What is monolithic?

A monolithic architecture is the traditional unified model for the design of a software program. **Monolithic**, in this context, means **composed all in one piece.**

Monolithic Advantages

* **Simple to develop.**
* **Simple to test.** For example, you can implement end-to-end testing by simply launching the application and testing the UI with Selenium.
* **Simple to deploy.** You just have to copy the packaged application to a server.
* **Simple to scale horizontally** by running multiple copies behind a load balancer.

Monolithic Disadvantages

* This **simple** **approach** has a limitation in size and complexity.
* **Application** **is** **too** **large** and complex to fully understand and made changes fast and correctly.
* The size of the application **can** **slow** **down** the start-up time.
* **You** **must** **redeploy** the entire application on each update.
* **Impact** **of** **a change** is usually not very well understood which leads to do extensive manual testing.
* **Continuous** **deployment** is difficult.
* Monolithic applications can also be difficult to scale when different modules have **conflicting** **resource** **requirements**.
* Another problem with **monolithic** applications is **reliability**. Bug in any module (e.g., memory leak) can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug will impact the availability of the entire application.
* Monolithic applications have a **barrier** **to adopting** **new** **technologies**. Since changes in frameworks or languages will affect an entire application, it is extremely expensive in both time and cost.

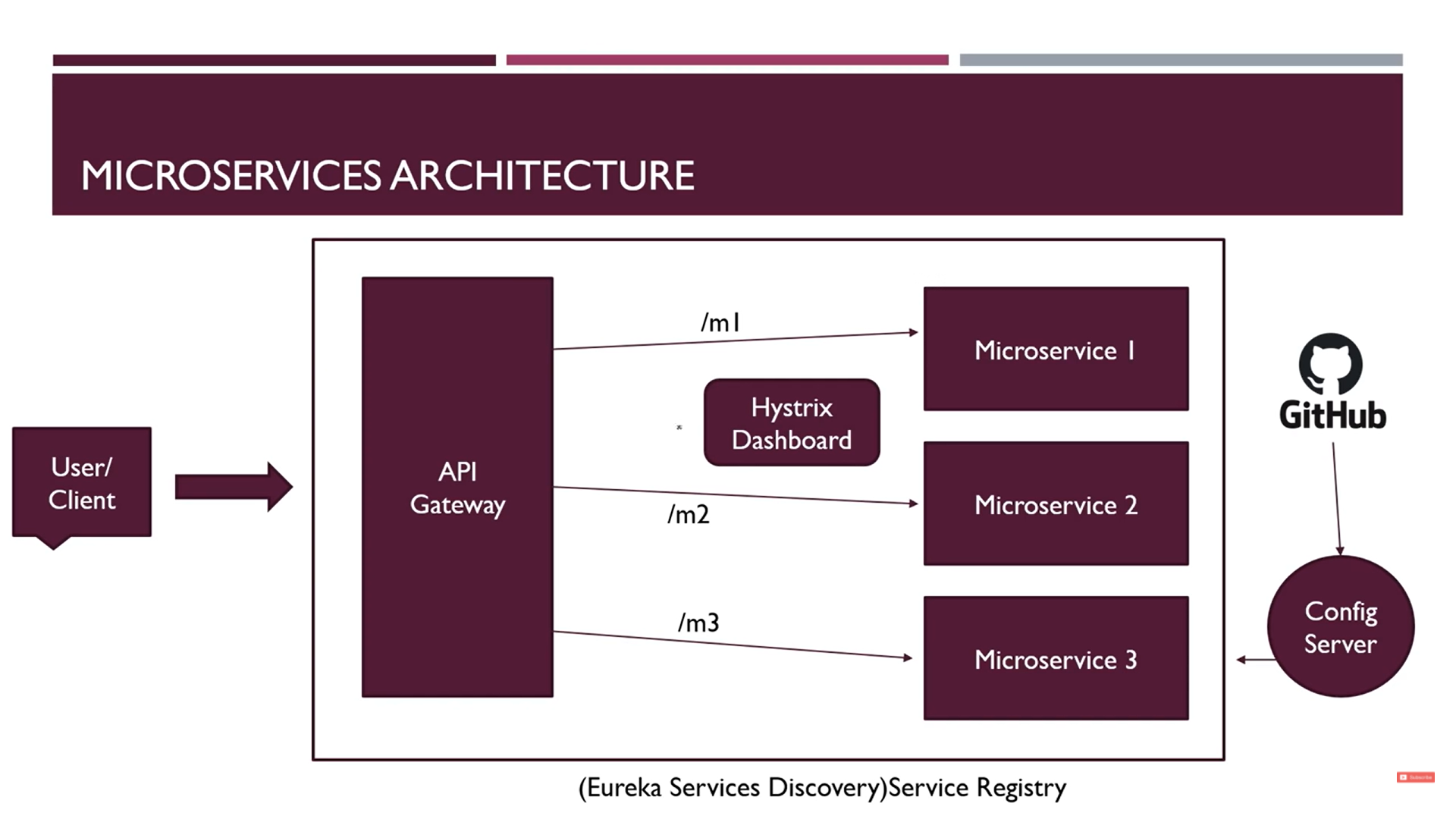
MicroservicesAdvantages

* **It tackles the problem of complexity** by decomposing application into a set of manageable services which are much faster to develop, and much easier to understand and maintain.
* It enables each service to be **developed independently** by a team that is focused on that service.
* It **reduces** barrier of **adopting new technologies** since the developers are free to choose whatever technologies make sense for their service and not bounded to the choices made at the start of the project.
* Microservice architecture enables each microservice to be **deployed** **independently**. As a result, it makes continuous deployment possible for complex applications.
* Microservice architecture enables each service to be **scaled** **independently**.

Microservices**Dis**advantages

* **Needs** **more** **collaboration** (each team has to cover the whole microservice lifecycle)
* **Harder** **to** **test** and [monitor](https://raygun.com/blog/monitoring-microservices/) because of the complexity of the architecture
* **Poorer** **performance**, as microservices need to communicate (network latency, message processing, etc.)
* **Harder to maintain the network** (has less fault tolerance, needs more load balancing, etc.)

Microservices**Architecture**



**API Gateway:** Request will not directly come M1, M2 or M3, first it will come to API Gateway and it will forward request to particular microservice. **All** the requests made by the client go through the API Gateway. After that, the API Gateway routes requests to the **appropriate microservice**. ***Disadvantage is There is a possibility of a single point of failure if API Gateway down.***

To implement this, we can use library like **Zuul, Spring Cloud**

