Spring Microservices in Action

Saturday, August 03, 2019 11:44 PM

Spring Microservices in Action

p-38

- *Flexible*—Decoupled services can be composed and rearranged to quickly deliver new functionality. The smaller the unit of code that one is working with, the less complicated it is to change the code and the less time it takes to test deploy the code.
- *Resilient*—Decoupled services mean an application is no longer a single "ball of mud" where a degradation in one part of the application causes the whole application to fail. Failures can be localized to a small part of the application and contained before the entire application experiences an outage. This also enables the applications to degrade gracefully in case of an unrecoverable error.
- *Scalable*—Decoupled services can easily be distributed horizontally across multiple servers, making it possible to scale the features/services appropriately. With a monolithic application where all the logic for the application is intertwined, the entire application needs to scale even if only a small part of the application is the bottleneck. Scaling on small services is localized and much more cost-effective.

To this end, as we begin our discussion of microservices keep the following in mind:

Small, Simple, and Decoupled Services = Scalable, Resilient, and Flexible Applications

p-40

Emerging cloud platforms

I've documented the three core cloud platform types (IaaS, PaaS, SaaS) that are in use today. However, new cloud platform types are emerging. These new platforms include Functions as a Service (FaaS) and Container as a Service (CaaS). FaaS-based (https://en.wikipedia.org/wiki/Function_as_a_Service) applications use technologies like Amazon's Lambda technologies and Google Cloud functions to build applications deployed as "serverless" chunks of code that run completely on the cloud provider's platform computing infrastructure. With a FaaS platform, you don't have to manage any server infrastructure and only pay for the computing cycles required to execute the function.

With the Container as a Service (CaaS) model, developers build and deploy their microservices as portable virtual containers (such as Docker) to a cloud provider. Unlike an IaaS model, where you the developer have to manage the virtual machine the service is deployed to, with CaaS you're deploying your services in a lightweight virtual container. The cloud provider runs the virtual server the container is running on as well as the provider's comprehensive tools for building, deploying, monitoring, and scaling containers. Amazon's Elastic Container Service (ECS) is an example of a CaaS-based platform. In chapter 10 of this book, we'll see how to deploy the microservices you've built to Amazon ECS.

major cloud providers. A microservice can be packaged up in a virtual machine image and multiple instances of the service can then be quickly deployed and started in either a IaaS private or public cloud.

- *Virtual container*—Virtual containers are a natural extension of deploying your microservices on a virtual machine image. Rather than deploying a service to a full virtual machine, many developers deploy their services as Docker containers (or equivalent container technology) to the cloud. Virtual containers run inside a virtual machine; using a virtual container, you can segregate a single virtual machine into a series of self-contained processes that share the same virtual machine image.

1.9

Microservice Patterns
https://dzone.com/articles/design-patterns-for-microservices [Must Read]
https://microservices.io/patterns/microservices.html
https://dzone.com/articles/microservice-design-patterns

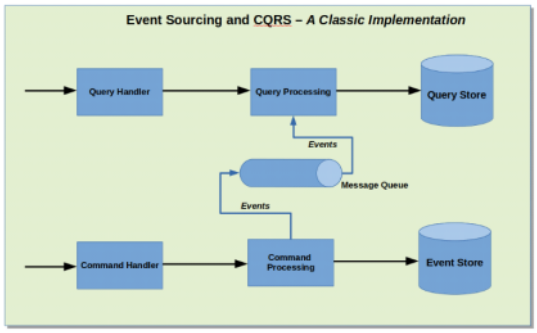
Command Query Responsibility Segregation (CQRS)

From <https://dzone.com/articles/design-patterns-for-microservices>

http://progressivecoder.com/event-sourcing-and-cqrs-with-axon-and-spring-boot-part-1/

Event Sourcing and CQRS – A Classic Implementation Approach

Event Sourcing and CQRS are basically two separate patterns serving a common use-case. Ideally, we should implement them as two separate applications. Below is a high-level view of how such an implementation can look like:



Some of the advantages of using Event Sourcing and CQRS are:

- You can scale up the command (or update) side separately than your query (or read) side. This could be a great advantage for a system where *reads* outnumber *writes* by a huge margin.
- You can chose different strategies for *event store* and *query store*. For example, event store can be a typical RDBMS. You can handle queries using NoSQL database (like MongoDB).
- Using Event Sourcing and CQRS together, you can basically get rid of data aggregation pattern.

