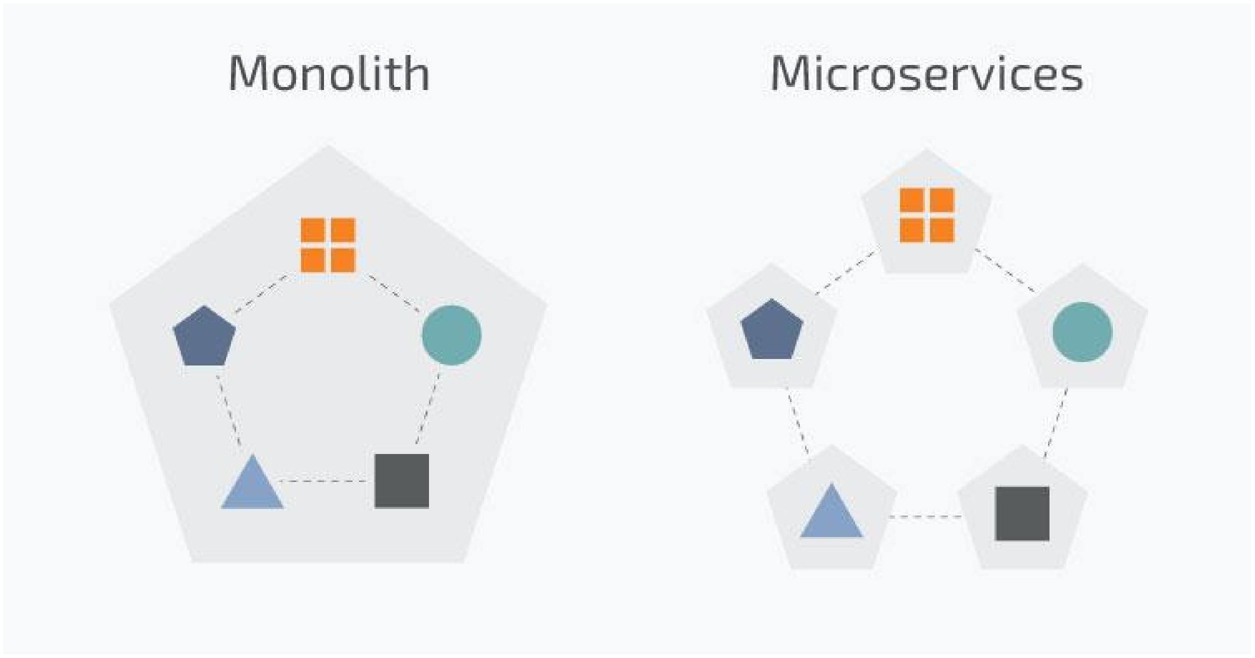
Microservices vs Monolith: which architecture is the best choice for your business?

Having come into light just a few years ago, microservices are an accelerating trend these days. Indeed, microservices approach offers tangible benefits including an increase in scalability, flexibility, agility, and other significant advantages. Netflix, Google, Amazon, and other tech leaders have successfully switched from monolithic architecture to microservices. Meanwhile, many companies consider following this example as the most efficient way for business growth.

On the contrary, the monolithic approach is a default model for creating a software application. Still, its trend is going down because building a monolithic application poses a number of challenges associated with handling a huge code base, adopting a new technology, scaling, deployment, implementing new changes and others.



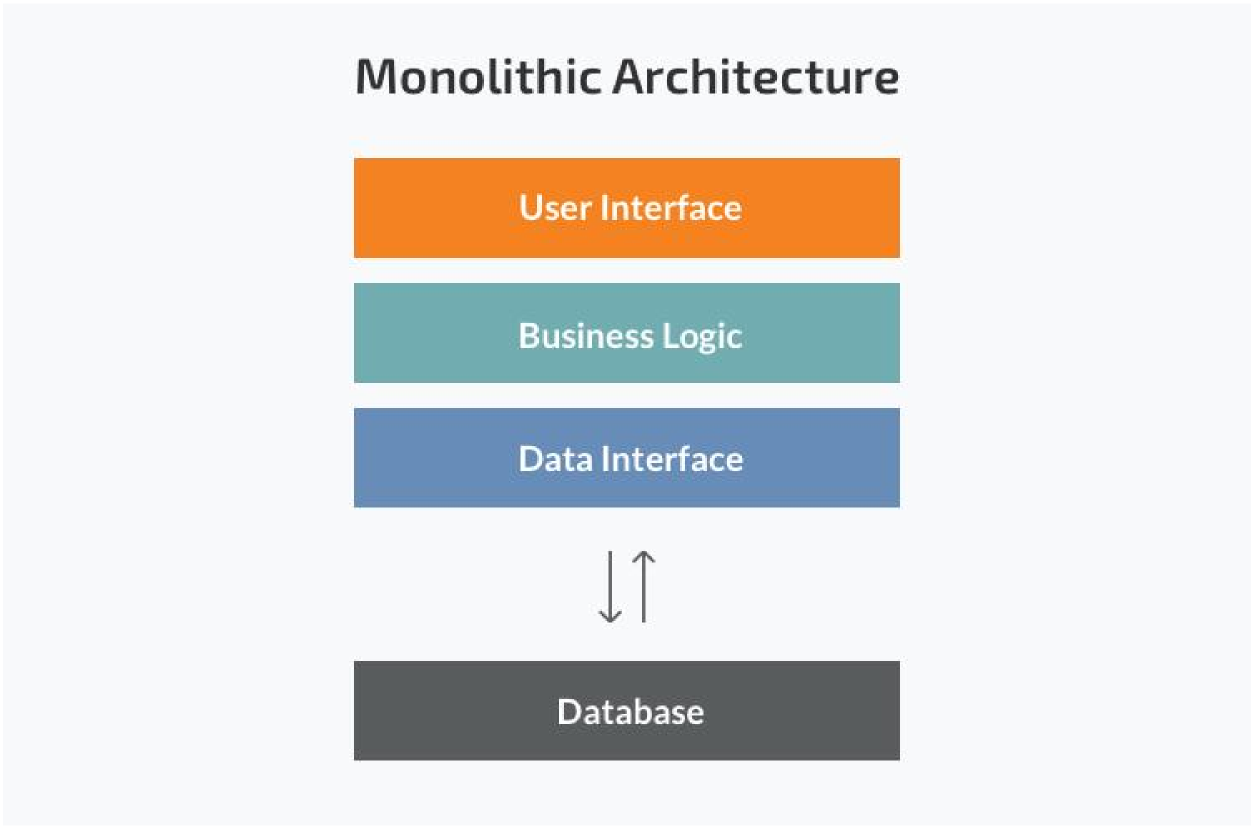
So is the monolithic approach outdated and should be left in the past? And is it worth shifting the whole application from a monolith to microservices? Will developing a microservices application help you reach your business goals?

In this article, we are going to compare microservices with the monolithic architecture, outline the strengths and weaknesses of both approaches, and find out which software architecture style will be the best fit for your business.

MONOLITHIC ARCHITECTURE

The monolithic architecture is considered to be a traditional way of building applications. A monolithic application is built as a single and indivisible unit. Usually, such a solution comprises a client-side user interface, a server side-application, and a database. It is unified and all the functions are managed and served in one place.

Normally, monolithic applications have one large code base and lack modularity. If developers want to update or change something, they access the same code base. So, they make changes in the whole stack at once.



Strengths of the Monolithic Architecture

- Less cross-cutting concerns. Cross-cutting concerns are the concerns that affect the whole application such as logging, handling, caching, and performance monitoring. In a monolithic application, this area of functionality concerns only one application so it is easier to handle it.
- Easier debugging and testing. In contrast to the microservices architecture, monolithic applications are much easier to debug and test. Since a monolithic app is a single indivisible unit, you can run end-to-end testing much faster.
- Simple to deploy. Another advantage associated with the simplicity of monolithic apps is easier deployment. When it comes to monolithic applications, you do not have to handle many deployments – just one file or directory.
- Simple to develop. As long as the monolithic approach is a standard way of building applications, any engineering team has the right knowledge and capabilities to develop a monolithic application.

Weaknesses of the Monolithic Architecture

- Understanding. When a monolithic application scales up, it becomes too complicated to understand. Also, a complex system of code within one application is hard to manage.
- Making changes. It is harder to implement changes in such a large and complex application with highly tight coupling. Any code change affects the whole system so it has to be thoroughly coordinated. This makes the overall development process much longer.
- Scalability. You cannot scale components independently, only the whole application.
- New technology barriers. It is extremely problematic to apply a new technology in a monolithic application because then the entire application has to be rewritten.

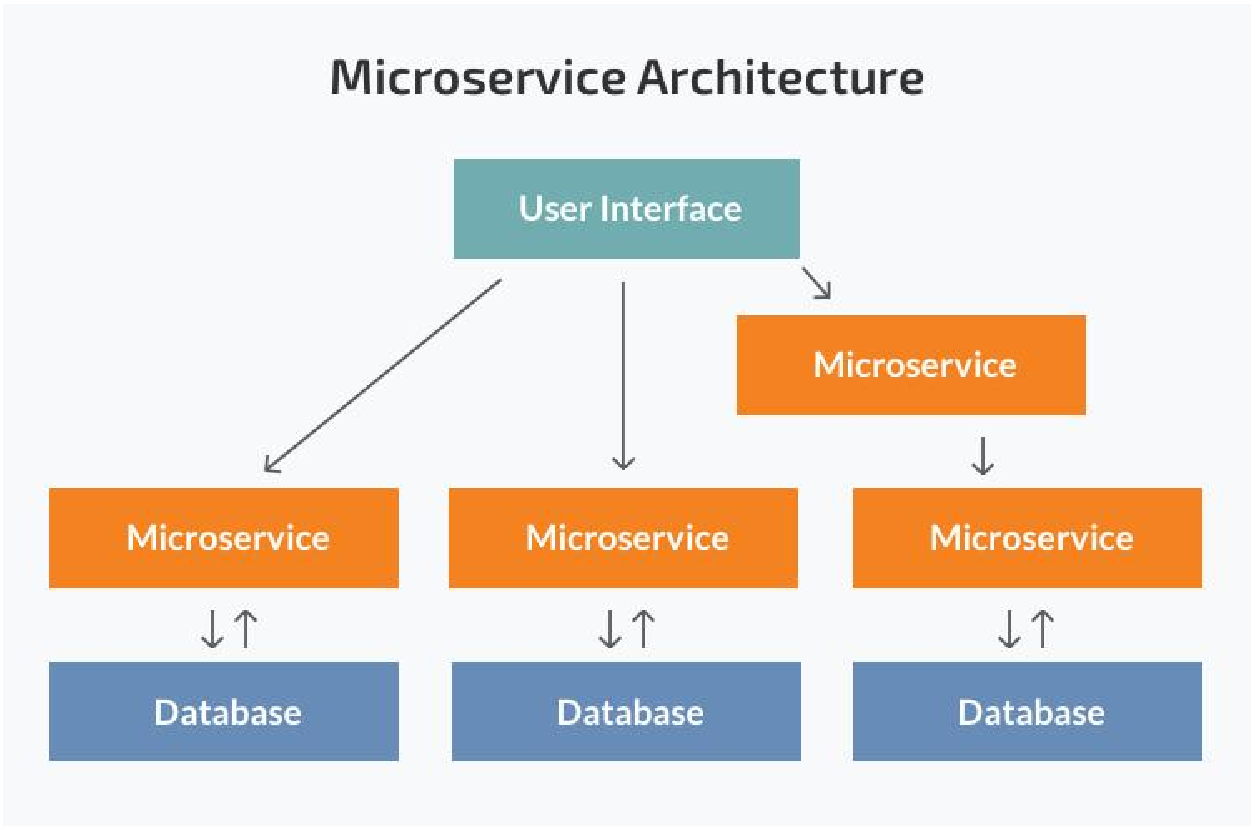
MICROSERVICES ARCHITECTURE

While a monolithic application is a single unified unit, a microservices architecture breaks it down into a collection of smaller independent units. These units carry out every application process as a separate service. So all the services have their own logic and the database as well as perform the specific functions.

