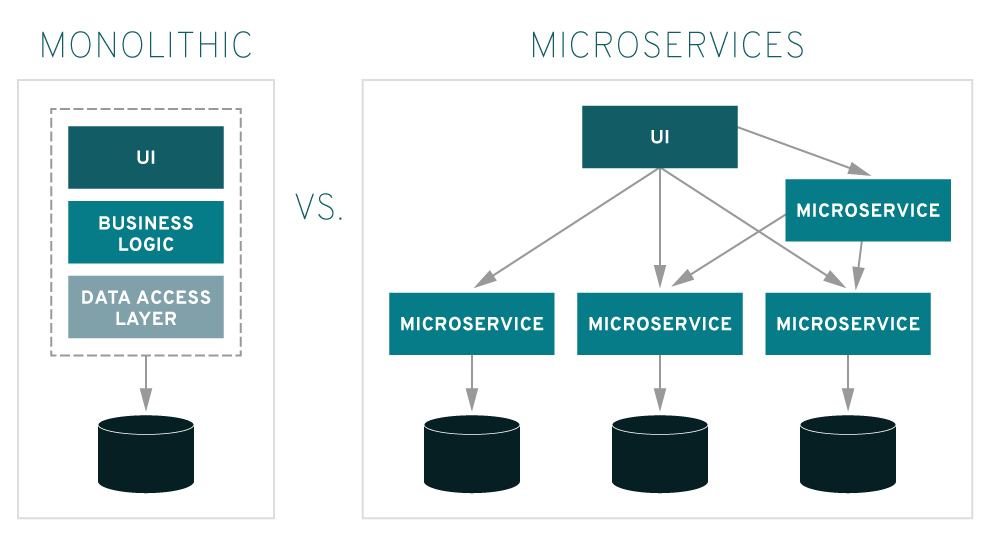
|  |  |
| --- | --- |
|  | 1. **Overview ( Microservices )** 2. **Advantages of Microservices Architechtures** 3. **Challenges with Microservices Architechtures** 4. **Best Languages for Microservices** 5. **Java Based Microservices** 6. **Microservices Design Pattern** 7. **Microservices ( Spring boot and spring cloud Architechture )**   C:\Users\anil\AppData\Local\Microsoft\Windows\INetCache\Content.Word\spring-boot1.png   1. **Setup Microservices -synchronous ( spring Cloud )** 2. **JMS ( kafka & kafka-stream)** 3. **Setup Microservices - Asynchronous ( Reactive Programming )** 4. **Reference**     **Documented by: @Anil Gupta** |

1. **Overview** ( Microservices **)**

## Microservices are an [architectural approach to building applications](https://www.redhat.com/en/topics/cloud-native-apps/what-is-an-application-architecture). As an architectural framework, microservices are distributed and loosely coupled, so one team’s changes won’t break the entire app. The benefit to using microservices is that development teams are able to rapidly build new components of apps to meet changing business needs.

A way to build apps, optimized for DevOps and CI/CD

What sets a microservices architecture apart from more traditional, monolithic approaches is how it breaks an app down into its core functions. Each function is called a service, and can be built and deployed independently, meaning individual services can function (and fail) without negatively affecting the others. This helps you to embrace the technology side of DevOps and make [constant iteration and delivery (CI/CD)](https://www.redhat.com/en/topics/devops/what-is-ci-cd) more seamless and achievable.



## Advantages of a microservices architecture

Microservices give your teams and routines a boost through distributed development. You can also develop multiple microservices concurrently. This means more developers working on the same app, at the same time, which results in less time spent in development.

### 1.Ready for market faster

Since development cycles are shortened, a microservices architecture supports more agile deployment and updates.

### 2.Highly scalable

As demand for certain services grows, you can deploy across multiple servers, and infrastructures, to meet your needs.

### 3. Resilient

These independent services, when constructed properly, do not impact one another. This means that if one piece fails, the whole app doesn’t go down, unlike the monolithic app model.

### 4. Easy to deploy

Because your microservice-based apps are more modular and smaller than traditional, monolithic apps, the worries that came with those deployments are negated. This requires more coordination, which a [service mesh layer](https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh) can help with, but the payoffs can be huge.

### 5. Accessible

Because the larger app is broken down into smaller pieces, developers can more easily understand, update, and enhance those pieces, resulting in faster development cycles, especially when combined with [agile development methodologies](https://www.redhat.com/en/topics/devops/what-is-agile-methodology).

### 6. More open

Due to the use of polyglot APIs, developers have the freedom to choose the best language and technology for the necessary function.

1. **Challenges with Microservice Architechtures**

If your organization is thinking about shifting to a microservices architecture, expect to change the way people work, not just the apps. Organizational and cultural changes are [identified as challenges](https://www.redhat.com/en/blog/state-microservices) in part because each team will have its own deployment cadence and will be responsible for a unique service with its own set of customers. Those may not be typical developer concerns, but they will be essential to a successful microservices architecture.

Beyond culture and process, complexity and efficiency are two major challenges of a microservice-based architecture. John Frizelle, platform architect for Red Hat Mobile, laid out these eight challenge categories in his [2017 talk at Red Hat Summit](https://rh2017.smarteventscloud.com/connect/sessionDetail.ww?SESSION_ID=104609&tclass=popup):

1. **Building:** You have to spend time identifying dependencies between your services. Be aware that completing one build might trigger several other builds, due to those dependencies. You also need to consider the effects that microservices [have on your data](https://developers.redhat.com/blog/2016/08/02/the-hardest-part-about-microservices-your-data/).
2. **Testing:** [Integration](https://www.redhat.com/en/solutions/agile-integration) testing, as well as end-to-end testing, can become more difficult, and more important than ever. Know that a failure in one part of the architecture could cause something a few hops away to fail, depending on how you’ve architected your services to support one another.
3. **Versioning:** When you update to new versions, keep in mind that you might break backward compatibility. You can build in conditional logic to handle this, but that gets unwieldy and nasty, fast. Alternatively, you could stand up multiple live versions for different clients, but that can be more complex in maintenance and management.
4. **Deployment:** Yes, this is also a challenge, at least in initial set up. To make deployment easier, you must first invest in quite a lot of [automation](https://www.redhat.com/en/topics/automation) as the complexity of microservices becomes overwhelming for human deployment. Think about how you’re going to roll services out and in what order.
5. **Logging:** With distributed systems, you need centralized logs to bring everything together. Otherwise, the scale is impossible to manage.
6. **Monitoring:** It’s critical to have a centralized view of the system to pinpoint sources of problems.
7. **Debugging:** Remote debugging through your local [integrated development environment (IDE)](https://www.redhat.com/en/topics/middleware/what-is-ide) isn’t an option and it won’t work across dozens or hundreds of services. Unfortunately there’s no single answer to how to debug at this time.
8. **Connectivity:** Consider service discovery, whether centralized or integrated.

## The future of microservices leads us closer to serverless architecture; particularly, the promise of cost savings by only paying the amount of compute utilized is even more appealing.  Microservices is an extensive concept that applies to churn apps, products or solutions to more granular and modular level. Keep in mind, it is not recommended to start microservices architecture from scratch since it is difficult to define the boundaries of each service at the beginning. There is no better way to choose the perfect technology for your microservices. Every technology decision depends on the tools you will use to develop other parts of your application. It also depends on the current knowledge of your development team.

## Best Languages for Microservices

Microservices can be implemented with a horde of frameworks, versions, and tools. Java, Python, C++, Node JS, and .Net are few of them. Let us explore the languages that support microservices development in detail:

### 1. Java

Annotation syntax, which is easy to read, is the key factor that makes Java a great programming language for developing microservices. This feature makes Java Microservices much easier to develop when powered by Microservices frameworks. It offers more value in readability, particularly while working with complex systems. Java includes many opinions to support developing & deploying Java Microservices. It offers a user Interface, model components as well as connectivity to back-end resources, everything within the boundaries of a single, isolated and independently deployed apps.

In addition, many of Java EE standards are well suited for microservices applications like:

* JAX-RS for APIs
* JPA for data handling
* CDI for dependency injection & lifecycle management

In addition, service discovery solutions like Consul, Netflix Eureka or Amalgam8 are effortless in connecting with Java Microservices.

There are several Frameworks for developing Microservices architecture. Some of the Java Microservices Frameworks are as follows:

* **Spring Boot** – This framework works on top of various languages for Aspect-Oriented programming, Inversion of Control and others
* **Dropwizard** – This Java microservices framework assembles stable and mature libraries of Java into a simple and light-weight package
* **Restlet** – It supports developers to build better web APIs, which trail the REST architecture model
* **Spark** – One of the best Java Microservices frameworks, supports creating web apps in Java 8 and Kotlin with less effort

### 2. Golang

If you want to enhance your existing project, the Golang can be a good choice for microservices development. Golang, also known as Go is popular for its concurrency and API support in terms of microservices architecture. With the Golang’s concurrency possibility, you can expect increased productivity of various machines and cores. It includes a powerful standard for developing web services. It is exclusively designed for creating large and complex applications. Go provides two impressive frameworks for microservices development:

* **GoMicro** – It is an RPC framework, which comes with the advantages like Load balancing, server packages, PRC client, and message encoding.
* **Go Kit** – The key difference of Go Kit from GoMirco is it needs to be imported into a binary package. Moreover, it is advanced for explicit dependencies, Domain-driven design, and declarative aspect compositions.

In addition to simple syntax, Go microservices architecture includes excellent testing support as it makes it simple to write robust tests as well as embed them flawlessly into workflows.

### 3. Python

Python is a high-level programming language that offers active support for integration with various technologies. Prototyping in Python is faster and easier when compared to other frameworks and languages. It includes powerful substitutes for heavy implementations like Django. Microservices Python ensures compatibility with legacy languages like ASP and PHP, which allows you to create web service front-ends to host Microservices.

With all these benefits, Microservices Python is considered to have an edge over other languages. Developers who implement Microservices Python use a RESTful API approach - a comprehensive way of utilizing web protocols & software to remotely manipulate objects. With this technology, it becomes easier to monitor the application since it is now broken into components. There is a broad range of Python microservices frameworks to choose from for your web application development. Some of them are as follows:

* **Flask** – Most popular Python Micro framework based on Jinja2 and Werkzeug
* **Falcom** – Create smart proxies, cloud APIs and app back-ends
* **Bottle** – Simple, lightweight and fast WSGI micro framework
* **Nameko**– Best among the Python Microservices frameworks that allows developers to concentrate on application logic
* **CherryPy** – Mature, Python object-oriented web framework

### 4. Node JS

Node JS became the go-to platform in the past few years for enterprises and startups who want to embrace microservices. Node JS is built with the V8 runtime; hence, microservices Node JS is going to be super-fast for Input-Output (IO) – bound tasks. Normally, Microservices Node JS is developed either using CPU-bound or IO-bound code. CPU-bound program demands many intensive calculations. Every time you run an IO call, Node JS doesn’t block the main-thread but submits the tasks to be executed by the internal IO daemon threads. Hence, Microservices Node JS gains popularity in terms of IO-bound tasks.

### 5. .Net

ASP.Net, the .Net framework for web development makes it simple to build the APIs that becomes the microservices. It includes built-in support for building and deploying microservices using Docker containers. .Net comes with APIs that can simply consume microservices from any application you developed including desktop, mobile, web, gaming and more. If you have an application, you can start adopting .Net microservices without entirely revamping that application. The initial setup for .Net Docker images has already been done and available on Docker Hub, helping you to concentrate only on building your microservices.

1. **JAVA Based Microservices**

**Basic Tools requirements**

|  |  |  |
| --- | --- | --- |
|  | purpose | Tools |
| 1 | Api management and testing | [ **Postman** ],Api Fortress , Tyk |
| 2 | Messaging (JMS) | [ **ApacheKafka] ,**RabbitMQ , Amazon Simple Queue Service (SQS) ,Google Cloud Pub/Sub |
| 3 | Monitoring | [ **Zipkins ],**Logstach , graylog , |
| 4 | Containerization | **[ Docker** ] ,CRI-O,RKTlet , Microsoft-container |
| 5 | orchestration | **[ Docker-swarn], [kubernetes ]**, Telepresence,istio,minicube |
| 6 | Toolkits | Fabric 8,sceneka (for node js) |
| 7 | Architectural Frameworks | **[ java-Spring-boot ]** ,Goa, kong |
| 8 | Serverless tools | **[ AWS Lamda ]** ,cloudia.js |

1. **Microservice design Pattern [** Asynchronous (**Proactor** ) and Synchronous (Reactor ) I/O pattern ]

## Comparing Two High-Performance I/O Design Patterns( based on TCP server)

System I/O can be blocking (synchronous) , or non-blocking (asynchronous). Blocking I/O means that the calling system does not return control to the caller until the operation is finished. As a result, the caller is blocked and cannot perform other activities during that time. Most important, the caller thread cannot be reused for other request processing while waiting for the I/O to complete, and becomes a wasted resource during that time

By contrast, a non-blocking synchronous call returns control to the caller immediately. The caller is not made to wait, and the invoked system immediately returns one of two responses: If the call was executed and the results are ready, then the caller is told of that. Alternatively, the invoked system can tell the caller that the system has no resources (no data in the socket) to perform the requested action. In that case, it is the responsibility of the caller may repeat the call until it succeeds.