- Core development patterns
- Routing patterns
- Client resiliency patterns
- Security patterns
- Logging and tracing patterns
- Build and deployment patterns

routing patterns
- Service routing
- Service discovery

resiliency patterns
- Client-side load balancing
- Circuit breakers pattern
- Fallback pattern
- Bulkhead pattern

security patterns
- Authentication
- Authorization
- Credential management and propagation

logging and tracing patterns
- Log correlation
- Log aggregation
- Microservice tracing

1.9.5 Microservice logging and tracing patterns

The beauty of the microservice architecture is that a monolithic application is broken down into small pieces of functionality that can be deployed independently of one another. The downside of a microservice architecture is that it's much more difficult to debug and trace what the heck is going on within your application and services.

For this reason, we'll look at three core logging and tracing patterns:

- *Log correlation*—How do you tie together all the logs produced between services for a single user transaction? With this pattern, we'll look at how to implement a correlation ID, which is a unique identifier that will be carried across all service calls in a transaction and can be used to tie together log entries produced from each service.
- *Log aggregation*—With this pattern we'll look at how to pull together all of the logs produced by your microservices (and their individual instances) into a single queryable database. We'll also look at how to use correlation IDs to assist in searching your aggregated logs.
- *Microservice tracing*—Finally, we'll explore how to visualize the flow of a client transaction across all the services involved and understand the performance characteristics of services involved in the transaction.

build/deployment patterns

- *Build and deployment pipeline*—How do you create a repeatable build and deployment process that emphasizes one-button builds and deployment to any environment in your organization?
- *Infrastructure as code*—How do you treat the provisioning of your services as code that can be executed and managed under source control?
- *Immutable servers*—Once a microservice image is created, how do you ensure that it's never changed after it has been deployed?
- *Phoenix servers*—The longer a server is running, the more opportunity for configuration drift. How do you ensure that servers that run microservices get torn down on a regular basis and recreated off an immutable image?

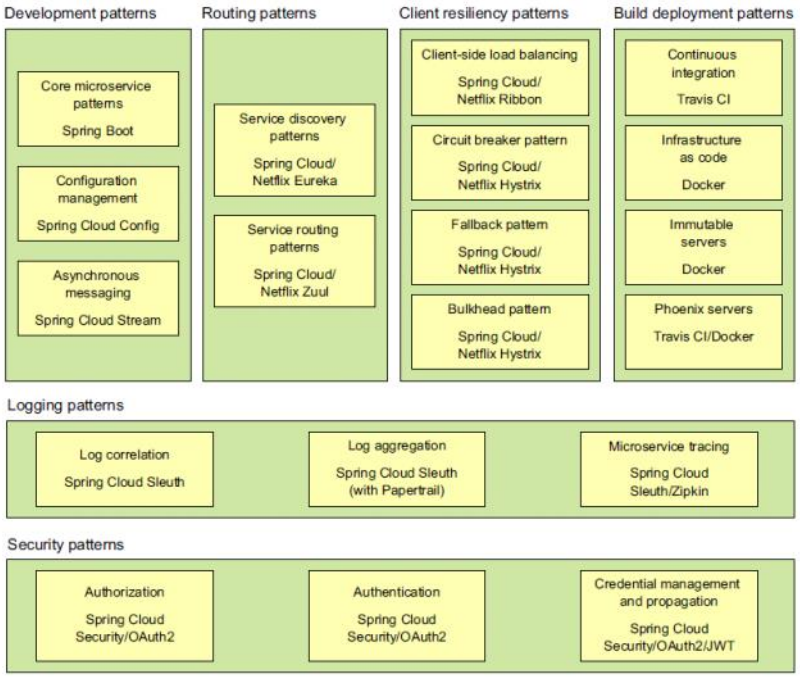
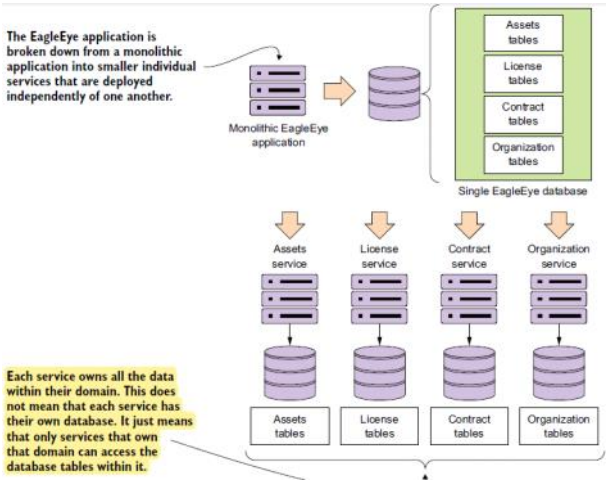


