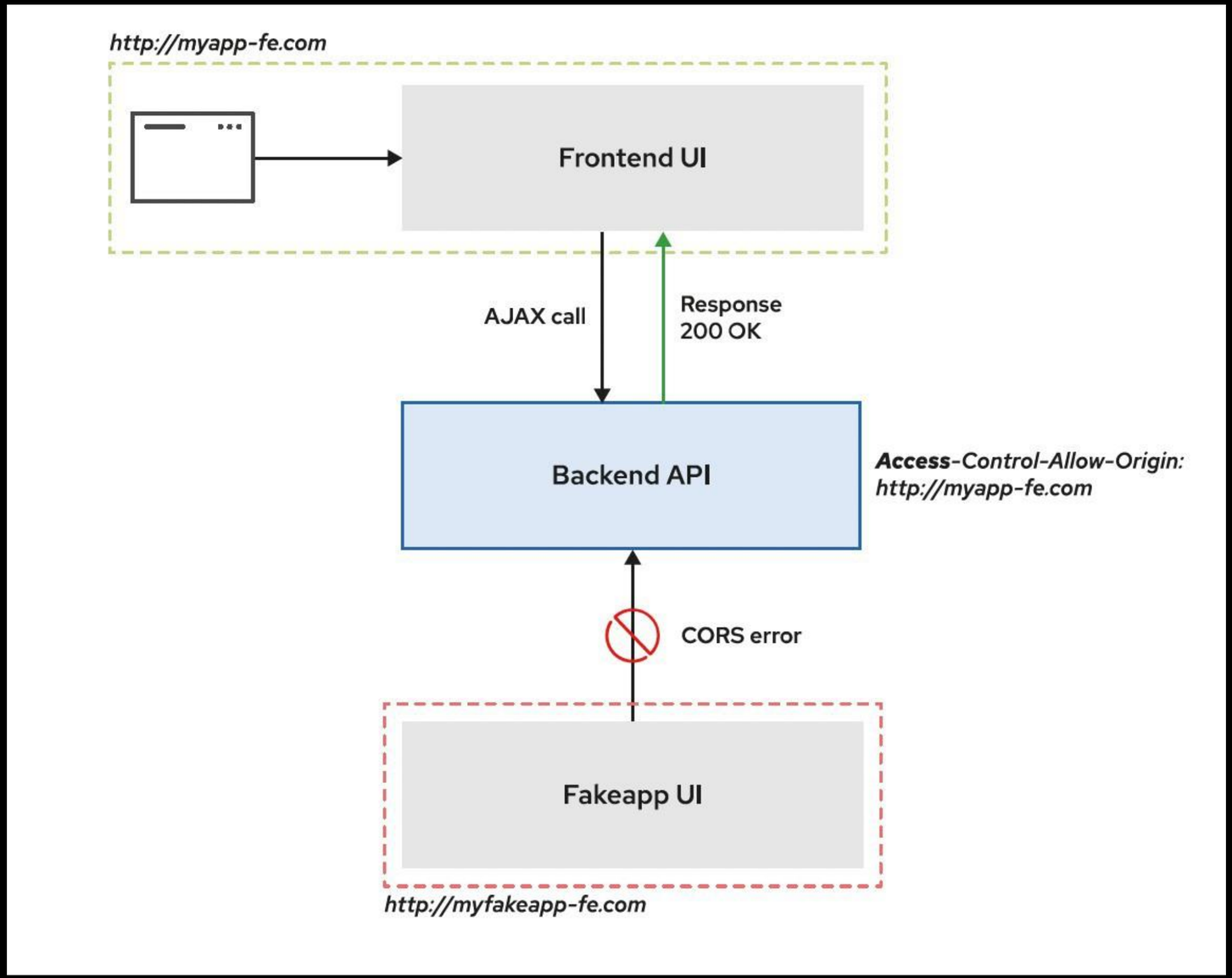
Seguridad

Microservicios

JoeDayz | 2026

Cross Origin resource sharing validation

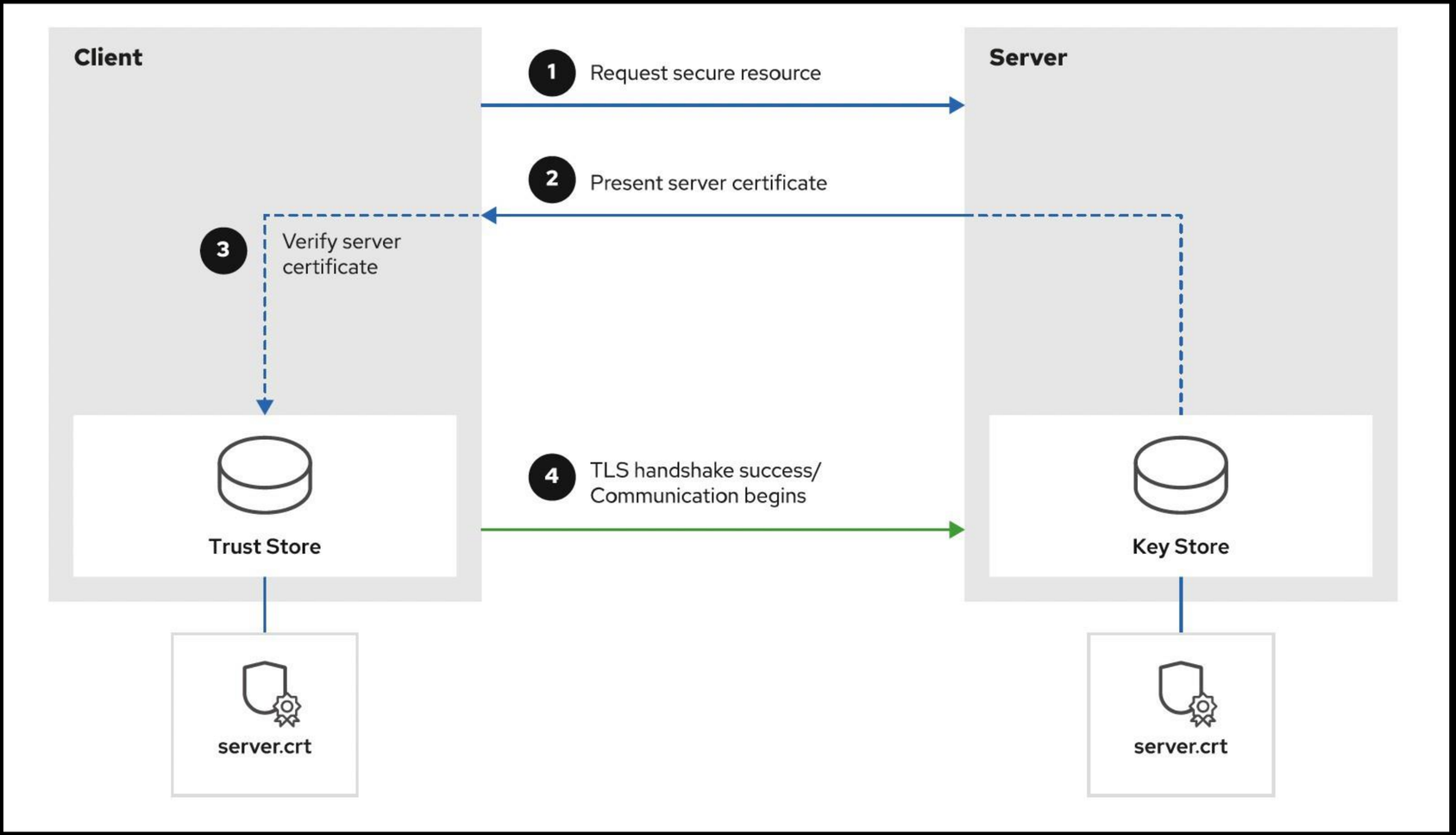


Configuración CORS en Quarkus

```
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
```

1) El default es false.
2) El default es *.

TLS (Transported Layer Security)



Sucesor de SSL (Secure Socket Layer)

Basado en certificados X.509

Configurar TLS en una aplicación Quarkus

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)
```

application.properties

2)

```
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.

