Microservices implementation — Netflix stack

Tharanga Thennakoon
Jun 18, 2017 · 10 min read

Hi. Today i am going to discuss and explain how to implements Microservices based system. There are lot of tools and technologies for implementing Microservices. Today i am focusing about doing with Netflix stack and SpringBoot. These days Microservices is one of the buzzing topic in the industry. Every one needs to know about Microservices and every one needs to doing there projects based on Microservices architecture.

Before going for the Microservices we must have clear idea about the concept of Microservices architecture, Why our project is going for Microservices, what are the pros and cons in Microservices architecture. In here Krish is doing very good tutorial series about Microservices. If you want to get very good understanding about Microservices architecture, go through these tutorials before going for the implementation.

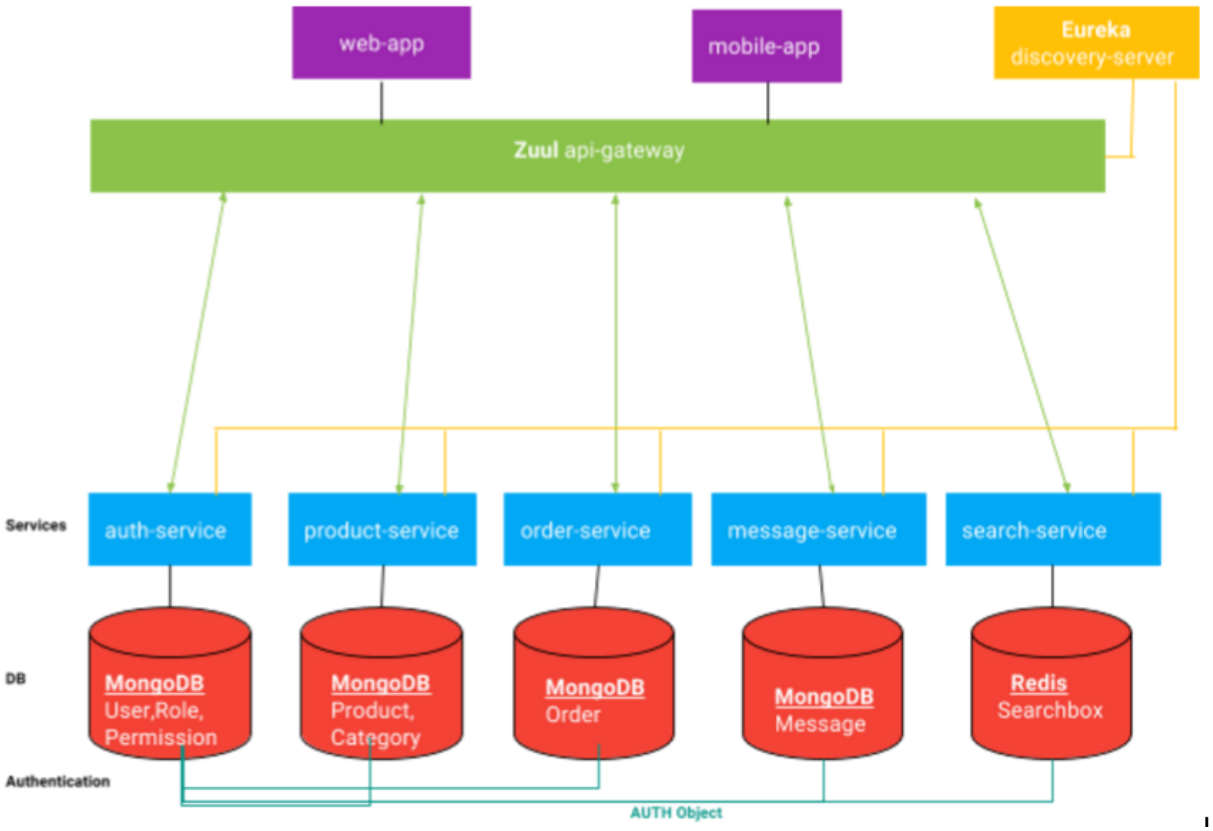
Main Topics

1. Introduction for the sample project
2. Core Services
3. Discovery Server
4. Discovery Client
5. Client Side Load Balancer
6. API Gateway
7. Security Flow
8. Service resilience and Fault tolerance
9. State-Less Server

10. Common-Lib

Introduction for the sample project

In the first step i will explain high level architecture of the system for getting idea about what we are going to do. This is just a sample application for easy of explanation. Just assume e-commerce based web application. All theses system design and implementation explanations is based on my previous projects experience .Some times your system design may be different according to your requirements.



