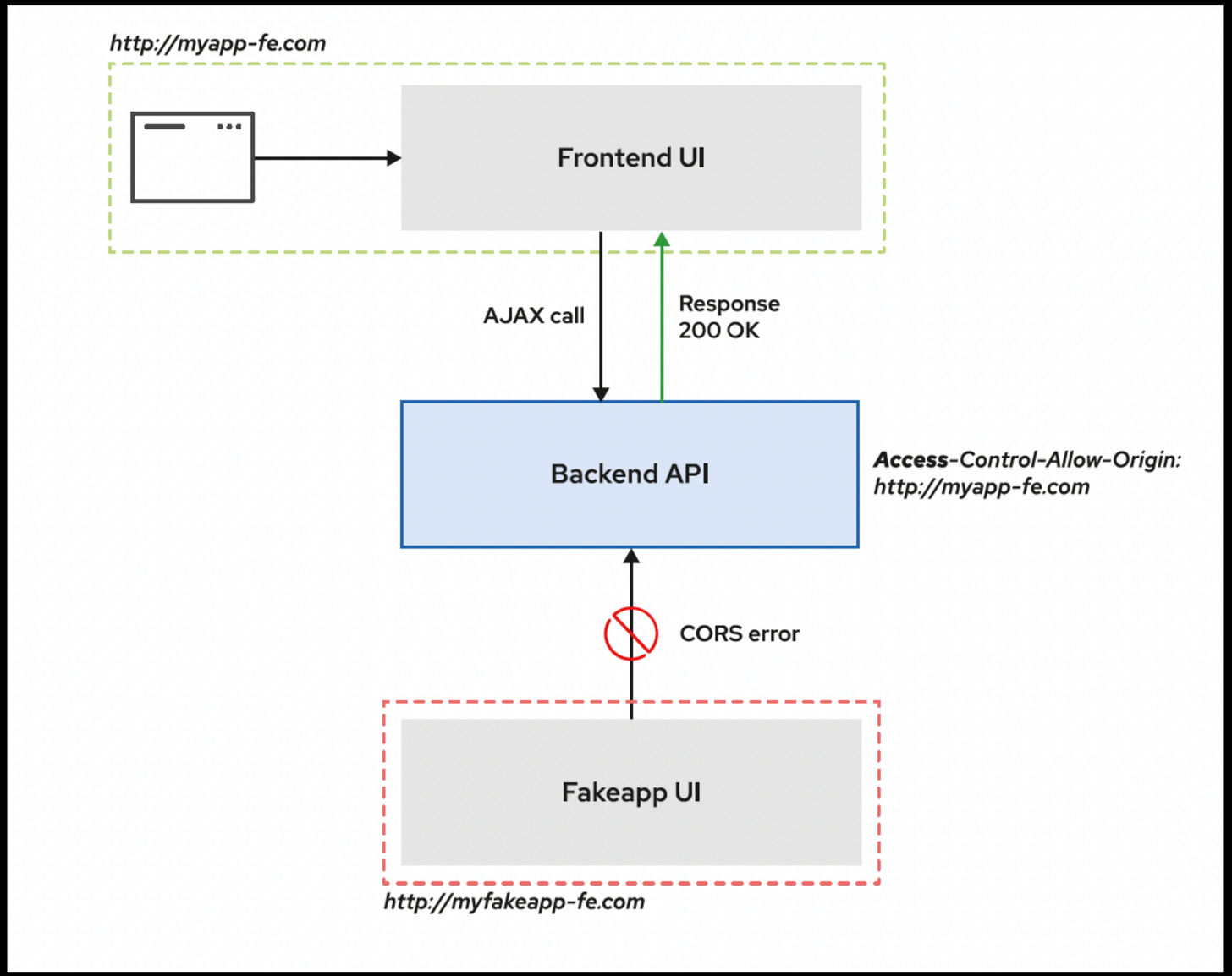
Seguridad

Microservicios

JoeDayz | 2025

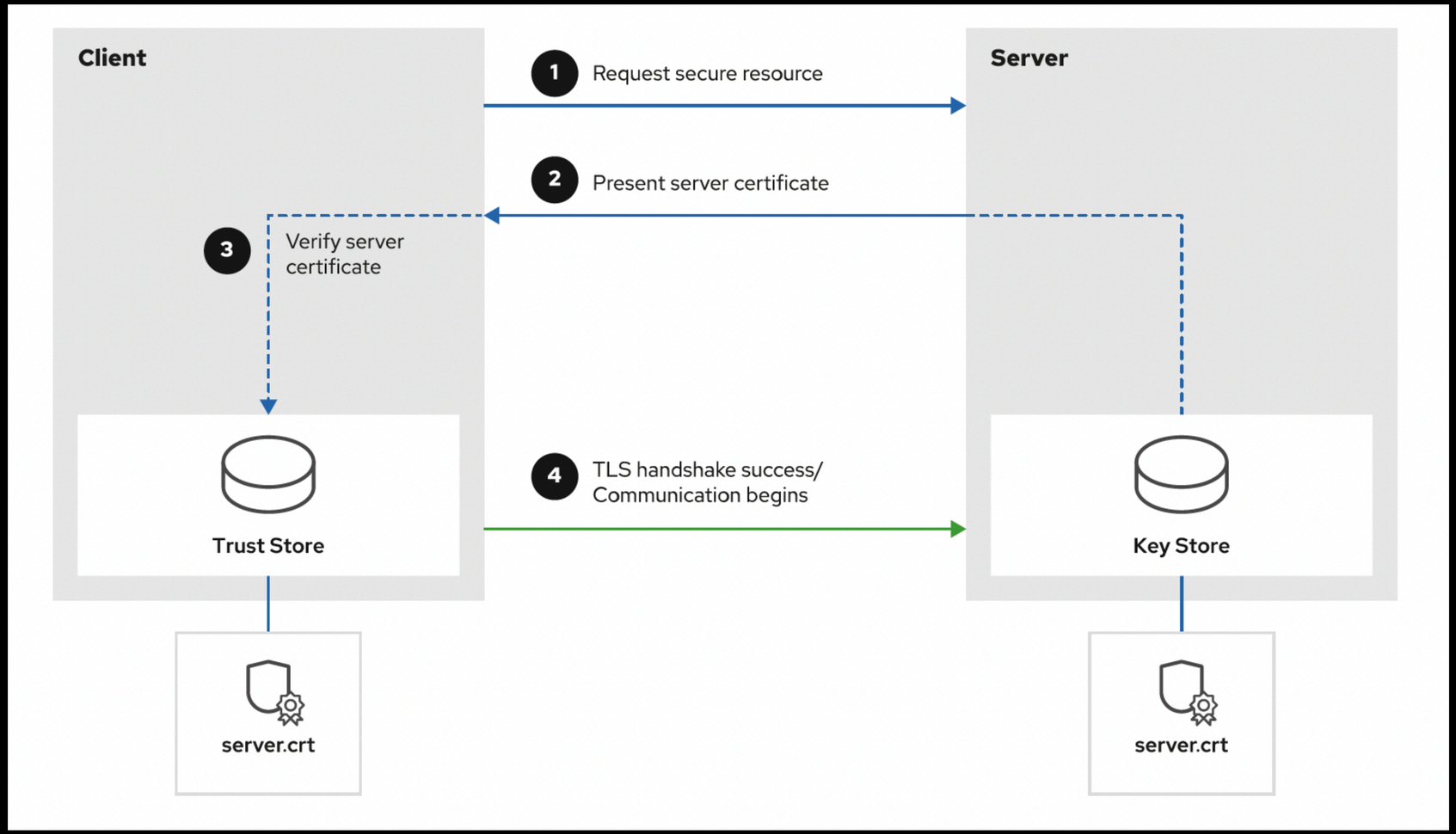
CORS


http://myapp-fe.com
Frontend UI
AJAX call
Response
200 OK
Backend API
Access-Control-Allow-Origin:
http://myapp-fe.com
CORS error
Fakeapp UI
http://myfakeapp-fe.com

Configuración CORS en Quarkus

```
quarkus.http.cors=true 1
quarkus.http.cors.origins=http://example.com,http://www.example.io 2
quarkus.http.cors.methods=GET,PUT,POST 3
quarkus.http.cors.headers=X-My-Custom-Header,X-Another-Header 4
```

TLS (Transported Secure Layer)



Sucesor de SSL (Secure Socket Layer)

