Spring Boot + spring cloud-microservices

What is Microservices based Architecture and why to use it:

- **Microservices based architecture means you should create small autonomous component of any business functions.**
- **Example**

you have an e-commerce application with various modules

It has thousands of products that users can search and then purchase. There would be several components, such as Product Search/Catalog, Core Function,

Billing,

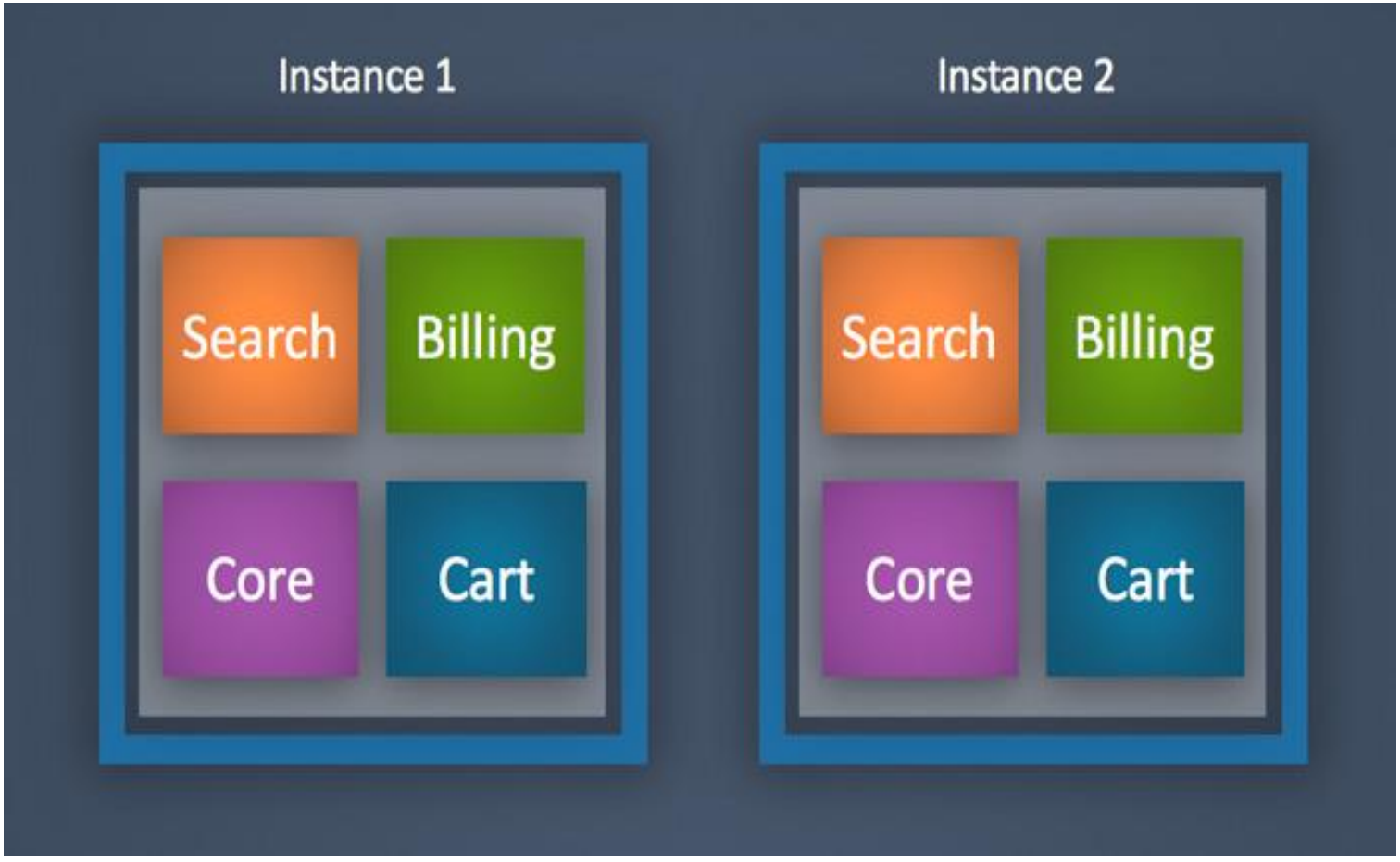
Shipping & Shopping Cart etc.

Generally we will have all these functionalities developed as one package (EAR in Java) then deployed to one or more application server in production environment. This is a typical monolithic application architecture style.

Why use microservice?

- **all the functionalities you have developed are not evenly used by the users.**
- **The e-commerce application might have 1000s of products, the search should be the most used functions.**
- **So in case if you want to increase the throughput of search, you need to deploy the entire application to another application server with all the functionalities, which is not really the objective though.**

- **Scalability is a major obstacle in Monolithic application architecture.**
- **Also there is risk, any issue with Billing (for an example) would impact the entire application and user won't be able to use search,**
- **since all the functions are part of a single package/container/JVM. Even if, say the billing component is not working we should allow users to search products and store them in the cart.**


Instance 1
Search
Billing
Core
Cart
Instance 2
Search
Billing
Core
Cart

- In a Microservices based Architecture the idea is to deploy autonomous business functions
- such as Search,
- Billing,
- Cart as separate service.
- Each service can communicate using a lightweight api like JSON.
- So now we can have more number of Search instances.
- This helps us to scale the application based on functions used.
- In case any service is down it shouldn't impact the other functions since all of them are packaged and deployed separately.

characteristics

- **Modularization by services.**
- **Autonomous endpoints loosely coupled with simplified lightweight API like REST.**
- **Build capabilities/functions and not projects/application.**
- **Decentralized Data Strategy.**
- **Fault tolerance by design.**
- **The final architecture can change\evolve over a period of time.**
- **DevOps – Developers friendly automated infrastructure.**

Advantages of Microservices:

- **Independent and Dynamic Scaling:**

each services can be scaled differently than others. This is a huge benefit towards cost, infrastructure and operation. In case Cloud Native Application, you can dynamically bring more instances of any services when the demand it high. This is big.

- **Use of any Technology:**

You can have different technologies for each of the services. In monolithic application you need to choose a single language/technology for all the functions, however you can choose any technology for the Microservices as long as they can communicate using a lightweight API like REST.