In a non-blocking asynchronous call, the calling function returns control to the caller immediately, reporting that the requested action was started. The calling system will execute the caller's request using additional system resources/threads and will notify the caller (by callback for example), when the result is ready for processing.

Two patterns that involve event demultiplexors are called **Reactor** and **Proactor** [[1](https://www.artima.com/articles/io_design_patterns3.html#resources)]. The Reactor patterns involve synchronous I/O, whereas the Proactor pattern involves asynchronous I/O. In Reactor, the event demultiplexor waits for events that indicate when a file descriptor or socket is ready for a read or write operation. The demultiplexor passes this event to the appropriate handler, which is responsible for performing the actual read or write.

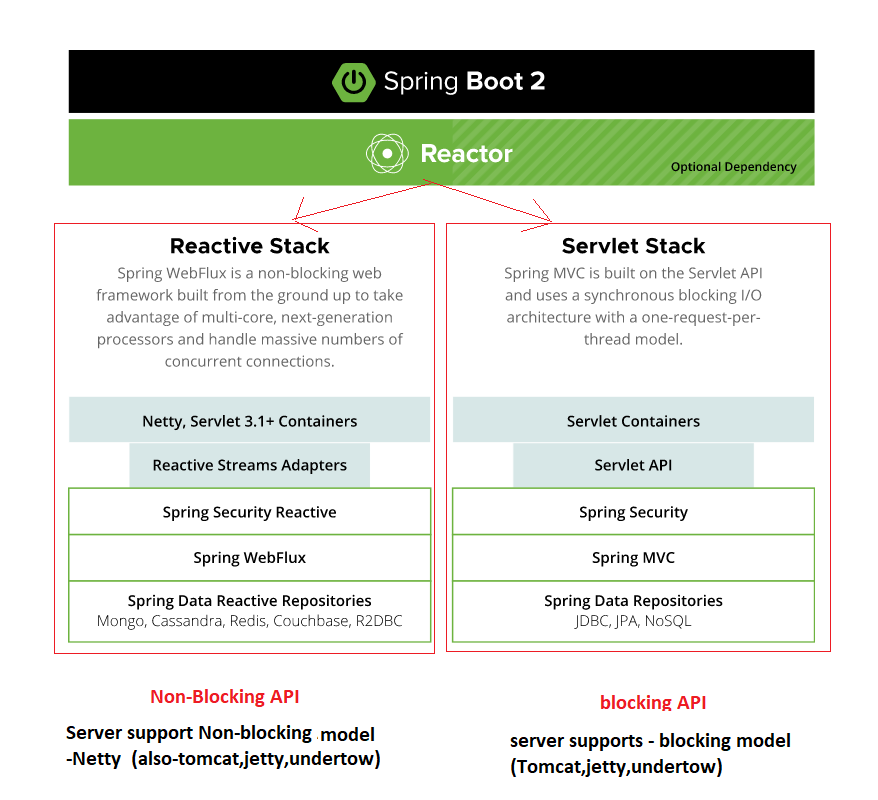
In the Proactor pattern, by contrast, the handler—or the event demultiplexor on behalf of the handler—initiates asynchronous read and write operations. The I/O operation itself is performed by the operating system (OS). The parameters passed to the OS include the addresses of user-defined data buffers from which the OS gets data to write, or to which the OS puts data read. The event demultiplexor waits for events that indicate the completion of the I/O operation, and forwards those events to the appropriate handlers.

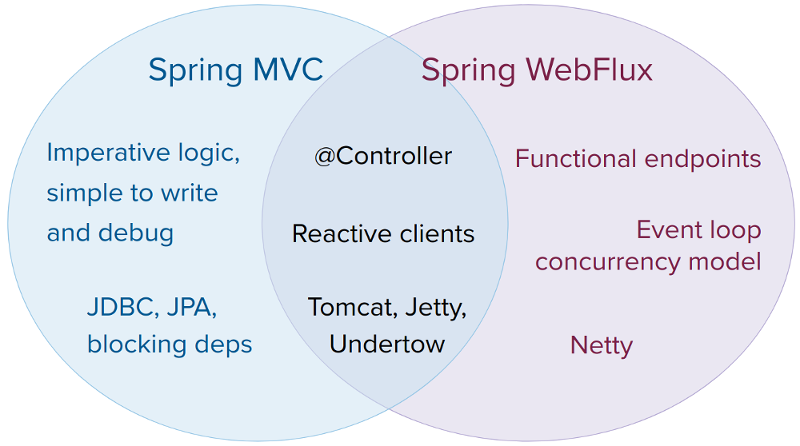
## Reactive Programming in Java (non-blocking I/O Libraries)

1. [**Reactive Streams**](http://www.reactive-streams.org/) (Reactive Streams have been incorporated into the JDK as java.util.concurrent.Flow in version 9)
2. [**RxJava**](https://github.com/ReactiveX/RxJava/wiki): (ReactiveX/RxJava)- 2nd Generation Reactive libraries
3. [**Akka**](http://akka.io/) : **3rd** Generation Reactive libraries (based on Akka Streams, and Reactive Streams)
4. **Reactor : 4th** Generation Reactive lib.
5. **Spring Framwork 5.0 : Webflux -** based onReactor or RxJava libraries
6. **Spring Boot (ratpack) :Ratpack** - A toolkit for web applications on the JVM.[ It is a set of Java libraries for building scalable HTTP applications It is a lean and powerful foundation. Its apps are lightweight, fast, composable with other tools and libraries, easy to test and enjoyable to develop.]

|  |  |  |
| --- | --- | --- |
| **RxJava 2** | **Reactor** | **Purpose** |
| [Completable](http://reactivex.io/RxJava/javadoc/io/reactivex/Completable.html) | N/A | Completes successfully or with failure, without emitting any value. Like CompletableFuture<Void> |
| [Maybe<T>](http://reactivex.io/RxJava/javadoc/io/reactivex/Maybe.html) | [Mono<T>](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html) | Completes successfully or with failure, may or may not emit a single value. Like an asynchronous Optional<T> |
| [Single<T>](http://reactivex.io/RxJava/javadoc/io/reactivex/Single.html) | N/A | Either complete successfully emitting exactly one item or fails. |
| [Observable<T>](http://reactivex.io/RxJava/javadoc/io/reactivex/Observable.html) | N/A | Emits an indefinite number of events (zero to infinite), optionally completes successfully or with failure. Does not support backpressure due to the nature of the source of events it represents. |
| [Flowable<T>](http://reactivex.io/RxJava/javadoc/io/reactivex/Flowable.html) | [Flux<T>](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) | Emits an indefinite number of events (zero to infinite), optionally completes successfully or with failure. Support backpressure (the source can be slowed down when the consumer cannot keep up) |
|  |  |  |

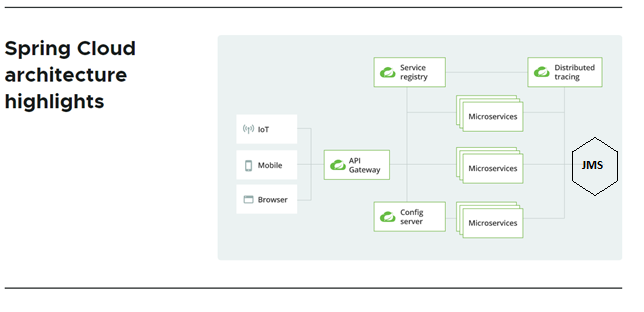
1. **Spring-boot (microservices with spring cloud libraries )**

****

****

**Spring Cloud**

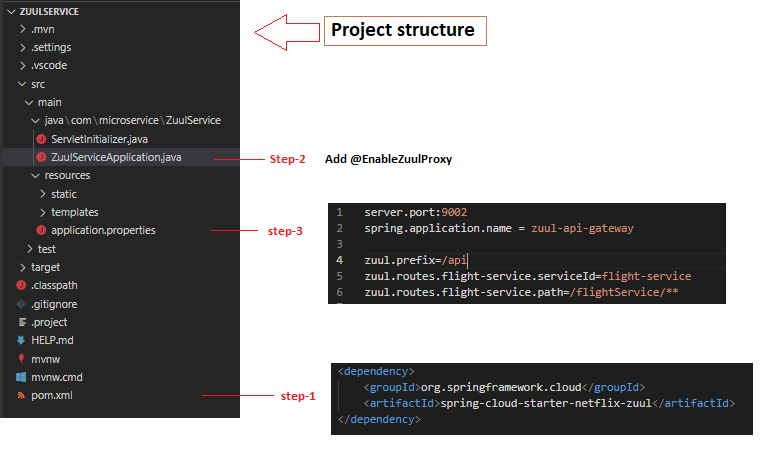
Developing distributed systems can be challenging. Complexity is moved from the application layer to the network layer and demands greater interaction between services. Making your code ‘cloud-native’ means dealing with 12-factor issues such as external configuration, statelessness, logging, and connecting to backing services. The Spring Cloud suite of projects contains many of the services you need to make your applications run in the cloud.