Figure 1.14 You can map the technologies you're going to use directly to the microservice patterns we've explored so far in this chapter.

1. *Describe the business problem, and listen to the nouns you're using to describe the problem.* Using the same nouns over and over in describing the problem is usually an indication of a core business domain and an opportunity for a microservice. Examples of target nouns for the EagleEye domain from chapter 1 might look something like *contracts*, *licenses*, and *assets*.
2. *Pay attention to the verbs.* Verbs highlight actions and often represent the natural contours of a problem domain. If you find yourself saying "transaction X needs



70 / 386 125%

Also keep in mind that no standard exists for performing transactions across microservices. If you need transaction management, you will need to build that logic yourself. In addition, as you'll see in chapter 7, microservices can communicate

74 / 386 125%

Map the behavior of the service to standard HTTP verbs—REST emphasizes having services map their behavior to the HTTP verbs of POST, GET, PUT, and DELETE verbs. These verbs map to the CRUD functions found in most services.

[a] Probably the most comprehensive coverage of the design of REST services is the book *REST in Practice* by Ian Robinson, et al (O'Reilly, 2010).

75 / 386 125%


```
@RestController
@RequestMapping(value="/v1/organizations/{organizationId}/licenses")
public class LicenseServiceController {
  //Body of the class removed for conciseness
}
```

@RestController tells Spring Boot this is a REST-based services and will automatically serialize/deserialize service request/response to JSON.

Exposes all the HTTP endpoints in this class with a prefix of /v1/organizations/{organizationId}/licenses

We'll begin our exploration by looking at the `@RestController` annotation. The `@RestController` is a class-level Java annotation and tells the Spring Container that this Java class is going to be used for a REST-based service. This annotation automatically handles the serialization of data passed into the services as JSON or XML (by default the `@RestController` class will serialize returned data into JSON). Unlike the traditional Spring `@Controller` annotation, the `@RestController` annotation doesn't require you as the developer to return a `ResponseBody` class from your controller class. This is all handled by the presence of the `@RestController` annotation, which includes the `@ResponseBody` annotation.

75 / 386 125%

Other mechanisms and protocols are more efficient than JSON for communicating between services. The Apache Thrift (http://thrift.apache.org) framework allows you to build multi-language services that can communicate with one another using a binary protocol. The Apache Avro protocol (http://avro.apache.org) is a data serialization protocol that converts data back and forth to a binary format between client and server calls.

80 / 386 125%

Building the Twelve-Factor microservice service application

One of my biggest hopes with this book is that you realize that a successful microservice architecture requires strong application development and DevOps practices. One of the most succinct summaries of these practices can be found in Heroku's Twelve-Factor Application manifesto (https://12factor.net/). This document provides 12 best practices you should always keep in the back of your mind when building microser-

Concurrency—When you need to scale, don't rely on a threading model within a single service. Instead, launch more microservice instances and scale out horizontally. This doesn't preclude using threading within your microservice, but don't rely on it as your sole mechanism for scaling. Scale out, not up.