- **Continuous Delivery & Integration:**

Microservices supports continuous delivery and integration. You can also have different small teams working on these services. This helps to develop faster and better software. Project management and tracking becomes very easy with Microservices.

- **Easy to Replace:**

You can easily replace part of your functionality without impacting the entire application. In case of monolithic application change of any business functions imposes a big risk and we end up retesting the entire application to make sure none of the other functions are impacted. However in case of Microservices based application, you can have legacy technology in your application and you can easily replace them slowly without impacting other part of the application.

- **Time to market:**

After the initial rollout, you can have small releases, addition of more functions/services deployed much faster then its possible to a monolithic application since the impacts are localized and far less risky.

- **Framework Standardization:**
- **In real world, we may have many small teams (or groups) working on developing the services. Now there could be a situation where each team develops the services very differently than each other. Later when we switch/rebuild teams its difficult to understand the code since each of the services are structured in different way. Spring Boot comes as the choice of a single framework so that the structure of the each services looks similar even though the business function will be different. Spring Boot helps with standardization of the services.**

- **Convention over Configuration:**
- **As you have seen in the Introduction of the Spring Boot section, Spring Boot has replaced all the XML configuration of Spring by simple Annotation. However you should be able to customize the configurations using annotation as well.**

- **Integrated Server for Development:**
- **Spring Boot attaches a Tomcat/Jetty server with the compiled Jar using Maven/Gradle. This helps the developer to run the application easily without going to the deployment process.**

- **Centralized Cloud based Configuration:**
- **Since we need to replicate all the configurations across multiple instances of the services, a Centralized Cloud based Configuration is a must needed function. Spring Boot provides a Cloud Server to manage and push the configurations to the services.**

- **12 Factor App style Configuration:**
- **Spring Boot also support the 12 Factor App style Configuration. If you want to read more about it, read it here—>**

- **Centralized Logging:**
- **You should have Centralized Logging with Microservices. Spring Boot has few Toolsets however your company might already have an enterprise solution for Centralized Logging.**

- **3rd Party Library Support:**

Spring Boot has taken a significant step and widen support for 3rd Party Open Source Library like Netflix OSS, No-SQL DB, Distributed Cache etc.

- **Security:**

security is another aspect where Spring Boot has support for OAuth 2.0 based authentication.

What is Netflix OSS?

- **Netflix is very popular with their online streaming services. They had to scale the services since their customer base was increasing and also had plan for expanding across geographic. They adapted Microservices architecture style to address the scalability, fault tolerance & availability. They created many softwares/tools to support the development which later they open sourced as part of Netflix Open Source Software (OSS).**

- **Here is the link to Netflix OSS:**

- **https://netflix.github.io/**

- **There are many tools available, however we will mainly look into the tools available under "Common Runtime Services & Libraries", such as Eureka, Zuul, Ribbon, Feign & Hystrix etc.**

- An application will have many business function exposed as services,
- Each service could also have more than one instance,
- There are different UI strategy that we will discuss later, however for now lets assume that all of our UI elements (HTML, CSS, JS etc) are inside the Web REST API service which is the UI point of entry.

NetFlix OSS

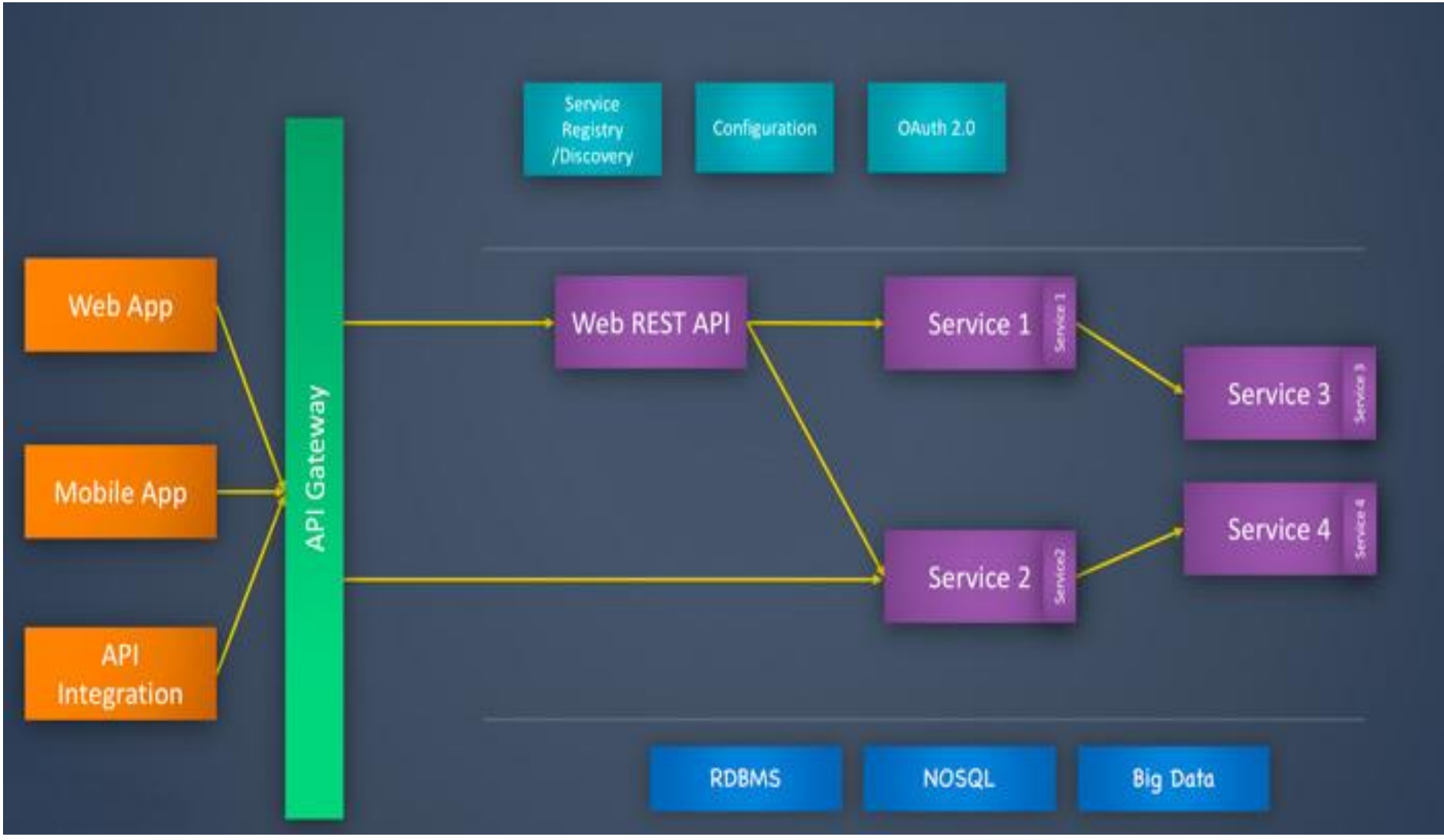
Operations Component	Netflix, Spring, ELK
Service Discovery server	Netflix Eureka
Dynamic Routing and Load Balancer	Netflix Ribbon
Circuit Breaker	Netflix Hystrix
Monitoring	Netflix Hystrix dashboard and Turbine
Edge Server	Netflix Zuul
Central Configuration server	Spring Cloud Config Server
OAuth 2.0 protected API's	Spring Cloud + Spring Security OAuth2
Centralised log analyses	Logstash, Elasticsearch, Kibana (ELK)

