Red Hat build of Quarkus 3.8

OpenID Connect (OIDC) authentication

Last Updated: 2025-02-28

Abstract

This guide provides an in-depth look at OpenID Connect (OIDC) authentication, using bearer token authentication to protect service applications, OIDC authorization code flow to protect web applications, and OIDC multitenancy to support multiple OIDC providers and flows.

Table of Contents

PROVIDING FEEDBACK ON RED HAT BUILD OF QUARKUS DOCUMENTATION

To report an error or to improve our documentation, log in to your Red Hat Jira account and submit an issue. If you do not have a Red Hat Jira account, then you will be prompted to create an account.

Procedure

1. Click the following link to **create a ticket**.
2. Enter a brief description of the issue in the **Summary**.
3. Provide a detailed description of the issue or enhancement in the **Description**. Include a URL to where the issue occurs in the documentation.
4. Clicking **Submit** creates and routes the issue to the appropriate documentation team.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

CHAPTER 1. OPENID CONNECT (OIDC) BEARER TOKEN AUTHENTICATION

Secure HTTP access to Jakarta REST (formerly known as JAX-RS) endpoints in your application with Bearer token authentication by using the Quarkus OpenID Connect (OIDC) extension.

1.1. OVERVIEW OF THE BEARER TOKEN AUTHENTICATION MECHANISM IN QUARKUS

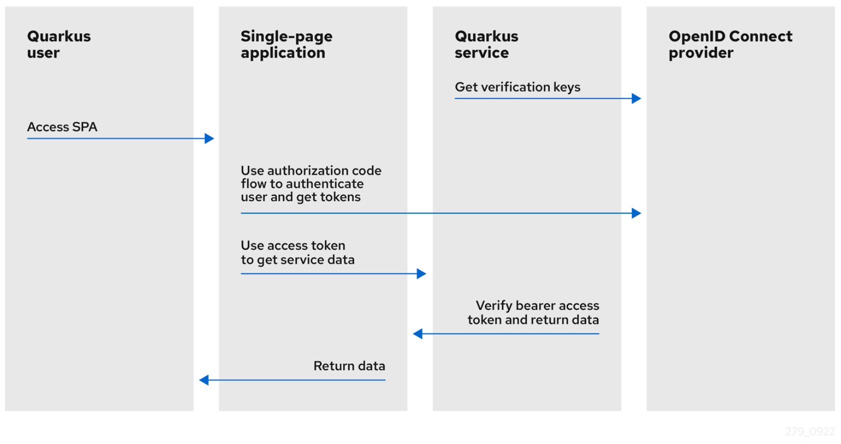
Quarkus supports the Bearer token authentication mechanism through the Quarkus OpenID Connect (OIDC) extension.

The bearer tokens are issued by OIDC and OAuth 2.0 compliant authorization servers, such as Keycloak.

Bearer token authentication is the process of authorizing HTTP requests based on the existence and validity of a bearer token. The bearer token provides information about the subject of the call, which is used to determine whether or not an HTTP resource can be accessed.

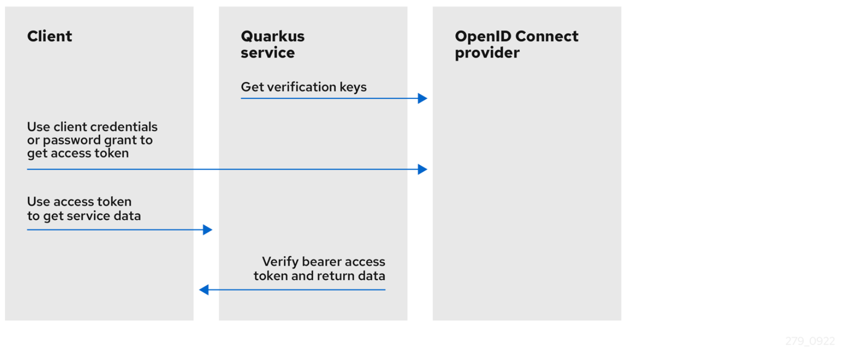
The following diagrams outline the Bearer token authentication mechanism in Quarkus:

Figure 1.1. Bearer token authentication mechanism in Quarkus with single-page application


Quarkus user
Single-page application
Quarkus service
OpenID Connect provider
Get verification keys
Access SPA
Use authorization code flow to authenticate user and get tokens
Use access token to get service data
Verify bearer access token and return data
Return data
279_0922

1. The Quarkus service retrieves verification keys from the OIDC provider. The verification keys are used to verify the bearer access token signatures.
2. The Quarkus user accesses the single-page application (SPA).
3. The single-page application uses Authorization Code Flow to authenticate the user and retrieve tokens from the OIDC provider.
4. The single-page application uses the access token to retrieve the service data from the Quarkus service.
5. The Quarkus service verifies the bearer access token signature by using the verification keys, checks the token expiry date and other claims, allows the request to proceed if the token is valid, and returns the service response to the single-page application.
6. The single-page application returns the same data to the Quarkus user.

Figure 1.2. Bearer token authentication mechanism in Quarkus with Java or command line client


Client
Quarkus service
OpenID Connect provider
Get verification keys
Use client credentials or password grant to get access token
Use access token to get service data
Verify bearer access token and return data
279_0922

1. The Quarkus service retrieves verification keys from the OIDC provider. The verification keys are used to verify the bearer access token signatures.

2. The client uses **client_credentials** that requires client id and secret or password grant, which requires client id, secret, username, and password to retrieve the access token from the OIDC provider.

3. The client uses the access token to retrieve the service data from the Quarkus service.

4. The Quarkus service verifies the bearer access token signature by using the verification keys, checks the token expiry date and other claims, allows the request to proceed if the token is valid, and returns the service response to the client.

If you need to authenticate and authorize users by using OIDC authorization code flow, see the Quarkus OpenID Connect authorization code flow mechanism for protecting web applications guide. Also, if you use Keycloak and bearer tokens, see the Quarkus Using Keycloak to centralize authorization guide.

To learn about how you can protect service applications by using OIDC Bearer token authentication, see the following tutorial: * Protect a web application by using OpenID Connect (OIDC) authorization code flow.

For information about how to support multiple tenants, see the Quarkus Using OpenID Connect Multi-Tenancy guide.

1.1.1. Accessing JWT claims

If you need to access JWT token claims, you can inject **JsonWebToken**:

```
package org.acme.security.openid.connect;

import org.eclipse.microprofile.jwt.JsonWebToken;
import jakarta.inject.Inject;
import jakarta.annotation.security.RolesAllowed;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

@Path("/api/admin")
public class AdminResource {

    @Inject
    JsonWebToken jwt;
    @GET
    @RolesAllowed("admin")
    @Produces(MediaType.TEXT_PLAIN)
    public String admin() {
        return "Access for subject " + jwt.getSubject() + " is granted";
    }
}
```

Injection of **JsonWebToken** is supported in **@ApplicationScoped**, **@Singleton**, and **@RequestScoped** scopes. However, the use of **@RequestScoped** is required if the individual claims are injected as simple types. For more information, see the Supported injection scopes section of the Quarkus "Using JWT RBAC" guide.

1.1.2. UserInfo

If you must request a UserInfo JSON object from the OIDC **UserInfo** endpoint, set **quarkus.oidc.authentication.user-info-required=true**. A request is sent to the OIDC provider **UserInfo** endpoint, and an **io.quarkus.oidc.UserInfo** (a simple **javax.json.JsonObject** wrapper) object is created. **io.quarkus.oidc.UserInfo** can be injected or accessed as a **SecurityIdentity userinfo** attribute.

1.1.3. Configuration metadata

The current tenant's discovered OpenID Connect Configuration Metadata is represented by **io.quarkus.oidc.OidcConfigurationMetadata** and can be injected or accessed as a **SecurityIdentity configuration-metadata** attribute.

The default tenant's **OidcConfigurationMetadata** is injected if the endpoint is public.

1.1.4. Token claims and SecurityIdentity roles

You can map **SecurityIdentity** roles from the verified JWT access tokens as follows:

- If the **quarkus.oidc.roles.role-claim-path** property is set, and matching array or string claims are found, then the roles are extracted from these claims. For example, **customroles**, **customroles/array**, **scope**, **"http://namespace-qualified-custom-claim"/roles**, **"http://namespace-qualified-roles"**.

- If a **groups** claim is available, then its value is used.

- If a **realm_access/roles** or **resource_access/client_id/roles** (where **client_id** is the value of the **quarkus.oidc.client-id** property) claim is available, then its value is used. This check supports the tokens issued by Keycloak.

For example, the following JWT token has a complex **groups** claim that contains a **roles** array that includes roles:

```
{
    "iss": "https://server.example.com",
    "sub": "24400320",
    "upn": "jdoe@example.com",
    "preferred_username": "jdoe",
    "exp": 1311281970,
    "iat": 1311280970,
    "groups": {
      "roles": [
        "microprofile_jwt_user"
      ],
    }
}
```

You must map the **microprofile_jwt_user** role to **SecurityIdentity** roles, and you can do so with this configuration: **quarkus.oidc.roles.role-claim-path=groups/roles**.

If the token is opaque (binary), then a **scope** property from the remote token introspection response is used.

If **UserInfo** is the source of the roles, then set **quarkus.oidc.authentication.user-info-required=true** and **quarkus.oidc.roles.source=userinfo**, and if needed, set **quarkus.oidc.roles.role-claim-path**.

Additionally, a custom **SecurityIdentityAugmentor** can also be used to add the roles. For more information, see the Security identity customization section of the Quarkus "Security tips and tricks" guide.

You can also map **SecurityIdentity** roles created from token claims to deployment-specific roles by using the HTTP Security policy .

1.1.5. Token scopes and SecurityIdentity permissions

SecurityIdentity permissions are mapped in the form of **io.quarkus.security.StringPermission** from the scope parameter of the source of the roles and using the same claim separator.

```
import java.util.List;
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;

import org.eclipse.microprofile.jwt.Claims;
import org.eclipse.microprofile.jwt.JsonWebToken;

import io.quarkus.security.PermissionsAllowed;

@Path("/service")
public class ProtectedResource {

    @Inject
    JsonWebToken accessToken;

    @PermissionsAllowed("email") 1
    @GET
    @Path("/email")
    public Boolean isUserEmailAddressVerifiedByUser() {
        return accessToken.getClaim(Claims.email_verified.name());
    }

    @PermissionsAllowed("orders_read") 2
    @GET
    @Path("/order")
    public List<Order> listOrders() {
        return List.of(new Order(1));
    }

    public static class Order {
        String id;
        public Order() {
        }
        public Order(String id) {
            this.id = id;
        }
        public String getId() {
            return id;
        }
        public void setId() {
            this.id = id;
        }
    }
}
```

1. Only requests with OpenID Connect scope **email** will be granted access.
2. The read access is limited to the client requests with the **orders_read** scope.

For more information about the **io.quarkus.security.PermissionsAllowed** annotation, see the Permission annotation section of the "Authorization of web endpoints" guide.

1.1.6. Token verification and introspection

If the token is a JWT token, then, by default, it is verified with a **JsonWebKey** (JWK) key from a local **JsonWebKeySet**, retrieved from the OIDC provider's JWK endpoint. The token's key identifier (**kid**) header value is used to find the matching JWK key. If no matching **JWK** is available locally, then **JsonWebKeySet** is refreshed by fetching the current key set from the JWK endpoint. The **JsonWebKeySet** refresh can be repeated only after the **quarkus.oidc.token.forced-jwk-refresh-interval** expires. The default expiry time is 10 minutes. If no matching **JWK** is available after the refresh, the JWT token is sent to the OIDC provider's token introspection endpoint.

If the token is opaque, which means it can be a binary token or an encrypted JWT token, then it is always sent to the OIDC provider's token introspection endpoint.

If you work only with JWT tokens and expect a matching **JsonWebKey** to always be available, for example, after refreshing a key set, you must disable token introspection, as shown in the following example:

```
quarkus.oidc.token.allow-jwt-introspection=false
quarkus.oidc.token.allow-opaque-token-introspection=false
```

There might be cases where JWT tokens must be verified through introspection only, which can be forced by configuring an introspection endpoint address only. The following properties configuration shows you an example of how you can achieve this with Keycloak:

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
quarkus.oidc.discovery-enabled=false
# Token Introspection endpoint: http://localhost:8180/realms/quarkus/protocol/openid-
```

```
connect/tokens/introspect
quarkus.oidc.introspection-path=/protocol/openid-connect/tokens/introspect
```

There are advantages and disadvantages to indirectly enforcing the introspection of JWT tokens remotely. An advantage is that you eliminate the need for two remote calls: a remote OIDC metadata discovery call followed by another remote call to fetch the verification keys that will not be used. A disadvantage is that you need to know the introspection endpoint address and configure it manually.

The alternative approach is to allow the default option of OIDC metadata discovery but also require that only the remote JWT introspection is performed, as shown in the following example:

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
quarkus.oidc.token.require-jwt-introspection-only=true
```

An advantage of this approach is that the configuration is simpler and easier to understand. A disadvantage is that a remote OIDC metadata discovery call is required to discover an introspection endpoint address, even though the verification keys will not be fetched.

The **io.quarkus.oidc.TokenIntrospection**, a simple **jakarta.json.JsonObject** wrapper object, will be created. It can be injected or accessed as a **SecurityIdentity introspection** attribute, providing either the JWT or opaque token has been successfully introspected.

1.1.7. Token introspection and UserInfo cache

All opaque access tokens must be remotely introspected. Sometimes, JWT access tokens might also have to be introspected. If **UserInfo** is also required, the same access token is used in a subsequent remote call to the OIDC provider. So, if **UserInfo** is required, and the current access token is opaque, two remote calls are made for every such token; one remote call to introspect the token and another to get **UserInfo**. If the token is JWT, only a single remote call to get **UserInfo** is needed, unless it also has to be introspected.

The cost of making up to two remote calls for every incoming bearer or code flow access token can sometimes be problematic.

If this is the case in production, consider caching the token introspection and **UserInfo** data for a short period, for example, 3 or 5 minutes.

quarkus-oidc provides **quarkus.oidc.TokenIntrospectionCache** and **quarkus.oidc.UserInfoCache** interfaces, usable for **@ApplicationScoped** cache implementation. Use **@ApplicationScoped** cache implementation to store and retrieve **quarkus.oidc.TokenIntrospection** and/or **quarkus.oidc.UserInfo** objects, as outlined in the following example:

```
@ApplicationScoped
@Alternative
@Priority(1)
public class CustomIntrospectionUserInfoCache implements TokenIntrospectionCache,
UserInfoCache {
...
}
```

Each OIDC tenant can either permit or deny the storing of its **quarkus.oidc.TokenIntrospection** data, **quarkus.oidc.UserInfo** data, or both with boolean **quarkus.oidc."tenant".allow-token-introspection-cache** and **quarkus.oidc."tenant".allow-user-info-cache** properties.

Additionally, **quarkus-oidc** provides a simple default memory-based token cache, which implements both **quarkus.oidc.TokenIntrospectionCache** and **quarkus.oidc.UserInfoCache** interfaces.

You can configure and activate the default OIDC token cache as follows:

```
# 'max-size' is 0 by default, so the cache can be activated by setting 'max-size' to a positive value:
quarkus.oidc.token-cache.max-size=1000
# 'time-to-live' specifies how long a cache entry can be valid for and will be used by a cleanup timer:
quarkus.oidc.token-cache.time-to-live=3M
# 'clean-up-timer-interval' is not set by default, so the cleanup timer can be activated by setting 'clean-up-timer-interval':
quarkus.oidc.token-cache.clean-up-timer-interval=1M
```

The default cache uses a token as a key, and each entry can have **TokenIntrospection**, **UserInfo**, or both. It will only keep up to a **max-size** number of entries. If the cache is already full when a new entry is to be added, an attempt is made to find a space by removing a single expired entry. Additionally, the cleanup timer, if activated, periodically checks for expired entries and removes them.

You can experiment with the default cache implementation or register a custom one.

1.1.8. JSON Web Token claim verification

After the bearer JWT token's signature has been verified and its **expires at** (**exp**) claim has been checked, the **iss** (**issuer**) claim value is verified next.

By default, the **iss** claim value is compared to the **issuer** property, which might have been discovered in the well-known provider configuration. However, if the **quarkus.oidc.token.issuer** property is set, then the **iss** claim value is compared to it instead.

In some cases, this **iss** claim verification might not work. For example, if the discovered **issuer** property contains an internal HTTP/IP address while the token **iss** claim value contains an external HTTP/IP address. Or when a discovered **issuer** property contains the template tenant variable, but the token **iss** claim value has the complete tenant-specific issuer value.

In such cases, consider skipping the issuer verification by setting **quarkus.oidc.token.issuer=any**. Only skip the issuer verification if no other options are available:

- If you are using Keycloak and observe the issuer verification errors caused by the different host addresses, configure Keycloak with a **KEYCLOAK_FRONTEND_URL** property to ensure the same host address is used.
- If the **iss** property is tenant-specific in a multitenant deployment, use the **SecurityIdentity tenant-id** attribute to check that the issuer is correct in the endpoint or the custom Jakarta filter. For example:

```
import jakarta.inject.Inject;
import jakarta.ws.rs.container.ContainerRequestContext;
import jakarta.ws.rs.container.ContainerRequestFilter;
import jakarta.ws.rs.core.Response;
import jakarta.ws.rs.ext.Provider;

import org.eclipse.microprofile.jwt.JsonWebToken;
import io.quarkus.oidc.OidcConfigurationMetadata;
import io.quarkus.security.identity.SecurityIdentity;

@Provider
```

```
public class IssuerValidator implements ContainerRequestFilter {
    @Inject
    OidcConfigurationMetadata configMetadata;

    @Inject JsonWebToken jwt;
    @Inject SecurityIdentity identity;

    public void filter(ContainerRequestContext requestContext) {
        String issuer = configMetadata.getIssuer().replace("{tenant-id}", identity.getAttribute("tenant-id"));
        if (!issuer.equals(jwt.getIssuer())) {
            requestContext.abortWith(Response.status(401).build());
        }
    }
}
```

NOTE

Consider using the **quarkus.oidc.token.audience** property to verify the token **aud** (**audience**) claim value.

1.1.9. Single-page applications

A single-page application (SPA) typically uses **XMLHttpRequest**(XHR) and the JavaScript utility code provided by the OIDC provider to acquire a bearer token to access Quarkus **service** applications.

For example, if you work with Keycloak, you can use **keycloak.js** to authenticate users and refresh the expired tokens from the SPA:

```
<html>
<head>
    <title>keycloak-spa</title>
    <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
    <script src="http://localhost:8180/js/keycloak.js"></script>
    <script>
        var keycloak = new Keycloak();
        keycloak.init({onLoad: 'login-required'}).success(function () {
            console.log('User is now authenticated.');
        }).error(function () {
            window.location.reload();
        });
        function makeAjaxRequest() {
            axios.get("/api/hello", {
                headers: {
                    'Authorization': 'Bearer ' + keycloak.token
                }
            })
            .then( function (response) {
                console.log("Response: ", response.status);
            }).catch(function (error) {
                console.log('refreshing');
                keycloak.updateToken(5).then(function () {
                    console.log('Token refreshed');
                }).catch(function () {
                    console.log('Failed to refresh token');
                    window.location.reload();
                });
            });
        }
    </script>
</head>
<body>
    <button onclick="makeAjaxRequest()">Request</button>
</body>
</html>
```

1.1.10. Cross-origin resource sharing

If you plan to use your OIDC **service** application from a single-page application running on a different domain, you must configure cross-origin resource sharing (CORS). For more information, see the CORS filter section of the "Cross-origin resource sharing" guide.

1.1.11. Provider endpoint configuration

An OIDC **service** application needs to know the OIDC provider's token, **JsonWebKey** (JWK) set, and possibly **UserInfo** and introspection endpoint addresses.

By default, they are discovered by adding a **/.well-known/openid-configuration** path to the configured **quarkus.oidc.auth-server-url**.