Logs—Logs are a stream of events. As logs are written out, they should be streamable to tools, such as Splunk (http://splunk.com) or Fluentd (http://fluentd.org), that will collate the logs and write them to a central location. The microservice should never be concerned about the mechanics of how this happens and the developer should visually look at the logs via STDOUT as they're being written out.

Fortunately, almost all Java microservice frameworks will include a runtime engine that can be packaged and deployed with the code. For instance, in the Spring Boot example in figure 2.7, you can use Maven and Spring Boot to build an executable Java jar file that has an embedded Tomcat engine built right into the JAR. In the following command-line example, you're building the licensing service as an executable JAR and then starting the JAR file from the command-line:

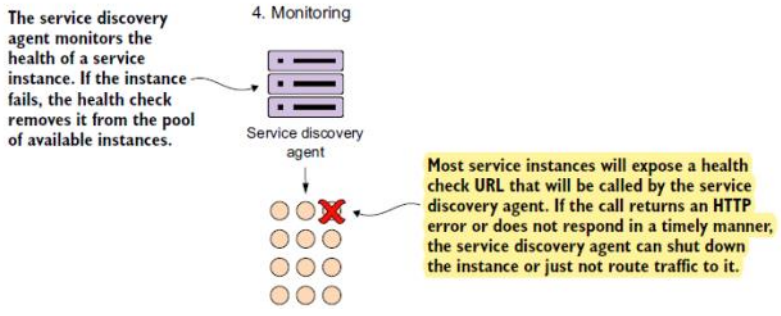
```
mvn clean package && java -jar target/licensing-service-0.0.1-SNAPSHOT.jar
```

For certain operation teams, the concept of embedding a runtime environment right in the JAR file is a major shift in the way they think about deploying applications. In a traditional J2EE enterprise organization, an application is deployed to an application server. This model implies that the application server is an entity in and of itself and would often be managed by a team of system administrators who managed the configuration of the servers independently of the applications being deployed to them.

This separation of the application server configuration from the application introduces failure points in the deployment process, because in many organizations the

configuration of the application servers isn't kept under source control and is managed through a combination of the user interface and home-grown management scripts. It's too easy for configuration drift to creep into the application server environment and suddenly cause what, on the surface, appear to be random outages.

see figure 2.9 for details on this process). When a microservice instance registers with a service discovery agent, it will tell the discovery agent two things: the physical IP address or domain address of the service instance, and a logical name that an application can use to look up in a service. Certain service discovery agents will also require a



After a microservice has come up, the service discovery agent will continue to monitor and ping the health check interface to ensure that that service is available. This is step 4 in figure 2.6. Figure 2.10 provides context for this step.

By building a consistent health check interface, you can use cloud-based monitoring tools to detect problems and respond to them appropriately.

If the service discovery agent discovers a problem with a service instance, it can take corrective action such as shutting down the ailing instance or bringing additional service instances up.

In a microservices environment that uses REST, the simplest way to build a health check interface is to expose an HTTP end-point that can return a JSON payload and HTTP status code. In a non-Spring-Boot-based microservice, it's often the developer's responsibility to write an endpoint that will return the health of the service.

In Spring Boot, exposing an endpoint is trivial and involves nothing more than modifying your Maven build file to include the Spring Actuator module. Spring Actuator provides out-of-the-box operational endpoints that will help you understand and manage the health of your service. To use Spring Actuator, you need to make sure you include the following dependencies in your Maven build file:

microservice candidates will emerge. Remember, too, that it's better to start with a "coarse-grained" microservice and refactor back to smaller services than to start with a large group of small services. Microservice architectures, like most good architectures, are emergent and not preplanned to-the-minute.

- Out of the box, Spring Boot allows you to deliver a service as a single executable JAR file. An embedded Tomcat server in the producer JAR file hosts the service.