- **Netflix Eureka - Service Discovery Server Netflix Eureka allows microservices to register themselves at runtime as they appear in the system landscape.**

- **Netflix Ribbon - Dynamic Routing and Load Balancer Netflix Ribbon can be used by service consumers to lookup services at runtime. Ribbon uses the information available in Eureka to locate appropriate service instances. If more than one instance is found, Ribbon will apply load balancing to spread the requests over the available instances. Ribbon does not run as a separate service but instead as an embedded component in each service consumer.**

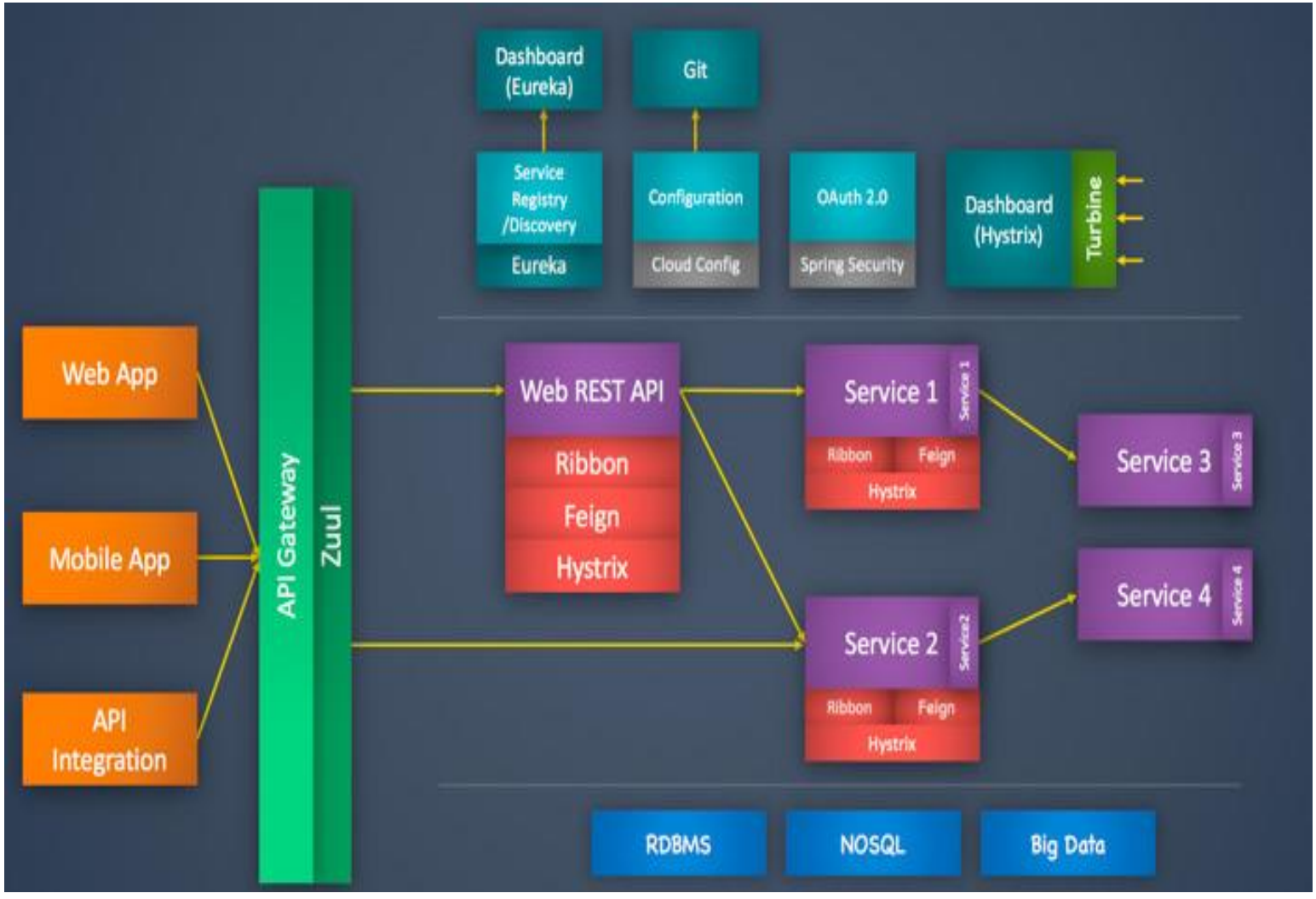
- **Netflix Zuul –**

Edge Server Zuul is (of course) our gatekeeper to the outside world, not allowing any unauthorized external requests pass through. Zuul also provides a well known entry point to the microservices in the system landscape. Using dynamically allocated ports is convenient to avoid port conflicts and to minimize administration but it makes it of course harder for any given service consumer. Zuul uses Ribbon to lookup available services and routes the external request to an appropriate service instance.