Core services

You can see here according to the Microservices architecture we divide whole application into separate serveries. Each service are salable and independently deploy-able and they have exact well defined scope. Those are built as SpringBoot projects. I will provide the source project in end of this article So don't worry about the coding try to understand the concept. In this example we have five core service.

The **auth-service** responsible for handle authentication process of the system, auth-user retrieve , auth-user store. It connects with User, Role, Permission db tables. In here i used MongoDB.

The **product-service** is responsible for products store, products listing like stock handling processes.

The **order-service** is responsible for handling orders which placed by buyers for the products. Then seller process the order which got for his products.

The **message-service** is responsible for handling real-time messaging flow between each users within the system.

The **search-service** is responsible for searching products, users, categories. In these kind of applications search is a most demanding operation so In here i used Redis with Searchbox for providing high performance service. Bus this is just an example project, you can choose whatever suitable technology.

Discovery Server

In the previous section we discussed about core services of our system. Those are independent services. Some of them are deploy inside a single server machine or other may deploy another server machine. Sometime some of services have to communicate with other services in some situations, i will explain what are the situations later. But how we communicate with another service if we don't known where it is locate. This is like we are trying to make a phone call for someone without knowing his phone number.

Solution is Discovery Sever. Discovery Sever helps to discover the service we required . When some service need to access another service , Discovery Sever provides all the endpoint details of the requested service to establish the connection. Discovery Sever act as Service Registry. All of the services needs to register with Discovery Sever , other wise Discovery Sever don't know about that service. There are multiple implementations for the Discovery Server Ex: Netflix Eureka, Consul, Zookeeper. In here i will discuss about Netflix Eureka.

We can easily setup Netflix Eureka as our Discovery Service. I am not going to each coding section in here, i am focus for important point of configurations and implementation. You can find it in source project. For making the Springboot application as a Eureka Discovery Server you have to mention it by putting **@EnableEurekaServer** annotation into startup (main method)of the application.

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

```
@SpringBootApplication
@EnableEurekaServer
public class App
{
    public static void main( String[] args )
    {
        SpringApplication.run(App.class, args);
    }
}
```

you have to put some configuration. In this case i put inside **application.yml** (or you can use **application.properties**)

```
# Configure this Discovery Server
eureka:
  instance:
    hostname: localhost
  client:  # Not a client, don't register with yourself
    registerWithEureka: false
    fetchRegistry: false

# HTTP Server
server:
  port: @http.port@
```

In here i mention this instance is not a Discovery client and in here server's port is getting from maven properties. But you can put whatever port number you need. Default port is 1111.

Discovery Client

In the Service Discovery section i told every services must register with the Discovery Server. So we have to mention each service as a Discovery client by using **@EnableDiscoveryClient .** This annotation can work with any Discovery Client implementations which implements in your project (Eureka, Consul, Zookeeper) **.** You can also use **@EnableEurekaClient** annotation but it works only with Eureka Discovery Client implementation**.**

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

@SpringBootApplication
@EnableDiscoveryClient
public class App
{
    public static void main( String[] args )
    {
        SpringApplication.run(App.class, args);
```

```
        SpringApplication.run(App.class, args);
    }
}
```

you have to put some configuration.