Service Registry will do
- get the physical ip address
- monitor health using actuator

spring boot comes with embedded tomcat server
while deploying microservice, if it comes with built in runtime engine, it can be deployed easily on other environments. it w ill be less prone to errors.
Otherwise if you use external servers, that setup needs to be done explicitly which will attracts errors.

At p-89

Service Discovery - ch4
service routing - ch6

Chapter-8
p-254

service granularity
Routing - policy enforcement point for things like authorization, authentication, and content checking.
Event Driven Architecture (EDA).
database per service
shared database per service
you'll need to make an aggregation service that joins data together

Microservices Advantages
1. Different customers want different features, and they don't want to have to wait for a long application release cycle before they can start using these features.
2. Downtime is low, microservice-based applications can more easily isolate faults and problems to specific parts of an application without taking down the entire application.
3. Reusability

spring-cloud-starter-eureka [*@EnableEurekaServer, @EnableDiscoveryClient*]

spring-cloud-starter-config
spring-cloud-config-server

spring-cloud-config-client

spring-cloud-starter-hystrix *[@EnableCircuitBreaker, @HystrixCommand]*

Zuul Configuration
Create a new microservice for *zuul* setup. It will act as a *eureka client*

```
@SpringBootApplication
@EnableEurekaClient
@EnableZuulProxy
public class PhotoAppApiZuulApiGatewayApplication {

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

}
```

Give desired port and configure eureka server url

PhotoAppApiZuulApiGatewayApplication.java | application.properties

```
spring.application.name=zuul
server.port=8011
eureka.client.serviceUrl.defaultZone = http://localhost:8010/eureka
```

In *eureka server* dashboard which is running at 8010 will shows all the services.

Now to call *user* microservice, we need to give *zuul* microservice url instead of *user* using port 8011.
i.e., localhost:8011/users-ws/status/check
It will automatically redirect it to *user microservice*, may be internally zuul will check with eureka server.

Eureka
1
Client ---> zuul ---> User Microservice

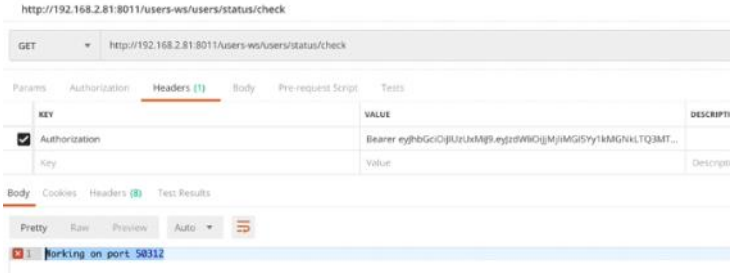
Spring Security
udemy - 84

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
```

User login get call will give token
use same token and call user data service, it will authenticate the token using secret key, if its valid, it will allow
Before every call to user data service, need to call *user login* service

```
private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest req) {
    String authorizationHeader = req.getHeader(environment.getProperty("authorization.token.header.name"));

    if (authorizationHeader == null) {
        return null;
    }

    String token = authorizationHeader.replace(environment.getProperty("authorization.token.header.prefix"), "");

    String userId = Jwts.parser()
            .setSigningKey(environment.getProperty("token.secret"))
            .parseClaimsJws(token)
            .getBody()
            .getSubject();

    if (userId == null) {
        return null;
    }

    return new UsernamePasswordAuthenticationToken(userId, null, new ArrayList<>());
}
```

In header, add a key *Authorization* and value *Bearer-[token got from user login service]*



In *user login* service, we create token like below
Here Secret key will be passed

```
@Override
protected void successfulAuthentication(HttpServletRequest req,
                                        HttpServletResponse res,
                                        FilterChain chain,
                                        Authentication auth) throws IOException, ServletException {
    String userName = ((User) auth.getPrincipal()).getUsername();
    UserDto userDetails = usersService.getUserDetailsByEmail(userName);