Basado en certificados X.509

Generate a KeyStore

1)

```
[user@host ~]$ keytool -noprompt -genkeypair -keyalg RSA -keysize 2048
  -validity 365 \ (1)
  -dname "CN=myapp,OU=myorgunit, O=myorg, L=myloc, ST=state, C=country" \ (2)
  -ext "SAN:c=DNS:myserver,IP:127.0.0.1" \ (3)
  -alias myapp \ (4)
  -storetype JKS \ (5)
  -storepass mypass -keypass mypass \ (6)
  -keystore myapp.keystore (7)
```

2)

application.properties

```
quarkus.http.ssl.certificate.key-store-file=/path/to/myapp.keystore
quarkus.http.ssl.certificate.key-store-password=mypass
quarkus.http.ssl-port=8443
```

En Unit Test es 8444

Configurable con:

quarkus.http.test-ssl-port

```
quarkus.http.ssl.certificate.key-store-file-type=[one of JKS, JCEKS, P12, PKCS12, PFX]
```

Desde Java 9 el default es PCKS12.

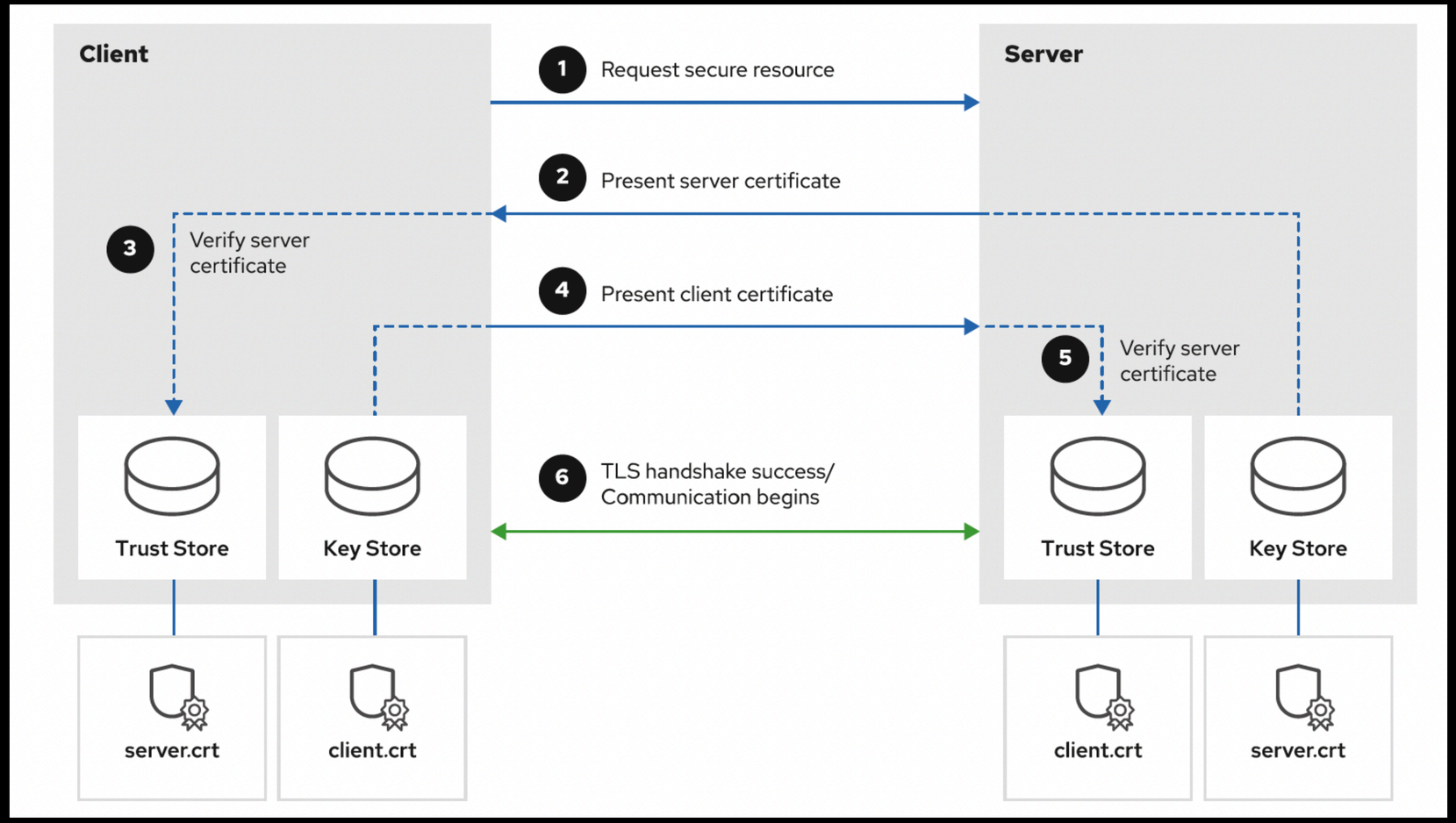
Desde Quarkus el default es JKS.