Alternatively, if the discovery endpoint is not available, or if you want to save on the discovery endpoint round-trip, you can disable the discovery and configure them with relative path values. For example:

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
quarkus.oidc.discovery-enabled=false
# Token endpoint: http://localhost:8180/realms/quarkus/protocol/openid-connect/token
quarkus.oidc.token-path=/protocol/openid-connect/token
# JWK set endpoint: http://localhost:8180/realms/quarkus/protocol/openid-connect/certs
quarkus.oidc.jwks-path=/protocol/openid-connect/certs
# UserInfo endpoint: http://localhost:8180/realms/quarkus/protocol/openid-connect/userinfo
quarkus.oidc.user-info-path=/protocol/openid-connect/userinfo
# Token Introspection endpoint: http://localhost:8180/realms/quarkus/protocol/openid-
connect/tokens/introspect
quarkus.oidc.introspection-path=/protocol/openid-connect/tokens/introspect
```

1.1.12. Token propagation

For information about bearer access token propagation to the downstream services, see the Token propagation section of the Quarkus "OpenID Connect (OIDC) and OAuth2 client and filters reference" guide.

1.1.13. OIDC provider client authentication

quarkus.oidc.runtime.OidcProviderClient is used when a remote request to an OIDC provider is required. If introspection of the Bearer token is necessary, then **OidcProviderClient** must authenticate to the OIDC provider. For more information about supported authentication options, see the OIDC provider client authentication section in the Quarkus "OpenID Connect authorization code flow mechanism for protecting web applications" guide.

1.1.14. Testing

NOTE

If you have to test Quarkus OIDC service endpoints that require Keycloak authorization, follow the Test Keycloak authorization section.

You can begin testing by adding the following dependencies to your test project:

- Using Maven:

```
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-junit5</artifactId>
    <scope>test</scope>
</dependency>
```

- Using Gradle:

```
testImplementation("io.rest-assured:rest-assured")
testImplementation("io.quarkus:quarkus-junit5")
```

1.1.14.1. WireMock

Add the following dependencies to your test project:

- Using Maven:

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

- Using Gradle:

```
testImplementation("io.quarkus:quarkus-test-oidc-server")
```

Prepare the REST test endpoint and set **application.properties**. For example:

```
# keycloak.url is set by OidcWiremockTestResource
quarkus.oidc.auth-server-url=${keycloak.url}/realms/quarkus/
quarkus.oidc.client-id=quarkus-service-app
quarkus.oidc.application-type=service
```

Finally, write the test code. For example:

```
import static org.hamcrest.Matchers.equalTo;
import java.util.Set;

import org.junit.jupiter.api.Test;

import io.quarkus.test.common.QuarkusTestResource;
import io.quarkus.test.junit.QuarkusTest;
import io.quarkus.test.oidc.server.OidcWiremockTestResource;
import io.restassured.RestAssured;
import io.smallrye.jwt.build.Jwt;

@QuarkusTest
@QuarkusTestResource(OidcWiremockTestResource.class)
public class BearerTokenAuthorizationTest {

    @Test
    public void testBearerToken() {
        RestAssured.given().auth().oauth2(getAccessToken("alice", Set.of("user")))
            .when().get("/api/users/me")
            .then()
            .statusCode(200)
            // The test endpoint returns the name extracted from the injected `SecurityIdentity` principal.
            .body("userName", equalTo("alice"));
    }

    private String getAccessToken(String userName, Set<String> groups) {
        return Jwt.preferredUserName(userName)
                .groups(groups)
                .issuer("https://server.example.com")
                .audience("https://service.example.com")
                .sign();
    }
}
```

The **quarkus-test-oidc-server** extension includes a signing RSA private key file in a **JSON Web Key** (**JWK**) format and points to it with a **smallrye.jwt.sign.key.location** configuration property. It allows you to sign the token by using a no-argument **sign()** operation.

Testing your **quarkus-oidc service** application with **OidcWiremockTestResource** provides the best coverage because even the communication channel is tested against the WireMock HTTP stubs. If you need to run a test with WireMock stubs that are not yet supported by **OidcWiremockTestResource**, you can inject a **WireMockServer** instance into the test class, as shown in the following example:

NOTE

OidcWiremockTestResource does not work with **@QuarkusIntegrationTest** against Docker containers because the WireMock server runs in the JVM that runs the test, which is inaccessible from the Docker container that runs the Quarkus application.

```
package io.quarkus.it.keycloak;

import static com.github.tomakehurst.wiremock.client.WireMock.matching;
import static org.hamcrest.Matchers.equalTo;

import org.junit.jupiter.api.Test;
import com.github.tomakehurst.wiremock.WireMockServer;
import com.github.tomakehurst.wiremock.client.WireMock;

import io.quarkus.test.junit.QuarkusTest;
import io.quarkus.test.oidc.server.OidcWireMock;
import io.restassured.RestAssured;

@QuarkusTest
public class CustomOidcWireMockStubTest {

    @OidcWireMock
    WireMockServer wireMockServer;

    @Test
    public void testInvalidBearerToken() {
        wireMockServer.stubFor(WireMock.post("/auth/realms/quarkus/protocol/openid-
connect/token/introspect")
                .withRequestBody(matching(".*token=invalid_token.*"))
                .willReturn(WireMock.aResponse().withStatus(400)));

        RestAssured.given().auth().oauth2("invalid_token").when()
                .get("/api/users/me/bearer")
                .then()
                .statusCode(401)
                .header("WWW-Authenticate", equalTo("Bearer"));
    }
}
```

1.1.15. OidcTestClient

If you use SaaS OIDC providers, such as **Auth0**, and want to run tests against the test (development) domain or to run tests against a remote Keycloak test realm, if you already have **quarkus.oidc.auth-server-url** configured, you can use **OidcTestClient**.

For example, you have the following configuration:

```
%test.quarkus.oidc.auth-server-url=https://dev-123456.eu.auth0.com/
%test.quarkus.oidc.client-id=test-auth0-client
%test.quarkus.oidc.credentials.secret=secret
```

To start, add the same dependency, **quarkus-test-oidc-server**, as described in the WireMock section.

Next, write the test code as follows:

```
package org.acme;

import org.junit.jupiter.api.AfterAll;
import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;

import java.util.Map;

import org.junit.jupiter.api.Test;
import io.quarkus.test.junit.QuarkusTest;
import io.quarkus.test.oidc.client.OidcTestClient;

@QuarkusTest
public class GreetingResourceTest {

    static OidcTestClient oidcTestClient = new OidcTestClient();

    @AfterAll
    public static void close() {
        oidcTestClient.close();
    }

    @Test
    public void testHelloEndpoint() {
        given()
          .auth().oauth2(getAccessToken("alice", "alice"))
          .when().get("/hello")
          .then()
            .statusCode(200)
            .body(is("Hello, Alice"));
    }

    private String getAccessToken(String name, String secret) {
        return oidcTestClient.getAccessToken(name, secret,
            Map.of("audience", "https://dev-123456.eu.auth0.com/api/v2/",
            "scope", "profile"));
    }
}
```

This test code acquires a token by using a **password** grant from the test **Auth0** domain, which has registered an application with the client id **test-auth0-client**, and created the user **alice** with password **alice**. For a test like this to work, the test **Auth0** application must have the **password** grant enabled. This example code also shows how to pass additional parameters. For **Auth0**, these are the **audience** and **scope** parameters.

1.1.15.1. Dev Services for Keycloak

The preferred approach for integration testing against Keycloak is Dev Services for Keycloak. **Dev Services for Keycloak** will start and initialize a test container. Then, it will create a **quarkus** realm and a **quarkus-app** client (**secret** secret) and add **alice** (**admin** and **user** roles) and **bob** (**user** role) users, where all of these properties can be customized.

First, add the following dependency, which provides a utility class **io.quarkus.test.keycloak.client.KeycloakTestClient** that you can use in tests for acquiring the access tokens:

- Using Maven:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-test-keycloak-server</artifactId>
    <scope>test</scope>
</dependency>
```

- Using Gradle:

  ```
  testImplementation("io.quarkus:quarkus-test-keycloak-server")
  ```

Next, prepare your **application.properties** configuration file. You can start with an empty **application.properties** file because **Dev Services for Keycloak** registers **quarkus.oidc.auth-server-url** and points it to the running test container, **quarkus.oidc.client-id=quarkus-app**, and **quarkus.oidc.credentials.secret=secret**.

However, if you have already configured the required **quarkus-oidc** properties, then you only need to associate **quarkus.oidc.auth-server-url** with the **prod** profile for `Dev Services for Keycloak`to start a container, as shown in the following example:

```
%prod.quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
```

If a custom realm file has to be imported into Keycloak before running the tests, configure **Dev Services for Keycloak** as follows:

```
%prod.quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
quarkus.keycloak.devservices.realm-path=quarkus-realm.json
```

Finally, write your test, which will be executed in JVM mode, as shown in the following examples:

Example of a test executed in JVM mode:

```
package org.acme.security.openid.connect;

import io.quarkus.test.junit.QuarkusTest;
import io.quarkus.test.keycloak.client.KeycloakTestClient;
import io.restassured.RestAssured;
import org.junit.jupiter.api.Test;

@QuarkusTest
public class BearerTokenAuthenticationTest {

    KeycloakTestClient keycloakClient = new KeycloakTestClient();

    @Test
    public void testAdminAccess() {
        RestAssured.given().auth().oauth2(getAccessToken("alice"))
                .when().get("/api/admin")
                .then()
                .statusCode(200);
        RestAssured.given().auth().oauth2(getAccessToken("bob"))
                .when().get("/api/admin")
                .then()
                .statusCode(403);
    }

    protected String getAccessToken(String userName) {
        return keycloakClient.getAccessToken(userName);
    }
}
```

Example of a test executed in native mode:

```java
package org.acme.security.openid.connect;

import io.quarkus.test.junit.QuarkusIntegrationTest;

@QuarkusIntegrationTest
public class NativeBearerTokenAuthenticationIT extends BearerTokenAuthenticationTest {
}
```

For more information about initializing and configuring Dev Services for Keycloak, see the Dev Services for Keycloak guide.

1.1.15.2. Local public key

You can use a local inlined public key for testing your **quarkus-oidc service** applications, as shown in the following example:

```properties
quarkus.oidc.client-id=test
quarkus.oidc.public-
key=MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAlivFI8qB4D0y2jy0CfEqFyy46R0o7S8T
Kpsx5xbHKoU1VWg6QkQm+ntyIv1p4kE1sPEQO73+HY8+Bzs75XwRTYL1BmR1w8J5hmjVWjc6R2BT
BGAYRPFRhor3kpM6ni2SPmNNhurEAHw7TaqszP5eUF/F9+KEBWkwVta+PZ37bwqSE4sCb1soZFrV
z/UT/LF4tYpuVYt3YbqToZ3pZOZ9AX2o1GCG3xwOjkc4x0W7ezbQZdC9iftPxVHR8irOijJRRjcPDtA6vP
KpzLl6CyYnsIYPd99ltwxTHjr3npfv/3Lw50bAkbT4HeLFxTx4flEoZLKO/g0bAoV2uqBhkA9xnQIDAQAB


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

To generate JWT tokens, copy **privateKey.pem** from the **integration-tests/oidc-tenancy** in the **main** Quarkus repository and use a test code similar to the one in the preceding WireMock section. You can use your own test keys, if preferred.

This approach provides limited coverage compared to the WireMock approach. For example, the remote communication code is not covered.

1.1.15.3. TestSecurity annotation

You can use **@TestSecurity** and **@OidcSecurity** annotations to test the **service** application endpoint code, which depends on either one, or all three, of the following injections:

- **JsonWebToken**
- **UserInfo**
- **OidcConfigurationMetadata**

First, add the following dependency:

- Using Maven:

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

```
    <scope>test</scope>
</dependency>
```

- Using Gradle:

```
testImplementation("io.quarkus:quarkus-test-security-oidc")
```

Write a test code as outlined in the following example:

```
import static org.hamcrest.Matchers.is;
import org.junit.jupiter.api.Test;
import io.quarkus.test.common.http.TestHTTPEndpoint;
import io.quarkus.test.junit.QuarkusTest;
import io.quarkus.test.security.TestSecurity;
import io.quarkus.test.security.oidc.Claim;
import io.quarkus.test.security.oidc.ConfigMetadata;
import io.quarkus.test.security.oidc.OidcSecurity;
import io.quarkus.test.security.oidc.OidcConfigurationMetadata;
import io.quarkus.test.security.oidc.UserInfo;
import io.restassured.RestAssured;

@QuarkusTest
@TestHTTPEndpoint(ProtectedResource.class)
public class TestSecurityAuthTest {

    @Test
    @TestSecurity(user = "userOidc", roles = "viewer")
    public void testOidc() {
        RestAssured.when().get("test-security-oidc").then()
                .body(is("userOidc:viewer"));
    }

    @Test
    @TestSecurity(user = "userOidc", roles = "viewer")
    @OidcSecurity(claims = {
            @Claim(key = "email", value = "user@gmail.com")
    }, userinfo = {
            @UserInfo(key = "sub", value = "subject")
    }, config = {
            @ConfigMetadata(key = "issuer", value = "issuer")
    })
    public void testOidcWithClaimsUserInfoAndMetadata() {
        RestAssured.when().get("test-security-oidc-claims-userinfo-metadata").then()
                .body(is("userOidc:viewer:user@gmail.com:subject:issuer"));
    }

}
```

The **ProtectedResource** class, which is used in this code example, might look like this:

```
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;

import io.quarkus.oidc.OidcConfigurationMetadata;
import io.quarkus.oidc.UserInfo;
import io.quarkus.security.Authenticated;

import org.eclipse.microprofile.jwt.JsonWebToken;

@Path("/service")
@Authenticated
public class ProtectedResource {

    @Inject
    JsonWebToken accessToken;
    @Inject
    UserInfo userInfo;
    @Inject
    OidcConfigurationMetadata configMetadata;

    @GET
    @Path("test-security-oidc")
    public String testSecurityOidc() {
        return accessToken.getName() + ":" + accessToken.getGroups().iterator().next();
    }

    @GET
    @Path("test-security-oidc-claims-userinfo-metadata")
    public String testSecurityOidcWithClaimsUserInfoMetadata() {
        return accessToken.getName() + ":" + accessToken.getGroups().iterator().next()
                + ":" + accessToken.getClaim("email")
                + ":" + userInfo.getString("sub")
                + ":" + configMetadata.get("issuer");
    }
}
```

You must always use the **@TestSecurity** annotation. Its **user** property is returned as **JsonWebToken.getName()** and its **roles** property is returned as **JsonWebToken.getGroups()**. The **@OidcSecurity** annotation is optional and you can use it to set the additional token claims and the **UserInfo** and **OidcConfigurationMetadata** properties. Additionally, if the **quarkus.oidc.token.issuer** property is configured, it is used as an **OidcConfigurationMetadata issuer** property value.

If you work with opaque tokens, you can test them as shown in the following code example:

```
import static org.hamcrest.Matchers.is;
import org.junit.jupiter.api.Test;
import io.quarkus.test.common.http.TestHTTPEndpoint;
import io.quarkus.test.junit.QuarkusTest;
import io.quarkus.test.security.TestSecurity;
import io.quarkus.test.security.oidc.OidcSecurity;
import io.quarkus.test.security.oidc.TokenIntrospection;
import io.restassured.RestAssured;

@QuarkusTest
@TestHTTPEndpoint(ProtectedResource.class)
public class TestSecurityAuthTest {

    @Test
    @TestSecurity(user = "userOidc", roles = "viewer")
    @OidcSecurity(introspectionRequired = true,
            introspection = {
                @TokenIntrospection(key = "email", value = "user@gmail.com")
            }
        )
        public void testOidcWithClaimsUserInfoAndMetadata() {
            RestAssured.when().get("test-security-oidc-claims-userinfo-metadata").then()
                    .body(is("userOidc:viewer:userOidc:viewer"));
        }

    }
```

The **ProtectedResource** class, which is used in this code example, might look like this:

```
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;

import io.quarkus.oidc.TokenIntrospection;
import io.quarkus.security.Authenticated;
import io.quarkus.security.identity.SecurityIdentity;

@Path("/service")
@Authenticated
public class ProtectedResource {

    @Inject
    SecurityIdentity securityIdentity;
    @Inject
    TokenIntrospection introspection;

    @GET
    @Path("test-security-oidc-opaque-token")
    public String testSecurityOidcOpaqueToken() {
        return securityIdentity.getPrincipal().getName() + ":" + securityIdentity.getRoles().iterator().next()
            + ":" + introspection.getString("username")
            + ":" + introspection.getString("scope")
            + ":" + introspection.getString("email");
    }
}
```

The **@TestSecurity**, **user**, and **roles** attributes are available as **TokenIntrospection**, **username**, and **scope** properties. Use **io.quarkus.test.security.oidc.TokenIntrospection** to add the additional introspection response properties, such as an **email**, and so on.

TIP

@TestSecurity and **@OidcSecurity** can be combined in a meta-annotation, as outlined in the following example:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.METHOD })
@TestSecurity(user = "userOidc", roles = "viewer")
@OidcSecurity(introspectionRequired = true,
    introspection = {
        @TokenIntrospection(key = "email", value = "user@gmail.com")
    }
)
public @interface TestSecurityMetaAnnotation {

}
```

This is particularly useful if multiple test methods must use the same set of security settings.

1.1.16. Check errors in the logs

To see more details about token verification errors, enable **io.quarkus.oidc.runtime.OidcProvider** and **TRACE** level logging:

```
quarkus.log.category."io.quarkus.oidc.runtime.OidcProvider".level=TRACE
quarkus.log.category."io.quarkus.oidc.runtime.OidcProvider".min-level=TRACE
```

To see more details about **OidcProvider** client initialization errors, enable **io.quarkus.oidc.runtime.OidcRecorder** and **TRACE** level logging as follows:

```
quarkus.log.category."io.quarkus.oidc.runtime.OidcRecorder".level=TRACE
quarkus.log.category."io.quarkus.oidc.runtime.OidcRecorder".min-level=TRACE
```

1.1.17. External and internal access to OIDC providers

The externally-accessible token of the OIDC provider and other endpoints might have different HTTP(S) URLs compared to the URLs that are auto-discovered or configured relative to the **quarkus.oidc.auth-server-url** internal URL. For example, suppose your SPA acquires a token from an external token endpoint address and sends it to Quarkus as a bearer token. In that case, the endpoint might report an issuer verification failure.

In such cases, if you work with Keycloak, start it with the **KEYCLOAK_FRONTEND_URL** system property set to the externally accessible base URL. If you work with other OIDC providers, refer to your provider's documentation.

1.1.18. Using the client-id property

The **quarkus.oidc.client-id** property identifies the OIDC client that requested the current bearer token. The OIDC client can be an SPA application running in a browser or a Quarkus **web-app** confidential client application propagating the access token to the Quarkus **service** application.

This property is required if the **service** application is expected to introspect the tokens remotely, which is always the case for the opaque tokens. This property is optional for local JSON Web Token (JWT) verification only.

Setting the **quarkus.oidc.client-id** property is encouraged even if the endpoint does not require access to the remote introspection endpoint. This is because when **client-id** is set, it can be used to verify the token audience. It will also be included in logs when the token verification fails, enabling better traceability of tokens issued to specific clients and analysis over a longer period.

For example, if your OIDC provider sets a token audience, consider the following configuration pattern:

```
# Set client-id
quarkus.oidc.client-id=quarkus-app
# Token audience claim must contain 'quarkus-app'
quarkus.oidc.token.audience=${quarkus.oidc.client-id}
```

If you set **quarkus.oidc.client-id**, but your endpoint does not require remote access to one of the OIDC provider endpoints (introspection, token acquisition, and so on), do not set a client secret with **quarkus.oidc.credentials** or similar properties because it will not be used.

NOTE

Quarkus **web-app** applications always require the **quarkus.oidc.client-id** property.