    String token = Jwts.builder()
            .setSubject(userDetails.getUserId())
            .setExpiration(new Date(System.currentTimeMillis() + Long.parseLong(environment.getProperty("token.expiratic
            .signWith(SignatureAlgorithm.HS512, environment.getProperty("token.secret") )
            .compact();

    res.addHeader("token", token);
    res.addHeader("userId", userDetails.getUserId());
}
```

Spring cloud Config Server using git repository
Add below dependencies in *config server* microservices

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
    </dependency>

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

<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>
```

```java
@SpringBootApplication
@EnableConfigServer
public class PhotoAppApiConfigServerApplication {

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

}
```

```
spring.application.name=PhotoAppAPIConfigServer
server.port=8012

spring.cloud.config.server.git.uri=https://github.com/simplyi/PhotoAppConfiguration
spring.cloud.config.server.git.username=simplyi
spring.cloud.config.server.git.password=
spring.cloud.config.server.git.clone-on-start=true
```

Below dependency needs to be added in *microservices clients*

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

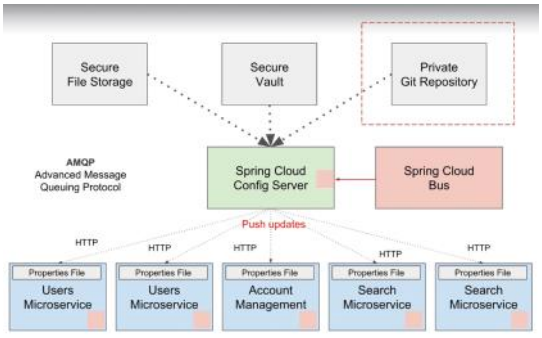
Run

- Discovery service
- Zuul apigateway
- config server
- our microservice

Config Server

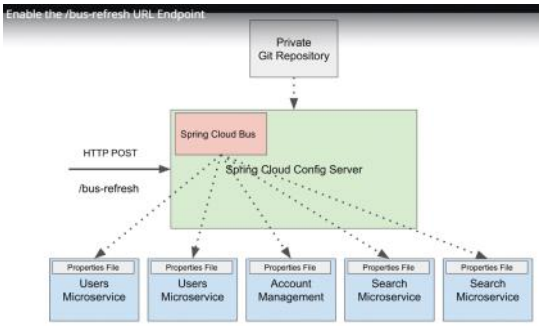
```
spring.application.name=PhotoAppAPIConfigServer
server.port=8012

spring.cloud.config.server.git.uri=https://github.com/simplyi/PhotoAppConfiguration
spring.cloud.config.server.git.username=simplyi
spring.cloud.config.server.git.password=
spring.cloud.config.server.git.clone-on-start=true
```

Spring Cloud Bus




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

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

Send post request to *git config server* with url /bus-refresh

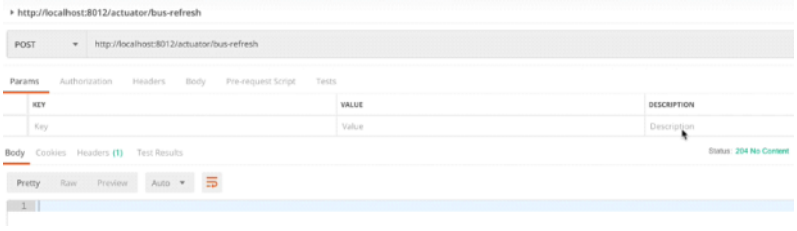



```
9
10 management.endpoints.web.exposure.include=bus-refresh
```

Configure *RabbitMq* in all configstore, client, zuul microservices

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

After changing the git store application.yml, to refresh and broadcast it to all microservices regarding the update need to fire *bus-refresh* post call
Now all the microservices will be use the latest value



