10.RELEASE
- Spring Cloud: Hoxton.SR11

3. In the Spring Boot Application class, add the **@EnableZuulProxy** annotation.
4. Create the different types of custom filters extending **ZuulFilter**:
 - PreFilter
 - PostFilter
 - RouteFilter
 - ErrorFilter
5. Register the Filters.

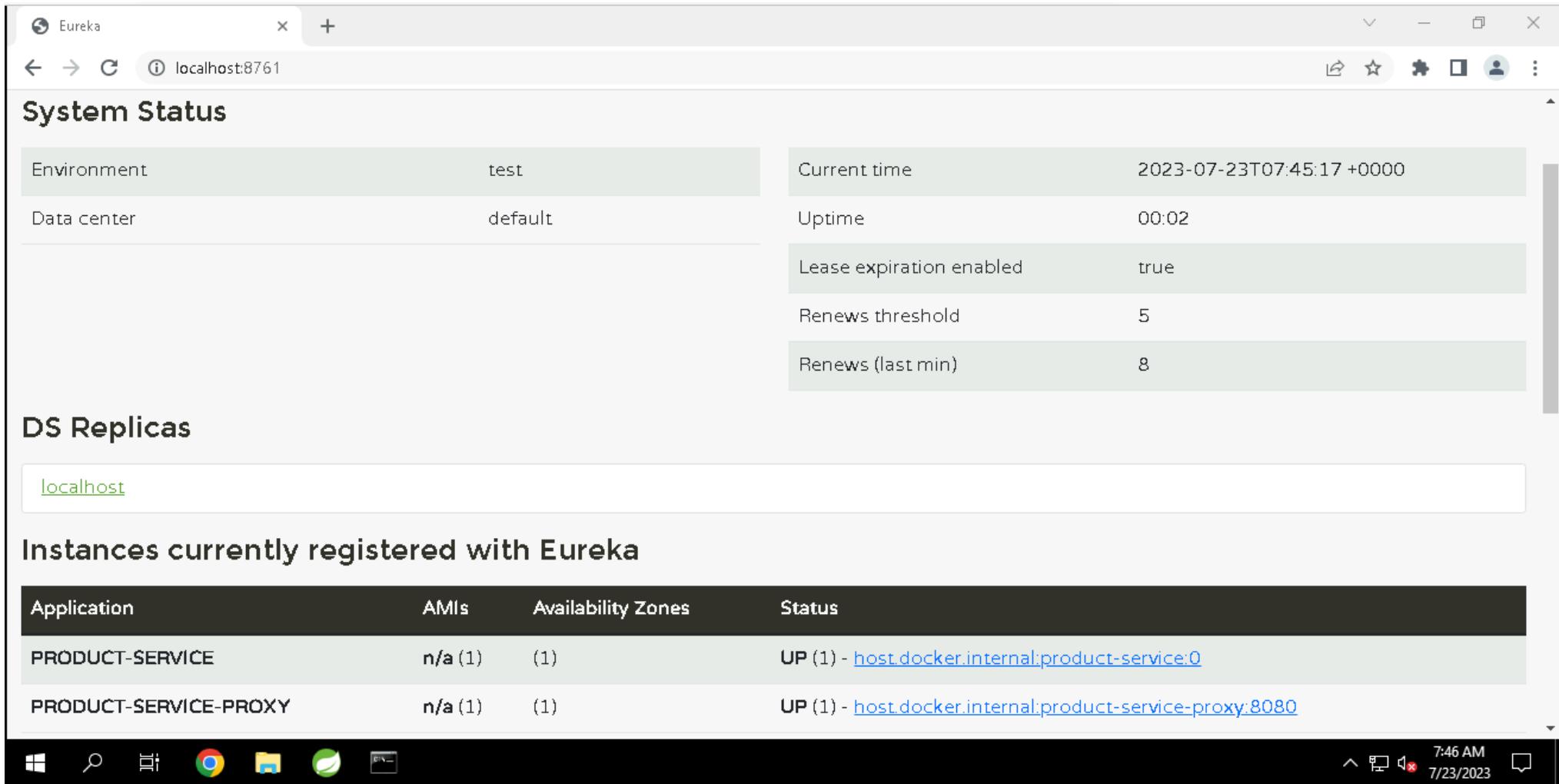
6. This configuration also sets a rule on how to forward traffic. In this case, any request coming on the **/api** endpoint of the API gateway should be sent to **product-service** which is registered inside the **Eureka Server**:

```
application.properties ✘  
1  
2 #will start the API Gateway Server @8080  
3 server.port=8080  
4 spring.application.name=product-service-proxy  
5  
6 #zuul routes  
7 zuul.routes.products.service-id=product-service  
8 zuul.routes.products.path=/api/**  
9  
10 #Eureka Server Details  
11 eureka.client.service-url.defaultZone=http://localhost:8761/eureka  
12 eureka.client.register-with-eureka=true  
13
```



Setting up Zuul

6. Check the product-service-proxy is registered inside the Eureka Server.



The screenshot shows a web browser window displaying the Eureka system status and registered instances.

System Status

Environment	test	Current time	2023-07-23T07:45:17 +0000
Data center	default	Uptime	00:02
		Lease expiration enabled	true
		Renews threshold	5
		Renews (last min)	8

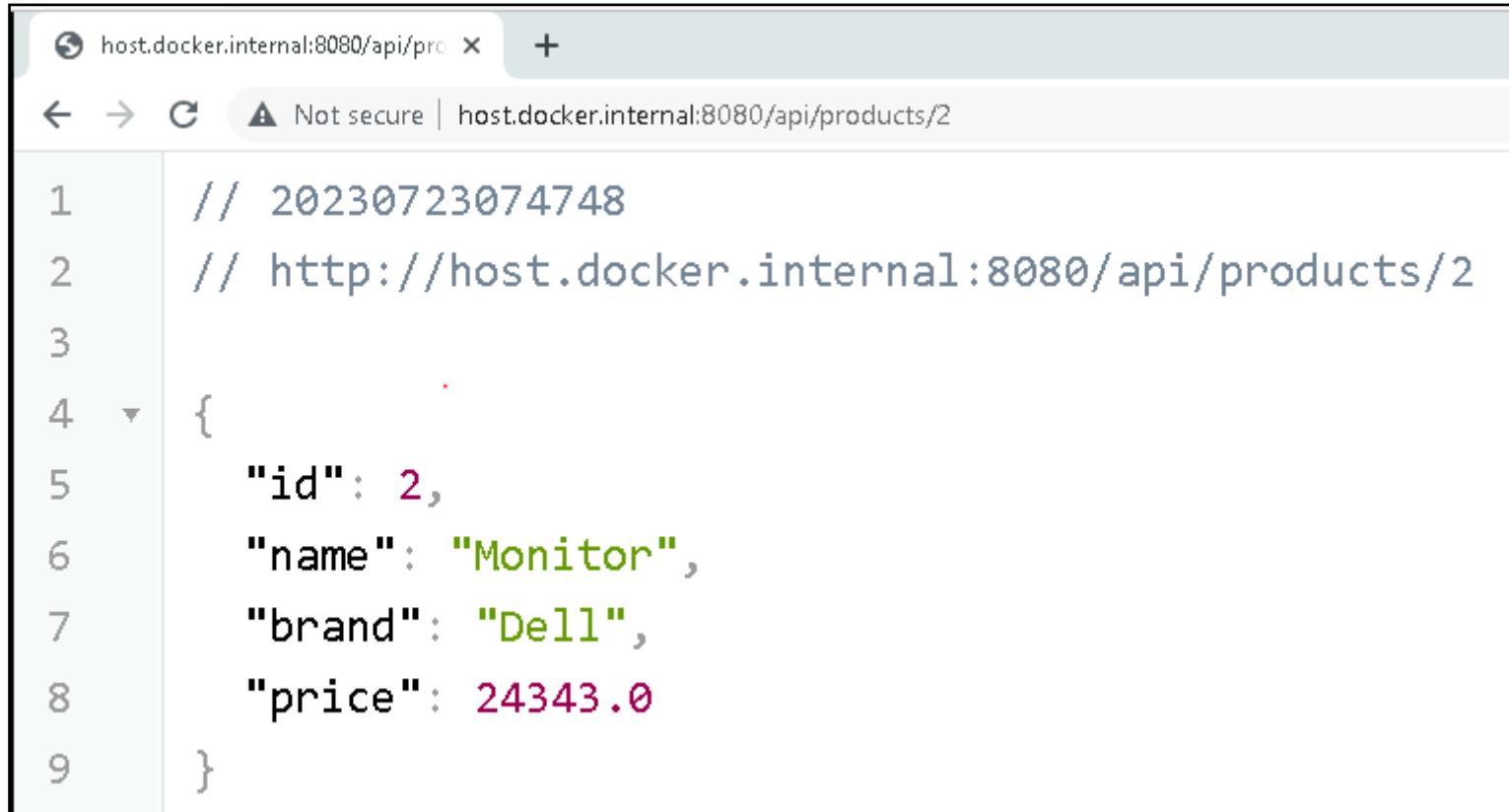
DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
PRODUCT-SERVICE	n/a (1)	(1)	UP (1) - host.docker.internal:product-service:0
PRODUCT-SERVICE-PROXY	n/a (1)	(1)	UP (1) - host.docker.internal:product-service-proxy:8080

6. Access the storeapp via Proxy.



A screenshot of a web browser window. The address bar shows "host.docker.internal:8080/api/pro". Below the address bar, it says "Not secure | host.docker.internal:8080/api/products/2". The main content area displays the following JSON response:

```
1 // 20230723074748
2 // http://host.docker.internal:8080/api/products/2
3
4 {
5     "id": 2,
6     "name": "Monitor",
7     "brand": "Dell",
8     "price": 24343.0
9 }
```



Recap of Day – 6

- What is API Gateway?
- Spring cloud routing – Zuul
- Setting up Zuul



Day - 7

- Replacement capabilities available in new Spring Cloud version
- Adding Resilience4j
- Actuator /health Endpoint
- RestTemplate client & Circuit Breaker Fallback method
- Feign Client & Circuit Breaker Fallback method
- Circuit Breaker configuration properties
- Configure Access to Actuator endpoints
- Monitoring Circuit Breaker events in Actuator
- Retry with Spring Cloud Resilience4j



Day - 7

Implementing Circuit Breaker using Resilience4J



Replacement capabilities available in new Spring Cloud version

Current (Placed into maintenance mode)	Replacement
Spring Cloud Netflix Hystrix	Resilience4j
Spring Cloud Netflix Hystrix Dashboard	Prometheus and Micrometer
Spring Cloud Netflix Zuul	Spring Cloud Gateway with Load Balancer
Spring Cloud Config	No replacement
Spring Cloud Netflix Eureka	No replacement
Spring Cloud Routing - Ribbon	No replacement
Spring Cloud Routing - OpenFeign	No replacement
Spring Cloud Sleuth and Zipkin	No replacement



Spring Cloud Resilience4j - Circuit Breaker

- Resilience4j is a lightweight fault tolerance library designed for functional programming.
- Resilience4j provides higher-order functions (decorators) to enhance any functional interface, lambda expression or method reference with a Circuit Breaker, Rate Limiter, Retry or Bulkhead.
- You can stack more than one decorator on any functional interface, lambda expression or method reference.
- The advantage is that you have the choice to select the decorators you need and nothing else.



Comparison to Netflix Hystrix

- In Hystrix calls to external systems must be wrapped in a HystrixCommand. This library, in contrast, provides higher-order functions (decorators) to enhance any functional interface, lambda expression or method reference with a Circuit Breaker, Rate Limiter or Bulkhead. Furthermore, the library provides decorators to retry failed calls or cache call results. You can stack more than one decorator on any functional interface, lambda expression or method reference. That means, you can combine a Bulkhead, RateLimiter and Retry decorator with a CircuitBreaker decorator. The advantage is that you have the choice to select the decorator you need and nothing else. Any decorated function can be executed synchronously or asynchronously by using a CompletableFuture or RxJava.
- The CircuitBreaker can open when too many calls exceed a certain response time threshold, even before the remote system is unresponsive and exceptions are thrown.



Comparison to Netflix Hystrix

- Hystrix only performs a single execution when in half-open state to determine whether to close a CircuitBreaker. This library allows to perform a configurable number of executions and compares the result against a configurable threshold to determine whether to close a CircuitBreaker.
- This library provides custom Reactor or RxJava operators to decorate any reactive type with a Circuit Breaker, Bulkhead or Ratelimiter.
- Hystrix and this library emit a stream of events which are useful to system operators to monitor metrics about execution outcomes and latency.



Core Modules

Resilience4j has the following 6 core modules:

- resilience4j-circuitbreaker: Circuit breaking
- resilience4j-retry: Automatic retrying (sync and async)
- resilience4j-ratelimiter: Rate limiting
- resilience4j-timelimiter: Timeout handling
- resilience4j-bulkhead: Bulkheading
- resilience4j-cache: Result caching



Resilience patterns

Name	How does it work?	Description
Circuit Breaker	temporarily blocks possible failures	When a system is seriously struggling, failing fast is better than making clients wait.
Retry	repeats failed executions	Many faults are transient and may self-correct after a short delay.
Rate Limiter	limits executions/period	Limit the rate of incoming requests.
Time Limiter	limits duration of execution	Beyond a certain wait interval, a successful result is unlikely.
Bulkhead	limits concurrent executions	Resources are isolated into pools so that if one fails, the others will continue working.
Cache	memorizes a successful result	Some proportion of requests may be similar.



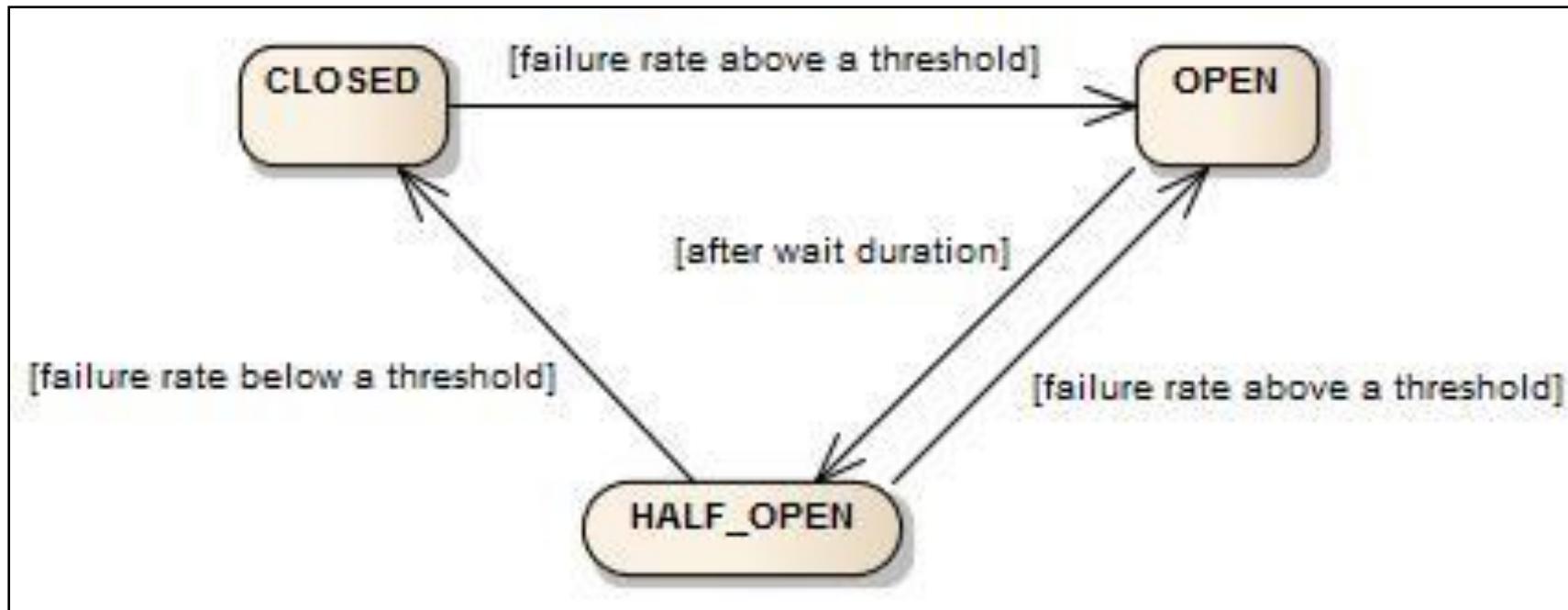
What is Circuit Breaker?

- The circuit breaker is essentially a pattern that helps to prevent cascading failures in a system.
- The circuit breaker pattern allows you to build a fault-tolerant and resilient system that can survive gracefully when key services are either unavailable or have high latency.
- Circuit breaker pattern is generally used in microservices architecture where there are multiple services involved but it can be used otherwise as well.

The circuit breaker has the following 3 states:

- **Closed** — Closed is when everything is normal, in the beginning, it will be in the closed state and if failures exceed the threshold value decided at the time of creating circuit breaker, the circuit will trip and go into an open state.
- **Open** — Open is the state when the calls start to fail without any latency i.e., calls will start to fail fast without even executing the function calls.
- **Half-open** — In half-open state what will happen is, the very first call will not fail fast, and all other calls will fail fast just as in the open state. If the first call succeeds then the circuit will go in the closed state again and otherwise, it will go into the open state again waiting for the reset timeout.

Circuit Breaker - 3 States



Source: <https://resilience4j.readme.io/docs/circuitbreaker>



Day - 7

Implementing Circuit Breaker using Resilience4J on Ribbon Consumer



Implementing Circuit Breaker using Resilience4J on Ribbon Consumer

1. Add the Resilience4j, AOP, and Actuator starter to the pom.xml of Ribbon - Consumer application:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

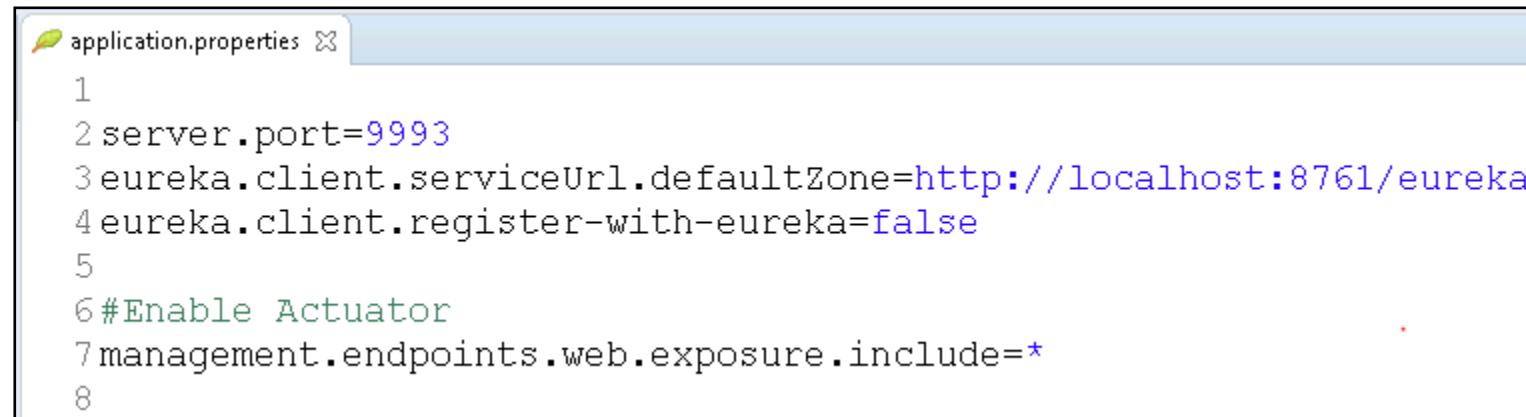
2. Change the version - Spring Boot and Spring Cloud:

- Spring Boot: 2.7.13
- Spring Cloud: 2021.0.8



Implementing Circuit Breaker using Resilience4J on Ribbon Consumer

3. Add the following property to application.properties file:



```
application.properties
1
2 server.port=9993
3 eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
4 eureka.client.register-with-eureka=false
5
6 #Enable Actuator
7 management.endpoints.web.exposure.include=*
8
```



Implementing Circuit Breaker using Resilience4J on Ribbon Consumer

4. Create a ProductService class with **@CircuitBreaker** annotation:

```
ProductService.java
10
11 @Service
12 public class ProductService {
13
14     @Autowired
15     private RestTemplate restTemplate;
16
17     @CircuitBreaker(name="product-service", fallbackMethod = "fallbackMethodForGetProductById")
18     public Product getProductId(int id) {
19
20         Product product = restTemplate.getForObject("http://product-service/products/" + id,
21             Product.class);
22         return product;
23     }
24
25     public Product fallbackMethodForGetProductById(int id, Exception ex) {
26
27         return new Product(id, "Monitor", "Jio", 34343.0);
28     }
29 }
30
```



Implementing Circuit Breaker using Resilience4J on Ribbon Consumer

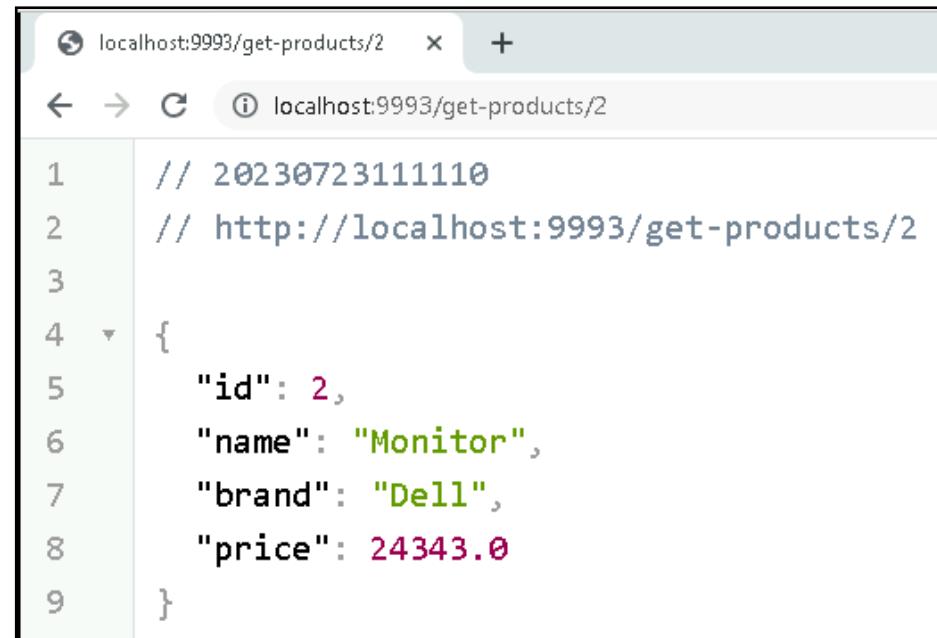
5. Update the ProductClientController class:

```
ProductClientController.java ✘
1 package com.mphasis.controller;
2
3+import org.springframework.beans.factory.annotation.Autowired;□
11
12 @RestController
13 @Scope("request")
14 public class ProductClientController {
15
16@Autowired
17 private ProductService productService;
18
19@GetMapping("/get-products/{id}")
20 public Product getProductById(@PathVariable("id") int id) {
21
22     return productService.getProductById(id);
23 }
24 }
```



Implementing Circuit Breaker using Resilience4J on Ribbon Consumer

6. Remove **@EnableCircuitBreaker** from the Application main class.
7. Ensure the RabbitMQ docker container, Config Server, Eureka Server and storeapp microservice is running.
8. Test the Consumer Microservices:



A screenshot of a web browser window displaying a JSON response. The URL in the address bar is `localhost:9993/get-products/2`. The content of the page is a JSON object representing a product:

```
1 // 20230723111110
2 // http://localhost:9993/get-products/2
3
4 {
5     "id": 2,
6     "name": "Monitor",
7     "brand": "Dell",
8     "price": 24343.0
9 }
```



Day - 7

Implementing Circuit Breaker using Resilience4J on Feign Consumer



Implementing Circuit Breaker using Resilience4J on Feign Consumer

1. Add the Resilience4j, AOP, and Actuator starter to the pom.xml of Ribbon - Consumer application:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

2. Change the version - Spring Boot and Spring Cloud:

- Spring Boot: 2.7.13
- Spring Cloud: 2021.0.8



Implementing Circuit Breaker using Resilience4J on Feign Consumer

3. Add the **feign.circuitbreaker.enabled** property to application.properties file:

```
application.properties ✘  
1  
2 server.port=9992  
3 eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka  
4 eureka.client.register-with-eureka=false  
5  
6 #Enable Actuator  
7 management.endpoints.web.exposure.include=*<br/>  
8  
9 #Imp  
10 feign.circuitbreaker.enabled=true  
11
```



Implementing Circuit Breaker using Resilience4J on Feign Consumer

4. Create a ProductServiceFallback class with **@Component** annotation:

```
ProductServiceFallback.java ✘
1 package com.mphasis.fallback;
2
3 import java.util.ArrayList;
4
5
6 @Component
7 public class ProductServiceFallback implements ProductServiceProxy {
8
9
10    @Override
11    public Product getProductId(Integer id) {
12        return new Product(id, "Monitor", "Jio", 34343.0);
13    }
14
15    @Override
16    public List<Product> getAllProducts() {
17        return new ArrayList<Product>();
18    }
19
20    @Override
21    public void updateProduct(Integer id, String name, String brand, double price) {
22    }
23
24}
```



Implementing Circuit Breaker using Resilience4J on Feign Consumer

5. Update the ProductServiceProxy class with **fallback** attribute:

```
ProductServiceProxy.java
1 package com.mphasis.proxy;
2
3+import java.util.List;
12
13 @FeignClient(name = "product-service",
14     fallback = ProductserviceFallback.class)
15 public interface ProductServiceProxy {
16
17@  @GetMapping(value = "/products/{id}", produces = {MediaType.APPLICATION_JSON_VALUE})
18  public Product getProductById(@PathVariable("id") Integer id) ;
19
20@  @GetMapping(value = "/products", produces = {MediaType.APPLICATION_JSON_VALUE})
21  public List<Product> getAllProducts() ;
22 }
23
```



Implementing Circuit Breaker using Resilience4J on Feign Consumer

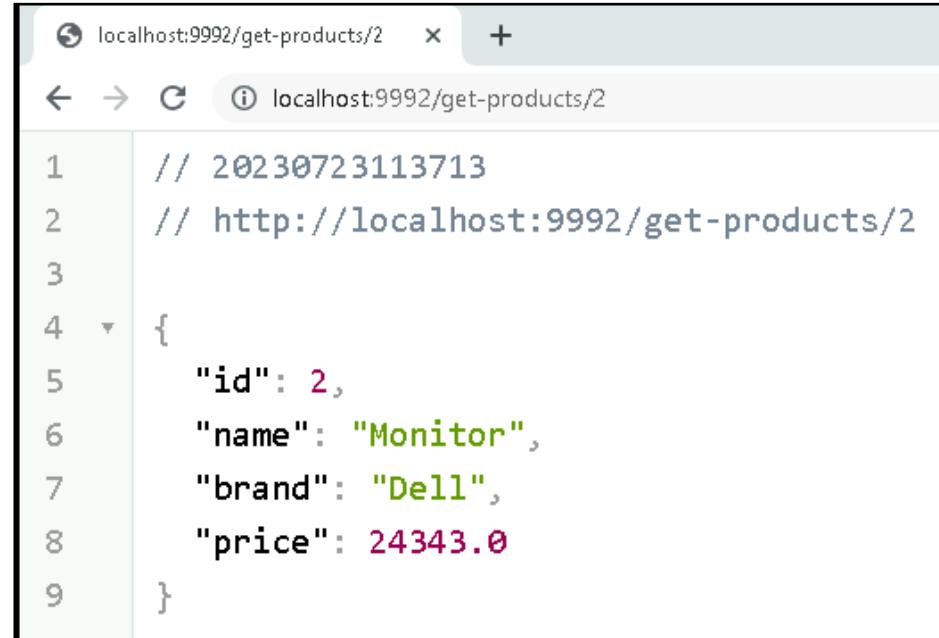
6. Update the ProductClientController class:

```
ProductClientController.java ✘
14 @RestController
15 @Scope("request")
16 public class ProductClientController {
17
18     @Autowired
19     private ProductserviceProxy productServiceProxy;
20
21     @GetMapping("/get-products/{id}")
22     public Product getProductById(@PathVariable("id") int id) {
23
24         Product product = productServiceProxy.getProductById(id);
25         return product;
26     }
27
28     @GetMapping("/get-products")
29     public List<Product> getAllProducts() {
30
31         List<Product> products = productServiceProxy.getAllProducts();
32         return products;
33     }
34 }
```



Implementing Circuit Breaker using Resilience4J on Feign Consumer

7. Add **@EnableFeignClients** in the Application main class.
6. Remove **@EnableCircuitBreaker** from the Application main class.
7. Ensure the RabbitMQ docker container, Config Server, Eureka Server and storeapp microservice is running.
8. Test the Consumer Microservices:



The screenshot shows a browser window with the URL `localhost:9992/get-products/2`. The page displays a JSON object representing a product:

```
// 20230723113713
// http://localhost:9992/get-products/2
{
  "id": 2,
  "name": "Monitor",
  "brand": "Dell",
  "price": 24343.0
}
```



Day - 7

Circuit Breaker Configuration Properties

- The CircuitBreaker uses a sliding window to store and aggregate the outcome of calls. You can choose between a count-based sliding window and a time-based sliding window.
- The **count-based** sliding window aggregates the outcome of the last N calls.
- The **time-based** sliding window aggregates the outcome of the calls of the last N seconds.

```
resilience4j.circuitbreaker.instances.product-service.sliding-window-type=time-based
```

```
resilience4j.circuitbreaker.instances.product-service.sliding-window-type=count-based
```



Circuit Breaker Configuration Properties

Property	Description
management.endpoint.health.show-details	Controls the level of details each health endpoint can expose.
management.health.circuitbreakers.enabled	By default the CircuitBreaker health indicators are disabled, but you can enable them via the configuration.
management.health.ratelimiters.enabled	By default the RateLimiter health indicators are disabled, but you can enable them via the configuration.



Circuit Breaker Configuration Properties

Config property	Default Value	Description
slidingWindowType	COUNT_BASED	Configures the type of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed. Sliding window can either be count-based or time-based.
slidingWindowSize	100	Configures the size of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed.
minimumNumberOfCalls	100	Configures the minimum number of calls which are required (per sliding window period) before the CircuitBreaker can calculate the error rate or slow call rate.
eventConsumerBufferSize	100	Configures the size of the buffers having events emitted by the Circuit Breaker, Retry, Rate Limiter, Bulkhead, and Time Limiter instances are stored separately in circular event consumer buffers.
failureRateThreshold	50	Configures the failure rate threshold in percentage.



Circuit Breaker Configuration Properties

automaticTransitionFromOpenToHalfOpenEnabled	FALSE	If set to true it means that the CircuitBreaker will automatically transition from open to half-open state and no call is needed to trigger the transition. A thread is created to monitor all the instances of CircuitBreakers to transition them to HALF_OPEN once waitDurationInOpenState passes. Whereas, if set to false the transition to HALF_OPEN only happens if a call is made, even after waitDurationInOpenState is passed. The advantage here is no thread monitors the state of all CircuitBreakers.
waitDurationInOpenState	60000 [ms]	The time that the CircuitBreaker should wait before transitioning from open to half-open.



Implementing Circuit Breaker using Resilience4J on Ribbon Consumer

1. Add the following property to application.properties file:

The screenshot shows a code editor window with the title bar "application.properties". The file contains Java-like configuration properties. Lines 1 through 21 are shown, with line 21 being the active input line.

```
1
2server.port=9993
3eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
4eureka.client.register-with-eureka=false
5
6#Enable Actuator
7management.endpoints.web.exposure.include=*
8management.endpoint.health.show-details=always
9management.health.circuitbreakers.enabled=true
10management.health.ratelimiters.enabled=true
11
12#resilience4j.circuitbreaker.instances.product-service.sliding-window-type=time-based
13resilience4j.circuitbreaker.instances.product-service.sliding-window-type=count-based
14resilience4j.circuitbreaker.instances.product-service.sliding-window-size=2
15resilience4j.circuitbreaker.instances.product-service.minimum-number-of-calls=1
16resilience4j.circuitbreaker.instances.product-service.event-consumer-buffer-size=10
17resilience4j.circuitbreaker.instances.product-service.failure-rate-threshold=50
18resilience4j.circuitbreaker.instances.product-service.automatic-transition-from-open-to-half-open-enabled=true
19resilience4j.circuitbreaker.instances.product-service.wait-duration-in-open-state=10s
20
21|
```



Implementing Circuit Breaker using Resilience4J on Feign Consumer

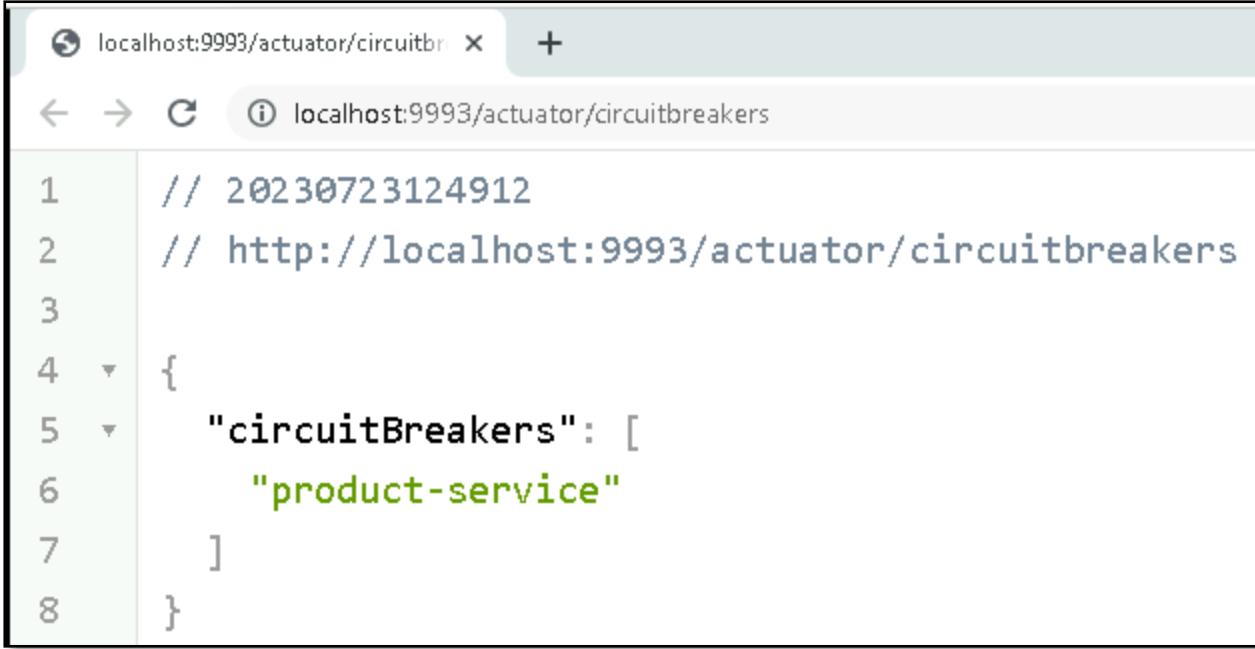
1. Add the following property to application.properties file:

```
application.properties
1
2server.port=9992
3eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
4eureka.client.register-with-eureka=false
5
6#Imp
7feign.circuitbreaker.enabled=true
8
9#Enable Actuator
10management.endpoints.web.exposure.include=*
11management.endpoint.health.show-details=always
12management.health.circuitbreakers.enabled=true
13management.health.ratelimiters.enabled=true
14
15#resilience4j.circuitbreaker.instances.product-service.sliding-window-type=time-based
16resilience4j.circuitbreaker.instances.product-service.sliding-window-type=count-based
17resilience4j.circuitbreaker.instances.product-service.sliding-window-size=2
18resilience4j.circuitbreaker.instances.product-service.minimum-number-of-calls=1
19resilience4j.circuitbreaker.instances.product-service.event-consumer-buffer-size=10
20resilience4j.circuitbreaker.instances.product-service.failure-rate-threshold=50
21resilience4j.circuitbreaker.instances.product-service.automatic-transition-from-open-to-half-open-enabled=true
22resilience4j.circuitbreaker.instances.product-service.wait-duration-in-open-state=10s
23|
```



Check the Circuit Breaker List

- Check the Circuit Breaker List : (<http://localhost:9993/actuator/circuitbreakers>)



The screenshot shows a browser window with the URL `localhost:9993/actuator/circuitbreakers`. The page displays a JSON object with the following structure:

```
1 // 20230723124912
2 // http://localhost:9993/actuator/circuitbreakers
3
4 {
5   "circuitBreakers": [
6     "product-service"
7   ]
8 }
```



Check the Circuit Breaker Health

- Check the Circuit Breaker Health: (<http://localhost:9993/actuator/health>)



A screenshot of a browser window displaying a JSON response. The title bar says "localhost:9993/actuator/health". The URL in the address bar is "localhost:9993/actuator/health". The content area shows the following JSON code:

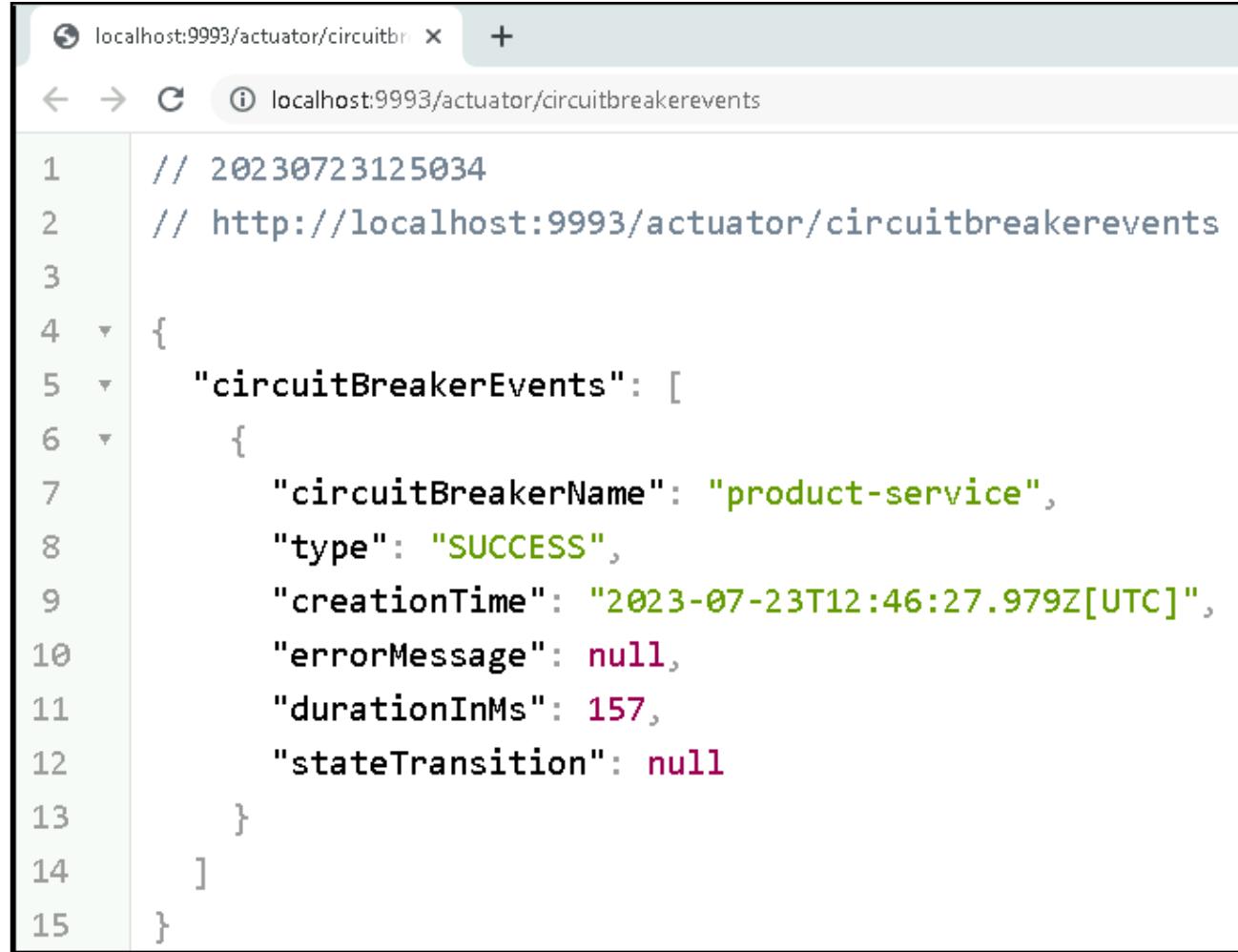
```
3
4  {
5      "status": "UP",
6      "components": {
7          "circuitBreakers": {
8              "status": "UNKNOWN"
9          },

```



Check the Circuit Breaker Events

- Check the Circuit Breaker Events : (<http://localhost:9993/actuator/circuitbreakerevents>)



```
// 20230723125034
// http://localhost:9993/actuator/circuitbreakerevents
{
  "circuitBreakerEvents": [
    {
      "circuitBreakerName": "product-service",
      "type": "SUCCESS",
      "creationTime": "2023-07-23T12:46:27.979Z[UTC]",
      "errorMessage": null,
      "durationInMs": 157,
      "stateTransition": null
    }
  ]
}
```



Day - 7

Retry with Spring Cloud Resilience4j



Retry Module

- **@Retry(name="product-service") Annotation**
- More Info: <https://resilience4j.readme.io/docs/retry>
- Retry decorator will help you to retry the failed decoration.
- If an error that takes place tells you that retrying can help you can repeat the same decoration otherwise use fallback method to return the predefined response.
- You can also configure how many times to retry and how long to wait before retry again.



Retry Configuration Properties

Config property	Default value	Description
maxAttempts	3	The maximum number of attempts (including the initial call as the first attempt)
waitDuration	500 [ms]	A fixed wait duration between retry attempts
enableExponentialBackoff	FALSE	The exponential backoff strategy progressively increases the wait duration between retry attempts, providing a controlled and resilient approach to handling failures.
exponentialBackoffMultiplier	0	The wait time increases exponentially between attempts because of the multiplier.