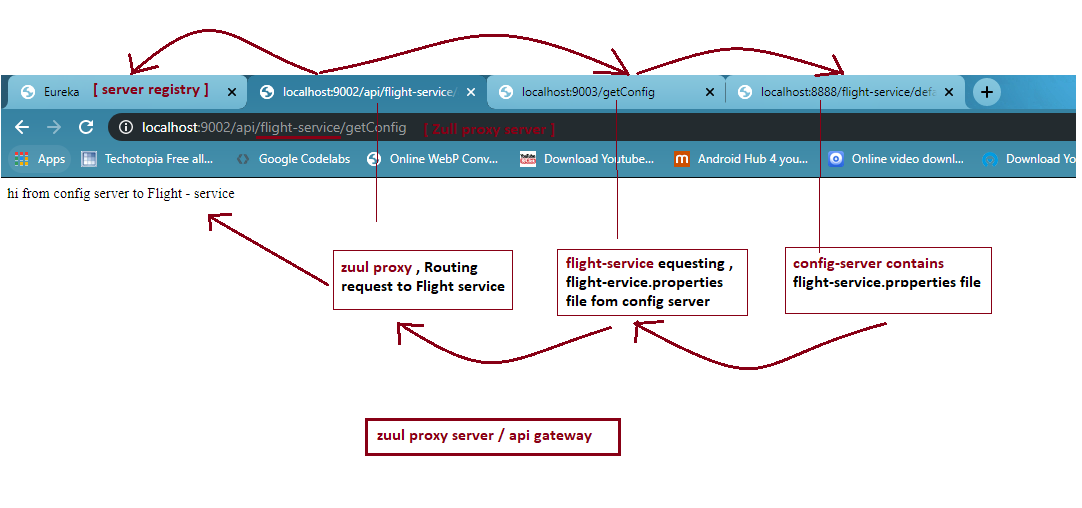
|  |  |
| --- | --- |
| Architecture components | **Description** |
| API gateway[ ZUUL PROXY ] | A gateway can take care of securing and routing messages, hiding services, throttling load, and many other useful things. [Spring Cloud Gateway](https://spring.io/projects/spring-cloud-gateway) gives you precise control of your API layer, integrating **Spring-Cloud-service-discovery** and client-side load-balancing solutions to simplify configuration and maintenance. |
| Service- discovery[ [Netflix Eureka](https://github.com/Netflix/eureka) Server ] | In the cloud, applications can’t always know the exact location of other services. A service registry, such as [Netflix Eureka](https://github.com/Netflix/eureka), or a sidecar solution, such as [HashiCorp Consul](https://www.consul.io/), can help. Spring Cloud provides DiscoveryClient implementations for popular registries such as [Eureka](https://spring.io/projects/spring-cloud-netflix), [Consul](https://spring.io/projects/spring-cloud-consul), [Zookeeper](https://spring.io/projects/spring-cloud-zookeeper), and even [Kubernetes'](https://spring.io/projects/spring-cloud-kubernetes) built-in system. There’s also a [Spring Cloud Load Balancer](https://spring.io/guides/gs/spring-cloud-loadbalancer/) to help you distribute the load carefully among your service instances. |
| Cloud configuration[[Spring Cloud Config](https://spring.io/projects/spring-cloud-config) Server ] | In the cloud, configuration can’t simply be embedded inside the application. The configuration has to be flexible enough to cope with multiple applications, environments, and service instances, as well as deal with dynamic changes without downtime. [Spring Cloud Config](https://spring.io/projects/spring-cloud-config) is designed to ease these burdens and offers integration with version control systems like Git to help you keep your configuration safe. |
| Circuit breakers[ [Hystrix](https://github.com/Netflix/Hystrix/wiki) ]Fault tolerance / resilience | Distributed systems can be unreliable. Requests might encounter timeouts or fail completely. A circuit breaker can help mitigate these issues, and [Spring Cloud Circuit Breaker](https://spring.io/projects/spring-cloud-circuitbreaker) gives you the choice of three popular options: [Resilience4J](https://resilience4j.readme.io/docs/getting-started), [Sentinel](https://github.com/alibaba/Sentinel/wiki/Circuit-Breaking), or [Hystrix](https://github.com/Netflix/Hystrix/wiki). |
| Tracing[[Spring Cloud Sleuth](https://spring.io/projects/spring-cloud-sleuth) ][ with [Zipkin](https://zipkin.io/) Server ] | Debugging distributed applications can be complex and take a long time. For any given failure, you might need to piece together traces of  information from several independent services. [Spring Cloud Sleuth](https://spring.io/projects/spring-cloud-sleuth) can instrument your applications in a predictable and repeatable way. And [Zipkin](https://zipkin.io/),  is an open-source tracing system designed specifically to trace calls between **microservices**. It is especially useful for analyzing latency problems. Zipkin includes both instrumentation libraries and the collector processes that gather and store tracing data  **Zipkin** is useful during debugging when lots of underlying systems are involved and the application becomes slow in any particular situation. In such case, we first need to identify see which underlying service is actually slow. Once the slow service is identified, we can work to fix that issue. Distributed tracing helps in identifying that slow component among in the ecosystem.  Internally it has 4 modules –   1. **Collector** – Once any component sends the trace data arrives to Zipkin collector daemon, it is validated, stored, and indexed for lookups by the Zipkin collector. 2. **Storage** – This module store and index the lookup data in backend. [Cassandra](https://cassandra.apache.org/), [ElasticSearch](https://www.elastic.co/) and [MySQL](https://howtodoinjava.com/mysql/how-to-installuninstallexecute-mysql-as-windows-service/) are supported. 3. **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. 4. **Web UI** – A very nice UI interface for viewing traces.   **Sleuth**  [Sleuth](https://cloud.spring.io/spring-cloud-sleuth/) is a tool from Spring cloud family. It is used to generate the ***trace id***, ***span id***and add these information to the service calls in the headers and MDC, so that It can be used by tools like Zipkin and [ELK](https://howtodoinjava.com/microservices/elk-stack-tutorial-example/) etc. to store, index and process log files. As it is from spring cloud family, once added to the CLASSPATH, it automatically integrated to the common communication channels like –   * requests made with the [RestTemplate](https://howtodoinjava.com/spring-boot2/resttemplate/spring-restful-client-resttemplate-example/) etc. * requests that pass through a [Netflix Zuul](https://howtodoinjava.com/spring/spring-cloud/spring-cloud-api-gateway-zuul/) microproxy * HTTP headers received at [Spring MVC](https://howtodoinjava.com/spring-mvc-tutorial/) controllers * requests over messaging technologies like Apache Kafka or RabbitMQ etc. |
| Testing | Contract-based testing is one technique that high-performing teams often use to stay on track. It helps by formalizing the content of APIs and building tests around them to ensure code remains in check. [Spring Cloud Contract](https://spring.io/projects/spring-cloud-contract) provides contract-based testing support for REST and messaging-based APIs with contracts written in Groovy, Java, or Kotlin. |
| **challenges for testing microservices architecture**  The first challenge is to define the microservice architecture in a project. Before we start development, we should think about:  \* services communication,  \* cyclic dependencies of each service,  \* distributed logging which may influence debugging,  \* security of each service,  \* data consistency / synchronization,  \* performance tracing,  \* failovers of each service.  \* docker container monitoring / kubernates monitoring  \*swagger documentation (end-point test)  **Strategy for testing microservices**  \* functional testing ( business related test cases)  \* non-functional testing ( failovers, recovery, performance, security)  **Testing microservices – best practices**  \* static code tests,  \* unit tests to validate each function of a microservice,  \* component tests to validate service itself,  \* integration tests, especially contract tests.  \* Verify the right security of services.  \* Verify performance of the system.  \* Test how your entire system behaves when one of the services is down (for example, using  the Chaos Monkey tool).  \* Test if your system is discovered automatically when it’s up  **Testing microservices – tools ( Third party api ,example )**  There are many tools that can help you with your testing microservices strategy.  \* **Chaos Monkey** – the previously mentioned resiliency tool from Netflix will help you with  managing and preventing the consequences of random services instance failure and test the  behavior of each service.  \* **JMeter, Locust, K6** – some of the most common performance and load testing tools and other  tools like these.  \* **Pact** – to facilitate your contract testing.  **\* Mocha, Jest, Supertest** or any other popular integration and unit testing tool and microservices  testing framework. | |

1. **Setup Microsrvices [Spring cloud Based Architechture]**
2. Zuul Proxy server

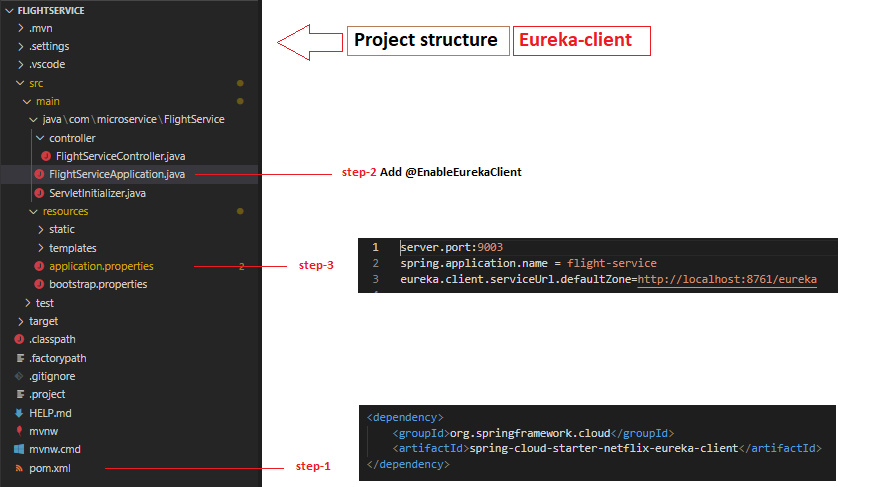
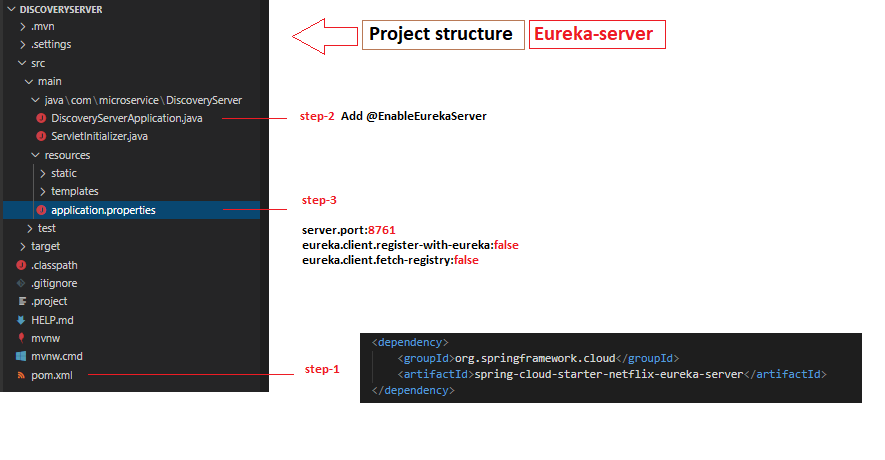
**Quick setup**