```
# Spring properties
spring:
  application:
    name: auth-service


# HTTP Server
server:
  port: @http.port@

# Discovery Server Access
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:1111/eureka/
  instance:
    easeRenewalIntervalInSeconds: 5   # DO NOT DO THIS IN PRODUCTION
```

In here you must mention the **spring.application.name.** Because that name will be the **Service-Id** used by others to access this service.

Other thing i need to point , we can keep clusters of the service according to the demand for the service by using same Service-Id. Actually Eureka Server keeps the tracks for his Discovery Clients by using **Instance-Id.** Someone can requests from Eureka for getting all the endpoints available for the particular service by providing Service-Id. Then Eureka is capable to provide all the list of the service endpoints for that service client.

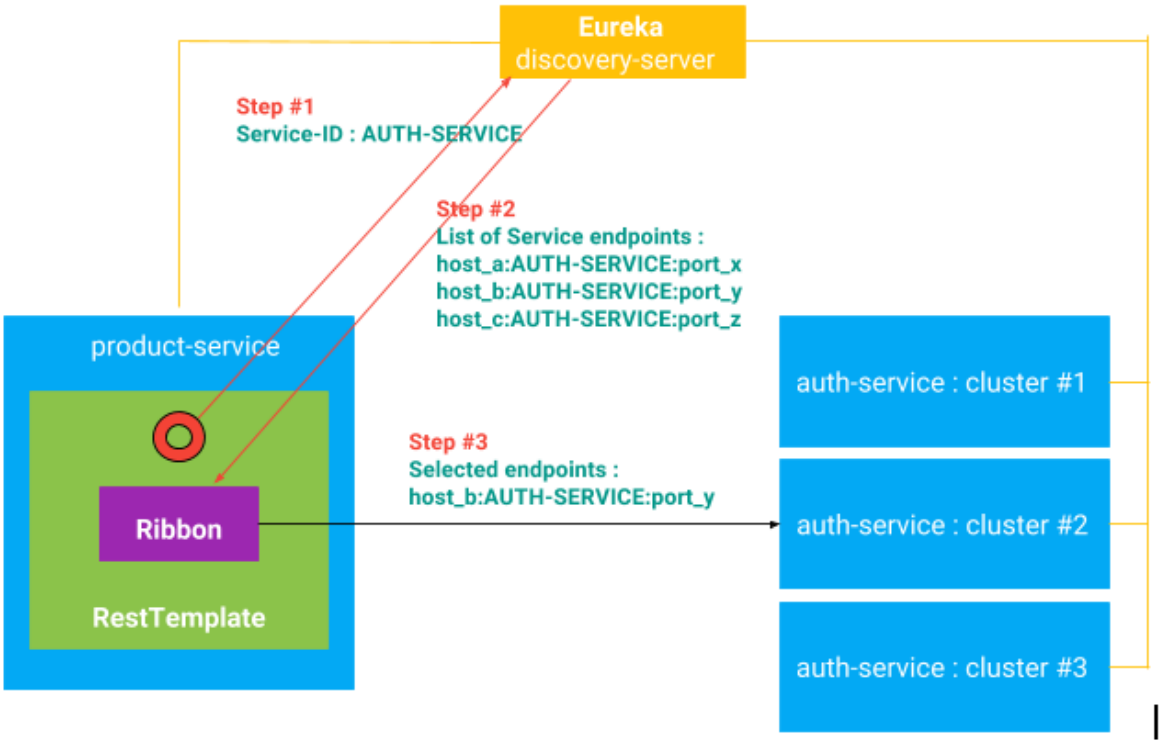
```
Instance-Id = {Server-Host} + ':' {Service-Id} + ':' + {Server-Port}
```

And you can see there are some simple configurations. You have to mention location of the Eureka Server by using **defaultZone** property**.** With **leaseRenewalIntervalInSeconds** you can change the Registration time. Registration takes up to 30 s because that is the default client refresh time.

Client Side Load Balancer

I already told you. In some situations , some of the services needs to get service or data from other service. In that scenario one service is become a client of the other service. The client-service can call required service by using Service-Id. But assume that required service keep clusters. Then Eureka provides all of the endpoints for the requested service. Now how client-service decide which endpoint is need to select to establish the connection. This is the time Client Side Load Balancer come into the play.

According to the Load Balancing algorithm , Client-Side-Load-Balancer will choose the best endpoint from the list for you to establish the connection. In our case I used Netflix Ribbon as a Client-Side-Load-Balancer.



According to your requirement you can choose what is the best Load Balancing algorithm. In Ribbon there are several implementations .Simple Round Robin LB, Weighted Response Time LB, Zone Aware Round Robin LB
Random LB. Or you can implement your own LB implementation. Default one is Simple Round Robin.

Spring framework provide easy way for access REST endpoints with **RestTemplate** class.

```
@LoadBalanced
```

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

Before used to RestTemplate you have to create instance within the spring context. **@LoadBalanced** annotation will helps to setup Ribbon configurations into RestTemplate. This is how to access another service by using RestTemplate.