- The Resilience4j Aspects order is the following:

```
Retry ( CircuitBreaker ( RateLimiter ( TimeLimiter ( Bulkhead ( Function ) ) ) ) )
```

- So, Retry is applied at the end (if needed).
- If you need a different order, you must use the functional chaining style instead of the Spring annotations style or explicitly set aspect order using the following properties:

- resilience4j.retry.retryAspectOrder
- resilience4j.circuitbreaker.circuitBreakerAspectOrder
- resilience4j.ratelimiter.rateLimiterAspectOrder
- resilience4j.timelimiter.timeLimiterAspectOrder
- resilience4j.bulkhead.bulkheadAspectOrder



Retry with Spring Cloud Resilience4j on Ribbon Consumer

1. Add the following property to application.properties file:

```
application.properties

11
12 resilience4j.circuitbreaker.circuit-breaker-aspect-order=1
13
14#resilience4j.circuitbreaker.instances.product-service.sliding-window-type=time-based
15 resilience4j.circuitbreaker.instances.product-service.sliding-window-type=count-based
16 resilience4j.circuitbreaker.instances.product-service.sliding-window-size=2
17 resilience4j.circuitbreaker.instances.product-service.minimum-number-of-calls=1
18 resilience4j.circuitbreaker.instances.product-service.event-consumer-buffer-size=10
19 resilience4j.circuitbreaker.instances.product-service.failure-rate-threshold=50
20 resilience4j.circuitbreaker.instances.product-service.automatic-transition-from-open-to-half-open-enabled=true
21 resilience4j.circuitbreaker.instances.product-service.wait-duration-in-open-state=10s
22
23 resilience4j.retry.retry-aspect-order=2
24
25 resilience4j.retry.instances.product-service.max-attempts=3
26 resilience4j.retry.instances.product-service.wait-duration=2s
27
28 resilience4j.retry.instances.product-service.enable-exponential-backoff=true
29 resilience4j.retry.instances.product-service.exponential-backoff-multiplier=5
30
31
32
33
```



Retry with Spring Cloud Resiliance4j on Ribbon Consumer

2. Add the **@Retry** annotation to the Product Service class.

```
 ProductService.java
11
12 @Service
13 public class ProductService {
14
15     @Autowired
16     private RestTemplate restTemplate;
17
18     @Retry(name = "product-service")
19     @CircuitBreaker(name="product-service", fallbackMethod = "fallbackMethodForGetProductById")
20     public Product getProductId(int id) {
21
22         Product product = restTemplate.getForObject("http://product-service/products/" + id,
23             Product.class);
24         return product;
25     }
26
27     public Product fallbackMethodForGetProductById(int id, Exception ex) {
28
29         System.out.println("Exception raised with message :====>" + ex.getMessage());
30         return new Product(id, "Monitor", "Jio", 34343.0);
31     }
32 }
33
```



Retry with Spring Cloud Resilience4j on Feign Consumer

1. Add the following property to application.properties file:

```
application.properties
10management.endpoints.web.exposure.include=*
11management.endpoint.health.show-details=always
12management.health.circuitbreakers.enabled=true
13management.health.ratelimiters.enabled=true
14
15resilience4j.circuitbreaker.circuit-breaker-aspect-order=1
16
17#resilience4j.circuitbreaker.instances.product-service.sliding-window-type=time-based
18resilience4j.circuitbreaker.instances.product-service.sliding-window-type=count-based
19resilience4j.circuitbreaker.instances.product-service.sliding-window-size=2
20resilience4j.circuitbreaker.instances.product-service.minimum-number-of-calls=1
21resilience4j.circuitbreaker.instances.product-service.event-consumer-buffer-size=10
22resilience4j.circuitbreaker.instances.product-service.failure-rate-threshold=50
23resilience4j.circuitbreaker.instances.product-service.automatic-transition-from-open-to-half-open-enabled=true
24resilience4j.circuitbreaker.instances.product-service.wait-duration-in-open-state=10s
25
26resilience4j.retry.retry-aspect-order=2
27
28resilience4j.retry.instances.product-service.max-attempts=3
29resilience4j.retry.instances.product-service.wait-duration=2s
30
31resilience4j.retry.instances.product-service.enable-exponential-backoff=true
32resilience4j.retry.instances.product-service.exponential-backoff-multiplier=5
--<-->
```



Retry with Spring Cloud Resiliency4j on Feign Consumer

2. Add the **@CircuitBreaker** annotation to the ProductServiceProxy. If you want to find the state-transition in /actuator/circuitbreakerevents do the below changes.
3. Add the **@Retry** annotation to the ProductServiceProxy interface.



Retry with Spring Cloud Resiliency4j on Feign Consumer

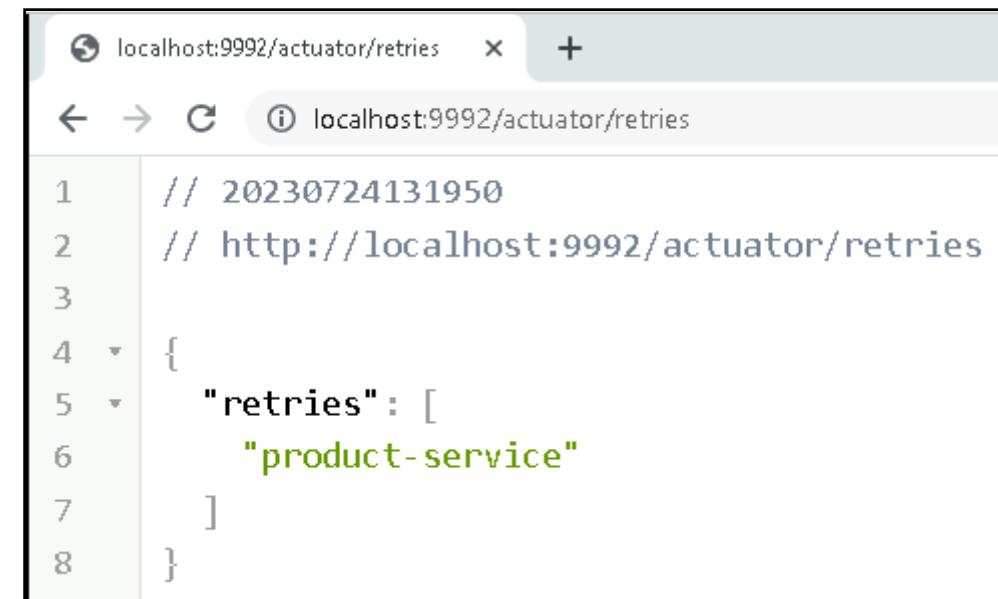
ProductServiceProxy.java

```
16 @FeignClient(name = "product-service")
17 public interface ProductServiceProxy {
18
19     @Retry(name = "product-service")
20     @CircuitBreaker(name = "product-service", fallbackMethod = "fallbackMethodGetProductById")
21     @GetMapping(value = "/products/{id}", produces = { MediaType.APPLICATION_JSON_VALUE })
22     public Product getProductById(@PathVariable("id") Integer id);
23
24     @Retry(name = "product-service")
25     @CircuitBreaker(name = "product-service", fallbackMethod = "fallbackMethodGetAllProducts")
26     @GetMapping(value = "/products", produces = { MediaType.APPLICATION_JSON_VALUE })
27     public List<Product> getAllProducts();
28
29     public default Product fallbackMethodGetProductById(Integer id, Throwable cause) {
30         System.out.println("Exception raised with message :===>" + cause.getMessage());
31         return new Product(id, "Monitor", "Jio", 34343.0);
32     }
33
34     public default List<Product> fallbackMethodGetAllProducts(Throwable cause) {
35         System.out.println("Exception raised with message :===>" + cause.getMessage());
36         return new ArrayList<Product>();
37     }
38 }
```



Check the Retries List

- Check the Retries List : (<http://localhost:9992/actuator/retries>)



The screenshot shows a browser window with the URL `localhost:9992/actuator/retries`. The page displays a JSON object with the following structure:

```
// 20230724131950
// http://localhost:9992/actuator/retries
{
  "retries": [
    "product-service"
  ]
}
```



Check the Retry Events

- Check the Retry Events: (<http://localhost:9992/actuator/retryevents>)



```
localhost:9992/actuator/retryevents
{
  "retryEvents": [
    {
      "retryName": "product-service",
      "type": "RETRY",
      "creationTime": "2023-07-24T13:26:42.611Z[UTC]",
      "errorMessage": "org.springframework.cloud.client.circuitbreaker.NoFallbackAvailableException: No fallback available.",
      "numberOfAttempts": 1
    },
    {
      "retryName": "product-service",
      "type": "ERROR",
      "creationTime": "2023-07-24T13:26:53.614Z[UTC]",
      "errorMessage": "org.springframework.cloud.client.circuitbreaker.NoFallbackAvailableException: No fallback available.",
      "numberOfAttempts": 2
    }
  ]
}
```



Recap of Day – 7

- Replacement capabilities available in new Spring Cloud version
- Adding Resilience4j
- Actuator /health Endpoint
- RestTemplate client & Circuit Breaker Fallback method
- Feign Client & Circuit Breaker Fallback method
- Circuit Breaker configuration properties
- Configure Access to Actuator endpoints
- Monitoring Circuit Breaker events in Actuator
- Retry with Spring Cloud Resilience4j



Day - 8

- What Problem Distributed Tracing Solves?
- How Distributed Tracing Works
- Distributed Tracing using Spring Cloud Sleuth and Zipkin
- Why monitoring?
- Introducing Prometheus
- Publishing metrics in Spring Boot 2.x: with Micrometer
- Adding Prometheus to Spring Boot
- Adding a custom metric
- Inspecting the metrics
- Getting metrics into Prometheus
- Observing the metrics in Prometheus



Day - 8

Spring Cloud – Zipkin and Sleuth



What is Zipkin?

- **Zipkin** is a very efficient tool for distributed tracing in the microservices ecosystem.
- Distributed tracing, in general, is the latency measurement of each component in a distributed transaction where multiple microservices are invoked to serve a single business use case.
- Distributed tracing is useful during debugging when lots of underlying systems are involved, and the application becomes slow in any situation.
- In such cases, we first need to identify which underlying service is slow. Once the slow service is identified, we can work to fix that issue. Distributed tracing helps in identifying that slow component in the ecosystem.



What is Zipkin?

- Zipkin was originally developed at Twitter, based on a concept of a Google paper that described Google's internally-built distributed app debugger – **dapper**. It manages both the collection and lookup of this data. To use Zipkin, applications are instrumented to report timing data to it.
- If you are troubleshooting latency problems or errors in an ecosystem, you can filter or sort all traces based on the application, length of trace, annotation, or timestamp. By analyzing these traces, you can decide which components are not performing as per expectations, and you can fix them.



What is Zipkin?

- Internally it has 4 modules –
 - **Collector** – Once any component sends the trace data, it arrives to Zipkin collector daemon. Here the trace data is validated, stored, and indexed for lookups by the Zipkin collector.
 - **Storage** – This module store and index the lookup data in backend. Cassandra, ElasticSearch and MySQL are supported.
 - **Search** – This module provides a simple JSON API for finding and retrieving traces stored in backend. The primary consumer of this API is the Web UI.
 - **Web UI** – A very nice UI interface for viewing traces.



Installing Zipkin

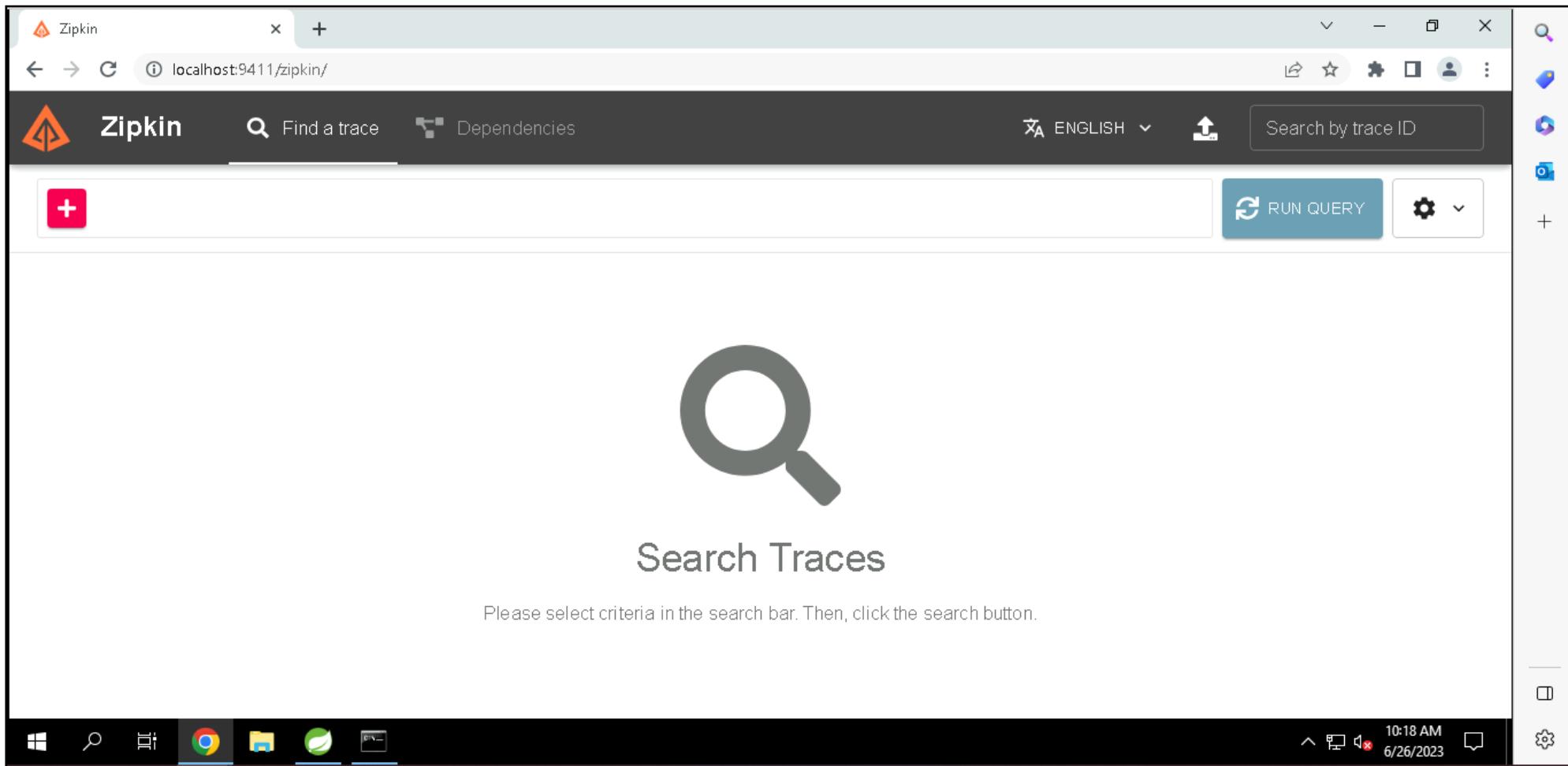
- Starting up Zipkin using Docker Zipkin project. Open the command prompt and execute the below docker command:

“docker run -d -p 9411:9411 openzipkin/zipkin”



Installing Zipkin

- Once Zipkin is up and running, you can open your web browser and go to <http://localhost:9411/zipkin>.





What is Sleuth?

- Sleuth is another tool from the Spring cloud family. It is used to generate the ***trace id***, ***span id*** and **add this information to the service calls in the headers**, so that it can be used by tools like Zipkin and ELK etc. to store, index and process log files.
- **Trace ID:**
 - A regional request that all comes from Microservices A, a Trace ID will be used.
 - All the microservice within that request will be labelled with same Trace ID.
 - When a new flow starts, a new Trace ID is generated.
- **Span ID:**
 - Is the unit of work.
 - For example, for every new HTTP request that your microservice will send a new span ID will be created.



Zipkin and Sleuth Integration with Ribbon Consumer Application

- Apply the steps to both provider and consumer either it may be Ribbon or Feign application.
1. Add the Zipkin and Sleuth starter dependency to pom.xml:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>

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



Zipkin and Sleuth Integration with Ribbon Consumer Application

2. Add the **Sampler.ALWAYS_SAMPLE** that traces each action in the Application class.

```
1 package com.mphasis;
2
3+import org.springframework.boot.SpringApplication;
4
5 @SpringBootApplication
6 public class StoreappConsumerEurekaRibbonApplication {
7
8
9     public static void main(String[] args) {
10         SpringApplication.run(StoreappConsumerEurekaRibbonApplication.class, args);
11     }
12
13     @Bean
14     public Sampler alwaysSampler() {
15
16         return Sampler.ALWAYS_SAMPLE;
17     }
18
19 }
20
21 }
```



Zipkin and Sleuth Integration with Ribbon Consumer Application

3. The ProductClientController which exposes an endpoint and invokes one downstream server using the RestTemplate. Let's apply the debug logger to the handler methods.

```
ProductClientController.java ✘
1 package com.mphasis.controller;
2
3+import org.slf4j.Logger;□
13
14 @RestController
15 @Scope("request")
16 public class ProductClientController {
17
18@Autowired
19 private ProductService productService;
20
21 private static final Logger log = LoggerFactory.getLogger(ProductClientController.class);
22
23@GetMapping("/get-products/{id}")
24 public Product getProductById(@PathVariable("id") int id) {
25
26     log.debug("In getProductById with Id:" + id);
27     Product product = productService.getProductById(id);
28     log.debug("In getProductById with return value product: " + product);
29     return product;
30 }
31 }
32
```



Zipkin and Sleuth Integration with Ribbon Consumer Application

4. Add the application name and DEBUG logger to the application.properties:

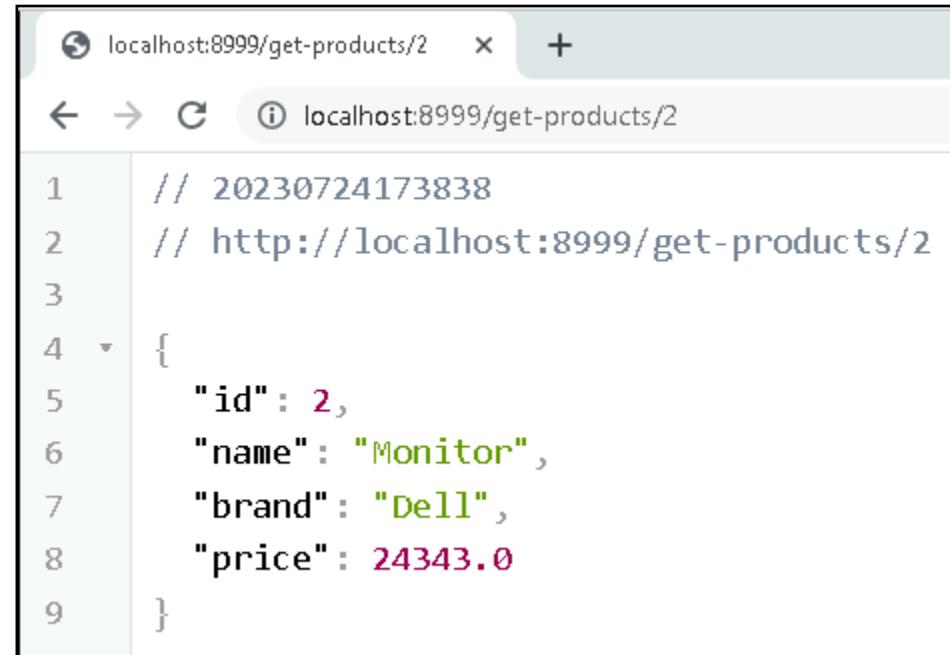
The screenshot shows a code editor window with the title "application.properties". The file contains configuration properties for a Spring Boot application. The properties include server port (8999), Eureka client configuration (service URL, register-with-eureka), application name (product-service-consumer), logging level for a specific controller (DEBUG), actuator endpoints (management endpoints), and resilience4j circuit breaker settings (instances for product-service, including sliding window type, size, minimum calls, event consumer buffer size, failure rate threshold, and automatic transition behavior).

```
1|  
2server.port=8999  
3eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka  
4eureka.client.register-with-eureka=false  
5  
6spring.application.name=product-service-consumer  
7logging.level.com.mphasis.controller.ProductClientController=DEBUG  
8  
9#Enable Actuator  
10management.endpoints.web.exposure.include=*  
11management.endpoint.health.show-details=always  
12management.health.circuitbreakers.enabled=true  
13management.health.ratelimiters.enabled=true  
14  
15resilience4j.circuitbreaker.circuit-breaker-aspect-order=1  
16  
17#resilience4j.circuitbreaker.instances.product-service.sliding-window-type=time-based  
18resilience4j.circuitbreaker.instances.product-service.sliding-window-type=count-based  
19resilience4j.circuitbreaker.instances.product-service.sliding-window-size=2  
20resilience4j.circuitbreaker.instances.product-service.minimum-number-of-calls=1  
21resilience4j.circuitbreaker.instances.product-service.event-consumer-buffer-size=10  
22resilience4j.circuitbreaker.instances.product-service.failure-rate-threshold=50  
23resilience4j.circuitbreaker.instances.product-service.automatic-transition-from-open-to-half-open-enabled=true
```



Zipkin and Sleuth Integration with Ribbon Consumer Application

5. Apply the similar steps to the provider i.e., storeapp application.
6. Test the Consumer Application:



A screenshot of a browser window displaying a JSON response. The URL in the address bar is `localhost:8999/get-products/2`. The response body contains the following JSON data:

```
1 // 20230724173838
2 // http://localhost:8999/get-products/2
3
4 {
5     "id": 2,
6     "name": "Monitor",
7     "brand": "Dell",
8     "price": 24343.0
9 }
```

7. Review the debug logs on the console of both the applications.



Zipkin and Sleuth Integration with Ribbon Consumer Application

- Let's go to Zipkin (<http://localhost:9411>) webpage and you can find a trace using the **traceId** or click on **RUN QUERY** button.

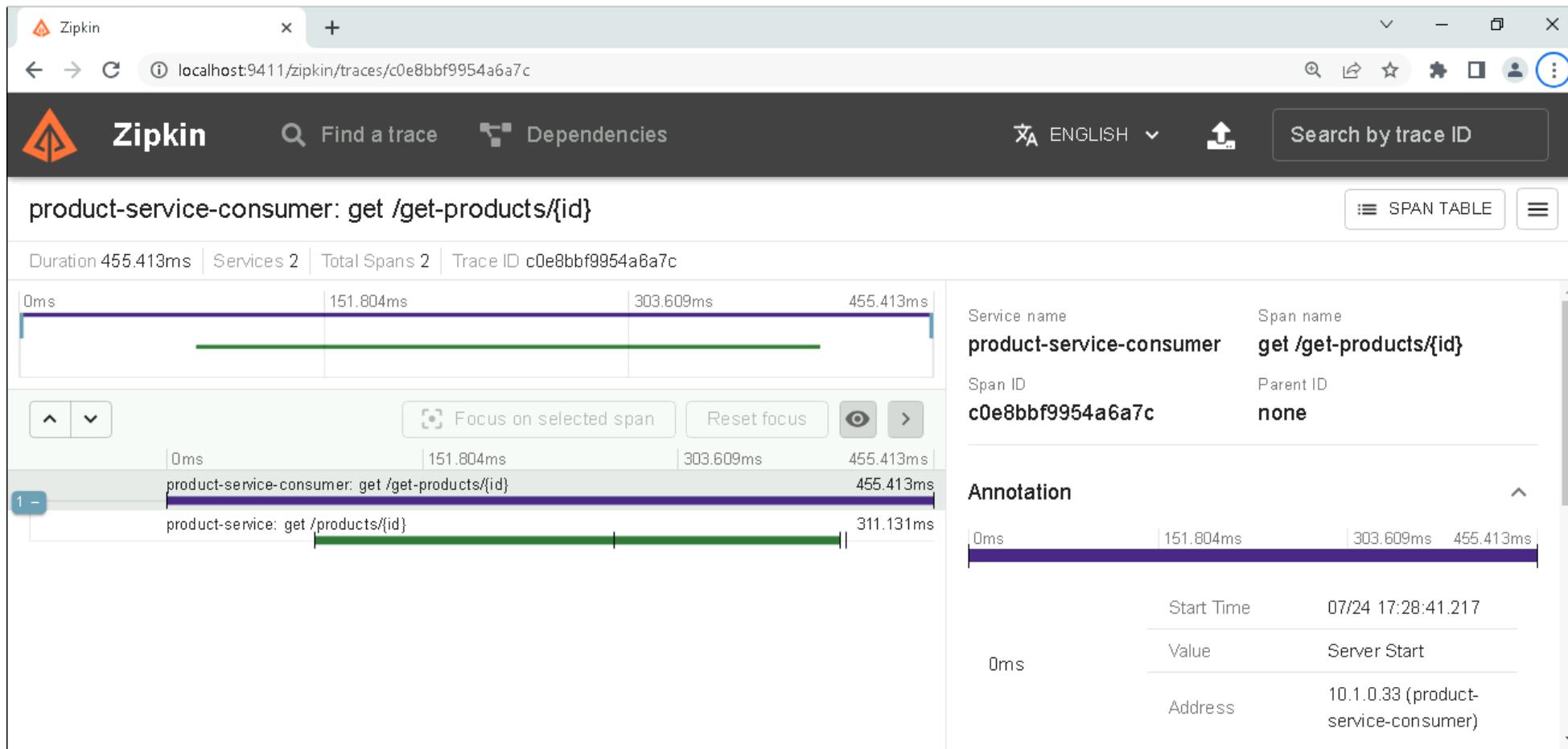
The screenshot shows the Zipkin web interface at <http://localhost:9411/zipkin/?lookback=15m&endTs=1690219918820&limit=10>. The interface includes a header with the Zipkin logo, a search bar, and tabs for 'Find a trace' and 'Dependencies'. Below the header, there are buttons for '+', 'RUN QUERY', and settings. The main area displays '2 Results' for a 'Root' trace. The first trace, 'product-service-consumer: get /get-products/{id}', has a start time of '5 minutes ago (07/24 17:31:06:131)' and contains 4 spans with a total duration of '15.052s'. The second trace, 'product-service-consumer: get /get-products/{id}', has a start time of '7 minutes ago (07/24 17:28:41:217)' and contains 3 spans with a total duration of '455.413ms'. Each trace entry includes a 'SHOW' button.

	Start Time	Spans	Duration	
product-service-consumer: get /get-products/{id}	5 minutes ago (07/24 17:31:06:131)	4	15.052s	SHOW
product-service-consumer: get /get-products/{id}	7 minutes ago (07/24 17:28:41:217)	3	455.413ms	SHOW



Zipkin and Sleuth Integration with Ribbon Consumer Application

- Click on SHOW button for the details about the Service i.e., Service name, Span name, Span ID and Parent ID.

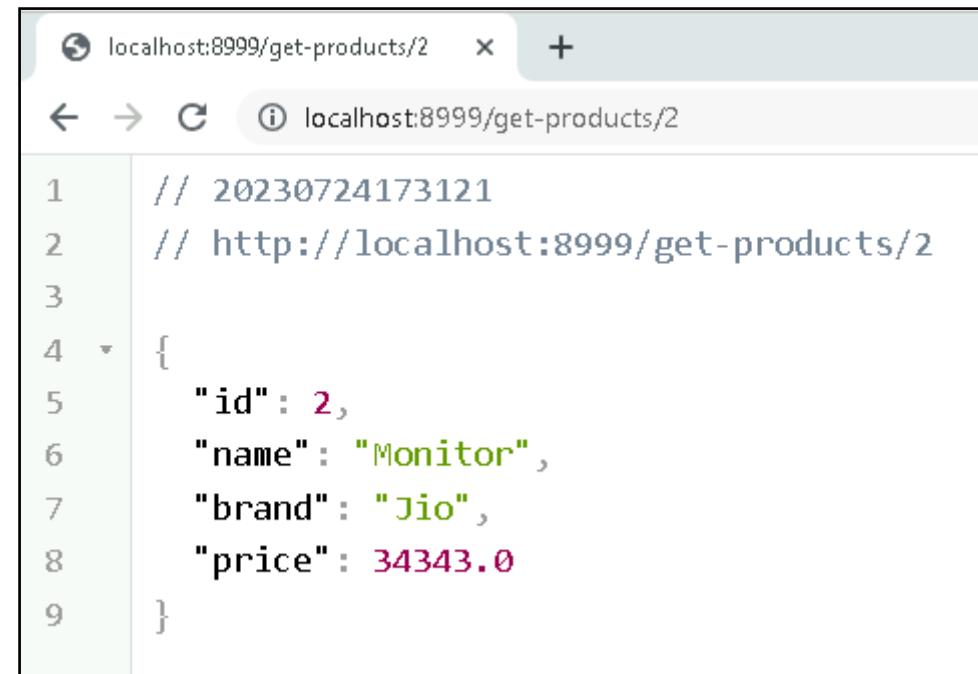




Zipkin and Sleuth Integration with Ribbon Consumer Application

10. Stop the storeapp application.

11. Test the Consumer Application, will go to fallback method:



A screenshot of a browser window displaying a JSON response. The URL in the address bar is `localhost:8999/get-products/2`. The response body shows a product object with the following fields and values:

```
1 // 20230724173121
2 // http://localhost:8999/get-products/2
3
4 {
5   "id": 2,
6   "name": "Monitor",
7   "brand": "Jio",
8   "price": 34343.0
9 }
```



Zipkin and Sleuth Integration with Ribbon Consumer Application

11. Let's go to Zipkin (<http://localhost:9411>) webpage and you can find a trace using the **traceId** or click on **RUN QUERY** button.

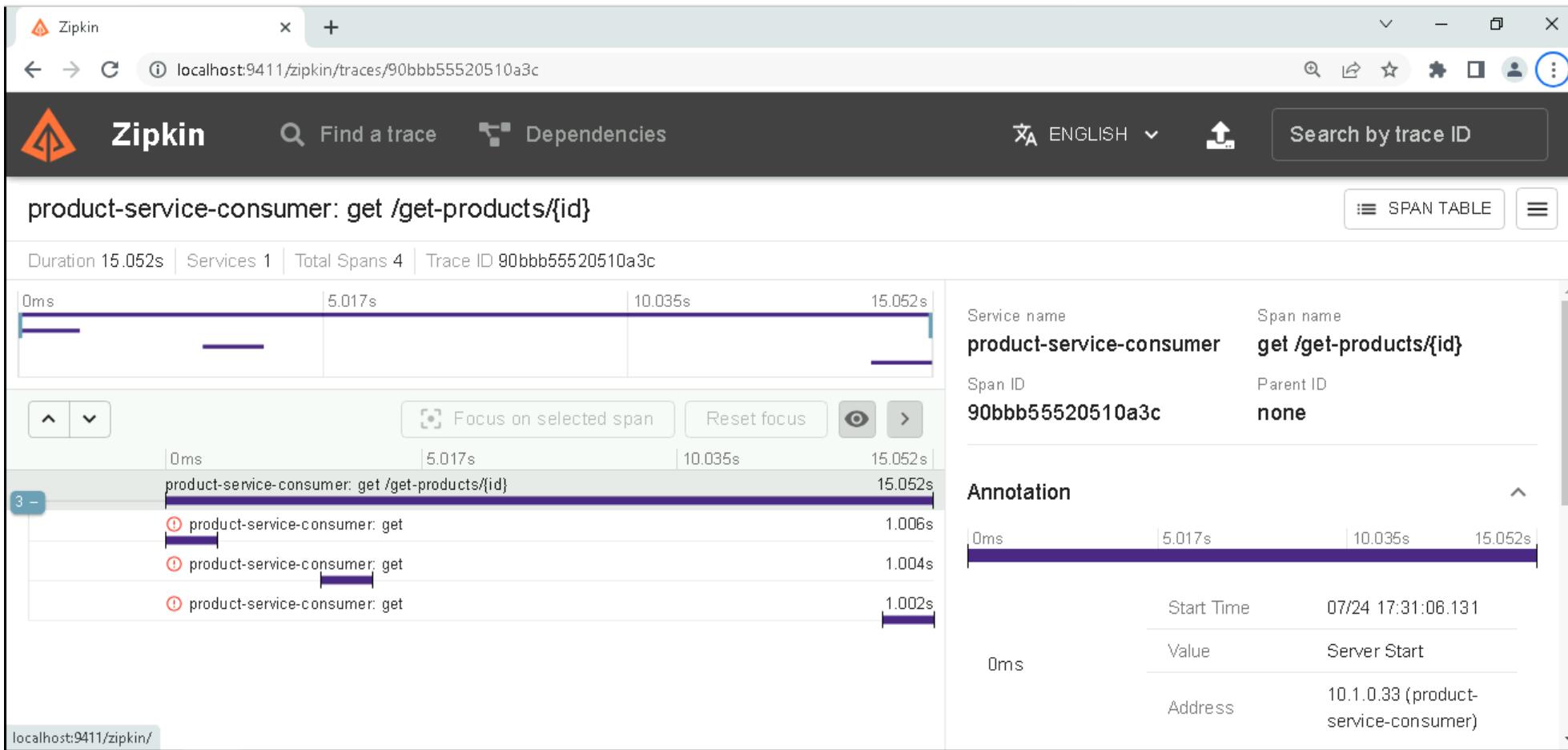
The screenshot shows the Zipkin web interface at <http://localhost:9411/zipkin/?lookback=15m&endTs=1690219918820&limit=10>. The interface includes a header with the Zipkin logo, a search bar, and language selection. Below the header, there are buttons for '+', 'Find a trace', 'Dependencies', and 'ENGLISH'. A 'RUN QUERY' button is prominently displayed. The main area shows '2 Results' with two entries:

	Start Time	Spans	Duration	Action
Root				
product-service-consumer: get /get-products/{id}	5 minutes ago (07/24 17:31:06:131)	4	15.052s	SHOW
product-service-consumer (4)				
product-service-consumer: get /get-products/{id}	7 minutes ago (07/24 17:28:41:217)	3	455.413ms	SHOW
product-service-consumer (2) product-service (1)				



Zipkin and Sleuth Integration with Ribbon Consumer Application

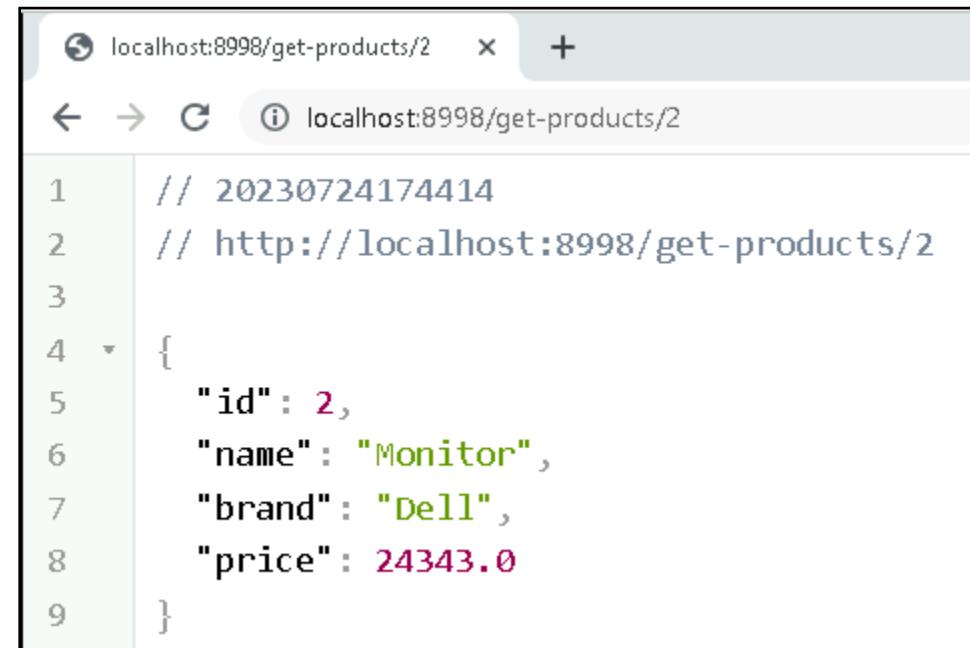
12. Click on SHOW button for the details about the Service i.e., Service name, Span name, Span ID and Parent ID.





Zipkin and Sleuth Integration with Feign Consumer Application

1. Apply the similar steps to the Consumer Application created using Feign.
2. Test the Consumer Application:



```
localhost:8998/get-products/2
localhost:8998/get-products/2

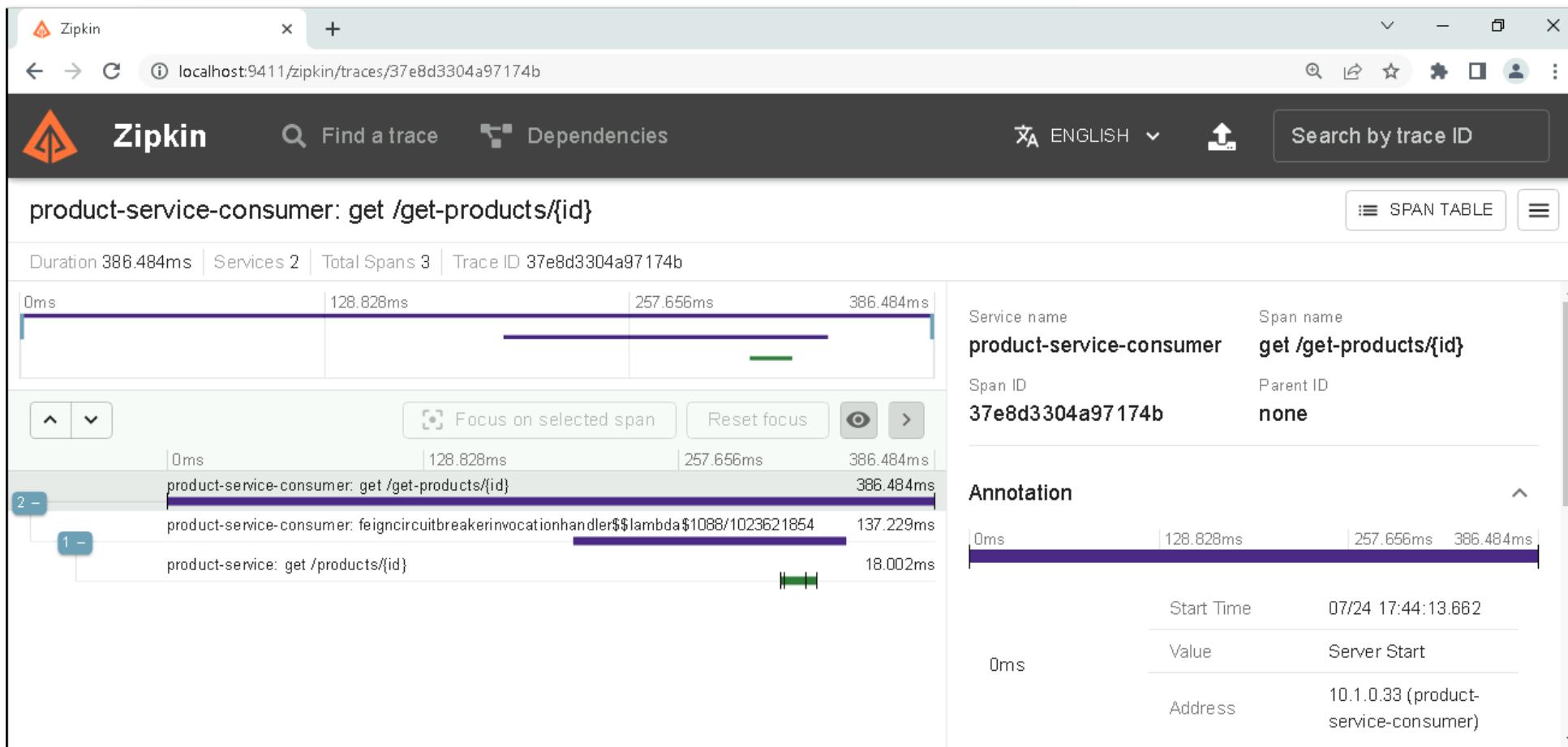
1 // 20230724174414
2 // http://localhost:8998/get-products/2
3
4 {
5     "id": 2,
6     "name": "Monitor",
7     "brand": "Dell",
8     "price": 24343.0
9 }
```

3. Review the debug logs on the console of both the applications.
4. Let's go to Zipkin (<http://localhost:9411>) webpage and you can find a trace using the **traceId** or click on **RUN QUERY** button.



Zipkin and Sleuth Integration with Feign Consumer Application

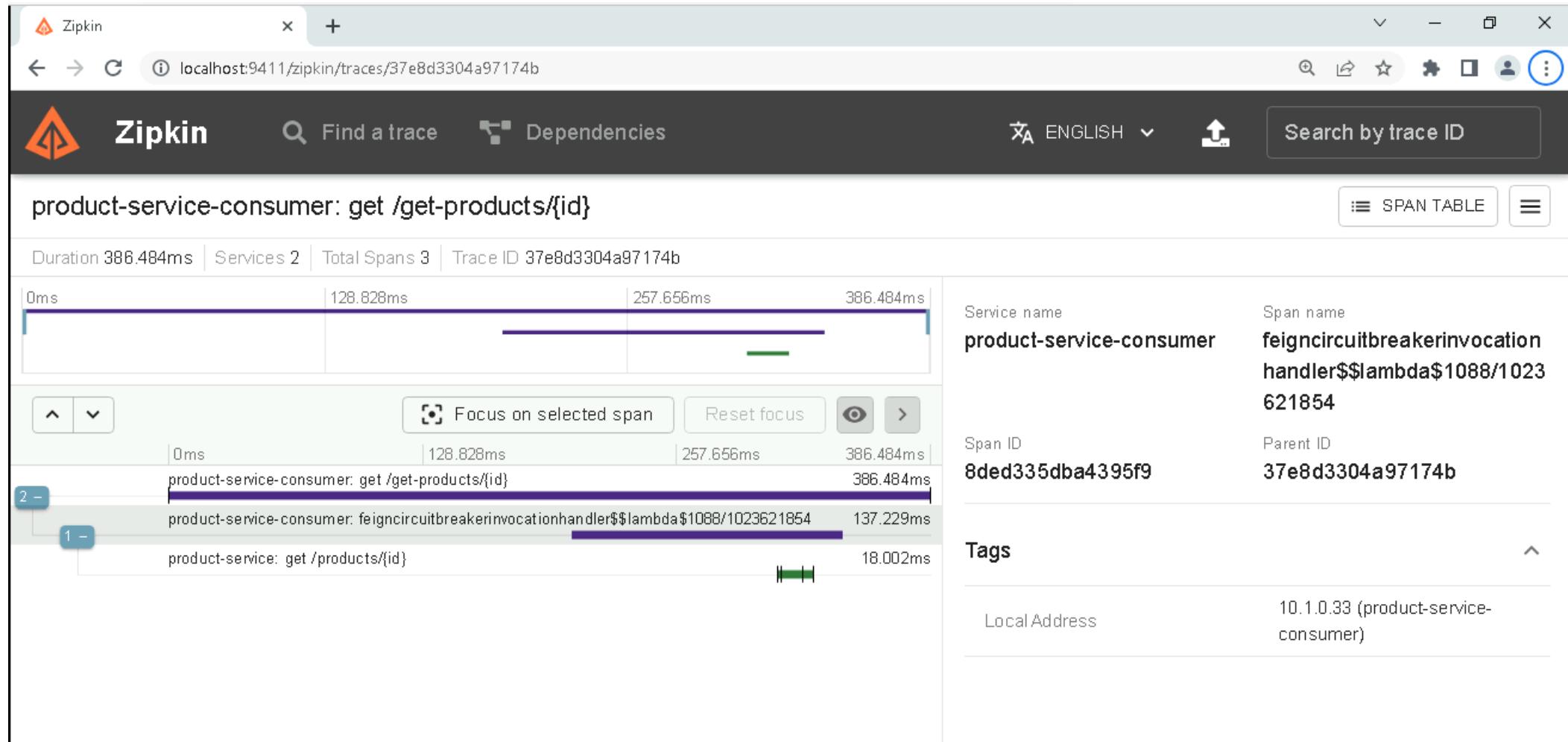
- Click on SHOW button for the details about the Service i.e., Service name, Span name, Span ID and Parent ID.