1.2. AUTHENTICATION AFTER AN HTTP REQUEST HAS COMPLETED

Sometimes, **SecurityIdentity** for a given token must be created when there is no active HTTP request context. The **quarkus-oidc** extension provides **io.quarkus.oidc.TenantIdentityProvider** to convert a token to a **SecurityIdentity** instance. For example, one situation when you must verify the token after the HTTP request has completed is when you are processing messages with Vert.x event bus. The example below uses the 'product-order' message within different CDI request contexts. Therefore, an injected **SecurityIdentity** would not correctly represent the verified identity and be anonymous.

```
package org.acme.quickstart.oidc;

import static jakarta.ws.rs.core.HttpHeaders.AUTHORIZATION;

import jakarta.inject.Inject;
import jakarta.ws.rs.HeaderParam;
import jakarta.ws.rs.POST;
import jakarta.ws.rs.Path;
import io.vertx.core.eventbus.EventBus;

@Path("order")
public class OrderResource {

    @Inject
    EventBus eventBus;

    @POST
    public void order(String product, @HeaderParam(AUTHORIZATION) String bearer) {
        String rawToken = bearer.substring("Bearer ".length()); 1
        eventBus.publish("product-order", new Product(product, rawToken));
    }

    public static class Product {
         public String product;
         public String customerAccessToken;
        public Product() {
        }
        public Product(String product, String customerAccessToken) {
            this.product = product;
            this.customerAccessToken = customerAccessToken;
        }
    }
}
```

1 At this point, the token is not verified when proactive authentication is disabled.

```
package org.acme.quickstart.oidc;

import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;

import io.quarkus.oidc.AccessTokenCredential;
import io.quarkus.oidc.TenantFeature;
import io.quarkus.oidc.TenantIdentityProvider;
import io.quarkus.security.identity.SecurityIdentity;
import io.quarkus.vertx.ConsumeEvent;
import io.smallrye.common.annotation.Blocking;

@ApplicationScoped
public class OrderService {

    @TenantFeature("tenantId")
    @Inject
    TenantIdentityProvider identityProvider;

    @Inject
    TenantIdentityProvider defaultIdentityProvider; 1

    @Blocking
    @ConsumeEvent("product-order")
    void processOrder(Product product) {
        AccessTokenCredential tokenCredential = new
AccessTokenCredential(product.customerAccessToken);
        SecurityIdentity securityIdentity =
identityProvider.authenticate(tokenCredential).await().indefinitely(); 2
        ...
    }

}
```

1 For the default tenant, the **TenantFeature** qualifier is optional.

2 Executes token verification and converts the token to a **SecurityIdentity**.

NOTE

When the provider is used during an HTTP request, the tenant configuration can be resolved as described in the Using OpenID Connect Multi-Tenancy guide. However, when there is no active HTTP request, you must select the tenant explicitly with the **io.quarkus.oidc.TenantFeature** qualifier.

WARNING

Dynamic tenant configuration resolution is currently not supported. Authentication that requires a dynamic tenant will fail.

1.3. OIDC REQUEST FILTERS

You can filter OIDC requests made by Quarkus to the OIDC provider by registering one or more **OidcRequestFilter** implementations, which can update or add new request headers, and log requests. For more information, see OIDC request filters.

1.4. REFERENCES

- OIDC configuration properties
- Protect a service application by using OIDC Bearer token authentication
- Keycloak documentation
- OpenID Connect
- JSON Web Token
- OpenID Connect and OAuth2 client and filters reference guide
- Dev Services for Keycloak
- Sign and encrypt JWT tokens with SmallRye JWT Build
- Choosing between OpenID Connect, SmallRye JWT, and OAuth2 authentication mechanisms
- Combining authentication mechanisms
- Quarkus Security overview
- Using OpenID Connect Multi-Tenancy

CHAPTER 2. PROTECT A SERVICE APPLICATION BY USING OPENID CONNECT (OIDC) BEARER TOKEN AUTHENTICATION

Use the Quarkus OpenID Connect (OIDC) extension to secure a Jakarta REST application with Bearer token authentication. The bearer tokens are issued by OIDC and OAuth 2.0 compliant authorization servers, such as Keycloak.

For more information about OIDC Bearer token authentication, see the Quarkus OpenID Connect (OIDC) Bearer token authentication guide.

If you want to protect web applications by using OIDC Authorization Code Flow authentication, see the OpenID Connect authorization code flow mechanism for protecting web applications guide.

2.1. PREREQUISITES

To complete this guide, you need:

- Roughly 15 minutes
- An IDE
- JDK 17+ installed with **JAVA_HOME** configured appropriately
- Apache Maven 3.9.6
- A working container runtime (Docker or Podman)
- Optionally the Quarkus CLI if you want to use it
- Optionally Mandrel or GraalVM installed and configured appropriately if you want to build a native executable (or Docker if you use a native container build)
- The jq command-line processor tool

2.2. ARCHITECTURE

This example shows how you can build a simple microservice that offers two endpoints:

- **/api/users/me**
- **/api/admin**

These endpoints are protected and can only be accessed if a client sends a bearer token along with the request, which must be valid (for example, signature, expiration, and audience) and trusted by the microservice.

A Keycloak server issues the bearer token and represents the subject for which the token was issued. Because it is an OAuth 2.0 authorization server, the token also references the client acting on the user's behalf.

Any user with a valid token can access the **/api/users/me** endpoint. As a response, it returns a JSON document with user details obtained from the information in the token.

The **/api/admin** endpoint is protected with RBAC (Role-Based Access Control), which only users with the **admin** role can access. At this endpoint, the **@RolesAllowed** annotation is used to enforce the access constraint declaratively.

2.3. SOLUTION

Follow the instructions in the next sections and create the application step by step. You can also go straight to the completed example.

You can clone the Git repository by running the command **git clone https://github.com/quarkusio/quarkus-quickstarts.git -b 3.8**, or you can download an archive.

The solution is located in the **security-openid-connect-quickstart** directory.

2.4. CREATE THE MAVEN PROJECT

You can either create a new Maven project with the **oidc** extension or you can add the extension to an existing Maven project. Complete one of the following commands:

To create a new Maven project, use the following command:

- Using the Quarkus CLI:

  ```
  quarkus create app org.acme:security-openid-connect-quickstart \
      --extension='oidc,resteasy-reactive-jackson' \
      --no-code
  cd security-openid-connect-quickstart
  ```

  To create a Gradle project, add the **--gradle** or **--gradle-kotlin-dsl** option.

  For more information about how to install and use the Quarkus CLI, see the Quarkus CLI guide.

- Using Maven:

  ```
  mvn io.quarkus.platform:quarkus-maven-plugin:3.8.5:create \
      -DprojectGroupId=org.acme \
      -DprojectArtifactId=security-openid-connect-quickstart \
      -Dextensions='oidc,resteasy-reactive-jackson' \
      -DnoCode
  cd security-openid-connect-quickstart
  ```

  To create a Gradle project, add the **-DbuildTool=gradle** or **-DbuildTool=gradle-kotlin-dsl** option.

For Windows users:

- If using cmd, (don’t use backward slash \ and put everything on the same line)
- If using Powershell, wrap **-D** parameters in double quotes e.g. **"-DprojectArtifactId=security-openid-connect-quickstart"**

If you already have your Quarkus project configured, you can add the **oidc** extension to your project by running the following command in your project base directory:

- Using the Quarkus CLI:

```
quarkus extension add oidc
```

- Using Maven:

```
./mvnw quarkus:add-extension -Dextensions='oidc'
```

- Using Gradle:

```
./gradlew addExtension --extensions='oidc'
```

This will add the following to your build file:

- Using Maven:

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

- Using Gradle:

```
implementation("io.quarkus:quarkus-oidc")
```

2.5. WRITE THE APPLICATION

1. Implement the **/api/users/me** endpoint as shown in the following example, which is a regular Jakarta REST resource:

```
package org.acme.security.openid.connect;

import jakarta.annotation.security.RolesAllowed;
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;

import org.jboss.resteasy.reactive.NoCache;
import io.quarkus.security.identity.SecurityIdentity;

@Path("/api/users")
public class UsersResource {

    @Inject
    SecurityIdentity securityIdentity;

    @GET
    @Path("/me")
    @RolesAllowed("user")
    @NoCache
    public User me() {
        return new User(securityIdentity);
    }

    public static class User {
        private final String userName;

        User(SecurityIdentity securityIdentity) {
            this.userName = securityIdentity.getPrincipal().getName();
        }

        public String getUserName() {
            return userName;
        }
    }
}
```

2. Implement the **/api/admin** endpoint as shown in the following example:

```
package org.acme.security.openid.connect;

import jakarta.annotation.security.RolesAllowed;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

@Path("/api/admin")
public class AdminResource {

    @GET
    @RolesAllowed("admin")
    @Produces(MediaType.TEXT_PLAIN)
    public String admin() {
        return "granted";
    }
}
```

NOTE

The main difference in this example is that the **@RolesAllowed** annotation is used to verify that only users granted the **admin** role can access the endpoint.

Injection of the **SecurityIdentity** is supported in both **@RequestScoped** and **@ApplicationScoped** contexts.

2.6. CONFIGURE THE APPLICATION

- Configure the Quarkus OpenID Connect (OIDC) extension by setting the following configuration properties in the **src/main/resources/application.properties** file.

```
%prod.quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
quarkus.oidc.client-id=backend-service
quarkus.oidc.credentials.secret=secret

# Tell Dev Services for Keycloak to import the realm file
# This property is not effective when running the application in JVM or native modes

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

Where:

- **%prod.quarkus.oidc.auth-server-url** sets the base URL of the OpenID Connect (OIDC) server. The **%prod.** profile prefix ensures that **Dev Services for Keycloak** launches a container when you run the application in development (dev) mode. For more information, see the Run the application in dev mode section.

- **quarkus.oidc.client-id** sets a client id that identifies the application.

- **quarkus.oidc.credentials.secret** sets the client secret, which is used by the **client_secret_basic** authentication method.

For more information, see the Quarkus OpenID Connect (OIDC) configuration properties guide.

2.7. START AND CONFIGURE THE KEYCLOAK SERVER

1. Put the realm configuration file on the classpath (**target/classes** directory) so that it gets imported automatically when running in dev mode. You do not need to do this if you have already built a complete solution, in which case, this realm file is added to the classpath during the build.

   NOTE

   Do not start the Keycloak server when you run the application in dev mode; **Dev Services for Keycloak** will start a container. For more information, see the Run the application in dev mode section.

2. To start a Keycloak server, you can use Docker to run the following command:

   ```
   docker run --name keycloak -e KEYCLOAK_ADMIN=admin -e
   KEYCLOAK_ADMIN_PASSWORD=admin -p 8180:8080 quay.io/keycloak/keycloak:
   {keycloak.version} start-dev
   ```

   - Where the **keycloak.version** is set to version **24.0.0** or later.

3. You can access your Keycloak server at localhost:8180.

4. To access the Keycloak Administration console, log in as the **admin** user by using the following login credentials:

   - Username: **admin**
   - Password: **admin**

5. Import the realm configuration file from the upstream community repository to create a new realm.

For more information, see the Keycloak documentation about creating and configuring a new realm.

2.8. RUN THE APPLICATION IN DEV MODE

1. To run the application in dev mode, run the following commands:
   - Using the Quarkus CLI:

     ```
     quarkus dev
     ```

   - Using Maven:

     ```
     ./mvnw quarkus:dev
     ```

   - Using Gradle:

     ```
     ./gradlew --console=plain quarkusDev
     ```

   - Dev Services for Keycloak will start a Keycloak container and import a **quarkus-realm.json**.
2. Open a Dev UI, which you can find at /q/dev-ui. Then, in an **OpenID Connect** card, click the **Keycloak provider** link .
3. When prompted to log in to a **Single Page Application** provided by **OpenID Connect Dev UI**, do the following steps:
   - Log in as **alice** (password: **alice**), who has a **user** role.
     - Accessing **/api/admin** returns a **403** status code.
     - Accessing **/api/users/me** returns a **200** status code.
   - Log out and log in again as **admin** (password: **admin**), who has both **admin** and **user** roles.
     - Accessing **/api/admin** returns a **200** status code.
     - Accessing **/api/users/me** returns a **200** status code.

2.9. RUN THE APPLICATION IN JVM MODE

When you are done with dev mode, you can run the application as a standard Java application.

1. Compile the application:
   - Using the Quarkus CLI:

     ```
     quarkus build
     ```

   - Using Maven:

     ```
     ./mvnw install
     ```

   - Using Gradle:

     ```
     ./gradlew build
     ```

2. Run the application:

```
java -jar target/quarkus-app/quarkus-run.jar
```

2.10. RUN THE APPLICATION IN NATIVE MODE

You can compile this same demo as-is into native mode without any modifications. This implies that you no longer need to install a JVM on your production environment. The runtime technology is included in the produced binary and optimized to run with minimal resources required.

Compilation takes a bit longer, so this step is disabled by default.

1. Build your application again by enabling the **native** profile:
   - Using the Quarkus CLI:

     ```
     quarkus build --native
     ```

   - Using Maven:

     ```
     ./mvnw install -Dnative
     ```

   - Using Gradle:

     ```
     ./gradlew build -Dquarkus.package.type=native
     ```

2. After waiting a little while, you run the following binary directly:

   ```
   ./target/security-openid-connect-quickstart-1.0.0-SNAPSHOT-runner
   ```

2.11. TEST THE APPLICATION

For information about testing your application in dev mode, see the preceding Run the application in dev mode section.

You can test the application launched in JVM or native modes with **curl**.

- Because the application uses Bearer token authentication, you must first obtain an access token from the Keycloak server to access the application resources:

```
export access_token=$(\
    curl --insecure -X POST http://localhost:8180/realms/quarkus/protocol/openid-connect/token \
    --user backend-service:secret \
    -H 'content-type: application/x-www-form-urlencoded' \
    -d 'username=alice&password=alice&grant_type=password' | jq --raw-output '.access_token' \
 )
```

The preceding example obtains an access token for the user **alice**.

- Any user can access the **http://localhost:8080/api/users/me** endpoint, which returns a JSON payload with details about the user.

```
curl -v -X GET \
  http://localhost:8080/api/users/me \
  -H "Authorization: Bearer "$access_token
```

- Only users with the **admin** role can access the **http://localhost:8080/api/admin** endpoint. If you try to access this endpoint with the previously-issued access token, you get a **403** response from the server.

```
curl -v -X GET \
   http://localhost:8080/api/admin \
   -H "Authorization: Bearer "$access_token
```

- To access the admin endpoint, obtain a token for the **admin** user:

```
export access_token=$(\
    curl --insecure -X POST http://localhost:8180/realms/quarkus/protocol/openid-connect/token \
    --user backend-service:secret \
    -H 'content-type: application/x-www-form-urlencoded' \
    -d 'username=admin&password=admin&grant_type=password' | jq --raw-output '.access_token' \
 )
```

For information about writing integration tests that depend on **Dev Services for Keycloak**, see the Dev Services for Keycloak section of the "OpenID Connect (OIDC) Bearer token authentication" guide.

2.12. REFERENCES

- OIDC configuration properties
- OpenID Connect (OIDC) Bearer token authentication
- Keycloak Documentation
- OpenID Connect
- JSON Web Token
- OpenID Connect and OAuth2 Client and Filters Reference Guide
- Dev Services for Keycloak
- Sign and encrypt JWT tokens with SmallRye JWT Build
- Combining authentication mechanisms
- Quarkus Security overview

CHAPTER 3. OPENID CONNECT AUTHORIZATION CODE FLOW MECHANISM FOR PROTECTING WEB APPLICATIONS

To protect your web applications, you can use the industry-standard OpenID Connect (OIDC) Authorization Code Flow mechanism provided by the Quarkus OIDC extension.

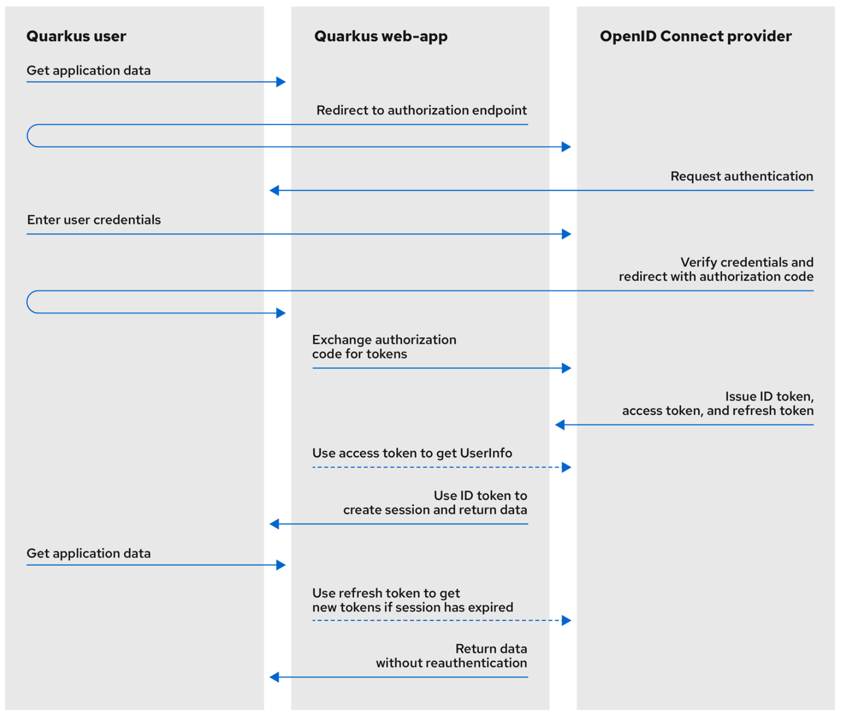
3.1. OVERVIEW OF THE OIDC AUTHORIZATION CODE FLOW MECHANISM

The Quarkus OpenID Connect (OIDC) extension can protect application HTTP endpoints by using the OIDC Authorization Code Flow mechanism supported by OIDC-compliant authorization servers, such as Keycloak.

The Authorization Code Flow mechanism authenticates users of your web application by redirecting them to an OIDC provider, such as Keycloak, to log in. After authentication, the OIDC provider redirects the user back to the application with an authorization code that confirms that authentication was successful. Then, the application exchanges this code with the OIDC provider for an ID token (which represents the authenticated user), an access token, and a refresh token to authorize the user's access to the application.

The following diagram outlines the Authorization Code Flow mechanism in Quarkus.

Figure 3.1. Authorization code flow mechanism in Quarkus


Quarkus user
Quarkus web-app
OpenID Connect provider
Get application data
Redirect to authorization endpoint
Request authentication
Enter user credentials
Verify credentials and redirect with authorization code
Exchange authorization code for tokens
Issue ID token, access token, and refresh token
Use access token to get UserInfo
Use ID token to create session and return data
Get application data
Use refresh token to get new tokens if session has expired
Return data without reauthentication

1. The Quarkus user requests access to a Quarkus **web-app** application.
2. The Quarkus web-app redirects the user to the authorization endpoint, that is, the OIDC provider for authentication.
3. The OIDC provider redirects the user to a login and authentication prompt.
4. At the prompt, the user enters their user credentials.

5. The OIDC provider authenticates the user credentials entered and, if successful, issues an authorization code and redirects the user back to the Quarkus web-app with the code included as a query parameter.

6. The Quarkus web-app exchanges this authorization code with the OIDC provider for ID, access, and refresh tokens.

The authorization code flow is completed and the Quarkus web-app uses the tokens issued to access information about the user and grants the relevant role-based authorization to that user. The following tokens are issued:

- ID token: The Quarkus **web-app** application uses the user information in the ID token to enable the authenticated user to log in securely and to provide role-based access to the web application.

- Access token: The Quarkus web-app might use the access token to access the UserInfo API to get additional information about the authenticated user or to propagate it to another endpoint.

- Refresh token: (Optional) If the ID and access tokens expire, the Quarkus web-app can use the refresh token to get new ID and access tokens.

See also the OIDC configuration properties reference guide.

To learn about how you can protect web applications by using the OIDC Authorization Code Flow mechanism, see Protect a web application by using OIDC authorization code flow .

If you want to protect service applications by using OIDC Bearer token authentication, see OIDC Bearer token authentication.

For information about how to support multiple tenants, see Using OpenID Connect Multi-Tenancy.

3.2. USING THE AUTHORIZATION CODE FLOW MECHANISM

3.2.1. Configuring access to the OIDC provider endpoint

The OIDC **web-app** application requires URLs of the OIDC provider's authorization, token, **JsonWebKey** (JWK) set, and possibly the **UserInfo**, introspection and end-session (RP-initiated logout) endpoints.

By convention, they are discovered by adding a **/.well-known/openid-configuration** path to the configured **quarkus.oidc.auth-server-url**.

Alternatively, if the discovery endpoint is not available, or you prefer to reduce the discovery endpoint round-trip, you can disable endpoint discovery and configure relative path values. For example:

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
quarkus.oidc.discovery-enabled=false
# Authorization endpoint: http://localhost:8180/realms/quarkus/protocol/openid-connect/auth
quarkus.oidc.authorization-path=/protocol/openid-connect/auth
# Token endpoint: http://localhost:8180/realms/quarkus/protocol/openid-connect/token
quarkus.oidc.token-path=/protocol/openid-connect/token
# JWK set endpoint: http://localhost:8180/realms/quarkus/protocol/openid-connect/certs
quarkus.oidc.jwks-path=/protocol/openid-connect/certs
# UserInfo endpoint: http://localhost:8180/realms/quarkus/protocol/openid-connect/userinfo
quarkus.oidc.user-info-path=/protocol/openid-connect/userinfo
```

```
# Token Introspection endpoint: http://localhost:8180/realms/quarkus/protocol/openid-
connect/token/introspect
quarkus.oidc.introspection-path=/protocol/openid-connect/token/introspect
# End-session endpoint: http://localhost:8180/realms/quarkus/protocol/openid-connect/logout
quarkus.oidc.end-session-path=/protocol/openid-connect/logout
```

Some OIDC providers support metadata discovery but do not return all the endpoint URL values required for the authorization code flow to complete or to support application functions, for example, user logout. To work around this limitation, you can configure the missing endpoint URL values locally, as outlined in the following example:

```
# Metadata is auto-discovered but it does not return an end-session endpoint URL

quarkus.oidc.auth-server-url=http://localhost:8180/oidcprovider/account

# Configure the end-session URL locally.
# It can be an absolute or relative (to 'quarkus.oidc.auth-server-url') address
quarkus.oidc.end-session-path=logout
```

You can use this same configuration to override a discovered endpoint URL if that URL does not work for the local Quarkus endpoint and a more specific value is required. For example, a provider that supports both global and application-specific end-session endpoints returns a global end-session URL such as **http://localhost:8180/oidcprovider/account/global-logout**. This URL will log the user out of all the applications into which the user is currently logged in. However, if the requirement is for the current application to log the user out of a specific application only, you can override the global end-session URL, by setting the **quarkus.oidc.end-session-path=logout** parameter.