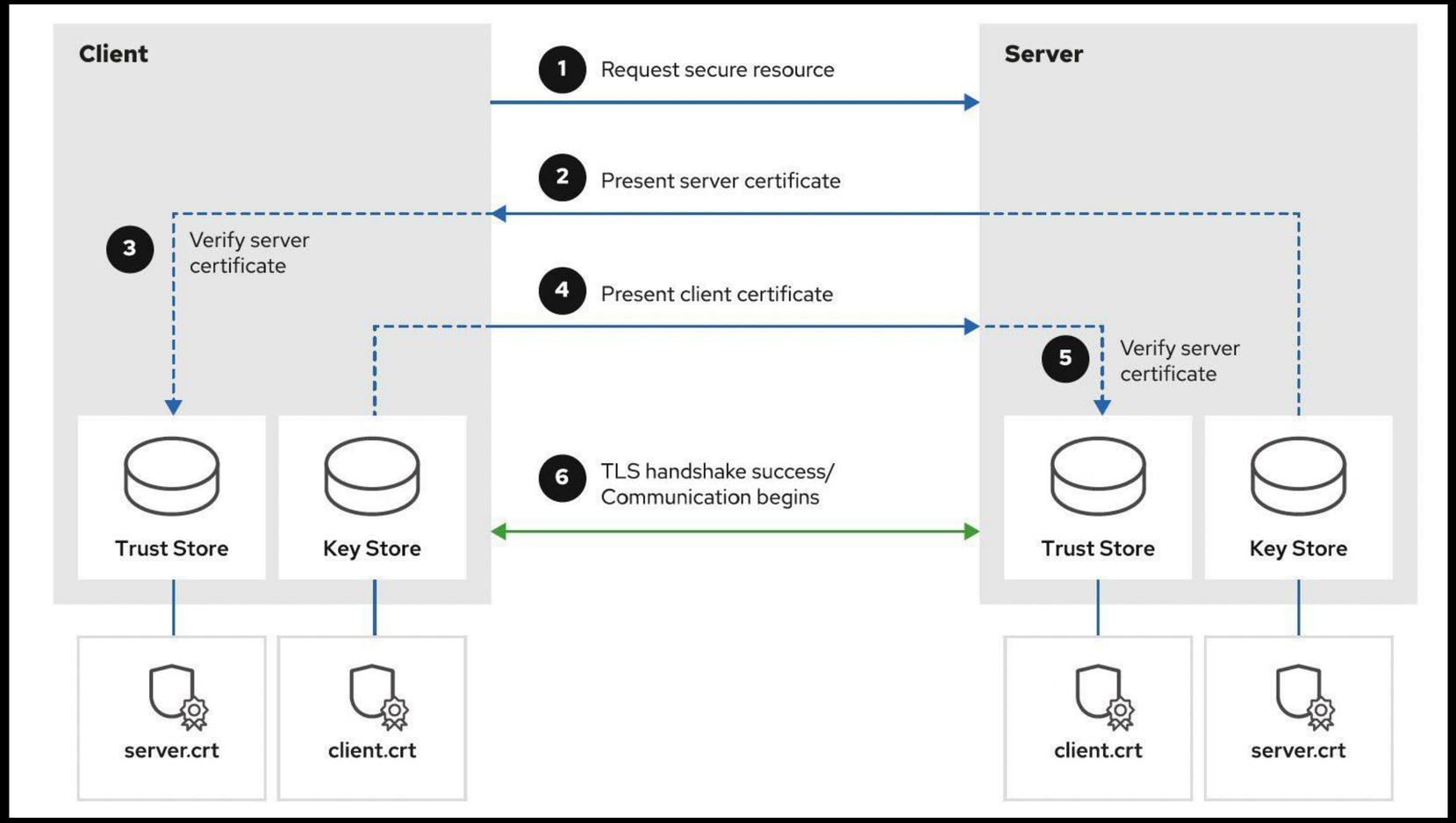
3)

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

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

quarkus.http.insecure-requests = 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.