**Hystrix Dashboard:** If one of the microservice failed then how to handle error that it will take care.

**Eureka Service discovery:** Eureka Server is an **application that holds the information** about all client-service applications. Every Micro service will register into the Eureka server and Eureka server knows all the client applications running on each **port** and **IP** address. Eureka Server is also known as **Discovery Server**

**Each and every module is a Spring boot project.**

***Example***

We will create 2 service **User-Service** and **Customer-Service, User** will internally call **Customer** using **RestTemplate**

**Go to Spring-initializr and create two projects with Spring-web dependency**

**User-Service**

**It will have Users.java and Contact.java**

**One User can have multiple contact**

Integer userId;

String userName;

List<Contact> contacts 🡪 cid, cName

**UserController.java**

@RequestMapping("/user") **above controller**

@Autowired

RestTemplate restTemplate; **// Create Bean in main method class**

@GetMapping(value = "getUsers/{userId}")

**public** User getUsers(@PathVariable Integer userId) {

List<User> users = Arrays.*asList*(

**new** User(1, "Sam"),

**new** User(2, "Pam"),

**new** User(3, "Dam"));

User user = users.stream().filter(x -> x.userId.equals(userId)).findAny().orElse(**null**);

***// Create Contact-service for this***

ResponseEntity<Contact[]> responseEntity = **this**.restTemplate.getForEntity("http://localhost:9002/getContacts/"+userId, Contact[].**class**);

user.setContacts(Arrays.*asList*(responseEntity.getBody()));

**return** user;

**Contact-Service**

**It will return List of Contact based on userId == cid from ContactController**

@RequestMapping("/contact") **above controller**

**@GetMapping**(value = "getContacts/{cid}")

**public** List<Contact> getContacts(@PathVariable Integer cid) {

List<Contact> contacts = Arrays.*asList*(**new** Contact(1, "Sam@gmail.com"),

**new** Contact(1, "Pam@gmail.com"),

**new** Contact(2, "Dam@gmail.com"));

**return** contacts.stream().filter(x -> x.cid.equals(cid)).collect(Collectors.*toList*());

**Now**

**We can access user-service, contact-service separately**

**Call user-service internally it will call contact-service and we will get list of Contact with users**

**Now**

We will create Eureka server and register our services over there

Go to spring.io and create one project **with dependency Eureka Server (project name: e-server)**

**e-server (Spring boot Project)**

Inside main method class below **@SpringBootApplication** add **@EnableEurekaServer**

**application.properties**

server.port=8761

**Here we're configuring an application port; the default one for***Eureka***servers is***8761*

eureka.client.registerWithEureka=false

**We're telling the built-in *Eureka Client* not to register with itself because our application should be acting as a server.**

eureka.client.fetchRegistry=false

***fetchRegistry controls whether or not this client is going to try to connect to the Eureka server(s) in order to download the information on other services' endpoint. It can do so without registering itself.***

eureka.server.waitTimeInMsWhenSyncEmpty=0

***How long should it wait to start registering another service***

**Now**

Open localhost:9761 we will get some predefined UI.