3.2.1.1. OIDC provider client authentication

OIDC providers typically require applications to be identified and authenticated when they interact with the OIDC endpoints. Quarkus OIDC, specifically the **quarkus.oidc.runtime.OidcProviderClient** class, authenticates to the OIDC provider when the authorization code must be exchanged for the ID, access, and refresh tokens, or when the ID and access tokens must be refreshed or introspected.

Typically, client id and client secrets are defined for a given application when it enlists to the OIDC provider. All OIDC client authentication options are supported. For example:

Example of client_secret_basic:

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus/
quarkus.oidc.client-id=quarkus-app
quarkus.oidc.credentials.secret=mysecret
```

Or:

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus/
quarkus.oidc.client-id=quarkus-app
quarkus.oidc.credentials.client-secret.value=mysecret
```

The following example shows the secret retrieved from a credentials provider:

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus/
quarkus.oidc.client-id=quarkus-app
```

```
# This is a key which will be used to retrieve a secret from the map of credentials returned from
CredentialsProvider
quarkus.oidc.credentials.client-secret.provider.key=mysecret-key
# Set it only if more than one CredentialsProvider can be registered
quarkus.oidc.credentials.client-secret.provider.name=oidc-credentials-provider
```

Example of client_secret_post

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus/
quarkus.oidc.client-id=quarkus-app
quarkus.oidc.credentials.client-secret.value=mysecret
quarkus.oidc.credentials.client-secret.method=post
```

Example of client_secret_jwt, where the signature algorithm is HS256:

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus/
quarkus.oidc.client-id=quarkus-app
quarkus.oidc.credentials.jwt.secret=AyM1SysPpbyDfgZld3umj1qzKObwVMkoqQ-EstJQLr_T-
1qS0gZH75aKtMN3Yj0iPS4hcgUuTwjAzZr1Z9CAow
```

Example of client_secret_jwt, where the secret is retrieved from a credentials provider:

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus/
quarkus.oidc.client-id=quarkus-app

# This is a key which will be used to retrieve a secret from the map of credentials returned from
CredentialsProvider
quarkus.oidc.credentials.jwt.secret-provider.key=mysecret-key
# Set it only if more than one CredentialsProvider can be registered
quarkus.oidc.credentials.jwt.secret-provider.name=oidc-credentials-provider
```

Example of **private_key_jwt** with the PEM key file, and where the signature algorithm is RS256:

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus/
quarkus.oidc.client-id=quarkus-app
quarkus.oidc.credentials.jwt.key-file=privateKey.pem
```

Example of private_key_jwt with the keystore file, where the signature algorithm is RS256:

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus/
quarkus.oidc.client-id=quarkus-app
quarkus.oidc.credentials.jwt.key-store-file=keystore.jks
quarkus.oidc.credentials.jwt.key-store-password=mypassword
quarkus.oidc.credentials.jwt.key-password=mykeypassword

# Private key alias inside the keystore
quarkus.oidc.credentials.jwt.key-id=mykeyAlias
```

Using **client_secret_jwt** or **private_key_jwt** authentication methods ensures that a client secret does not get sent to the OIDC provider, therefore avoiding the risk of a secret being intercepted by a 'man-in-the-middle' attack.

3.2.1.1.1. Additional JWT authentication options

If **client_secret_jwt**, **private_key_jwt**, or an Apple **post_jwt** authentication methods are used, then you can customize the JWT signature algorithm, key identifier, audience, subject and issuer. For example:

```
# private_key_jwt client authentication

quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus/
quarkus.oidc.client-id=quarkus-app
quarkus.oidc.credentials.jwt.key-file=privateKey.pem

# This is a token key identifier 'kid' header - set it if your OIDC provider requires it:
# Note if the key is represented in a JSON Web Key (JWK) format with a `kid` property, then
# using 'quarkus.oidc.credentials.jwt.token-key-id' is not necessary.
quarkus.oidc.credentials.jwt.token-key-id=mykey

# Use RS512 signature algorithm instead of the default RS256
quarkus.oidc.credentials.jwt.signature-algorithm=RS512

# The token endpoint URL is the default audience value, use the base address URL instead:
quarkus.oidc.credentials.jwt.audience=${quarkus.oidc-client.auth-server-url}

# custom subject instead of the client id:
quarkus.oidc.credentials.jwt.subject=custom-subject

# custom issuer instead of the client id:
quarkus.oidc.credentials.jwt.issuer=custom-issuer
```

3.2.1.1.2. Apple POST JWT

The Apple OIDC provider uses a **client_secret_post** method whereby a secret is a JWT produced with a **private_key_jwt** authentication method, but with the Apple account-specific issuer and subject claims.

In Quarkus Security, **quarkus-oidc** supports a non-standard **client_secret_post_jwt** authentication method, which you can configure as follows:

```
# Apple provider configuration sets a 'client_secret_post_jwt' authentication method
quarkus.oidc.provider=apple

quarkus.oidc.client-id=${apple.client-id}
quarkus.oidc.credentials.jwt.key-file=ecPrivateKey.pem
quarkus.oidc.credentials.jwt.token-key-id=${apple.key-id}
# Apple provider configuration sets ES256 signature algorithm

quarkus.oidc.credentials.jwt.subject=${apple.subject}
quarkus.oidc.credentials.jwt.issuer=${apple.issuer}
```

3.2.1.1.3. mutual TLS (mTLS)

Some OIDC providers might require that a client is authenticated as part of the mutual TLS authentication process.

The following example shows how you can configure **quarkus-oidc** to support **mTLS**:

```
quarkus.oidc.tls.verification=certificate-validation

# Keystore configuration
quarkus.oidc.tls.key-store-file=client-keystore.jks
quarkus.oidc.tls.key-store-password=${key-store-password}

# Add more keystore properties if needed:
#quarkus.oidc.tls.key-store-alias=keyAlias
#quarkus.oidc.tls.key-store-alias-password=keyAliasPassword

# Truststore configuration
quarkus.oidc.tls.trust-store-file=client-truststore.jks
quarkus.oidc.tls.trust-store-password=${trust-store-password}
# Add more truststore properties if needed:
#quarkus.oidc.tls.trust-store-alias=certAlias
```

3.2.1.1.4. POST query

Some providers, such as the Strava OAuth2 provider, require client credentials be posted as HTTP POST query parameters:

```
quarkus.oidc.provider=strava
quarkus.oidc.client-id=quarkus-app
quarkus.oidc.credentials.client-secret.value=mysecret
quarkus.oidc.credentials.client-secret.method=query
```

3.2.1.2. Introspection endpoint authentication

Some OIDC providers require authentication to its introspection endpoint by using Basic authentication and with credentials that are different from the **client_id** and **client_secret**. If you have previously configured security authentication to support either the **client_secret_basic** or **client_secret_post** client authentication methods as described in the OIDC provider client authentication section, you might need to apply the additional configuration as follows.

If the tokens have to be introspected and the introspection endpoint-specific authentication mechanism is required, you can configure **quarkus-oidc** as follows:

```
quarkus.oidc.introspection-credentials.name=introspection-user-name
quarkus.oidc.introspection-credentials.secret=introspection-user-secret
```

3.2.1.3. OIDC request filters

You can filter OIDC requests made by Quarkus to the OIDC provider by registering one or more **OidcRequestFilter** implementations, which can update or add new request headers and can also log requests.

For example:

```
package io.quarkus.it.keycloak;

import jakarta.enterprise.context.ApplicationScoped;

import io.quarkus.arc.Unremovable;
import io.quarkus.oidc.common.OidcRequestContextProperties;
import io.quarkus.oidc.common.OidcRequestFilter;
import io.vertx.mutiny.core.buffer.Buffer;
import io.vertx.mutiny.ext.web.client.HttpRequest;

@ApplicationScoped
@Unremovable
public class OidcTokenRequestCustomizer implements OidcRequestFilter {
    @Override
    public void filter(HttpRequest<Buffer> request, Buffer buffer, OidcRequestContextProperties
contextProps) {
        OidcConfigurationMetadata metadata =
contextProps.get(OidcConfigurationMetadata.class.getName()); 1
        // Metadata URI is absolute, request URI value is relative
        if (metadata.getTokenUri().endsWith(request.uri())) { 2
            request.putHeader("TokenGrantDigest", calculateDigest(buffer.toString()));
        }
    }
    private String calculateDigest(String bodyString) {
        // Apply the required digest algorithm to the body string
    }
}
```

1 Get **OidcConfigurationMetadata**, which contains all supported OIDC endpoint addresses.

2 Use **OidcConfigurationMetadata** to filter requests to the OIDC token endpoint only.

Alternatively, you can use **OidcRequestFilter.Endpoint** enum to apply this filter to the token endpoint requests only:

```
import jakarta.enterprise.context.ApplicationScoped;

import io.quarkus.arc.Unremovable;
import io.quarkus.oidc.common.OidcEndpoint;
import io.quarkus.oidc.common.OidcEndpoint.Type;
import io.quarkus.oidc.common.OidcRequestContextProperties;
import io.quarkus.oidc.common.OidcRequestFilter;
import io.vertx.mutiny.core.buffer.Buffer;
import io.vertx.mutiny.ext.web.client.HttpRequest;

@ApplicationScoped
@Unremovable
@OidcEndpoint(value = Type.DISCOVERY) 1
public class OidcDiscoveryRequestCustomizer implements OidcRequestFilter {

    @Override
    public void filter(HttpRequest<Buffer> request, Buffer buffer, OidcRequestContextProperties
contextProps) {
        request.putHeader("Discovery", "OK");
    }
}
```

1 Restrict this filter to requests targeting the OIDC discovery endpoint only.

3.2.1.4. Redirecting to and from the OIDC provider

When a user is redirected to the OIDC provider to authenticate, the redirect URL includes a **redirect_uri** query parameter, which indicates to the provider where the user has to be redirected to when the authentication is complete. In our case, this is the Quarkus application.

Quarkus sets this parameter to the current application request URL by default. For example, if a user is trying to access a Quarkus service endpoint at **http://localhost:8080/service/1**, then the **redirect_uri** parameter is set to **http://localhost:8080/service/1**. Similarly, if the request URL is **http://localhost:8080/service/2**, then the **redirect_uri** parameter is set to **http://localhost:8080/service/2**.

Some OIDC providers require the **redirect_uri** to have the same value for a given application, for example, **http://localhost:8080/service/callback**, for all the redirect URLs. In such cases, a **quarkus.oidc.authentication.redirect-path** property has to be set. For example, **quarkus.oidc.authentication.redirect-path=/service/callback**, and Quarkus will set the **redirect_uri** parameter to an absolute URL such as **http://localhost:8080/service/callback**, which will be the same regardless of the current request URL.

If **quarkus.oidc.authentication.redirect-path** is set, but you need the original request URL to be restored after the user is redirected back to a unique callback URL, for example, **http://localhost:8080/service/callback**, set **quarkus.oidc.authentication.restore-path-after-redirect** property to **true**. This will restore the request URL such as **http://localhost:8080/service/1**.

3.2.1.5. Customizing authentication requests

By default, only the **response_type** (set to **code**), **scope** (set to **openid**), **client_id**, **redirect_uri**, and **state** properties are passed as HTTP query parameters to the OIDC provider's authorization endpoint when the user is redirected to it to authenticate.

You can add more properties to it with **quarkus.oidc.authentication.extra-params**. For example, some OIDC providers might choose to return the authorization code as part of the redirect URI's fragment, which would break the authentication process. The following example shows how you can work around this issue:

```
quarkus.oidc.authentication.extra-params.response_mode=query
```

3.2.1.6. Customizing the authentication error response

When the user is redirected to the OIDC authorization endpoint to authenticate and, if necessary, authorize the Quarkus application, this redirect request might fail, for example, when an invalid scope is included in the redirect URI. In such cases, the provider redirects the user back to Quarkus with **error** and **error_description** parameters instead of the expected **code** parameter.

For example, this can happen when an invalid scope or other invalid parameters are included in the redirect to the provider.

In such cases, an HTTP **401** error is returned by default. However, you can request that a custom public error endpoint be called to return a more user-friendly HTML error page. To do this, set the **quarkus.oidc.authentication.error-path** property, as shown in the following example:

```
quarkus.oidc.authentication.error-path=/error
```

Ensure that the property starts with a forward slash (/) character and the path is relative to the base URI of the current endpoint. For example, if it is set to '/error' and the current request URI is

https://localhost:8080/callback?error=invalid_scope, then a final redirect is made to **https://localhost:8080/error?error=invalid_scope**.

IMPORTANT

To prevent the user from being redirected to this page to be re-authenticated, ensure that this error endpoint is a public resource.

3.2.2. Accessing authorization data

You can access information about authorization in different ways.

3.2.2.1. Accessing ID and access tokens

The OIDC code authentication mechanism acquires three tokens during the authorization code flow: ID token, access token, and refresh token.

The ID token is always a JWT token and represents a user authentication with the JWT claims. You can use this to get the issuing OIDC endpoint, the username, and other information called *claims*. You can access ID token claims by injecting **JsonWebToken** with an **IdToken** qualifier:

```
import jakarta.inject.Inject;
import org.eclipse.microprofile.jwt.JsonWebToken;
import io.quarkus.oidc.IdToken;
import io.quarkus.security.Authenticated;

@Path("/web-app")
@Authenticated
public class ProtectedResource {

    @Inject
    @IdToken
    JsonWebToken idToken;

    @GET
    public String getUserName() {
        return idToken.getName();
    }
}
```

The OIDC **web-app** application usually uses the access token to access other endpoints on behalf of the currently logged-in user. You can access the raw access token as follows:

```
import jakarta.inject.Inject;
import org.eclipse.microprofile.jwt.JsonWebToken;
import io.quarkus.oidc.AccessTokenCredential;
import io.quarkus.security.Authenticated;

@Path("/web-app")
@Authenticated
public class ProtectedResource {

    @Inject
    JsonWebToken accessToken;
    // or
    // @Inject
    // AccessTokenCredential accessTokenCredential;

    @GET
    public String getReservationOnBehalfOfUser() {
        String rawAccessToken = accessToken.getRawToken();
        //or
        //String rawAccessToken = accessTokenCredential.getToken();

        // Use the raw access token to access a remote endpoint.
        // For example, use RestClient to set this token as a `Bearer` scheme value of the HTTP
`Authorization` header:
        // `Authorization: Bearer rawAccessToken`.
        return getReservationfromRemoteEndpoint(rawAccesstoken);
    }
}
```

NOTE

AccessTokenCredential is used if the access token issued to the Quarkus **web-app** application is opaque (binary) and cannot be parsed to a **JsonWebToken** or if the inner content is necessary for the application.

Injection of the **JsonWebToken** and **AccessTokenCredential** is supported in both **@RequestScoped** and **@ApplicationScoped** contexts.

Quarkus OIDC uses the refresh token to refresh the current ID and access tokens as part of its session management process.

3.2.2.2. User info

If the ID token does not provide enough information about the currently authenticated user, you can get more information from the **UserInfo** endpoint. Set the **quarkus.oidc.authentication.user-info-required=true** property to request a UserInfo JSON object from the OIDC **UserInfo** endpoint.

A request is sent to the OIDC provider **UserInfo** endpoint by using the access token returned with the authorization code grant response, and an **io.quarkus.oidc.UserInfo** (a simple **jakarta.json.JsonObject** wrapper) object is created. **io.quarkus.oidc.UserInfo** can be injected or accessed as a SecurityIdentity **userinfo** attribute.

3.2.2.3. Accessing the OIDC configuration information

The current tenant's discovered OpenID Connect configuration metadata is represented by **io.quarkus.oidc.OidcConfigurationMetadata** and can be injected or accessed as a **SecurityIdentity configuration-metadata** attribute.

The default tenant's **OidcConfigurationMetadata** is injected if the endpoint is public.

3.2.2.4. Mapping token claims and **SecurityIdentity** roles

The way the roles are mapped to the SecurityIdentity roles from the verified tokens is identical to how it is done for the Bearer tokens. The only difference is that ID token is used as a source of the roles by default.

NOTE

If you use Keycloak, set a **microprofile-jwt** client scope for the ID token to contain a **groups** claim. For more information, see the Keycloak server administration guide.

However, depending on your OIDC provider, roles might be stored in the access token or the user info.

If the access token contains the roles and this access token is not meant to be propagated to the downstream endpoints, then set **quarkus.oidc.roles.source=accesstoken**.

If UserInfo is the source of the roles, then set **quarkus.oidc.roles.source=userinfo**, and if needed, **quarkus.oidc.roles.role-claim-path**.

Additionally, you can also use a custom **SecurityIdentityAugmentor** to add the roles. For more information, see SecurityIdentity customization. You can also map **SecurityIdentity** roles created from token claims to deployment-specific roles with the HTTP Security policy .

3.2.3. Ensuring validity of tokens and authentication data

A core part of the authentication process is ensuring the chain of trust and validity of the information. This is done by ensuring tokens can be trusted.

3.2.3.1. Token verification and introspection

The verification process of OIDC authorization code flow tokens follows the Bearer token authentication token verification and introspection logic. For more information, see the Token verification and introspection section of the "Quarkus OpenID Connect (OIDC) Bearer token authentication" guide.