Zipkin and Sleuth Integration with Feign Consumer Application

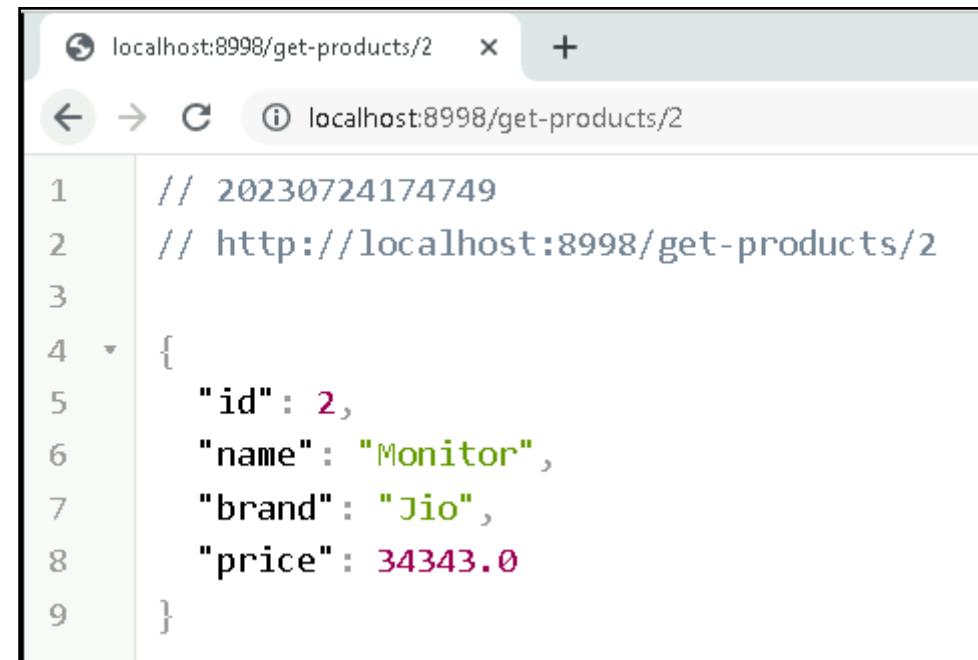
6. You will find the Span Name for the feigncircuitbreakerinvocationhandler:





Zipkin and Sleuth Integration with Feign Consumer Application

7. Stop the storeapp application.
8. Test the Consumer Application, will go to fallback method:



A screenshot of a browser window displaying a JSON response. The URL in the address bar is `localhost:8998/get-products/2`. The response body contains the following JSON data:

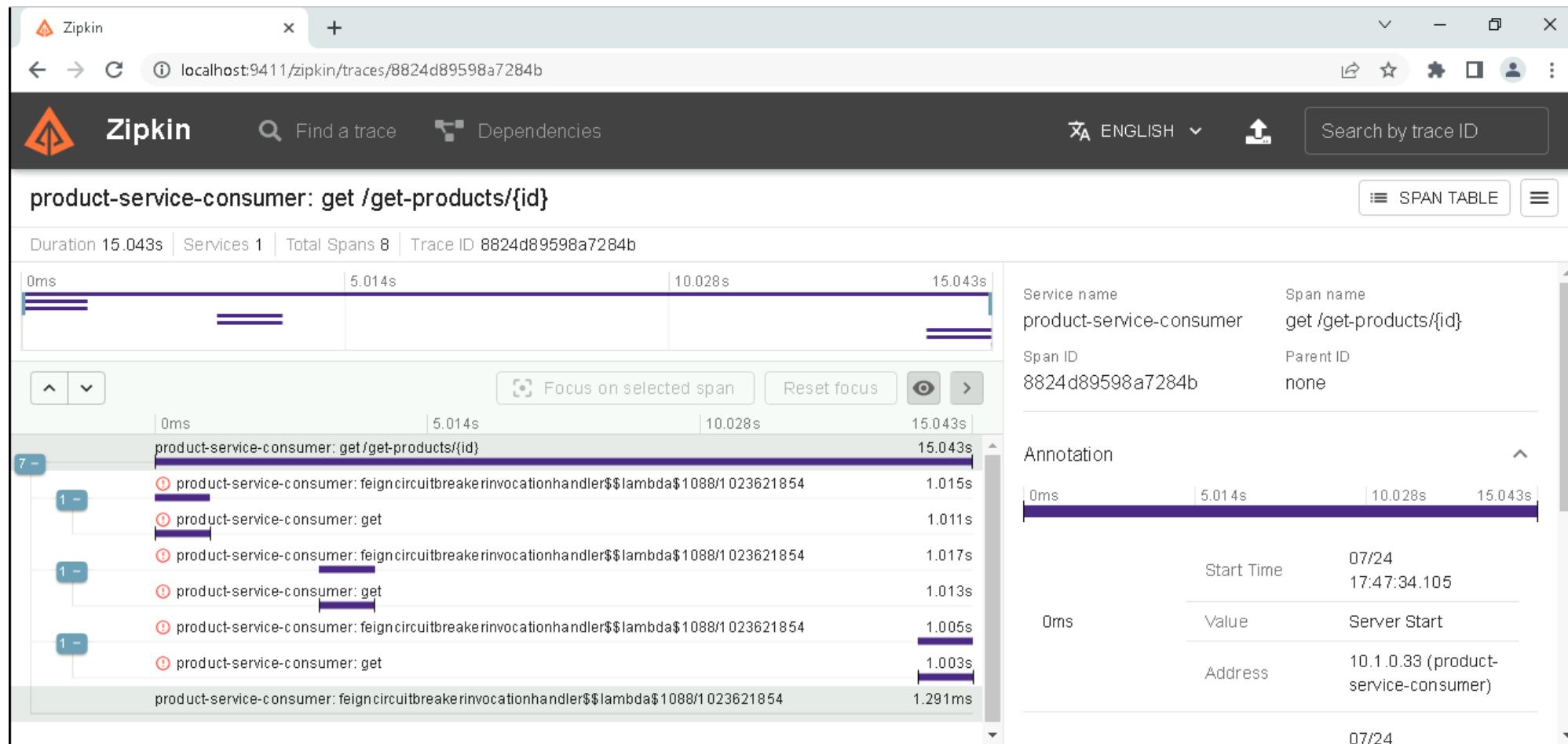
```
1 // 20230724174749
2 // http://localhost:8998/get-products/2
3
4 {
5   "id": 2,
6   "name": "Monitor",
7   "brand": "Jio",
8   "price": 34343.0
9 }
```

9. Let's go to Zipkin (<http://localhost:9411>) webpage and you can find a trace using the **traceId** or click on **RUN QUERY** button.



Zipkin and Sleuth Integration with Feign Consumer Application

- Click on SHOW button for the details about the Service i.e., Service name, Span name, Span ID and Parent ID.





Day - 8

Spring Boot app metrics - with Prometheus and Micrometer



Why monitoring?

- Historically, it's been quite difficult to implement a standard way of monitoring many different types of applications.
- Each language and framework has its own way of doing things. Unless you wanted to spend a lot of money on enterprise software and install a bunch of agents everywhere, you probably had to roll your own solution.
- But in the last few years, people have started to realize that finding a standard way of monitoring applications is becoming rather essential.
- I think this is for a few reasons:
 - Firstly, the number of applications that we are deploying is growing so much larger. Apps are getting smaller, and there are simply far more of them.
 - Secondly, the business now expects much more return from its investment in software. And part of this expectation is being able to see clear metrics about how well software is doing.



Why monitoring?

- And finally, the DevOps age is upon us. One of the key tenets of DevOps is **measurement and learning**. We want to measure the success of our applications and learn from the results.
- So, is there a tool that can help us record technical metrics, and make it easy to see trends?
- Yes, that tool right now is **Prometheus**.



Introducing Prometheus

- Prometheus is one way of solving this problem. It is a time-series database, which stores a sequence of data points, across time.
- It's generally used to store metrics and performance data from your applications. And this allows you to perform **time-series analysis** of metrics.
- The time-series analysis allows you to look at a set of data over time, and see trends, how you're performing, and perhaps make some estimations about the future.
- Prometheus runs separately from your application. So, you can run a single instance of Prometheus, and it can fetch and store metrics from dozens of your apps.
- Prometheus uses a **pull-based** approach for getting metrics.
- Prometheus polls your application for its latest metrics data – this is known as **scraping**. Then, Prometheus adds the results into its time-series database.



Why use Prometheus?

- Prometheus isn't the only time-series database and metrics tool, but it is certainly the one which has got a lot of traction.
- It's got built-in support for scraping applications, it's got a built-in UI and query language for drawing charts, and it's a CNCF Graduated Project.



Publishing metrics in Spring Boot 2.x: with Micrometer

- Micrometer can help you take measurements from your application and publish those metrics ready to be scraped by many different applications, including Prometheus.
- Micrometer is a set of libraries for Java that allow you to capture metrics and expose them to several different tools – including Prometheus.
- Micrometer acts as a facade – an intermediate layer – between your application and some of the more popular monitoring tools. This makes it easier to publish metrics to Prometheus and other tools like Elastic, Datadog or Dynatrace.
- This is great for us, because once we have added Micrometer to our application, it will happily sit there collecting and exposing metrics. We can choose to enable the publishing of metrics to a specific tool, like Prometheus, with a simple switch.

1. Add the following dependencies to Ribbon Customer Application or any application.
 - Adding Actuator to Spring Boot:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

- This will configure the Actuator, which already includes Micrometer.

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-core</artifactId>
</dependency>
```



Adding Prometheus to Spring Boot

- Next, you will need to add the Micrometer registry dependency which specifically enables Prometheus support.
- This allows the metrics collected by Micrometer to be exposed in a Prometheus way:

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
    <scope>runtime</scope>
</dependency>
```

2. Add the following property to application.properties file:

```
6 #Enable Actuator  
7 #management.endpoints.web.exposure.include=*<br/>  
8 management.endpoints.web.exposure.include=health,info,prometheus  
9
```

3. Now we're good to go! With these dependencies added to your Spring Boot 2.x application, many metrics will be exposed automatically on the actuator endpoint, which is **/actuator/prometheus**.
4. You can stop here if you like, and just expose some basic metrics: start the application as normal, and send a request to **http://localhost:8080/actuator/prometheus**



Adding a custom metric

- To add a custom timer, we need to add a new dependency – the AOP dependency for Spring Boot.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```



Adding a custom metric

6. We also need to register the **TimedAspect** bean in our Spring context. This will allow Micrometer to add a timer to custom methods. Register the bean to your **@SpringBootApplication** or **@Configuration** class:

```
1 package com.mphasis;
2
3+import org.springframework.boot.SpringApplication;
4
5 @SpringBootApplication
6 public class StoreappConsumerEurekaRibbonApplication {
7
8     public static void main(String[] args) {
9         SpringApplication.run(StoreappConsumerEurekaRibbonApplication.class, args);
10    }
11
12    @Bean
13    public TimedAspect timedAspect(MeterRegistry registry) {
14        return new TimedAspect(registry);
15    }
16
17 }
```



Adding a custom metric

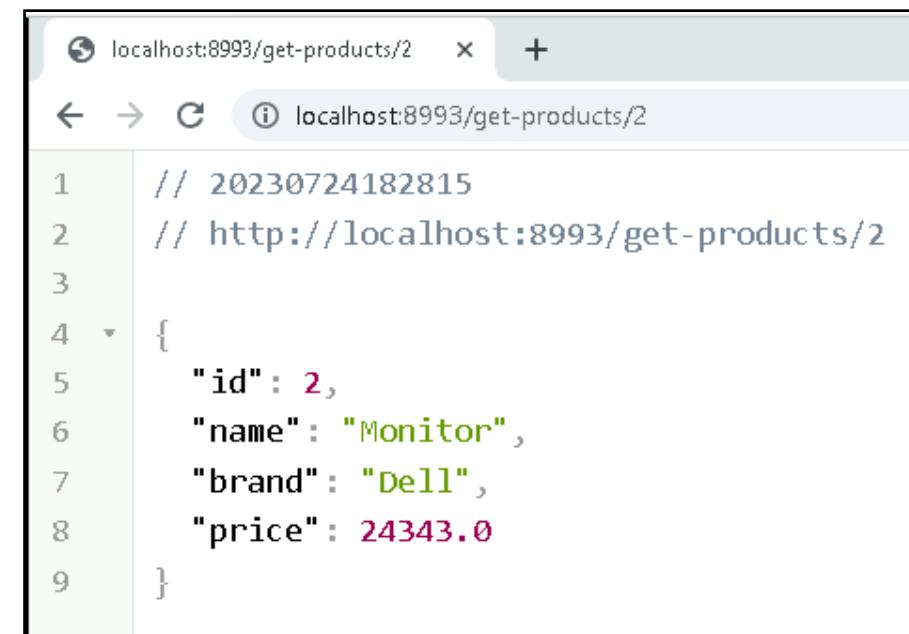
7. Then, find the method that you want to time, and add the **@Timed** annotation to it. Use the value attribute to give the metric a name.

```
J ProductClientController.java ✘
1 package com.mphasis.controller;
2
3+import org.springframework.beans.factory.annotation.Autowired;□
13
14 @RestController
15 @Scope("request")
16 public class ProductClientController {
17
18@    @Autowired
19    private ProductService productService;
20
21@    @Timed(value = "getProductById.time", description = "Time taken to return Product")
22    @GetMapping("/get-products/{id}")
23    public Product getProductById(@PathVariable("id") int id) {
24
25        return productService.getProductById(id);
26    }
27}
28
```



Inspecting the metrics

8. Let's look at the metrics now being exposed by Spring Boot's Actuator, and Micrometer.
9. Start your application, and send an HTTP get request to **http://localhost:8080/actuator/prometheus**. You will see all the metrics being exposed.
10. Test the Consumer Application:



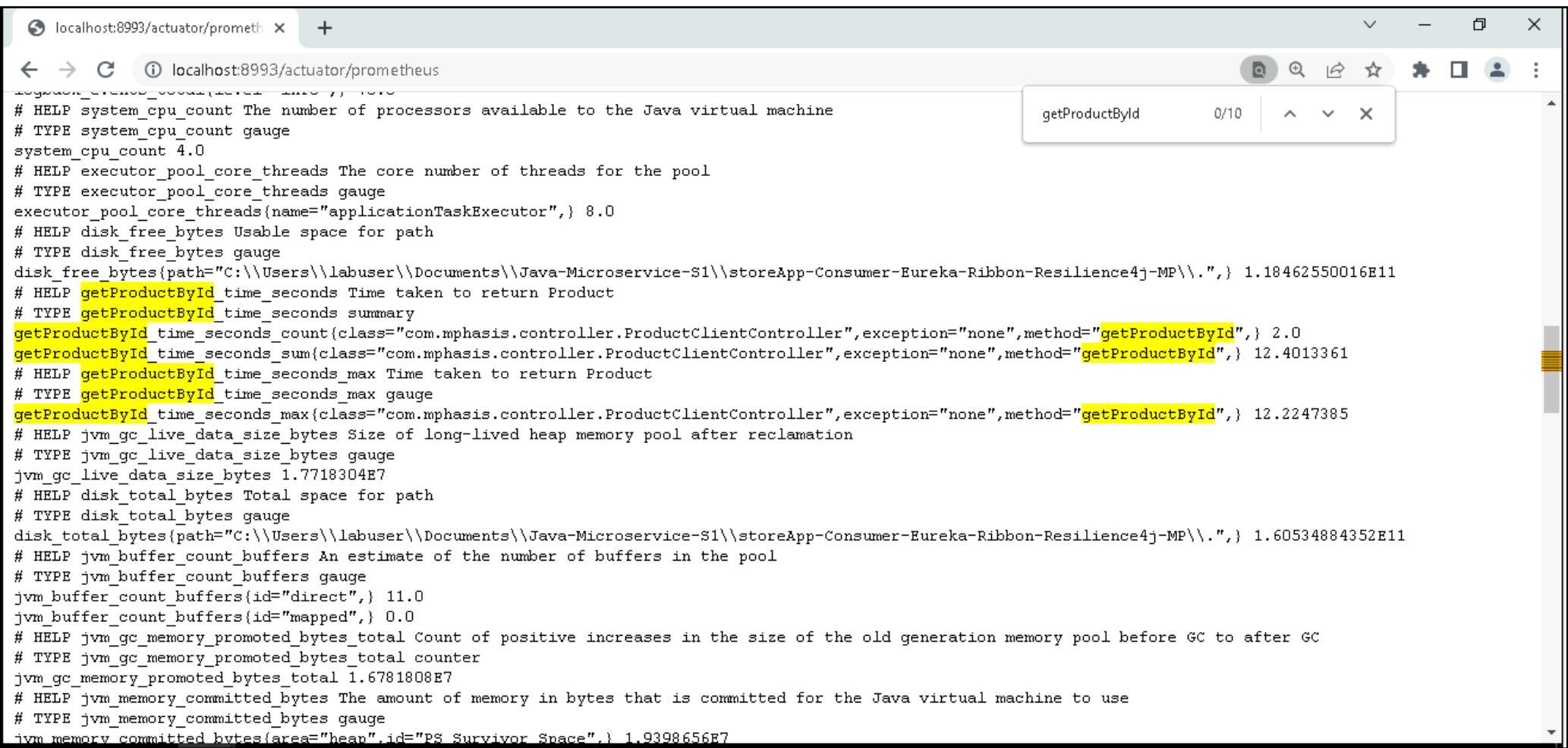
A screenshot of a browser window displaying the JSON response for a GET request to `localhost:8993/get-products/2`. The response shows a single product object with the following details:

```
// 20230724182815
// http://localhost:8993/get-products/2
{
  "id": 2,
  "name": "Monitor",
  "brand": "Dell",
  "price": 24343.0
}
```



Inspecting the metrics

- Now I can see the `getProductById_time_seconds_*` metrics being exposed, which show the number of times my method was executed, the total time taken, and the maximum:



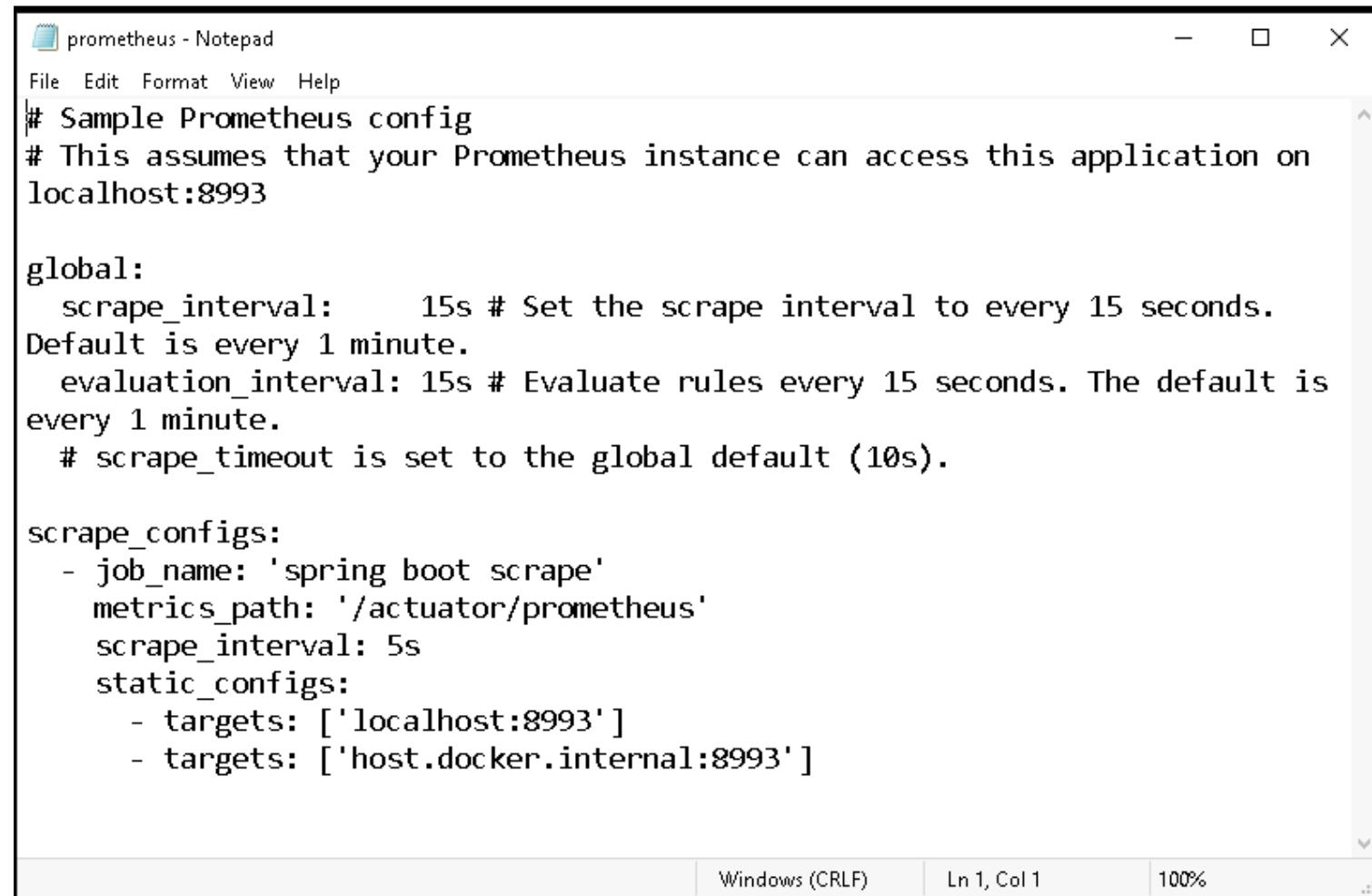
```
localhost:8993/actuator/prometh x +  
localhost:8993/actuator/prometheus  
logback_level_value{level='INFO'} 1.0  
# HELP system_cpu_count The number of processors available to the Java virtual machine  
# TYPE system_cpu_count gauge  
system_cpu_count 4.0  
# HELP executor_pool_core_threads The core number of threads for the pool  
# TYPE executor_pool_core_threads gauge  
executor_pool_core_threads{name="applicationTaskExecutor",} 8.0  
# HELP disk_free_bytes Usable space for path  
# TYPE disk_free_bytes gauge  
disk_free_bytes(path="C:\\\\Users\\\\labuser\\\\Documents\\\\Java-Microservice-S1\\\\storeApp-Consumer-Eureka-Ribbon-Resilience4j-MP\\\\..") 1.18462550016E11  
# HELP getProductById_time_seconds Time taken to return Product  
# TYPE getProductById_time_seconds summary  
getProductById_time_seconds_count{class="com.mphasis.controller.ProductClientController",exception="none",method="getProductById",} 2.0  
getProductById_time_seconds_sum{class="com.mphasis.controller.ProductClientController",exception="none",method="getProductById",} 12.4013361  
# HELP getProductById_time_seconds_max Time taken to return Product  
# TYPE getProductById_time_seconds_max gauge  
getProductById_time_seconds_max{class="com.mphasis.controller.ProductClientController",exception="none",method="getProductById",} 12.2247385  
# HELP jvm_gc_live_data_size_bytes Size of long-lived heap memory pool after reclamation  
# TYPE jvm_gc_live_data_size_bytes gauge  
jvm_gc_live_data_size_bytes 1.7718304E7  
# HELP disk_total_bytes Total space for path  
# TYPE disk_total_bytes gauge  
disk_total_bytes(path="C:\\\\Users\\\\labuser\\\\Documents\\\\Java-Microservice-S1\\\\storeApp-Consumer-Eureka-Ribbon-Resilience4j-MP\\\\..") 1.60534884352E11  
# HELP jvm_buffer_count_buffers An estimate of the number of buffers in the pool  
# TYPE jvm_buffer_count_buffers gauge  
jvm_buffer_count_buffers{id="direct",} 11.0  
jvm_buffer_count_buffers{id="mapped",} 0.0  
# HELP jvm_gc_memory_promoted_bytes_total Count of positive increases in the size of the old generation memory pool before GC to after GC  
# TYPE jvm_gc_memory_promoted_bytes_total counter  
jvm_gc_memory_promoted_bytes_total 1.6781808E7  
# HELP jvm_memory_committed_bytes The amount of memory in bytes that is committed for the Java virtual machine to use  
# TYPE jvm_memory_committed_bytes gauge  
jvm_memory_committed_bytes{area="heap",id="PS Survivor Space",} 1.9398656E7
```



Getting metrics into Prometheus

12. Let's specify the scraping rules in a settings YAML file:

13. Create a “**prometheus.yml**” file on C drive:



```
prometheus - Notepad
File Edit Format View Help
# Sample Prometheus config
# This assumes that your Prometheus instance can access this application on
localhost:8993

global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds.
Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is
every 1 minute.
  # scrape_timeout is set to the global default (10s).

scrape_configs:
  - job_name: 'spring boot scrape'
    metrics_path: '/actuator/prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:8993']
      - targets: ['host.docker.internal:8993']
```

14. Next, we need to start Prometheus and feed this config to it.
15. Bind-mount your “**prometheus.yml**” from the host by running:

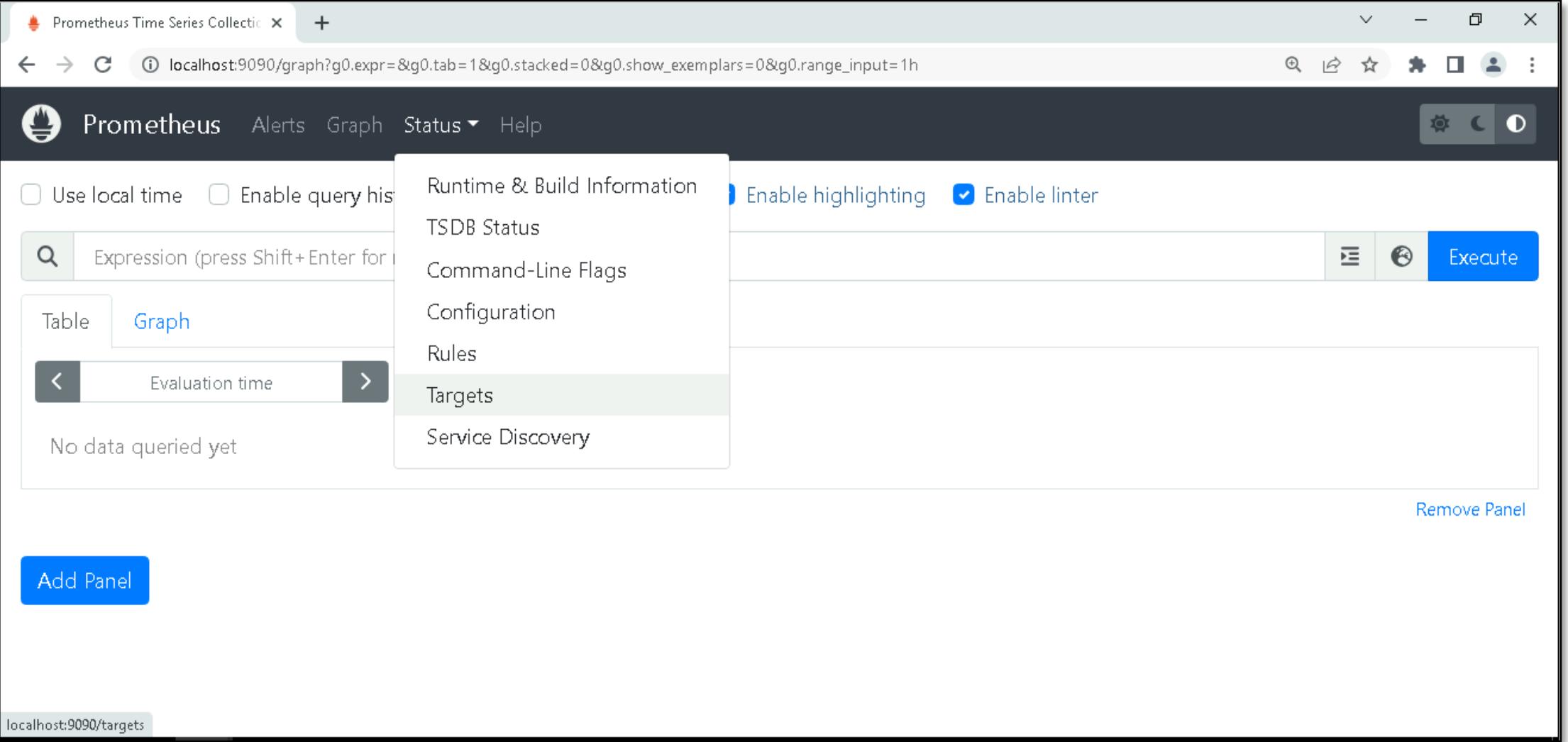
```
docker run -p 9090:9090 -v  
C:/prometheus.yml:/etc/prometheus/prometheus.yml  
prom/prometheus
```

16. Prometheus will be running on <http://localhost:9090> .



Observing the metrics in Prometheus

17. Click on Targets:



The screenshot shows the Prometheus web interface. At the top, there is a navigation bar with links for Prometheus, Alerts, Graph, Status, and Help. Below the navigation bar, there is a search bar with the placeholder "Expression (press Shift+Enter for help)" and a dropdown menu showing "localhost:9090/graph?g0.expr=&g0.tab=1&g0.stacked=0&g0.show_exemplars=0&g0.range_input=1h". On the left, there is a sidebar with options for "Use local time", "Enable query history", and a search field. Below these are tabs for "Table" and "Graph", with "Graph" currently selected. A date range selector shows "Evaluation time" with arrows for navigating between dates. A message "No data queried yet" is displayed. To the right of the search bar, there is a panel with several configuration options: "Runtime & Build Information", "TSDB Status", "Command-Line Flags", "Configuration", "Rules", "Targets" (which is highlighted in blue), and "Service Discovery". There are also checkboxes for "Enable highlighting" and "Enable linter", and a "Execute" button. At the bottom of the interface, there is a footer with the URL "localhost:9090/targets".



Observing the metrics in Prometheus

- Now we're satisfied that Prometheus is scraping from our application, we can try searching for a metric, and drawing a chart.

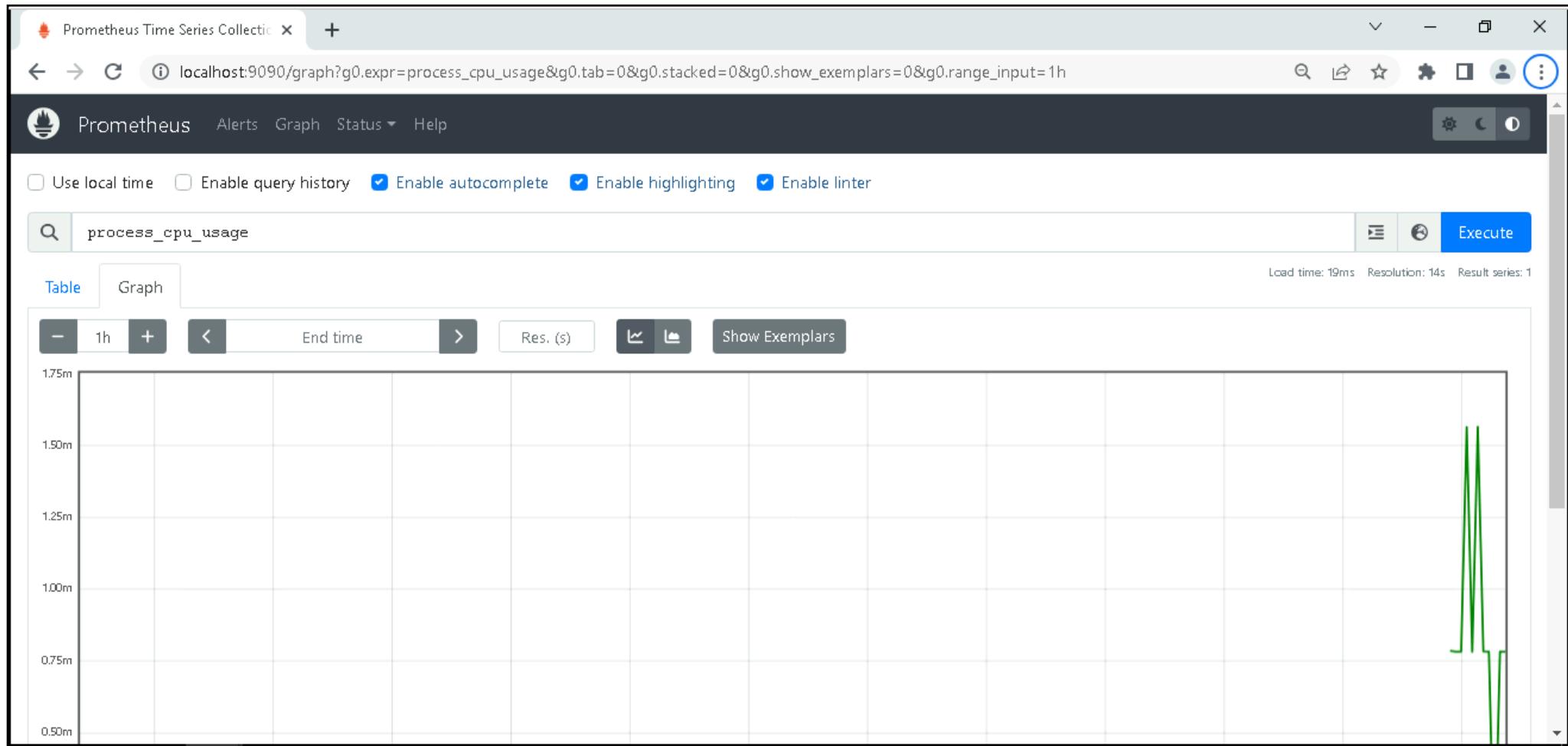
The screenshot shows the Prometheus Targets page at `localhost:9090/targets?search=`. The page has a dark header with the Prometheus logo and navigation links for Alerts, Graph, Status, and Help. Below the header is a search bar and filter buttons for Unknown, Unhealthy, and Healthy states. A table lists a single target: "spring boot scrape (1/1 up)". The table columns are Endpoint, State, Labels, Last Scrape, Scrape Duration, and Error. The endpoint is `http://host.docker.internal:8993/actuator/prometheus`, state is UP, labels include `instance="host.docker.internal:8993"` and `job="spring boot scrape"`, last scrape was 27.379s ago, and scrape duration was 44.541ms.

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://host.docker.internal:8993/actuator/prometheus	UP	<code>instance="host.docker.internal:8993"</code> <code>job="spring boot scrape"</code>	27.379s ago	44.541ms	



Observing the metrics in Prometheus

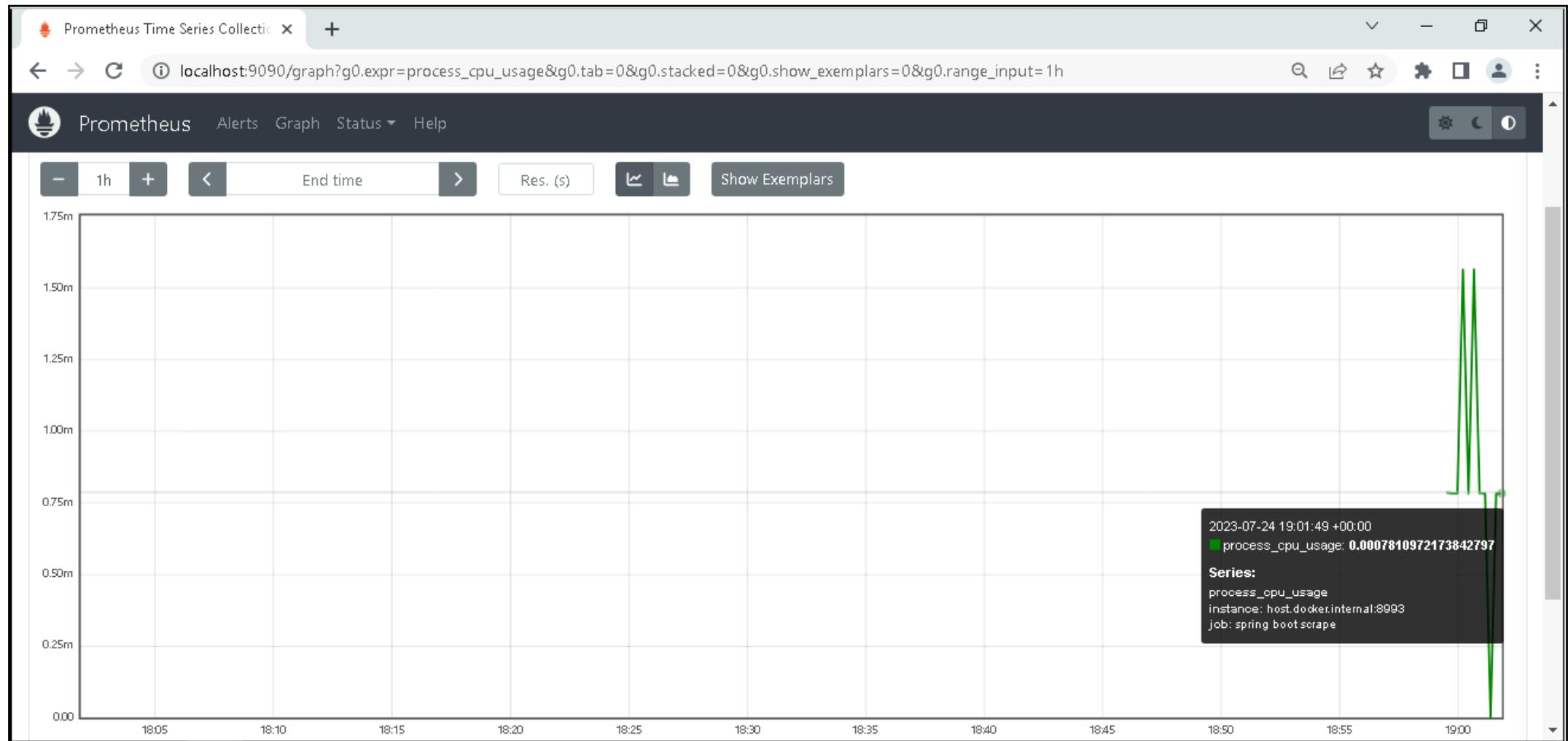
19. Go to the Graph tab. Search for the metric process_cpu_usage and Prometheus will create a chart from it:





Observing the metrics in Prometheus

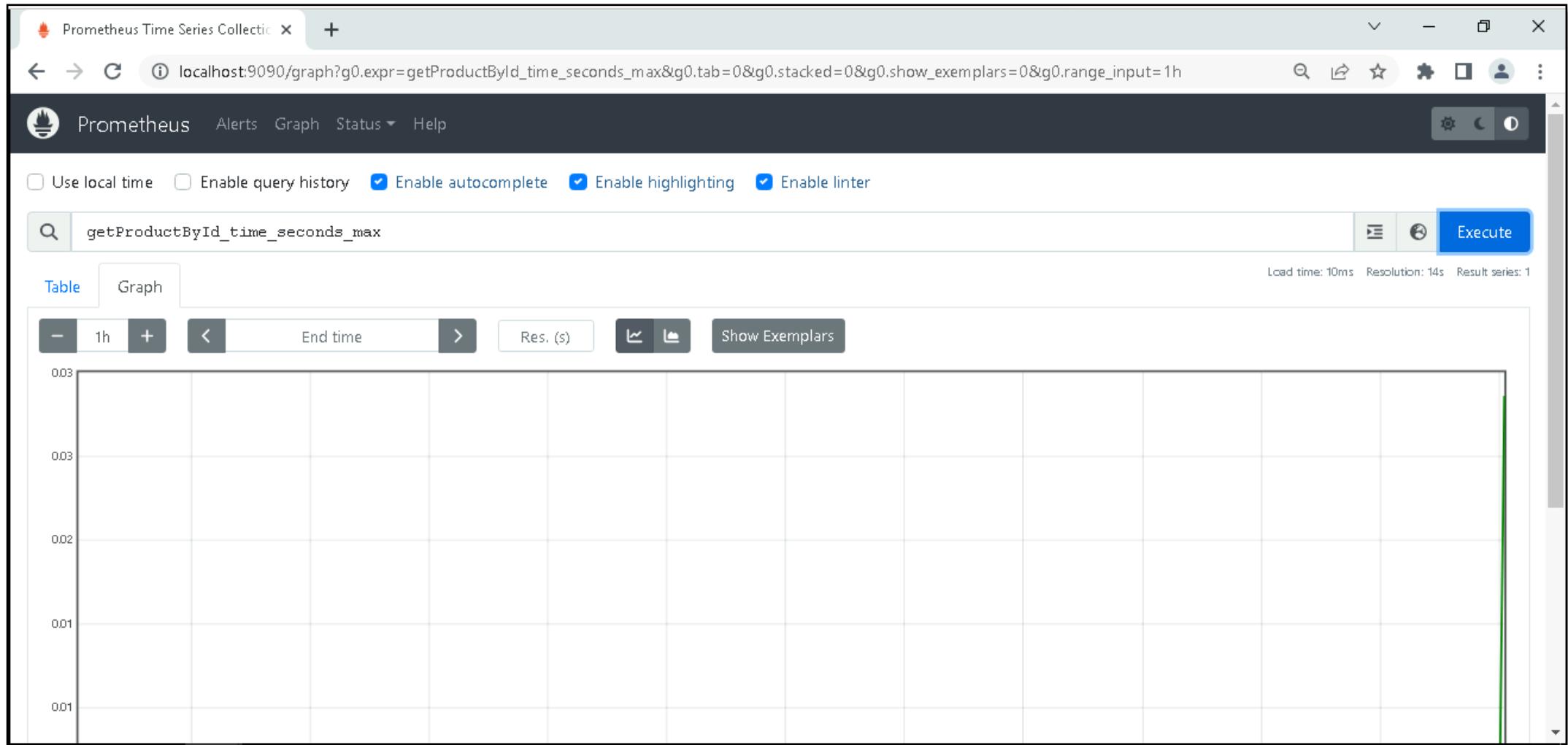
20. Rendering a Chart:





Observing the metrics in Prometheus

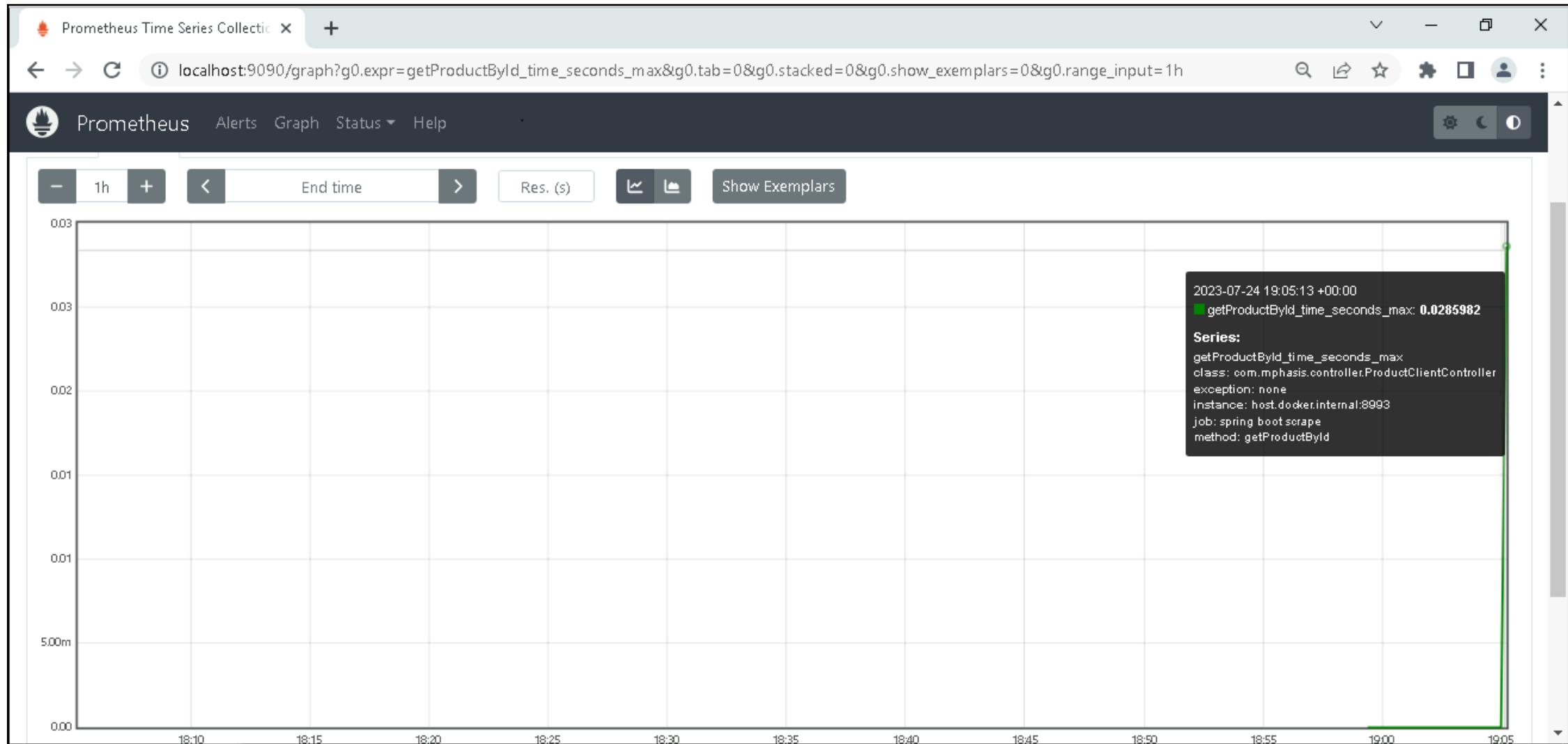
21. Go to the Graph tab. Search for the custom metric `getProductId_time_seconds_max` and Prometheus will create a chart from it:





Observing the metrics in Prometheus

22. Rendering a Chart:



- What Problem Distributed Tracing Solves?
- How Distributed Tracing Works
- Distributed Tracing using Spring Cloud Sleuth and Zipkin
- Why monitoring?
- Introducing Prometheus
- Publishing metrics in Spring Boot 2.x: with Micrometer
- Adding Prometheus to Spring Boot
- Adding a custom metric
- Inspecting the metrics
- Getting metrics into Prometheus
- Observing the metrics in Prometheus



Day - 9



Day – 9 Agenda

- What is an API Gateway?
- Spring Cloud Gateway Architecture
- Configuring routes in Spring Cloud Gateway
- Dynamically reload route configuration
- Built-in Predicates and Filters Factories
- Creating Custom Filters
- Global Filters using Spring Cloud Gateway
- Netflix Eureka Discovery Service Implementation



Day - 9

Spring Cloud Gateway with Load Balancing



What is an API Gateway?