**Now** we will register our user-service and contact-service inside eureka server

**Now**

Go to Spring.io Search **Eureka Discovery Client dependency, below 3 buttons click Explore**

User-service pom.xml

<properties>

<java.version>11</java.version>

**<spring-cloud.version>2021.0.2</spring-cloud.version>**

</properties>

<**dependency**>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</**dependency**>

**Above build tag outside dependencies tag**

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

**Restart** user-service then refresh eureka UI we can see our **user-service** got registered over there.

***We first need to start eureka server before starting any service***

**Now** we can see in UI 2 Application is registered but the Application name is UNKNOWN we can change it

User-service application.properties

spring.application.name=user-service

***Same for contact-service also***

Restart all the 3 server and check in Eureka UI

Directly use service name instead localhost?

In our user-service we are calling contact-service like this <http://localhost:9002/getContacts/> change localhost:9002 to contact-service

UserServiceApplication.java

Above **RestTemplate** method add **@LoadBalanced**

**@LoadBalanced** above localhost we have changed with service name @LoadBalanced will do service discovery (search that service) with the help of eureka server with just service name, **it’s also doing client-side load balancing using algorithm ROUND-ROBIN**

**Restart all the 3 server and call getUsers API from browser**

**API-Gateway:** Till the time we have created two services and registered both of them inside **Eureka server**.

But if **client** want to access **getUsers** they directly contacting with **user-service** and for **getContacts** **contact-service**

There should be **only** **one** **endpoint** for client and that endpoint will internally call respective **service**

**Now**

Go to spring.io and create one project **(apigateway)** with dependencies

***Eureka Discovery client***

***Spring Boot Actuator***

***Gateway***

Apigateway **application.yml**

server:

port: 9003

spring:

application:

name: api-gateway

cloud:

gateway:

routes:

**- id: user-service**

**uri: lb://user-service**

**predicates:**

**- Path=/user/\*\* //Any request after /user will go to user-service**

*- id: contact-service*

*uri: lb://contact-service*

*predicates:*

*- Path=/contact/\*\**

**Restart all the service but first Eureka server and check in UI apigateway got registered**

**Now**

**Try to call getUsers using localhost:9003/user/getUsers/1**

**Note** we calling **API gateway** project and that’s internally calling **user-service**

**If Errors: Its spring.cloud.gateway.routes**

**Service Failure**

We have multiple microservice how can we handle if any of the microservice goes down or failed

**Apigateway FallBackController**

@GetMapping(value = "userServiceFallBack")

**public** String userServiceFallBack() {

**return** "User service is down";

}

@GetMapping(value = "contactServiceFallBack")

**public** String contactServiceFallBack() {

**return** "Contact service is down";

}

**Apigateway pom.xml**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-circuitbreaker-reactor-resilience4j</artifactId>

</dependency>

Apigateway **application.yml**

spring:

application:

name: api-gateway

cloud:

gateway:

routes:

- id: user-service

uri: lb://user-service

predicates:

- Path=/user/\*\*

**filters:**

**- name: CircuitBreaker**

**args:**

**name: user-service**

**fallbackuri: forward:/userServiceFallBack**

- id: contact-service

uri: lb://contact-service

predicates:

- Path=/contact/\*\*

**filters:**

**- name: CircuitBreaker**

**args:**

**name: contact-service**

**fallbackuri: forward:/contactServiceFallBack**

Now stop **user-service** and try to call user-service API via API-Gateway

**Now**

We have successfully created Eureka Server, Services and API gateway now we will Hystrix library

**Spring Cloud config server**

* 1. We have multiple microservice and each microservice we have separate ***application.properties*** files, So if we have change any value in particular ***application.properties*** file we have to change it in all the microservice ***application.properties*** file. In order to overcome this issue we can create one centralized ***application.properties*** file and all microservice will read it from there.
  2. **We can keep that file to** Git, SVN, File System (locally), JDBC, Redis, AWS S3 Etc.
  3. **No need to restart/redeploy** microservice after change in centralized ***application.properties*** file