NOTE

With Quarkus **web-app** applications, only the **IdToken** is verified by default because the access token is not used to access the current Quarkus web-app endpoint and is intended to be propagated to the services expecting this access token. If you expect the access token to contain the roles required to access the current Quarkus endpoint (**quarkus.oidc.roles.source=accesstoken**), then it will also be verified.

3.2.3.2. Token introspection and UserInfo cache

Code flow access tokens are not introspected unless they are expected to be the source of roles. However, they will be used to get **UserInfo**. There will be one or two remote calls with the code flow access token if the token introspection, **UserInfo**, or both are required.

For more information about using the default token cache or registering a custom cache implementation, see Token introspection and UserInfo cache .

3.2.3.3. JSON web token claim verification

For information about the claim verification, including the **iss** (issuer) claim, see the JSON Web Token claim verification section. It applies to ID tokens and also to access tokens in a JWT format, if the **web-app** application has requested the access token verification.

3.2.3.4. Further security with Proof Key for Code Exchange (PKCE)

Proof Key for Code Exchange (PKCE) minimizes the risk of authorization code interception.

While PKCE is of primary importance to public OIDC clients, such as SPA scripts running in a browser, it can also provide extra protection to Quarkus OIDC **web-app** applications. With PKCE, Quarkus OIDC **web-app** applications act as confidential OIDC clients that can securely store the client secret and use it to exchange the code for the tokens.

You can enable PKCE for your OIDC web-app endpoint with a **quarkus.oidc.authentication.pkce-required** property and a 32-character secret that is required to encrypt the PKCE code verifier in the state cookie, as shown in the following example:

```
quarkus.oidc.authentication.pkce-required=true
quarkus.oidc.authentication.state-secret=eUk1p7UB3nFiXZGUXi0uph1Y9p34YhBU
```

If you already have a 32-character client secret, you do not need to set the **quarkus.oidc.authentication.pkce-secret** property unless you prefer to use a different secret key. This secret will be auto-generated if it is not configured and if the fallback to the client secret is not possible in cases where the client secret is less than 16 characters long.

The secret key is required to encrypt a randomly generated PKCE **code_verifier** while the user is redirected with the **code_challenge** query parameter to an OIDC provider to authenticate. The **code_verifier** is decrypted when the user is redirected back to Quarkus and sent to the token endpoint alongside the **code**, client secret, and other parameters to complete the code exchange. The provider will fail the code exchange if a **SHA256** digest of the **code_verifier** does not match the **code_challenge** that was provided during the authentication request.

3.2.4. Handling and controlling the lifetime of authentication

Another important requirement for authentication is to ensure that the data the session is based on is up-to-date without requiring the user to authenticate for every single request. There are also situations where a logout event is explicitly requested. Use the following key points to find the right balance for securing your Quarkus applications:

3.2.4.1. Cookies

The OIDC adapter uses cookies to keep the session, code flow, and post-logout state. This state is a key element controlling the lifetime of authentication data.

Use the **quarkus.oidc.authentication.cookie-path** property to ensure that the same cookie is visible when you access protected resources with overlapping or different roots. For example:

- **/index.html** and **/web-app/service**
- **/web-app/service1** and **/web-app/service2**
- **/web-app1/service** and **/web-app2/service**

By default, **quarkus.oidc.authentication.cookie-path** is set to / but you can change this to a more specific path if required, for example, **/web-app**.

To set the cookie path dynamically, configure the **quarkus.oidc.authentication.cookie-path-header** property. Set the **quarkus.oidc.authentication.cookie-path-header** property. For example, to set the cookie path dynamically by using the value of the`X-Forwarded-Prefix` HTTP header, configure the property to **quarkus.oidc.authentication.cookie-path-header=X-Forwarded-Prefix**.

If **quarkus.oidc.authentication.cookie-path-header** is set but no configured HTTP header is available in the current request, then the **quarkus.oidc.authentication.cookie-path** will be checked.

If your application is deployed across multiple domains, set the **quarkus.oidc.authentication.cookie-domain** property so that the session cookie is visible to all protected Quarkus services. For example, if you have Quarkus services deployed on the following two domains, then you must set the **quarkus.oidc.authentication.cookie-domain** property to **company.net**:

- https://whatever.wherever.company.net/
- https://another.address.company.net/

3.2.4.2. Session cookie and default TokenStateManager

OIDC **CodeAuthenticationMechanism** uses the default **io.quarkus.oidc.TokenStateManager** interface implementation to keep the ID, access, and refresh tokens returned in the authorization code or refresh grant responses in an encrypted session cookie.

It makes Quarkus OIDC endpoints completely stateless and it is recommended to follow this strategy to achieve the best scalability results.

See the Session cookie and custom TokenStateManager section for alternative methods of token storage. This is ideal for those seeking customized solutions for token state management, especially when standard server-side storage does not meet your specific requirements.

You can configure the default **TokenStateManager** to avoid saving an access token in the session cookie and to only keep ID and refresh tokens or a single ID token only.

An access token is only required if the endpoint needs to do the following actions:

- Retrieve **UserInfo**
- Access the downstream service with this access token
- Use the roles associated with the access token, which are checked by default

In such cases, use the **quarkus.oidc.token-state-manager.strategy** property to configure the token state strategy as follows:

To…	Set the property to …
Keep the ID and refresh tokens only	**quarkus.oidc.token-state-manager.strategy=id-refresh-tokens**
Keep the ID token only	**quarkus.oidc.token-state-manager.strategy=id-token**

If your chosen session cookie strategy combines tokens and generates a large session cookie value that is greater than 4KB, some browsers might not be able to handle such cookie sizes. This can occur when the ID, access, and refresh tokens are JWT tokens and the selected strategy is **keep-all-tokens** or with ID and refresh tokens when the strategy is **id-refresh-token**. To work around this issue, you can set **quarkus.oidc.token-state-manager.split-tokens=true** to create a unique session token for each token.

The default **TokenStateManager** encrypts the tokens before storing them in the session cookie. The following example shows how you configure it to split the tokens and encrypt them:

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
quarkus.oidc.client-id=quarkus-app
quarkus.oidc.credentials.secret=secret
quarkus.oidc.application-type=web-app
quarkus.oidc.token-state-manager.split-tokens=true
quarkus.oidc.token-state-manager.encryption-secret=eUk1p7UB3nFiXZGUXi0uph1Y9p34YhBU
```

The token encryption secret must be at least 32 characters long. If this key is not configured, then either **quarkus.oidc.credentials.secret** or **quarkus.oidc.credentials.jwt.secret** will be hashed to create an encryption key.

Configure the **quarkus.oidc.token-state-manager.encryption-secret** property if Quarkus authenticates to the OIDC provider by using one of the following authentication methods:

- mTLS
- **private_key_jwt**, where a private RSA or EC key is used to sign a JWT token

Otherwise, a random key is generated, which can be problematic if the Quarkus application is running in the cloud with multiple pods managing the requests.

You can disable token encryption in the session cookie by setting **quarkus.oidc.token-state-manager.encryption-required=false**.

3.2.4.3. Session cookie and custom TokenStateManager

If you want to customize the way the tokens are associated with the session cookie, register a custom **io.quarkus.oidc.TokenStateManager** implementation as an **@ApplicationScoped** CDI bean.

For example, you might want to keep the tokens in a cache cluster and have only a key stored in a session cookie. Note that this approach might introduce some challenges if you need to make the tokens available across multiple microservices nodes.

Here is a simple example:

```
package io.quarkus.oidc.test;

import jakarta.annotation.Priority;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.inject.Alternative;
import jakarta.inject.Inject;

import io.quarkus.oidc.AuthorizationCodeTokens;
import io.quarkus.oidc.OidcTenantConfig;
import io.quarkus.oidc.TokenStateManager;
import io.quarkus.oidc.runtime.DefaultTokenStateManager;
import io.smallrye.mutiny.Uni;
import io.vertx.ext.web.RoutingContext;
```

```
@ApplicationScoped
@Alternative
@Priority(1)
public class CustomTokenStateManager implements TokenStateManager {

    @Inject
    DefaultTokenStateManager tokenStateManager;

    @Override
    public Uni<String> createTokenState(RoutingContext routingContext, OidcTenantConfig
oidcConfig,
            AuthorizationCodeTokens sessionContent,
TokenStateManager.CreateTokenStateRequestContext requestContext) {
        return tokenStateManager.createTokenState(routingContext, oidcConfig, sessionContent,
requestContext)
                .map(t -> (t + "|custom"));
    }

    @Override
    public Uni<AuthorizationCodeTokens> getTokens(RoutingContext routingContext,
OidcTenantConfig oidcConfig,
            String tokenState, TokenStateManager.GetTokensRequestContext requestContext) {
        if (!tokenState.endsWith("|custom")) {
            throw new IllegalStateException();
        }
        String defaultState = tokenState.substring(0, tokenState.length() - 7);
        return tokenStateManager.getTokens(routingContext, oidcConfig, defaultState, requestContext);
    }

    @Override
    public Uni<Void> deleteTokens(RoutingContext routingContext, OidcTenantConfig oidcConfig,
String tokenState,
            TokenStateManager.DeleteTokensRequestContext requestContext) {
        if (!tokenState.endsWith("|custom")) {
            throw new IllegalStateException();
        }
        String defaultState = tokenState.substring(0, tokenState.length() - 7);
        return tokenStateManager.deleteTokens(routingContext, oidcConfig, defaultState,
requestContext);
    }
}
```

For information about the default **TokenStateManager** storing tokens in an encrypted session cookie, see Session cookie and default TokenStateManager.

3.2.4.4. Logout and expiration

There are two main ways for the authentication information to expire: the tokens expired and were not renewed or an explicit logout operation was triggered.

Let's start with explicit logout operations.

3.2.4.4.1. User-initiated logout

Users can request a logout by sending a request to the Quarkus endpoint logout path set with a **quarkus.oidc.logout.path** property. For example, if the endpoint address is

https://application.com/webapp and the **quarkus.oidc.logout.path** is set to **/logout**, then the logout request must be sent to **https://application.com/webapp/logout**.

This logout request starts an RP-initiated logout. The user will be redirected to the OIDC provider to log out, where they can be asked to confirm the logout is indeed intended.

The user will be returned to the endpoint post-logout page once the logout has been completed and if the **quarkus.oidc.logout.post-logout-path** property is set. For example, if the endpoint address is **https://application.com/webapp** and the **quarkus.oidc.logout.post-logout-path** is set to **/signin**, then the user will be returned to **https://application.com/webapp/signin**. Note, this URI must be registered as a valid **post_logout_redirect_uri** in the OIDC provider.

If the **quarkus.oidc.logout.post-logout-path** is set, then a **q_post_logout** cookie will be created and a matching **state** query parameter will be added to the logout redirect URI and the OIDC provider will return this **state** once the logout has been completed. It is recommended for the Quarkus **web-app** applications to check that a **state** query parameter matches the value of the **q_post_logout** cookie, which can be done, for example, in a Jakarta REST filter.

Note that a cookie name varies when using OpenID Connect Multi-Tenancy. For example, it will be named **q_post_logout_tenant_1** for a tenant with a **tenant_1** ID, and so on.

Here is an example of how to configure a Quarkus application to initiate a logout flow:

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
quarkus.oidc.client-id=frontend
quarkus.oidc.credentials.secret=secret
quarkus.oidc.application-type=web-app

quarkus.oidc.logout.path=/logout
# Logged-out users should be returned to the /welcome.html site which will offer an option to re-login:
quarkus.oidc.logout.post-logout-path=/welcome.html

# Only the authenticated users can initiate a logout:
quarkus.http.auth.permission.authenticated.paths=/logout
quarkus.http.auth.permission.authenticated.policy=authenticated

# All users can see the Welcome page:
quarkus.http.auth.permission.public.paths=/welcome.html
quarkus.http.auth.permission.public.policy=permit
```

You might also want to set **quarkus.oidc.authentication.cookie-path** to a path value common to all the application resources, which is / in this example. For more information, see the Cookies section.

NOTE

Some OIDC providers do not support a RP-initiated logout specification and do not return an OpenID Connect well-known **end_session_endpoint** metadata property. However, this is not a problem for Quarkus because the specific logout mechanisms of such OIDC providers only differ in how the logout URL query parameters are named.

According to the RP-initiated logout specification, the **quarkus.oidc.logout.post-logout-path** property is represented as a **post_logout_redirect_uri** query parameter, which is not recognized by the providers that do not support this specification.

You can use **quarkus.oidc.logout.post-logout-url-param** to work around this issue. You can also request more logout query parameters added with **quarkus.oidc.logout.extra-params**. For example, here is how you can support a logout with **Auth0**:

```
quarkus.oidc.auth-server-url=https://dev-xxx.us.auth0.com
quarkus.oidc.client-id=redacted
quarkus.oidc.credentials.secret=redacted
quarkus.oidc.application-type=web-app

quarkus.oidc.tenant-logout.logout.path=/logout
quarkus.oidc.tenant-logout.logout.post-logout-path=/welcome.html

# Auth0 does not return the `end_session_endpoint` metadata property. Instead, you
must configure it:
quarkus.oidc.end-session-path=v2/logout
# Auth0 will not recognize the 'post_logout_redirect_uri' query parameter so ensure it
is named as 'returnTo':
quarkus.oidc.logout.post-logout-uri-param=returnTo

# Set more properties if needed.
# For example, if 'client_id' is provided, then a valid logout URI should be set as the
Auth0 Application property, without it - as Auth0 Tenant property:
quarkus.oidc.logout.extra-params.client_id=${quarkus.oidc.client-id}
```

3.2.4.4.2. Back-channel logout

The OIDC provider can force the logout of all applications by using the authentication data. This is known as back-channel logout. In this case, the OIDC will call a specific URL from each application to trigger that logout.

OIDC providers use Back-channel logout to log out the current user from all the applications into which this user is currently logged in, bypassing the user agent.

You can configure Quarkus to support Back-channel logout as follows:

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
quarkus.oidc.client-id=frontend
quarkus.oidc.credentials.secret=secret
quarkus.oidc.application-type=web-app

quarkus.oidc.logout.backchannel.path=/back-channel-logout
```

[illegible]

The absolute **back-channel logout** URL is calculated by adding **quarkus.oidc.back-channel-logout.path** to the current endpoint URL, for example, **http://localhost:8080/back-channel-logout**. You will need to configure this URL in the admin console of your OIDC provider.

You will also need to configure a token age property for the logout token verification to succeed if your OIDC provider does not set an expiry claim in the current logout token. For example, set **quarkus.oidc.token.age=10S** to ensure that no more than 10 seconds elapse since the logout token's **iat** (issued at) time.

3.2.4.4.3. Front-channel logout

You can use Front-channel logout to log out the current user directly from the user agent, for example, its browser. It is similar to Back-channel logout but the logout steps are executed by the user agent, such as the browser, and not in the background by the OIDC provider. This option is rarely used.

You can configure Quarkus to support Front-channel logout as follows:

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
quarkus.oidc.client-id=frontend
quarkus.oidc.credentials.secret=secret
quarkus.oidc.application-type=web-app

quarkus.oidc.logout.frontchannel.path=/front-channel-logout
```

This path will be compared to the current request's path, and the user will be logged out if these paths match.

3.2.4.4.4. Local logout

User-initiated logout will log the user out of the OIDC provider. If it is used as single sign-on, it might not be what you require. If, for example, your OIDC provider is Google, you will be logged out from Google and its services. Instead, the user might just want to log out of that specific application. Another use case might be when the OIDC provider does not have a logout endpoint.

By using OidcSession, you can support a local logout, which means that only the local session cookie is cleared, as shown in the following example:

```
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;

import io.quarkus.oidc.OidcSession;

@Path("/service")
public class ServiceResource {

    @Inject
    OidcSession oidcSession;

    @GET
    @Path("logout")
    public String logout() {
        oidcSession.logout().await().indefinitely();
        return "You are logged out".
    }
```

3.2.4.4.4.1. Using **OidcSession** for local logout

io.quarkus.oidc.OidcSession is a wrapper around the current **IdToken**, which can help to perform a Local logout, retrieve the current session's tenant identifier, and check when the session will expire. More useful methods will be added to it over time.

3.2.4.5. Session management

By default, logout is based on the expiration time of the ID token issued by the OIDC provider. When the ID token expires, the current user session at the Quarkus endpoint is invalidated, and the user is redirected to the OIDC provider again to authenticate. If the session at the OIDC provider is still active, users are automatically re-authenticated without needing to provide their credentials again.

The current user session can be automatically extended by enabling the **quarkus.oidc.token.refresh-expired** property. If set to **true**, when the current ID token expires, a refresh token grant will be used to refresh the ID token as well as access and refresh tokens.

TIP

If you have a single page application for service applications where your OIDC provider script such as **keycloak.js** is managing an authorization code flow, then that script will also control the SPA authentication session lifespan.

If you work with a Quarkus OIDC **web-app** application, then the Quarkus OIDC code authentication mechanism manages the user session lifespan.

To use the refresh token, you should carefully configure the session cookie age. The session age should be longer than the ID token lifespan and close to or equal to the refresh token lifespan.

You calculate the session age by adding the lifespan value of the current ID token and the values of the **quarkus.oidc.authentication.session-age-extension** and **quarkus.oidc.token.lifespan-grace** properties.

TIP

You use only the **quarkus.oidc.authentication.session-age-extension** property to significantly extend the session lifespan, if required. You use the **quarkus.oidc.token.lifespan-grace** property only to consider some small clock skews.

When the current authenticated user returns to the protected Quarkus endpoint and the ID token associated with the session cookie has expired, then, by default, the user is automatically redirected to the OIDC Authorization endpoint to re-authenticate. The OIDC provider might challenge the user again if the session between the user and this OIDC provider is still active, which might happen if the session is configured to last longer than the ID token.

If the **quarkus.oidc.token.refresh-expired** is set to **true**, then the expired ID token (and the access token) is refreshed by using the refresh token returned with the initial authorization code grant response. This refresh token might also be recycled (refreshed) itself as part of this process. As a result, the new session cookie is created, and the session is extended.

NOTE

In instances where the user is not very active, you can use the **quarkus.oidc.authentication.session-age-extension** property to help handle expired ID tokens. If the ID token expires, the session cookie might not be returned to the Quarkus endpoint during the next user request as the cookie lifespan would have elapsed. Quarkus assumes that this request is the first authentication request. Set **quarkus.oidc.authentication.session-age-extension** to be *reasonably* long for your barely-active users and in accordance with your security policies.

You can go one step further and proactively refresh ID tokens or access tokens that are about to expire. Set **quarkus.oidc.token.refresh-token-time-skew** to the value you want to anticipate the refresh. If, during the current user request, it is calculated that the current ID token will expire within this **quarkus.oidc.token.refresh-token-time-skew**, then it is refreshed, and the new session cookie is created. This property should be set to a value that is less than the ID token lifespan; the closer it is to this lifespan value, the more often the ID token is refreshed.

You can further optimize this process by having a simple JavaScript function ping your Quarkus endpoint periodically to emulate the user activity, which minimizes the time frame during which the user might have to be re-authenticated.

NOTE

You cannot extend the user session indefinitely. The returning user with the expired ID token will have to re-authenticate at the OIDC provider endpoint once the refresh token has expired.

3.2.5. Integration with GitHub and non-OIDC OAuth2 providers

Some well-known providers such as GitHub or LinkedIn are not OpenID Connect providers, but OAuth2 providers that support the **authorization code flow**. For example, GitHub OAuth2 and LinkedIn OAuth2. Remember, OIDC is built on top of OAuth2.

The main difference between OIDC and OAuth2 providers is that OIDC providers return an **ID Token** that represents a user authentication, in addition to the standard authorization code flow **access** and **refresh** tokens returned by **OAuth2** providers.

OAuth2 providers such as GitHub do not return **IdToken**, and the user authentication is implicit and indirectly represented by the **access** token. This **access** token represents an authenticated user authorizing the current Quarkus **web-app** application to access some data on behalf of the authenticated user.