In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

Martin Fowler

Within a microservices architecture, the entire functionality is split up into independently deployable modules which communicate with each other through defined methods called APIs (Application Programming Interfaces). Each service covers its own scope and can be updated, deployed, and scaled independently.



Strengths of the Microservice Architecture

- Independent components. Firstly, all the services can be deployed and updated independently, which gives more flexibility. Secondly, a bug in one microservice has an impact only on a particular service and does not influence the entire application. Also, it is much easier to add new features to a microservice application than a monolithic one.
- Easier understanding. Split up into smaller and simpler components, a microservice application is easier to understand and manage. You just concentrate on a specific service that is related to a business goal you have.
- Better scalability. Another advantage of the microservices approach is that each element can be scaled independently. So the entire process is more cost- and time-effective than with monoliths when the whole application has to be scaled even if there is no need in it. In addition, every monolith has limits in terms of scalability, so the more users you acquire, the more problems you have with your monolith. Therefore, many companies, end up rebuilding their monolithic architectures.

For instance, our partner Currencycloud needed to migrate to microservices due to the growing number of transactions their platform processed. Founded in 2012, the company offers a global B2B payments platfrom. Initially, their monolithic architecture could handle the number of transactions they had. Yet with the company's growing success, they needed a more efficient solution they could scale even further in the future. So they have switched to microservices.

- Flexibility in choosing the technology. The engineering teams are not limited by the technology chosen from the start. They are free to apply various technologies and frameworks for each microservice.
- The higher level of agility. Any fault in a microservices application affects only a particular service and not the whole solution. So all the changes and experiments are implemented with lower risks and fewer errors.

Weaknesses of the Microservice Architecture

- Extra complexity. Since a microservices architecture is a distributed system, you have to choose and set up the connections between all the modules and databases. Also, as long as such an application includes independent services, all of them have to be deployed independently.
- System distribution. A microservices architecture is a complex system of multiple modules and databases so all the connections have to be handled carefully.
- Cross-cutting concerns. When creating a microservices application, you will have to deal with a number of cross-cutting concerns. They include externalized configuration, logging, metrics, health checks, and others.
- Testing. A multitude of independently deployable components makes testing a microservices-based solution much harder.

SO WHICH SOFTWARE ARCHITECTURE SUITS YOUR SOLUTION AND YOUR BUSINESS BEST?

Choosing a monolithic architecture

- Small team. If you are a startup and your team is small, you may not need to deal with the complexity of the microservices architecture. A monolith can meet all your business needs so there is no emergency to follow the hype and start with microservices.
- A simple application. Small applications which do not demand much business logic, superior scalability, and flexibility work better with monolithic architectures.
- No microservices expertise. Microservices require profound expertise to work well and bring business value. If you want to start a microservices application from scratch with no technical expertise in it, most probably, it will not pay off.
- Quick launch. If you want to develop your application and launch it as soon as possible, a monolithic model is the best choice. It works well when you aim to spend less initially and validate your business idea.

Choosing a microservices architecture

- Microservices expertise. Without proper skills and knowledge, building a microservice application is extremely risky. Still, just having the architecture knowledge is not enough. You need to have DevOps and Containers experts since the concepts are tightly coupled with microservices. Also, domain modelling expertise is a must. Dealing with microservices means splitting the system into separate functionalities and dividing responsibilities.
- A complex and scalable application. The microservices architecture will make scaling and adding new capabilities to your application much easier. So if you plan to develop a large application with multiple modules and user journeys, a microservice pattern would be the best way to handle it.
- Enough engineering skills. Since a microservice project comprises multiple teams responsible for multiple services, you need to have enough resources to handle all the processes.

For example, one of our clients, a Fortune 100 company, partnered with N-iX to scale their solution. They built the logistics platform to improve the logistics between its 400+

warehouses in over 60 countries. Yet, after the solution was used for several months, it appeared to be ineffective. It was hard for them to add new functionality and scale the monolithic platform. And scaling was essential as our client has many factories, warehouses, and suppliers, as well as a lot of raw materials and finished goods, which circulate among them. Although our partner had a vision that they needed to migrate to microservices, they did not have the comprehensive in-house expertise to address multiple technical issues and make the platform more efficient and scalable.

The core reason why the platform was not scalable and inefficient was its monolithic architecture. Therefore, our Solution Architect designed and presented a new cloud-native infrastructure of the platform based on Azure Kubernetes, along with the suggested tech stack and the most efficient roadmap.

Migrating to microservices allows smooth adding of new SaaS services: anomaly detection, delivery prediction, route recommendations, object detection in logistics, OCR (optical character recognition) of labels on boxes, Natural Language Processing for document verification, data mining, and sensor data processing.

AFTERWORD

Adopting a microservices architecture is not a one-size-fits-all approach. Despite being less and less popular, a monolith has its strong and durable advantages which work better for many use cases.

If your business idea is fresh and you want to validate it, you should start with a monolith. With a small engineering team aiming to develop a simple and lightweight application, there is no need to implement microservices. This way, a monolithic application will be much easier to build, make changes, deploy, and provide testing.

The microservices architecture is more beneficial for complex and evolving applications. It offers effective solutions for handling a complicated system of different functions and services within one application. Microservices are ideal when it comes to the platforms covering many user journeys and workflows. But without proper microservices expertise, applying this model would be impossible.

Microservices Architecture Spring Boot **Table of Contents**

1. Introduction

Microservices is not a new term. It was coined in 2005 by Dr Peter Rodgers then called micro web services based on SOAP. It has become more popular since 2010. Microservices allows us to break our large system into a number of independent collaborating processes. Let us see below microservices architecture.

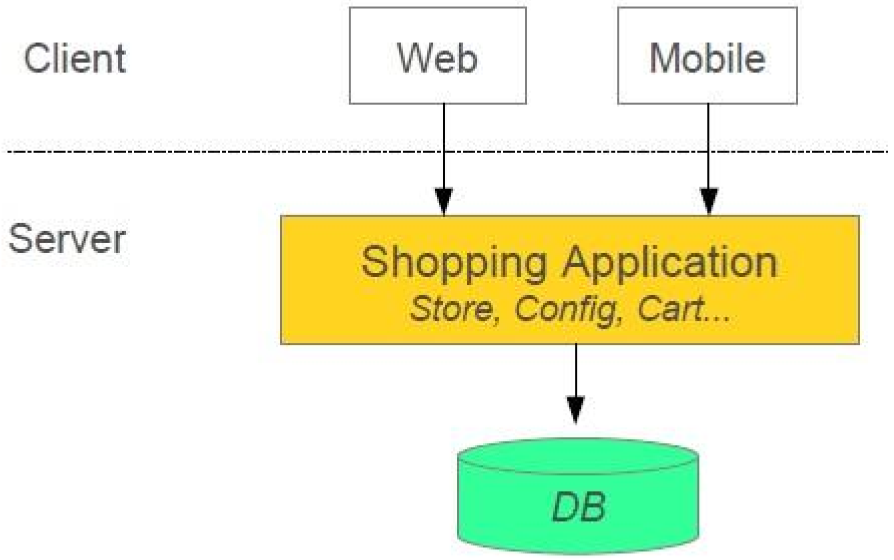
1.1 What is Microservices Architecture?

Microservices architecture allows avoiding monolith application for the large system. It provides loose coupling between collaborating processes which run independently in different environments with tight cohesion. So let's discuss it by an example as below.

For example imagine an online shop with separate microservices for user-accounts, product-catalog order-processing and shopping carts. So these components are inevitably important for such a large online shopping portal. For an online shopping system, we could use the following architectures.

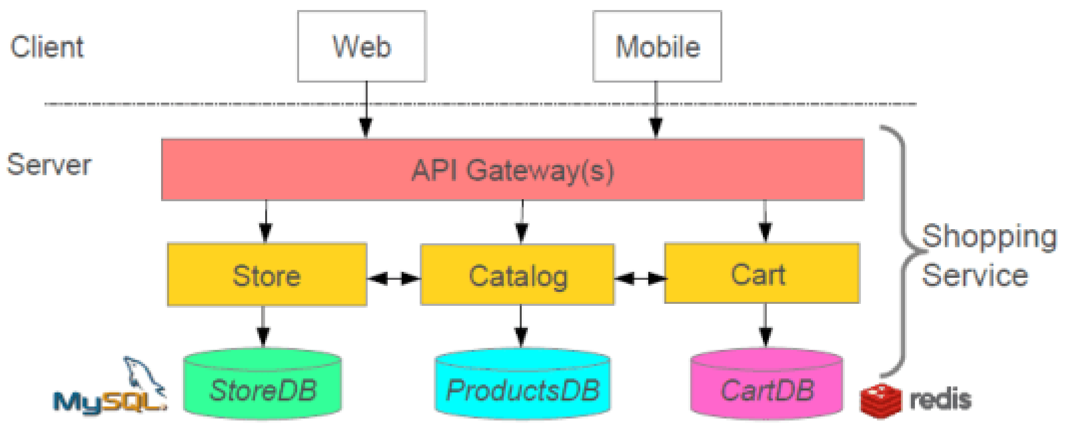
1.2 Shopping system without Microservices (Monolith architecture)

In this architecture we are using Monolith architecture i.e. all collaborating components combine all in one application.



1.3 Shopping system with Microservices

In this architecture style, the main application divided into a set of sub-applications called microservices. One large Application divided into multiple collaborating processes as below.



Spring enables separation-of-concerns

- ***Loose Coupling***– Effect of changes isolated
- ***Tight Cohesion***– Code perform a single well-defined task

Microservices provide the same strength as Spring provide

- ***Loose Coupling***– Application build from collaboration services or processes, so any process change without affecting another process.
- ***Tight Cohesion***-An individual service or process that deals with a single view of data.