**Example**

**Here we are using Git to store** centralized ***application.properties*** file

**GitHub.com**

1. Create one repository with name **config-server** (any name can take)
2. Inside repository create one **application.yml** file and add key **message: Hello world**
3. Inside repository create one **application-dev.yml** file and add key **message: Hello world from Dev**

**Config-server (New project)**

1. Create one new project name **config-server** with dependency **Eureka Discovery client** and **config server**
2. **ConfigServerApplication.java** add **@EnableConfigServer**
3. Create one **application.yml** and add

server:

port: 9005

spring:

application:

name: CONFIG-SERVER

cloud:

config:

server:

git:

uri: https://github.com/shahnawaz1711/config-server

searchPaths: configuration

**URI**: if we have Git repo in our local PC like **c:/Desktop/MyFolder** then we can pass

uri:file://c:/Desktop/MyFolder we need to put both the file in this folder

**Now** open URL In browser we can see our properties which we have define in Git yml file

<http://localhost:9005/config-server/default>

**Now** we will read message property which we have defined in Git in our services

**User-service**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

**User-service application.yml**

spring:

application:

name: user-service

config:

import: optional:configserver:**http://localhost:9005**

profiles:

active: dev

if we remove active: dev then it will consider **application.yml** otherwise **application-dev.yml**

[**http://localhost:9005**](http://localhost:9005) it’s a port of our config server application

**UserController.java**

@Value("${message}")

String message;

**@RefreshScope:** If we update message value in Git **application.yml** still it will fetch the old one to solve that issue follow below.

**UserController.java** add **@RefreshScope** on top

**User-service pom.xml**

<dependency>

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

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

</dependency>

**User-service application.yml**

management:

endpoints:

web:

exposure:

include: "\*"

"\*" means all endpoints we want to enable if particular we can **refresh**

**Now** Get the value of message through **user-service** after getting change it from **Git**

After changing again get **message** key from **user-service** we can see **message key value not change**

**To apply changes**

**POST** <http://localhost:9001/actuator/refresh>

Then again get **message** key from **user-service**

**This we need to do in all the services**

**Spring Cloud Resilience4J (Circuit Breaker)**

As of now we have implemented CB in API gateway but if I want to do in a service itself, we can do. As we already know if other service is down, it will call **fallback method** or do something else.

Now we will in depth of **Resilience4J** total concept we can look in it is

**Circuit-Breaker, Retry, Rate Limiter, Bulk-Head, Time Limiter, Cache**

As of now we will focus on **Circuit-Breaker, Retry, Rate Limiter** because it’s used a lot

**Circuit-Breaker:** Here we have 3 states **CLOSED, OPEN and HALF\_OPEN**

**CLOSED:** means all the services are up and running fine

**OPEN:** suppose out of 10 5 requests got failed (we need to define threshold like 5 or 6) then it will change state from **CLOSED** to **OPEN** and when state is **OPEN** no request will be allowed.

**HALF\_OPEN:** now our CB is an **OPEN** state so how long it needs to be in an **OPEN** state we need to define a time, so suppose we have defined 5 seconds so after 5 seconds state will change from **OPEN** to **HALF\_OPEN** then it will check request is working or not, so when state is **HALF\_OPEN** we need to define a threshold like out of 4 at-least 2 request must be success, if 2 request got success then it will change state to **HALF\_OPEN** to **CLOSED** otherwise again **OPEN** and same scenario will be executed

**Example:**

**User-service pom.xml**

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-circuitbreaker resilience4j </artifactId>

</dependency>

<dependency>

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

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

</dependency>

**User-service application.yml**

server:

port: 9001

spring:

application:

name: user-service

jackson:

serialization:

INDENT\_OUTPUT: **true // Format JSON response in Browser**

management:

health:

circuitbreakers:

enabled: **true // Enable circuit-breaker info in actuator/health URL**

endpoints:

web:

exposure:

include: health **// Enable actuator/health URL**

endpoint:

health:

show-details: always **// Allowing to show circuit-breaker and details on browser**

resilience4j:

circuitbreaker:

instances:

userService: // Name which we have defined in @CurcuitBreaker annotation must be same

registerHealthIndicator: **true** // Info related to CB want to see in actuator/health

eventConsumerBufferSize: 10 // count of request within defined time

failureRateThreshold: 50 // 50% request failed it will change state

minimumNumberOfCalls: 5 // When it’s in OPEN state

automaticTransitionFromOpenToHalfOpenEnabled: **true**

waitDurationInOpenState: 5s

permittedNumberOfCallsInHalfOpenState: 3

slidingWindowSize: 100

slidingWindowType: COUNT\_BASED // It can be TIME\_BASED also like for particular time if service not respond then it will change state