```
ResponseEntity<String> responseEntity =
        restTemplate.exchange(
                "http://AUTH-SERVICE/auth/current"
                , HttpMethod.POST
                , entity
                , String.class);
```

I hope now you know how we can make inter-services communication by using Ribbon and Eureka.

API Gateway

Now we are going implement the front door of the system. How external users (web app, mobile app) are going to access our service Or in other words how we can expose micro services for external users. Yes solution is API Gateway. The external users of the system access our core services through the API Gateway. In here we use Netflix Zuul API Gateway. We can use Zuul as a proxy and as a request filter. I will explain more details by looking at the configurations.

```
# Spring properties
spring:
  application:
    name: apigateway

# Server
server:
  port: @http.port@   # HTTP (Tomcat) port

# Discovery Server Access
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:1111/eureka/
  instance:
    leaseRenewalIntervalInSeconds: 5

# Api-Gate-Way
zuul:
  prefix: /api
  routes:
```

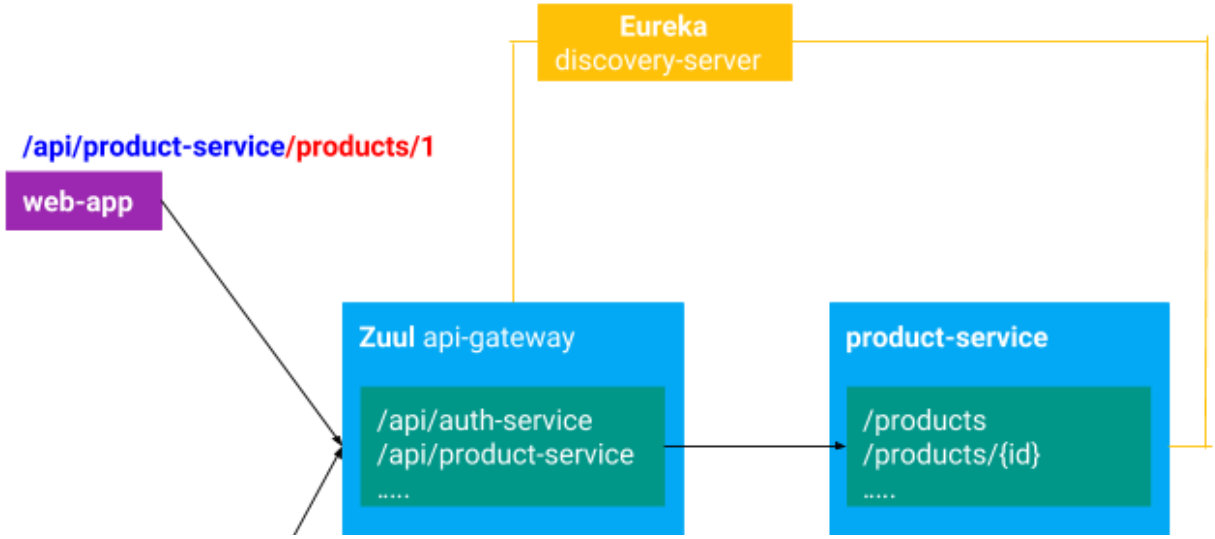
```
auth-service:
  path: /auth-service/**
  serviceId: AUTH-SERVICE