There are a number of moving parts that you have to set up and configure to build such a system. For implementing this system is not too obvious you have to know about spring boot, spring cloud and Netflix. In this post, I will discuss one example for this architecture before the example lets first discuss pros and cons of microservices architecture.

2. Microservices Benefits

- The smaller code base is easy to maintain.
- Easy to scale as an individual component.

- Technology diversity i.e. we can mix libraries, databases, frameworks etc.
- Fault isolation i.e. a process failure should not bring the whole system down.
- Better support for smaller and parallel team.
- Independent deployment
- Deployment time reduce

3. Microservices Challenges

- Difficult to achieve strong consistency across services
- ACID transactions do not span multiple processes.
- Distributed System so hard to debug and trace the issues
- Greater need for an end to end testing
- Required cultural changes in across teams like Dev and Ops working together even in the same team.

4. Microservices Infrastructure

- Platform as a Service like Pivotal Cloud Foundry help to deployment, easily run, scale, monitor etc.
- It supports for continuous deployment, rolling upgrades fo new versions of code, running multiple versions of the same service at same time.

5. Microservices Tooling Supports

5.1 Using Spring for creating Microservices

- Setup new service by using Spring Boot
- Expose resources via a RestController
- Consume remote services using RestTemplate

5.2 Adding Spring Cloud and Discovery server

What is Spring Cloud?

- It is building blocks for Cloud and Microservices
- It provides microservices infrastructure like provide use services such as Service Discovery, a Configuration server and Monitoring.
- It provides several other open source projects like Netflix OSS.
- It provides PaaS like Cloud Foundry, AWS and Heroku.
- It uses Spring Boot style starters

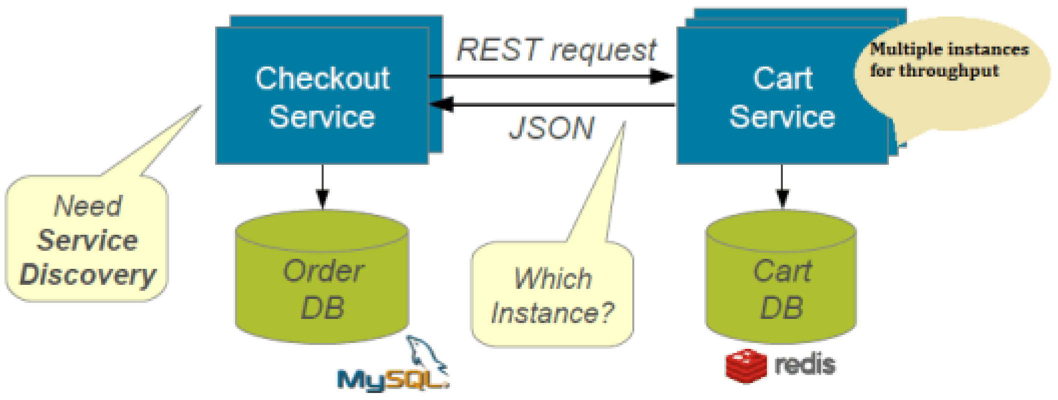
There are many use-cases supported by Spring Cloud like Cloud Integration, Dynamic Reconfiguration, Service Discovery, Security, Client-side Load Balancing etc. But in this post we concentrate on following microservices support

- Service Discovery (How do services find each other?)
- Client-side Load Balancing (How do we decide which service instance to use?)

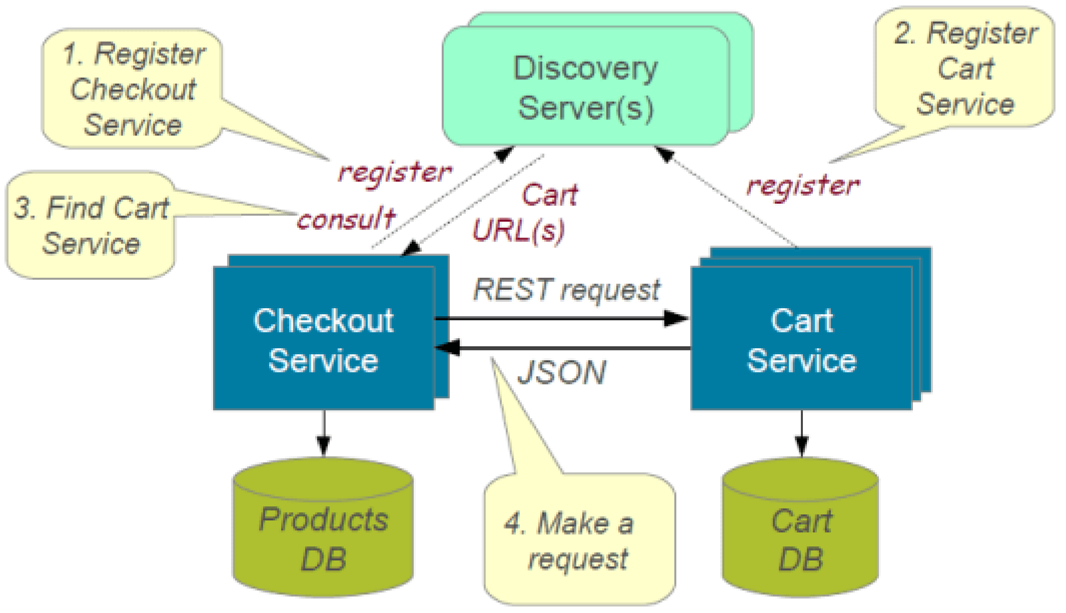
Service Discovery

Problem without discovery

- How do services find each other?
- What happens if we run multiple instances for a service



Resolution with service discovery



Implementing Service Discovery

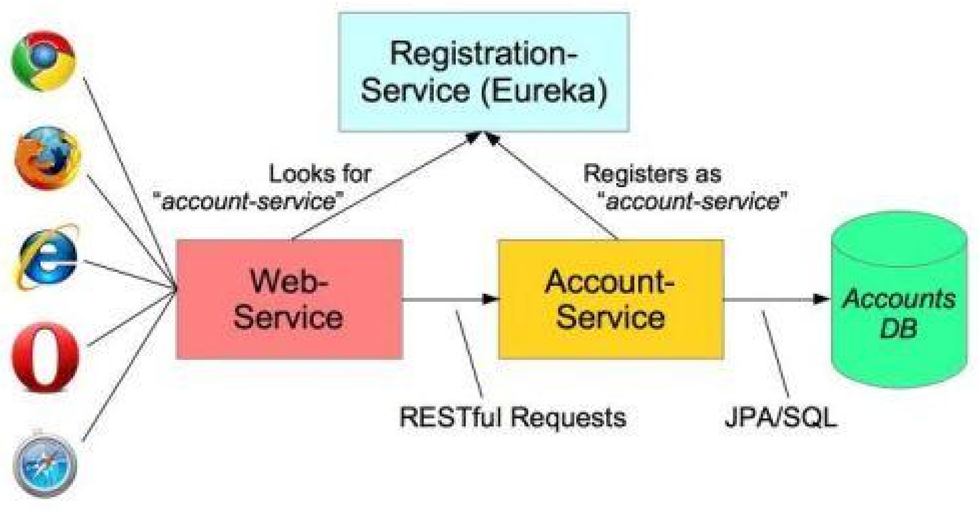
Spring Cloud support several ways to implement service discovery but for this, I am going to use Eureka created by Netflix. Spring Cloud provides several annotation to make it use easy and hiding lots of complexity.

Client-side Load Balancing

Each service typically deployed as multiple instances for fault tolerance and load sharing. But there is the problem how to decide which instance to use?

Implementing Client-Side Load Balancing

We will use Netflix Ribbon, it provides several algorithms for Client-Side Load Balancing. Spring provides smart **RestTemplate** for service discovery and load balancing by using ***@LoadBalanced*** annotation with **RestTemplate** instance.


Registration-Service (Eureka)
Looks for "account-service"
Registers as "account-service"
Web-Service
Account-Service
Accounts DB
RESTful Requests
JPA/SQL

6. Developing Simple Microservices Example

To build a simple microservices system following steps required

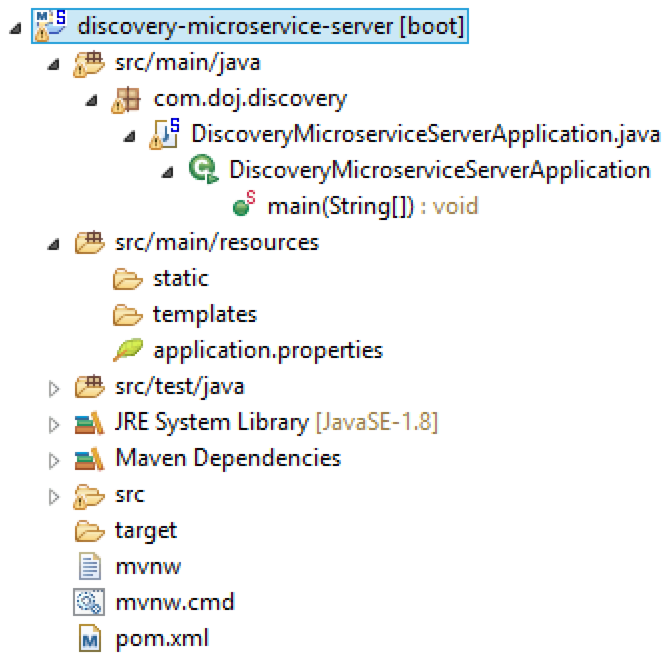
1. Creating Discovery Service (Creating Eureka Discovery Service)
2. Creating MicroService (the Producer)
    1. Register itself with Discovery Service with logical service.
3. Create Microservice Consumers find Service registered with Discovery Service
    1. Discovery client using a smart **RestTemplate** to find microservice.