- An API Gateway acts as a single-entry point for a collection of microservices. Any external client cannot access the microservices directly but can access them only through the application gateway.
- In a real-world scenario, an external client can be any one of the three-
 - Mobile Application
 - Desktop Application
 - External Services or third-party Apps



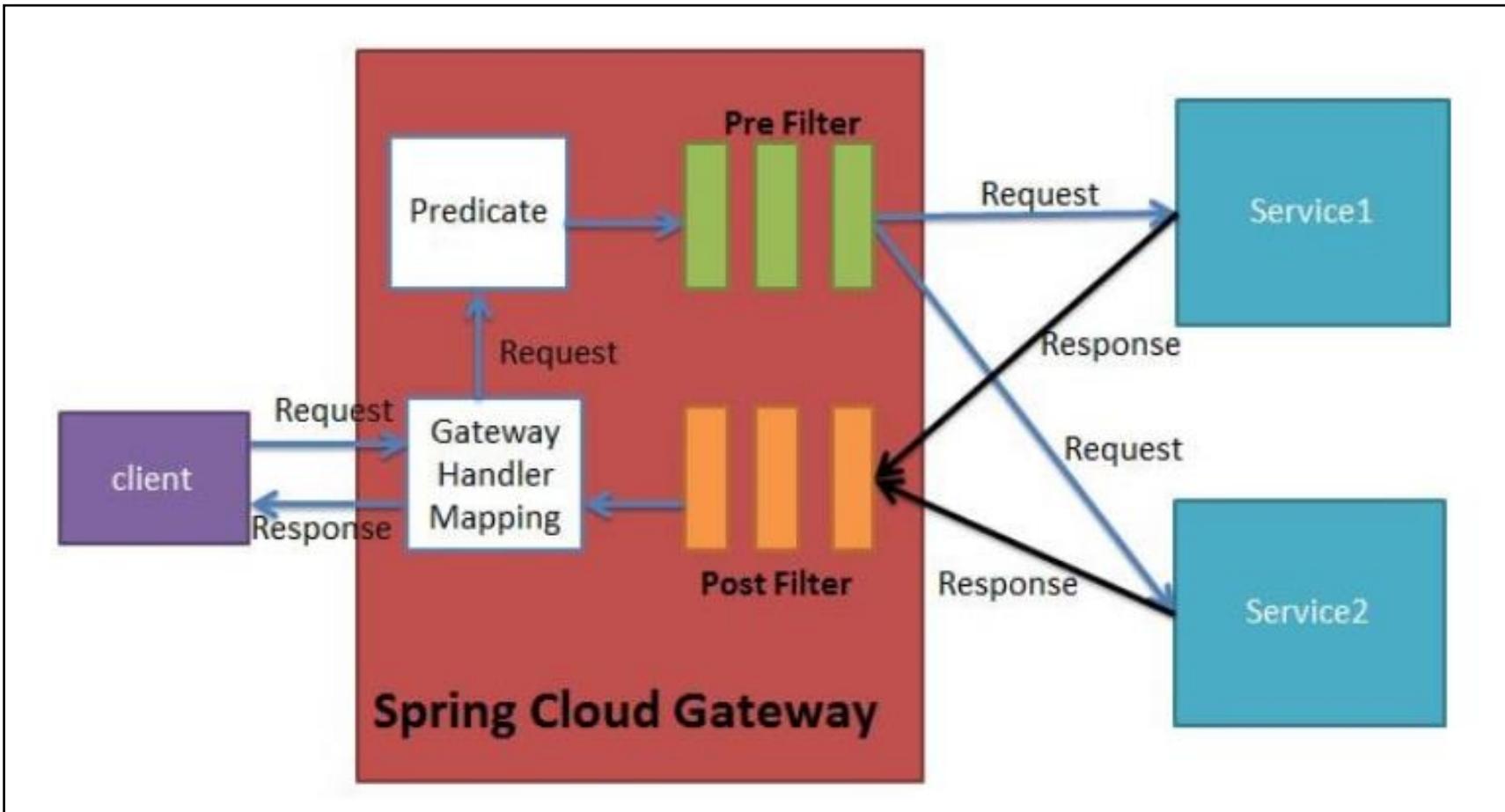
Advantages of API Gateway

- This improves the security of the microservices as we limit the access of external calls to all our services.
- The cross-cutting concerns like authentication, monitoring/metrics, and resiliency will be needed to be implemented only in the API Gateway as all our calls will be routed through it.
- The client does not know about the internal architecture of our microservices system. Client will not be able to determine the location of the microservice instances.
- Simplifies client interaction as he will need to access only a single service for all the requirements.

- Zuul is a blocking API. A blocking gateway API makes use of as many threads as the number of incoming requests. So, this approach is more resource intensive. If no threads are available to process incoming request, then the request must wait in queue.
- Spring Cloud Gateway is a non-blocking API.
- When using non-blocking API, a thread is always available to process the incoming request. These request are then processed asynchronously in the background and once completed the response is returned. So, no incoming request never gets blocked when using Spring Cloud Gateway.
- Spring Cloud API Gateway has an in-built Load Balancer.

- Spring Cloud Gateway is API Gateway implementation by Spring Cloud team on top of Spring reactive ecosystem.
- It consists of the following building blocks:
 - **Route:** Route is the basic building block of the gateway. It consists of ID, Destination URI and Collection of predicates and a collection of filters.
 - A route is matched if aggregate predicate is true.
 - **Predicate:** This is like Java 8 Function Predicate. Using this functionality we can match HTTP request, such as headers , URL, cookies or parameters.
 - **Filter:** These are instances Spring Framework GatewayFilter. Using this we can modify the request or response as per the requirement.

Spring Cloud Gateway Architecture



1. Gateway Client

- This is the client application who sends the Http Request to a destination microservice that is running behind the API Gateway.

2. Gateway Handler Mapping

- Determines that request matches a configured route and it's send to the Gateway Web Handler.

3. Gateway Web Handler

- Will take the request through a set of filters configured for the Route.

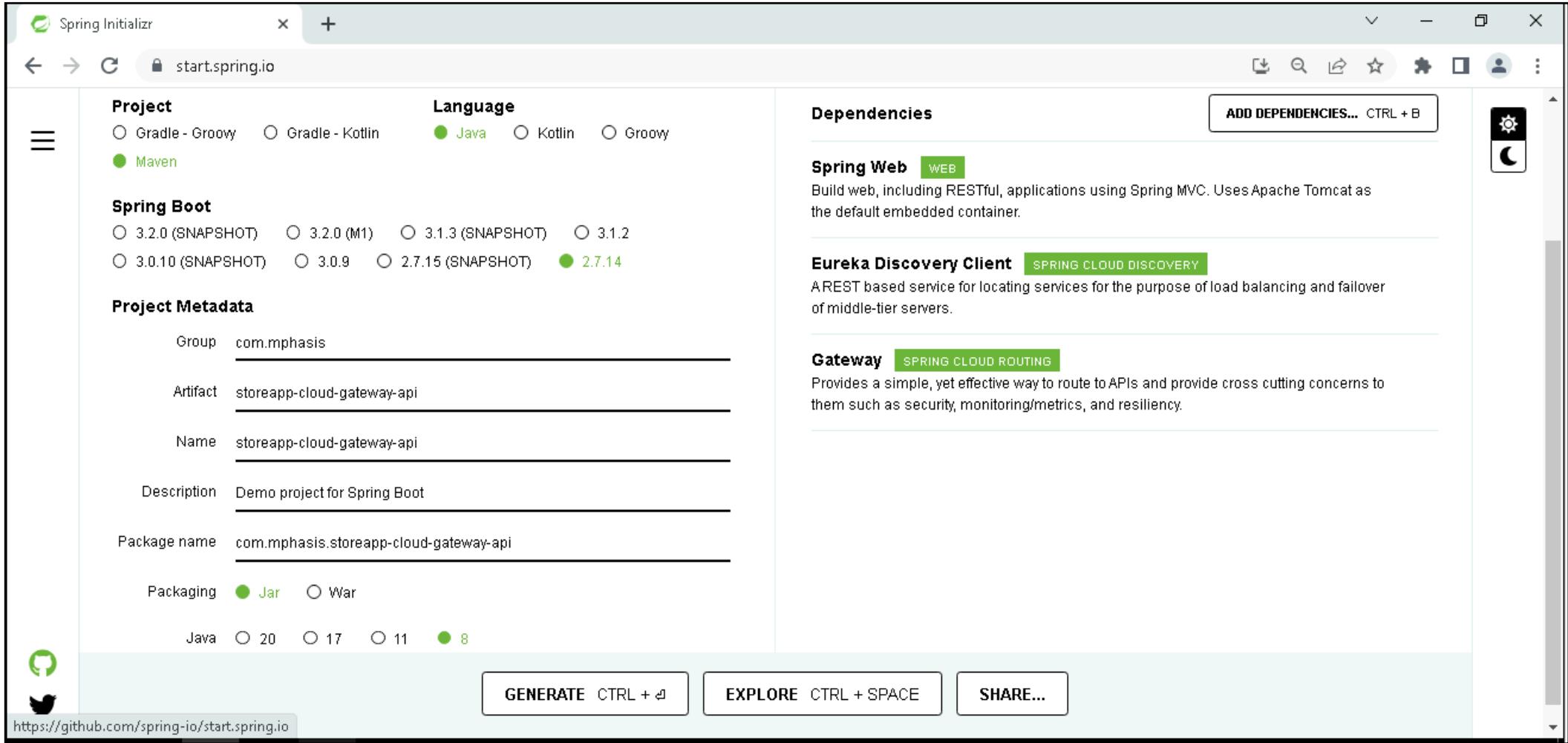
4. Proxied Service

- Filter can run logic both before and after the Proxied request is made. There can be pre-filters and post-filters.



Steps for implementing Spring Cloud Gateway

1. Create a new Project with Spring Web, Eureka Discovery Client and Gateway starters.



The screenshot shows the Spring Initializr web interface at start.spring.io. The project configuration is as follows:

- Project:** Maven
- Language:** Java
- Spring Boot:** 2.7.14
- Project Metadata:**
 - Group: com.mphasis
 - Artifact: storeapp-cloud-gateway-api
 - Name: storeapp-cloud-gateway-api
 - Description: Demo project for Spring Boot
 - Package name: com.mphasis.storeapp-cloud-gateway-api
 - Packaging: Jar
- Dependencies:**
 - Spring Web** (selected): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
 - Eureka Discovery Client**: SPRING CLOUD DISCOVERY: REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.
 - 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.

At the bottom, there are links for [GENERATE](https://github.com/spring-io/start.spring.io), [EXPLORE](#), and [SHARE...](#).



Automatic Mapping of Gateway Routes

2. Add **@EnableEurekaClient** in the Application class.
3. In application.properties file, enable the automatic mapping of gateway routes and add the application name and eureka client serviceUrl.

```
application.properties
1
2server.port=8082
3spring.application.name=product-service-proxy
4
5eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
6
7spring.cloud.gateway.discovery.locator.enabled=true
8spring.cloud.gateway.discovery.locator.lower-case-service-id=true
```

4. Start the storeapp-cloud-gateway-api.



Steps for implementing Spring Cloud Gateway

5. Check the proxy running instances is also registered with the Eureka Server.

The screenshot shows the Eureka dashboard interface. At the top, there's a header bar with tabs and a search bar containing 'localhost'. Below the header, the title 'DS Replicas' is displayed. A search input field contains 'localhost'. The main content area has a heading 'Instances currently registered with Eureka'. A table lists two instances:

Application	AMIs	Availability Zones	Status
PRODUCT-SERVICE	n/a (1)	(1)	UP (1) - host.docker.internal:product-service:0
PRODUCT-SERVICE-PROXY	n/a (1)	(1)	UP (1) - host.docker.internal:product-service-proxy:8082

Below this, there's a section titled 'General Info' with a table:

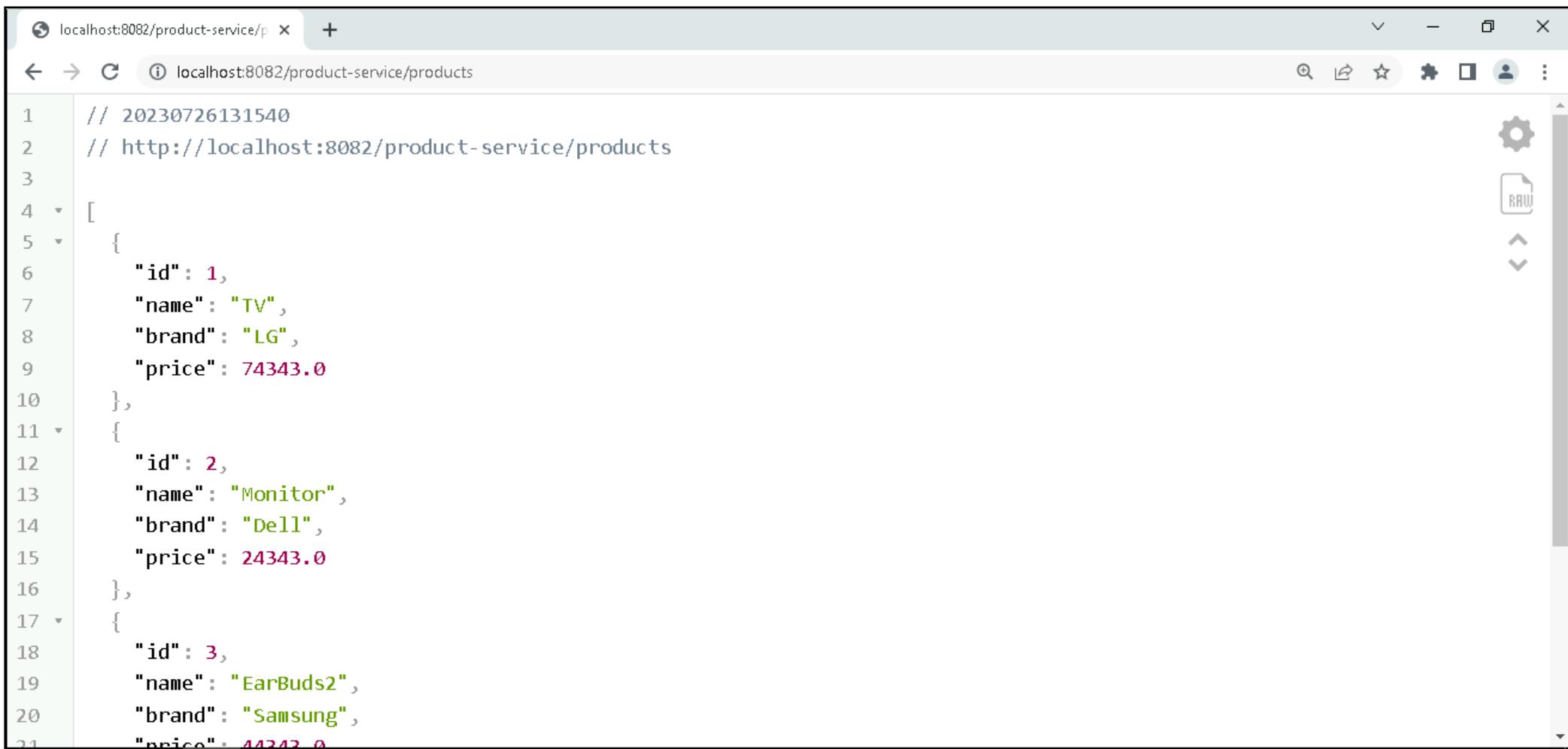
Name	Value
total-avail-memory	306mb
num-of-cpus	4
current-memory-	161mb (52%)

A tooltip for the 'current-memory-' row contains the command: 'Command Prompt - docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.12-management'.



Steps for implementing Spring Cloud Gateway

6. Test the Proxy: <http://localhost:8082/product-service/products>



The screenshot shows a browser window with the URL `localhost:8082/product-service/products` in the address bar. The page content displays a JSON array of product data, with line numbers on the left side.

```
1 // 20230726131540
2 // http://localhost:8082/product-service/products
3 [
4   {
5     "id": 1,
6     "name": "TV",
7     "brand": "LG",
8     "price": 74343.0
9   },
10  {
11    "id": 2,
12    "name": "Monitor",
13    "brand": "Dell",
14    "price": 24343.0
15  },
16  {
17    "id": 3,
18    "name": "EarBuds2",
19    "brand": "Samsung",
20    "price": 44343.0
21  }
```



Manually Configuring API Gateway Routes with Netflix Eureka Discovery Service

- Let's configure API Gateway Routes manually. In application.properties file, add the routes and build-in filters.

```
application.properties ✘  
1  
2 server.port=8082  
3 spring.application.name=product-service-proxy  
4  
5 eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka  
6  
7 #spring.cloud.gateway.discovery.locator.enabled=true  
8 #spring.cloud.gateway.discovery.locator.lower-case-service-id=true  
9  
10 spring.cloud.gateway.routes[0].id=productServiceModule  
11 spring.cloud.gateway.routes[0].uri=lb://product-service  
12 spring.cloud.gateway.routes[0].predicates[0]=Path=/products/**  
13 spring.cloud.gateway.routes[0].predicates[1]=Method=GET  
14 spring.cloud.gateway.routes[0].filters[0]=RemoveRequestHeader=Cookie  
15
```



Steps for implementing Spring Cloud Gateway

8. Check the proxy running instances is also registered with the Eureka Server.

Eureka

localhost:8761

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
PRODUCT-SERVICE	n/a (1)	(1)	UP (1) - host.docker.internal:product-service:0
PRODUCT-SERVICE-PROXY	n/a (1)	(1)	UP (1) - host.docker.internal:product-service-proxy:8082

General Info

Name	Value
total-avail-memory	306mb
num-of-cpus	4
current-memory-	161mb (52%)

Command Prompt - docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.12-management



Steps for implementing Spring Cloud Gateway

9. Test the Proxy: <http://localhost:8082/products>

```
// 20230726132220
// http://localhost:8082/products
[
  {
    "id": 1,
    "name": "TV",
    "brand": "LG",
    "price": 74343.0
  },
  {
    "id": 2,
    "name": "Monitor",
    "brand": "Dell",
    "price": 24343.0
  },
  {
    "id": 3,
    "name": "EarBuds2",
    "brand": "Samsung",
    "price": 44343.0
  }
]
```

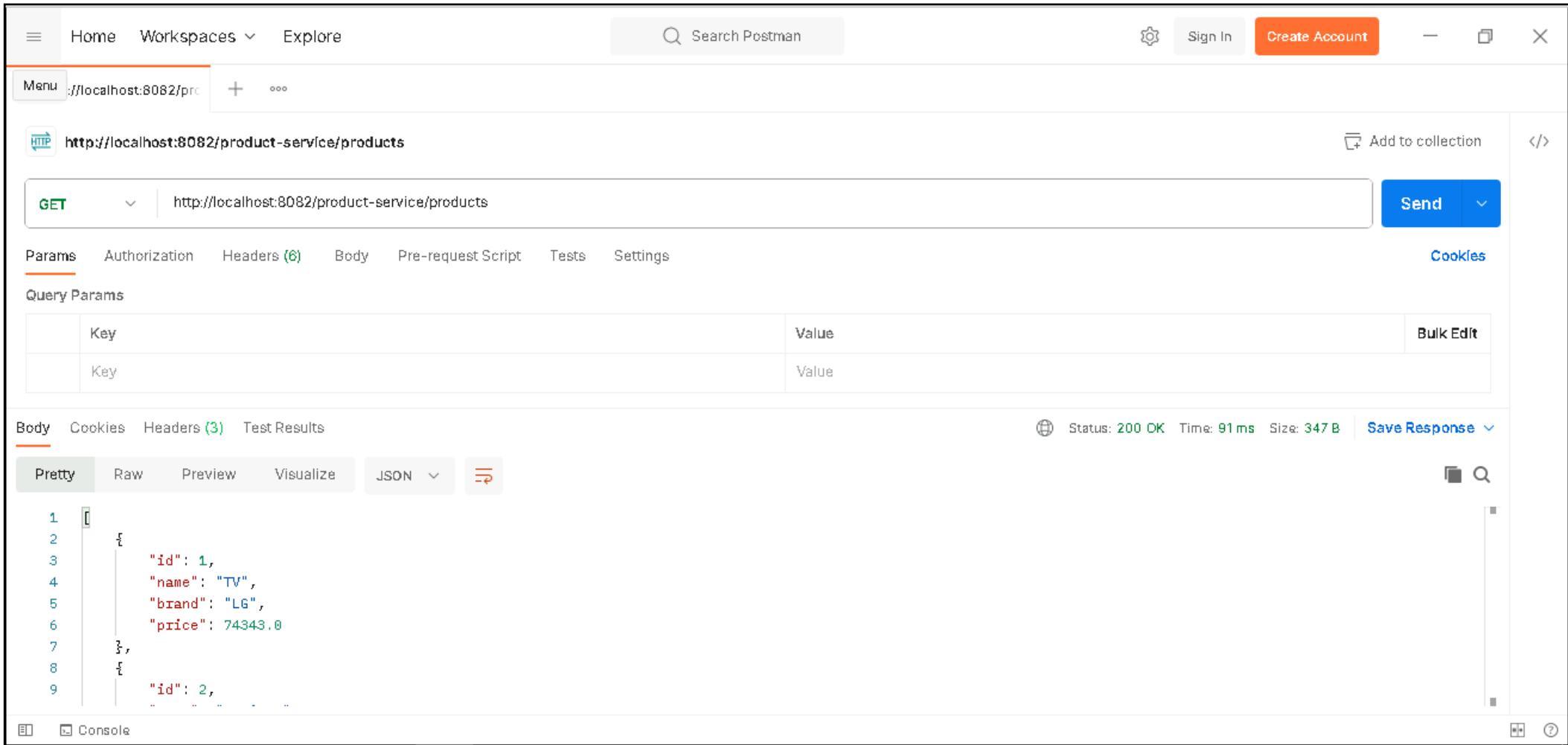
10. Let's configure Dynamically reload route configuration. In application.properties file, add the RewritePath.

```
application.properties ✘  
1  
2 server.port=8082  
3 spring.application.name=product-service-proxy  
4  
5 eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka  
6  
7 #spring.cloud.gateway.discovery.locator.enabled=true  
8 #spring.cloud.gateway.discovery.locator.lower-case-service-id=true  
9  
10 spring.cloud.gateway.routes[0].id=productServiceModule  
11 spring.cloud.gateway.routes[0].uri=lb://product-service  
12 spring.cloud.gateway.routes[0].predicates[0]=Path=/product-service/products/**  
13 spring.cloud.gateway.routes[0].predicates[1]=Method=GET  
14 spring.cloud.gateway.routes[0].filters[0]=RemoveRequestHeader=Cookie  
15  
16 #spring.cloud.gateway.routes[0].filters[1]=RewritePath=/product-service/products, /products  
17  
18 #you can also use reg-ex  
19 spring.cloud.gateway.routes[0].filters[1]=RewritePath=/product-service/(?<segment>.*), /${segment}  
20
```



Steps for implementing Spring Cloud Gateway

11. Test the Proxy from Postman: <http://localhost:8082/product-service/products>



The screenshot shows the Postman application interface. At the top, there's a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options. Below the header, a URL bar shows 'Menu /localhost:8082/pr...' and a collection name 'http://localhost:8082/product-service/products'. A 'GET' request is selected, pointing to 'http://localhost:8082/product-service/products'. The 'Params' tab is active. In the 'Query Params' section, there are two rows: one with 'Key' and 'Value' both empty, and another with 'Key' empty and 'Value' also empty. Below this, tabs for 'Body', 'Cookies', 'Headers (3)', and 'Test Results' are visible. The 'Body' tab is selected, showing a JSON response. The response body is:

```
1 [ { 2   "id": 1, 3     "name": "TV", 4     "brand": "LG", 5     "price": 74343.0 6   }, 7   { 8     "id": 2, 9     "name": "Laptop", 10    "brand": "Dell", 11    "price": 12345.0 12   } ]
```

At the bottom, there are tabs for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'JSON'. On the right side of the preview area, there are icons for copy, search, and refresh. The status bar at the bottom shows 'Status: 200 OK', 'Time: 91 ms', 'Size: 347 B', and a 'Save Response' button.

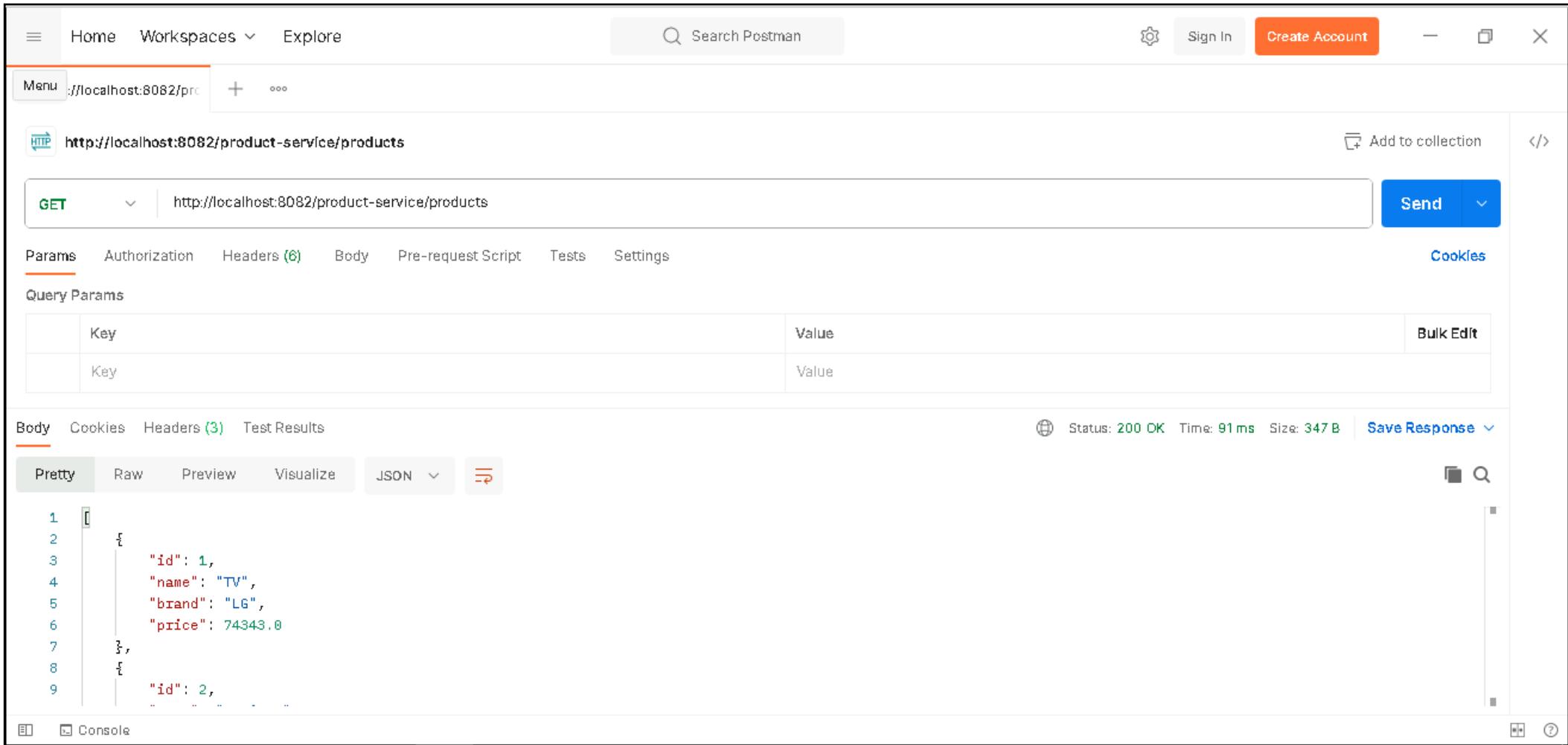
12. Let's configure Automatic & Manual Routing:

```
application.properties ✘
1
2server.port=8082
3spring.application.name=product-service-proxy
4
5eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
6
7#you can have automatic routing
8spring.cloud.gateway.discovery.locator.enabled=true
9spring.cloud.gateway.discovery.locator.lower-case-service-id=true
10
11spring.cloud.gateway.routes[0].id=productServiceModule
12spring.cloud.gateway.routes[0].uri=lb://product-service
13spring.cloud.gateway.routes[0].predicates[0]=Path=/products/**
14spring.cloud.gateway.routes[0].predicates[1]=Method=GET
15spring.cloud.gateway.routes[0].filters[0]=RemoveRequestHeader=Cookie
16
17#spring.cloud.gateway.routes[0].filters[1]=RewritePath=/product-service/products, /products
18
19#you can also use reg-ex
20spring.cloud.gateway.routes[0].filters[1]=RewritePath=/product-service/(?<segment>.*), /$\\{segment}
21
```



Steps for implementing Spring Cloud Gateway

13. Test the Proxy from Postman: <http://localhost:8082/product-service/products>



The screenshot shows the Postman application interface. At the top, there's a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options. Below the header, a URL bar shows 'Menu /localhost:8082/pr...' and a collection name 'http://localhost:8082/product-service/products'. A 'GET' request is selected, pointing to 'http://localhost:8082/product-service/products'. The 'Params' tab is active. In the 'Query Params' section, there are two rows: one with 'Key' and 'Value' both empty, and another with 'Key' empty and 'Value' also empty. Below this, tabs for 'Body', 'Cookies', 'Headers (3)', and 'Test Results' are visible. The 'Pretty' tab is selected in the preview area, which displays the following JSON response:

```
1 [ { 2   "id": 1, 3     "name": "TV", 4     "brand": "LG", 5     "price": 74343.0 6   }, 7   { 8     "id": 2, 9     "name": "Laptop", 10    "brand": "Dell", 11    "price": 12345.0 12   } ]
```

The status bar at the bottom indicates 'Status: 200 OK'.



Steps for implementing Spring Cloud Gateway

14. Test the Proxy from Postman: <http://localhost:8082/products>

The screenshot shows the Postman application interface. At the top, there is a navigation bar with 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and account options 'Sign In' and 'Create Account'. Below the navigation bar, a list of requests is shown, with one request highlighted: 'GET http://localhost:8082/products'. The main workspace shows a 'GET' request to 'http://localhost:8082/products'. The 'Params' tab is selected, showing two query parameters: 'Key' and 'Value'. The 'Body' tab is selected, displaying a JSON response. The response is a list of products:

```
1 [  
2 {  
3   "id": 1,  
4   "name": "TV",  
5   "brand": "LG",  
6   "price": 74343.0  
7 },  
8 {  
9   "id": 2,  
..
```



Built-in Predicates and Filters Factories

Build-in Predicate Factories:

- <https://cloud.spring.io/spring-cloud-gateway/reference/html/#gateway-request-predicates-factories>

Gateway Filters:

- <https://cloud.spring.io/spring-cloud-gateway/reference/html/#gatewayfilter-factories>
- <https://cloud.spring.io/spring-cloud-gateway/reference/html/#global-filters>
- <https://cloud.spring.io/spring-cloud-gateway/reference/html/#httpheadersfilters>



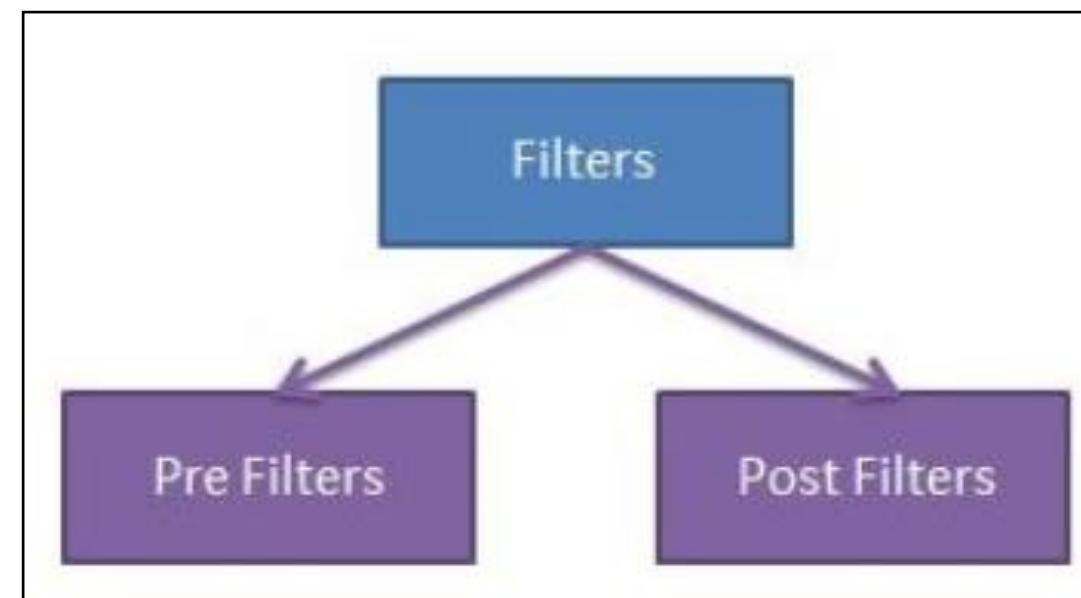
Day - 9

Creating Custom Filters



Implementing Spring Cloud Gateway Filters

- Spring Cloud Gateway filters can be classified as
 - Spring Cloud Gateway Pre Filters
 - Spring Cloud Gateway Post Filters





Defining the GatewayFilterFactory

- In order to implement a GatewayFilter, we'll have to implement the GatewayFilterFactory interface.
- Spring Cloud Gateway also provides an abstract class to simplify the process, the **AbstractGatewayFilterFactory** class.