****

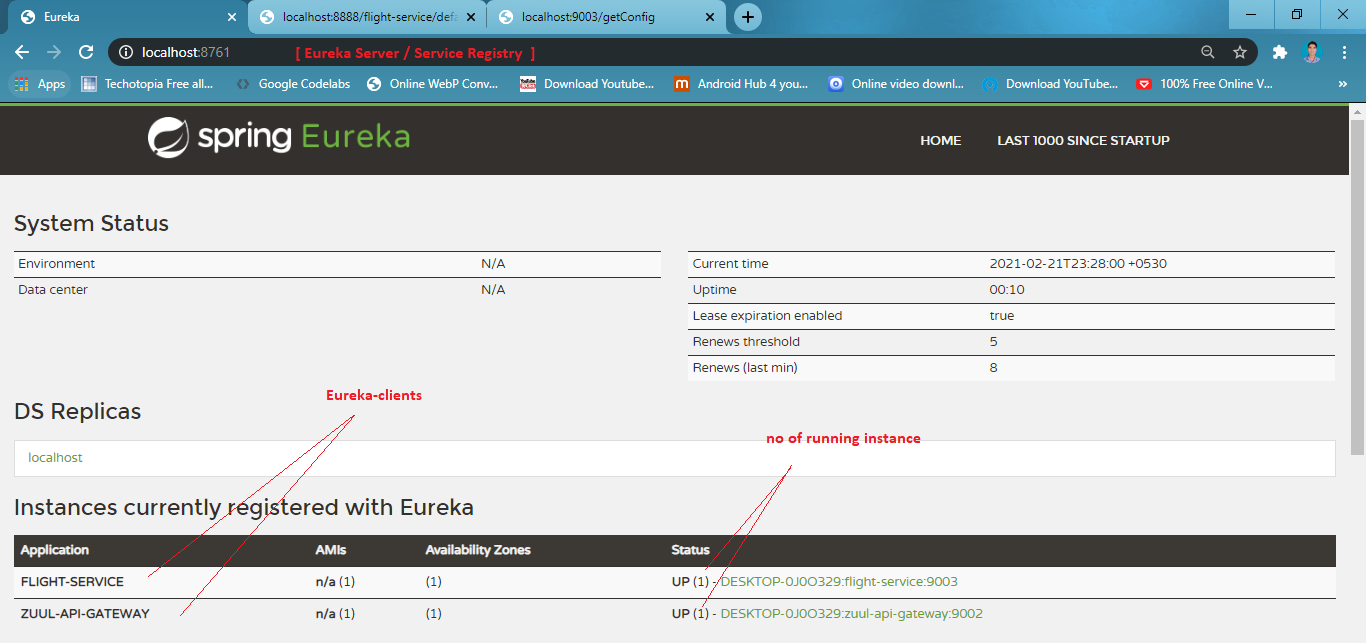
**Output:**

****

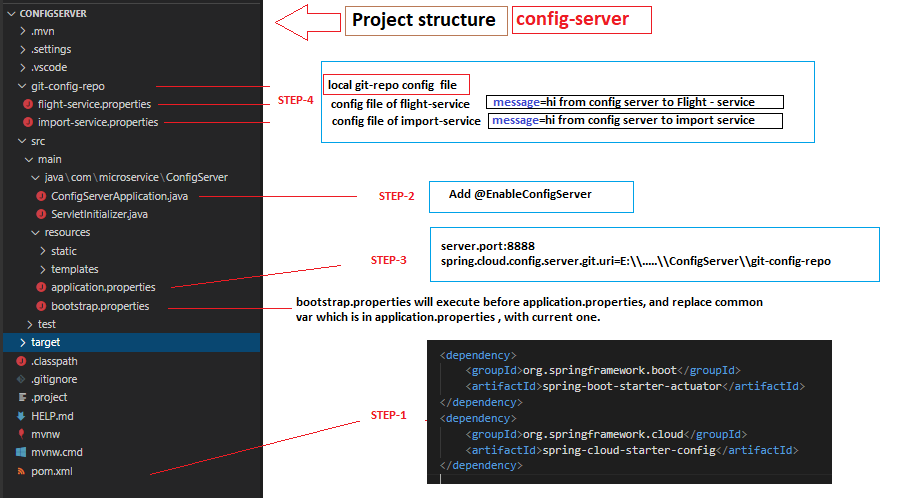
1. Eureka - server

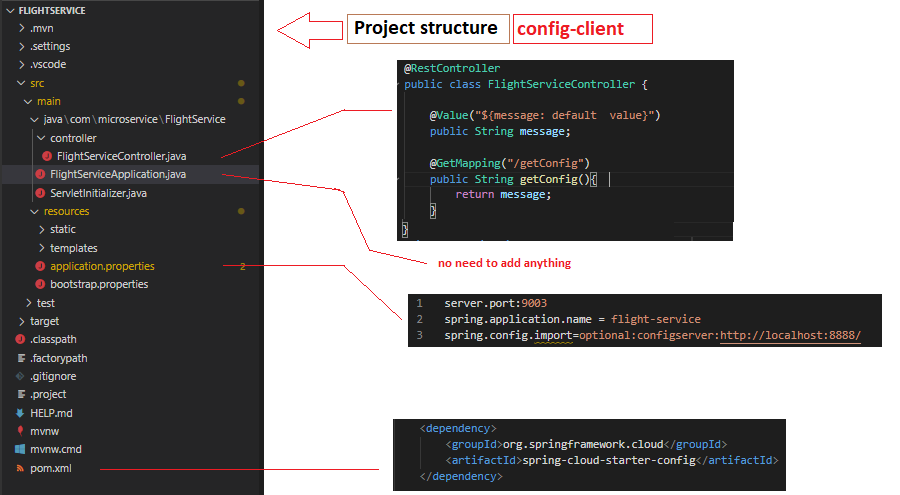


Output:-

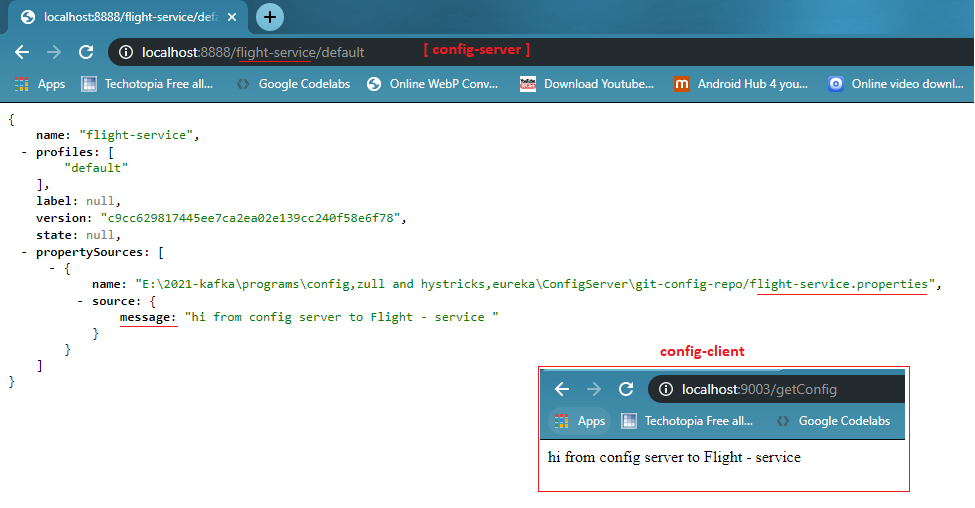


1. config server

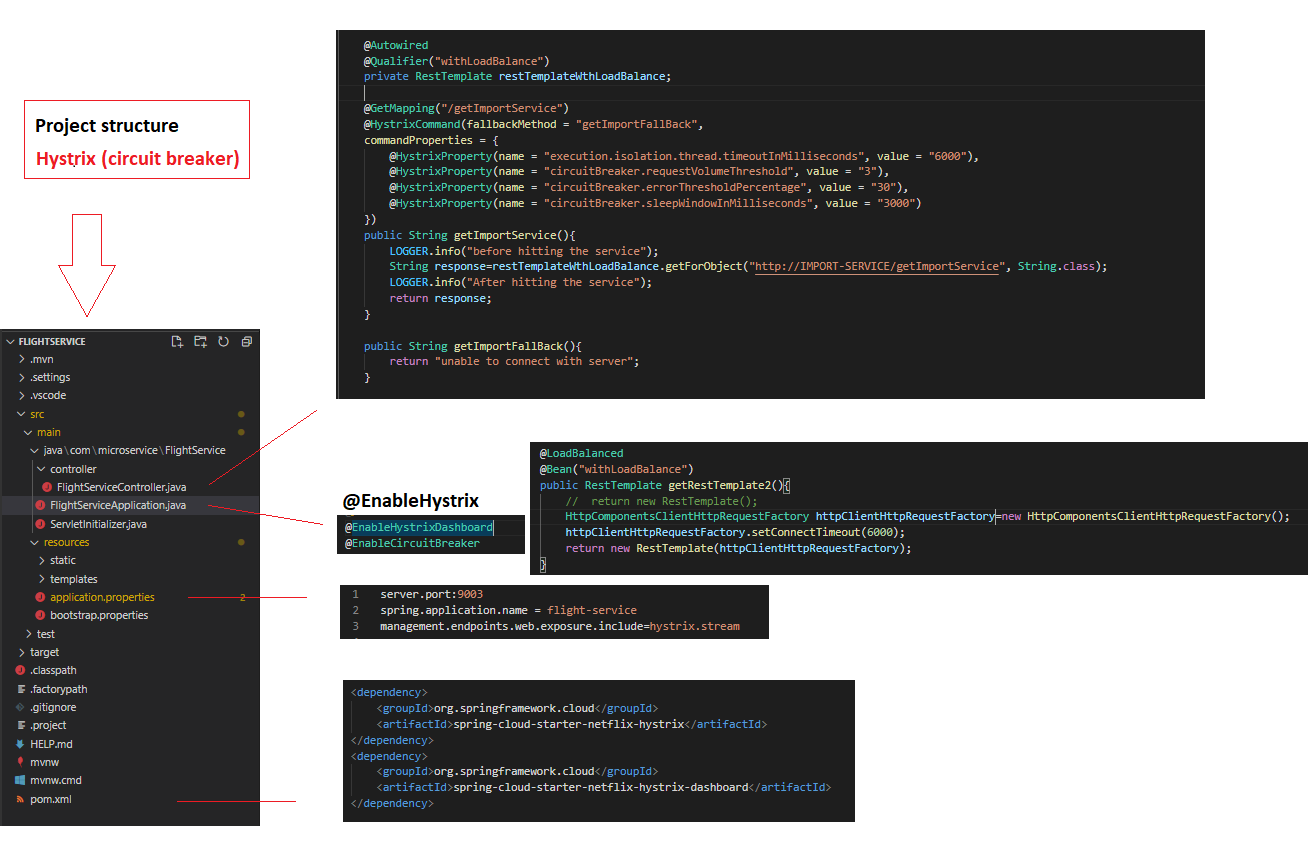
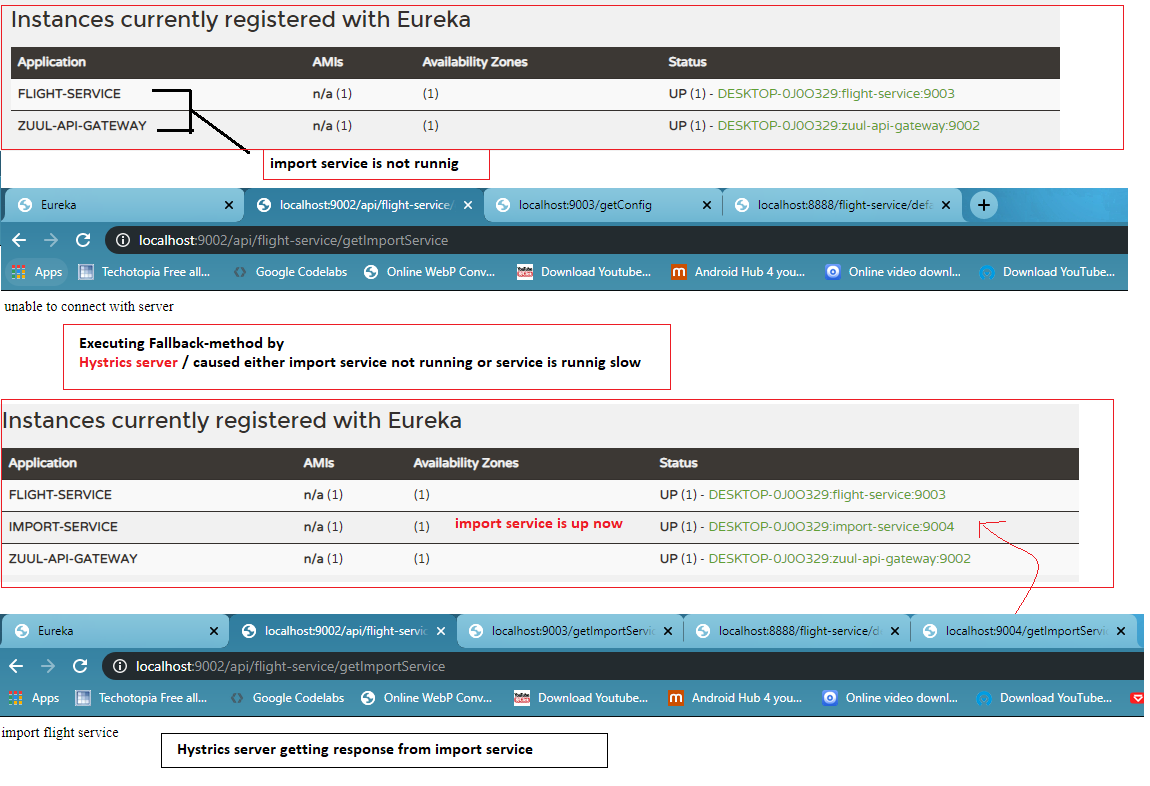