product-service:
  path: /product-service/**
  serviceId: PRODUCT-SERVICE

search-service:
  path: /search-service/**
  serviceId: SEARCH-SERVICE
```

You can see here, We have to mention Eureka Server details as we did in Discovery Client section. The reason is when the request is coming into the Zuul, He is going to access the particular core service by using Service-Id. Just like previously we did. We don't need to worry about Client-Side-Load-Balancing , Zuul is doing the Load-Balancing by using Ribbon.

In side the Zuul properties we can mention what is the url prefix. In the routes properties section we have to mention each core services with identical name. Inside that we can mention url path to access the core service. Finally we have to mention Service-Id for the particular service.

As a example if external user need to access core product service. His requested endpoint URL may be like theses structures. Assume Zuul is running on 8080.

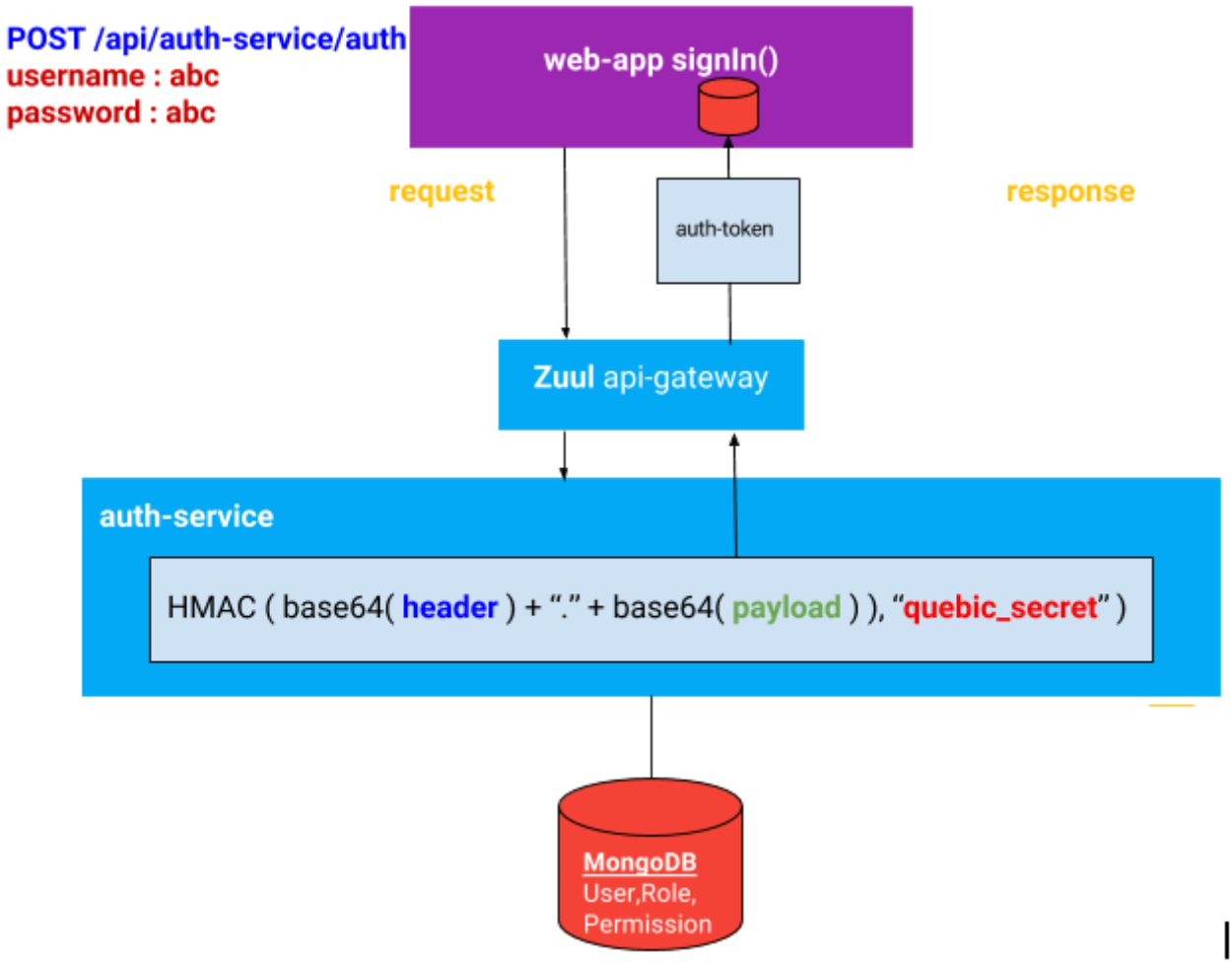
> *http://localhost:8080/api/product-service/{**core-product-service-end-point**}*
>
> *http://localhost:8080/api/product-service/**products***
>
> *http://localhost:8080/api/product-service/**products/10***



Security Flow

In this section i am going to discuss how security handling in side our application. For this , I mainly used JWT (JSON Web Token) and Spring Security.

Sign-in



You can see here sign-in request comes into the auth-service. Lets see how **auth token** is generated.