Service Registry /Discovery
Configuration
OAuth 2.0
Web App
Mobile App
API Integration
API Gateway
Web REST API
Service 1
Service 2
Service 3
Service 4
RDBMS
NOSQL
Big Data

- We would need an API Gateway to have a single URL at the client end and multiple URLs at the service end. The API Gateway can control the traffic (Server side routing) and route service invocation based on the client (Web App/ Mobile App/ API Integration).

- We would also need a service discovery so that the system can be aware of all the services and routing can use the information during runtime. In this way you can dynamically add/remove service instances. Centralized configuration and OAuth 2.0 are the other features needed.


Dashboard (Eureka)
Git
Service Registry /Discovery
Eureka
Configuration
Cloud Config
OAuth 2.0
Spring Security
Dashboard (Hystrix)
Turbine
Web App
Mobile App
API Integration
API Gateway
Zuul
Web REST API
Ribbon
Feign
Hystrix
Service 1
Service 1
Ribbon
Feign
Hystrix
Service 3
Service 3
Service 2
Service2
Ribbon
Feign
Hystrix
Service 4
Service 4
RDBMS
NOSQL
Big Data

Hystrix Circuit Breaker Pattern

- **Spring cloud Netflix stack component called Hystrix to implement circuit breaker while invoking underlying microservice.**
- **It is generally required to enable fault tolerance in the application where some underlying service is down/throwing error permanently, we need to fall back to different path of program execution automatically.**
- **This is related to distributed computing style of Eco system using lots of underlying Microservices. This is where circuit breaker pattern helps and Hystrix is an tool to build this circuit breaker.**

Hystrix configuration is done in four major steps.

1. Add Hystrix starter and dashboard dependencies.

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

2. Add @EnableCircuitBreaker annotation

3. Add @EnableHystrixDashboard annotation

4. Add annotation @HystrixCommand(fallbackMethod = "myFallbackMethod")

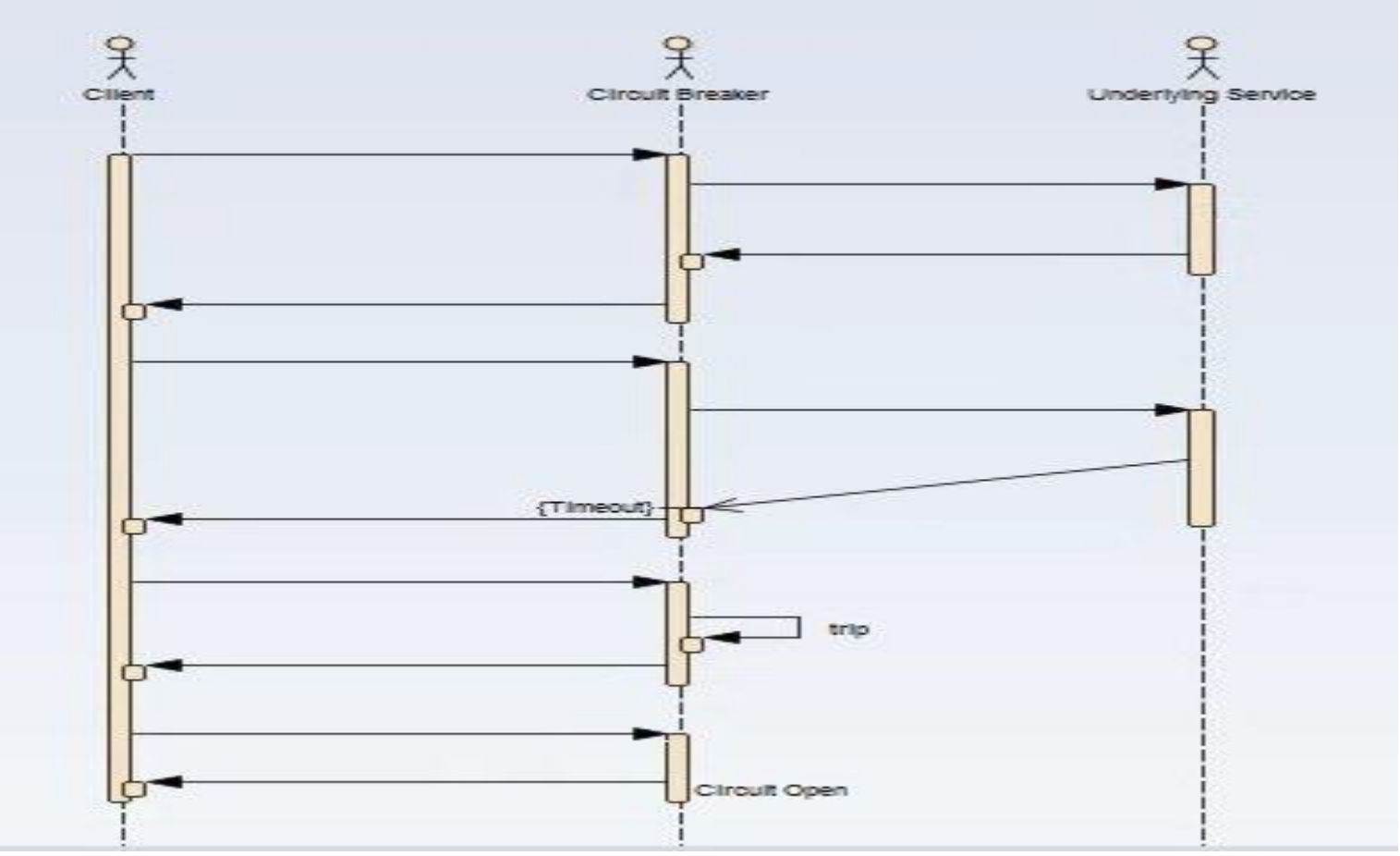
What is Circuit Breaker Pattern?