Defining the GatewayFilterFactory

```
LoggingGatewayFilterFactory.java ✘
1 package com.mphasis.filters;
2
3+import org.slf4j.Logger;◻
13
14 @Component
15 public class LoggingGatewayFilterFactory extends AbstractGatewayFilterFactory<LoggingGatewayFilterFactory.Config> {
16
17@ 17@ Data
18 @NoArgsConstructor
19 @AllArgsConstructor
20 public static class Config {
21     // Specify your configurations
22     private String baseMessage;
23     private boolean preLogger;
24     private boolean postLogger;
25 }
26
27 private final Logger logger = LoggerFactory.getLogger(LoggingGatewayFilterFactory.class);
28
29@ 29@ public LoggingGatewayFilterFactory() {
30     super(Config.class);
31 }
32 |
```



Defining the GatewayFilterFactory



```
LoggingGatewayFilterFactory.java
32
33@Override
34public GatewayFilter apply(Config config) {
35
36    return (exchange, chain) -> {
37
38        // Pre-processing
39        if (config.isPreLogger()) {
40            logger.info("Pre GatewayFilter logging: " + config.getBaseMessage());
41        }
42
43        return chain.filter(exchange).then(Mono.fromRunnable(() -> {
44            // Post-processing
45            if (config.isPostLogger()) {
46                logger.info("Post GatewayFilter logging: " + config.getBaseMessage());
47            }
48        }));
49    };
50}
51}
52}
53}
54|
```



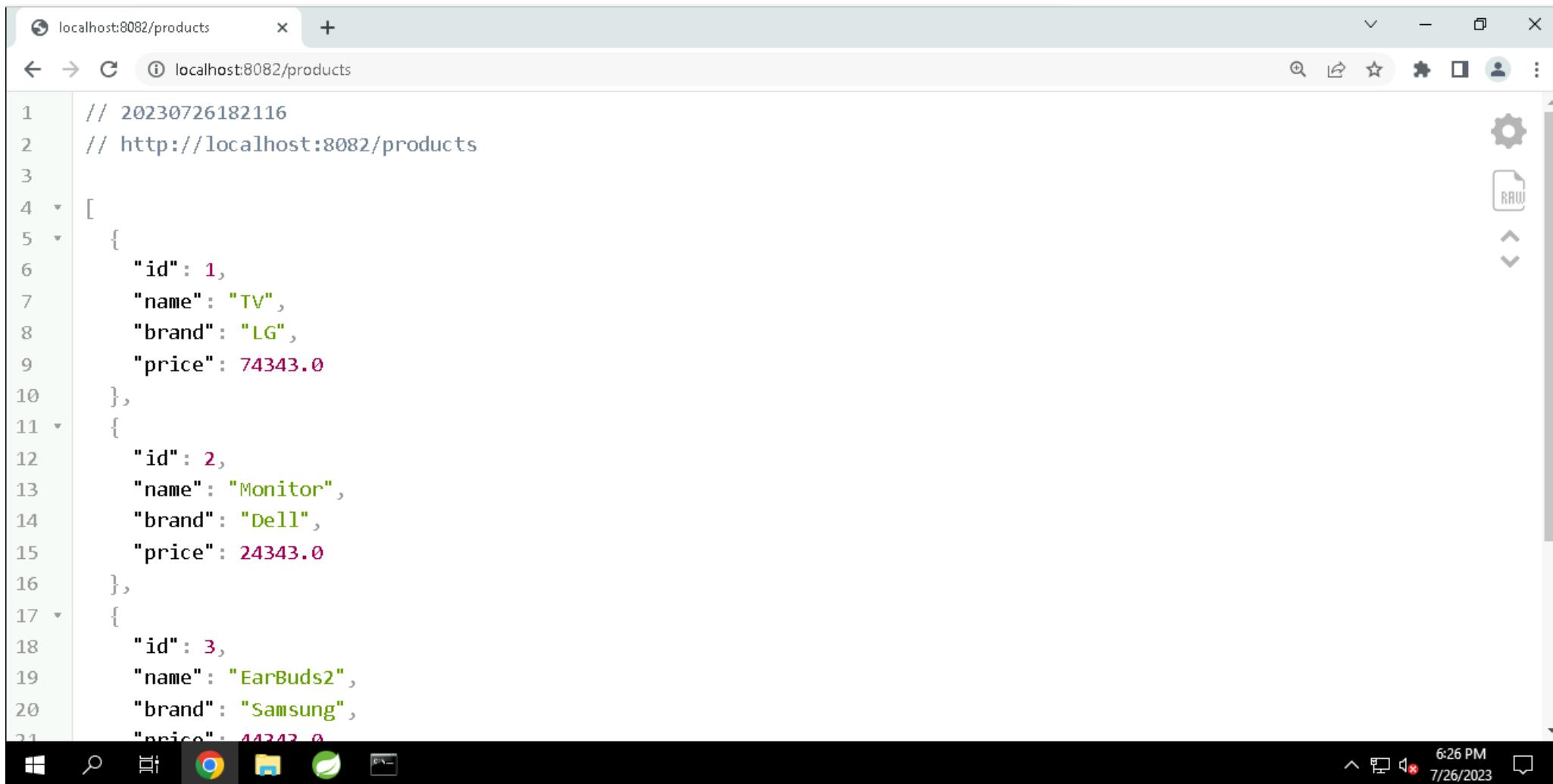
Registering the GatewayFilter with Properties

```
application.properties ✘
4
5 eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
6
7 #you can have automatic routing
8 spring.cloud.gateway.discovery.locator.enabled=true
9 spring.cloud.gateway.discovery.locator.lower-case-service-id=true
10
11spring.cloud.gateway.routes[0].id=productServiceModule
12spring.cloud.gateway.routes[0].uri=lb://product-service
13spring.cloud.gateway.routes[0].predicates[0]=Path=/products/**
14spring.cloud.gateway.routes[0].predicates[1]=Method=GET
15spring.cloud.gateway.routes[0].filters[0]=RemoveRequestHeader=Cookie
16spring.cloud.gateway.routes[0].filters[1]=RewritePath=/product-service/products, /products
17#you can also use reg-ex
18#spring.cloud.gateway.routes[0].filters[1]=RewritePath=/product-service/(?<segment>.*), /${segment}
19
20#register our filter to the route
21spring.cloud.gateway.routes[0].filters[2].name=Logging
22spring.cloud.gateway.routes[0].filters[2].args.baseMessage=My Custom Message
23spring.cloud.gateway.routes[0].filters[2].args.preLogger=true
24spring.cloud.gateway.routes[0].filters[2].args.postLogger=true
25
```



Test the Proxy

- Test the Proxy: <http://localhost:8082/products>



The screenshot shows a web browser window with the URL `localhost:8082/products` in the address bar. The page content displays a JSON array of product objects. The JSON is formatted with line numbers on the left side of the code block.

```
// 20230726182116
// http://localhost:8082/products
[{"id": 1, "name": "TV", "brand": "LG", "price": 74343.0}, {"id": 2, "name": "Monitor", "brand": "Dell", "price": 24343.0}, {"id": 3, "name": "EarBuds2", "brand": "Samsung", "price": 44242.0}]
```



Check the Logs on Console

```
Console ✘ Progress Problems Servers
storeapp-cloud-gateway-api - StoreappCloudGatewayApiApplication [Spring Boot App] C:\Program Files\ojdkbuild\java-1.8.0-openjdk-1.8.0.302-1\bin\javaw.exe (Jul 26, 2023, 6:30:43 PM)
main] com.netflix.discovery.DiscoveryClient . initializing Eureka in region us-east-1
main] c.n.d.s.r.aws.ConfigClusterResolver : Resolving eureka endpoints via configuration
main] com.netflix.discovery.DiscoveryClient : Disable delta property : false
main] com.netflix.discovery.DiscoveryClient : Single vip registry refresh property : null
main] com.netflix.discovery.DiscoveryClient : Force full registry fetch : false
main] com.netflix.discovery.DiscoveryClient : Application is null : false
main] com.netflix.discovery.DiscoveryClient : Registered Applications size is zero : true
main] com.netflix.discovery.DiscoveryClient : Application version is -1: true
main] com.netflix.discovery.DiscoveryClient : Getting all instance registry info from the eureka server
main] com.netflix.discovery.DiscoveryClient : The response status is 200
main] com.netflix.discovery.DiscoveryClient : Starting heartbeat executor: renew interval is: 30
main] c.n.discovery.InstanceInfoReplicator : InstanceInfoReplicator onDemand update allowed rate per min is 4
main] com.netflix.discovery.DiscoveryClient : Discovery Client initialized at timestamp 1690396247172 with initia
main] o.s.c.n.e.s.EurekaServiceRegistry : Registering application PRODUCT-SERVICE-PROXY with eureka with sta
main] com.netflix.discovery.DiscoveryClient : Saw local status change event StatusChangeEvent [timestamp=1690396
replicator-0] com.netflix.discovery.DiscoveryClient : DiscoveryClient_PRODUCT-SERVICE-PROXY/host.docker.internal:product
replicator-0] com.netflix.discovery.DiscoveryClient : DiscoveryClient_PRODUCT-SERVICE-PROXY/host.docker.internal:product
main] o.s.b.web.embedded.netty.NettyWebServer : Netty started on port 8082
main] o.s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 8082
main] c.m.StoreappCloudGatewayApiApplication : Started StoreappCloudGatewayApiApplication in 3.589 seconds (JVM r
-http-nio-2] c.m.filters.LoggingGatewayFilterFactory : Pre GatewayFilter logging: My Custom Message
-http-nio-2] c.m.filters.LoggingGatewayFilterFactory : Post GatewayFilter logging: My Custom Message
```



Day - 9

Global Filters using Spring Cloud Gateway



Global Filters using Spring Cloud Gateway

- Till now we have created filters which we are applying to a specific route.
- But suppose now we want to apply some filter to all the routes.
- We can do this by creating **Global Filters**.



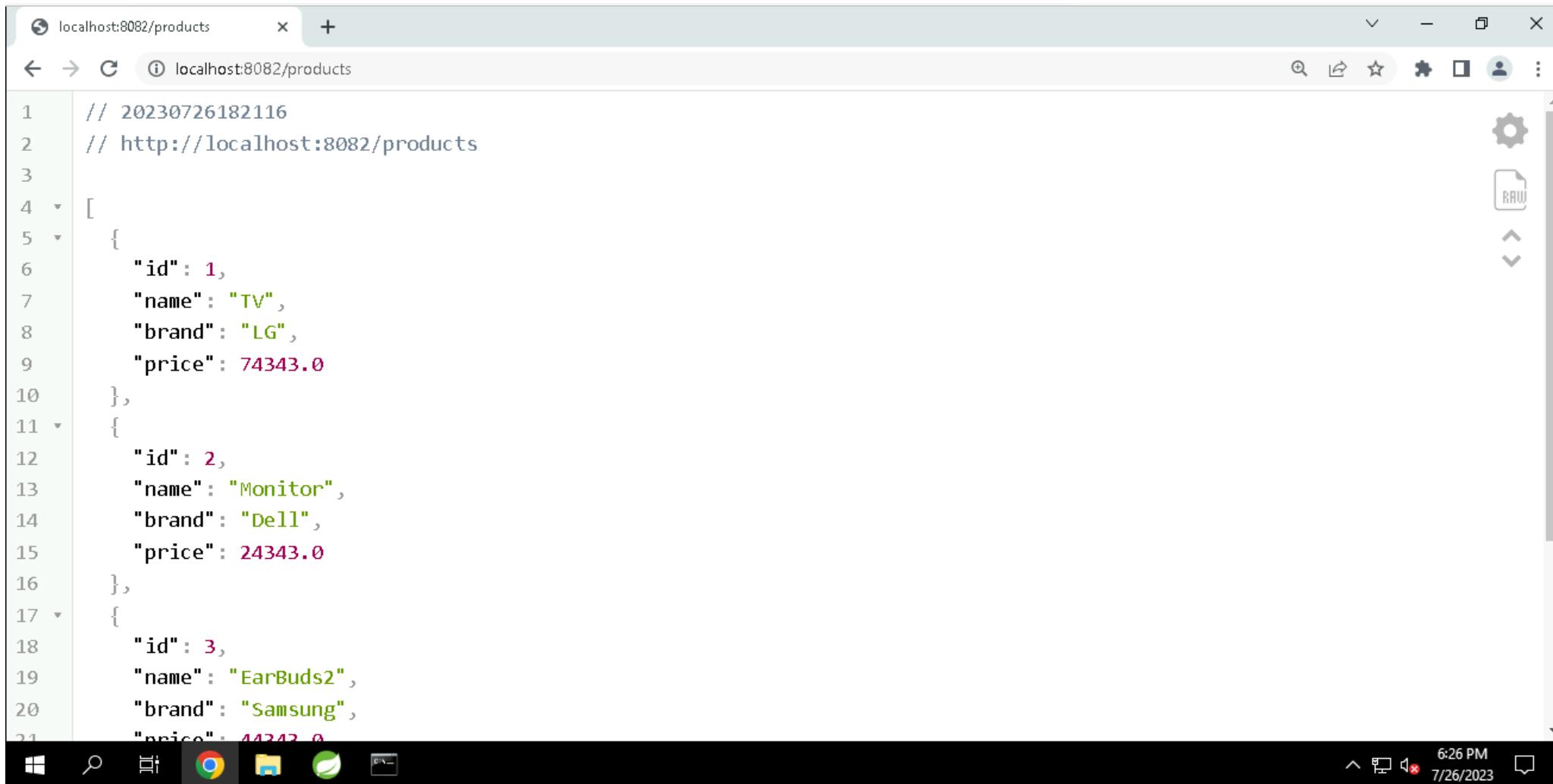
Registering the Global Filters using Property Based Configuration

```
application.properties
1server.port=8082
2spring.application.name=product-service-proxy
3
4eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
5
6#you can have automatic routing
7spring.cloud.gateway.discovery.locator.enabled=true
8spring.cloud.gateway.discovery.locator.lower-case-service-id=true
9
10#register as a global filter
11spring.cloud.gateway.default-filters[0].name=Logging
12spring.cloud.gateway.default-filters[0].args.baseMessage=My Custom Message
13spring.cloud.gateway.default-filters[0].args.preLogger=true
14spring.cloud.gateway.default-filters[0].args.postLogger=true
15
16
17spring.cloud.gateway.routes[0].id=productServiceModule
18spring.cloud.gateway.routes[0].uri=lb://product-service
19spring.cloud.gateway.routes[0].predicates[0]=Path=/products/**
20spring.cloud.gateway.routes[0].predicates[1]=Method=GET
21spring.cloud.gateway.routes[0].filters[0]=RemoveRequestHeader=Cookie
22spring.cloud.gateway.routes[0].filters[1]=RewritePath=/product-service/products, /products
23#you can also use reg-ex
24#spring.cloud.gateway.routes[0].filters[1]=RewritePath=/product-service/(?<segment>.*), /${segment}
```



Test the Proxy

- Test the Proxy: <http://localhost:8082/products>



The screenshot shows a web browser window with the URL `localhost:8082/products` in the address bar. The page content displays a JSON array of three product objects. The products are:

```
// 20230726182116
// http://localhost:8082/products
[{"id": 1, "name": "TV", "brand": "LG", "price": 74343.0}, {"id": 2, "name": "Monitor", "brand": "Dell", "price": 24343.0}, {"id": 3, "name": "EarBuds2", "brand": "Samsung", "price": 44242.0}]
```

The browser interface includes standard navigation buttons (back, forward, search), a refresh button, and a toolbar with icons for file operations. The status bar at the bottom shows the time as 6:26 PM and the date as 7/26/2023.



Check the Logs on Console

```
Console ✘ Progress Problems Servers
storeapp-cloud-gateway-api - StoreappCloudGatewayApiApplication [Spring Boot App] C:\Program Files\ojdkbuild\java-1.8.0-openjdk-1.8.0.302-1\bin\javaw.exe (Jul 26, 2023, 6:30:43 PM)
main] com.netflix.discovery.DiscoveryClient : Initializing Eureka in region us-east-1
main] c.n.d.s.r.aws.ConfigClusterResolver : Resolving eureka endpoints via configuration
main] com.netflix.discovery.DiscoveryClient : Disable delta property : false
main] com.netflix.discovery.DiscoveryClient : Single vip registry refresh property : null
main] com.netflix.discovery.DiscoveryClient : Force full registry fetch : false
main] com.netflix.discovery.DiscoveryClient : Application is null : false
main] com.netflix.discovery.DiscoveryClient : Registered Applications size is zero : true
main] com.netflix.discovery.DiscoveryClient : Application version is -1: true
main] com.netflix.discovery.DiscoveryClient : Getting all instance registry info from the eureka server
main] com.netflix.discovery.DiscoveryClient : The response status is 200
main] com.netflix.discovery.DiscoveryClient : Starting heartbeat executor: renew interval is: 30
main] c.n.discovery.InstanceInfoReplicator : InstanceInfoReplicator onDemand update allowed rate per min is 4
main] com.netflix.discovery.DiscoveryClient : Discovery Client initialized at timestamp 1690396247172 with initia
main] o.s.c.n.e.s.EurekaServiceRegistry : Registering application PRODUCT-SERVICE-PROXY with eureka with sta
main] com.netflix.discovery.DiscoveryClient : Saw local status change event StatusChangeEvent [timestamp=1690396
main] com.netflix.discovery.DiscoveryClient : DiscoveryClient_PRODUCT-SERVICE-PROXY/host.docker.internal:product
replicator-0] com.netflix.discovery.DiscoveryClient : DiscoveryClient_PRODUCT-SERVICE-PROXY/host.docker.internal:product
replicator-0] com.netflix.discovery.DiscoveryClient : Netty started on port 8082
main] o.s.b.web.embedded.netty.NettyWebServer : Updating port to 8082
main] o.s.c.n.e.s.EurekaAutoServiceRegistration : Started StoreappCloudGatewayApiApplication in 3.589 seconds (JVM r
main] c.m.StoreappCloudGatewayApiApplication : Pre GatewayFilter logging: My Custom Message
-http-nio-2] c.m.filters.LoggingGatewayFilterFactory : Post GatewayFilter logging: My Custom Message
```

- What is an API Gateway?
- Spring Cloud Gateway Architecture
- Configuring routes in Spring Cloud Gateway
- Dynamically reload route configuration
- Built-in Predicates and Filters Factories
- Creating Custom Filters
- Global Filters using Spring Cloud Gateway
- Netflix Eureka Discovery Service Implementation



Day - 10

- Event Driven Microservices with Spring Cloud Stream and RabbitMQ
- Stream Processing Using Spring Cloud Data Flow
- Autoscaling microservices
- Scaling microservices with Spring Cloud
- Understanding the concept of autoscaling
- The benefits of autoscaling
- Different autoscaling models
- Autoscaling Approaches



Day - 10

Event Driven Microservices with Spring Cloud Stream and RabbitMQ



What is RabbitMQ?

- RabbitMQ is a message queue software (message broker/queue manager) that acts as an intermediary platform where different applications can send and receive messages.
- RabbitMQ originally implements the **Advanced Message Queuing Protocol (AMQP)**. But now RabbitMQ also supports several other API protocols such as STOMP, MQTT, and HTTP.

- **Producer:** Application that sends the messages.
- **Consumer:** Application that receives the messages.
- **Queue:** Buffer that stores messages.
- **Message:** Information that is sent from the producer to a consumer through RabbitMQ.
- **Connection:** A connection is a TCP connection between your application and the RabbitMQ broker.
- **Channel:** A channel is a virtual connection inside a connection. When you are publishing or consuming messages from a queue - it's all done over a channel.
- **Exchange:** Receives messages from producers and pushes them to queues depending on rules defined by the exchange type. To receive messages, a queue needs to be bound to at least one exchange.

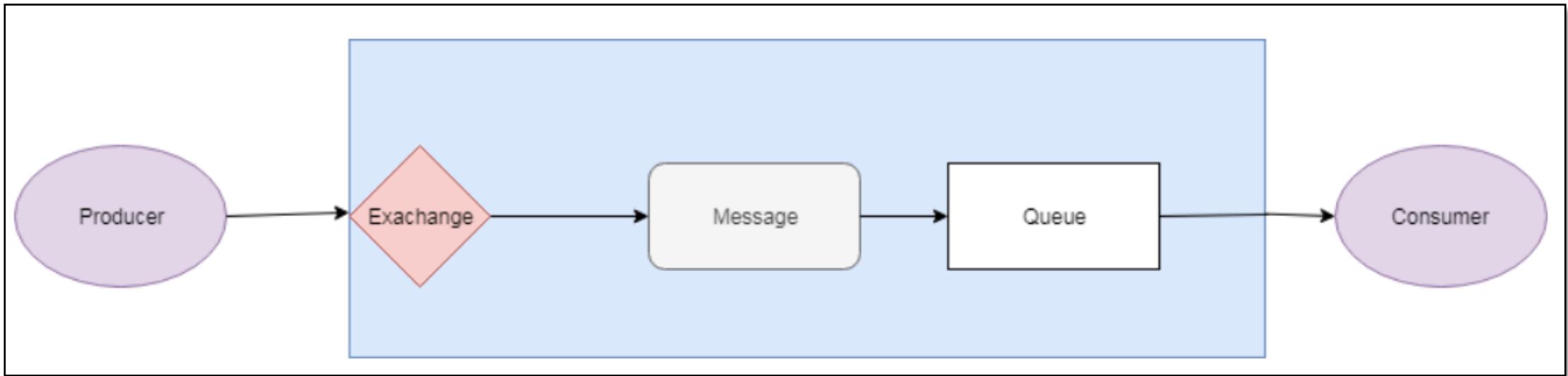
- **Binding:** A binding is a link between a queue and an exchange.
- **Routing key:** The routing key is a key that the exchange looks at to decide how to route the message to queues. The routing key is like an address for the message.
- **AMQP:** AMQP (Advanced Message Queuing Protocol) is the protocol used by RabbitMQ for messaging.
- **Users:** It is possible to connect to RabbitMQ with a given username and password. Every user can be assigned permissions such as rights to read, write and configure privileges within the instance.



How does RabbitMQ Works?

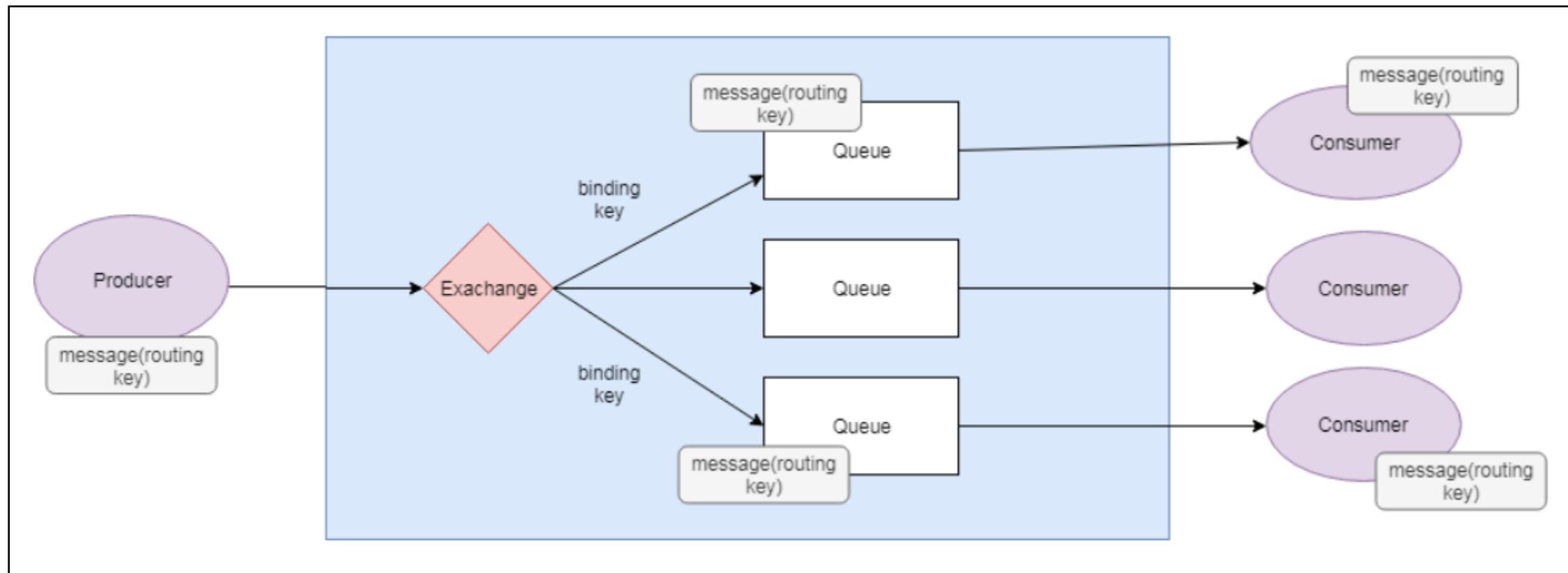
- **Producers** send/publish the messages to the broker -> **Consumers** receive the messages from the broker. **RabbitMQ** acts a communication middleware between both producers and consumers even if they run on different machines.
- While the producer is sending a message to the queue, it will not be sent directly, but sent using the exchange instead. The design below demonstrates how are the main three components are connected to each other.
- The exchanges agents that are responsible for routing the messages to different queues. So that the message can be received from the producer to the exchange and then again forwarded to the queue. This is known as the '**Publishing**' method.
- The message will be picked up and consumed from the queue; this is called '**Consuming**'.

How does RabbitMQ Works?



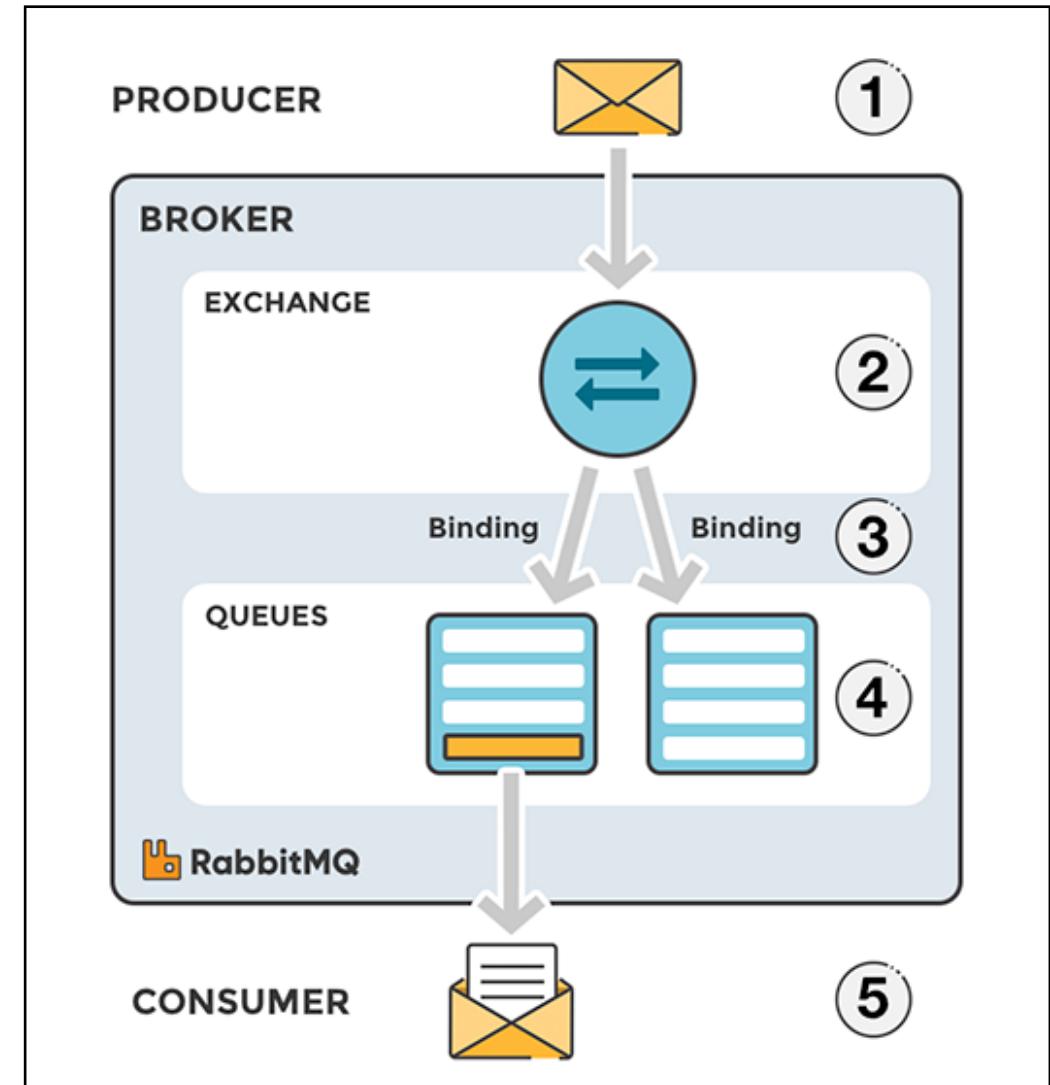
Send Message to Multiple Queues

- By having a more complex application we would have multiple queues. So, the messages will send it in the multiple queues.

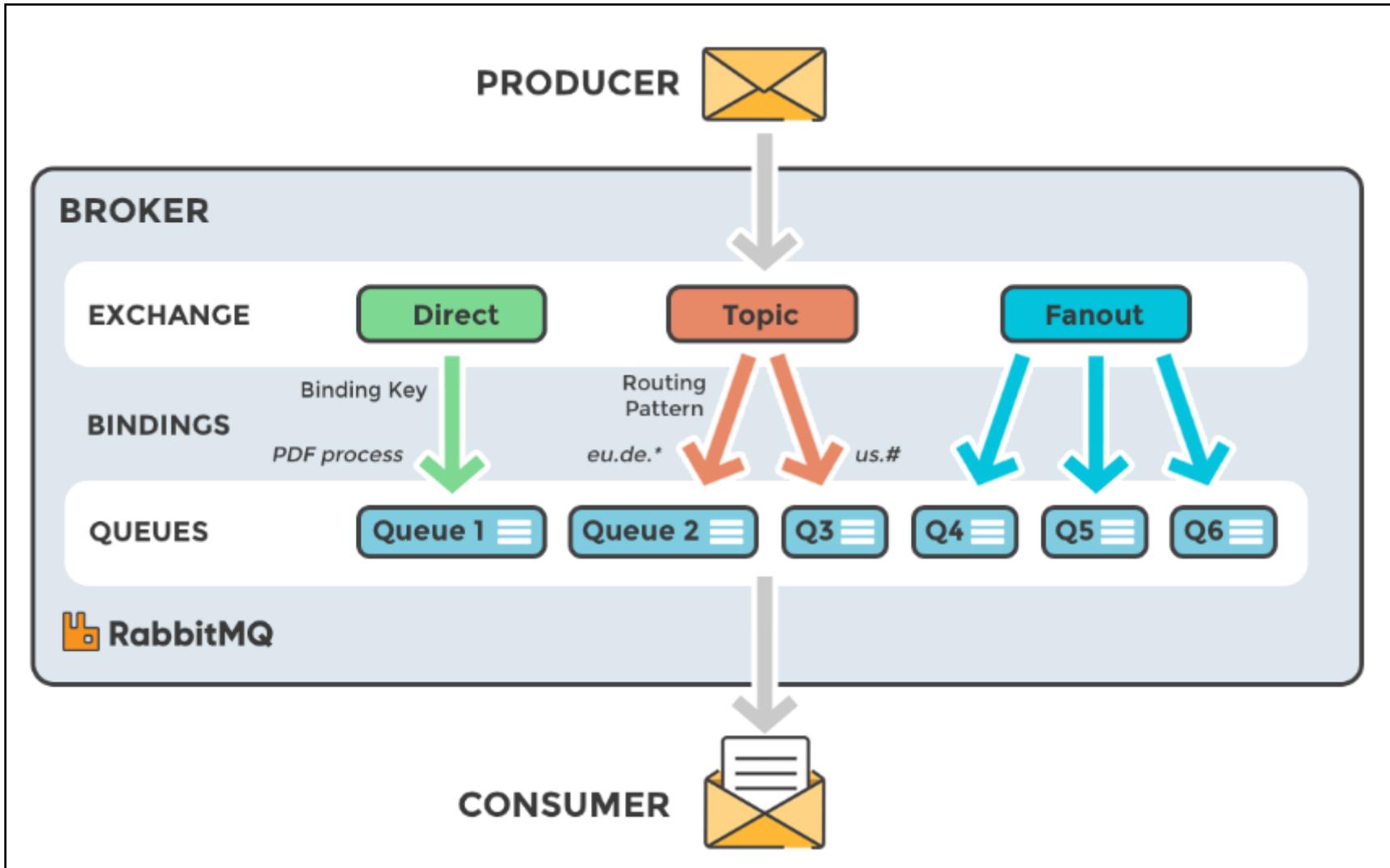


Exchanges

- Messages are not published directly to a queue, instead, the producer sends messages to an exchange.
- An exchange is responsible for the routing of the messages to the different queues.
- An exchange accepts messages from the producer application and routes them to message queues with the help of bindings and routing keys.
- A binding is a link between a queue and an exchange.



Types of Exchanges





What is Spring Cloud Stream?

- Spring Cloud Stream is a framework for building highly scalable **event-driven microservices** connected with shared messaging systems.
- The framework provides a flexible programming model built on already established and familiar Spring idioms and best practices, including support for persistent pub/sub semantics, consumer groups, and stateful partitions.
- Spring Cloud Stream supports a variety of **binder implementations**, and the following are few of them.
 - RabbitMQ
 - Apache Kafka
 - Kafka Streams
 - Amazon Kinesis



Day - 10

Spring Cloud Stream – Publish Message to RabbitMQ



Installing RabbitMQ

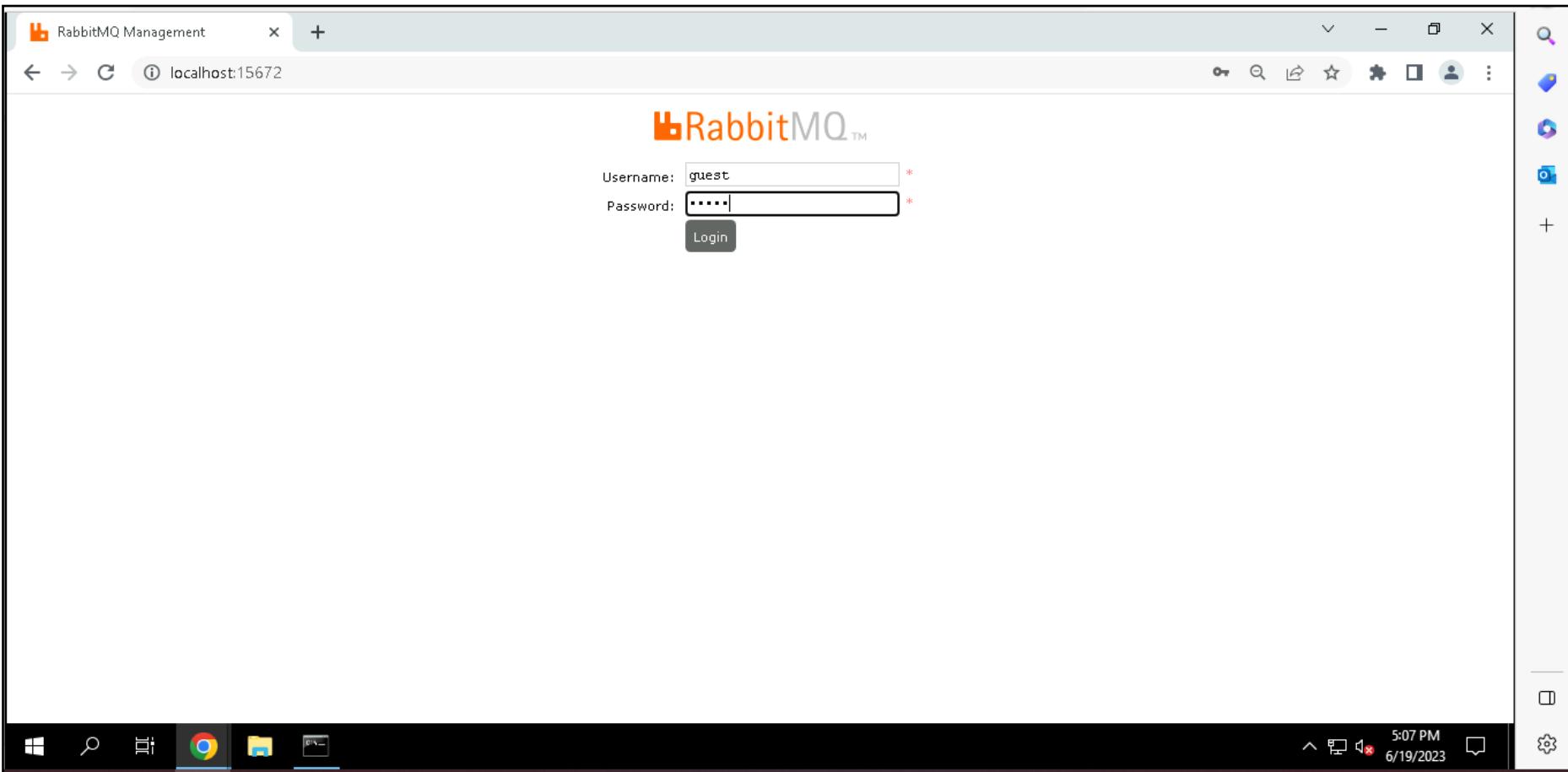
1. To Download and Run Rabbit MQ, go to <https://rabbitmq.com>
2. Click on Download + Installation button.
3. Open the command prompt and execute the below docker command:

```
# Latest RabbitMQ 3.12
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.12-management
```



Installing RabbitMQ

4. Access the RabbitMQ dashboard via <http://localhost:15672>



5. The default username and password of RabbitMQ is guest.



Spring Cloud Stream – Publish Message to RabbitMQ

6. Create a new Project i.e., employee-registration-producer:

The screenshot shows the Spring Initializr web application at start.spring.io. The project is set up for a Maven build with Java 27.14 as the language. The dependencies section includes Cloud Stream (Spring Cloud Messaging), Spring Web (Web), Lombok (Developer Tools), and Spring for RabbitMQ (Messaging). The project metadata includes Group (com.mphasis), Artifact (employee-registration-producer), Name (employee-registration-producer), Description (Demo project for Spring Boot), Package name (com.mphasis), and Packaging (Jar). The Java version selected is 27. The bottom of the page features buttons for GENERATE (CTRL + ⌘), EXPLORE (CTRL + SPACE), and SHARE... .

Project Language
○ Gradle - Groovy ○ Gradle - Kotlin ● Java ○ Kotlin ○ Groovy
Maven

Spring Boot
○ 3.2.0 (SNAPSHOT) ○ 3.2.0 (M1) ○ 3.1.3 (SNAPSHOT) ○ 3.1.2
○ 3.0.10 (SNAPSHOT) ○ 3.0.9 ○ 2.7.15 (SNAPSHOT) ● 2.7.14

Project Metadata
Group: com.mphasis
Artifact: employee-registration-producer
Name: employee-registration-producer
Description: Demo project for Spring Boot
Package name: com.mphasis
Packaging: ● Jar ○ War
Java: ○ 20 ○ 17 ○ 11 ● 8

Dependencies ADD DEPENDENCIES... CTRL + B

Cloud Stream SPRING CLOUD MESSAGING
Framework for building highly scalable event-driven microservices connected with shared messaging systems (requires a binder, e.g. Apache Kafka, Apache Pulsar, RabbitMQ, or Solace PubSub+).

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Lombok DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.

Spring for RabbitMQ MESSAGING
Gives your applications a common platform to send and receive messages, and your messages a safe place to live until received.

GENERATE CTRL + ⌘ EXPLORE CTRL + SPACE SHARE...



Spring Cloud Stream – Publish Message to RabbitMQ

7. Configure the rabbitmq properties in application.properties file:

The screenshot shows a code editor window titled "application.properties". The file contains the following configuration properties:

```
1 server.port=9080
2
3 spring.rabbitmq.host=localhost
4 spring.rabbitmq.port=5672
5 spring.rabbitmq.username=guest
6 spring.rabbitmq.password=guest
7
8 rabbitmq.queue.name=employeeRegistrationQueue
9 rabbitmq.exchange.name=employeeRegistrationExchange
10 rabbitmq.routing.key=employeeRegistrationRoutingKey
11
12
13|
```

8. Firstly, we will implement the RabbitMQConfig class. It will handle the configuration of the RabbitMq like below.

```
RabbitMQConfig.java ✘
1 package com.mphasis;
2
3+import org.springframework.amqp.core.*;
4
5
10 @Configuration
11 public class RabbitMQConfig {
12
13@Value("${rabbitmq.queue.name}")
14 private String queue;
15
16@Value("${rabbitmq.exchange.name}")
17 private String exchange;
18
19@Value("${rabbitmq.routing.key}")
20 private String routingKey;
21
22@Bean
23 public Queue createQueue() {
24
25     return new Queue(queue);
26 }
27}
```



Spring Cloud Stream – Publish Message to RabbitMQ

```
RabbitMQConfig.java
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

```
    ...
27
28 @Bean
29 public TopicExchange exchange() {
30
31     return new TopicExchange(exchange);
32 }
33
34 @Bean
35 public Binding binding(Queue queue, TopicExchange exchange) {
36
37     return BindingBuilder.bind(queue).to(exchange).with(routingKey);
38 }
39
40 @Bean
41 public SimpleMessageListenerContainer container(ConnectionFactory connectionFactory) {
42
43     SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
44     container.setConnectionFactory(connectionFactory);
45     return container;
46 }
47 }
48 }
```



9. Define the model class – Employee:

```
Employee.java ✘
1 package com.mphasis.domain;
2
3+import com.fasterxml.jackson.annotation.JsonIgnoreProperties;□
8
▲ 9 @Data
10 @NoArgsConstructor
11 @AllArgsConstructor
12 @JsonIgnoreProperties
13 public class Employee {
14
15     private String empId;
16     private String empName;
17 }
18
```



Spring Cloud Stream – Publish Message to RabbitMQ

10. Create RabbitMQ Producer. We use RabbitTemplate to convert and send a message using RabbitMQ.