Maven Dependencies

```
<dependencies>
 <dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter</artifactId>
 </dependency>
 <dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
 </dependency>
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
 </dependency>

 <dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <scope>runtime</scope>
 </dependency>
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
 </dependency>
 </dependencies>
```

Step 1: Creating Discovery Service (Creating Eureka Discovery Service)

- Eureka Server using Spring Cloud
- We need to implement our own registry service as below.



discovery-microservice-server [boot]
src/main/java
com.doj.discovery
DiscoveryMicroserviceServerApplication.java
DiscoveryMicroserviceServerApplication
main(String[]) : void
src/main/resources
static
templates
application.properties
src/test/java
JRE System Library [JavaSE-1.8]
Maven Dependencies
src
target
mvnw
mvnw.cmd
pom.xml

In Spring Boot, we can use YAML files instead of properties files. ... YAML stands for YAML Ain't Markup Language (a recursive acronym). YAML (from version 1.2) is a superset of JSON and is a very convenient format for specifying hierarchical configuration data.

application.yml

```
# Configure this Discovery Server
eureka:
  instance:
    hostname: localhost
  client: #Not a client
    registerWithEureka: false
    fetchRegistry: false

# HTTP (Tomcat) port
server:
  port: 1111
```

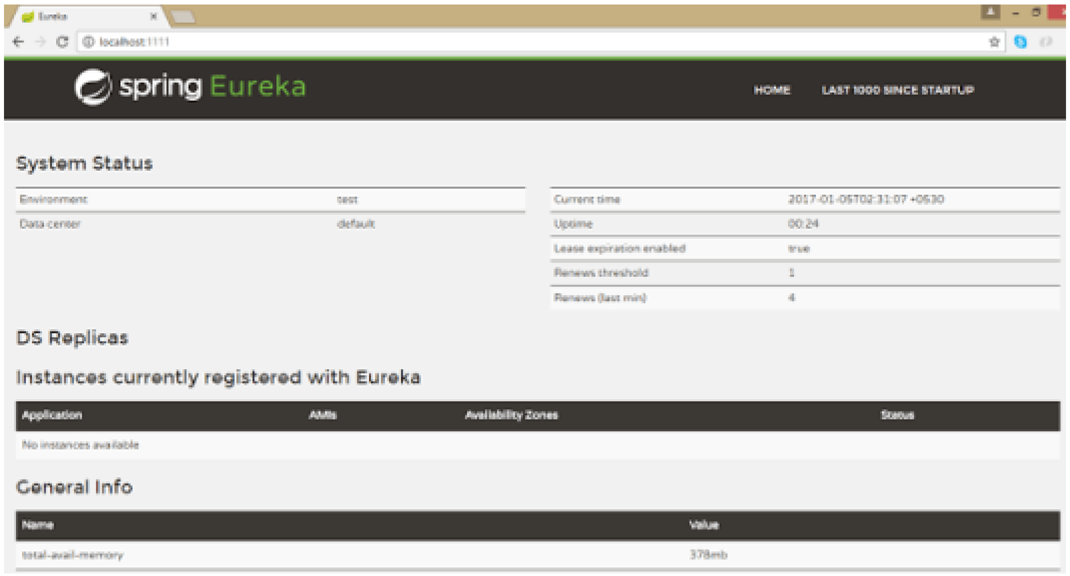
DiscoveryMicroserviceServerApplication.java

```
package com.doj.discovery;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class DiscoveryMicroserviceServerApplication {

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

pom.xml

```
<!-- Eureka registration server -->
  <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-eureka-server</artifactId>
  </dependency>
```

Run this Eureka Server application with right click and run as Spring Boot Application and open in browser **http://localhost:1111/**



System Status

Environment	test
Data center	default

Current time	2017-01-05T02:31:07 +0530
Uptime	00:24
Lease expiration enabled	true
Renews threshold	1
Renews (last min)	4

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

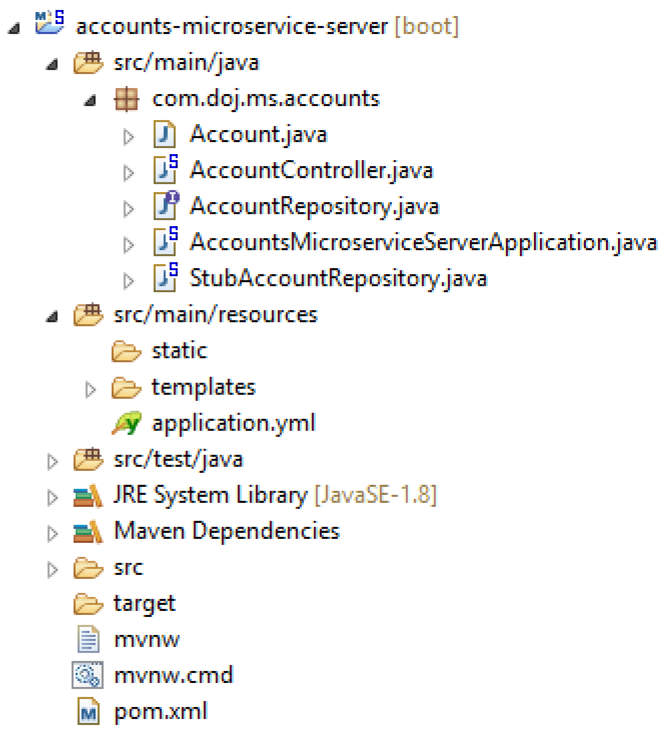
Name	Value
total-avail-memory	378mb

Step 2: Creating Account Producer MicroService

Microservice declares itself as an available service and register to Discovery Server created in **Step 1**.

- Using ***@EnableDiscoveryClient***
- Registers using its application name

Let's see the service producer application structure as below.


```
accounts-microservice-server [boot]
  src/main/java
    com.doj.ms.accounts
      Account.java
      AccountController.java
      AccountRepository.java
      AccountsMicroserviceServerApplication.java
      StubAccountRepository.java
  src/main/resources
    static
    templates
    application.yml
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

application.yml

```
### Spring properties
# Service registers under this name
spring:
  application:
    name: accounts-microservice

# Discovery Server Access
eureka:
    client:
      serviceUrl:
        defaultZone: http://localhost:1111/eureka/
```

```
# HTTP Server (Tomcat) Port
server:
  port: 2222

# Disable Spring Boot's "Whitelabel" default error page, so we can use our own
error:
  whitelabel:
    enabled: false
```

AccountsMicroserviceServerApplication.java

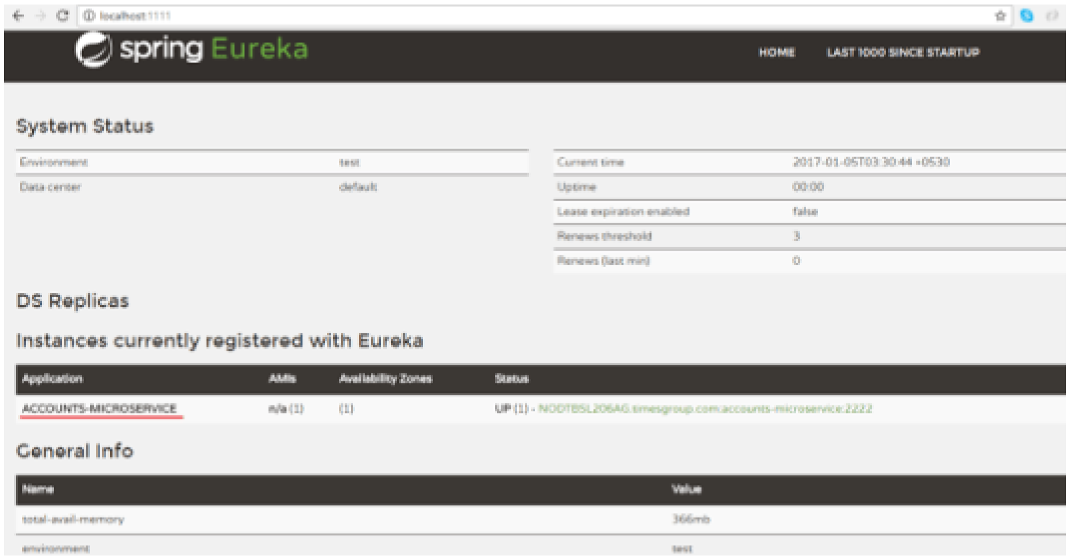
```
package com.doj.ms.accounts;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
public class AccountsMicroserviceServerApplication {

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

}
```

pom.xml

```
<dependencies>
  <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter</artifactId>
  </dependency>
  <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
```

```
 </dependency>

</dependencies>
```

Now run this account service application as **Spring Boot application** and after few seconds refresh the browser to the home page of **Eureka Discovery Server** at **http://localhost:1111/** in previous **Step 1**. Now one Service registered to the Eureka registered instances with Service Name “**ACCOUNT-MICROSERVICE**” as below



System Status

Environment	test
Data center	default

Current time	2017-01-05T03:30:44 +0530
Uptime	00:00
Lease expiration enabled	false
Renews threshold	3
Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ACCOUNTS-MICROSERVICE	n/a (1)	(1)	UP (1) - NODTBSL206AG.timesgroup.com:accounts-microservice:2222

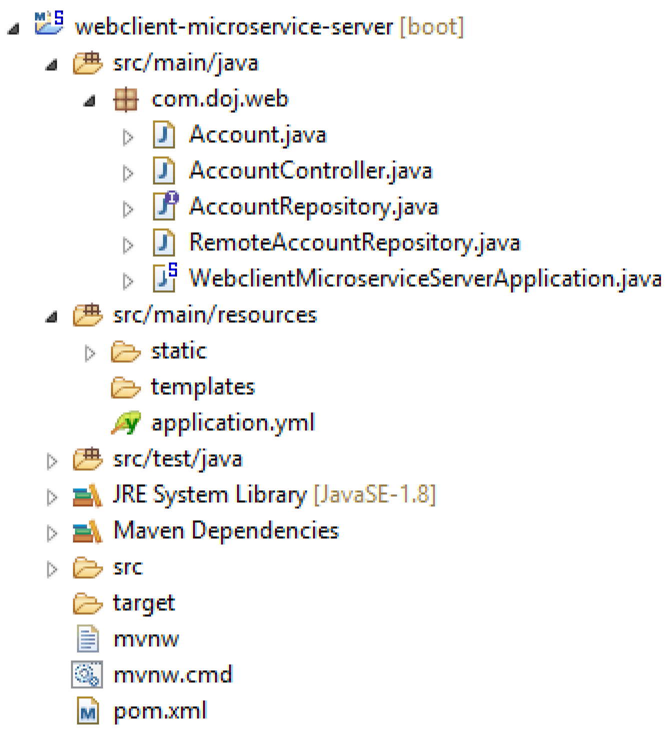
General Info

Name	Value
total-avail-memory	366mb
environment	test

Step 3: Consumer Service

- Create Consumers to find the Producer Service registered with Discovery Service at Step 1.
- ***@EnableDiscoveryClient*** annotation also allows us to query Discovery server to find microservices.