```
header = {"alg": "HS256", "typ": "JWT"}
```

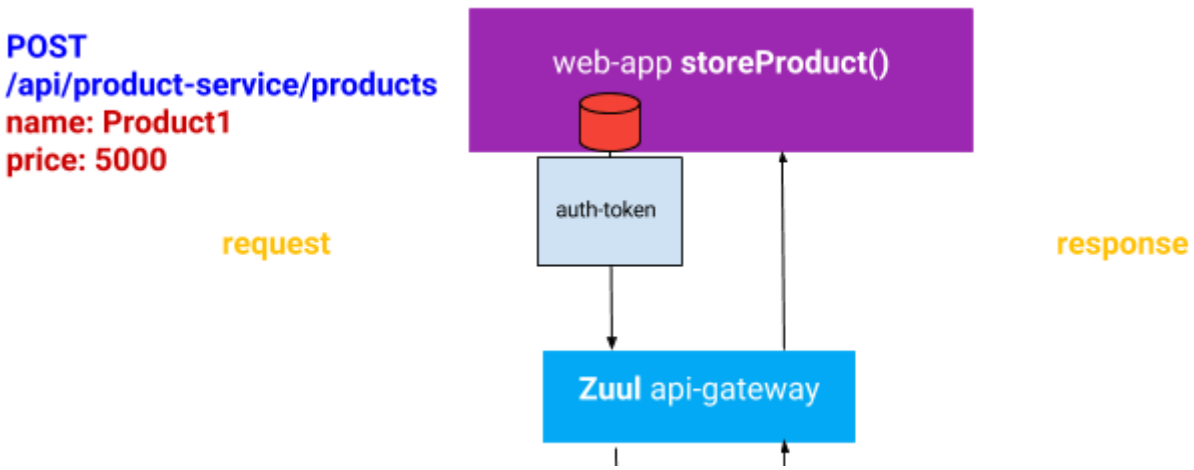
```
payload = {
                "exp": "2017-08-09 12:00:00",
                "user_name": "user",
                "authorities": [
                     "ROLE_SELLER"
                 ],
                 ...
                }

secret_key = "quebic_secret"

Token = HMAC( base64(header) + "." + base64(payload) , secret_key)
```

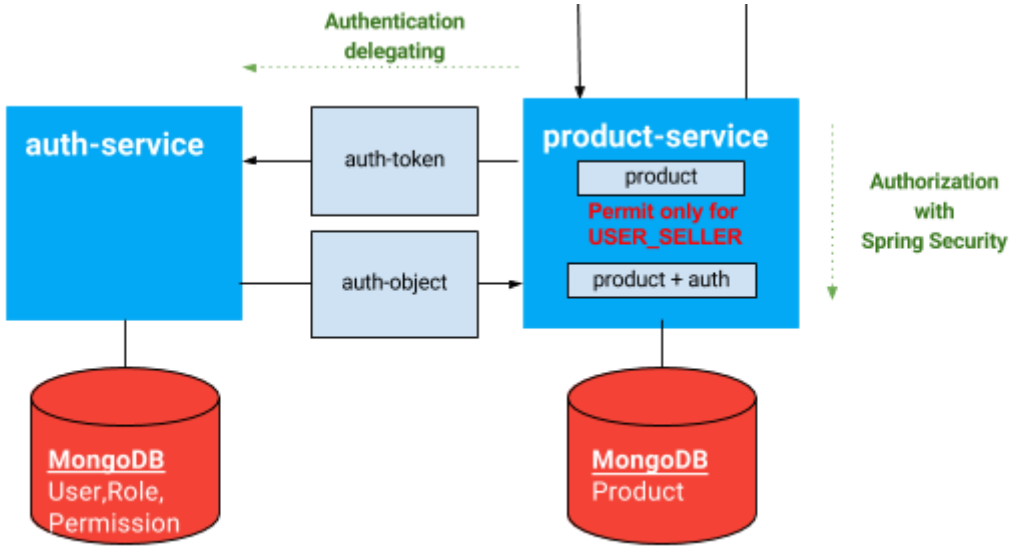
The service checks username and password is valid, If credentials are correct then auth-service creates the payload. The payload contain user name, authorities and expiration of the token. The secret key is stored inside the configurations (application.yml) of each services.

```
# Auth JWT
jwt:
  header: Authorization
  secret: quebic_secret
  expiration: 172800 #seconds
  route:
    authentication:
      path: auth
      refresh: refresh
```

In side jwt configurations we can set what is the request header which contained auth-token. For this we use Authorization header. And you can change secret and expiration values according to your requirement. In this scenario web-app stores the generated auth-token in side the browser storage.

Authorization





According to our sample project the product store operation is allow only for Sellers (USER_SELLER). You can see here web-app send the new product details for product-service.

product_service don't know anything about incoming users so he calls for auth-service for getting **auth-object**. auth-service is capable to decrypt auth-token and if token is valid auth-service return auth-object. auth-object contains userId, username and authorities. Now according to authorities product_service will continue the process.

In here you have understand main key points of our security flow. First thing is any of our core services not going maintain session for about logging user. All services are state-less. And second thing is product-service, order-service, message-service and search-service are delegate authentication process for the auth-service. After getting the auth-object they can handle the Authorization process because each core-service contain permission rules which implemented by using Spring Security.

Authentication Delegating Filter

CommonAuthenticationTokenFilter is the implementation of the Authentication Delegating. This is contain in our common-lib dependency. You can find this filter class inside com.quebic.common.security package. By override the **doFilterInternal()** method of the **OncePerRequestFilter** class, I implemented Auth Delegating logic.