**UserController**

@RequestMapping(value = "getUsers/{userId}")

@CircuitBreaker(name = "userService", fallbackMethod = "fallBackMe")

**public** User getUsers(@PathVariable(name = "userId") Integer userId) {

**// Code for other service call here**

**return** user;

}

**public** User fallBackMe(Exception exception) {

**return** **new** User();

} **// Must take Exception as a parameter and return must be same as actual method**

**URL**: <http://localhost:9001/actuator/health> we can see all the details

**NOW**

Start both the service User and Contact and try to make some calls, **then check in actuator URL info will change like buffered call, failure rate and all**

**NOW**

Stop **contact** service and call **user** service 5 time because we have defined **eventConsumerBufferSize** is **10** and out of which 5 failed then it will reach **failureRateThreshold** which 50%

We can see circuit state got changed from **CLOSED** to **OPEN**

Then after **5s** it will auto become **OPEN** to **HALF\_OPEN**

In **HALF\_OPEN** we can try 3 requests if all 3 requests got failed then again it will go to **OPEN** state, **try to call user service,** After 5s again it will back to **HALF\_OPEN**

**NOW**

Start contact service and hit 3-time user service we can see state change from **HALF\_OPEN** to **CLOSED**

**Retry:** As we know we have 6 concepts in **Resilience4J** we have seen **Circuit-Breaker** now we will see **@Retry, its like we want to retry within some time gap**

**UserController**

@RequestMapping(value = "getUsers/{userId}")

//@CircuitBreaker(name = "userService", fallbackMethod = "fallBackMe")

@Retry(name = "userService", fallbackMethod = "fallBackMe")

**public** User getUsers(@PathVariable(name = "userId") Integer userId) {

**User-service application.yml**

**It will be under** resilience4J:

retry:

instances:

userService:

registerHealthIndicator: **true**

maxRetryAttempts: 5

waitDuration: 10s

**It will retry for 3 times and then it will call fallback method**

**RateLimiter:** Like in **10s** only **10** request is allowed for **11th** request it should wait for **3s** if in **3s** **11th request won’t get response** then it should call fallback method

**UserController**

@RequestMapping(value = "getUsers/{userId}")

//@CircuitBreaker(name = "userService", fallbackMethod = "fallBackMe")

//@Retry(name = "userService", fallbackMethod = "fallBackMe")

@RateLimiter(name = "userService", fallbackMethod = "fallBackMe")

**public** User getUsers(@PathVariable(name = "userId") Integer userId) {

**User-service application.yml**

**It will be under** resilience4J:

ratelimiter:

instances:

userService:

registerHealthIndicator: **false**

limitForPeriod: 10

limitRefreshPeriod: 10s

timeoutDuration: 3s

**UP both the service makes 11 requests in 10s will see fallback for 11th request because 11th request took more then 3s because it waiting for 10s to complete**

Which request took how much time we can see in browser > Network Bottom

**Spring Cloud Zipkin and Sleuth**

We have multiple microservice and each microservice having its separate log file but we want in a centralized form we can this

**Trace-Id and Span-Id**

We have single request from client and **user-service** internally call **contact-service** so here will be only **Trace-id** but multiple **span-id**

At initial when request come to first service **Trace-id** and **span-id** is same

**Example**

* 1. Go to zipkin.io > quickstart > Download jar - **zipkin-server-2.23.16-exec.jar**
  2. Put jar in a workspace **then** open CMD in workspace then **java -jar zipkin-server-2.23.16-exec.jar**
  3. Open [**http://127.0.0.1:9411/zipkin**](http://127.0.0.1:9411/zipkin)in browser
  4. Now in all the service add dependency

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-sleuth-zipkin</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

* 1. In all the service application.yml or properties

**spring.zipkin.base-url=http://127.0.0.1:9411/**

* 1. Add some logger also in controller or somewhere else in service
  2. Now in Zipkin search via service name we can see how the flow was gone through

**Hystrix**

* 1. Fault tolerance library
  2. Developed by Netflix and it’s a part of Spring via **Spring cloud Netflix project**

**Hystrix Dashboard**

* 1. Use to monitor our services in real time