- **If we design our systems on microservice based architecture, we will generally develop many Microservices and those will interact with each other heavily in achieving certain business goals. Now, all of us can assume that this will give expected result if all the services are up and running and response time of each service is satisfactory.**

- **Now what will happen if any service, of the current Eco system, has some issue and stopped servicing the requests. It will result in timeouts/exception and the whole Eco system will get unstable due to this single point of failure.**

- **Here circuit breaker pattern comes handy and it redirects traffic to a fall back path once it sees any such scenario. Also it monitors the defective service closely and restore the traffic once the service came back to normalcy.**

- **So circuit breaker is a kind of a wrapper of the method which is doing the service call and it monitors the service health and once it gets some issue, the circuit breaker trips and all further calls goto the circuit breaker fall back and finally restores automatically once the service came back !!**


Client
Circuit Breaker
Underlying Service
{Timeout}
trip
Circuit Open

Hystrix Circuit Breaker Example

- **To demo circuit breaker, we will create following two microservices where first is dependent on another.**

- **Student Microservice – Which will give some basic functionality on Student entity. It will be a REST based service. We will call this service from School Service to understand Circuit Breaker. It will run on port 8098 in localhost.**
- **School Microservice – a simple REST based microservice where we will implement circuit breaker using Hystrix. Student Service will be invoked from here and we will test the fall back path once student service will be unavailable. It will run on port 9098 in localhost.**

- **Create Student Service**
- **Follow these steps to create and run Student Service – a simple REST service providing some basic functionality of Student entity.**

- **Create spring boot project**
- **with three dependencies i.e. Web, Rest Repositories and Actuator.**

- **Server Port Settings**
- **Open application.properties and add port information.**

- **server.port = 8098**
- **This will enable this application run on default port 8098.**

Create REST APIs

add one REST controller class called StudentServiceController and expose one rest endpoint for getting all the student details for a particular school. Here we are exposing /studentForSchool/{schoolname} endpoint to serve the business purpose. For simplicity, we are hard coding the student details.

- StudentServiceController.java

- package com.demo.controller;
-
- import java.util.ArrayList;
- import java.util.HashMap;
- import java.util.List;
- import java.util.Map;
- import org.springframework.web.bind.annotation.PathVariable;
- import org.springframework.web.bind.annotation.RequestMapping;
- import org.springframework.web.bind.annotation.RequestMethod;
- import org.springframework.web.bind.annotation.RestController;
- import com.demo.model.Student;
-

```
@RestController
public class StudentServiceController {

    private static Map<String, List<Student>> schooDB = new
HashMap<String, List<Student>>();

    static {
```

```
schooDB = new HashMap<String, List<Student>>();

    List<Student> lst = new ArrayList<Student>();
    Student std = new Student("Sajal", "Class IV");
    lst.add(std);
    std = new Student("Lokesh", "Class V");
    lst.add(std);

    schooDB.put("abcschool", lst);

    lst = new ArrayList<Student>();
    std = new Student("Kajal", "Class III");
    lst.add(std);
    std = new Student("Sukesh", "Class VI");
    lst.add(std);

    schooDB.put("xyzschool", lst);

}
```

```
@RequestMapping(value = "/StudentForSchool/{schoolname}", method =
RequestMethod.GET)
   public List<Student> getStudents(@PathVariable String schoolname) {
      System.out.println("Getting Student details for " + schoolname);

      List<Student> studentList = schooDB.get(schoolname);
      if (studentList == null) {
         studentList = new ArrayList<Student>();
         Student std = new Student("Not Found", "N/A");
         studentList.add(std);
      }
      return studentList;
   }
}
```

Student.java

```
package com.demo.model;

public class Student {

    private String name;
    private String className;

    public Student(String name, String className) {
        super();
        this.name = name;
        this.className = className;
    }
```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getClassName() {
        return className;
    }

    public void setClassName(String className) {
        this.className = className;
    }
}
```

- **Build and Test Student Service**

Create School Service – Hystrix Enabled

- **Similar to Student service, create another microservice for School. It will internally invoke already developed Student Service.**

- **Create a Spring boot project**

- **Web – REST Endpoints**
- **Actuator – providing basic management URL**
- **Hystrix – Enable Circuit Breaker**
- **Hystrix Dashboard – Enable one Dashboard screen related to the Circuit Breaker monitoring**

- **Server Port Settings**
- **application.properties and add port information.**

- **server.port = 9098**
- **This will enable this application run on default port 9098.**

Enable Hystrix Settings

Open SpringHystrixSchoolServiceApplication i.e the generated class with @SpringBootApplication and add @EnableHystrixDashboard and @EnableCircuitBreaker annotations.

This will enable Hystrix circuit breaker in the application and also will add one useful dashboard running on localhost provided by Hystrix.

```
package com.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
import
org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;
```

```
@SpringBootApplication
@EnableHystrixDashboard
@EnableCircuitBreaker
public class SpringHystrixSchoolServiceApplication {

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

Add REST controller

Add SchoolServiceController Rest Controller where we will expose /SchoolDetails/{schoolname} endpoint which will simply return school details along with its student details. For Student Details it will call the already developed Student service endpoint. We will create a Delegate layer StudentServiceDelegate.java to call the Student Service. This simple Code will look like

SchoolServiceController.java

```
package com.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import com.demo.delegate.StudentServiceDelegate;
```

```
@RestController
public class SchoolServiceController {

   @Autowired
   StudentServiceDelegate studentServiceDelegate;

   @RequestMapping(value = "/SchoolDetails/{schoolname}", method =
RequestMethod.GET)
   public String getStudents(@PathVariable String schoolname) {
      System.out.println("Going to call student service to get data!");
      return
studentServiceDelegate.callStudentServiceAndGetData(schoolname);
   }
}
```