quarkus.http.insecure-request = enabled (HTTP (8080)y HTTPS(8443) son abiertos)

quarkus.http.insecure-request = redirect (Los requests HTTP son redirecciovados al puerto HTTPS)

quarkus.http.insecure-request = disabled (Solo trabajarías con HTTPS)

Configure Mutual TLS (mTLS) for Quarkus Applications



Mutual TLS es para implementar un modelo de seguridad zero-trust.

Es también, recomendado en escenarios donde tu aplicación API es invocado por partes externas, fuera de la red de tu organización.

Quarkus implementa Mutual TLS

1)

```
[user@host ~]$ keytool -exportcert -keystore myclient.keystore \ (1)
  -storepass mypass -alias myclient \ (2)
  -rfc -file myclient.crt (3)
```

2)

```
[user@host ~]$ keytool -noprompt -keystore myapp.truststore \ (1)
  -importcert -file myapp.crt \ (2)
  -alias myclient (3)
  -storepass mypass
```

3) Repetir 1) y 2) para exportar los certificados públicos del keystore del servidor e importar el certificado en el store del cliente.

4)

```
quarkus.http.ssl.client-auth=required
quarkus.http.ssl.certificate.trust-store-file=/path/to/myapp.truststore
quarkus.http.ssl.certificate.trust-store-password=mypass
```

Quarkus implementa Mutual TLS

5) En el lado del cliente, también hay que configurar el uso de Mutual TLS.

```
org.acme.client.mtls.GreetingService/mp-rest/url=https://myserver:8443 (1)
org.acme.client.mtls.GreetingService/mp-rest/trustStore=/path/to/
myclient.truststore (2)
org.acme.client.mtls.GreetingService/mp-rest/trustStorePassword=mypass
org.acme.client.mtls.GreetingService/mp-rest/keyStore=/path/myclient.keystore (3)
org.acme.client.mtls.GreetingService/mp-rest/keyStorePassword=mypass
```

Nota: Quarkus coloca seguridad a nivel de la aplicación. Pero, a nivel de networking, tendrías que usar Service Mesh.

Recursos

- Cross Origin Resource Sharing (CORS) https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
- CORS filter en Quarkus https://quarkus.io/version/2.13/guides/http-reference#cors-filter
- Cómo implementar Mutual TLS en Quarkus https://quarkus.io/blog/quarkus-mutual-tls/
- Ejemplo de mTLS en Github: https://github.com/openlab-red/quarkus-mtls-quickstart
- RedHat OpenShift Service Mesh https://access.redhat.com/documentation/en-us/openshift_container_platform/4.11/html-single/service_mesh/index
- Istio https://istio.io/latest/docs/

Quiz

1. **In your application, a `wishlist` service with the `wishlist.clouda.example.com` domain sends requests to the `user` service with the `user.cloudb.example.com` domain. Based on the communication requirements between two back end services, which of the following statements about CORS configuration is correct?**

   a. The `user` service must put the `wishlist.clouda.example.com` domain to the `Access-Control-Allow-Origin` header.

   b. The `wishlist` service must put the `user.cloudb.example.com` domain to the `Access-Control-Allow-Origin` header.

   c. Neither service must configure CORS for the provided communication pattern because a browser is not involved.

   d. Both services must add the domain of the other service to the `Access-Control-Allow-Origin` header.

2. **Which of the following statements about TLS is correct?**

a. CORS requires TLS to function.

b. TLS uses X.509 certificates to encrypt network traffic. The server encrypts the traffic with a private key, and then sends the private key to the client for data decryption.

c. Applications store public and private keys in a trust store.

d. TLS uses X.509 certificates to encrypt network traffic. The server encrypts the traffic with a private key. The client decrypts the data by using the server's public key.