For OIDC, you validate the ID token as proof of authentication validity whereas in the case of OAuth2, you validate the access token. This is done by subsequently calling an endpoint that requires the access token and that typically returns user information. This approach is similar to the OIDC UserInfo approach, with **UserInfo** fetched by Quarkus OIDC on your behalf.

For example, when working with GitHub, the Quarkus endpoint can acquire an **access** token, which allows the Quarkus endpoint to request a GitHub profile for the current user.

To support the integration with such OAuth2 servers, **quarkus-oidc** needs to be configured a bit differently to allow the authorization code flow responses without **IdToken**: **quarkus.oidc.authentication.id-token-required=false**.

NOTE

Even though you configure the extension to support the authorization code flows without **IdToken**, an internal **IdToken** is generated to standardize the way **quarkus-oidc** operates. You use an **IdToken** to support the authentication session and to avoid redirecting the user to the provider, such as GitHub, on every request. In this case, the session lifespan is set to 5 minutes, which you can extend further as described in the session management section.

This simplifies how you handle an application that supports multiple OIDC providers.

The next step is to ensure that the returned access token can be useful and is valid to the current Quarkus endpoint. The first way is to call the OAuth2 provider introspection endpoint by configuring **quarkus.oidc.introspection-path**, if the provider offers such an endpoint. In this case, you can use the access token as a source of roles using **quarkus.oidc.roles.source=accesstoken**. If no introspection endpoint is present, you can attempt instead to request UserInfo from the provider as it will at least validate the access token. To do so, specify **quarkus.oidc.token.verify-access-token-with-user-info=true**. You also need to set the **quarkus.oidc.user-info-path** property to a URL endpoint that fetches the user info (or to an endpoint protected by the access token). For GitHub, since it does not have an introspection endpoint, requesting the UserInfo is required.

NOTE

Requiring UserInfo involves making a remote call on every request. Therefore, you might want to consider caching **UserInfo** data. For more information, see the Token Introspection and UserInfo cache section of the "OpenID Connect (OIDC) Bearer token authentication" guide.

Alternatively, you might want to request that **UserInfo** is embedded into the internal generated **IdToken** with the **quarkus.oidc.cache-user-info-in-idtoken=true** property. The advantage of this approach is that, by default, no cached **UserInfo** state will be kept with the endpoint - instead it will be stored in a session cookie. You might also want to consider encrypting **IdToken** in this case if **UserInfo** contains sensitive data. For more information, see Encrypt tokens with TokenStateManager.

OAuth2 servers might not support a well-known configuration endpoint. In this case, you must disable the discovery and configure the authorization, token, and introspection and **UserInfo** endpoint paths manually.

For well-known OIDC or OAuth2 providers, such as Apple, Facebook, GitHub, Google, Microsoft, Spotify, and Twitter, Quarkus can help significantly simplify your application’s configuration with the **quarkus.oidc.provider** property. Here is how you can integrate **quarkus-oidc** with GitHub after you have created a GitHub OAuth application . Configure your Quarkus endpoint like this:

```
quarkus.oidc.provider=github
quarkus.oidc.client-id=github_app_clientid
quarkus.oidc.credentials.secret=github_app_clientsecret

# user:email scope is requested by default, use 'quarkus.oidc.authentication.scopes' to request
different scopes such as `read:user`.
# See https://docs.github.com/en/developers/apps/building-oauth-apps/scopes-for-oauth-apps for
more information.

# Consider enabling UserInfo Cache
# quarkus.oidc.token-cache.max-size=1000
# quarkus.oidc.token-cache.time-to-live=5M
#
# Or having UserInfo cached inside IdToken itself
# quarkus.oidc.cache-user-info-in-idtoken=true
```

For more information about configuring other well-known providers, see OpenID Connect providers.

This is all that is needed for an endpoint like this one to return the currently-authenticated user's profile with **GET http://localhost:8080/github/userinfo** and access it as the individual **UserInfo** properties:

```
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;

import io.quarkus.oidc.UserInfo;
import io.quarkus.security.Authenticated;

@Path("/github")
@Authenticated
public class TokenResource {

    @Inject
    UserInfo userInfo;

    @GET
    @Path("/userinfo")
    @Produces("application/json")
    public String getUserInfo() {
        return userInfo.getUserInfoString();
    }
}
```

If you support more than one social provider with the help of OpenID Connect Multi-Tenancy, for example, Google, which is an OIDC provider that returns **IdToken**, and GitHub, which is an OAuth2 provider that does not return **IdToken** and only allows access to **UserInfo**, then you can have your endpoint working with only the injected **SecurityIdentity** for both Google and GitHub flows. A simple augmentation of **SecurityIdentity** will be required where a principal created with the internally-generated **IdToken** will be replaced with the **UserInfo**-based principal when the GitHub flow is active:

```
package io.quarkus.it.keycloak;

import java.security.Principal;

import jakarta.enterprise.context.ApplicationScoped;

import io.quarkus.oidc.UserInfo;
import io.quarkus.security.identity.AuthenticationRequestContext;
import io.quarkus.security.identity.SecurityIdentity;
import io.quarkus.security.identity.SecurityIdentityAugmentor;
import io.quarkus.security.runtime.QuarkusSecurityIdentity;
import io.smallrye.mutiny.Uni;
import io.vertx.ext.web.RoutingContext;

@ApplicationScoped
```

```
public class CustomSecurityIdentityAugmentor implements SecurityIdentityAugmentor {

    @Override
    public Uni<SecurityIdentity> augment(SecurityIdentity identity, AuthenticationRequestContext
context) {
        RoutingContext routingContext = identity.getAttribute(RoutingContext.class.getName());
        if (routingContext != null && routingContext.normalizedPath().endsWith("/github")) {
         QuarkusSecurityIdentity.Builder builder = QuarkusSecurityIdentity.builder(identity);
         UserInfo userInfo = identity.getAttribute("userinfo");
         builder.setPrincipal(new Principal() {

             @Override
             public String getName() {
                 return userInfo.getString("preferred_username");
             }

         });
         identity = builder.build();
        }
        return Uni.createFrom().item(identity);
    }

}
```

Now, the following code will work when the user signs into your application by using Google or GitHub:

```
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;

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

@Path("/service")
@Authenticated
public class TokenResource {

    @Inject
    SecurityIdentity identity;

    @GET
    @Path("/google")
    @Produces("application/json")
    public String getUserName() {
        return identity.getPrincipal().getName();
    }

    @GET
    @Path("/github")
    @Produces("application/json")
    public String getUserName() {
        return identity.getPrincipal().getUserName();
    }
}
```

Possibly a simpler alternative is to inject both **@IdToken JsonWebToken** and **UserInfo** and use **JsonWebToken** when handling the providers that return **IdToken** and use **UserInfo** with the providers that do not return **IdToken**.

You must ensure that the callback path you enter in the GitHub OAuth application configuration matches the endpoint path where you want the user to be redirected after a successful GitHub authentication and application authorization. In this case, it has to be set to **http:localhost:8080/github/userinfo**.

3.2.6. Listening to important authentication events

You can register the **@ApplicationScoped** bean which will observe important OIDC authentication events. When a user logs in for the first time, re-authenticates, or refreshes the session, the listener is updated. In the future, more events might be reported. For example:

```
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.event.Observes;

import io.quarkus.oidc.IdTokenCredential;
import io.quarkus.oidc.SecurityEvent;
import io.quarkus.security.identity.AuthenticationRequestContext;
import io.vertx.ext.web.RoutingContext;

@ApplicationScoped
public class SecurityEventListener {

    public void event(@Observes SecurityEvent event) {
        String tenantId = event.getSecurityIdentity().getAttribute("tenant-id");
        RoutingContext vertxContext =
event.getSecurityIdentity().getAttribute(RoutingContext.class.getName());
        vertxContext.put("listener-message", String.format("event:%s,tenantId:%s",
event.getEventType().name(), tenantId));
    }
}
```

TIP

You can listen to other security events as described in the Observe security events section of the Security Tips and Tricks guide.

3.2.7. Propagating tokens to downstream services

For information about Authorization Code Flow access token propagation to downstream services, see the Token Propagation section.

3.3. INTEGRATION CONSIDERATIONS

Your application secured by OIDC integrates in an environment where it can be called from single-page applications. It must work with well-known OIDC providers, run behind HTTP Reverse Proxy, require external and internal access, and so on.

This section discusses these considerations.

3.3.1. Single-page applications

You can check if implementing single-page applications (SPAs) the way it is suggested in the Single-page applications section of the "OpenID Connect (OIDC) Bearer token authentication" guide meets your requirements.

If you prefer to use SPAs and JavaScript APIs such as **Fetch** or **XMLHttpRequest**(XHR) with Quarkus web applications, be aware that OIDC providers might not support cross-origin resource sharing (CORS) for authorization endpoints where the users are authenticated after a redirect from Quarkus. This will lead to authentication failures if the Quarkus application and the OIDC provider are hosted on different HTTP domains, ports, or both.

In such cases, set the **quarkus.oidc.authentication.java-script-auto-redirect** property to **false**, which will instruct Quarkus to return a **499** status code and a **WWW-Authenticate** header with the **OIDC** value.

The browser script must set a header to identify the current request as a JavaScript request for a **499** status code to be returned when the **quarkus.oidc.authentication.java-script-auto-redirect** property is set to **false**.

If the script engine sets an engine-specific request header itself, then you can register a custom **quarkus.oidc.JavaScriptRequestChecker** bean, which will inform Quarkus if the current request is a JavaScript request. For example, if the JavaScript engine sets a header such as **HX-Request: true**, then you can have it checked like this:

```
import jakarta.enterprise.context.ApplicationScoped;

import io.quarkus.oidc.JavaScriptRequestChecker;
import io.vertx.ext.web.RoutingContext;

@ApplicationScoped
public class CustomJavaScriptRequestChecker implements JavaScriptRequestChecker {

    @Override
    public boolean isJavaScriptRequest(RoutingContext context) {
        return "true".equals(context.request().getHeader("HX-Request"));
    }
}
```

and reload the last requested page in case of a **499** status code.

Otherwise, you must also update the browser script to set the **X-Requested-With** header with the **JavaScript** value and reload the last requested page in case of a **499** status code.

For example:

```
Future<void> callQuarkusService() async {
    Map<String, String> headers = Map.fromEntries([MapEntry("X-Requested-With", "JavaScript")]);

    await http
        .get("https://localhost:443/serviceCall")
        .then((response) {
            if (response.statusCode == 499) {
                window.location.assign("https://localhost.com:443/serviceCall");
            }
        });
  }
```

3.3.2. Cross-origin resource sharing

If you plan to consume this application from a single-page application running on a different domain, you need to configure cross-origin resource sharing (CORS). For more information, see the CORS filter section of the "Cross-origin resource sharing" guide.

3.3.3. Running Quarkus application behind a reverse proxy

The OIDC authentication mechanism can be affected if your Quarkus application is running behind a reverse proxy, gateway, or firewall when HTTP **Host** header might be reset to the internal IP address and HTTPS connection might be terminated, and so on. For example, an authorization code flow **redirect_uri** parameter might be set to the internal host instead of the expected external one.

In such cases, configuring Quarkus to recognize the original headers forwarded by the proxy will be required. For more information, see the Running behind a reverse proxy Vert.x documentation section.

For example, if your Quarkus endpoint runs in a cluster behind Kubernetes Ingress, then a redirect from the OIDC provider back to this endpoint might not work because the calculated **redirect_uri** parameter might point to the internal endpoint address. You can resolve this problem by using the following configuration, where **X-ORIGINAL-HOST** is set by Kubernetes Ingress to represent the external endpoint address.:

```
quarkus.http.proxy.proxy-address-forwarding=true
quarkus.http.proxy.allow-forwarded=false
quarkus.http.proxy.enable-forwarded-host=true
quarkus.http.proxy.forwarded-host-header=X-ORIGINAL-HOST
```

quarkus.oidc.authentication.force-redirect-https-scheme property can also be used when the Quarkus application is running behind an SSL terminating reverse proxy.

3.3.4. External and internal access to the OIDC provider

The OIDC provider externally-accessible authorization, logout, and other endpoints can have different HTTP(S) URLs compared to the URLs auto-discovered or configured relative to the **quarkus.oidc.auth-server-url** internal URL. In such cases, the endpoint might report an issuer verification failure and redirects to the externally-accessible OIDC provider endpoints might fail.

If you work with Keycloak, then start it with a **KEYCLOAK_FRONTEND_URL** system property set to the externally-accessible base URL. If you work with other OIDC providers, check the documentation of your provider.

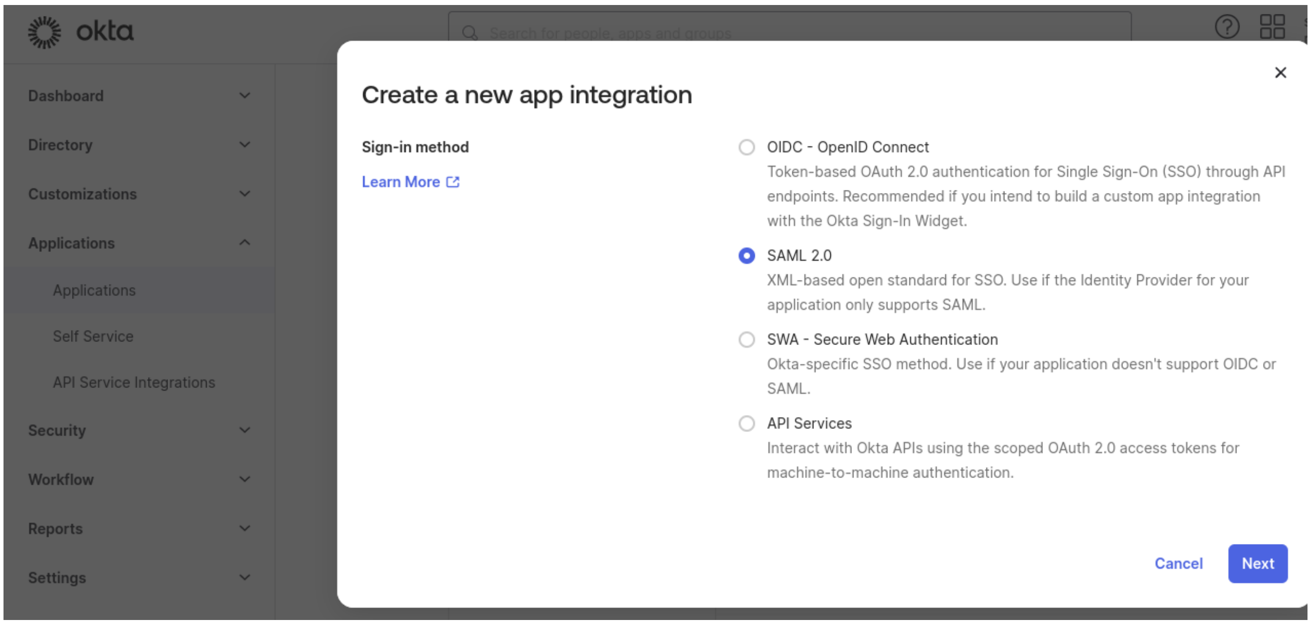
3.4. OIDC SAML IDENTITY BROKER

If your identity provider does not implement OpenID Connect but only the legacy XML-based SAML2.0 SSO protocol, then Quarkus cannot be used as a SAML 2.0 adapter, similarly to how **quarkus-oidc** is used as an OIDC adapter.

However, many OIDC providers such as Keycloak, Okta, Auth0, and Microsoft ADFS offer OIDC to SAML 2.0 bridges. You can create an identity broker connection to a SAML 2.0 provider in your OIDC provider and use **quarkus-oidc** to authenticate your users to this SAML 2.0 provider, with the OIDC provider coordinating OIDC and SAML 2.0 communications. As far as Quarkus endpoints are concerned, they can continue using the same Quarkus Security, OIDC API, annotations such as **@Authenticated**, **SecurityIdentity**, and so on.

For example, assume **Okta** is your SAML 2.0 provider and **Keycloak** is your OIDC provider. Here is a typical sequence explaining how to configure **Keycloak** to broker with the **Okta** SAML 2.0 provider.

First, create a new **SAML2** integration in your **Okta Dashboard/Applications**:



Create a new app integration

Sign-in method

Learn More

OIDC - OpenID Connect
Token-based OAuth 2.0 authentication for Single Sign-On (SSO) through API endpoints. Recommended if you intend to build a custom app integration with the Okta Sign-In Widget.

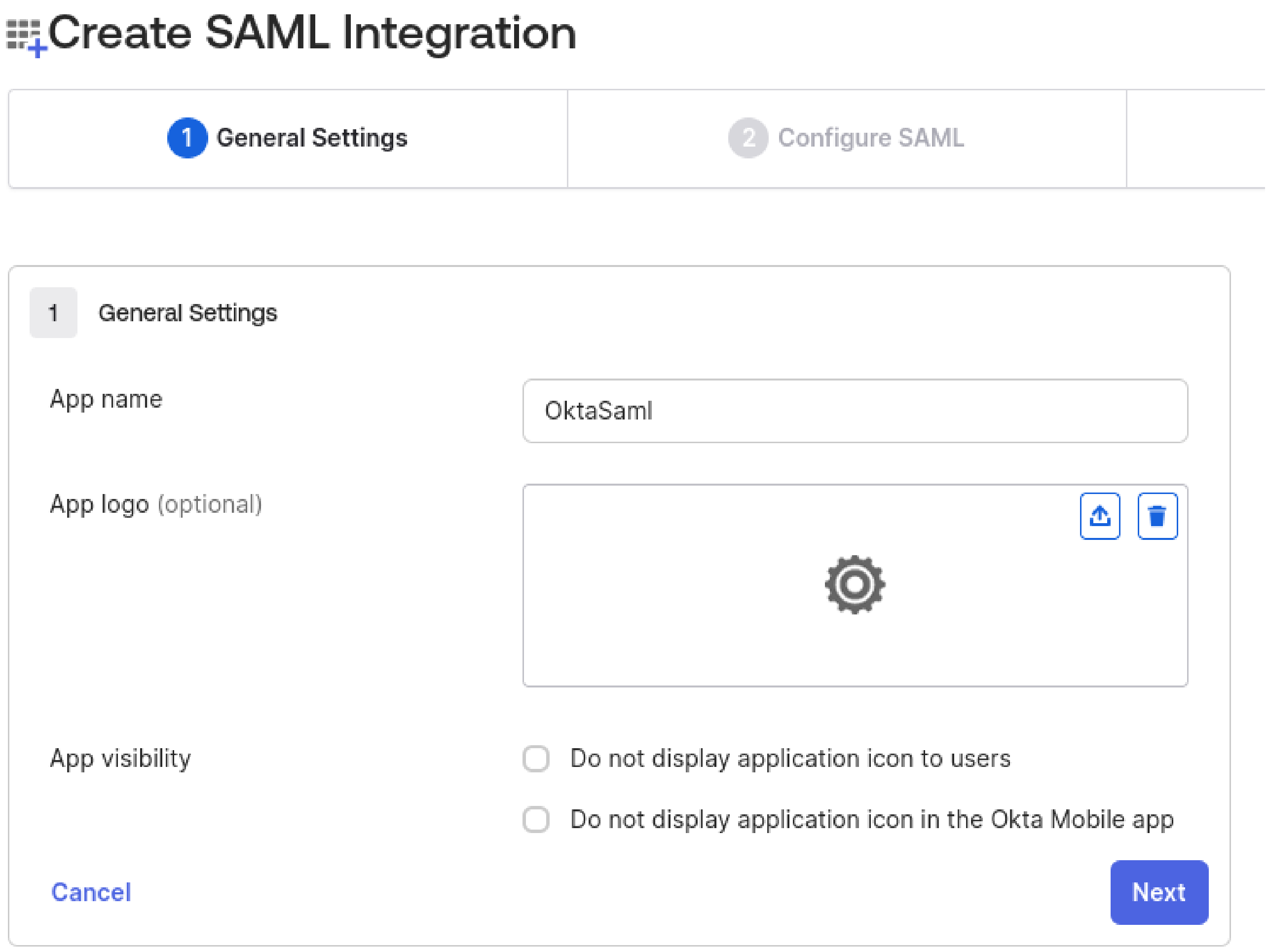
[✓ SAML 2.0]
XML-based open standard for SSO. Use if the Identity Provider for your application only supports SAML.