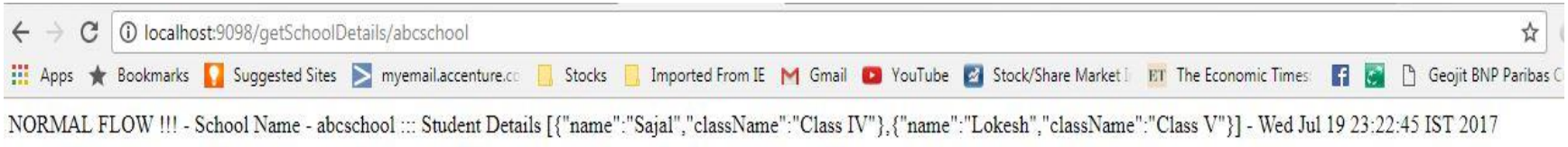
- StudentServiceDelegate
- do the following things here to enable Hystrix circuit breaker.
- Invoke Student Service through spring framework provided RestTemplate
- Add Hystrix Command to enable fallback method – @HystrixCommand(fallbackMethod = "callStudentServiceAndGetData_Fallback") –
- this means that we will have to add another method callStudentServiceAndGetData_Fallback with same signature, which will be invoked when actual Student service will be down.
- Add fallback method – callStudentServiceAndGetData_Fallback which will simply return some default value.

```
package com.demo.service;

import java.util.Date;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;

@Service
public class StudentServiceDelegate {
```

```
@Autowired
   RestTemplate restTemplate;

   @HystrixCommand(fallbackMethod =
"callStudentServiceAndGetData_Fallback")
   public String callStudentServiceAndGetData(String schoolname) {

      System.out.println("Getting School details for " + schoolname);

      String response = restTemplate
            .exchange("http://localhost:8098/StudentForSchool/{schoolname}"
            , HttpMethod.GET
            , null
            , new ParameterizedTypeReference<String>() {
         }, schoolname).getBody();
```

```
System.out.println("Response Received as " + response + " -  " + new Date());

    return "NORMAL FLOW !!! - School Name -  " + schoolname + " :::  " +
            " Student Details " + response + " -  " + new Date();
  }

  @SuppressWarnings("unused")
  private String callStudentServiceAndGetData_Fallback(String schoolname) {

    System.out.println("Student Service is down!!! fallback route enabled...");

    return "CIRCUIT BREAKER ENABLED!!! No Response From Student Service at this moment.
" +
            " Service will be back shortly - " + new Date();
  }

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

- **Build and Test of School Service**


localhost:9098/getSchoolDetails/abcschool
Apps
Bookmarks
Suggested Sites
myemail.accenture.co
Stocks
Imported From IE
Gmail
YouTube
Stock/Share Market
The Economic Times
Geojit BNP Paribas
NORMAL FLOW !!! - School Name - abcschool ::: Student Details [{"name":"Sajal","className":"Class IV"},{"name":"Lokesh","className":"Class V"}] - Wed Jul 19 23:22:45 IST 2017

- **Test Hystrix Circuit Breaker – Demo**
- **Opening browser and type http://localhost:9098/SchoolDetails/abcschool.**

- **It should show the list of students in abcschool**

- Now we already know that School service is calling student service internally, and it is getting student details from that service. So if both the services are running, school service is displaying the data returned by student service as we have seen in the school service browser output above. This is CIRCUIT CLOSED State.

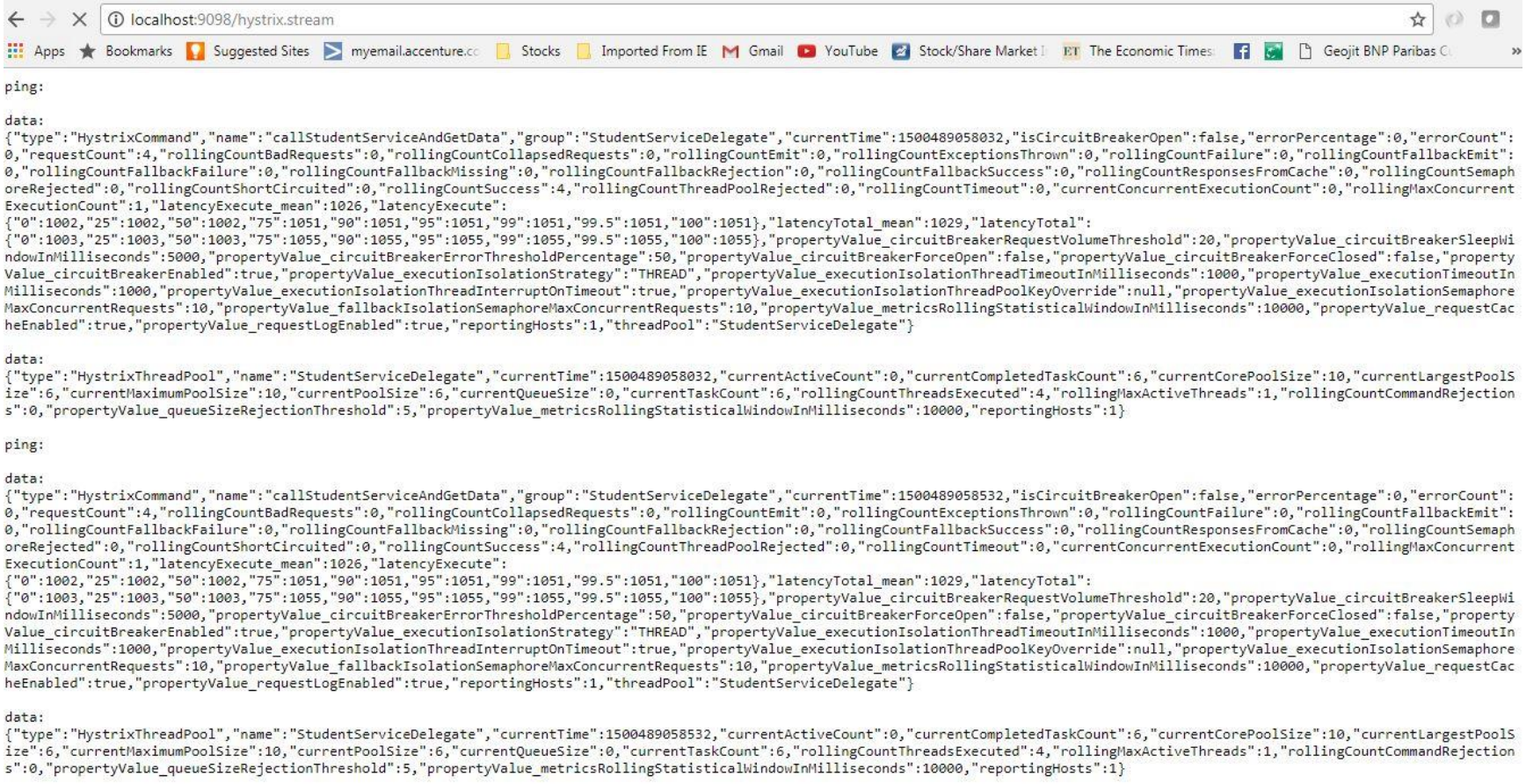
- Now let us stop the student service and test the school service again from browser. This time it will return the fall back method response. Here Hystrix comes into picture, it monitors Student service in frequent interval and as it is down, Hystrix component has opened the Circuit and fallback path enabled.