Let’s see the consumer application structure as below.



```
webclient-microservice-server [boot]
  src/main/java
    com.doj.web
      Account.java
      AccountController.java
      AccountRepository.java
      RemoteAccountRepository.java
      WebclientMicroserviceServerApplication.java
  src/main/resources
    static
    templates
    application.yml
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

application.yml

```
# Service registers under this name
# Control the InternalResourceViewResolver:
spring:
  application:
    name: accounts-web
  mvc:
    view:
      prefix: /WEB-INF/views/
      suffix: .jsp

# Discovery Server Access
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:1111/eureka/

# Disable Spring Boot's "Whitelabel" default error page, so we can use our own
error:
  whitelabel:
    enabled:  false
```

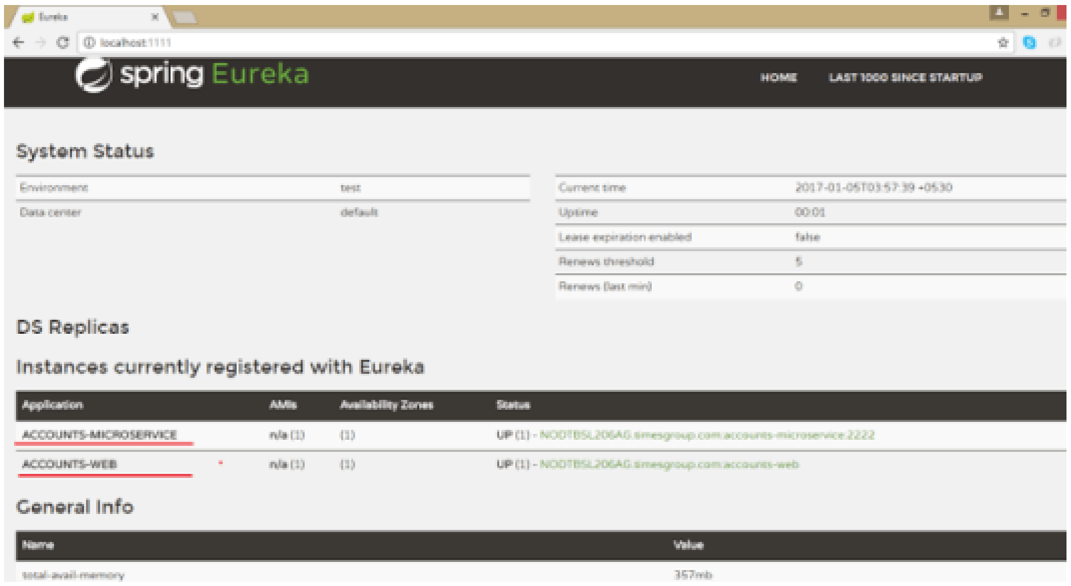
WebclientMicroserviceServerApplication.java

```java
package com.doj.web;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
@EnableDiscoveryClient
public class WebclientMicroserviceServerApplication {

 public static final String ACCOUNTS_SERVICE_URL = "http://ACCOUNTS-MICROSERVICE";

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

 @Bean
 @LoadBalanced
 public RestTemplate restTemplate() {
  return new RestTemplate();
 }
 @Bean
 public AccountRepository accountRepository(){
  return new RemoteAccountRepository(ACCOUNTS_SERVICE_URL);
 }
}
```

pom.xml

```xml
<dependencies>
  <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>
  <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-ribbon</artifactId>
  </dependency>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
   <scope>test</scope>
  </dependency>
  <!-- These dependencies enable JSP usage -->
  <dependency>
   <groupId>org.apache.tomcat.embed</groupId>
   <artifactId>tomcat-embed-jasper</artifactId>
   <scope>provided</scope>
  </dependency>
  <dependency>
   <groupId>javax.servlet</groupId>
   <artifactId>jstl</artifactId>
  </dependency>
 </dependencies>
```

Now run this consumer service application as **Spring Boot application** and after few seconds refresh the browser to the home page of **Eureka Discovery Server** at **http://localhost:1111/** in previous **Step 1**. Now one more Service registered to the Eureka registered instances with Service Name "**ACCOUNTS-WEB**" as below



System Status

Environment	test
Data center	default

Current time	2017-01-05T03:57:39 +0530
Uptime	00:01
Lease expiration enabled	false
Renews threshold	5
Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ACCOUNTS-MICROSERVICE	n/a (1)	(1)	UP (1) - NODTBSL206AG.timesgroup.com:accounts-microservice:2222
ACCOUNTS-WEB	n/a (1)	(1)	UP (1) - NODTBSL206AG.timesgroup.com:accounts-web

General Info

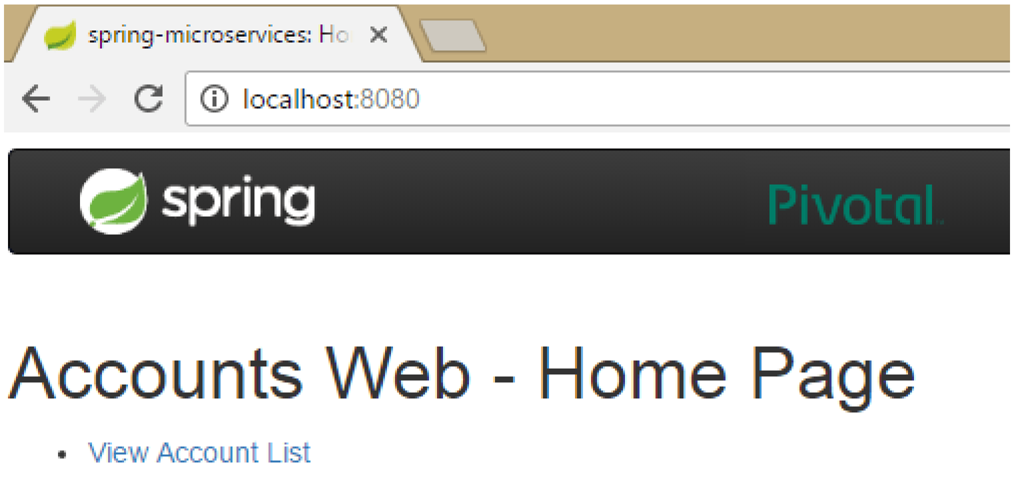
Name	Value
total-avail-memory	357mb

Lets our consumer consume the service of producer registered at discovery server.

```
package com.doj.web;

import java.util.Arrays;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.client.RestTemplate;

/**
 * @author Dinesh.Rajput
 *
 */
public class RemoteAccountRepository implements AccountRepository {

 @Autowired
 protected RestTemplate restTemplate;

 protected String serviceUrl;

 public RemoteAccountRepository(String serviceUrl) {
  this.serviceUrl = serviceUrl.startsWith("http") ? serviceUrl
    : "http://" + serviceUrl;
 }

 @Override
 public List<Account> getAllAccounts() {
  Account[] accounts = restTemplate.getForObject(serviceUrl+"/accounts", Account[].class);
  return Arrays.asList(accounts);
 }

 @Override
 public Account getAccount(String number) {
  return restTemplate.getForObject(serviceUrl + "/accounts/{id}",
    Account.class, number);
 }

}
```

Let’s open web application which is a consumer of the account microservice registered at Eureka Discovery Server.

http://localhost:8080/ as below

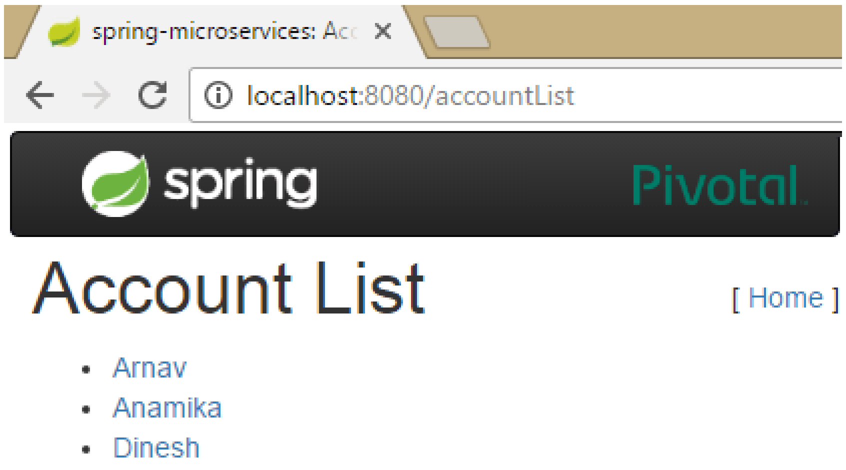


Accounts Web - Home Page

- View Account List

Now click on **View Account List** then fetch all accounts from account microservice.

http://localhost:8080/accountList

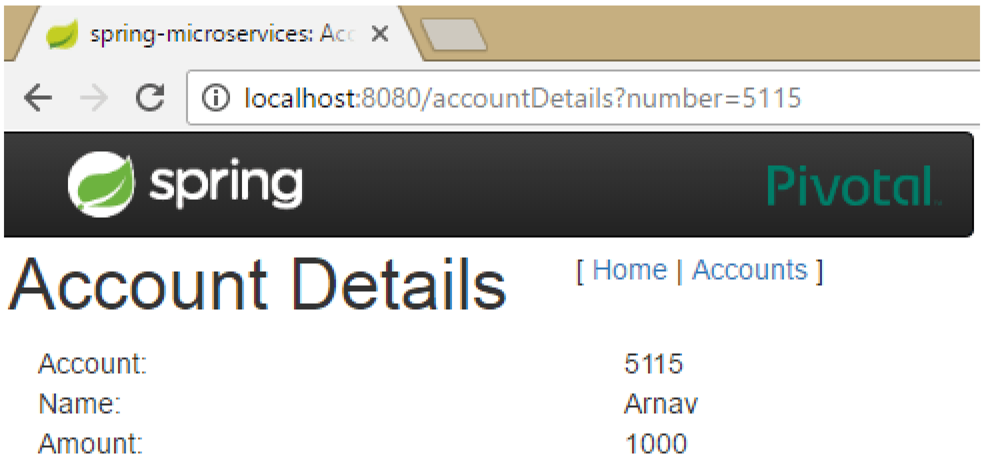


Account List

- Arnav
- Anamika
- Dinesh

Now click on any account from the list of accounts to fetch the details of the account for account number from account microservice.

http://localhost:8080/accountDetails?number=5115


spring-microservices: Acc
localhost:8080/accountDetails?number=5115
spring
Pivotal

Account Details [Home | Accounts]

Account:	5115
Name:	Arnav
Amount:	1000

Load Balanced *RestTemplate*

Create using *@LoadBalanced*– Spring enhances it to service lookup & load balancing

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

Must inject using the same qualifier-

- If there are multiple ***RestTemplate*** you get the right one.
- It can be used to access multiple microservices