20. Microservice Communication

1. Feign is an HTTP Client,
2. Declarative,
3. Load balanced.

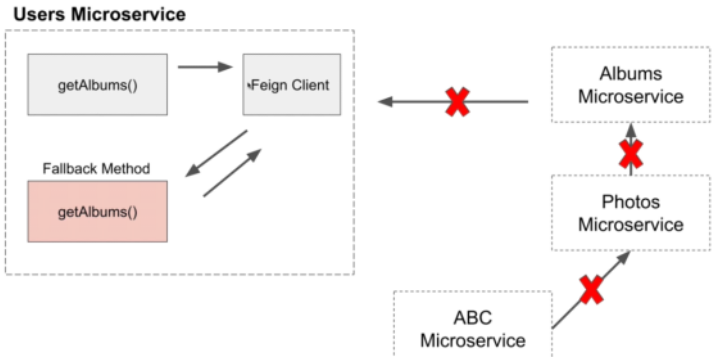
```
@FeignClient(name="PhotoAlbums", url="https://mywebsite.com")


public interface AlbumsServiceClient {

@GetMapping("/users/${id}/albums")
public List<AlbumResponseModel> getAlbums(@PathVariable String id);

}
```

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

```
import java.util.List;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

import com.appsdeveloperblog.photoapp.api.users.ui.model.AlbumResponseModel;

@FeignClient(name="albums-ws")
public interface AlbumsServiceClient {

    @GetMapping("/users/{id}/albums")
    public List<AlbumResponseModel> getAlbums(@PathVariable String id);
}
```

Hystrix and Circuit breaker

If Microservice is down, circuit is opened and it will call fallback method.
Once the microservice is available, it will close the circuit and call actual method in the working microservice.




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

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
@EnableCircuitBreaker
public class PhotoAppApiUsersApplication {
```

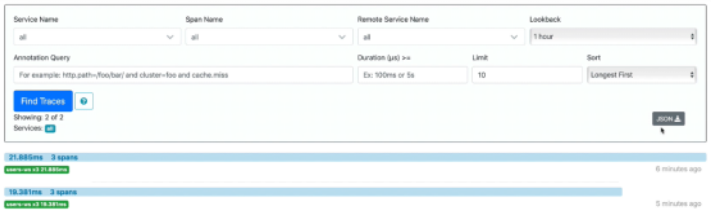
```
@FeignClient(name="albums-ws", fallback=AlbumsFallback.class)
public interface AlbumsServiceClient {

    @GetMapping("/users/{id}/albums")
    public List<AlbumResponseModel> getAlbums(@PathVariable String id);
}

@Component
class AlbumsFallback implements AlbumsServiceClient
{

    @Override
    public List<AlbumResponseModel> getAlbums(String id) {
        // TODO Auto-generated method stub
        return new ArrayList<>();
    }
}
```

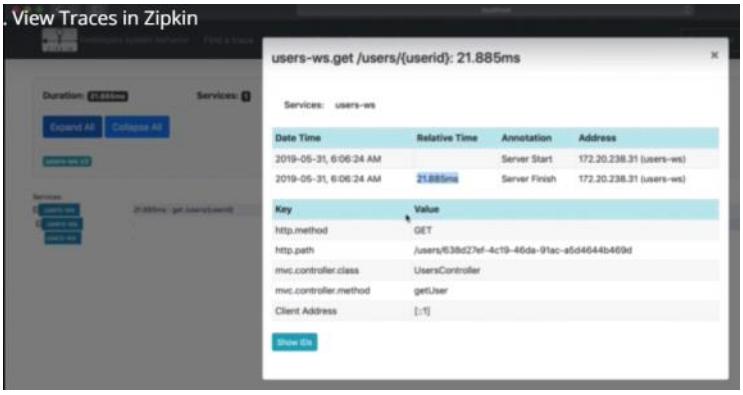
21. Distributed Tracing with Sleuth and Zipkin

For every request Sleuth adds a *traceId* and it will be seen in gui in *zipkin*

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

```
spring.zipkin.base-url=http://localhost:9411
spring.zipkin.sender.type=web
spring.sleuth.sampler.probability=1
```






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

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

The Spring Cloud projects you're going to use in this specific service

152