Ejemplo práctico en Quarkus

Supongamos que tienes dos microservicios: ServiceA y ServiceB.

- ServiceA quiere llamar a ServiceB.
- ServiceB exige mTLS.
- Configuración en Quarkus (simplificada):

```
# ServiceA (cliente) HTTPS con certificado
quarkus.http.ssl.certificate.key-store-file=client-keystore.p12
quarkus.http.ssl.certificate.key-store-password=changeit

# ServiceB (servidor) HTTPS y mTLS
quarkus.http.ssl.certificate.key-store-file=server-keystore.p12
quarkus.http.ssl.certificate.key-store-password=changeit
quarkus.http.ssl.client-auth=required
```

✅ `client-auth=required` indica que el servidor **exige certificado del cliente**.

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/

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

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.

JWT encriptado

Header.Payload.Signature

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

JWT Claims

- Claims Registrados: Issuer (iss), Issued at (iat) y Subject (sub).

- Claims Públicos: Internet Assigned Numbers Authority (IANA), address, phone number

- Claims Privados: Son claims personalizados o privados para guardar información específica de la aplicación.

JWT Authentication Workflow

1. Un usuario inicia sesión en la aplicación.

2. La aplicación verifica las credenciales del usuario. Si las credenciales son válidas, entonces la aplicación emite un JWT para el usuario. Este JWT comúnmente incluye información como identificadores únicos de usuario, roles y permisos.

3. El usuario recibe y almacena el JWT. En los frontends, por ejemplo, el navegador guarda el JWT en una cookie o en el almacenamiento local del navegador.

4. El usuario realiza una solicitud a un endpoint seguro, incluyendo el JWT en el encabezado HTTP Authorization. El valor del encabezado Authorization debe usar el formato Bearer. En consecuencia, cada solicitud utiliza el encabezado Authorization: Bearer TOKEN.

5. La aplicación recibe la solicitud, extrae el JWT del encabezado Authorization y valida la firma del JWT para verificar la autenticidad del usuario.

Construyendo Tokens con SmallRye JWT

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

Configuración:

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

ó JWK

La extensión usa el algoritmo RS256 para firmar los tokens.

Usando el JWT Builder Language

- SmallRye JWT implementa la especificación Eclipse Microprofile Interoperable JWT RBAC.
- La especificación introduce 2 nuevos claims:

upn

Un identificador único del sujeto o del usuario del token, como un ID de usuario. A este usuario se le llama *principal*. **SmallRye JWT** utiliza este *claim* para identificar al usuario principal asociado con el token.

groups

Los grupos a los que pertenece el *principal*. **SmallRye JWT** asigna los valores de este *claim* a roles de seguridad.

```
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
```

Autenticando Requests con SmallRye JWT

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

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

Flujo de Seguridad

- Quarkus usa la interface *io.quarkus.vertx.http.runtime.security.HttpAuthenticationMechanism* para extraer los datos de autenticación desde los requests HTTP y pasa dichos datos al proveedor de identidad.
- Un proveedor de identidad es una instancia de *io.quarkus.security.identity.IdentityProvider* y es responsable de verificar los datos de autenticación y retorna esta data encapsulada en un objeto *SecurityIdentity*.

```
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)
```

```
        if ( principal != null ) {
            return me.getName(); (3)
        }

        ...implementation omitted...
    }

}
```

Configurar SmallRye JWT Authentication

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

Autorizar 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 Autorización 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