3. **Which two of the following statements about mutual TLS is correct? (Choose two.)**

a. Mutual TLS is a two-way authentication method, which means that the client verifies both the public and private keys of the server.

b. Mutual TLS is a two-way authentication method, which means that the client verifies the server identity, and the server verifies the client identity.

c. In mutual TLS, the public certificate provided in the initial server response must match a client certificate identity. The client public certificate then must match the server certificate identity.

d. In mutual TLS, the public certificate provided in the initial server response must match an entry in the client trust store. The client public certificate then must match an entry in the server trust store.

4. **You configured the back end service to use mutual TLS. However, your testing revealed that the back end service serves requests with invalid certificates. Based on the provided back end `application.properties` file, what is the most likely problem with the configuration?**

```
quarkus.http.ssl.certificate.key-store-file=META-INF/resources/server.keystore
quarkus.http.ssl.certificate.key-store-password=password
quarkus.http.ssl.certificate.trust-store-file=META-INF/resources/server.truststore
quarkus.http.ssl.certificate.trust-store-password=password
```

a. The `server.keystore` file does not contain the client certificate.
b. The password for the `server.keystore` file is incorrect.
c. The trust store file does not contain the client certificate.
d. The configuration file is missing the CORS configuration.
e. The configuration file is missing the `quarkus.http.ssl.client-auth` configuration property.

Asegurando los microservicios y control de accesos basado en Roles usándo JSON Web Tokens (JWT)

Seguridad usando JSON Web Tokens (JWT)

- JWT es un estándar de intercambio de información by HTTP. El incluye la identidad del usuario y algunos permisos.

- JWT tiene 3 partes: Header (HS256), Payload (user info, claims), Signature (JWT issuer)

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

```
{
    "sub": "48473",
    "name": "Student User",
    "iat": 1516239022
}
```

```
hs256(
    toBase64(header) + "." + toBase64(payload),
    secret
)
```

- JWT es codificado en Base64.

3. La firma fue con el algoritmo HS256 y el secreto my-secret-1234.

JWT Authentication Workflow

A typical JWT-based authentication workflow is similar to the following steps:

1. A user logs in to the application.
2. The application verifies the user credentials. If the credentials are valid, then the application issues a JWT for the user. This JWT commonly includes information such as unique user IDs, roles, and permissions.
3. The user receives and stores the JWT. In front ends, for example, the browser stores the JWT in a cookie or in the browser local storage.
4. The user makes a request to a secure endpoint, by including the JWT in the `Authorization` HTTP header. The value of the `Authorization` header must use the *Bearer* format. Consequently, each request uses the `Authorization: Bearer` *`TOKEN`* header.
5. The application receives the request, extracts the JWT from the `Authorization` header, and validates the JWT signature to verify the authenticity of the user.

Construyendo Tokens con SmallRye JWT

```
[user@host myapp]$ mvn quarkus:add-extension -Dextensions=smallrye-jwt-build
```

```
smallrye.jwt.sign.key.location = /path/to/privateKey.pem
```

JWK soportado.

La extensión usa el algoritmo RS26 para firmar los tokens.

Usando el JWT Builder Language

```
String token = return Jwt.issuer( "my-issuer" ) 1
        .upn( "john_1234" ) 2
        .subject( "48373758673" ) 3
        .audience("myapp.example.com") 4
        .claim("main_interest", "books") 5
        .groups( new HashSet<>(
            Arrays.asList( "REVIEWER", "DEVELOPER", "ADMIN" ) ) ) 6
        .sign(); 7
```

Mecanismos de Autenticación con SmallRye JWT

Basic HTTP, OpenID Connect (OIDC), OAuth2, y JWT.

```
[user@host myapp]$ mvn quarkus:add-extension -Dextensions=smallrye-jwt
```

```
import io.quarkus.security.identity.SecurityIdentity;

@Path( "/me" )
public class MyUserResource {

    @Inject
    SecurityIdentity securityIdentity; (1)

    @GET
    @Path( "/username" )
    public String getMyUsername() {
        var me = securityIdentity.getPrincipal(); (2)
```

me.getName();

Configurar la autenticación JWT

```
mp.jwt.verify.issuer=https://example.com/issuer
mp.jwt.verify.publickey.location=/path/to/publicKey.pem
```

AUTORIZAR LOS REQUESTS

```
    @Path("review")
    @RolesAllowed("REVIEWER") (1)
    public String getForReviewers(@Context SecurityContext context) { (2)
        Principal authenticatedUser = context.getUserPrincipal();

        if ( authenticatedUser != null ) {
            String username = authenticatedUser.getName();
        }
        ...implementation omitted...
    }

    @GET
    @Path("secured")
    @RolesAllowed("ADMIN") (3)
    public String getAdminsOnly() {
        ...implementation omitted...
    }

    @GET
    @Path("unsecured")
    @PermitAll (4)
    public String getUnsecured() {
        ...implementation omitted...
    }
}
```