```
@Autowired
       @LoadBalanced
 protected RestTemplate restTemplate;
```

Load Balancing with Ribbon

Our smart RestTemplate automatically integrates two Netflix utilities

- ***Eureka*** Service Discovery
- ***Ribbon*** Client Side Load Balancer

Eureka returns the URL of all available instances

Ribbon determine the best available service too use

Just inject the load balanced ***RestTemplate*** automatic lookup by ***logical service-name***

7. Summary

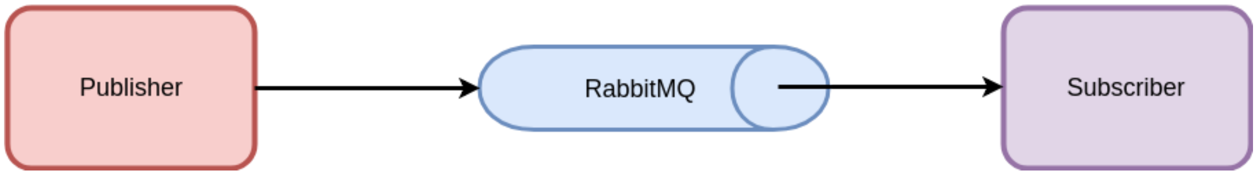
After completion of this article you should have learned:

- What is the MicroServices Architecture
- Advantages and Challenges of MicroServices
- And some information about Spring Cloud such as Eureka Discover Server by Netflix and Ribbon.

Microservice Communication using Spring Cloud Stream and RabbitMQ

Communication between microservices is the backbone of distributed systems. Usually, developers try to avoid it altogether fearing an increase in complexity. However, a combination of **Spring Cloud Stream and RabbitMQ** can make it *relatively easy to handle communication between microservices.*

Spring Cloud Stream is also part of the Spring Cloud group of projects. It provides easy integration with various message brokers with minimum configuration. Below is a **high-level view** on the overall pattern.



Spring Cloud Stream helps in exchanging messages between two applications or microservices. It can work seamlessly with message brokers such as RabbitMQ, Kafka and so on.

In this post, we will implement **communication between microservices** using **Spring Boot and Spring Cloud Stream**. Below is our high-level plan.

- **Step 1 –** Create a **RabbitMQ Server** using Docker
- **Step 2 –** Create a Spring Boot application using Spring Cloud Stream to **listen to messages**. We call this application as **subscriber-application**.
- **Step 3 –** Publish messages on RabbitMQ so that our **subscriber-application** can listen to those messages.
- **Step 4** – Create another Spring Boot application using Spring Cloud Stream to **publish messages** to Rabbit MQ. We call this application **publisher-application**.
- **Step 5** – Publish Messages using **publisher-application** that will be consumed by the **subscriber-application**.

So let’s start the process.

Step 1 – Creating RabbitMQ Server using Docker

For demonstrating **Spring Cloud Stream**, we will use RabbitMQ as the message broker. However, Spring Cloud Stream can easily work with other message brokers as well such as Kafka or Amazon Kinesis. So you can use those well.

For simplicity, however, we will go with **RabbitMQ**.

RabbitMQ can be easily setup on your machine. One approach is to follow the official download guide and get RabbitMQ for your operating system of choice based on the guidelines.

However, another easy way to get RabbitMQ is through Docker. If you are not familiar with Docker, you can follow my detailed post on understanding the basics of Docker. The official Docker image for RabbitMQ is available at this link.

RabbitMQ server can be easily started with the below command.

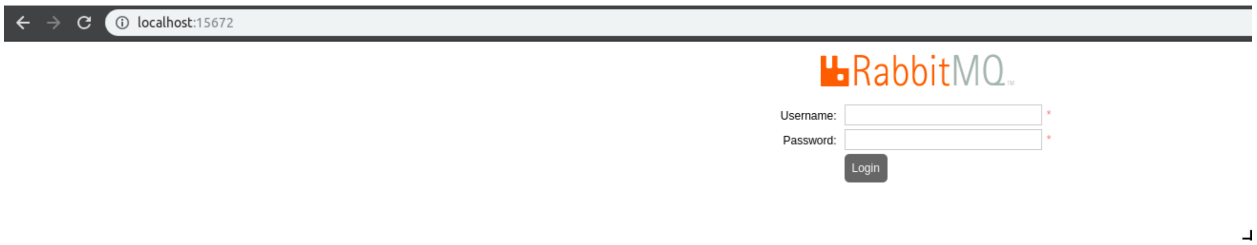
```
docker run -d --hostname my-test-rabbit --name test-rabbit -p 15672:15672 -p
5672:5672 rabbitmq:3-management
```

This command pulls in the **RabbitMQ** image from Docker hub and starts the server on port 5672. We also expose the **RabbitMQ management console** on port 15672.

RabbitMQ stores data based on node name that defaults to the **hostname**. In this case, we provide the **hostname** as *my-test-rabbit*. We could also check the logs for RabbitMQ server using the below command.

```
docker logs test-rabbit
```

However, we have also exposed a management console for our **RabbitMQ** server. Basically, it provides a **graphical user interface** to manage our RabbitMQ. You can access it at *http://localhost:15672*.

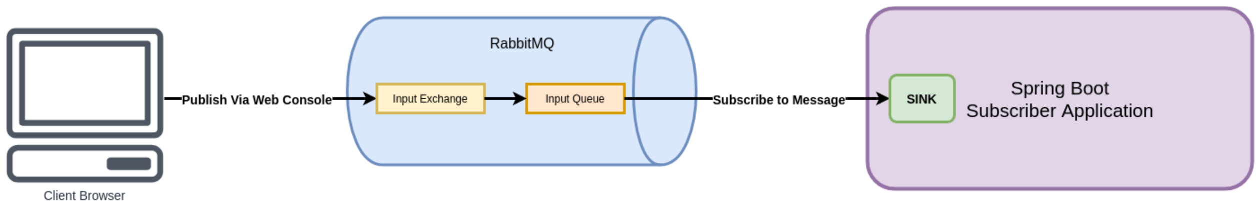


If you see the above login page, the **RabbitMQ** setup on your machine has been successful. Now we can move to Step 2.

Step 2 – Create the Subscriber Application

In this step, we can create an application using **Spring Boot and Spring Cloud Stream** to listen to messages. If you are not aware of Spring Boot, I have a detailed guide on Spring Boot Microservices.

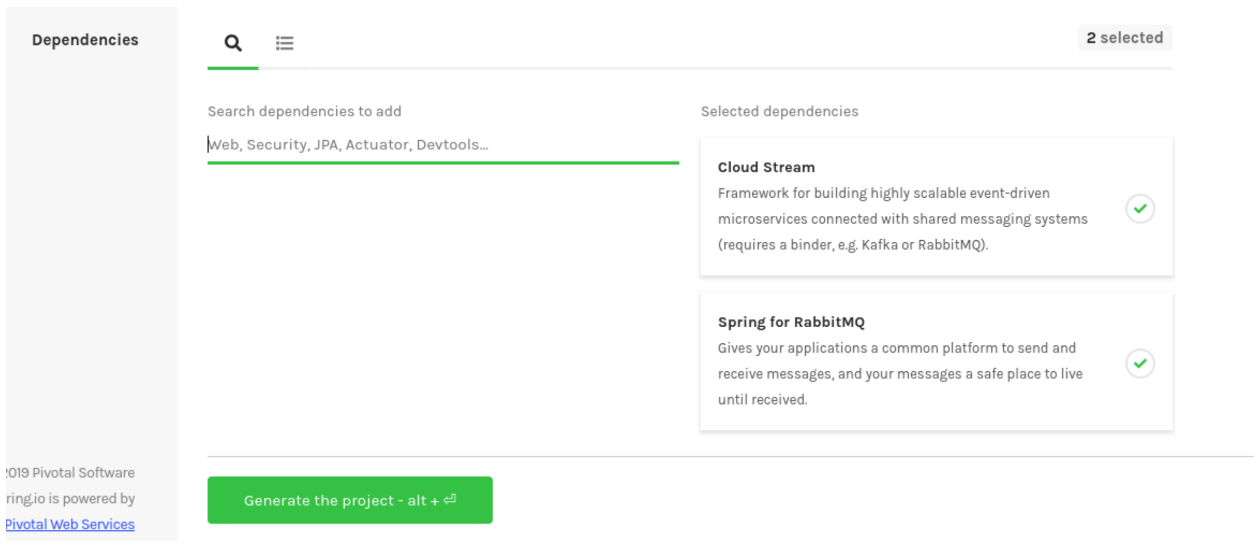
Below is our basic plan at this point.



Basically, we will be first building our **subscriber-application** that connects to the RabbitMQ server. We will publish the messages directly through the **RabbitMQ web console**. And our Spring Boot application will have something known as **Sink** that will help us process the incoming messages. We will look at what a **Sink** is in the next section.

Selecting the Dependencies

We can quickly bootstrap an application using https://start.spring.io. Important thing to note is the below dependencies for **Spring Cloud Stream and RabbitMQ**. These two dependencies together form the basis of our integration. At this point, you can also select Kafka or some other option (instead of RabbitMQ) depending on your choice.



Cloud Stream
Framework for building highly scalable event-driven microservices connected with shared messaging systems (requires a binder, e.g. Kafka or RabbitMQ).

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

You can generate the project after which you can look at the *POM.xml* file (in your IDE of choice) where these dependencies are now mentioned.

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

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>
		<dependency>
			<groupId>org.springframework.cloud</groupId>
			<artifactId>spring-cloud-stream-test-support</artifactId>
			<scope>test</scope>
```

```
        </dependency>
</dependencies>
```

Adding Message Handler

Even at this point, our Spring Boot application can be run. However, it does not have any functionality.

As per our plan, we want this application to listen to messages published on the RabbitMQ. To do so, we need to add a listener that can handle incoming messages.

Below is how we will do it.

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class SubscriberApplication {

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

    @StreamListener(Sink.INPUT)
    public void handleMessage(Message message){
        System.out.println("Received Message is: " + message);
    }

    public static class Message{
        private String message;

        public String getMessage() {
            return message;
        }

        public void setMessage(String message) {
            this.message = message;
        }

        @Override
        public String toString() {
```

```
                    return "Message{" +
                            "message='" + message + '\'' +
                            '}';
            }
        }

}
```

Important things to note here are as follows:

- We have enabled **Sink** binding by using *@EnableBinding* annotation. This step signals the underlying framework to create the necessary bindings to the messaging middleware. In other words, it will create the destination items such as queue, topic etc.
- Also, we have added a **handler** method. This method is used to receiving incoming messages of type **Message**. This is one of the most powerful features of **Spring Cloud Stream**. The framework tries to automatically convert incoming messages to the type **Message**.

With this we are basically done with the minimal setup required in our application.

Step 3 – Publish Message through RabbitMQ

We are ready to test our subscriber application and see it in action.

You can simply start the application using the below command:

```
clean package spring-boot:run
```

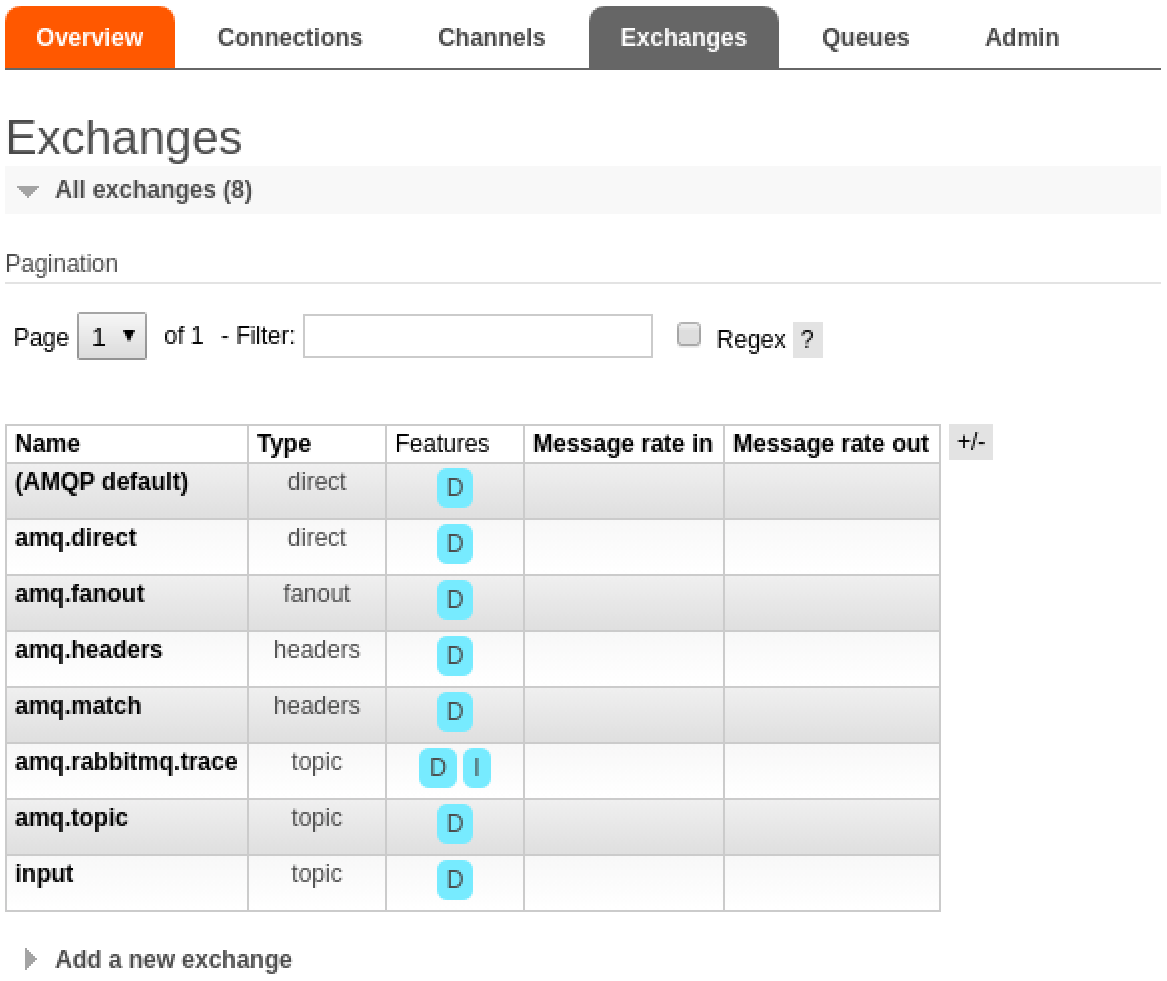
The application will automatically try to connect to a RabbitMQ server at *http://localhost:5672*. You should see something like this in the application startup logs.

```
2019-06-25 17:35:22.304  INFO 8115 --- [           main]
o.s.a.r.c.CachingConnectionFactory       : Created new connection:
rabbitConnectionFactory#34688e58:0/SimpleConnection@1a981cf0
[delegate=amqp://guest@127.0.0.1:5672/, localPort= 52000]
2019-06-25 17:35:22.355  INFO 8115 --- [           main]
o.s.i.monitor.IntegrationMBeanExporter   : Registering MessageChannel
input.anonymous.KWrQWmqZSYKPjOBkx4DotA.errors
2019-06-25 17:35:22.432  INFO 8115 --- [           main]
o.s.c.stream.binder.BinderErrorChannel   : Channel
'application.input.anonymous.KWrQWmqZSYKPjOBkx4DotA.errors' has 1 subscriber(s).
2019-06-25 17:35:22.432  INFO 8115 --- [           main]
o.s.c.stream.binder.BinderErrorChannel   : Channel
'application.input.anonymous.KWrQWmqZSYKPjOBkx4DotA.errors' has 2 subscriber(s).
2019-06-25 17:35:22.454  INFO 8115 --- [           main]
o.s.i.a.i.AmqpInboundChannelAdapter      : started
inbound.input.anonymous.KWrQWmqZSYKPjOBkx4DotA
2019-06-25 17:35:22.464  INFO 8115 --- [           main]
c.p.d.s.SubscriberApplication            : Started SubscriberApplication in 3.52
seconds (JVM running for 18.305)
```

To test whether our application is able to listen to messages, we can publish a message through the **RabbitMQ management console**.

To do so, you can login to the console at *http://localhost:15672* using default *userid/password* as *guest/guest*.

At this point, you should see the list of exchanges as below. Note the last exchange in the list known as *input*. This is basically created automatically when we started our application.



Exchanges

All exchanges (8)

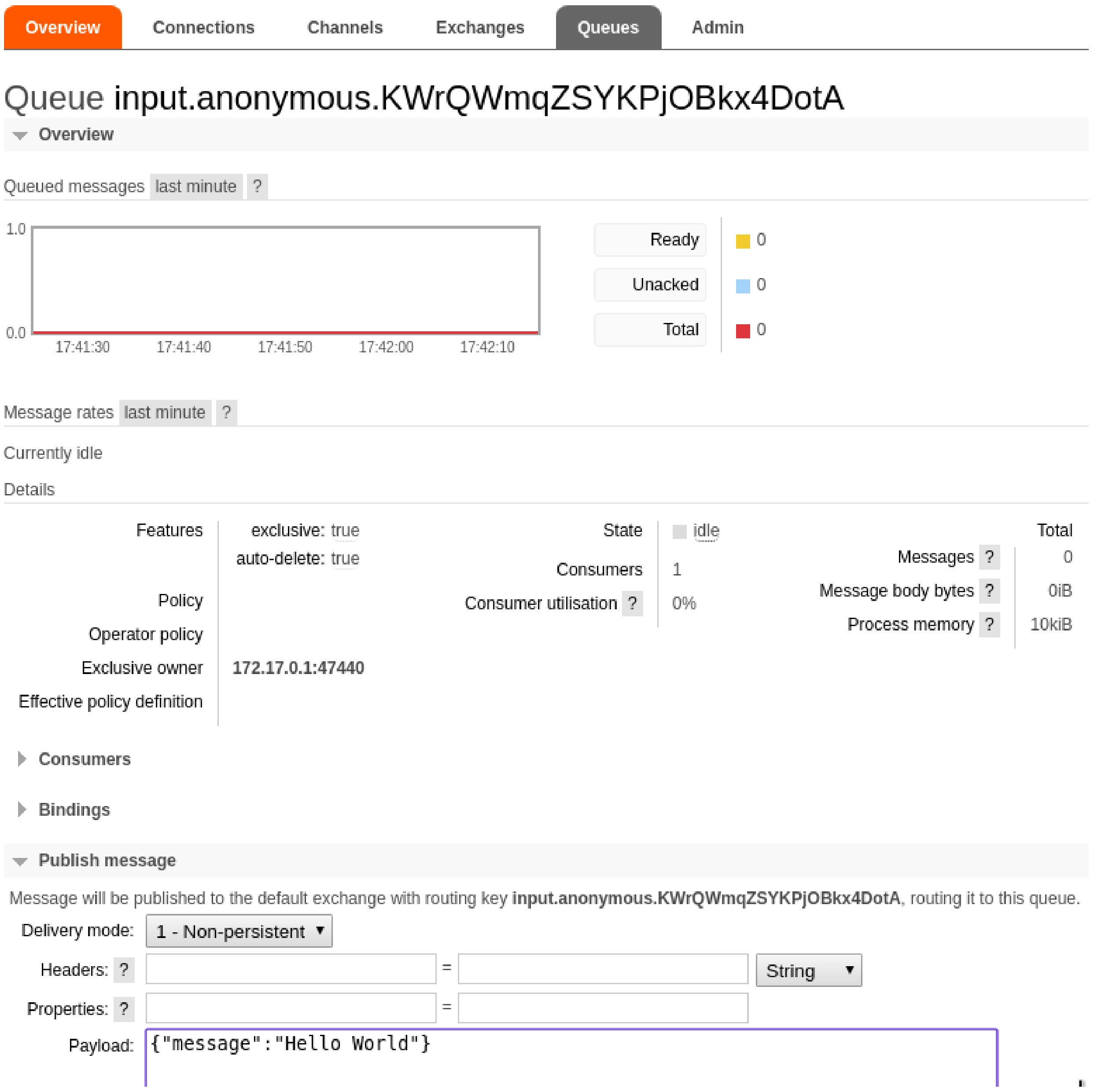
Pagination

Page 1 of 1 - Filter: Regex ?

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		
input	topic	D		

Add a new exchange

Also, a queue under the *input* exchange is created where we can publish messages.



Queue input.anonymous.KWrQWmqZSYKPjOBkx4DotA

Overview

Queued messages last minute ?

Ready	🟨 0
Unacked	🟦 0
Total	🟥 0

Message rates last minute ?

Currently idle

Details

Features	exclusive: true auto-delete: true
Policy	
Operator policy	
Exclusive owner	172.17.0.1:47440
Effective policy definition	

State	idle
Consumers	1
Consumer utilisation ?	0%

	Total
Messages ?	0
Message body bytes ?	0iB
Process memory ?	10kiB

Consumers

Bindings

Publish message

Message will be published to the default exchange with routing key **input.anonymous.KWrQWmqZSYKPjOBkx4DotA**, routing it to this queue.

Delivery mode: 1 - Non-persistent

Headers: ? = String

Properties: ? =

Payload: {"message":"Hello World"}

Once you click **Publish**, you will see the message printed in the *subscriber-application* logs as below.