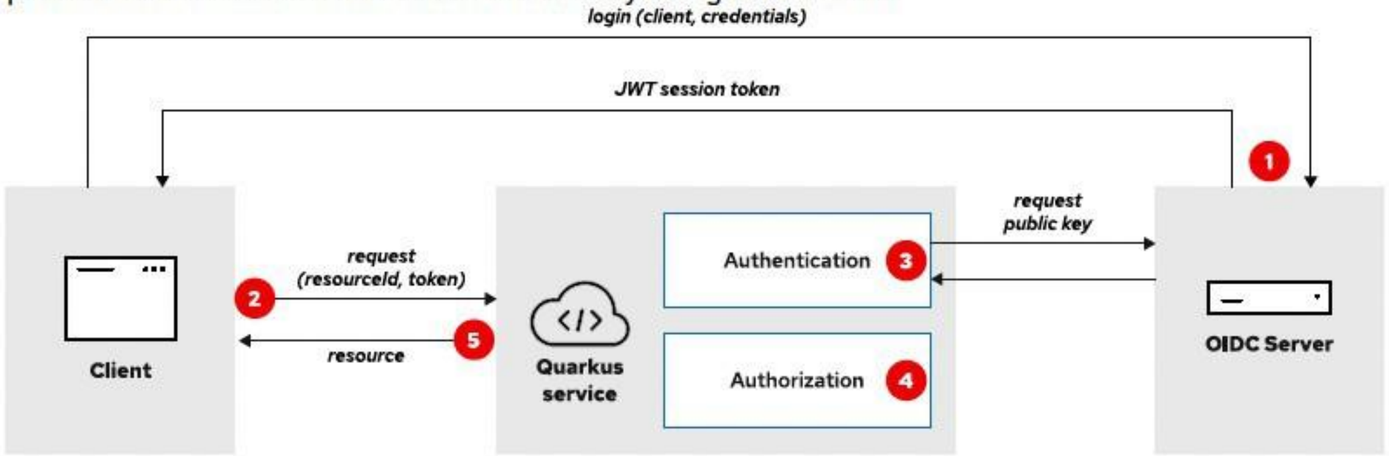
Demo: secure-jwt

Implementando Single Sign-On usando OpenID y OAuth

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.