CONFIGURAR AUTHORIZATION como Properties

```
quarkus.http.auth.policy.policy1.roles-allowed=user,admin
```

```
quarkus.http.auth.permission.permission1.policy=policy1
quarkus.http.auth.permission.permission1.paths=/secured/*
quarkus.http.auth.permission.permission1.methods=GET,POST
```

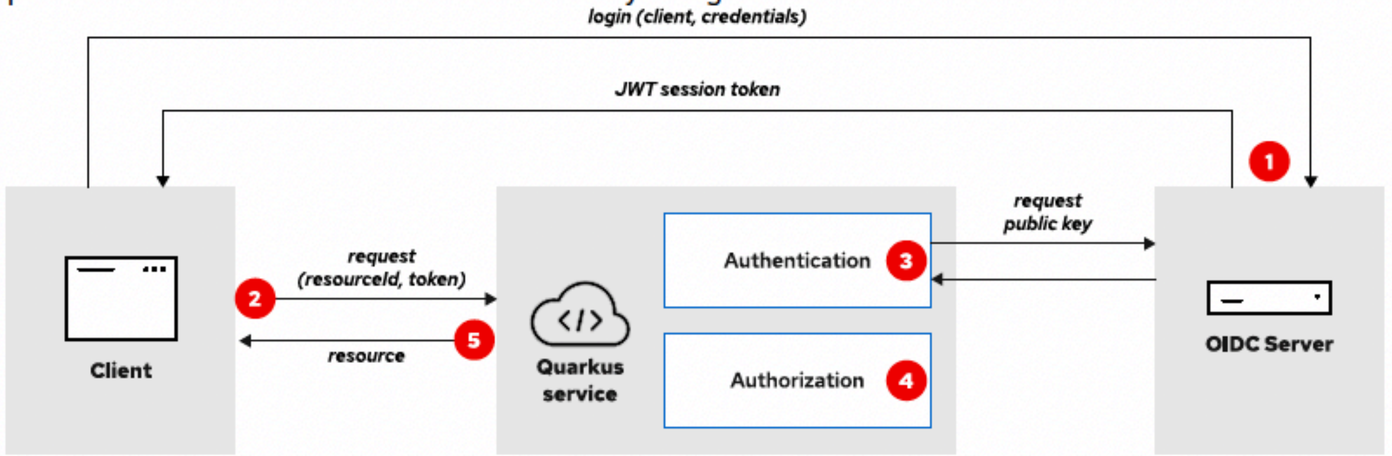
```
quarkus.http.auth.permission.permission2.policy=permit
quarkus.http.auth.permission.permission2.paths=/public/*
```

quarkus.security.jaxrs.deny-unannotated-endpoints = true

Implementar SSO usándo OpenID y OAuth

Implement Authentication with OIDC

OpenID Connect (OIDC) is an authentication and authorization layer added to the OAuth 2.0 protocol. OIDC provides sign-in flows that let application developers delegate authentication and authorization to an OIDC-compliant server, such as Red Hat Single Sign-On (RHSSO). Developers commonly use the Keycloak server to implement OIDC for local development purposes, which is the upstream project of RHSSO. OIDC uses JSON Web Tokens (JWT) to authenticate a user and provide further information about the user by using JWT claims.



1. The client provides credentials to the OIDC server and obtains a JWT token.
2. The client requests a resource from the application and provides the token.
3. The application validates the cryptographic token identity by using the issuer verification keys. The application also validates the token contents, such as expiration date and its claims.
4. If the validation succeeds, then the application checks if the token is authorized to access the resources, for example by checking the roles associated with the token.
5. If the token is authorized, then the application returns the requested resource.

```
[user@host ~]$ mvn quarkus:add-extension -Dextensions=oidc
```