```
2019-06-25 17:35:22.454  INFO 8115 --- [           main]
o.s.i.a.i.AmqpInboundChannelAdapter      : started
inbound.input.anonymous.KWrQWmqZSYKPjOBkx4DotA
2019-06-25 17:35:22.464  INFO 8115 --- [           main]
c.p.d.s.SubscriberApplication            : Started SubscriberApplication in 3.52
seconds (JVM running for 18.305)
```

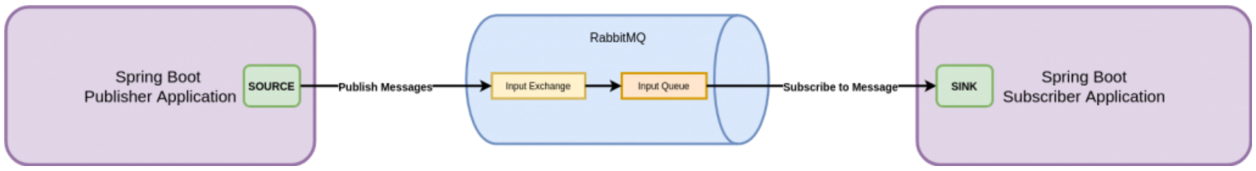
```
Received Message is: Message{message='Hello World'}
```

With this, we are done with the subscriber part of our application. Now we can move onto the next step.

Step 4 – Create the Publisher Application

In this step, we will create a publisher application that has an end-point to publish messages to the **RabbitMQ** server.

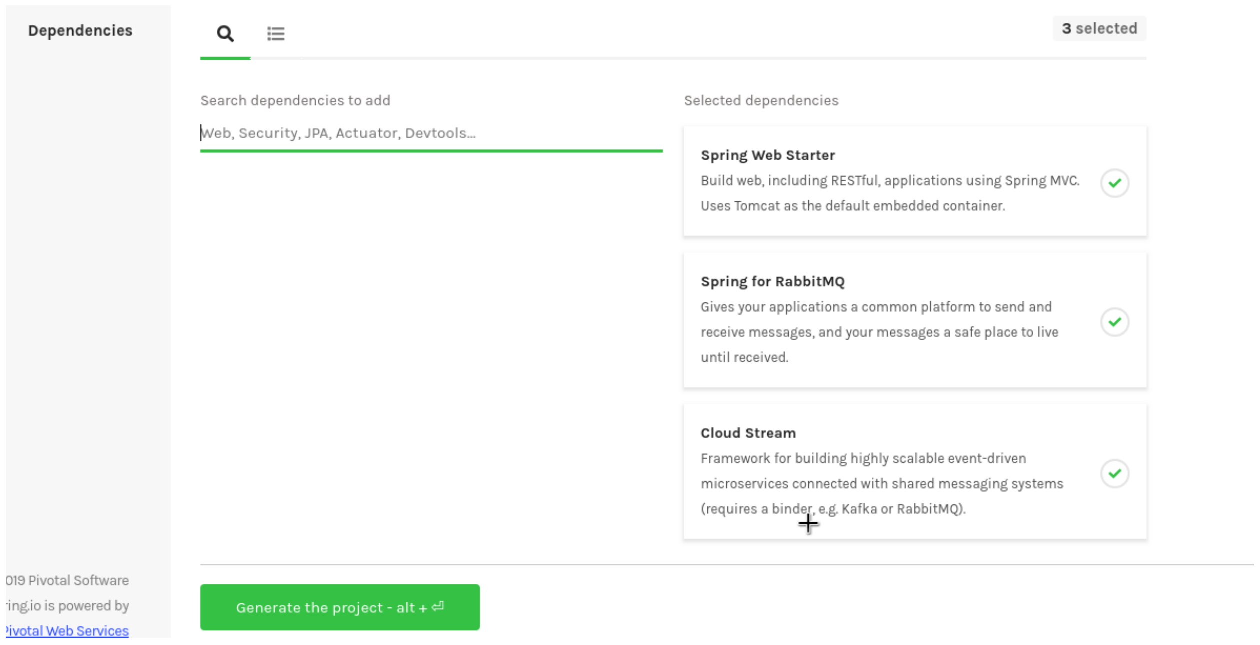
Below is our high-level plan for the same.



As you can see, now we have a **publisher-application** instead of the web console. In the publisher application, we have something known as **Source**. Think of it as an opposite of the **Sink** interface we saw earlier in the **subscriber-application**. We will look at the **Source** interface in detail in this section.

Selecting the Dependencies

To quickly bootstrap the application, we can use the **Spring Initializr** like we did for the subscriber-application. This time we will include an extra dependency known as **Spring Web Starter**.



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

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

Cloud Stream
Framework for building highly scalable event-driven microservices connected with shared messaging systems (requires a binder, e.g. Kafka or RabbitMQ).

You can click **Generate Project** to get the zip file on your local machine.

Adding Message Publishing Logic

To publish message using **Spring Cloud Stream and RabbitMQ**, we modify our publisher-application's main class as below.

```java
@SpringBootApplication
public class PublisherApplication {

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

@RestController
@EnableBinding(Source.class)
class MessagePublisher{

	@Autowired
	private Source source;

	@GetMapping(value = "/api/publish")
	public void sendMessage(){
		Message message = new Message("Hello World from Publisher");

		source.output().send(MessageBuilder.withPayload(message).build());

	}
}

class Message{
	String message;

	public Message(String message) {
		this.message = message;
	}

	public String getMessage() {
		return message;
	}

	public void setMessage(String message) {
		this.message = message;
	}
}
```

The important things to note here are:

- We create a **MessagePublisher** class and annotate it as *@RestController*.
- Also, we annotate the controller class with **@EnableBinding**. However, instead of binding to **Sink** (as we did in the subscriber), we bind this class with the **Source.class**. Basically,

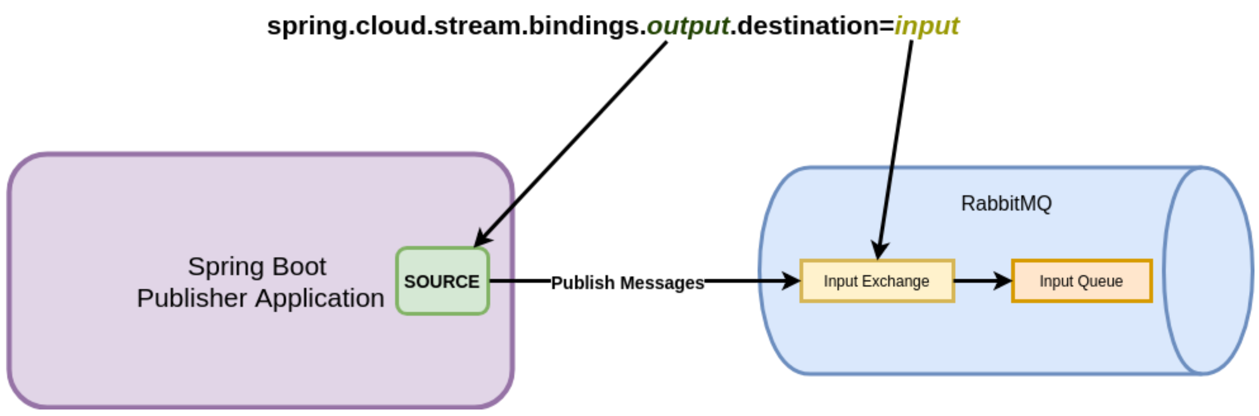
Source and Sink are *binding interfaces* provided by Spring Cloud Stream.

- We auto-wire an instance of the **Source** class and in the */api/publish* call we use it to publish a Message object to RabbitMQ.
- We also define the **Message** class to create a new message.

There is another important setting you need to do in the *application.properties* file. We need to define the output binding as below.

```
spring.cloud.stream.bindings.output.destination=input
spring.cloud.stream.default.contentType=application/json
```

The below illustration can help understand what we mean by this property.


spring.cloud.stream.bindings.output.destination=input
Spring Boot Publisher Application
SOURCE
Publish Messages
RabbitMQ
Input Exchange
Input Queue

Basically, the **Source** interface exposes an **Output** channel that we are binding to the ***input*** exchange. This relation is required for the publisher and the subscriber to be able to exchange messages.

Step 5 – Publish Message using publisher-application

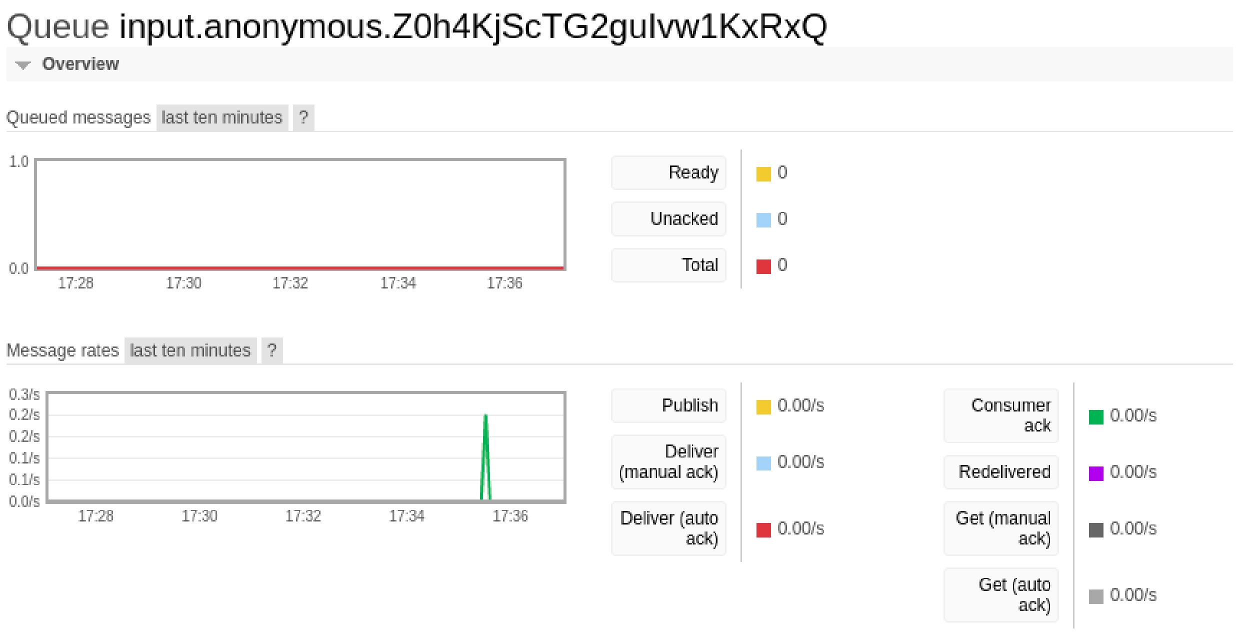
Now we can start-up the *publisher-application* as well. Once the application starts up, we need to trigger the endpoint *http://localhost:8080/api/publish*.

Nothing will be shown in the browser or the client as response. However, if you visit the logs of the **subscriber-application**, we will be able to see the message printed.

```
2019-06-26 17:18:50.366  INFO 5138 --- [           main]
o.s.i.a.i.AmqpInboundChannelAdapter      : started
inbound.input.anonymous.Z0h4KjScTG2guIvw1KxRxQ
2019-06-26 17:18:50.374  INFO 5138 --- [           main]
c.p.d.s.SubscriberApplication            : Started SubscriberApplication in 2.939
seconds (JVM running for 11.335)
```

```
Received Message is: Message{message='Hello World from Publisher'}
```

Also, you will be able to see a green spike in the **RabbitMQ console** in the input queue. This shows the transmission of the message to the input queue to which the **subscriber-application** is listening.


Queue input.anonymous.Z0h4KjScTG2guIvw1KxRxQ

Overview

Queued messages last ten minutes ?

Ready	🟨 0
Unacked	🟦 0
Total	🟥 0

Message rates last ten minutes ?

Publish	🟨 0.00/s	Consumer ack	🟩 0.00/s
Deliver (manual ack)	🟦 0.00/s	Redelivered	🟪 0.00/s
Deliver (auto ack)	🟥 0.00/s	Get (manual ack)	⬛ 0.00/s
		Get (auto ack)	⬜ 0.00/s

Conclusion

With this, we have successfully developed a small application using **Spring Cloud Stream and RabbitMQ to publish and subscribe to messages**.