SWA - Secure Web Authentication
Okta-specific SSO method. Use if your application doesn't support OIDC or SAML.

API Services
Interact with Okta APIs using the scoped OAuth 2.0 access tokens for machine-to-machine authentication.

Cancel Next

For example, name it as **OktaSaml**:



Create SAML Integration

1 General Settings | 2 Configure SAML

1 General Settings

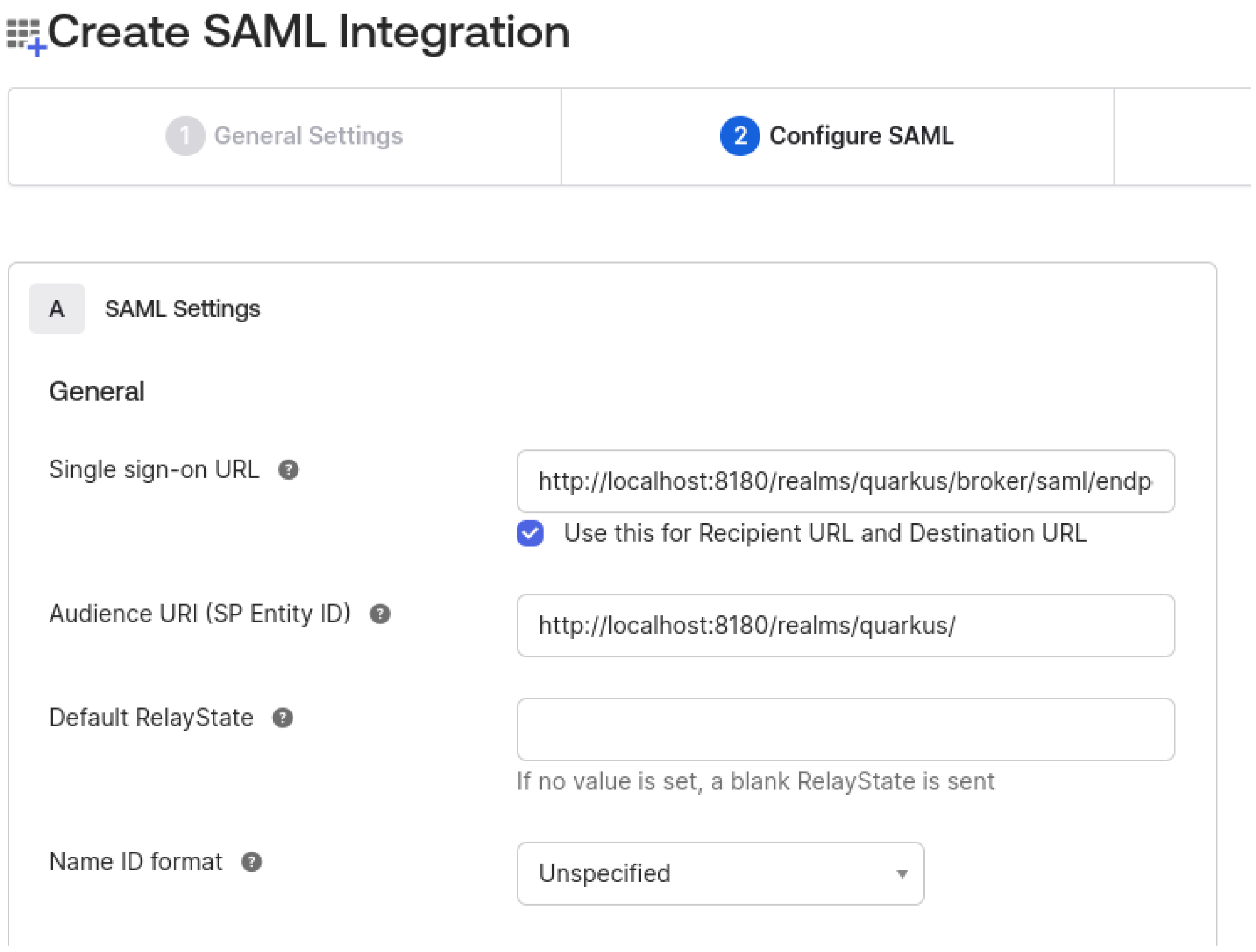
App name: OktaSaml

App logo (optional)

App visibility
- Do not display application icon to users
- Do not display application icon in the Okta Mobile app

Next, configure it to point to a Keycloak SAML broker endpoint. At this point, you need to know the name of the Keycloak realm, for example, **quarkus**, and, assuming that the Keycloak SAML broker alias is **saml**, enter the endpoint address as **http:localhost:8081/realms/quarkus/broker/saml/endpoint**.

Enter the service provider (SP) entity ID as **http:localhost:8081/realms/quarkus**, where **http://localhost:8081** is a Keycloak base address and **saml** is a broker alias:


Create SAML Integration

A SAML Settings

General

Single sign-on URL: http://localhost:8180/realms/quarkus/broker/saml/endp

✓ Use this for Recipient URL and Destination URL

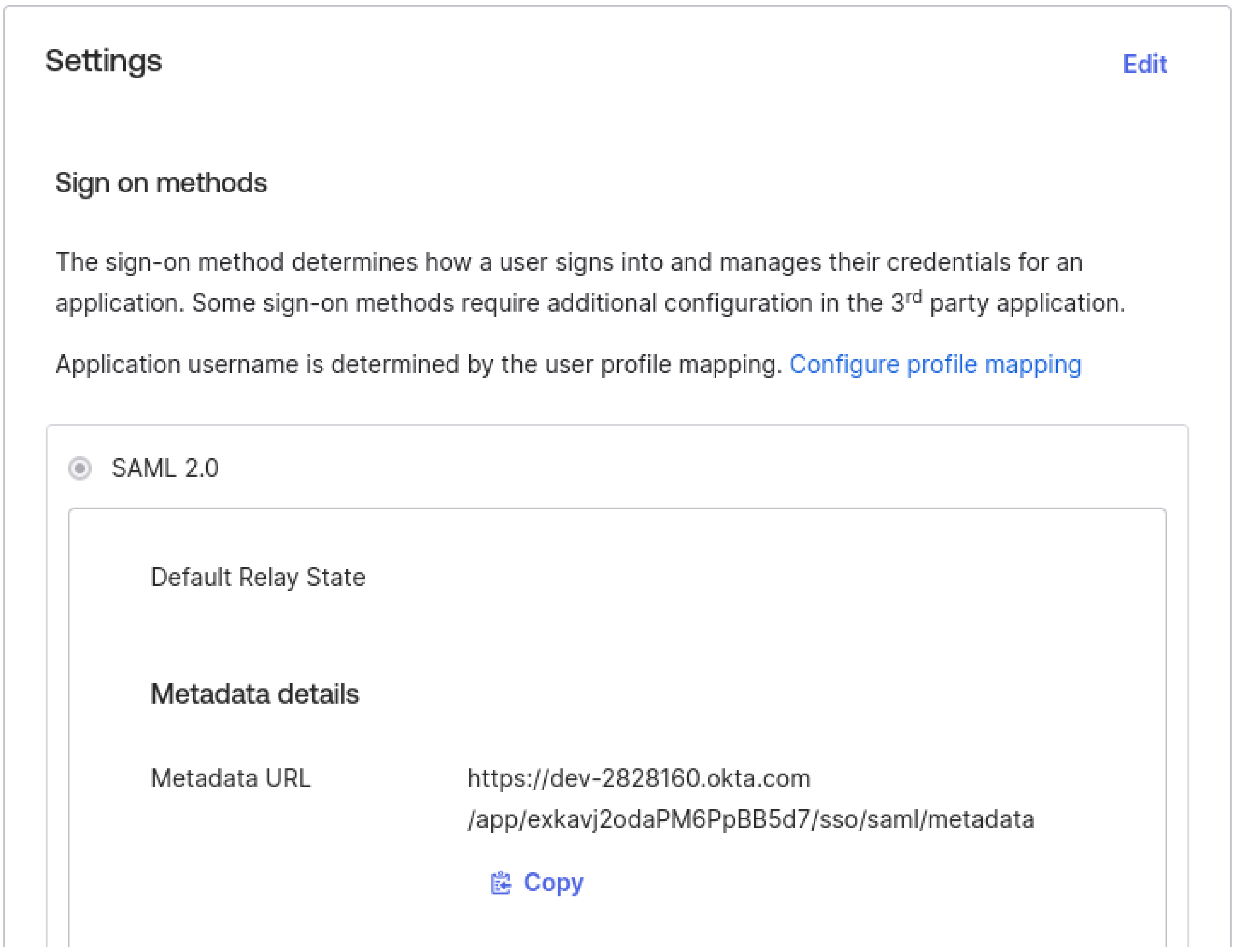
Audience URI (SP Entity ID): http://localhost:8180/realms/quarkus/

Default RelayState:

If no value is set, a blank RelayState is sent

Name ID format: Unspecified

Next, save this SAML integration and note its Metadata URL:


Settings

Edit

Sign on methods

The sign-on method determines how a user signs into and manages their credentials for an application. Some sign-on methods require additional configuration in the 3rd party application.

Application username is determined by the user profile mapping. Configure profile mapping

SAML 2.0

Default Relay State

Metadata details

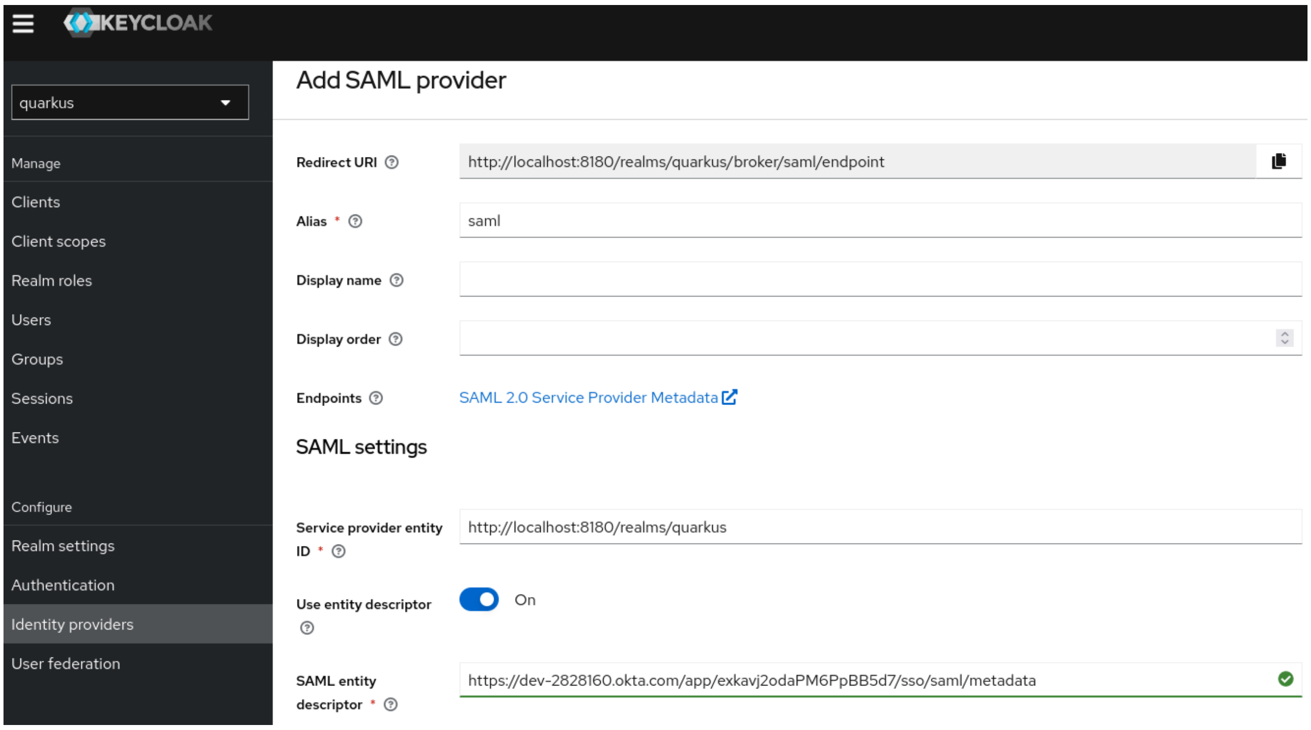
Metadata URL https://dev-2828160.okta.com/app/exkavj2odaPM6PpBB5d7/sso/saml/metadata

Copy

Next, add a SAML provider to Keycloak:

First, as usual, create a new realm or import the existing realm to **Keycloak**. In this case, the realm name has to be **quarkus**.

Now, in the **quarkus** realm properties, navigate to **Identity Providers** and add a new SAML provider:



Add SAML provider

Redirect URI	http://localhost:8180/realms/quarkus/broker/saml/endpoint
Alias *	saml
Display name	
Display order	
Endpoints	SAML 2.0 Service Provider Metadata

SAML settings

Service provider entity ID *	http://localhost:8180/realms/quarkus
Use entity descriptor	On
SAML entity descriptor *	https://dev-2828160.okta.com/app/exkavj2odaPM6PpBB5d7/sso/saml/metadata

Note the alias is set to **saml**, **Redirect URI** is **http:localhost:8081/realms/quarkus/broker/saml/endpoint** and **Service provider entity ID** is **http:localhost:8081/realms/quarkus** - these are the same values you entered when creating the Okta SAML integration in the previous step.

Finally, set **Service entity descriptor** to point to the Okta SAML Integration Metadata URL you noted at the end of the previous step.

Next, if you want, you can register this Keycloak SAML provider as a default provider by navigating to **Authentication/browser/Identity Provider Redirector config** and setting both the **Alias** and **Default Identity Provider** properties to **saml**. If you do not configure it as a default provider then, at authentication time, Keycloak offers 2 options:

- Authenticate with the SAML provider
- Authenticate directly to Keycloak with the name and password

Now, configure the Quarkus OIDC **web-app** application to point to the Keycloak **quarkus** realm, **quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus**. Then, you are ready to start authenticating your Quarkus users to the Okta SAML 2.0 provider by using an OIDC to SAML bridge that is provided by Keycloak OIDC and Okta SAML 2.0 providers.

You can configure other OIDC providers to provide a SAML bridge similarly to how it can be done for Keycloak.

3.5. TESTING

Testing is often tricky when it comes to authentication to a separate OIDC-like server. Quarkus offers several options from mocking to a local run of an OIDC provider.

Start by adding the following dependencies to your test project:

- Using Maven:

```
<dependency>
    <groupId>net.sourceforge.htmlunit</groupId>
    <artifactId>htmlunit</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>*</artifactId>
        </exclusion>
    </exclusions>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-junit5</artifactId>
    <scope>test</scope>
</dependency>
```

- Using Gradle:

```
testImplementation("net.sourceforge.htmlunit:htmlunit")
testImplementation("io.quarkus:quarkus-junit5")
```

3.5.1. Wiremock

Add the following dependency:

- Using Maven:

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

- Using Gradle:

  ```
  testImplementation("io.quarkus:quarkus-test-oidc-server")
  ```

Prepare the REST test endpoints and set **application.properties**. For example:

```
# keycloak.url is set by OidcWiremockTestResource
quarkus.oidc.auth-server-url=${keycloak.url}/realms/quarkus/
quarkus.oidc.client-id=quarkus-web-app
quarkus.oidc.credentials.secret=secret
quarkus.oidc.application-type=web-app
```

Finally, write the test code, for example:

```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

import com.gargoylesoftware.htmlunit.SilentCssErrorHandler;
import com.gargoylesoftware.htmlunit.WebClient;
import com.gargoylesoftware.htmlunit.html.HtmlForm;
import com.gargoylesoftware.htmlunit.html.HtmlPage;

import io.quarkus.test.common.QuarkusTestResource;
import io.quarkus.test.junit.QuarkusTest;
import io.quarkus.test.oidc.server.OidcWiremockTestResource;

@QuarkusTest
@QuarkusTestResource(OidcWiremockTestResource.class)
public class CodeFlowAuthorizationTest {

    @Test
    public void testCodeFlow() throws Exception {
        try (final WebClient webClient = createWebClient()) {
            // the test REST endpoint listens on '/code-flow'
            HtmlPage page = webClient.getPage("http://localhost:8081/code-flow");

            HtmlForm form = page.getFormByName("form");
            // user 'alice' has the 'user' role
            form.getInputByName("username").type("alice");
            form.getInputByName("password").type("alice");

            page = form.getInputByValue("login").click();
```

```
            assertEquals("alice", page.getBody().asText());
        }
    }

    private WebClient createWebClient() {
        WebClient webClient = new WebClient();
        webClient.setCssErrorHandler(new SilentCssErrorHandler());
        return webClient;
    }
}
```

OidcWiremockTestResource recognizes **alice** and **admin** users. The user **alice** has the **user** role only by default - it can be customized with a **quarkus.test.oidc.token.user-roles** system property. The user **admin** has the **user** and **admin** roles by default - it can be customized with a **quarkus.test.oidc.token.admin-roles** system property.

Additionally, **OidcWiremockTestResource** sets the token issuer and audience to **https://service.example.com**, which can be customized with **quarkus.test.oidc.token.issuer** and **quarkus.test.oidc.token.audience** system properties.

OidcWiremockTestResource can be used to emulate all OIDC providers.

3.5.2. Dev Services for Keycloak

Using Dev Services for Keycloak is recommended for integration testing against Keycloak. **Dev Services for Keycloak** will start and initialize a test container: it will create a **quarkus** realm, a **quarkus-app** client (**secret** secret), and add **alice** (**admin** and **user** roles) and **bob** (**user** role) users, where all of these properties can be customized.

First, prepare **application.properties**. You can start with a completely empty **application.properties** file as **Dev Services for Keycloak** will register **quarkus.oidc.auth-server-url** pointing to the running test container as well as **quarkus.oidc.client-id=quarkus-app** and **quarkus.oidc.credentials.secret=secret**.

However, if you already have all the required **quarkus-oidc** properties configured, then you only need to associate **quarkus.oidc.auth-server-url** with the **prod** profile for **Dev Services for Keycloak** to start a container. For example:

```
%prod.quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
```

If a custom realm file has to be imported into Keycloak before running the tests, then you can configure **Dev Services for Keycloak** as follows:

```
%prod.quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
quarkus.keycloak.devservices.realm-path=quarkus-realm.json
```

Finally, write a test code the same way as it is described in the Wiremock section. The only difference is that **@QuarkusTestResource** is no longer needed:

```
@QuarkusTest
public class CodeFlowAuthorizationTest {
}
```

3.5.3. TestSecurity annotation

You can use @TestSecurity and @OidcSecurity annotations to test the **web-app** application endpoint code, which depends on either one of the following injections, or all four:

- ID **JsonWebToken**
- Access **JsonWebToken**
- **UserInfo**
- **OidcConfigurationMetadata**

For more information, see Use TestingSecurity with injected JsonWebToken.

3.5.4. Checking errors in the logs

To see details about the token verification errors, you must enable **io.quarkus.oidc.runtime.OidcProvider TRACE** level logging:

```
quarkus.log.category."io.quarkus.oidc.runtime.OidcProvider".level=TRACE
quarkus.log.category."io.quarkus.oidc.runtime.OidcProvider".min-level=TRACE
```

To see details about the OidcProvider client initialization errors, enable **io.quarkus.oidc.runtime.OidcRecorder TRACE** level logging:

```
quarkus.log.category."io.quarkus.oidc.runtime.OidcRecorder".level=TRACE
quarkus.log.category."io.quarkus.oidc.runtime.OidcRecorder".min-level=TRACE
```

From the **quarkus dev** console, type **j** to change the application global log level.