```
EmployeeRegistrationProducer.java
1 package com.mphasis.producer;
2
3+ import org.springframework.amqp.rabbit.core.RabbitTemplate;□
4
5
6 @Service
7 public class EmployeeRegistrationProducer {
8
9
10    @Value("${rabbitmq.exchange.name}")
11    private String exchange;
12
13    @Value("${rabbitmq.routing.key}")
14    private String routingKey;
15
16    @Autowired
17    private RabbitTemplate rabbitTemplate;
18
19    public void sendMessage(Employee employee) {
20
21        rabbitTemplate.convertAndSend(exchange, routingKey, employee);
22    }
23
24 }
```

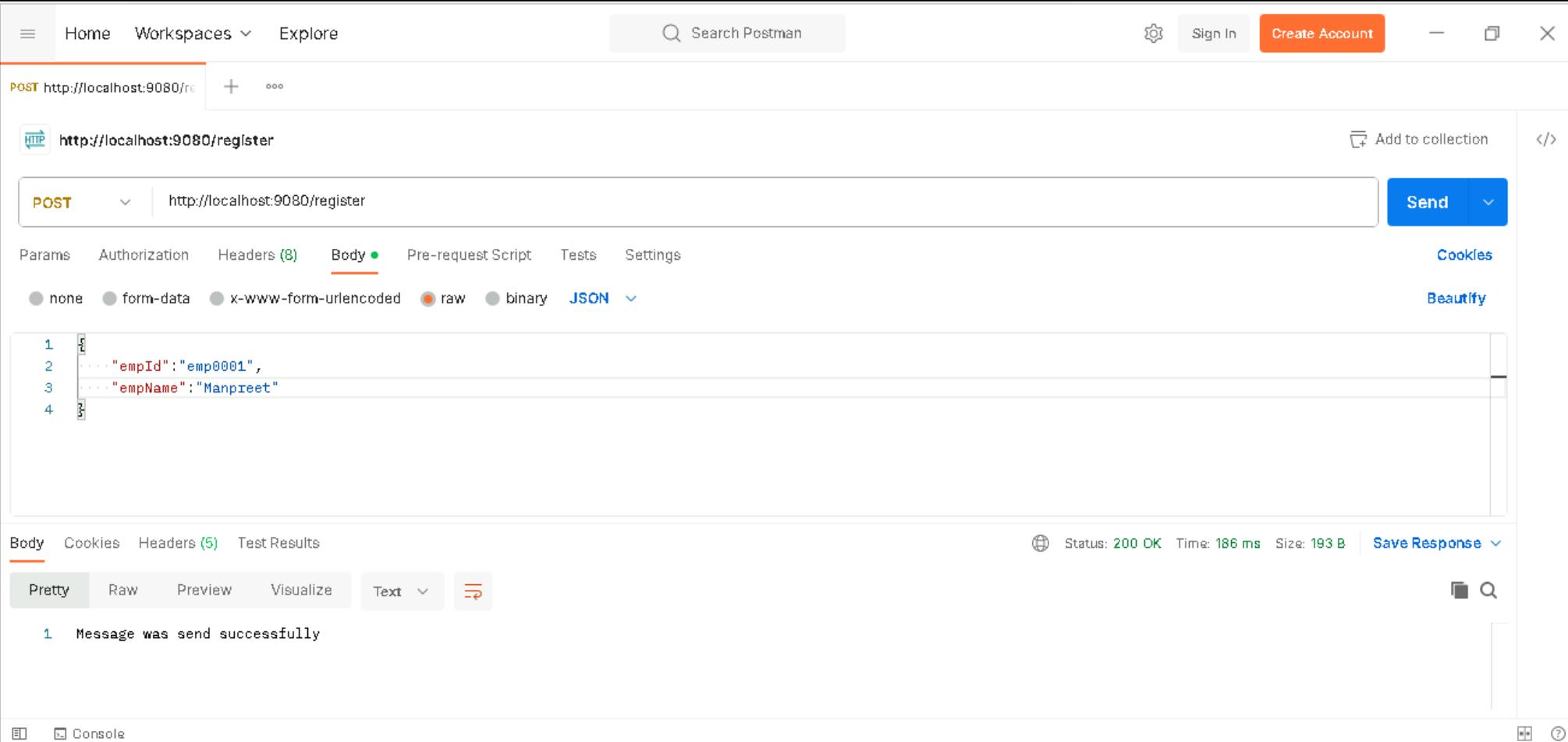
11. Create EmployeeRegistrationController:

```
J EmployeeRegistrationController.java ✘
1 package com.mphasis.controller;
2
3+import org.springframework.beans.factory.annotation.Autowired;□
10
11 @RestController
12 public class EmployeeRegistrationController {
13
14@  @Autowired
15  private EmployeeRegistrationProducer employeeRegistrationProducer;
16
17@  @PostMapping(value = "register")
18  public String orderFood(@RequestBody Employee employee) {
19
20      employeeRegistrationProducer.sendMessage(employee);
21      return "Message was send successfully";
22  }
23 }
```



Spring Cloud Stream – Publish Message to RabbitMQ

12. Start your application. Open a Postman or any other tool and send a message to the producer:

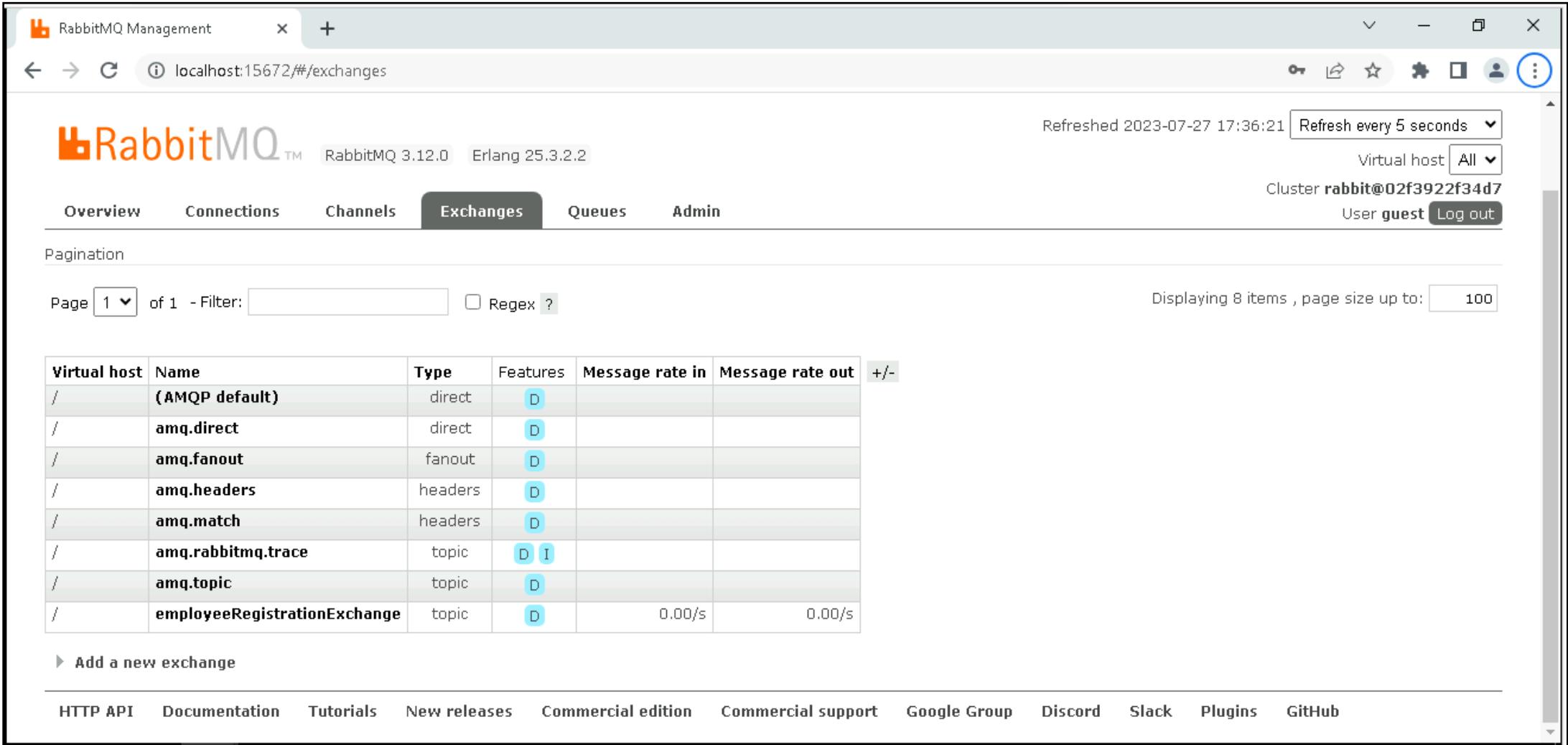


The screenshot shows the Postman interface with the following details:

- Request URL:** POST <http://localhost:9080/register>
- Method:** POST
- Headers:** (8)
- Body:** (highlighted in green)
- JSON Content:**

```
1 {  
2   ... "empId": "emp0001",  
3   ... "empName": "Manpreet"  
4 }
```
- Status:** 200 OK
- Time:** 186 ms
- Size:** 193 B
- Message:** Message was send successfully

13. Review the Exchanges in RabbitMQ Dashboard:



The screenshot shows the RabbitMQ Management interface at localhost:15672/#/exchanges. The title bar says "RabbitMQ Management". The top right shows "Refreshed 2023-07-27 17:36:21" and "Refresh every 5 seconds". It also indicates "Virtual host All" and "Cluster rabbit@02f3922f34d7" with a user "guest" logged in.

The navigation menu includes "Overview", "Connections", "Channels", "Exchanges" (which is selected), "Queues", and "Admin". Below the menu is a "Pagination" section with "Page 1" and a "Filter" input field. To the right, it says "Displaying 8 items, page size up to: 100".

The main content is a table of exchanges:

Virtual host	Name	Type	Features	Message rate in	Message rate out
/	(AMQP default)	direct	D		
/	amq.direct	direct	D		
/	amq.fanout	fanout	D		
/	amq.headers	headers	D		
/	amq.match	headers	D		
/	amq.rabbitmq.trace	topic	D I		
/	amq.topic	topic	D		
/	employeeRegistrationExchange	topic	D	0.00/s	0.00/s

At the bottom left, there's a link "▶ Add a new exchange". The footer contains links to "HTTP API", "Documentation", "Tutorials", "New releases", "Commercial edition", "Commercial support", "Google Group", "Discord", "Slack", "Plugins", and "GitHub".



Spring Cloud Stream – Publish Message to RabbitMQ

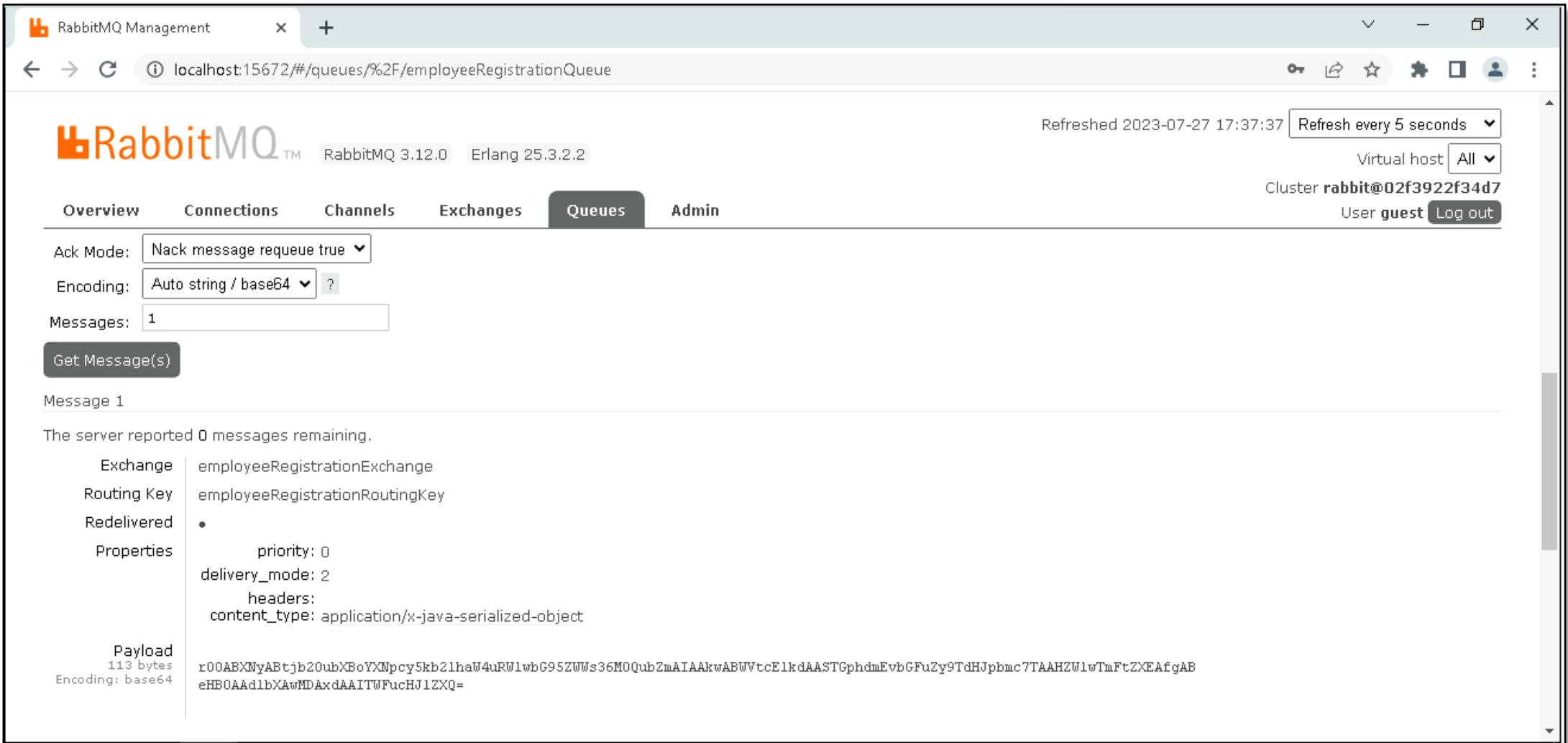
14. Review the Queues in RabbitMQ Dashboard:

The screenshot shows the RabbitMQ Management interface at `localhost:15672/#/queues`. The top navigation bar includes tabs for Overview, Connections, Channels, Exchanges, Queues (which is selected), and Admin. The main content area is titled "Queues" and shows a single queue named "employeeRegistrationQueue". The table provides an overview of the queue's state:

Overview					Messages			Message rates		
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/	employeeRegistrationQueue	classic	D	idle	1	0	1	0.00/s	0.00/s	0.00/s

Below the table, there is a section for "Add a new queue" with fields for Virtual host, Type, Name, and Durability.

15. Get Message(s) in RabbitMQ Dashboard:



The screenshot shows the RabbitMQ Management interface for the queue `employeeRegistrationQueue`. The interface includes the following details:

- Ack Mode:** Nack message requeue true
- Encoding:** Auto string / base64
- Messages:** 1
- Get Message(s)** button
- Message 1:** The server reported 0 messages remaining.
- Message Properties:**
 - Exchange: employeeRegistrationExchange
 - Routing Key: employeeRegistrationRoutingKey
 - Redelivered: *
 - Properties:
 - priority: 0
 - delivery_mode: 2
 - headers:
 - content_type: application/x-java-serialized-object
- Payload:** 113 bytes
Encoding: base64
The payload hex dump is: r00ABXNyABtjb20ubXBoYXNpcy5kb21haW4uRW1wbG95ZWWs36M0QubZmAlAAkwABWVtcElkdAASTGphdmEvbGFuZy9TdHJpbmc7TAAHZW1wTmFtZXEAfgABeHB0AAAd1bXAwMDAx dAAITWFucHJ1ZXQ=



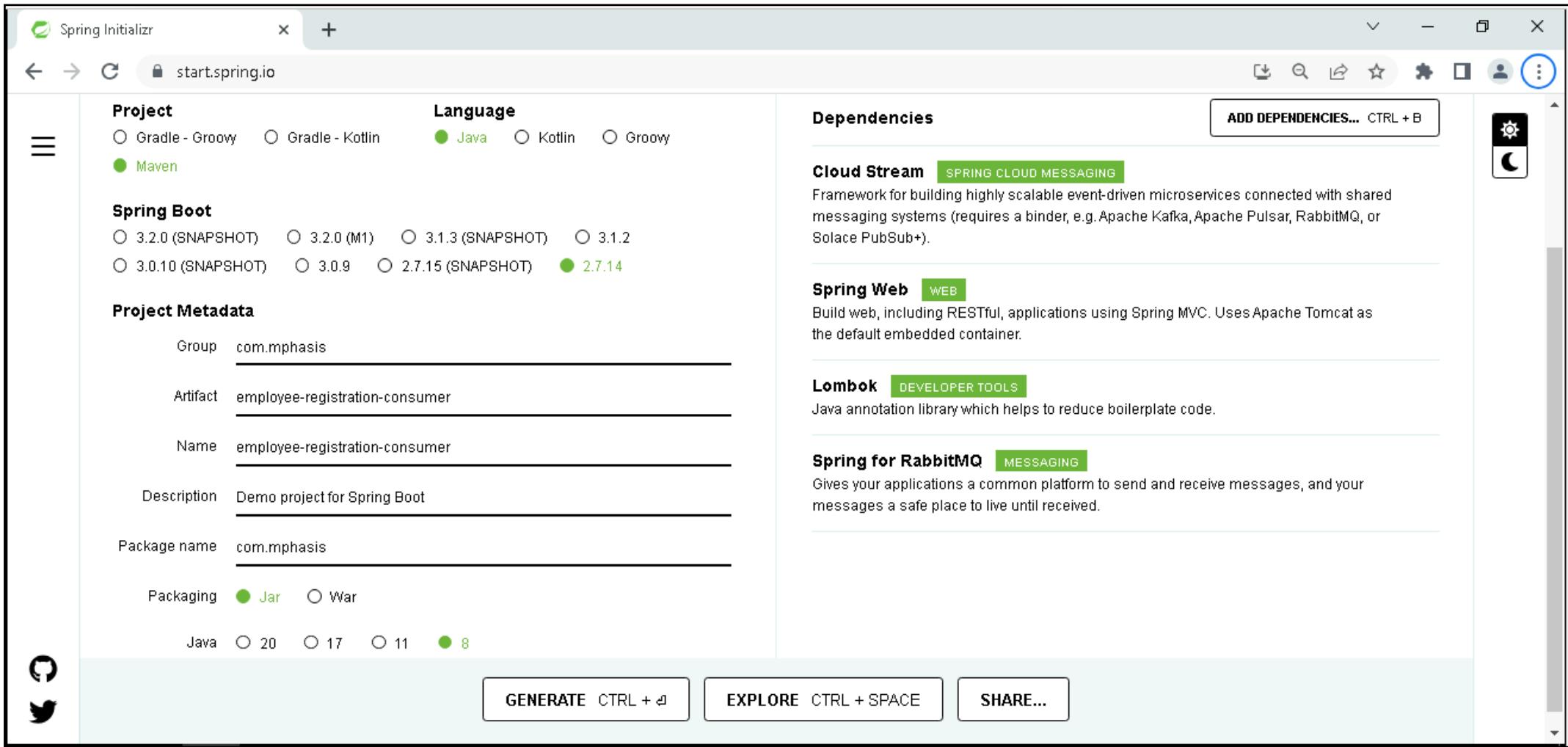
Day - 10

Spring Cloud Stream – Consume Message from RabbitMQ



Spring Cloud Stream – Consume Message from RabbitMQ

1. Create a new Project i.e., employee-registration-consumer:



The screenshot shows the Spring Initializr web application at start.spring.io. The project is titled "employee-registration-consumer". The configuration includes:

- Project:** Maven
- Language:** Java
- Spring Boot:** 2.7.14
- Project Metadata:**
 - Group: com.mphasis
 - Artifact: employee-registration-consumer
 - Name: employee-registration-consumer
 - Description: Demo project for Spring Boot
 - Package name: com.mphasis
 - Packaging: Jar
 - Java: 8
- Dependencies:**
 - Cloud Stream** (selected): SPRING CLOUD MESSAGING
Framework for building highly scalable event-driven microservices connected with shared messaging systems (requires a binder, e.g. Apache Kafka, Apache Pulsar, RabbitMQ, or Solace PubSub+).
 - Spring Web** (disabled): WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
 - Lombok** (disabled): DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.
 - Spring for RabbitMQ** (disabled): MESSAGING
Gives your applications a common platform to send and receive messages, and your messages a safe place to live until received.

At the bottom, there are buttons for **GENERATE** (CTRL + ⌘) and **EXPLORE** (CTRL + SPACE), and a **SHARE...** button.



Spring Cloud Stream – Consume Message from RabbitMQ

2. Configure the rabbitmq properties in application.properties file:

The screenshot shows a code editor window with the title bar "application.properties". The file contains the following configuration properties:

```
1 server.port=9082
2
3 spring.rabbitmq.host=localhost
4 spring.rabbitmq.port=5672
5 spring.rabbitmq.username=guest
6 spring.rabbitmq.password=guest
7
8 rabbitmq.queue.name=employeeRegistrationQueue
9
10|
```



3. Define the model class – Employee:

```
Employee.java ✘
1 package com.mphasis.domain;
2
3+import com.fasterxml.jackson.annotation.JsonIgnoreProperties;□
4
5+@JsonIgnoreProperties("empId")
6 @Data
7 @NoArgsConstructor
8 @AllArgsConstructor
9 @JsonIgnoreProperties
10 public class Employee {
11
12     private String empId;
13     private String empName;
14 }
15
16
17
18
```



Spring Cloud Stream – Consume Message from RabbitMQ

4. Create RabbitMQ Consumer. The Consumer app will listen the queue and consume the messages from the queue. Also consumed messages will be printed to console.

```
J EmployeeRegistrationConsumer.java X
1 package com.mphasis.consumer;
2
3+import org.springframework.amqp.rabbit.annotation.RabbitListener;□
4
5 @Service
6 public class EmployeeRegistrationConsumer {
7
8     @RabbitListener(queues = {"${rabbitmq.queue.name}"))
9     public void consume(Employee employee) {
10         System.out.println("Message arrived! Message: " + employee);
11     }
12 }
13
14 }
```



Spring Cloud Stream – Consume Message from RabbitMQ

5. Start your consumer application. The console of the consumer application now prints:

The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The title bar indicates the project is 'employee-registration-consumer - EmployeeRegistrationConsumerApplication [Spring Boot App]' and the log file is 'C:\Program Files\ojdkbuild\java-1.8.0-openjdk-1.8.0.302-1\bin\javaw.exe' (Jul 27, 2023, 5:48:12 PM). The console output displays the application's startup logs, including the configuration of Spring components like BeanPostProcessorChecker, TomcatWebServer, and StandardEngine, followed by the successful connection to RabbitMQ and the receipt of a message containing an Employee object with empId=emp0001 and empName=Manpreet.

```
2023-07-27 17:48:14.300 INFO 6288 --- [           main] trationDelegate$BeanPostProcessorChecker : Bean 'spelConverter' c^
2023-07-27 17:48:14.314 INFO 6288 --- [           main] trationDelegate$BeanPostProcessorChecker : Bean 'spring.jmx-org.s^
2023-07-27 17:48:14.322 INFO 6288 --- [           main] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework^
2023-07-27 17:48:14.326 INFO 6288 --- [           main] trationDelegate$BeanPostProcessorChecker : Bean 'mbeanServer' of^
2023-07-27 17:48:14.541 INFO 6288 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized wit^
2023-07-27 17:48:14.548 INFO 6288 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomc^
2023-07-27 17:48:14.549 INFO 6288 --- [           main] org.apache.catalina.core.StandardEngine : Starting Servlet engin^
2023-07-27 17:48:14.648 INFO 6288 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[] : Initializing Spring em^
2023-07-27 17:48:14.649 INFO 6288 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationCor^
2023-07-27 17:48:15.370 INFO 6288 --- [           main] c.f.c.c.BeanFactoryAwareFunctionRegistry : Can't determine defau^
2023-07-27 17:48:15.575 INFO 6288 --- [           main] o.s.i.monitor.IntegrationMBeanExporter : Registering MessageCh^
2023-07-27 17:48:15.610 INFO 6288 --- [           main] o.s.i.monitor.IntegrationMBeanExporter : Registering MessageCh^
2023-07-27 17:48:15.618 INFO 6288 --- [           main] o.s.i.monitor.IntegrationMBeanExporter : Registering MessageHar^
2023-07-27 17:48:15.653 INFO 6288 --- [           main] o.s.i.endpoint.EventDrivenConsumer : Adding {logging-channel^
2023-07-27 17:48:15.654 INFO 6288 --- [           main] o.s.i.channel.PublishSubscribeChannel : Channel 'application.e^
2023-07-27 17:48:15.654 INFO 6288 --- [           main] o.s.i.endpoint.EventDrivenConsumer : started bean '_org.spi^
2023-07-27 17:48:15.672 INFO 6288 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port^
2023-07-27 17:48:15.675 INFO 6288 --- [           main] o.s.a.r.c.CachingConnectionFactory : Attempting to connect^
2023-07-27 17:48:15.722 INFO 6288 --- [           main] o.s.a.r.c.CachingConnectionFactory : Created new connection^
2023-07-27 17:48:15.790 INFO 6288 --- [           main] .EmployeeRegistrationConsumerApplication : Started EmployeeRegis^
Message arrived! Message: Employee(empId=emp0001, empName=Manpreet)
```



Day - 10

Stream Processing Using Spring Cloud Data Flow

- Spring Cloud Data Flow provides tools to create complex topologies for streaming and batch data pipelines. The data pipelines consist of Spring Boot apps, built using the Spring Cloud Stream or Spring Cloud Task microservice frameworks.
- Spring Cloud Data Flow supports a range of data processing use cases, from ETL to import/export, event streaming, and predictive analytics.
- Spring Cloud Data Flow is a toolkit to build real-time data integration and data processing pipelines by establishing message flows between Spring Boot applications that could be deployed on top of different runtimes.

1. Long-lived applications.

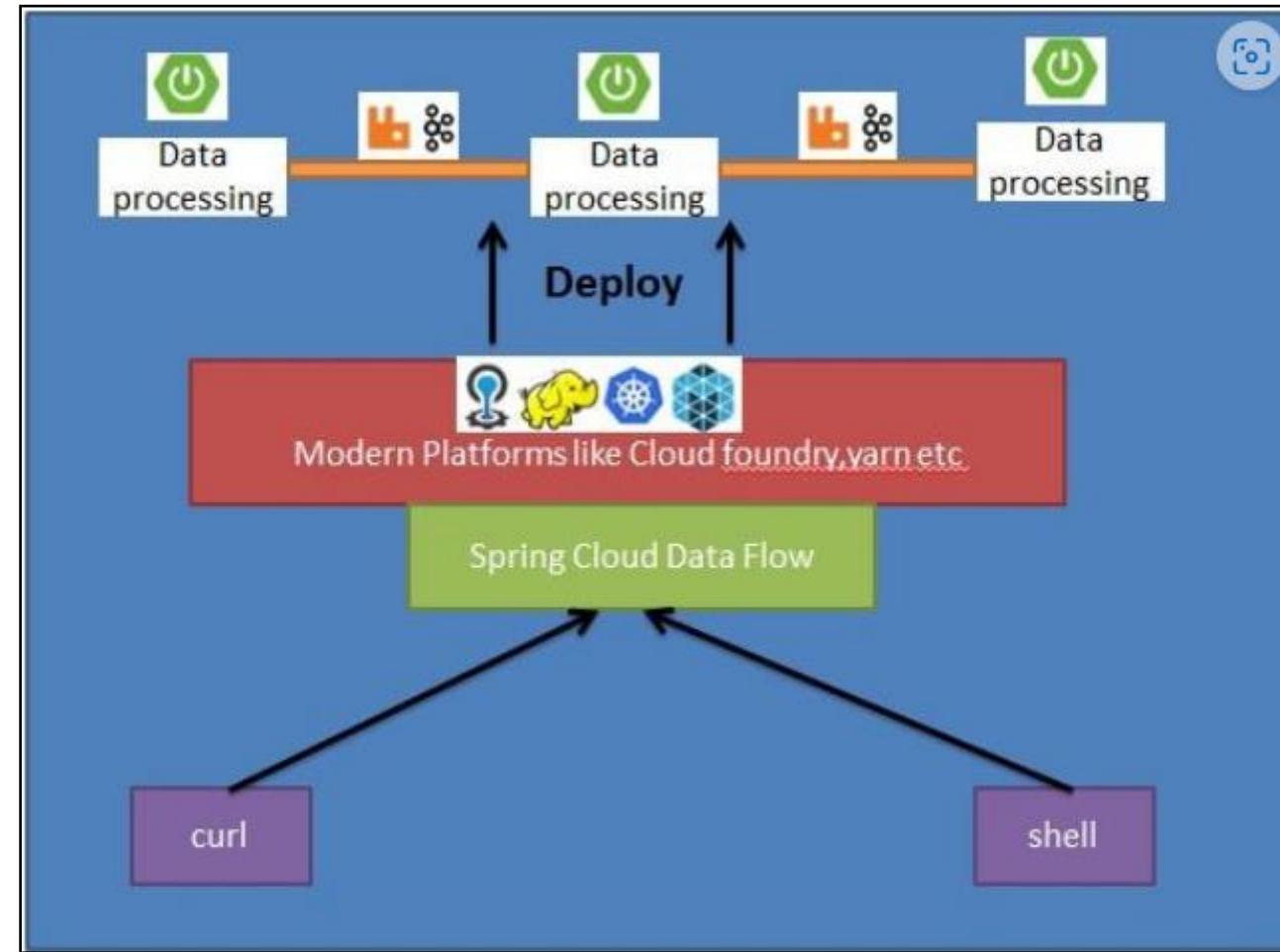
- There are two types of long-lived applications:
 - Message-driven applications where an unbounded amount of data is consumed or produced through a single input or output (or both).
 - The second is a message-driven application that can have multiple inputs and outputs. It could also be an application that does not use messaging middleware at all.

2. Short-lived applications.

- Short-lived applications that process a finite set of data and then terminate. There are two variations of short-lived applications.
 - The first is a Task that runs your code and records the status of the execution in the Data Flow database. It can, optionally, use the Spring Cloud Task framework and need not be a Java application. However, the application does need to record its run status in Data Flow's database.
 - The second is an extension of the first that includes the Spring Batch framework as the foundation for performing batch processing.

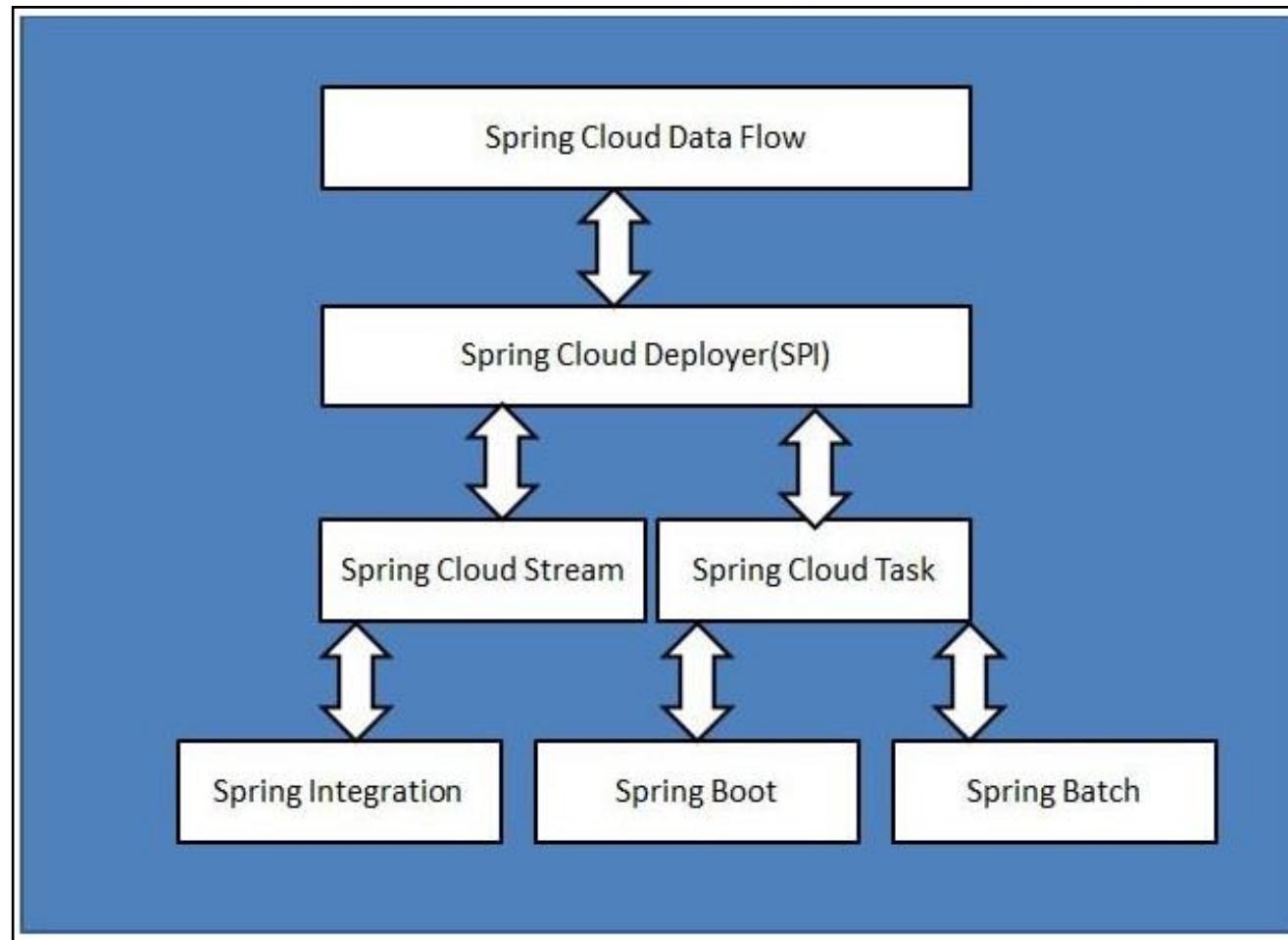


Stream Applications





SCDF is composed of the following Spring Projects





Streams with Sources, Processors, and Sinks

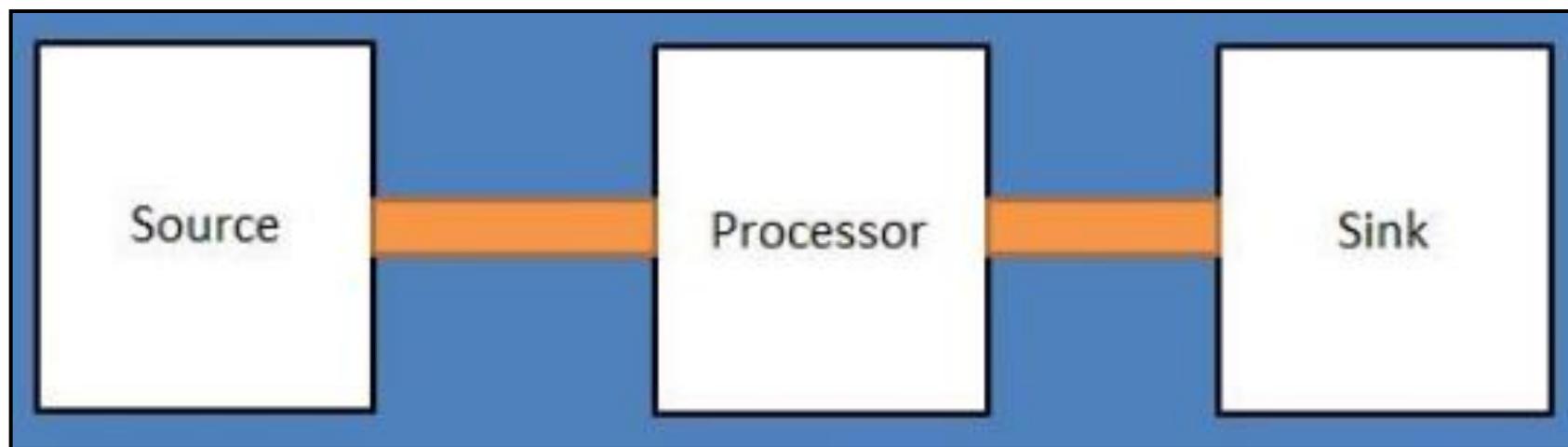
- **Source:** Message producer that sends messages to a destination.
- **Sink:** Message consumer that reads messages from a destination.
- **Processor:** The combination of a source and a sink. A processor consumes message from a destination and produces messages to send to another destination.

- Applications of these three types are registered with **Data Flow** by using the source, processor and sink to describe the type of the application being registered.



Streams with Sources, Processors, and Sinks

- We will be creating Spring Boot Microservices as follows and deploy them using SCDF.





Day - 10

Implementing Stream Processing Using Spring Cloud Data Flow



Installing RabbitMQ

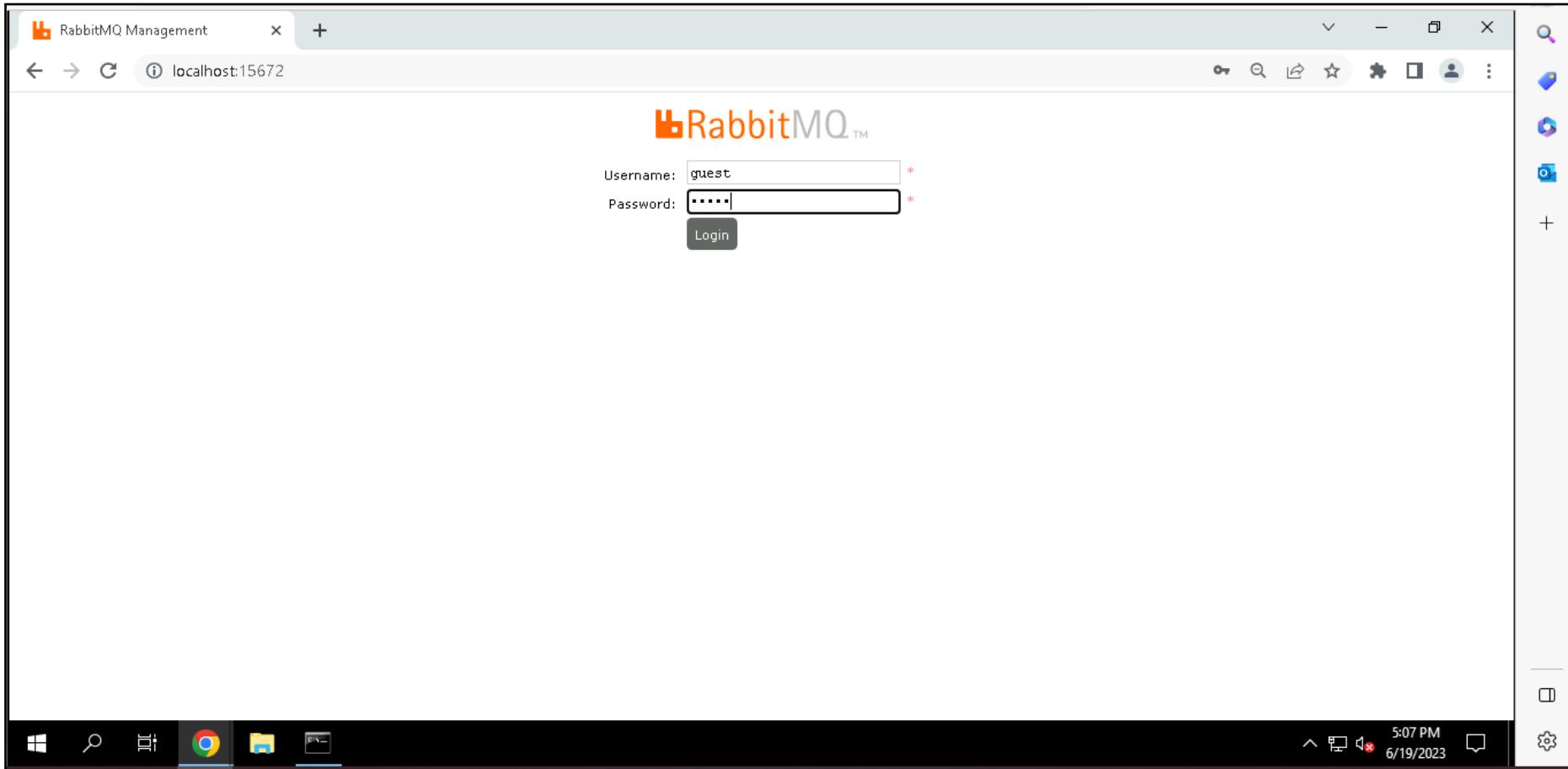
1. To Download and Run Rabbit MQ, go to <https://rabbitmq.com>
2. Click on Download + Installation button.
3. Open the command prompt and execute the below docker command:

```
# Latest RabbitMQ 3.12
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.12-management
```



Installing RabbitMQ

4. Access the RabbitMQ dashboard via <http://localhost:15672>

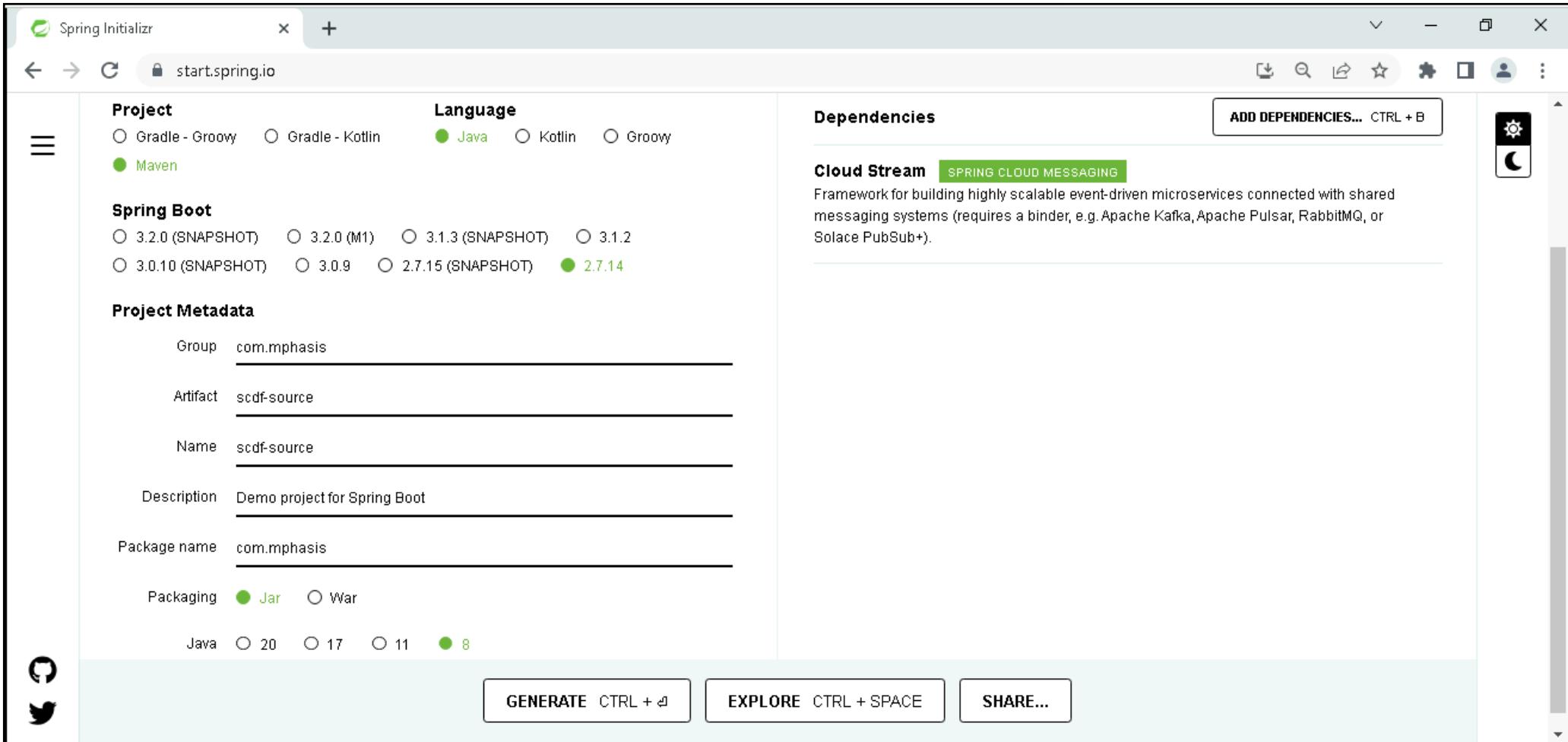


5. The default username and password of RabbitMQ is guest.



Create the Source Module

6. Create a new Project i.e., scdf-source:



The screenshot shows the Spring Initializr web application interface. The project name is set to 'scdf-source'. Under 'Project', 'Maven' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', '2.7.14' is selected. In the 'Dependencies' section, 'Cloud Stream' is selected, which includes 'SPRING CLOUD MESSAGING'. This dependency is described as a framework for building highly scalable event-driven microservices connected with shared messaging systems (requires a binder, e.g. Apache Kafka, Apache Pulsar, RabbitMQ, or Solace PubSub+). The 'Project Metadata' section shows the following values: Group: com.mphasis, Artifact: scdf-source, Name: scdf-source, Description: Demo project for Spring Boot, Package name: com.mphasis, and Packaging: Jar. Java version 8 is selected. At the bottom, there are 'GENERATE' and 'EXPLORE' buttons, and social sharing icons for GitHub and Twitter.

7. Add the Spring Cloud Stream with Rabbit dependency to the source.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

8. Change the version - Spring Boot and Spring Cloud:

- Spring Boot: 2.3.10.RELEASE
- Spring Cloud: Hoxton.SR11



Create the Source Module

9. Create the SourceApplication.java as follows:

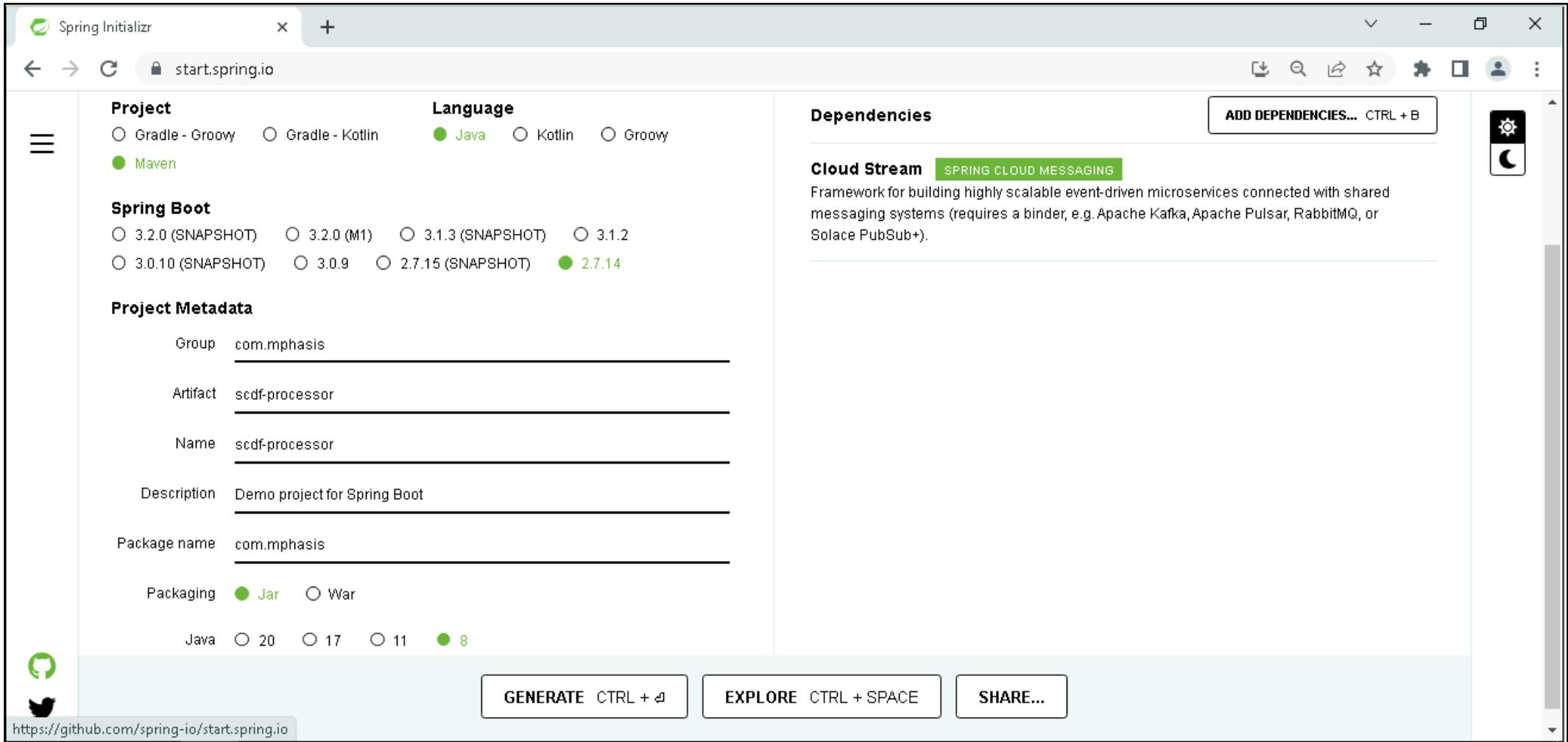
```
J ScdfSourceApplication.java ✘
1 package com.mphasis;
2
3+import java.util.Date;□
14
15 @EnableBinding(Source.class)
16 @SpringBootApplication
17 public class ScdfSourceApplication {
18
19@  @Bean
20     @InboundChannelAdapter(value = Source.OUTPUT,
21                         poller = @Poller(fixedDelay = "10000", maxMessagesPerPoll = "1"))
22     public MessageSource<Long> timeMessageSource() {
23
24         return () -> MessageBuilder.withPayload(new Date().getTime()).build();
25     }
26
27@  public static void main(String[] args) {
28     SpringApplication.run(ScdfSourceApplication.class, args);
29 }
30 }
```

10. Go to Maven – Build – use **clean, install** as goals.



Create the Processor Module

11. Create a new Project i.e., scdf-processor:



The screenshot shows the Spring Initializr web application at start.spring.io. The configuration for a new project named 'scdf-processor' is displayed.

Project: Maven

Language: Java

Spring Boot: 2.7.14

Project Metadata:

- Group: com.mphasis
- Artifact: scdf-processor
- Name: scdf-processor
- Description: Demo project for Spring Boot
- Package name: com.mphasis
- Packaging: Jar
- Java: 8

Dependencies: Cloud Stream (Spring Cloud Messaging)

Cloud Stream (Spring Cloud Messaging) is selected. The description states: "Framework for building highly scalable event-driven microservices connected with shared messaging systems (requires a binder, e.g. Apache Kafka, Apache Pulsar, RabbitMQ, or Solace PubSub+)."

At the bottom, there are buttons for **GENERATE** (CTRL + ⌘), **EXPLORE** (CTRL + SPACE), and **SHARE...**.

12. Add the Spring Cloud Stream with Rabbit dependency to the source.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

13. Change the version - Spring Boot and Spring Cloud:

- Spring Boot: 2.3.10.RELEASE
- Spring Cloud: Hoxton.SR11



Create the Processor Module

14. Create the ProcessorApplication.java as follows:

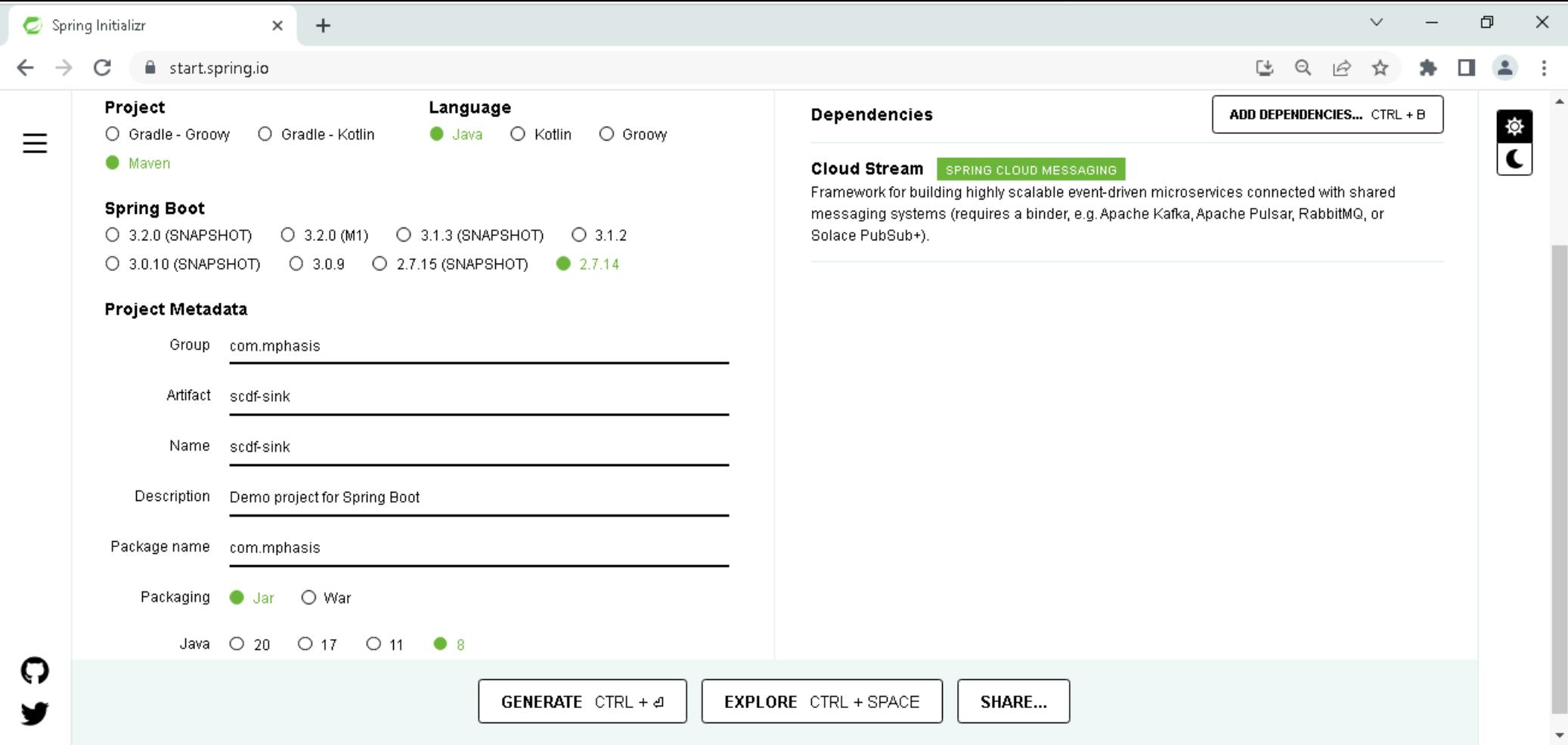
```
J ScdfProcessorApplication.java X
1 package com.mphasis;
2
3+import java.text.DateFormat;[]
11
12 @EnableBinding(Processor.class)
13 @SpringBootApplication
14 public class ScdfProcessorApplication {
15
16@    @Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
17    public Object transform(Long date) {
18
19        DateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
20        return dateFormat.format(date);
21    }
22
23@    public static void main(String[] args) {
24        SpringApplication.run(ScdfProcessorApplication.class, args);
25    }
26
27 }
```

15. Go to Maven – Build – use **clean, install** as goals.



Create the Sink Module

16. Create a new Project i.e., scdf-sink:



The screenshot shows the Spring Initializr interface at start.spring.io. The configuration for the 'scdf-sink' project is as follows:

- Project:** Maven
- Language:** Java
- Spring Boot:** 2.7.14
- Project Metadata:**
 - Group: com.mphasis
 - Artifact: scdf-sink
 - Name: scdf-sink
 - Description: Demo project for Spring Boot
 - Package name: com.mphasis
 - Packaging: Jar
- Dependencies:** Cloud Stream (Spring Cloud Messaging)

At the bottom, there are buttons for GENERATE (CTRL + D), EXPLORE (CTRL + SPACE), and SHARE... .

17. Add the Spring Cloud Stream with Rabbit dependency to the source.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

18. Change the version - Spring Boot and Spring Cloud:

- Spring Boot: 2.3.10.RELEASE
- Spring Cloud: Hoxton.SR11



Create the Sink Module

19. Create the SinkApplication.java as follows:

```
ScdfSinkApplication.java
1 package com.mphasis;
2
3 import org.slf4j.Logger;
4
5 import org.springframework.boot.SpringApplication;
6 import org.springframework.boot.autoconfigure.SpringBootApplication;
7
8 import org.springframework.cloud.stream.annotation.EnableBinding;
9 import org.springframework.cloud.stream.annotation.StreamListener;
10
11 @EnableBinding(Sink.class)
12 @SpringBootApplication
13 public class ScdfSinkApplication {
14
15     private static Logger logger = LoggerFactory.getLogger(ScdfSinkApplication.class);
16
17     @StreamListener(Sink.INPUT)
18     public void loggerSink(String date) {
19
20         logger.info("Sink Received: " + date);
21     }
22
23     public static void main(String[] args) {
24         SpringApplication.run(ScdfSinkApplication.class, args);
25     }
26
27 }
28
```

20. Go to Maven – Build – use **clean, install** as goals.



Download the Spring Cloud Data Flow

21. Download the Spring Cloud Data Flow and start it.

<http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-server-local/1.3.0.M1/spring-cloud-dataflow-server-local-1.3.0.M1.jar>

The screenshot shows a browser window with the URL <http://repo.spring.io/ui/native/milestone/org/springframework/cloud/spring-cloud-dataflow-server-local/1.3.0.M1/>. The page title is "Index of milestone/org/springframework/cloud/spring-cloud-dataflow-server-local/1.3.0.M1". The table lists the following files:

Name	Last Modified	Size	Download Link
...			
spring-cloud-dataflow-server-local-1.3.0.M1-javadoc.jar	04-08-17 10:45:32 +0000	25.0 KB	spring-cloud-dataflow-server-local-1.3.0.M1-javadoc.jar
spring-cloud-dataflow-server-local-1.3.0.M1-sources.jar	04-08-17 10:45:32 +0000	3.4 KB	spring-cloud-dataflow-server-local-1.3.0.M1-sources.jar
spring-cloud-dataflow-server-local-1.3.0.M1.jar	04-08-17 10:45:31 +0000	50.0 MB	spring-cloud-dataflow-server-local-1.3.0.M1.jar
spring-cloud-dataflow-server-local-1.3.0.M1.pom	04-08-17 10:45:32 +0000	7.1 KB	spring-cloud-dataflow-server-local-1.3.0.M1.pom



Download the Spring Cloud Data Flow

22. Start Spring Cloud using-

```
java -jar spring-cloud-dataflow-server-local-1.3.0.M1.jar
```

The screenshot shows a Windows Command Prompt window with the title 'C:\Windows\System32\cmd.exe - java -jar spring-cloud-dataflow-server-local-1.3.0.M1.jar'. The window displays the following output:

```
Microsoft Windows [Version 10.0.17763.4645]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\labuser\Downloads>java -jar spring-cloud-dataflow-server-local-1.3.0.M1.jar



Spring Cloud Data Flow Local Server (v1.3.0.M1)

2023-07-27 18:45:13.072 INFO 8896 --- [           main] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at: http://localhost:8888
2023-07-27 18:45:14.363 WARN 8896 --- [           main] c.c.c.ConfigServicePropertySourceLocator : Could not locate PropertySource: I/O error on GET request for "http://localhost:8888/spring-cloud-dataflow-server-local/default": Connection refused: connect
2023-07-27 18:45:14.366 INFO 8896 --- [           main] o.s.c.d.s.local.LocalDataFlowServer : No active profile set, falling back to default profiles: default
2023-07-27 18:45:18.867 INFO 8896 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Multiple Spring Data modules found, entering strict repository configuration mode!
2023-07-27 18:45:19.968 INFO 8896 --- [           main] o.s.cloud.context.scope.GenericScope : BeanFactory id=b15ff760-91d3-31de-a2a8-f2cc5ce41a5c
2023-07-27 18:45:23.308 INFO 8896 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
```



Download Spring Cloud Shell

23. Download the Spring Cloud Shell jars and start it.

<http://repo.spring.io/milestone/org/springframework/cloud/spring-cloud-dataflow-shell/1.3.0.M1/spring-cloud-dataflow-shell-1.3.0.M1.jar>

The screenshot shows a web browser window with the URL <http://repo.spring.io/ui/native/milestone/org/springframework/cloud/spring-cloud-dataflow-shell/1.3.0.M1/>. The page title is "Index of milestone/org/springframework/cloud/spring-cloud-dataflow-shell/1.3.0.M1". The table lists the following files:

Name	Last Modified	Size	Download Link
..			
spring-cloud-dataflow-shell-1.3.0.M1-javadoc.jar	04-08-17 10:45:37 +0000	24.8 KB	spring-cloud-dataflow-shell-1.3.0.M1-javadoc.jar
spring-cloud-dataflow-shell-1.3.0.M1-sources.jar	04-08-17 10:45:38 +0000	2.6 KB	spring-cloud-dataflow-shell-1.3.0.M1-sources.jar
spring-cloud-dataflow-shell-1.3.0.M1.jar	04-08-17 10:45:38 +0000	19.1 MB	spring-cloud-dataflow-shell-1.3.0.M1.jar
spring-cloud-dataflow-shell-1.3.0.M1.pom	04-08-17 10:45:38 +0000	1.1 KB	spring-cloud-dataflow-shell-1.3.0.M1.pom



Download the Spring Cloud Data Flow

22. Start Spring Cloud using-

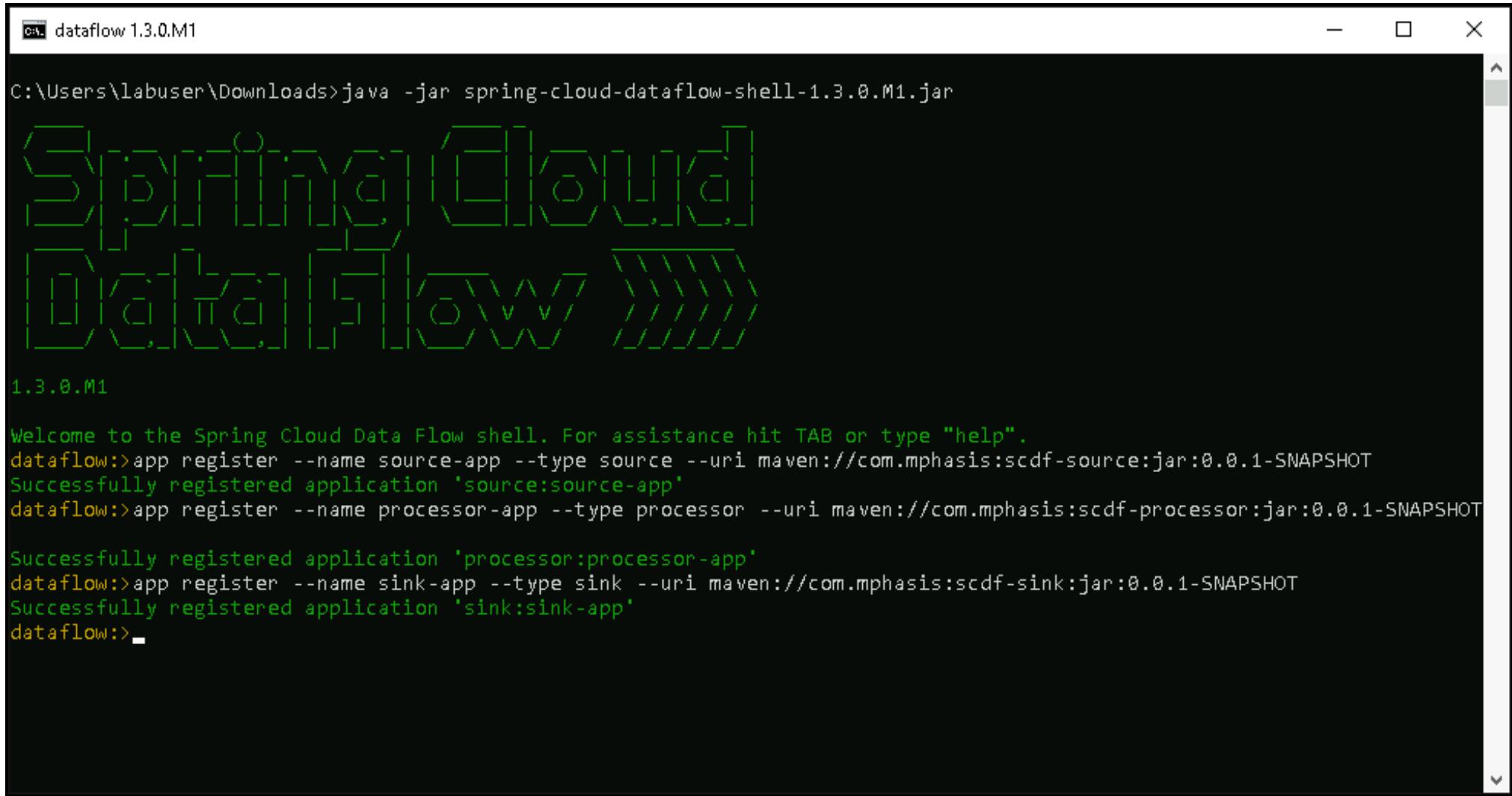
```
java -jar spring-cloud-dataflow-shell-1.3.0.M1.jar
```

A screenshot of a terminal window titled "dataflow 1.3.0.M1". The window shows the command "java -jar spring-cloud-dataflow-shell-1.3.0.M1.jar" being run, followed by the Spring Cloud Data Flow logo, which is a grid of green symbols representing data flow. Below the logo, the text "1.3.0.M1" is displayed. At the bottom, the message "Welcome to the Spring Cloud Data Flow shell. For assistance hit TAB or type "help". dataflow:>" is shown. The terminal has a dark background and light-colored text.



Install the applications to runtime

23. In the Spring Cloud Data Shell, install the application



```
dataflow 1.3.0.M1
C:\Users\labuser\Downloads>java -jar spring-cloud-dataflow-shell-1.3.0.M1.jar
[SPRING CLOUD DATA FLOW] [SPRING CLOUD DATA FLOW]
[ ] [G] [T] [G] [ ] [G] [V] [V] [V] [V] [V] [V]
1.3.0.M1

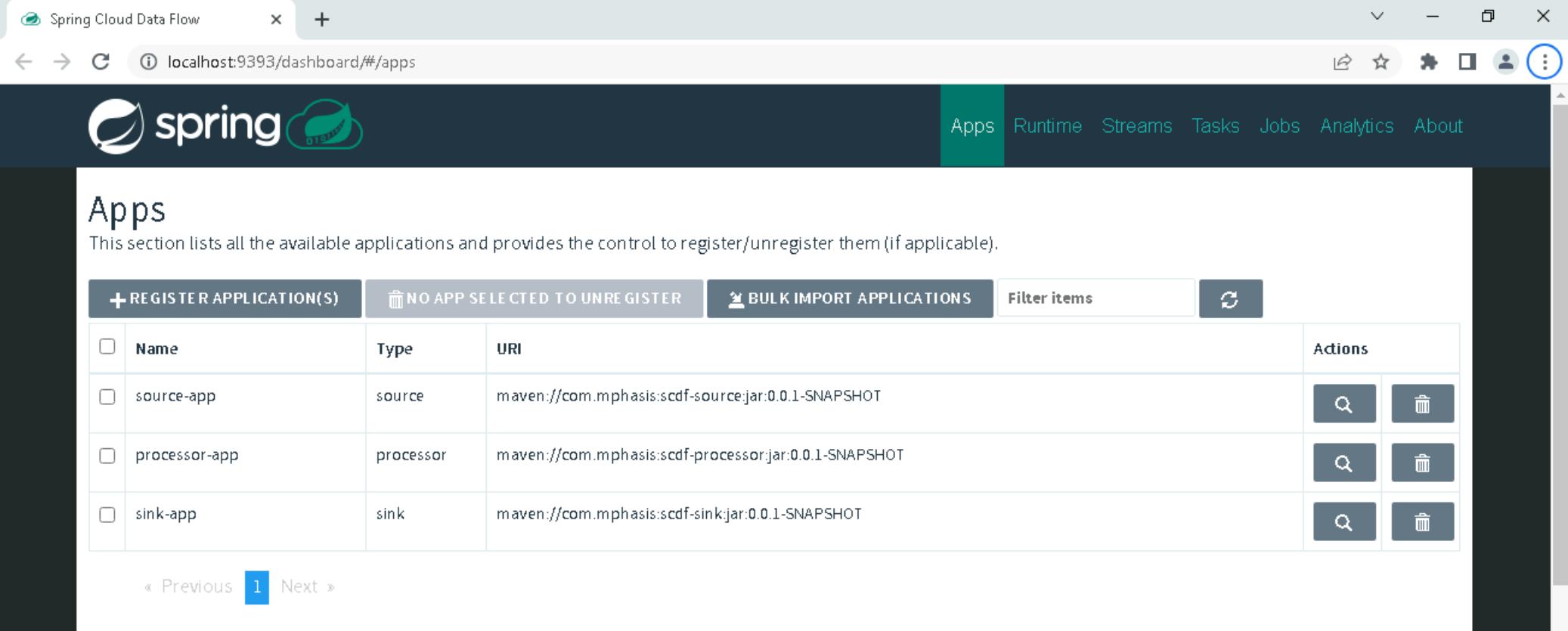
Welcome to the Spring Cloud Data Flow shell. For assistance hit TAB or type "help".
dataflow:>app register --name source-app --type source --uri maven://com.mphasis:scdf-source:jar:0.0.1-SNAPSHOT
Successfully registered application 'source:source-app'
dataflow:>app register --name processor-app --type processor --uri maven://com.mphasis:scdf-processor:jar:0.0.1-SNAPSHOT

Successfully registered application 'processor:processor-app'
dataflow:>app register --name sink-app --type sink --uri maven://com.mphasis:scdf-sink:jar:0.0.1-SNAPSHOT
Successfully registered application 'sink:sink-app'
dataflow:>
```



Install the applications to runtime

24. Go to Spring Cloud Data Flow UI Console - <http://localhost:9393/dashboard>



The screenshot shows the Spring Cloud Data Flow UI Console at <http://localhost:9393/dashboard/#/apps>. The page title is "Spring Cloud Data Flow". The navigation bar includes links for Apps (which is highlighted in green), Runtime, Streams, Tasks, Jobs, Analytics, and About. The main content area is titled "Apps" and contains a sub-instruction: "This section lists all the available applications and provides the control to register/unregister them (if applicable)". Below this is a table with the following data:

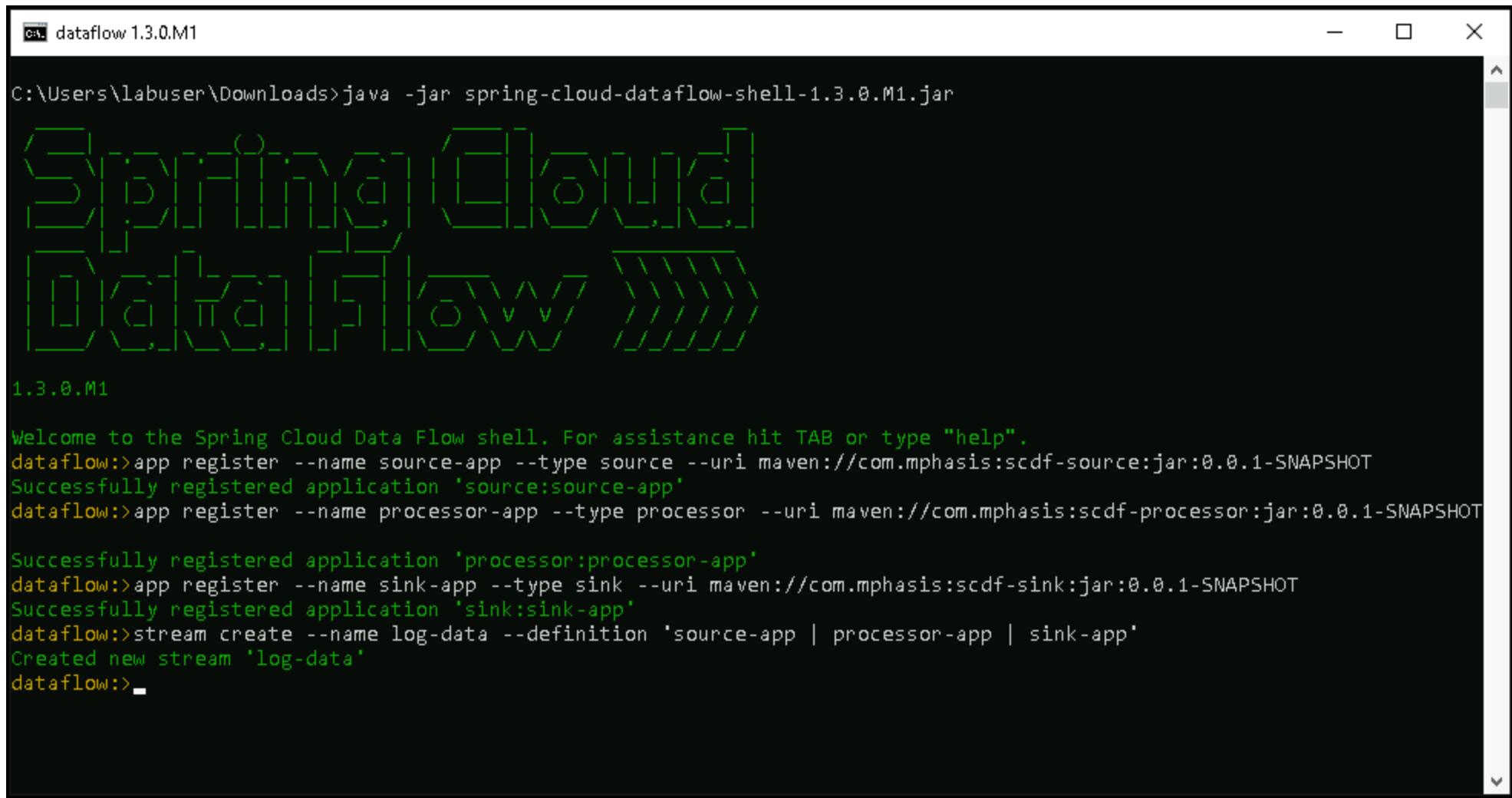
<input type="checkbox"/>	Name	Type	URI	Actions
<input type="checkbox"/>	source-app	source	maven://com.mphasis:scdf-source:jar:0.0.1-SNAPSHOT	
<input type="checkbox"/>	processor-app	processor	maven://com.mphasis:scdf-processor:jar:0.0.1-SNAPSHOT	
<input type="checkbox"/>	sink-app	sink	maven://com.mphasis:scdf-sink:jar:0.0.1-SNAPSHOT	

Pagination controls at the bottom left include "« Previous", a blue-highlighted "1", and "Next »". At the bottom of the page, there are links for "PROJECT", "DOCUMENTATION", and "NEED HELP?", along with copyright information: "© 2017 Pivotal Software, Inc." and the date "8/22/2023".



Create Stream

25. Create Stream:

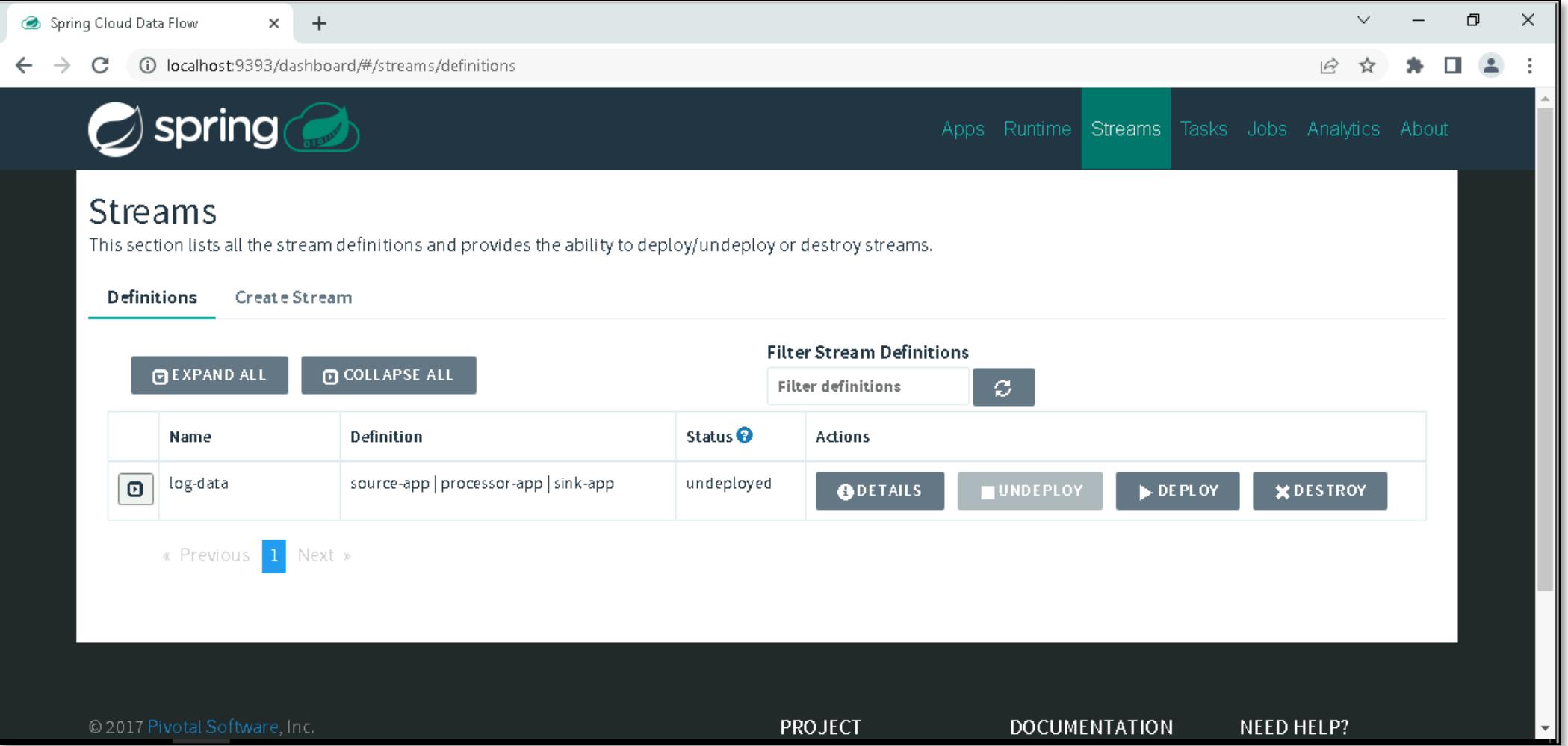


```
C:\Users\labuser\Downloads>java -jar spring-cloud-dataflow-shell-1.3.0.M1.jar
dataflow 1.3.0.M1
Welcome to the Spring Cloud Data Flow shell. For assistance hit TAB or type "help".
dataflow:>app register --name source-app --type source --uri maven://com.mphasis:scdf-source:jar:0.0.1-SNAPSHOT
Successfully registered application 'source:source-app'
dataflow:>app register --name processor-app --type processor --uri maven://com.mphasis:scdf-processor:jar:0.0.1-SNAPSHOT
Successfully registered application 'processor:processor-app'
dataflow:>app register --name sink-app --type sink --uri maven://com.mphasis:scdf-sink:jar:0.0.1-SNAPSHOT
Successfully registered application 'sink:sink-app'
dataflow:>stream create --name log-data --definition 'source-app | processor-app | sink-app'
Created new stream 'log-data'
dataflow:>
```



Create Stream

26. Go to Spring Cloud Data Flow UI Console - <http://localhost:9393/dashboard>



The screenshot shows the Spring Cloud Data Flow UI Streams page. The title bar indicates the URL is `localhost:9393/dashboard/#/streams/definitions`. The top navigation bar includes links for Apps, Runtime, Streams (which is selected), Tasks, Jobs, Analytics, and About. The main content area is titled "Streams" and contains the following information:

This section lists all the stream definitions and provides the ability to deploy/undeploy or destroy streams.

Below this, there are two tabs: "Definitions" (selected) and "Create Stream".

On the left, there are "EXPAND ALL" and "COLLAPSE ALL" buttons. On the right, there is a "Filter Stream Definitions" section with a "Filter definitions" input field and a refresh icon.

A table displays the stream definitions:

	Name	Definition	Status	Actions
	log-data	source-app processor-app sink-app	undeployed	 DETAILS  UNDEPLOY  DEPLOY  DESTROY

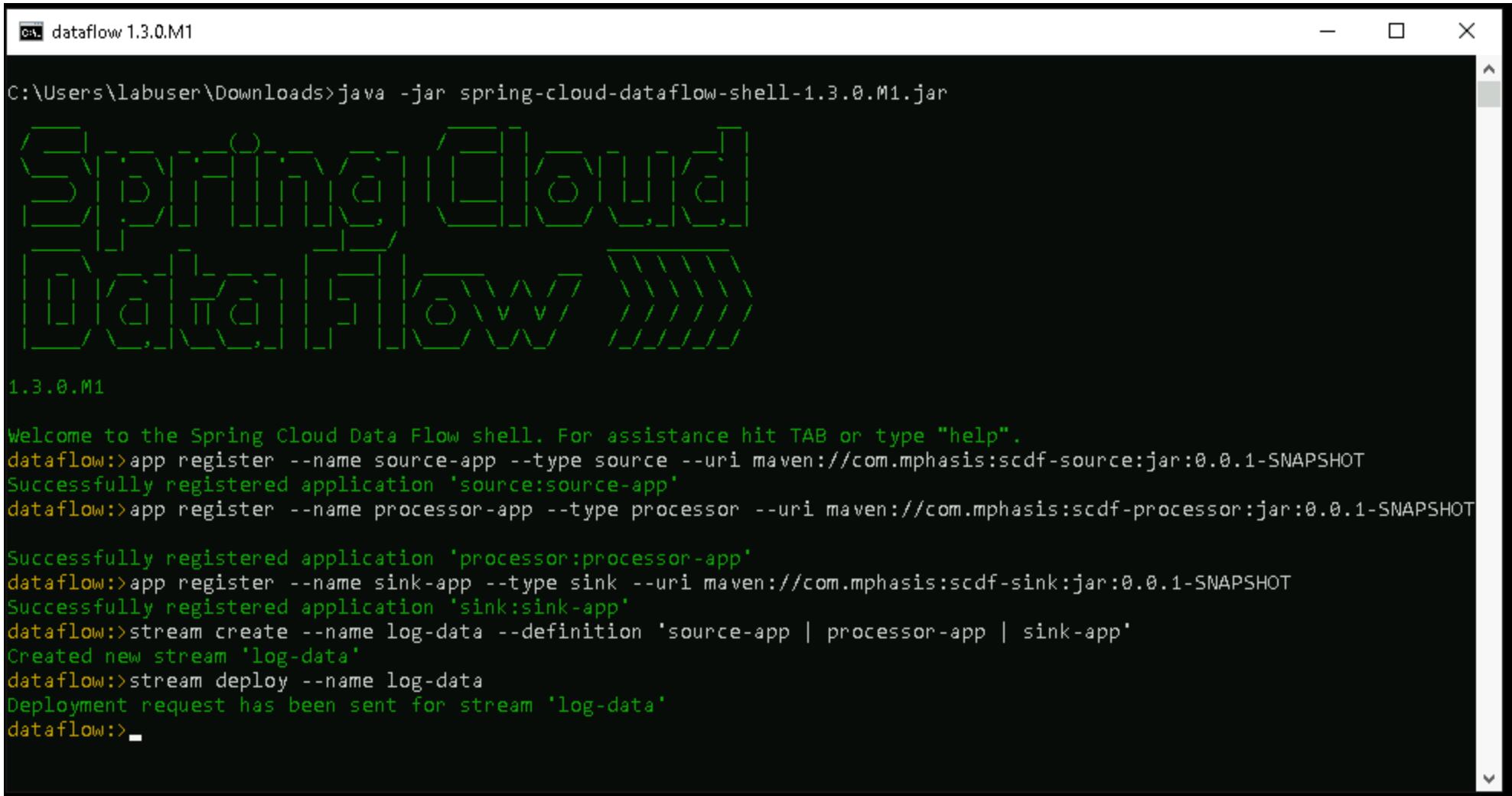
Pagination controls at the bottom show "« Previous 1 Next »".

The footer contains links for PROJECT, DOCUMENTATION, NEED HELP?, and copyright information: © 2017 Pivotal Software, Inc.



Start the Stream

27. Start the Stream:

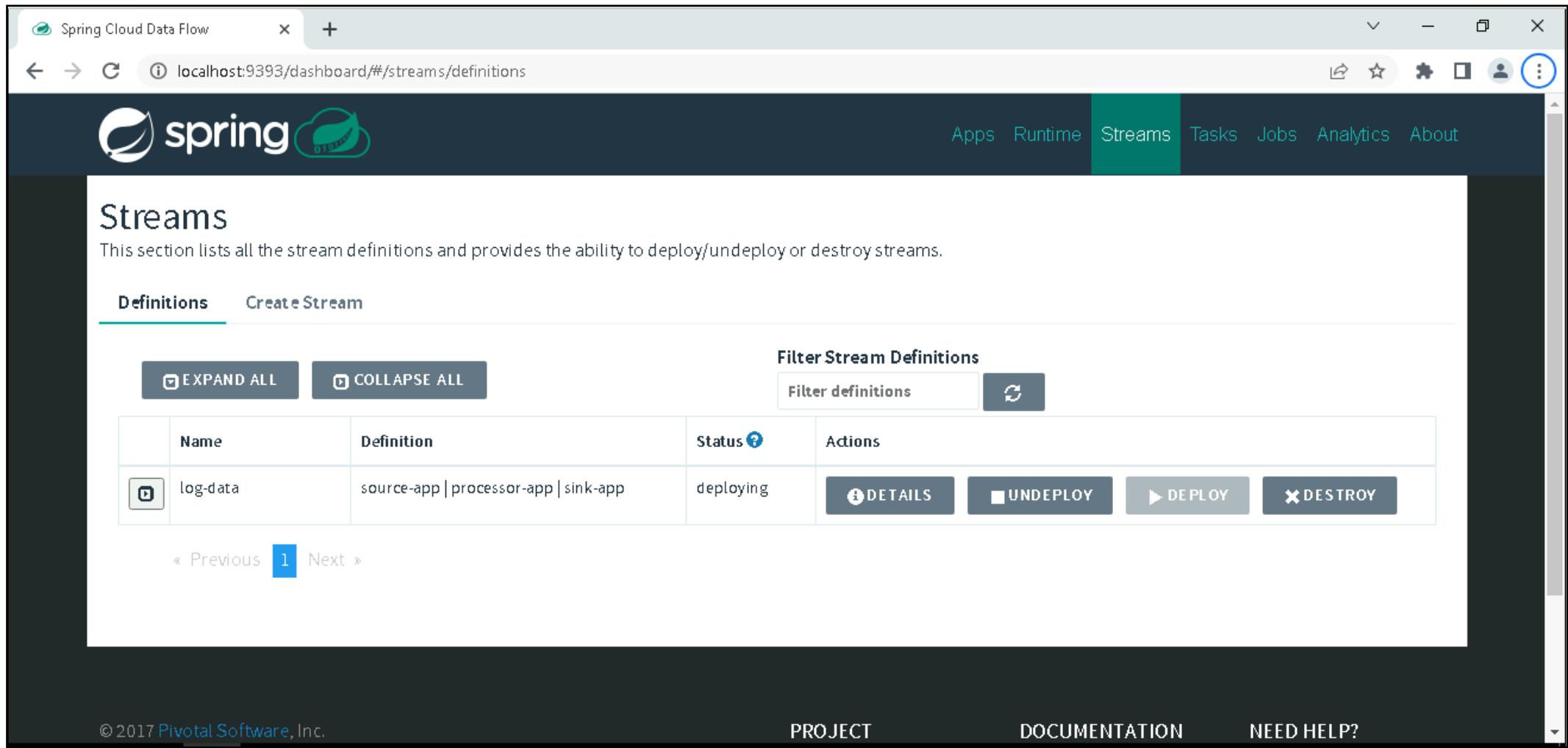