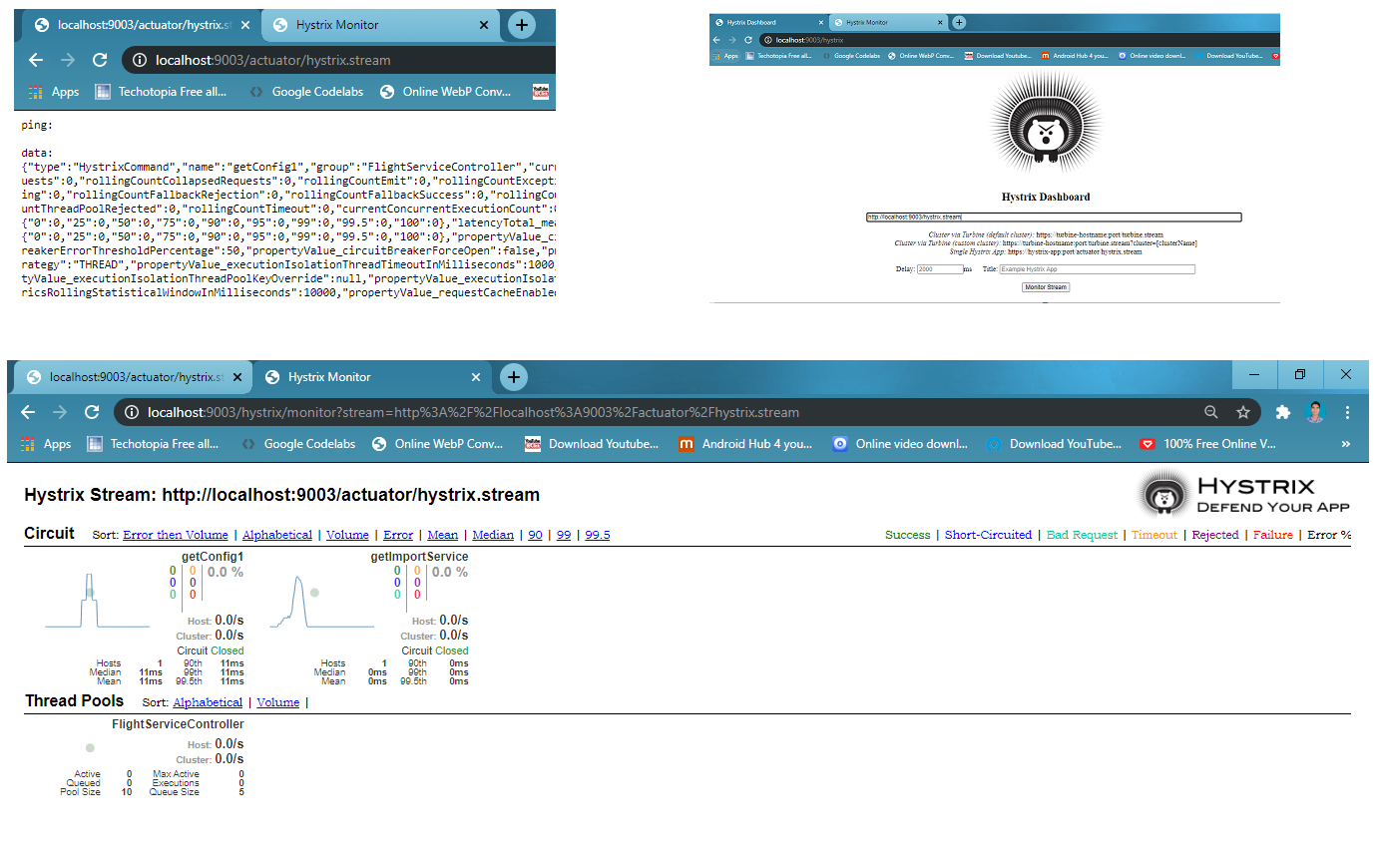
Output : -



1. Hystrix Circuit bracker and Hystrix Dashboard

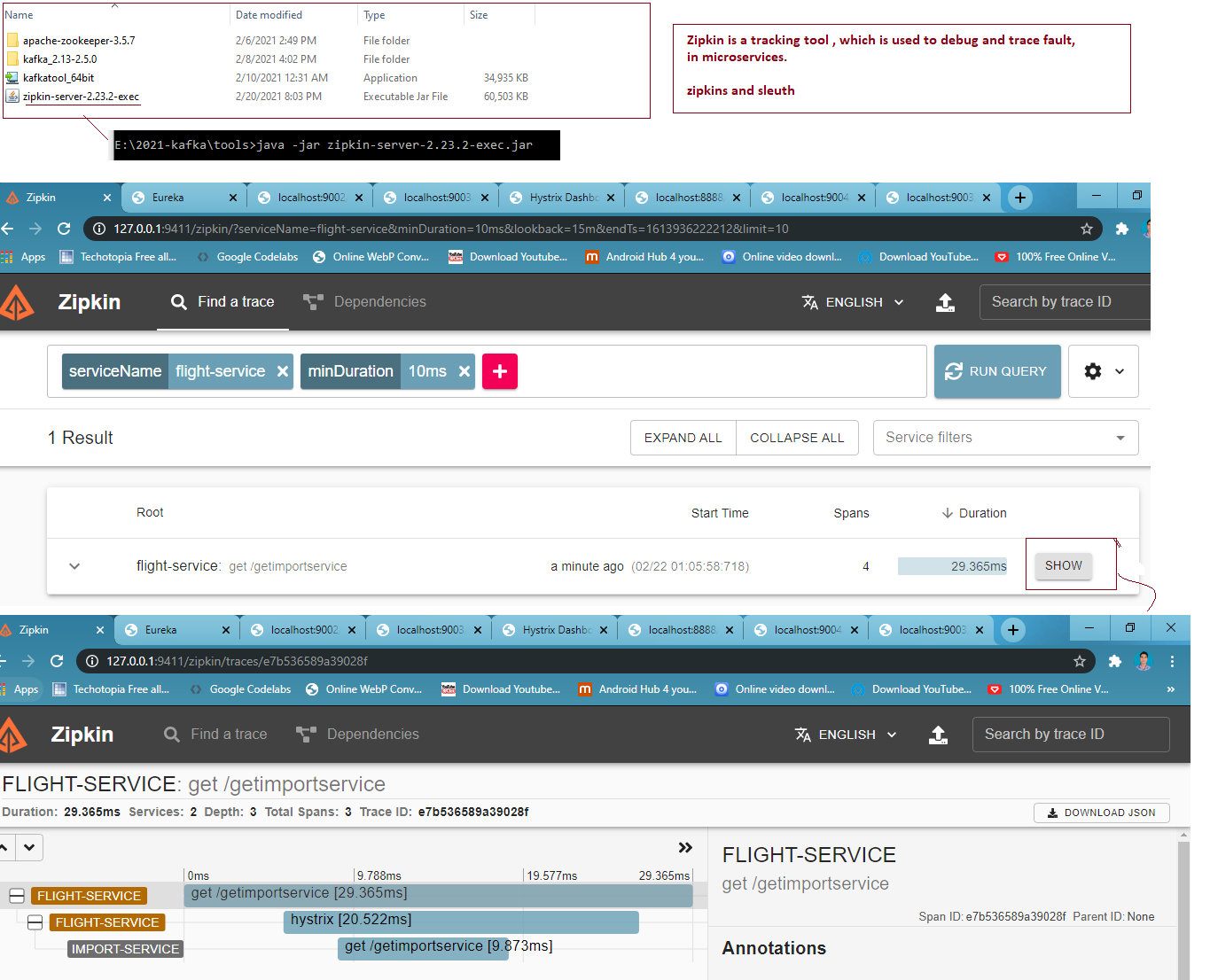
 

Output :-



1. zipkin server

##### E:\2021-kafka\screen short\5 zipkins-server.png



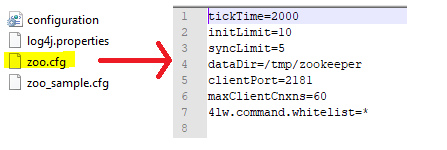
1. **kafka & kafka-stream**

* **kafka setup (**download and unzip **apache-zookeeper** and **kafka** for window **)**

****

* 1. **Add zoo.cfg file on config folder**

**apache-zookeeper-3.5.7\apache-zookeeper-3.5.7-bin\conf**

****

* 1. **Change server.properties in apache kafka**

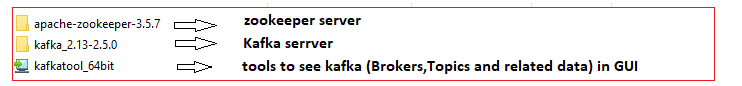
**kafka\_2.13-2.5.0\config**

****

* **Zooker and kafka basic running commands (use cmd )**

|  |
| --- |
| **# Start zookeeper server**  apache-zookeeper-3.5.7\apache-zookeeper-3.5.7-bin\zkServer.cmd |
| **# start kafka server**  .\bin\windows\kafka-server-start.bat .\config\server.properties  **# create topics in kafka**  bin\windows\kafka-topics --bootstrap-server localhost:9092 --create --topic mytopic partitionCount:1 replicationFactor:1  **# list all topics**  bin\windows\kafka-topics --bootstrap-server localhost:9092 –list  **# describes topic (topic name, partitions,leader ,replication and insinc replicas(isr))**  bin\windows\kafka-topics --bootstrap-server localhost:9092 --describe --topic mytopic      **--------- create kafka producer for individual created topics --------------**  bin\windows\kafka-console-producer --bootstrap-server localhost:9092 --topic mytopic  **--------- create kafka consumer for individual created topics ---------------**  **[ read from beginning ]**  bin\windows\kafka-console-consumer --bootstrap-server localhost:9092 --topic mytopic --from-beginning    **[ read from currently produced msg ]**  bin\windows\kafka-console-consumer --bootstrap-server localhost:9092 --topic mytopic  **# consumer group list**  bin\windows\kafka-consumer-groups --bootstrap-server localhost:9092 --list  bin\windows\kafka-consumer-groups --bootstrap-server localhost:9092 --describe --group group\_name  **-------------- delete topics --------------------------------------------------**  delete.topic.enable=true (add in server.properties)  bin/kafka-topics.sh --zookeeper localhost:2181 --delete --topic test (delete topicss) |

* **Working with zookeeper and kafka ( operating with console )**

****

* **Start zookeeper**

****

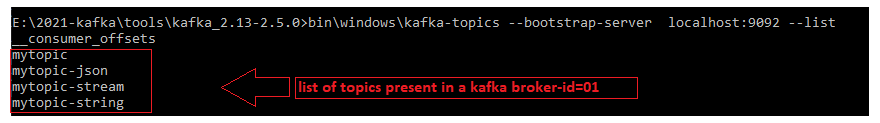
* **Start Kafka**

****

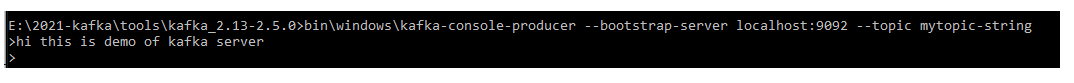
* **Create kafka-topic**

****

* **List kafka topics**

****

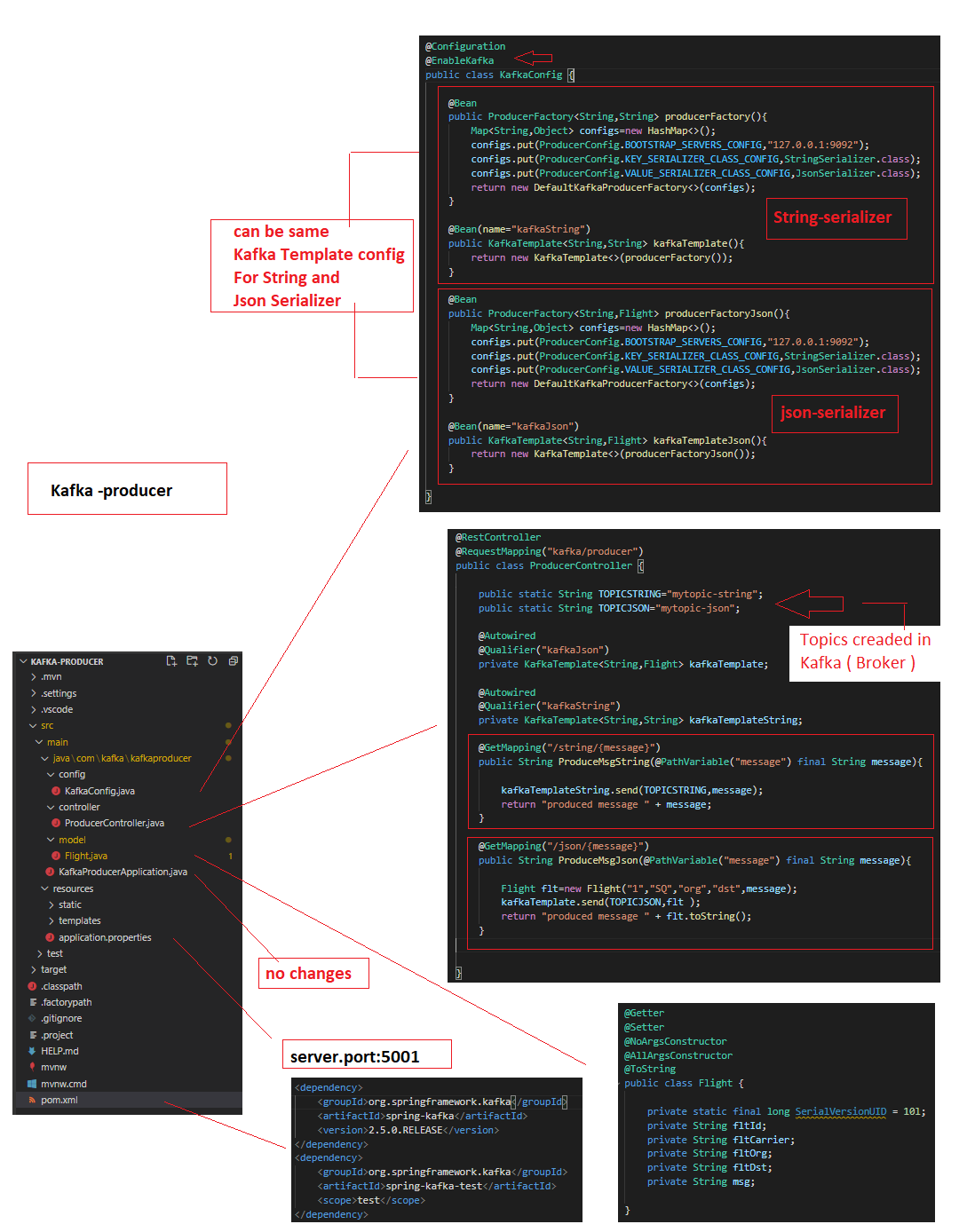
* **Create kafka-producer based on topic & Generate message**

****

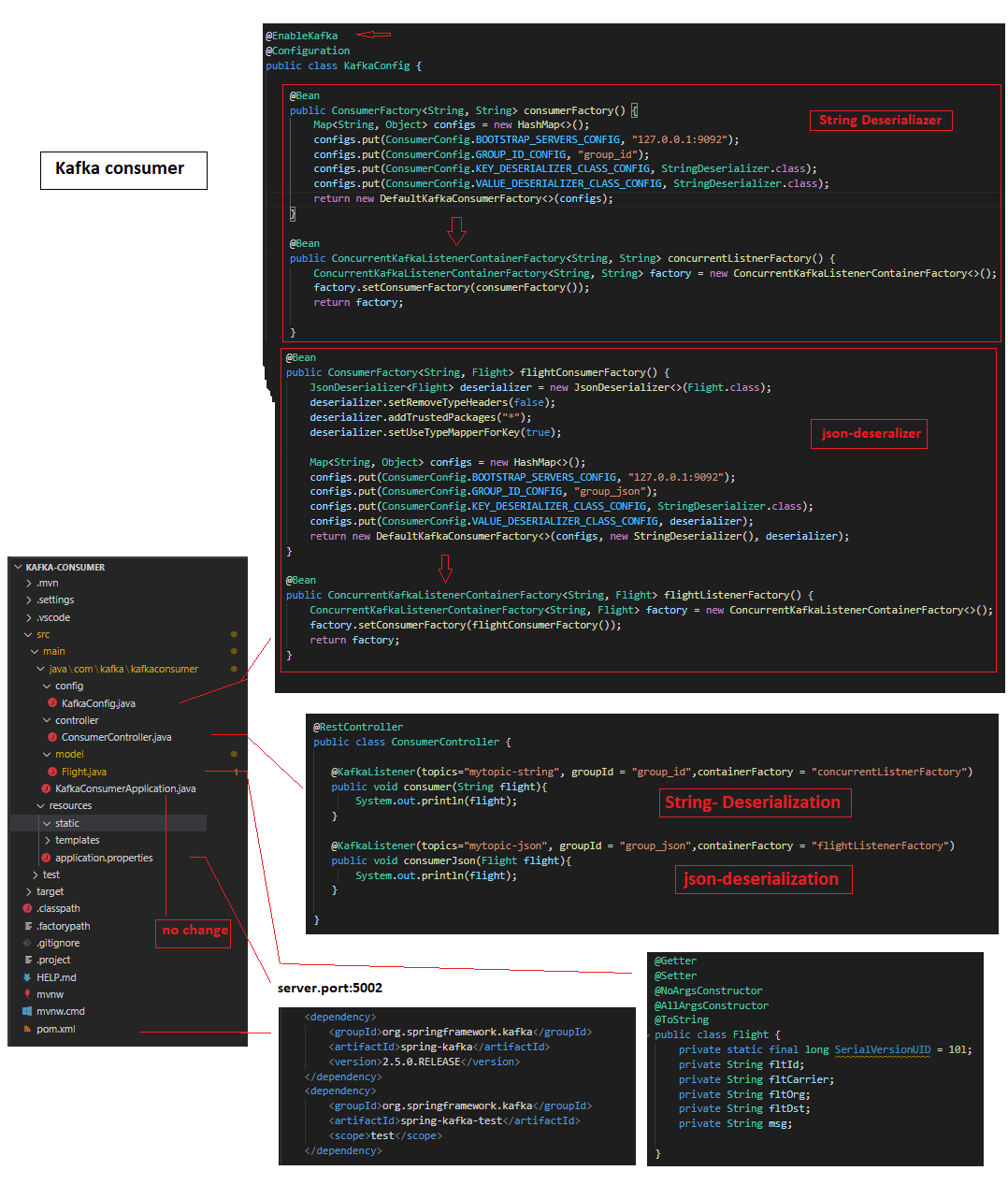
* **Create kafka-consumer based on topic & Consume message**