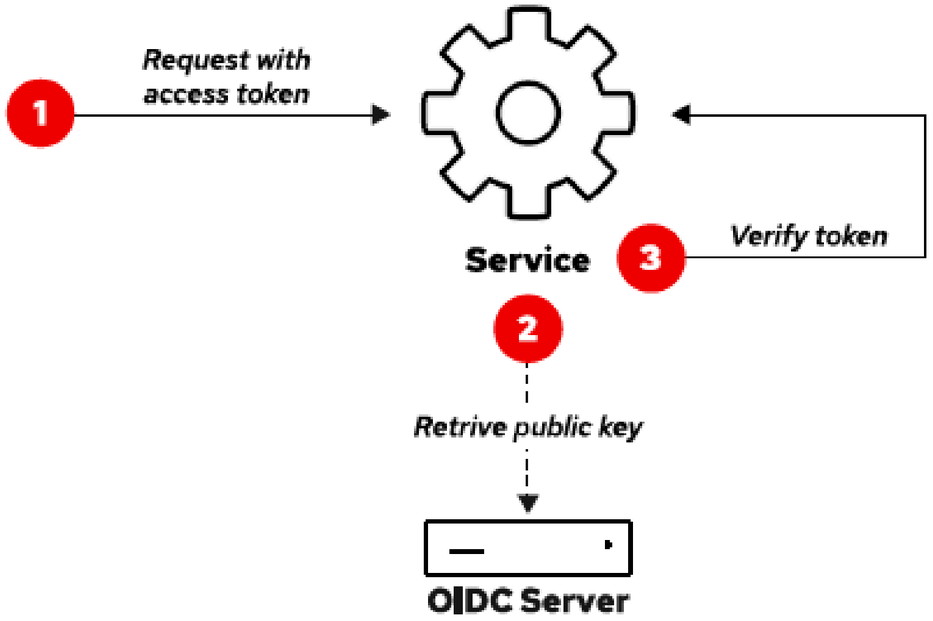


Figure 5.4: Offline signature verification

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

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

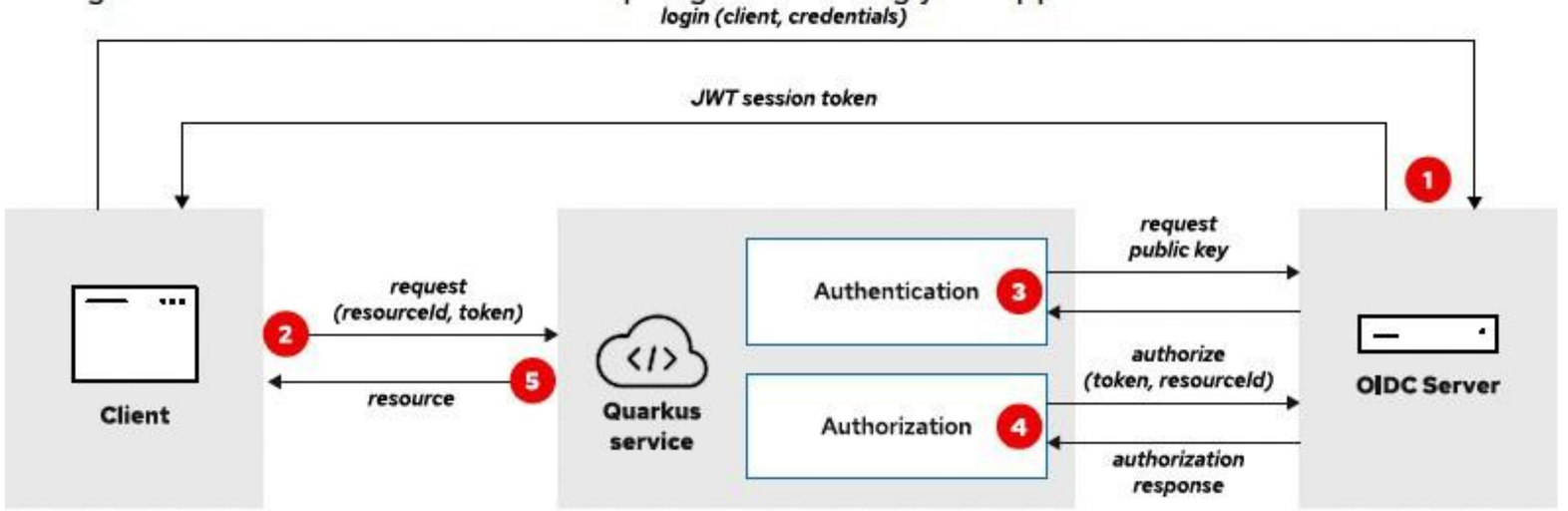
```
quarkus.oidc.auth-server-url=https://SERVER/realms/REALM_NAME (1)
quarkus.oidc.client-id=CLIENT_NAME (2)
quarkus.oidc.credentials.secret=CLIENT_SECRET (3)
```

```
%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
```

```
{
  "allowRemoteResourceManagement": true,
  "policyEnforcementMode": "ENFORCING",
  "resources": [
    {
      "name": "User Resource",
      "ownerManagedAccess": false,
      "attributes": {},
      "_id": "df1b74a9-3f10-499d-a581-368de48e512b",
      "uris": [
        "/api/*"
      ],
      "scopes": [
        {
          "name": "GET"
        }
      ]
    }
  ],
  "policies": [
    {
      "id": "b8710fa6-160e-4de0-adf3-398c7007a0af",
      "name": "Any User Policy",
      "description": "Any user granted with the user role can access this endpoint",
      "type": "role",
      "logic": "POSITIVE",
      "decisionStrategy": "UNANIMOUS",
      "config": {
        "roles": "[{\"id\":\"user\",\"required\":false}]"
      }
    },
    {
      "id": "3479dd56-02e9-4222-94fe-6a13cd065195",
      "name": "User Resource Permission",
      "type": "resource",
      "logic": "POSITIVE",
      "decisionStrategy": "UNANIMOUS",
      "config": {
        "resources": "[\"User Resource\"]",
        "applyPolicies": "[\"Any User Policy\"]"
      }
    }
  ],
  "scopes": [
    {
      "id": "d24e24fe-63b5-4086-83b2-50c184036477",
      "name": "GET",
      "iconUri": ""
    }
  ],
  "decisionStrategy": "UNANIMOUS"
}
```

Demo: secure-sso

Demo: secure-review