```
@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)

    String authToken = request.getHeader(this.tokenHeader);

    if (!StringUtils.isEmpty(authToken) && SecurityContextHolder.getContext().getAuthentication() == null) {

        try{
```

```
        HttpHeaders headers = new HttpHeaders();
        headers.add("Authorization", authToken);
        headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));

        HttpEntity<String> entity = new HttpEntity<String>("", headers);

        ResponseEntity<String> responseEntity =
                restTemplate.exchange(
                        "http://AUTH-SERVICE/auth/current"
                        , HttpMethod.POST
                        , entity
                        , String.class);

        String jsonUserDetails = responseEntity.getBody();
        UserDetails userDetails = prepareUserDetails(jsonUserDetails);
```

Authorization with Spring Security

You can find the Authorization rules from **WebSecurityConfig** class. Each service contains its own implementation and this class is locate under security.config package.

```
@Override
protected void configure(HttpSecurity httpSecurity) throws Exception {
    httpSecurity
            // we don't need CSRF because our token is invulnerable
            .csrf().disable()

            .exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()

            // don't create session
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()

            .authorizeRequests()
            .antMatchers(HttpMethod.OPTIONS, "/**").permitAll()

            .antMatchers(HttpMethod.GET, "/products/**").permitAll()
            .antMatchers(HttpMethod.POST, "/products/**").hasRole(Permission.USER_SELLER)
            .antMatchers(HttpMethod.PUT, "/products/**").hasRole(Permission.USER_SELLER)

            //authenticated requests
            .anyRequest().authenticated();

    // Custom JWT based security filter
    httpSecurity
            .addFilterBefore(authenticationTokenFilterBean(), UsernamePasswordAuthenticationFilter.class);

    // disable page caching
    httpSecurity.headers().cacheControl();
```

Sign-out

When going as state-less, it is impossible to invalidate auth-token before they are expired. So when sign-out throw away the token from the client side. As an example if client is a web app, We can released auth-token from the browser storage.

Service resilience and Fault tolerance

When we design Microservices based project, we must consider about implementation about Service resilience and Fault tolerance mechanism. There are several ways to

implements this and Circuit Breaker Pattern is a good way to handle this. **Netflix Hystrix** is an implementation of the Circuit Breaker Pattern.

```
@HystrixCommand(fallbackMethod = "reliable")
  public String readingList() {
    URI uri = URI.create("http://localhost:8090/recommended");

    return this.restTemplate.getForObject(uri, String.class);
  }

  public String reliable() {
    return "Cloud Native Java (O'Reilly)";
  }
```

If readingList() method is failed within its execution , The fall-back method will fire immediately without breaking the main process.

State-Less Server

As i mentioned in the earlier, all the core services must be salable and independently deploy-able. According to demand we can keep multiple instance from same service. But if we keep server based session we have to share those session data when we deploy new instance. It will kill the freedom of the Microservices architecture. So all the services are designed as state-less server.

Don't keep session data or cache inside server memory, store those inside distributed in-memory storage. There are lots of good solutions available. Ex: Hazelcast, Redis, Memcache.

Common Lib

If you have some thing common for all the services , don't duplicate it inside the each service. Put those inside common places. In this example i keep separate project called common-lib. Each core services have to add common-lib as a dependency.

OK now we came up to the end of the discussion. I think you got a considerable knowledge about the implementation of Microservices architecture. I hope this will helps for your future projects. This is the GitHub source project and please follow the instructions which i given in the Readme to run the project. You can host the

Microservices inside AWS EC2, Pivotal WebServices or heroku. I will explain more about hosting in the future article. Thanks for reading. Good luck.

Microservices Java Spring Boot Netflixoss Web Development