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-oidc</artifactId>
</dependency>
```

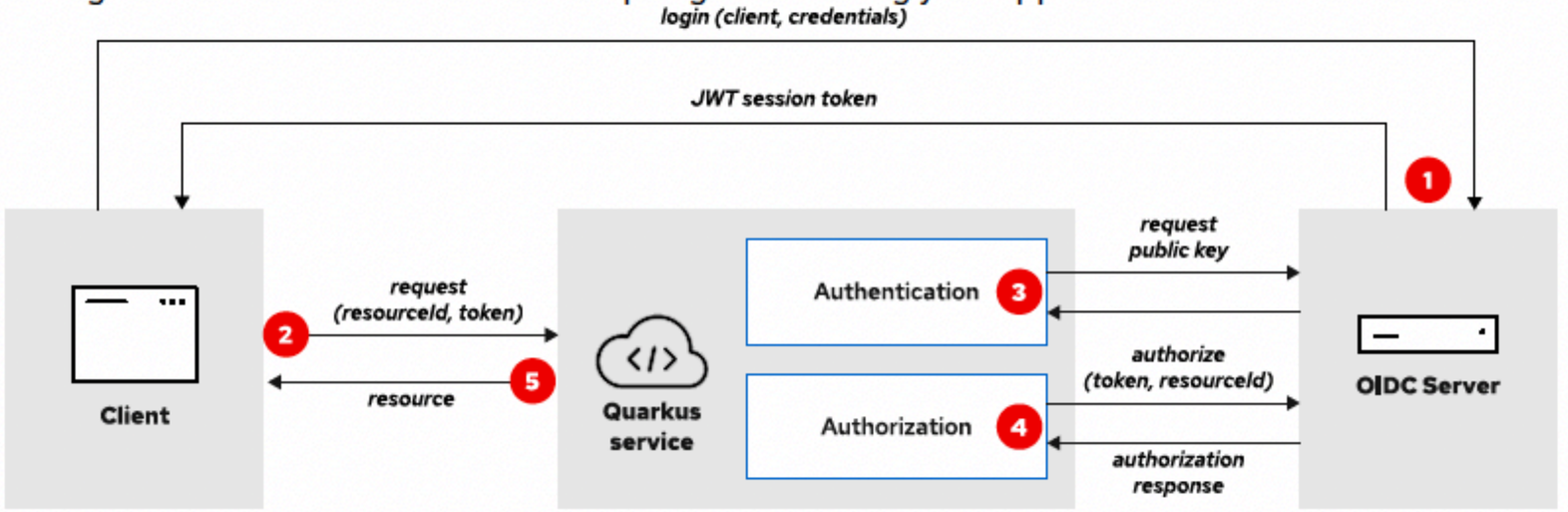
```
quarkus.oidc.auth-server-url=https://SERVER/realms/REALM_NAME (1)
quarkus.oidc.client-id=CLIENT_NAME (2)
quarkus.oidc.credentials.secret=CLIENT_SECRET (3)
```

```
%prod.quarkus.oidc.auth-server-url=https://SERVER/realms/REALM_NAME
quarkus.oidc.client-id=CLIENT_NAME
quarkus.oidc.credentials.secret=CLIENT_SECRET

quarkus.keycloak.devservices.realm-path=import.json
```

```
public class ExampleResource {
  @Inject
  JsonWebToken jwt;

  public SecurityContext debugSecurityContext(@Context SecurityContext ctx) {
      return ctx;
  }
}
```

Implement Authorization with OIDC

Applications can delegate authorization configuration to the RHSSO server. Consequently, the application does not declare authorization per resource. Instead, developers configure authorization in the RHSSO server, which permits or denies access to the requested resource based on the authorization configuration. This means that you can change the authorization configuration at runtime without recompiling or restarting your application.



1. The client provides credentials to the OIDC server and obtains a JWT token.
2. The client requests a resource from the application and provides the token.
3. The application validates the cryptographic token identity by using the issuer verification keys. The application also validates the token contents, such as expiration date and its claims.
4. The application forwards the token and the resource ID to the OIDC server. The server authorizes the access and returns the appropriate response.
5. If the token is authorized, then the application returns the requested resource.

```
[user@host ~]$ mvn quarkus:add-extension -Dextensions=keycloak-authorization
```

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-keycloak-authorization</artifactId>
</dependency>
```

```
# Common OIDC configuration
quarkus.oidc.auth-server-url=https://SERVER/realms/REALM_NAME
quarkus.oidc.client-id=CLIENT_NAME
quarkus.oidc.credentials.secret=CLIENT_SECRET

# Enable authorization delegation
quarkus.keycloak.policy-enforcer.enable=true
```

```
quarkus.keycloak.policy-enforcer.http-method-as-scope=true
```