- Here is the fall back output in the browser.

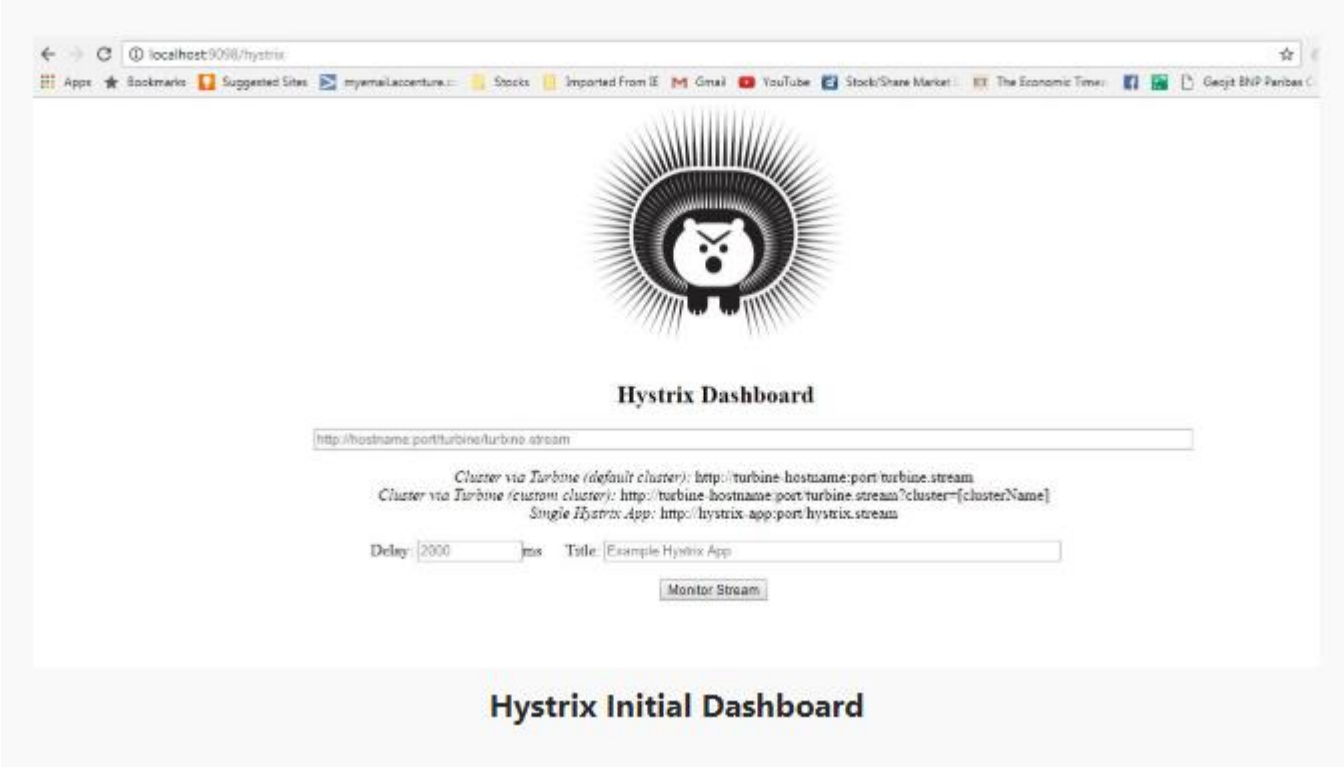
- Restart the Student service, wait for few moments and go back to school service and it will again start responding in normal flow.

- Hystrix Dashboard
- As we have added hystrix dashboard dependency, hystrix has provided one nice Dashboard and a Hystrix Stream in the bellow URLS:

- http://localhost:9098/hystrix.stream – It's a continuous stream that Hystrix generates. It is just a health check result along with all the service calls that are being monitored by Hystrix. Sample output will look like in browser –

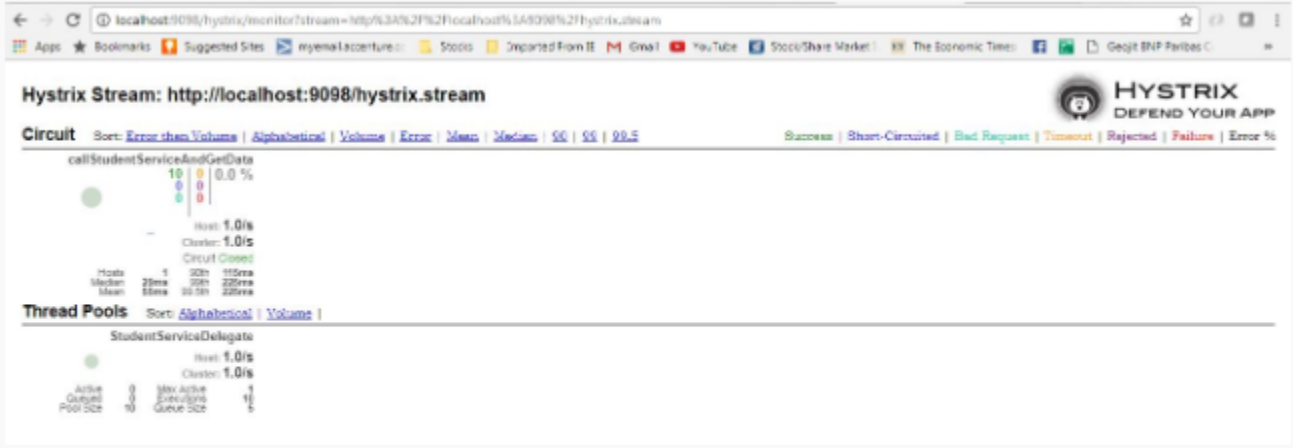

localhost:9098/hystrix.stream
Apps
Bookmarks
Suggested Sites
myemail.accenture.cc
Stocks
Imported From IE
Gmail
YouTube
Stock/Share Market I
The Economic Times
Geojit BNP Paribas C
ping:
data:
{"type":"HystrixCommand","name":"callStudentServiceAndGetData","group":"StudentServiceDelegate","currentTime":1500489058032,"isCircuitBreakerOpen":false,"errorPercentage":0,"errorCount":0,"requestCount":4,"rollingCountBadRequests":0,"rollingCountCollapsedRequests":0,"rollingCountEmit":0,"rollingCountExceptionsThrown":0,"rollingCountFailure":0,"rollingCountFallbackEmit":0,"rollingCountFallbackFailure":0,"rollingCountFallbackMissing":0,"rollingCountFallbackRejection":0,"rollingCountFallbackSuccess":0,"rollingCountResponsesFromCache":0,"rollingCountSemaphoreRejected":0,"rollingCountShortCircuited":0,"rollingCountSuccess":4,"rollingCountThreadPoolRejected":0,"rollingCountTimeout":0,"currentConcurrentExecutionCount":0,"rollingMaxConcurrentExecutionCount":1,"latencyExecute_mean":1026,"latencyExecute":
{"0":1002,"25":1002,"50":1002,"75":1051,"90":1051,"95":1051,"99":1051,"99.5":1051,"100":1051},"latencyTotal_mean":1029,"latencyTotal":
{"0":1003,"25":1003,"50":1003,"75":1055,"90":1055,"95":1055,"99":1055,"99.5":1055,"100":1055},"propertyValue_circuitBreakerRequestVolumeThreshold":20,"propertyValue_circuitBreakerSleepWindowInMilliseconds":5000,"propertyValue_circuitBreakerErrorThresholdPercentage":50,"propertyValue_circuitBreakerForceOpen":false,"propertyValue_circuitBreakerForceClosed":false,"propertyValue_circuitBreakerEnabled":true,"propertyValue_executionIsolationStrategy":"THREAD","propertyValue_executionIsolationThreadTimeoutInMilliseconds":1000,"propertyValue_executionTimeoutInMilliseconds":1000,"propertyValue_executionIsolationThreadInterruptOnTimeout":true,"propertyValue_executionIsolationThreadPoolKeyOverride":null,"propertyValue_executionIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_fallbackIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"propertyValue_requestCacheEnabled":true,"propertyValue_requestLogEnabled":true,"reportingHosts":1,"threadPool":"StudentServiceDelegate"}
data:
{"type":"HystrixThreadPool","name":"StudentServiceDelegate","currentTime":1500489058032,"currentActiveCount":0,"currentCompletedTaskCount":6,"currentCorePoolSize":10,"currentLargestPoolSize":6,"currentMaximumPoolSize":10,"currentPoolSize":6,"currentQueueSize":0,"currentTaskCount":6,"rollingCountThreadsExecuted":4,"rollingMaxActiveThreads":1,"rollingCountCommandRejections":0,"propertyValue_queueSizeRejectionThreshold":5,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"reportingHosts":1}
ping:
data:
{"type":"HystrixCommand","name":"callStudentServiceAndGetData","group":"StudentServiceDelegate","currentTime":1500489058532,"isCircuitBreakerOpen":false,"errorPercentage":0,"errorCount":0,"requestCount":4,"rollingCountBadRequests":0,"rollingCountCollapsedRequests":0,"rollingCountEmit":0,"rollingCountExceptionsThrown":0,"rollingCountFailure":0,"rollingCountFallbackEmit":0,"rollingCountFallbackFailure":0,"rollingCountFallbackMissing":0,"rollingCountFallbackRejection":0,"rollingCountFallbackSuccess":0,"rollingCountResponsesFromCache":0,"rollingCountSemaphoreRejected":0,"rollingCountShortCircuited":0,"rollingCountSuccess":4,"rollingCountThreadPoolRejected":0,"rollingCountTimeout":0,"currentConcurrentExecutionCount":0,"rollingMaxConcurrentExecutionCount":1,"latencyExecute_mean":1026,"latencyExecute":
{"0":1002,"25":1002,"50":1002,"75":1051,"90":1051,"95":1051,"99":1051,"99.5":1051,"100":1051},"latencyTotal_mean":1029,"latencyTotal":
{"0":1003,"25":1003,"50":1003,"75":1055,"90":1055,"95":1055,"99":1055,"99.5":1055,"100":1055},"propertyValue_circuitBreakerRequestVolumeThreshold":20,"propertyValue_circuitBreakerSleepWindowInMilliseconds":5000,"propertyValue_circuitBreakerErrorThresholdPercentage":50,"propertyValue_circuitBreakerForceOpen":false,"propertyValue_circuitBreakerForceClosed":false,"propertyValue_circuitBreakerEnabled":true,"propertyValue_executionIsolationStrategy":"THREAD","propertyValue_executionIsolationThreadTimeoutInMilliseconds":1000,"propertyValue_executionTimeoutInMilliseconds":1000,"propertyValue_executionIsolationThreadInterruptOnTimeout":true,"propertyValue_executionIsolationThreadPoolKeyOverride":null,"propertyValue_executionIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_fallbackIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"propertyValue_requestCacheEnabled":true,"propertyValue_requestLogEnabled":true,"reportingHosts":1,"threadPool":"StudentServiceDelegate"}
data:
{"type":"HystrixThreadPool","name":"StudentServiceDelegate","currentTime":1500489058532,"currentActiveCount":0,"currentCompletedTaskCount":6,"currentCorePoolSize":10,"currentLargestPoolSize":6,"currentMaximumPoolSize":10,"currentPoolSize":6,"currentQueueSize":0,"currentTaskCount":6,"rollingCountThreadsExecuted":4,"rollingMaxActiveThreads":1,"rollingCountCommandRejections":0,"propertyValue_queueSizeRejectionThreshold":5,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"reportingHosts":1}

- **http://localhost:9098/hystrix – This is visual dashboard initial state**



Hystrix Initial Dashboard

- **Now add http://localhost:9098/hystrix.stream in dashboard to get a meaningful dynamic visual representation of the circuit being monitored by the Hystrix component. Visual Dashboard after providing the Stream input in the home page –**



Hystrix visual Dashboard