```
C:\Users\labuser\Downloads>java -jar spring-cloud-dataflow-shell-1.3.0.M1.jar
dataflow:~>
dataflow:~>1.3.0.M1
dataflow:~>Welcome to the Spring Cloud Data Flow shell. For assistance hit TAB or type "help".
dataflow:>app register --name source-app --type source --uri maven://com.mphasis:scdf-source:jar:0.0.1-SNAPSHOT
Successfully registered application 'source:source-app'
dataflow:>app register --name processor-app --type processor --uri maven://com.mphasis:scdf-processor:jar:0.0.1-SNAPSHOT
Successfully registered application 'processor:processor-app'
dataflow:>app register --name sink-app --type sink --uri maven://com.mphasis:scdf-sink:jar:0.0.1-SNAPSHOT
Successfully registered application 'sink:sink-app'
dataflow:>stream create --name log-data --definition 'source-app | processor-app | sink-app'
Created new stream 'log-data'
dataflow:>stream deploy --name log-data
Deployment request has been sent for stream 'log-data'
dataflow:>
```



Start the Stream

26. Go to Spring Cloud Data Flow UI Console - <http://localhost:9393/dashboard>



The screenshot shows the Spring Cloud Data Flow UI Streams page. The title bar indicates the URL is `localhost:9393/dashboard/#/streams/definitions`. The top navigation bar includes links for Apps, Runtime, Streams (which is highlighted in green), Tasks, Jobs, Analytics, and About. The main content area is titled "Streams" and contains the following information:

This section lists all the stream definitions and provides the ability to deploy/undeploy or destroy streams.

Definitions Create Stream

EXPAND ALL COLLAPSE ALL

Filter Stream Definitions
Filter definitions

	Name	Definition	Status <small>?</small>	Actions
	log-data	source-app processor-app sink-app	deploying	<input type="button" value="DETAILS"/> <input type="button" value="UNDEPLOY"/> <input type="button" value="DEPLOY"/> <input type="button" value="DESTROY"/>

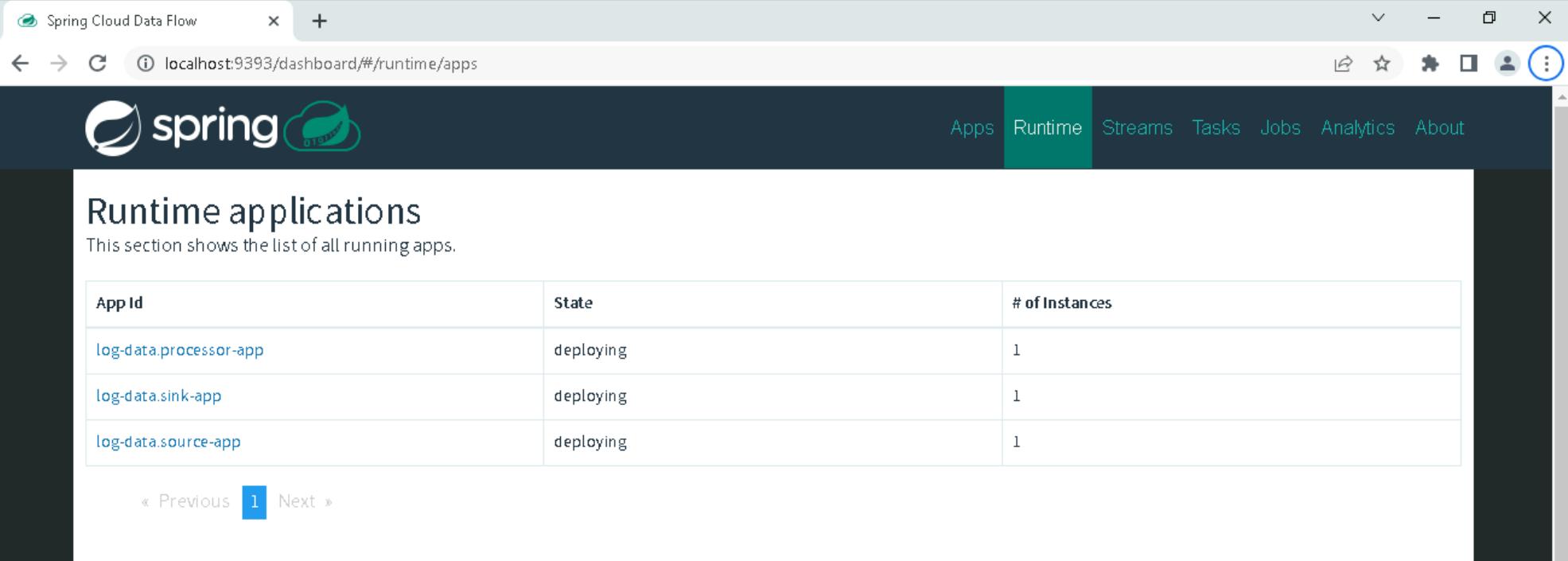
« Previous 1 Next »

At the bottom of the page, there are links for PROJECT, DOCUMENTATION, and NEED HELP?.



Find the Sink logs

27. Go to Spring Cloud Data Flow UI Console - <http://localhost:9393/dashboard>
28. Go to Runtime. Click on the running: log-data.sink-app Application.



The screenshot shows the Spring Cloud Data Flow UI Runtime applications page. The URL in the browser is localhost:9393/dashboard/#/runtime/apps. The page title is "Runtime applications". It displays a table of running applications:

App Id	State	# of Instances
log-data.processor-app	deploying	1
log-data.sink-app	deploying	1
log-data.source-app	deploying	1

At the bottom of the table, there are navigation links: « Previous, 1, Next ».

At the bottom of the page, there is footer information:

© 2017 Pivotal Software, Inc.
All Rights Reserved.

PROJECT
[Project Page](#)
[Issue Tracker](#)

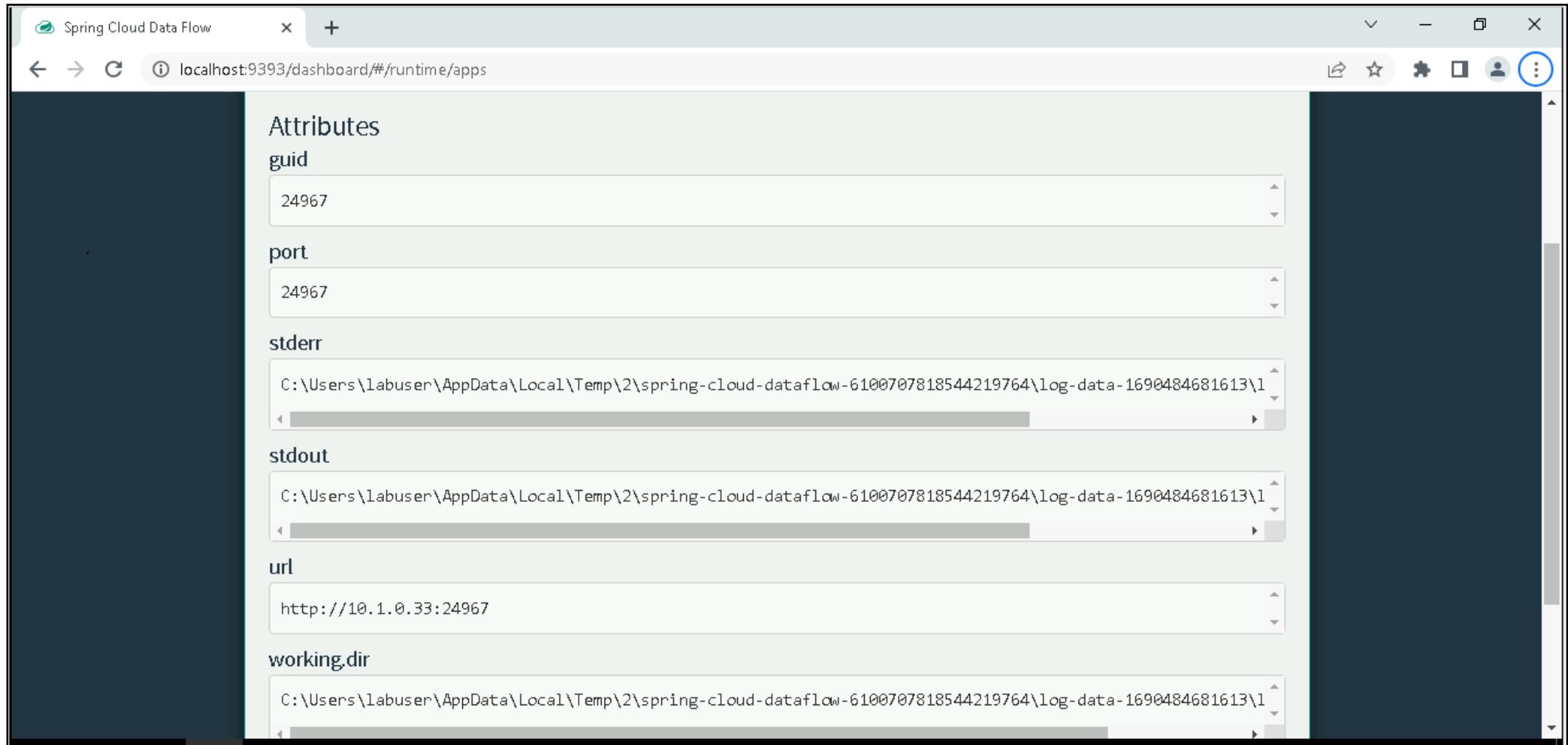
DOCUMENTATION
[Docs](#)
[Sources](#)
[Api Docs](#)

NEED HELP?
For questions + support:
[Stackoverflow](#)



Find the Sink logs

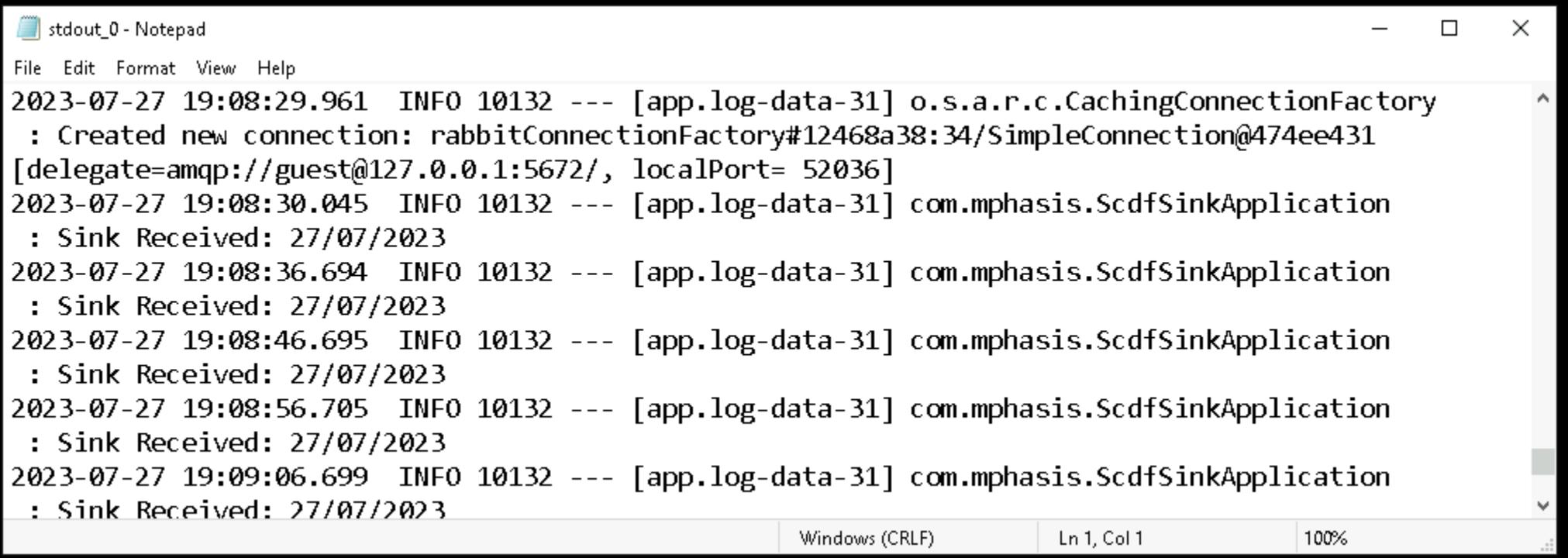
27. Go to working.dir path for logs:





Find the Sink logs

28. We can see Sink Log; the application is receiving the messages:



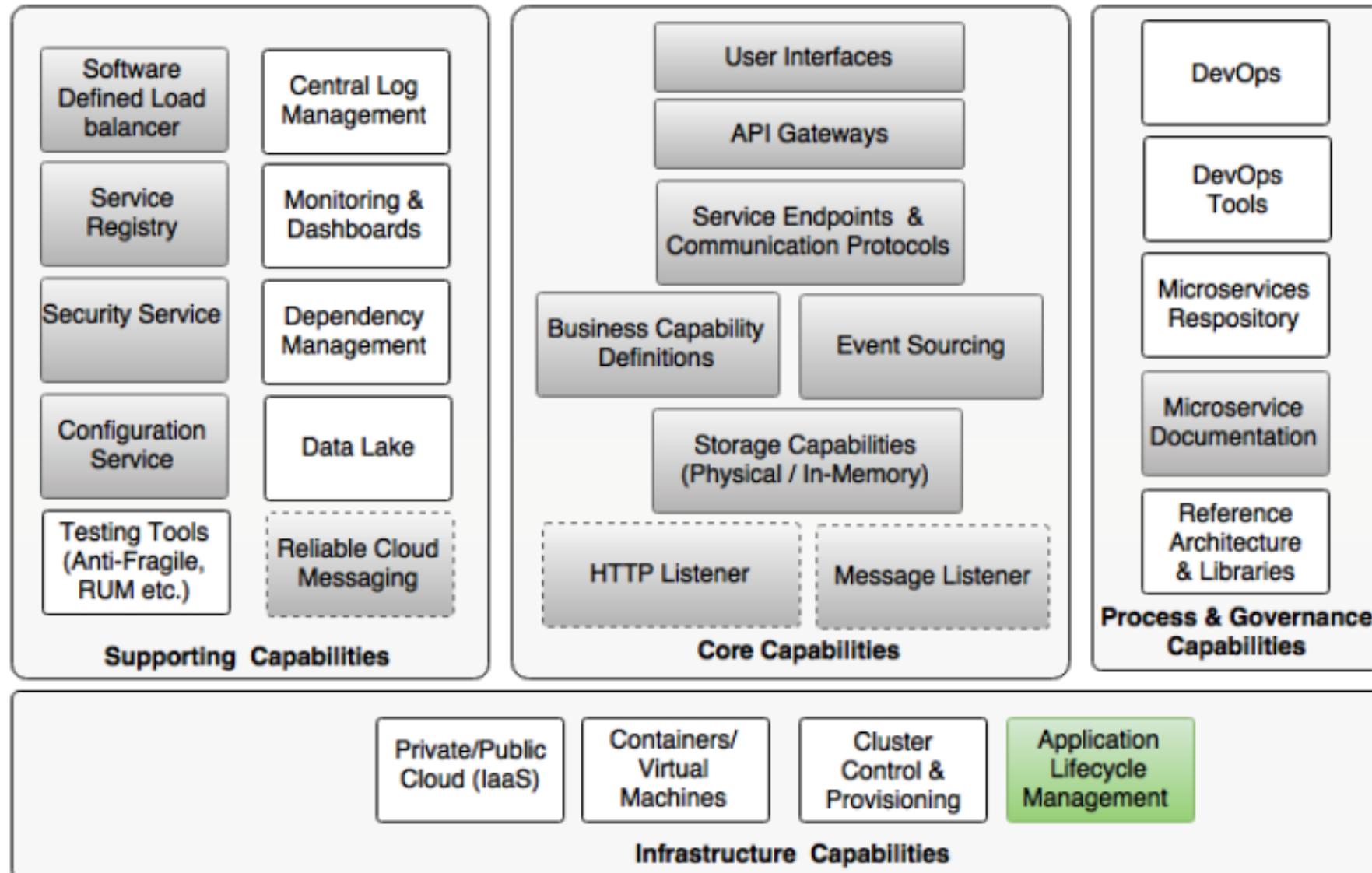
```
stdout_0 - Notepad
File Edit Format View Help
2023-07-27 19:08:29.961 INFO 10132 --- [app.log-data-31] o.s.a.r.c.CachingConnectionFactory
: Created new connection: rabbitConnectionFactory#12468a38:34/SimpleConnection@474ee431
[delegate=amqp://guest@127.0.0.1:5672/, localPort= 52036]
2023-07-27 19:08:30.045 INFO 10132 --- [app.log-data-31] com.mphasis.ScdfSinkApplication
: Sink Received: 27/07/2023
2023-07-27 19:08:36.694 INFO 10132 --- [app.log-data-31] com.mphasis.ScdfSinkApplication
: Sink Received: 27/07/2023
2023-07-27 19:08:46.695 INFO 10132 --- [app.log-data-31] com.mphasis.ScdfSinkApplication
: Sink Received: 27/07/2023
2023-07-27 19:08:56.705 INFO 10132 --- [app.log-data-31] com.mphasis.ScdfSinkApplication
: Sink Received: 27/07/2023
2023-07-27 19:09:06.699 INFO 10132 --- [app.log-data-31] com.mphasis.ScdfSinkApplication
: Sink Received: 27/07/2023
```



Day - 10

Autoscaling Microservices

Reviewing the microservice capability model

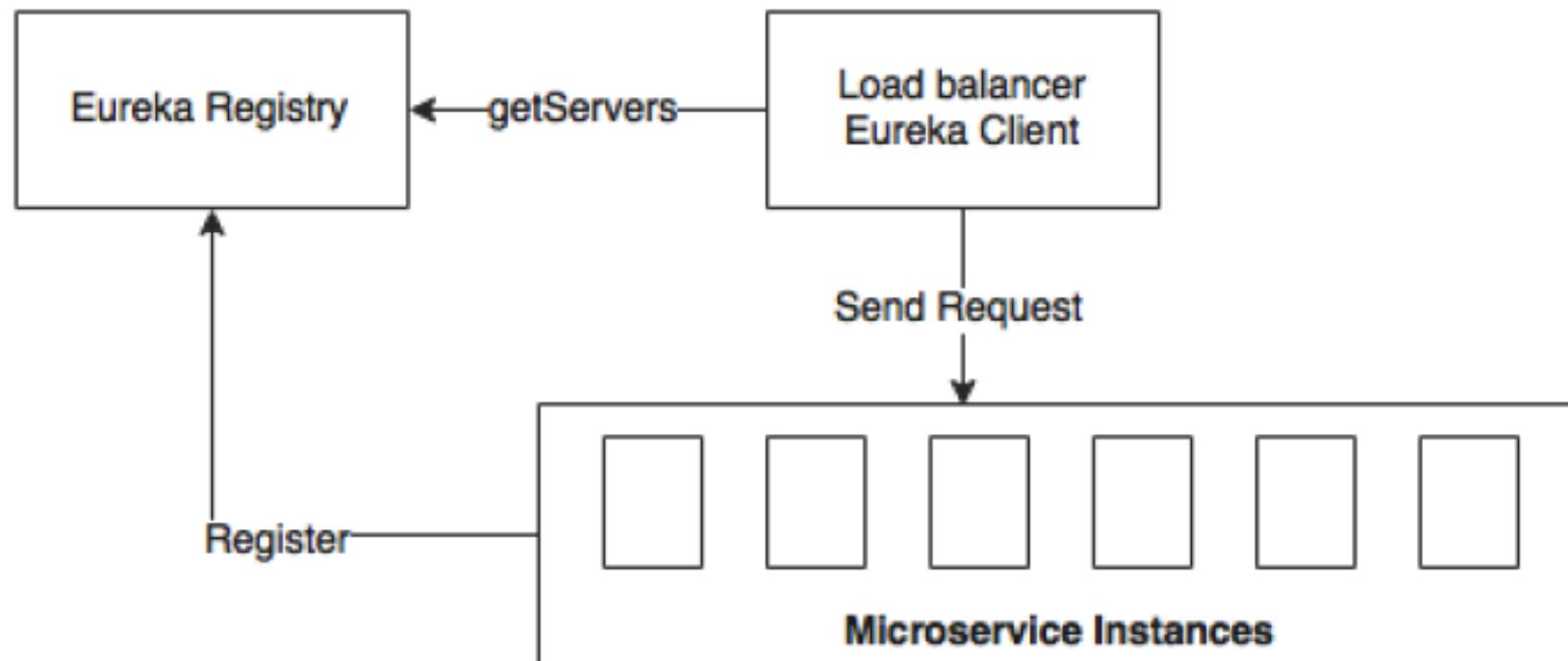




Scaling microservices with Spring Cloud

- The two key concepts of Spring Cloud that we implemented are **self-registration** and **self-discovery**.
- These two capabilities enable automated microservices deployments.
- With self-registration, microservices can automatically advertise the service availability by registering service metadata to a central service registry as soon as the instances are ready to accept traffic.
- Once the microservices are registered, consumers can consume the newly registered services from the very next moment by discovering service instances using the registry service. Registry is at the heart of this automation.

- The registry approach decouples the service instances.
- It also eliminates the need to manually maintain service addresses in the load balancer or configure virtual IPs:

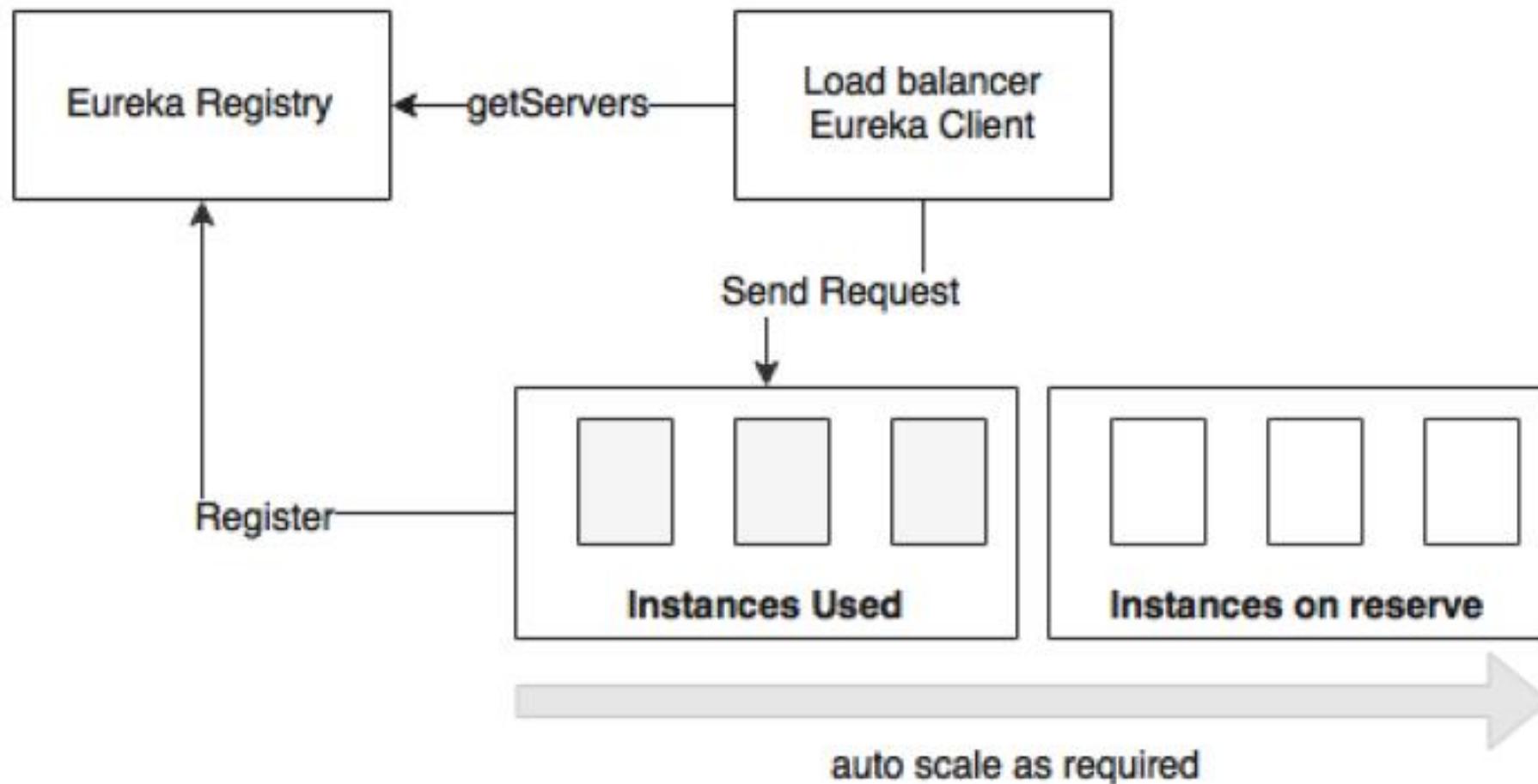




Understanding the concept of autoscaling

- Autoscaling is an approach to automatically scaling out instances based on the resource usage to meet the SLAs by replicating the services to be scaled.
- The system automatically detects an increase in traffic, spins up additional instances, and makes them available for traffic handling.
- Similarly, when the traffic volumes go down, the system automatically detects and reduces the number of instances by taking active instances back from the service:

Understanding the concept of autoscaling





The benefits of autoscaling

- It has high availability and is fault tolerant
- It increases scalability
- It has optimal usage and is cost saving
- It gives priority to certain services or group of services



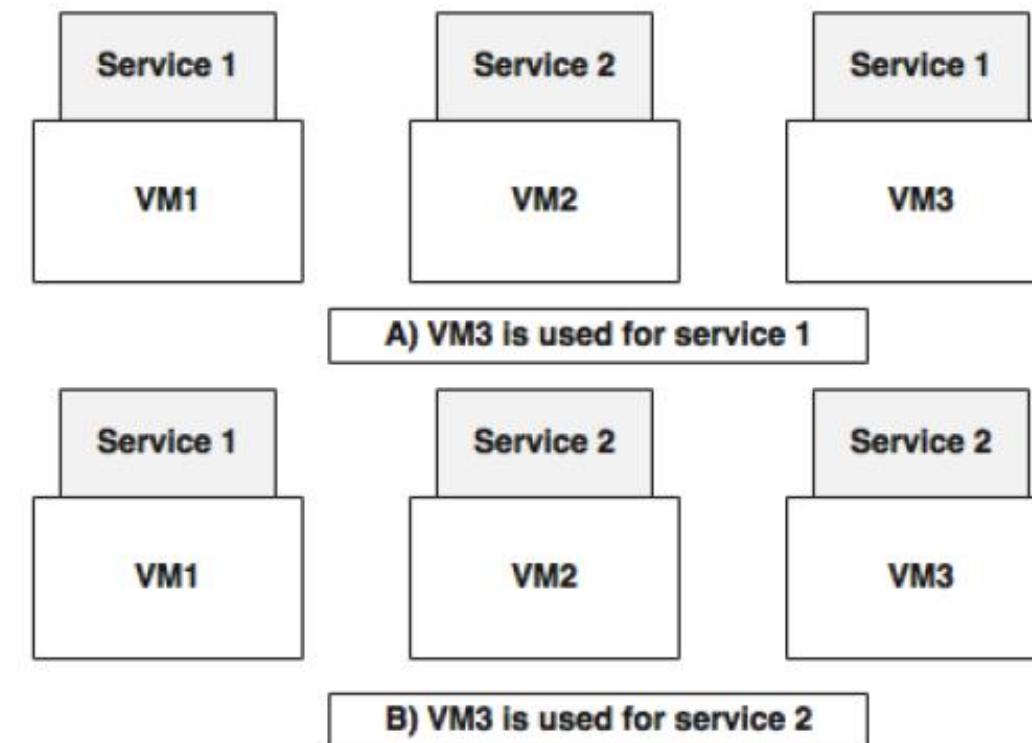
Different autoscaling models

- Autoscaling can be applied at the application level or at the infrastructure level.
- In a nutshell, application scaling is scaling by replicating application binaries only, whereas infrastructure scaling is replicating the entire virtual machine, including application binaries.



Autoscaling an application

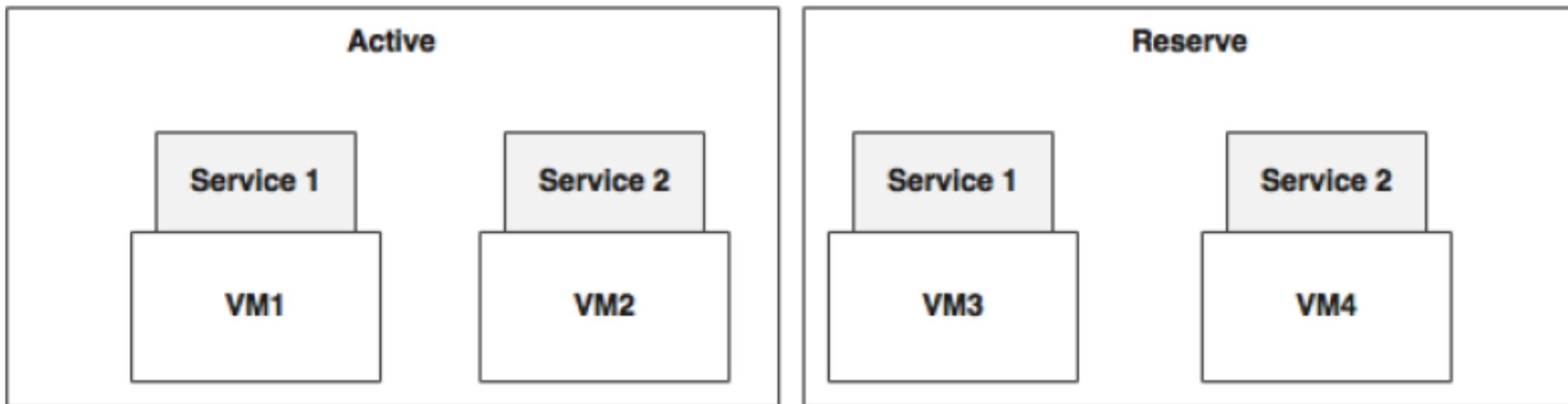
- In this scenario, scaling is done by replicating the microservices, not the underlying infrastructure, such as virtual machines.
- This gives flexibility in reusing the same virtual or physical machines for different services:





Autoscaling an infrastructure

- In most cases, this will create a new VM on the fly or destroy the VMs based on the demand:





Day - 10

Autoscaling Approaches



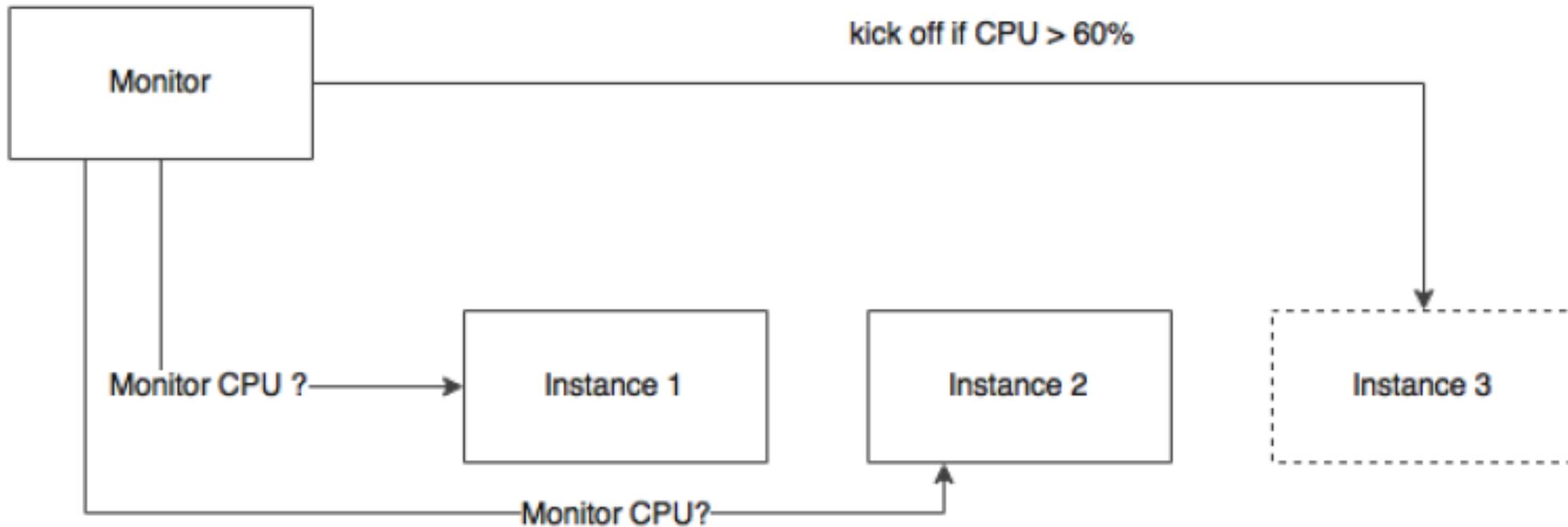
Autoscaling approaches

- Autoscaling is handled by considering different parameters and thresholds.
- In this section, we will discuss the different approaches and policies that are typically applied to take decisions on when to scale up or down.



1. Scaling with resource constraints

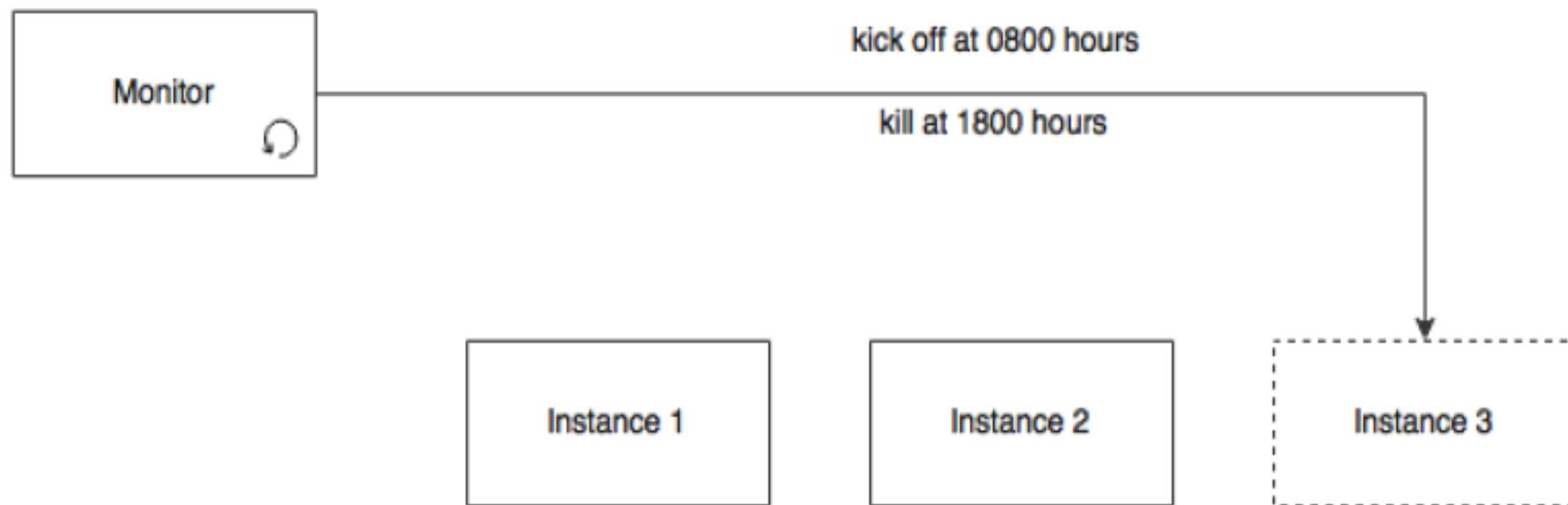
- This approach is based on real-time service metrics collected through monitoring mechanisms.





2. Scaling during specific time periods

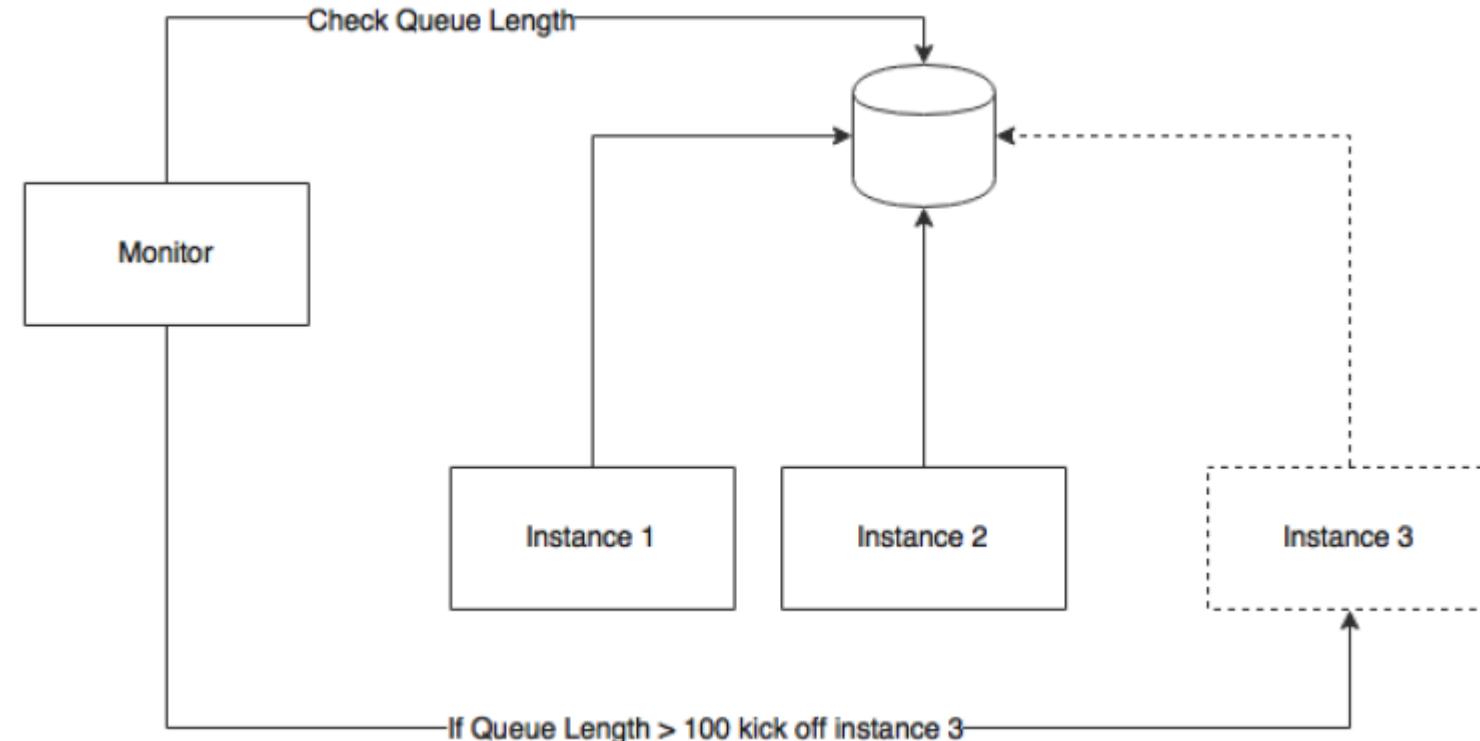
- Time-based scaling is an approach to scaling services based on certain periods of the day, month, or year to handle seasonal or business peaks.





3. Scaling based on the message queue length

- This is particularly useful when the microservices are based on asynchronous messaging.
- In this approach, new consumers are automatically added when the messages in the queue go beyond certain limits:





4. Scaling based on business parameters

- In this case, adding instances is based on certain business parameters





5. Predictive autoscaling

- Predictive scaling is a new paradigm of autoscaling that is different from the traditional real-time metrics-based autoscaling.
- A prediction engine will take multiple inputs, such as historical information, current trends, and so on, to predict possible traffic patterns.
- Autoscaling is done based on these predictions.
- Predictive autoscaling helps avoid hardcoded rules and time windows.
- Instead, the system can automatically predict such time windows.
- In more sophisticated deployments, predictive analysis may use **cognitive computing mechanisms** to predict autoscaling.



5. Predictive autoscaling

- In the cases of sudden traffic spikes, traditional autoscaling may not help. Before the autoscaling component can react to the situation, the spike would have hit and damaged the system. The predictive system can understand these scenarios and predict them before their actual occurrence. An example will be handling a flood of requests immediately after a planned outage.
- **Netflix Scryer** is an example of such a system that can predict resource requirements in advance.



Recap of Day – 10

- Event Driven Microservices with Spring Cloud Stream and RabbitMQ
- Stream Processing Using Spring Cloud Data Flow
- Autoscaling microservices
- Scaling microservices with Spring Cloud
- Understanding the concept of autoscaling
- The benefits of autoscaling
- Different autoscaling models
- Autoscaling Approaches



Recap of overall Agenda

- Understanding of SOA or similar design principles
- Understanding of MVC - Spring or any Framework
- Understanding of Micro Services architecture
- Compare Micro Services with other architectural styles (SOA/Monolithic applications)
- Understand 12 factor principles for micro service architecture
- Ability to develop a stand-alone micro service
- Understand inter-service communication
- Hands on experience with API (Spring boot, Data, REST web services)
- Deploy services that use Netflix Eureka, Hystrix, and Ribbon to create resilient and scalable services



Recap of overall Agenda

- Distributed Tracing with Sleuth and Zipkin
- Understanding the replacement available in new Spring Cloud version
- Implementing Circuit Breaker using Resilience4J
- Client-Side Load-Balancing with Spring Cloud Load Balancer
- Monitoring Spring Boot with Micrometer and Prometheus
- Implementing Message-Driven Microservices using Spring Cloud Stream & RabbitMQ
- Publishing Custom Events with Spring Cloud Bus
- Understanding Stream Processing using Spring Cloud Data Flow
- Autoscaling microservices



Additional Recommended Reading

- [Spring Microservices in Action](#)
- [Spring Boot Intermediate Microservices](#)
- [Enterprise Java Microservices](#)



THANK YOU

About Mphasis

Mphasis (BSE: 526299; NSE: MPHASIS) applies next-generation technology to help enterprises transform businesses globally. Customer centricity is foundational to Mphasis and is reflected in the Mphasis' Front2Back™ Transformation approach. Front2Back™ uses the exponential power of cloud and cognitive to provide hyper-personalized ($C=X^2C^2$) digital experience to clients and their end customers. Mphasis' Service Transformation approach helps 'shrink the core' through the application of digital technologies across legacy environments within an enterprise, enabling businesses to stay ahead in a changing world. Mphasis' core reference architectures and tools, speed and innovation with domain expertise and specialization are key to building strong relationships with marquee clients. Click [here](#) to know

Important Confidentiality Notice

This document is the property of, and is proprietary to Mphasis, and identified as "Confidential". Those parties to whom it is distributed shall exercise the same degree of custody and care afforded their own such information. It is not to be disclosed, in whole or in part to any third parties, without the express written authorization of Mphasis. It is not to be duplicated or used, in whole or in part, for any purpose other than the evaluation of, and response to, Mphasis' proposal or bid, or the performance and execution of a contract awarded to Mphasis. This document will be returned to Mphasis upon request.



Any Questions?

Manpreet.Bindra@mphasis.com

FOLLOW US IN THE LINK BELOW

@Mphasis