****

* **Working with kafka spring-boot project ( kafka & kafka-stream )**
* **Create Kafka-producer**

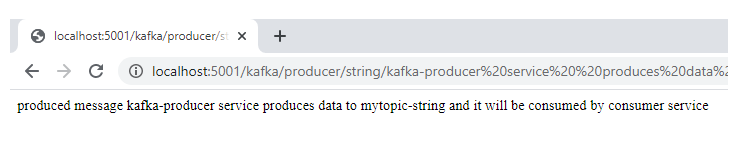
****

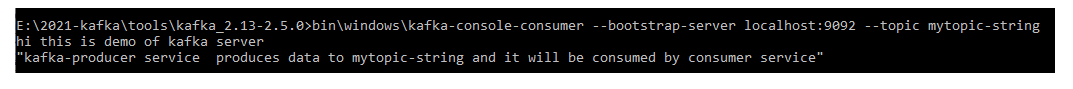
* **Create Kafka-consumer**

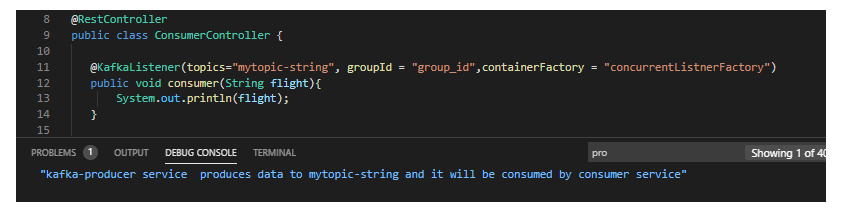
****

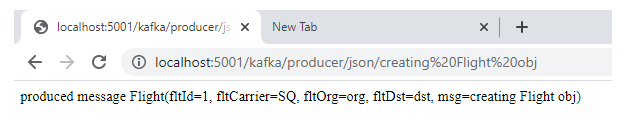
**Output : -**

**kafka-producer on mytopic-string => localhost:5001/kafka/producer/string/{msg}**

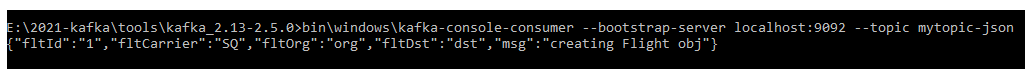
****

**Consumed by consumer ( cmd )**

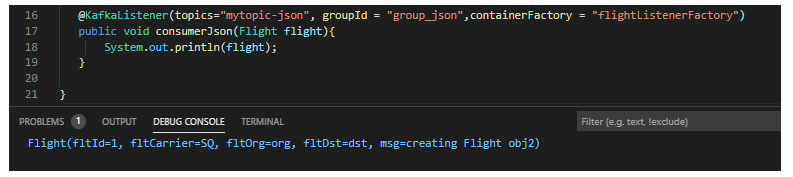
**Consumerd by consumer ( spring-boot-consumer-project )**

**kafka-producer on mytopic-json => localhost:5001/kafka/producer/json/{msg}**

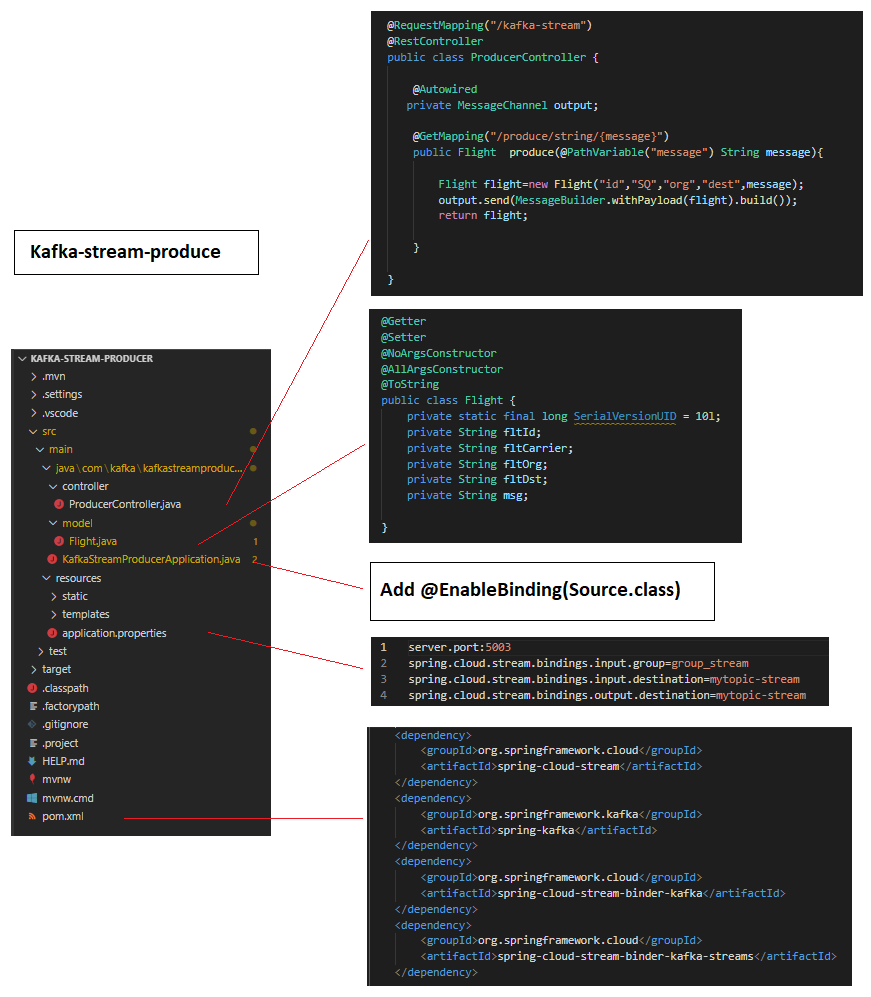
**Json consumer ( cmd )**

****

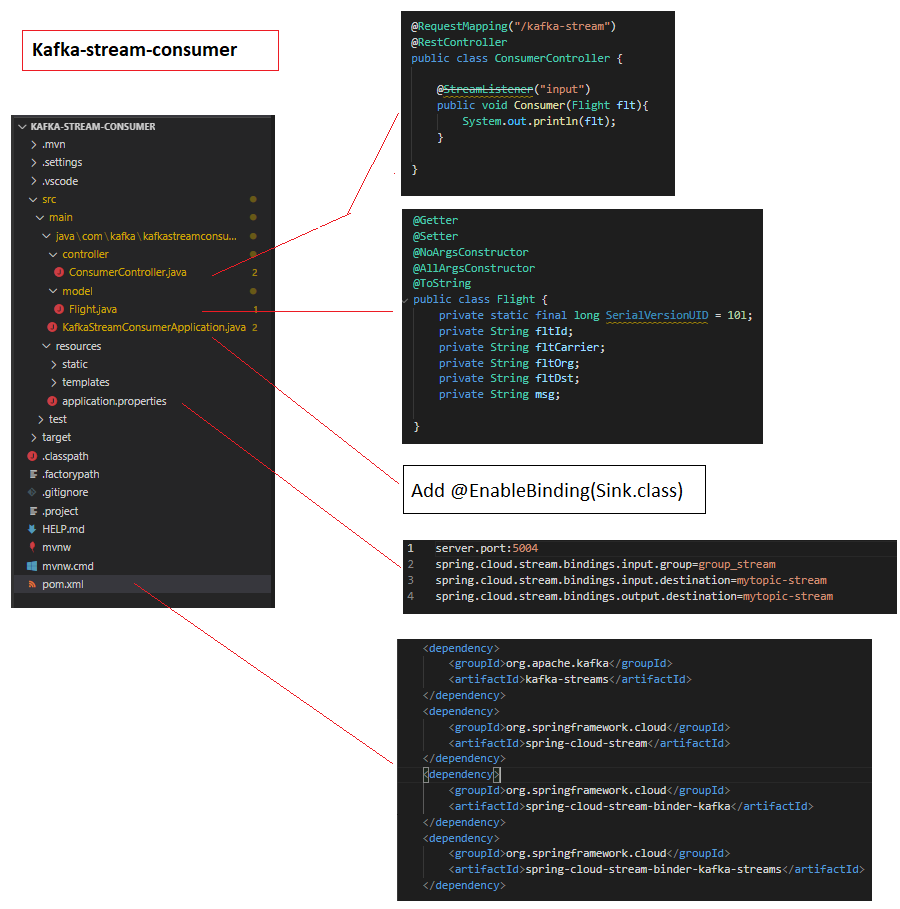
**Consumerd by consumer ( spring-boot-consumer-project )**

****

* **Create kafka-stream-producer**

****

* **Create kafka-stream-consumer**

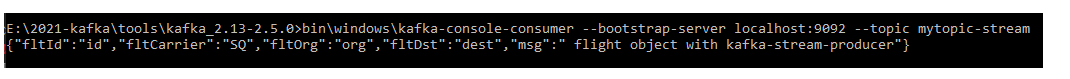
****

**Output : -**

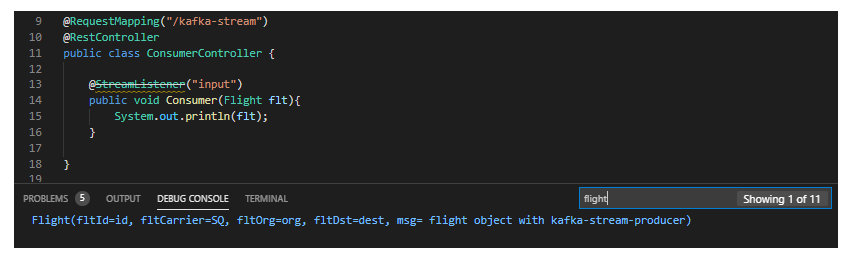
**Kafka-stream-Producer => localhost:5003/kafka-stream/producer/string/{msg}**

****

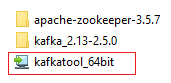
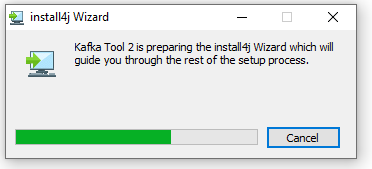
**consumer (console )**

****

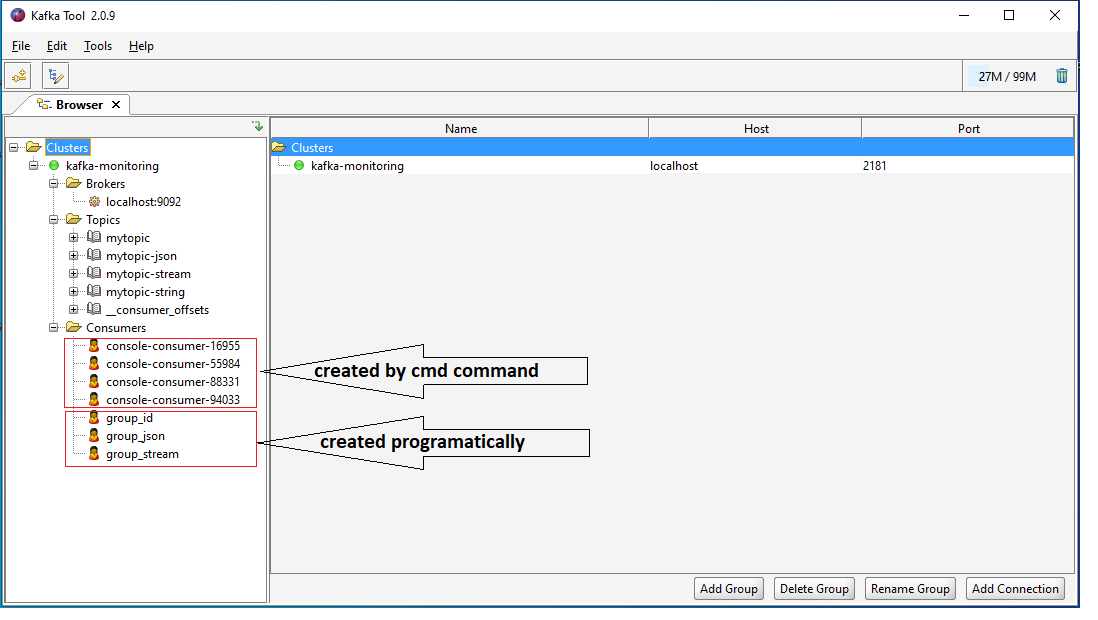
**Kafka-stream-consumer (project )**

****

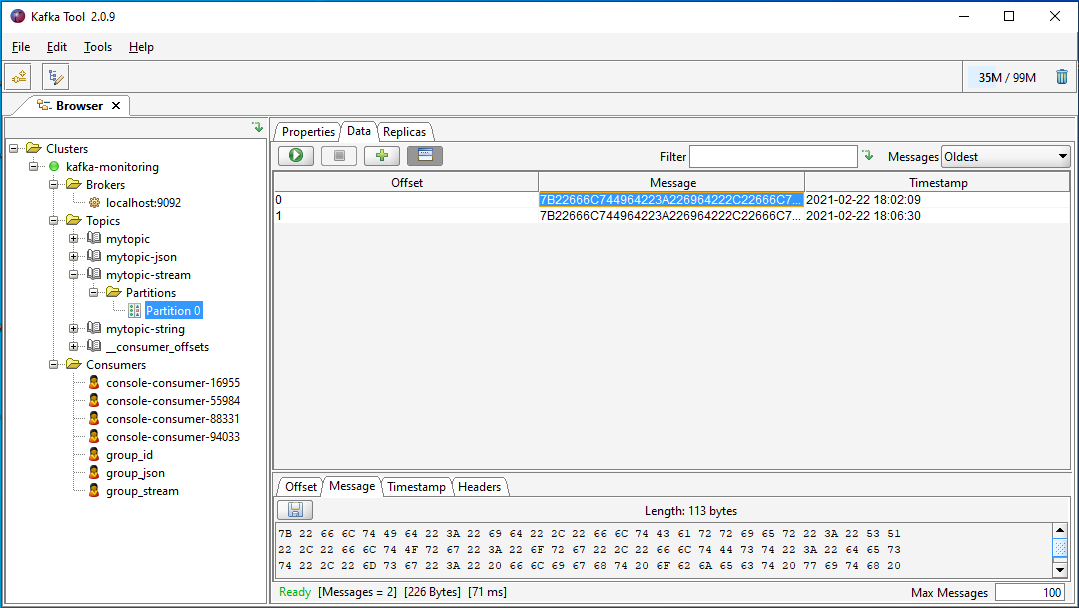
* **Kafka-gui tool installation**

** **

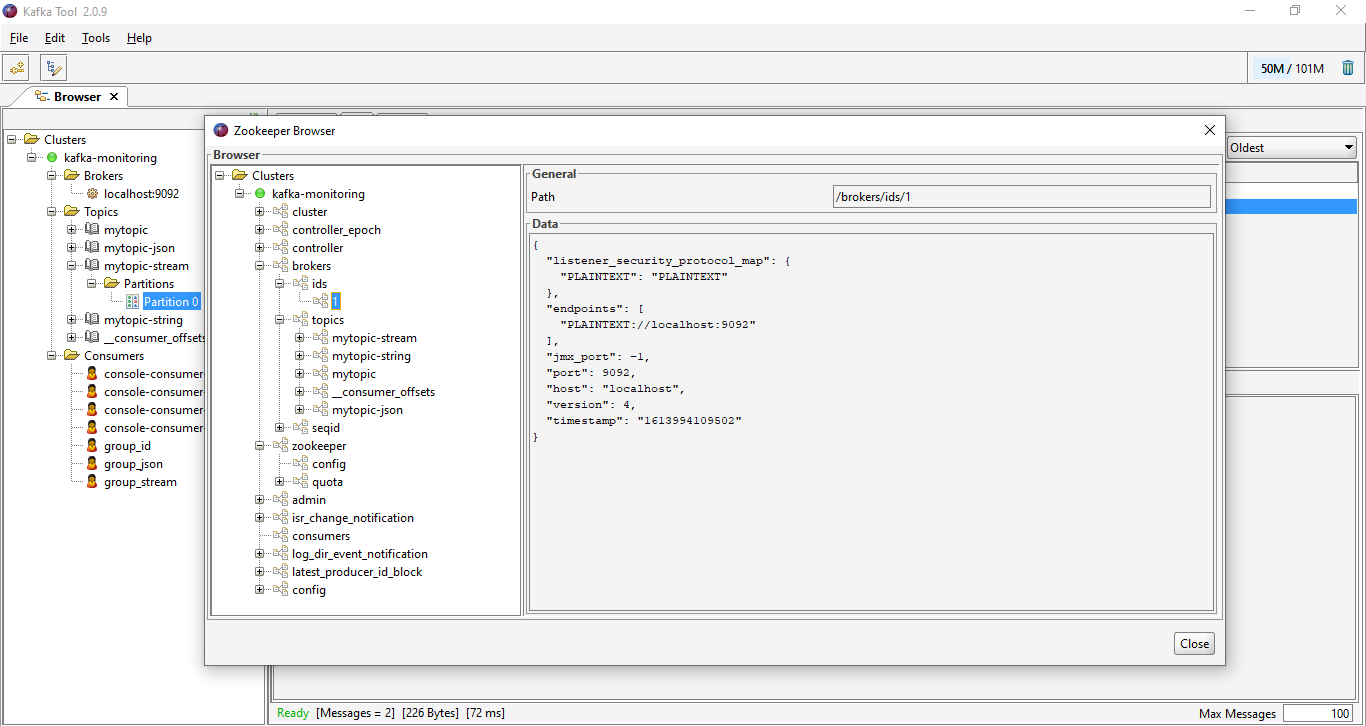
* **Download / install kafkatool**
* **GUI Guide**

****

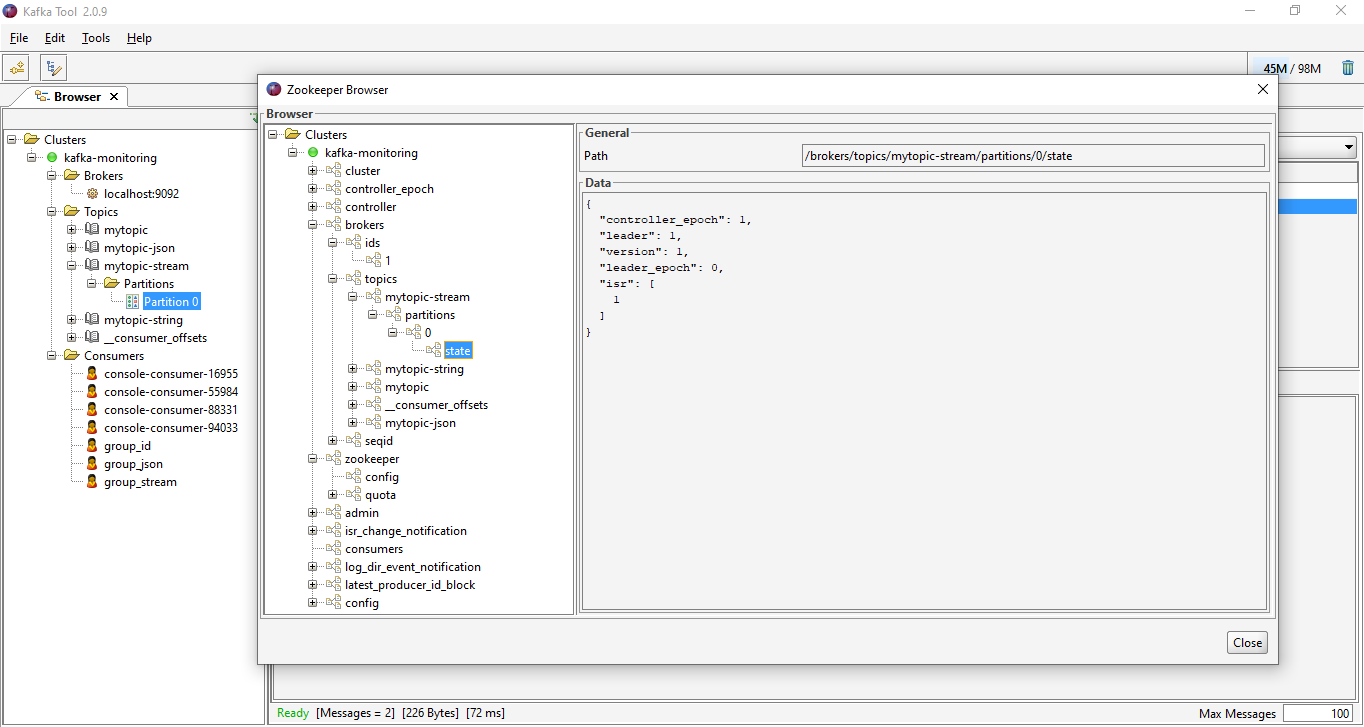
**Data saved in Broker->topic->partition0 (we created only one partitions of topic )**

****

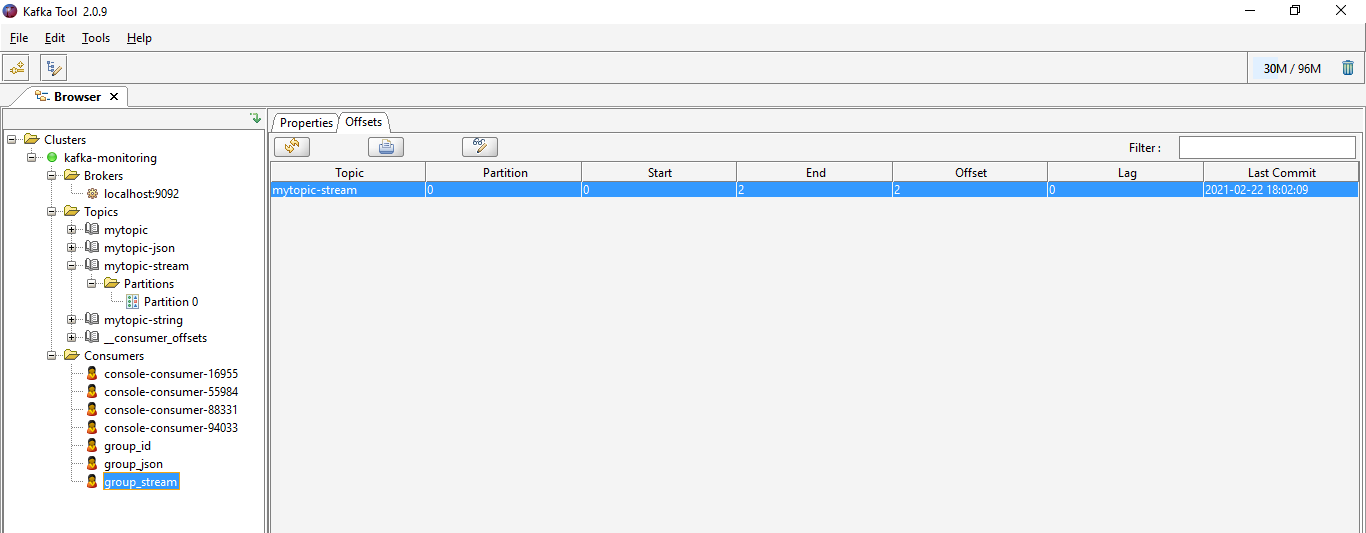
**Zookeeper browser**



Topic based partition info



Consumer-group offset info



1. **Setup Microservice -Asyncronous ( Reactive Programming with Reactor/WebFlux )**

**1. create project Reactor-Flight-service & Reactor-import-service**

* **Depandancies ( pom.xml )**

<dependency>

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

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

 </dependency>

* Add port in Application.properties (server.port:6001)
* Create controller flightServiceController.java

|  |
| --- |
| @*RestController*  *public* *class* FlightServiceController {  // 1. create WebClient for accessing other API call like RestTemplate  *private* *WebClient* *client* *=* *WebClient*.*create*("http://localhost:6002");  //2. return basic flux ,it will return response after collecting all value      @*GetMapping*("/flux")  *public* *Flux*<*Integer*> *getFlux*(){  *return* *Flux*.*just*(1,2,3)                     .*delayElements*(*Duration*.*ofSeconds*(1))                     .*log*();      }  //3. return basic mono      @*GetMapping*("/mono")  *public* *Mono*<*Integer*> *getMono*(){  *return* *Mono*.*just*(1)                     .*log*();      }  // 4. Return flux-stream , sequencely based on subscription/consumption of elements        @*GetMapping*(value*=*"/flux-stream",produces*=org*.*springframework*.*http*.*MediaType*.*APPLICATION\_STREAM\_JSON\_VALUE*)  *public* *Flux*<*Integer*> *getFluxStream*(){  *return* *Flux*.*just*(1,2,3,4,5,6,7)                     .*delayElements*(*Duration*.*ofSeconds*(1))                     .*log*();      }  // call other services in non-blocking manner and return flux-stream as ,it will consume data-steam one-by-one continuously until producer stop sending data.      @*GetMapping*(value*=*"/flightTest/flux-stream",produces*=org*.*springframework*.*http*.*MediaType*.*APPLICATION\_STREAM\_JSON\_VALUE*)  *public* *Flux*<*Integer*> *getFluxStream1*(){  *Flux*<*Integer*> *flux* *=* *client*.*get*()          .*uri*("/import/flux-stream")          .*accept*(*MediaType*.*APPLICATION\_STREAM\_JSON*)          .*exchange*() // depricated use ExchangeToFlux/Mono          .*flatMapMany*(response *->* *response*.*bodyToFlux*(*Integer*.*class*)).*log*();    *return* flux;      }  // commented mediaType for Test-Case success/failure      @*GetMapping*(value*=*"/flightTest2/flux-stream",produces*=org*.*springframework*.*http*.*MediaType*.*APPLICATION\_STREAM\_JSON\_VALUE*)  *public* *Flux*<*Integer*> *getFluxStream2*(){    *Flux*<*Integer*> *flux* *=* *client*.*get*()          .*uri*("/import/flux-stream")  *// .accept(MediaType.APPLICATION\_JSON\_UTF8)*          .*retrieve*()          .*bodyToFlux*(*Integer*.*class*); // collect response as flux  *return* flux;      }  // exchange method with more control      @*GetMapping*(value*=*"/flightTest3/flux-stream",produces*=org*.*springframework*.*http*.*MediaType*.*APPLICATION\_STREAM\_JSON\_VALUE*)  *public* *Flux*<*Integer*> *getFluxStream3*(){    *Flux*<*Integer*> *flux* *=* *client*.*get*()          .*uri*("/import/flux-stream")          .*exchangeToFlux*(response*->*{  *if* (*response*.*statusCode*().*equals*(*HttpStatus*.*OK*)) {  *return* *response*.*bodyToFlux*(*Integer*.*class*);              }  *else* *if* (*response*.*statusCode*().*is4xxClientError*()) {  *return* *response*.*bodyToFlux*(*Integer*.*class*);              }  *else* {  *return* *Flux*.*error*(*new* *RuntimeException*("message"));              }            });  *return* flux;      }  // collect mono from response body      @*GetMapping*(value*=*"/flightTest1/mono",produces*=org*.*springframework*.*http*.*MediaType*.*APPLICATION\_STREAM\_JSON\_VALUE*)  *public* *Mono*<*Integer*> *getMono1*(){    *Mono*<*Integer*> *mono=*client                              .*get*()                              .*uri*("/import/mono")                              .*exchangeToMono*(response*->*{  *return* *response*.*bodyToMono*(*Integer*.*class*);                              });  *return* mono;      }  } |

* **FlightControllerServiceTest.java ( TestCases )**

|  |
| --- |
| @*WebFluxTest* // 1. <= Add @WebFluxTest  *public* *class* FlightControllerTest {        @*Autowired*  *WebTestClient* *webTestClient*; // <= Aurowire WebTestClient for Rest Api call (Reactive web)  // Note = AutoWire TestRestTemplate for (non-reactive)      @*Test*  *public* *void* *flux\_approach1*(){  *Flux*<*Integer*> *integerFlux=webTestClient*.*get*().*uri*("/flux")          .*accept*(*MediaType*.*APPLICATION\_JSON\_UTF8*) // define response media type should be…          .*exchange*()          .*expectStatus*().*isOk*() // set expectedStatus          .*returnResult*(*Integer*.*class*)          .*getResponseBody*();  *StepVerifier*.*create*(integerFlux) // use StepVerifier to test result in (Reactive-web)          .*expectSubscription*()          .*expectNext*(1) // first response should be 1 and next … next …until VerifyComplete();          .*expectNext*(2) // expectNext will check OnNext(2) is coming or not          .*expectNext*(3)          .*verifyComplete*(); // verifyComplete will check onComplete() response is coming or not      }      @*Test*  *public* *void* *flux\_approach2*(){  *webTestClient*.*get*().*uri*("/flux")          .*accept*(*MediaType*.*APPLICATION\_JSON\_UTF8*)          .*exchange*()          .*expectStatus*().*isOk*()          .*expectHeader*()          .*contentType*(*MediaType*.*APPLICATION\_JSON\_UTF8*)          .*expectBodyList*(*Integer*.*class*)          .*hasSize*(3);       // do not verify Step-by-step , check response size only      }      @*Test*  *public* *void* *flux\_approach3*(){  *List*<*Integer*> *expectedList=Arrays*.*asList*(1,2,3,4);  *EntityExchangeResult*<*List*<*Integer*>> *entityExchangeResult* *=*  webTestClient                  .*get*().*uri*("http://localhost:6001/flightTest2/flux-stream")  *//  .accept(MediaType.APPLICATION\_JSON\_UTF8)*                  .*exchange*()                  .*expectStatus*().*isOk*()                  .*expectBodyList*(*Integer*.*class*)                  .*returnResult*();  // check whether response list is as matching or not  *assertEquals*(expectedList, *entityExchangeResult*.*getResponseBody*());      }      @*Test*  *public* *void* *flux\_approach4*(){  *List*<*Integer*> *expectedList=Arrays*.*asList*(1,2,3);  // other way of matching response          webTestClient              .*get*().*uri*("/flux")              .*accept*(*MediaType*.*APPLICATION\_JSON\_UTF8*)              .*exchange*()              .*expectStatus*().*isOk*()              .*expectBodyList*(*Integer*.*class*)              .*consumeWith*((response)*->*{  *assertEquals*(expectedList, *response*.*getResponseBody*());              });      }    } |

**2. create project Reactor-import-service**

* **Depandancies ( pom.xml )**

<dependency>

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

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

 </dependency>

* Add port in Application.properties (server.port:6002)
* Create controller ImportServiceController.java
* There are more than 100 methods to create/filter/map flux and Mono we need use as per requirnment.

|  |
| --- |
| @*RestController*  *public* *class* ImportServiceController {      @*GetMapping*(value *=* "import/flux-stream", produces *=* *org*.*springframework*.*http*.*MediaType*.*APPLICATION\_STREAM\_JSON\_VALUE*)  *public* *Flux*<*Integer*> *getFluxStream*() {  *return* *Flux*.*just*(1, 2, 3, 4)                     .*delayElements*(*Duration*.*ofSeconds*(1))                     .*log*();      }      @*GetMapping*(value *=* "/import/flux2", produces *=* *org*.*springframework*.*http*.*MediaType*.*APPLICATION\_STREAM\_JSON\_VALUE*)  *public* *Flux*<*String*> *getFlux2*() {  *List*<*String*> *words* *=* *Arrays*.*asList*("the", "quick", "brown");  *Flux*<*String*> *manyLetters* *=* *Flux*                  .*fromIterable*(words)                  .*flatMap*(word *->* *Flux*.*fromArray*(*word*.*split*("")))                  .*distinct*()                  .*sort*()                  .*zipWith*(*Flux*.*range*(1, *Integer*.*MAX\_VALUE*), (string, count) *->* *String*.*format*(" %2d. %s", count, string))                  .*delayElements*(*Duration*.*ofSeconds*(1))                  .*log*();  *return* manyLetters;      }      @*GetMapping*(value *=* "/import/flux1", produces *=* *org*.*springframework*.*http*.*MediaType*.*APPLICATION\_STREAM\_JSON\_VALUE*)  *public* *Flux*<*String*> *getFlux1*() {  *List*<*String*> *words* *=* *Arrays*.*asList*("the", "quick", "brown");  *Flux*<*String*> *manyWords* *=* *Flux*                                      .*fromIterable*(words)                                      .*delayElements*(*Duration*.*ofSeconds*(1))                                      .*log*();  *return* manyWords;      } |

* **RestTempate Test Cases ( demo for comparison with Reactive programming )**
* Using **RestTemplate**

|  |
| --- |
| @*GetMapping*(path*=*"/employees", produces *=* "application/json")  *public* *Employees* *getEmployees*()  {  *return* *employeeDao*.*getAllEmployees*();  } |
| @*Test*  *public* *void* *testGetEmployeeListSuccess*() throws *URISyntaxException*  {  *RestTemplate* *restTemplate* *=* *new* *RestTemplate*();    *final* *String* *baseUrl* *=* "http://localhost:" *+* randomServerPort *+* "/employees";  *URI* *uri* *=* *new* *URI*(baseUrl);    *ResponseEntity*<*String*> *result* *=* *restTemplate*.*getForEntity*(uri, *String*.*class*);    *//Verify request succeed*  *Assert*.*assertEquals*(200, *result*.*getStatusCodeValue*());  *Assert*.*assertEquals*(true, *result*.*getBody*().*contains*("employeeList"));  } |

* Using **TestRestTemplate**

|  |
| --- |
| @*RestController*  @*RequestMapping*(path *=* "/employees")  *public* *class* EmployeeController  {      @*Autowired*  *private* *EmployeeDAO* *employeeDao*;        @*PostMapping*(path*=* "/", consumes *=* "application/json", produces *=* "application/json")  *public* *ResponseEntity*<*Object*> *addEmployee*(                          @*RequestHeader*(name *=* "X-COM-PERSIST", required *=* true) *String* headerPersist,                          @*RequestHeader*(name *=* "X-COM-LOCATION", required *=* false, defaultValue *=* "ASIA") *String* headerLocation,                          @*RequestBody* *Employee* employee)  *throws* *Exception*      {  *//Generate resource id*  *Integer* *id* *=* *employeeDao*.*getAllEmployees*().*getEmployeeList*().*size*() *+* 1;  *employee*.*setId*(id);    *//add resource*  *employeeDao*.*addEmployee*(employee);    *//Create resource location*  *URI* *location* *=* *ServletUriComponentsBuilder*.*fromCurrentRequest*()                                      .*path*("/{id}")                                      .*buildAndExpand*(*employee*.*getId*())                                      .*toUri*();    *//Send location in response*  *return* *ResponseEntity*.*created*(location).*build*();      }  } |
| // TestCases  @*RunWith*(*SpringRunner*.*class*)  @*SpringBootTest*(webEnvironment*=WebEnvironment*.*RANDOM\_PORT*)  *public* *class* SpringBootDemoApplicationTests  {      @*Autowired*  *private* *TestRestTemplate* *restTemplate*;        @*LocalServerPort*  *int* *randomServerPort*;        @*Test*  *public* *void* *testAddEmployeeSuccess*() *throws* *URISyntaxException*      {  *final* *String* *baseUrl* *=* "http://localhost:"*+*randomServerPort*+*"/employees/";  *URI* *uri* *=* *new* *URI*(baseUrl);  *Employee* *employee* *=* *new* *Employee*(null, "Adam", "Gilly", "test@email.com");    *HttpHeaders* *headers* *=* *new* *HttpHeaders*();  *headers*.*set*("X-COM-PERSIST", "true");    *HttpEntity*<*Employee*> *request* *=* *new* *HttpEntity*<>(employee, headers);    *ResponseEntity*<*String*> *result* *=* *this*.*restTemplate*.*postForEntity*(uri, request, *String*.*class*);    *//Verify request succeed*  *Assert*.*assertEquals*(201, *result*.*getStatusCodeValue*());      }  } |

1. Referances

* <https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html>
* <https://projectreactor.io/docs>
* <https://spring.io/projects/spring-cloud>
* <https://kafka.apache.org/documentation/>
* <https://www.youtube.com/playlist?list=PLqq-6Pq4lTTZSKAFG6aCDVDP86Qx4lNas> (microservices java brain)
* <https://www.youtube.com/playlist?list=PLxv3SnR5bZE82Cv4wozg2uZvaOlDEbO67> (kafka)
* <https://www.youtube.com/playlist?list=PLnXn1AViWyL70R5GuXt_nIDZytYBnvBdd> (Reactive programming)
* Others websites … ( based on top-search record on google with related topics )

Thanks