3.6. REFERENCES

- OIDC configuration properties
- Configuring well-known OpenID Connect providers
- OpenID Connect and OAuth2 client and filters reference guide
- Dev Services for Keycloak
- Choosing between OpenID Connect, SmallRye JWT, and OAuth2 authentication mechanisms
- Combining authentication mechanisms
- Quarkus Security overview
- Keycloak documentation
- OpenID Connect
- JSON Web Token

CHAPTER 4. PROTECT A WEB APPLICATION BY USING OPENID CONNECT (OIDC) AUTHORIZATION CODE FLOW

Discover how to secure application HTTP endpoints by using the Quarkus OpenID Connect (OIDC) authorization code flow mechanism with the Quarkus OIDC extension, providing robust authentication and authorization.

For more information, see OIDC code flow mechanism for protecting web applications .

To learn about how well-known social providers such as Apple, Facebook, GitHub, Google, Mastodon, Microsoft, Twitch, Twitter (X), and Spotify can be used with Quarkus OIDC, see Configuring well-known OpenID Connect providers. See also, Authentication mechanisms in Quarkus .

If you want to protect your service applications by using OIDC Bearer token authentication, see OIDC Bearer token authentication.

4.1. PREREQUISITES

To complete this guide, you need:

- Roughly 15 minutes
- An IDE
- JDK 17+ installed with **JAVA_HOME** configured appropriately
- Apache Maven 3.9.6
- A working container runtime (Docker or Podman)
- Optionally the Quarkus CLI if you want to use it
- Optionally Mandrel or GraalVM installed and configured appropriately if you want to build a native executable (or Docker if you use a native container build)

4.2. ARCHITECTURE

In this example, we build a simple web application with a single page:

- **/index.html**

This page is protected, and only authenticated users can access it.

4.3. SOLUTION

Follow the instructions in the next sections and create the application step by step. Alternatively, you can go right to the completed example.

Clone the Git repository by running the **git clone https://github.com/quarkusio/quarkus-quickstarts.git -b 3.8** command. Alternatively, download an archive.

The solution is located in the **security-openid-connect-web-authentication-quickstart** directory.

4.4. CREATE THE MAVEN PROJECT

First, we need a new project. Create a new project by running the following command:

- Using the Quarkus CLI:

  ```
  quarkus create app org.acme:security-openid-connect-web-authentication-quickstart \
      --extension='resteasy-reactive,oidc' \
      --no-code
  cd security-openid-connect-web-authentication-quickstart
  ```

  To create a Gradle project, add the **--gradle** or **--gradle-kotlin-dsl** option.

  For more information about how to install and use the Quarkus CLI, see the Quarkus CLI guide.

- Using Maven:

  ```
  mvn io.quarkus.platform:quarkus-maven-plugin:3.8.5:create \
      -DprojectGroupId=org.acme \
      -DprojectArtifactId=security-openid-connect-web-authentication-quickstart \
      -Dextensions='resteasy-reactive,oidc' \
      -DnoCode
  cd security-openid-connect-web-authentication-quickstart
  ```

  To create a Gradle project, add the **-DbuildTool=gradle** or **-DbuildTool=gradle-kotlin-dsl** option.

For Windows users:

- If using cmd, (don't use backward slash \ and put everything on the same line)
- If using Powershell, wrap **-D** parameters in double quotes e.g. **"-DprojectArtifactId=security-openid-connect-web-authentication-quickstart"**

If you already have your Quarkus project configured, you can add the **oidc** extension to your project by running the following command in your project base directory:

- Using the Quarkus CLI:

  ```
  quarkus extension add oidc
  ```

- Using Maven:

  ```
  ./mvnw quarkus:add-extension -Dextensions='oidc'
  ```

- Using Gradle:

  ```
  ./gradlew addExtension --extensions='oidc'
  ```

This adds the following dependency to your build file:

- Using Maven:

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

–

- Using Gradle:

```
implementation("io.quarkus:quarkus-oidc")
```

4.5. WRITE THE APPLICATION

Let's write a simple Jakarta REST resource that has all the tokens returned in the authorization code grant response injected:

```
package org.acme.security.openid.connect.web.authentication;

import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;

import org.eclipse.microprofile.jwt.Claims;
import org.eclipse.microprofile.jwt.JsonWebToken;

import io.quarkus.oidc.IdToken;
import io.quarkus.oidc.RefreshToken;

@Path("/tokens")
public class TokenResource {

  /**
   * Injection point for the ID token issued by the OpenID Connect provider
   */
  @Inject
  @IdToken
  JsonWebToken idToken;

  /**
   * Injection point for the access token issued by the OpenID Connect provider
   */
  @Inject
  JsonWebToken accessToken;

  /**
   * Injection point for the refresh token issued by the OpenID Connect provider
   */
  @Inject
  RefreshToken refreshToken;

  /**
   * Returns the tokens available to the application.
   * This endpoint exists only for demonstration purposes.
   * Do not expose these tokens in a real application.
   *
   * @return an HTML page containing the tokens available to the application.
   */
  @GET
  @Produces("text/html")
  public String getTokens() {
        StringBuilder response = new StringBuilder().append("<html>")
                .append("<body>")
                .append("<ul>");


        Object userName = this.idToken.getClaim(Claims.preferred_username);

        if (userName != null) {
            response.append("<li>username: ").append(userName.toString()).append("</li>");
        }

        Object scopes = this.accessToken.getClaim("scope");

        if (scopes != null) {
            response.append("<li>scopes: ").append(scopes.toString()).append("</li>");
        }

        response.append("<li>refresh_token: ").append(refreshToken.getToken() != null).append("</li>");

        return response.append("</ul>").append("</body>").append("</html>").toString();
    }
}
```

This endpoint has ID, access, and refresh tokens injected. It returns a **preferred_username** claim from the ID token, a **scope** claim from the access token, and a refresh token availability status.

You only need to inject the tokens if the endpoint needs to use the ID token to interact with the currently authenticated user or use the access token to access a downstream service on behalf of this user.

For more information, see the Access ID and Access Tokens section of the reference guide.

4.6. CONFIGURE THE APPLICATION

The OIDC extension allows you to define the configuration by using the **application.properties** file in the **src/main/resources** directory.

```
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
quarkus.oidc.client-id=frontend
quarkus.oidc.credentials.secret=secret
quarkus.oidc.application-type=web-app
quarkus.http.auth.permission.authenticated.paths=/*
quarkus.http.auth.permission.authenticated.policy=authenticated
```

This is the simplest configuration you can have when enabling authentication to your application.

The **quarkus.oidc.client-id** property references the **client_id** issued by the OIDC provider, and the **quarkus.oidc.credentials.secret** property sets the client secret.

The **quarkus.oidc.application-type** property is set to **web-app** to tell Quarkus that you want to enable the OIDC authorization code flow so that your users are redirected to the OIDC provider to authenticate.

Finally, the **quarkus.http.auth.permission.authenticated** permission is set to tell Quarkus about the paths you want to protect. In this case, all paths are protected by a policy that ensures only **authenticated** users can access them. For more information, see Security Authorization Guide.

4.7. START AND CONFIGURE THE KEYCLOAK SERVER

To start a Keycloak server, use Docker and run the following command:

```
docker run --name keycloak -e KEYCLOAK_ADMIN=admin -e
KEYCLOAK_ADMIN_PASSWORD=admin -p 8180:8080 quay.io/keycloak/keycloak:
{keycloak.version} start-dev
```

where **keycloak.version** is set to **24.0.0** or later.

You can access your Keycloak Server at localhost:8180.

To access the Keycloak Administration Console, log in as the **admin** user. The username and password are both **admin**.

To create a new realm, import the realm configuration file. For more information, see the Keycloak documentation about how to create and configure a new realm.

4.8. RUN THE APPLICATION IN DEV AND JVM MODES

To run the application in dev mode, use:

- Using the Quarkus CLI:

  ```
  quarkus dev
  ```

- Using Maven:

  ```
  ./mvnw quarkus:dev
  ```

- Using Gradle:

  ```
  ./gradlew --console=plain quarkusDev
  ```

After exploring the application in dev mode, you can run it as a standard Java application.

First, compile it:

- Using the Quarkus CLI:

  ```
  quarkus build
  ```

- Using Maven:

  ```
  ./mvnw install
  ```

- Using Gradle:

  ```
  ./gradlew build
  ```

Then, run it:

```
java -jar target/quarkus-app/quarkus-run.jar
```

4.9. RUN THE APPLICATION IN NATIVE MODE

This same demo can be compiled into native code. No modifications are required.

This implies that you no longer need to install a JVM on your production environment, as the runtime technology is included in the produced binary and optimized to run with minimal resources.

Compilation takes longer, so this step is turned off by default. You can build again by enabling the native build:

- Using the Quarkus CLI:

  ```
  quarkus build --native
  ```

- Using Maven:

  ```
  ./mvnw install -Dnative
  ```

- Using Gradle:

  ```
  ./gradlew build -Dquarkus.package.type=native
  ```

After a while, you can run this binary directly:

```
./target/security-openid-connect-web-authentication-quickstart-runner
```

4.10. TEST THE APPLICATION

To test the application, open your browser and access the following URL:

- http://localhost:8080/tokens

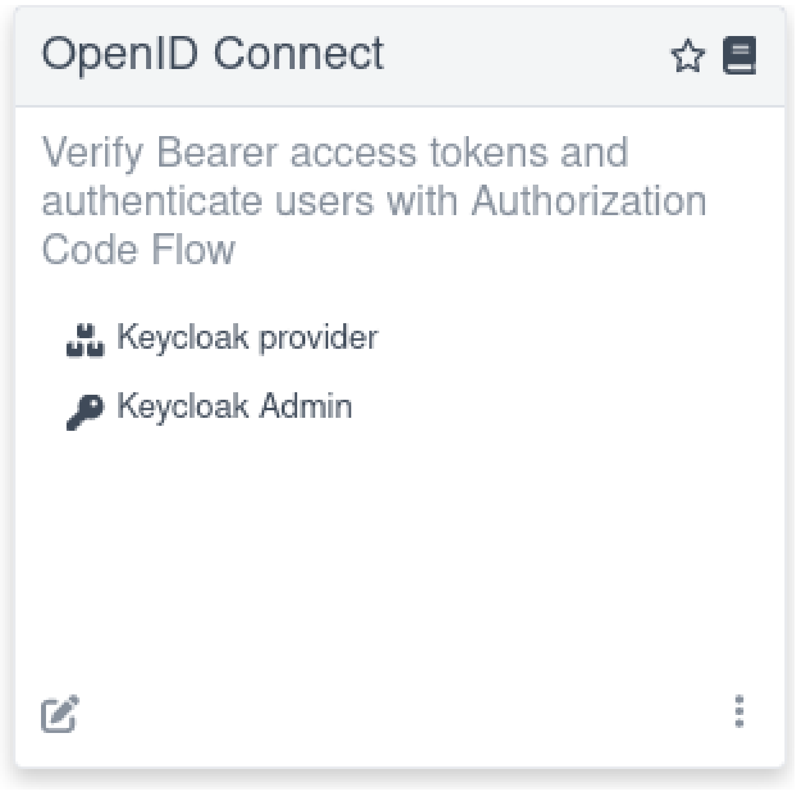
If everything works as expected, you are redirected to the Keycloak server to authenticate.

To authenticate to the application, enter the following credentials at the Keycloak login page:

- Username: **alice**
- Password: **alice**

After clicking the **Login** button, you are redirected back to the application, and a session cookie will be created.

The session for this demo is valid for a short period of time and, on every page refresh, you will be asked to re-authenticate. For information about how to increase the session timeouts, see the Keycloak session timeout documentation. For example, you can access the Keycloak Admin console directly from the dev UI by clicking the **Keycloak Admin** link if you use Dev Services for Keycloak in dev mode:


OpenID Connect
Verify Bearer access tokens and authenticate users with Authorization Code Flow
Keycloak provider
Keycloak Admin

For more information about writing the integration tests that depend on **Dev Services for Keycloak**, see the Dev Services for Keycloak section.

4.11. SUMMARY

You have learned how to set up and use the OIDC authorization code flow mechanism to protect and test application HTTP endpoints. After you have completed this tutorial, explore OIDC Bearer token authentication and other authentication mechanisms.

4.12. REFERENCES

- Quarkus Security overview
- OIDC code flow mechanism for protecting web applications
- Configuring well-known OpenID Connect providers
- OpenID Connect and OAuth2 Client and Filters reference guide
- Dev Services for Keycloak
- Sign and encrypt JWT tokens with SmallRye JWT Build
- Choosing between OpenID Connect, SmallRye JWT, and OAuth2 authentication mechanisms
- Keycloak Documentation
- Protect Quarkus web application by using Auth0 OpenID Connect provider
- OpenID Connect
- JSON Web Token

CHAPTER 5. USING OPENID CONNECT (OIDC) MULTITENANCY

This guide demonstrates how your OpenID Connect (OIDC) application can support multitenancy to serve multiple tenants from a single application. These tenants can be distinct realms or security domains within the same OIDC provider or even distinct OIDC providers.

Each customer functions as a distinct tenant when serving multiple customers from the same application, such as in a SaaS environment. By enabling multitenancy support to your applications, you can support distinct authentication policies for each tenant, even authenticating against different OIDC providers, such as Keycloak and Google.

To authorize a tenant by using Bearer Token Authorization, see the OpenID Connect (OIDC) Bearer token authentication guide.

To authenticate and authorize a tenant by using the OIDC authorization code flow, read the OpenID Connect authorization code flow mechanism for protecting web applications guide.

Also, see the OpenID Connect (OIDC) configuration properties reference guide.

5.1. PREREQUISITES

To complete this guide, you need:

- Roughly 15 minutes
- An IDE
- JDK 17+ installed with **JAVA_HOME** configured appropriately
- Apache Maven 3.9.6
- A working container runtime (Docker or Podman)
- Optionally the Quarkus CLI if you want to use it
- Optionally Mandrel or GraalVM installed and configured appropriately if you want to build a native executable (or Docker if you use a native container build)
- jq tool

5.2. ARCHITECTURE

In this example, we build a very simple application that supports two resource methods:

- **/{tenant}**

This resource returns information obtained from the ID token issued by the OIDC provider about the authenticated user and the current tenant.

- **/{tenant}/bearer**

This resource returns information obtained from the Access Token issued by the OIDC provider about the authenticated user and the current tenant.

5.3. SOLUTION

For a thorough understanding, we recommend you build the application by following the upcoming step-by-step instructions.

Alternatively, if you prefer to start with the completed example, clone the Git repository: **git clone https://github.com/quarkusio/quarkus-quickstarts.git -b 3.8**, or download an archive.

The solution is located in the **security-openid-connect-multi-tenancy-quickstart** directory.

5.4. CREATING THE MAVEN PROJECT

First, we need a new project. Create a new project with the following command:

- Using the Quarkus CLI:

  ```
  quarkus create app org.acme:security-openid-connect-multi-tenancy-quickstart \
      --extension='oidc,resteasy-reactive-jackson' \
      --no-code
  cd security-openid-connect-multi-tenancy-quickstart
  ```

  To create a Gradle project, add the **--gradle** or **--gradle-kotlin-dsl** option.

  For more information about how to install and use the Quarkus CLI, see the Quarkus CLI guide.

- Using Maven:

  ```
  mvn io.quarkus.platform:quarkus-maven-plugin:3.8.5:create \
      -DprojectGroupId=org.acme \
      -DprojectArtifactId=security-openid-connect-multi-tenancy-quickstart \
      -Dextensions='oidc,resteasy-reactive-jackson' \
      -DnoCode
  cd security-openid-connect-multi-tenancy-quickstart
  ```

  To create a Gradle project, add the **-DbuildTool=gradle** or **-DbuildTool=gradle-kotlin-dsl** option.

For Windows users:

- If using cmd, (don't use backward slash \ and put everything on the same line)
- If using Powershell, wrap **-D** parameters in double quotes e.g. **"-DprojectArtifactId=security-openid-connect-multi-tenancy-quickstart"**

If you already have your Quarkus project configured, add the **oidc** extension to your project by running the following command in your project base directory:

- Using the Quarkus CLI:

  ```
  quarkus extension add oidc
  ```

- Using Maven:

  ```
  ./mvnw quarkus:add-extension -Dextensions='oidc'
  ```

- Using Gradle:

```
./gradlew addExtension --extensions='oidc'
```

This adds the following to your build file:

- Using Maven:

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

- Using Gradle:

```
implementation("io.quarkus:quarkus-oidc")
```

5.5. WRITING THE APPLICATION

Start by implementing the /**{tenant}** endpoint. As you can see from the source code below, it is just a regular Jakarta REST resource:

```
package org.acme.quickstart.oidc;

import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;

import org.eclipse.microprofile.jwt.JsonWebToken;

import io.quarkus.oidc.IdToken;

@Path("/{tenant}")
public class HomeResource {
    /**
     * Injection point for the ID Token issued by the OIDC provider.
     */
    @Inject
    @IdToken
    JsonWebToken idToken;

    /**
     * Injection point for the Access Token issued by the OIDC provider.
     */
    @Inject
    JsonWebToken accessToken;

    /**
     * Returns the ID Token info.
     * This endpoint exists only for demonstration purposes.
     * Do not expose this token in a real application.
     *
     * @return ID Token info
     */
    @GET
    @Produces("text/html")
    public String getIdTokenInfo() {
        StringBuilder response = new StringBuilder().append("<html>")
                .append("<body>");

        response.append("<h2>Welcome, ").append(this.idToken.getClaim("email").toString()).append("
</h2>\n");
        response.append("<h3>You are accessing the application within tenant
<b>").append(idToken.getIssuer()).append(" boundaries</b></h3>");

        return response.append("</body>").append("</html>").toString();
    }

    /**
     * Returns the Access Token info.
     * This endpoint exists only for demonstration purposes.
     * Do not expose this token in a real application.
     *
     * @return Access Token info
     */
    @GET
    @Produces("text/html")
    @Path("bearer")
    public String getAccessTokenInfo() {
        StringBuilder response = new StringBuilder().append("<html>")
                .append("<body>");

        response.append("<h2>Welcome,
").append(this.accessToken.getClaim("email").toString()).append("</h2>\n");
        response.append("<h3>You are accessing the application within tenant
<b>").append(accessToken.getIssuer()).append(" boundaries</b></h3>");

        return response.append("</body>").append("</html>").toString();
    }
}
```

To resolve the tenant from incoming requests and map it to a specific **quarkus-oidc** tenant configuration in **application.properties**, create an implementation for the **io.quarkus.oidc.TenantConfigResolver** interface, which can dynamically resolve tenant configurations:

```
package org.acme.quickstart.oidc;

import jakarta.enterprise.context.ApplicationScoped;

import org.eclipse.microprofile.config.ConfigProvider;

import io.quarkus.oidc.OidcRequestContext;
import io.quarkus.oidc.OidcTenantConfig;
import io.quarkus.oidc.OidcTenantConfig.ApplicationType;
import io.quarkus.oidc.TenantConfigResolver;
import io.smallrye.mutiny.Uni;
import io.vertx.ext.web.RoutingContext;

@ApplicationScoped
```

```
public class CustomTenantResolver implements TenantConfigResolver {

    @Override
    public Uni<OidcTenantConfig> resolve(RoutingContext context,
OidcRequestContext<OidcTenantConfig> requestContext) {
        String path = context.request().path();

        if (path.startsWith("/tenant-a")) {
           String keycloakUrl = ConfigProvider.getConfig().getValue("keycloak.url", String.class);

            OidcTenantConfig config = new OidcTenantConfig();
            config.setTenantId("tenant-a");
            config.setAuthServerUrl(keycloakUrl + "/realms/tenant-a");
            config.setClientId("multi-tenant-client");
            config.getCredentials().setSecret("secret");
            config.setApplicationType(ApplicationType.HYBRID);
            return Uni.createFrom().item(config);
        } else {
            // resolve to default tenant config
            return Uni.createFrom().nullItem();
        }
    }
}
```

In the preceding implementation, tenants are resolved from the request path. If no tenant can be inferred, **null** is returned to indicate that the default tenant configuration should be used.

The **tenant-a** application type is **hybrid**; it can accept HTTP bearer tokens if provided. Otherwise, it initiates an authorization code flow when authentication is required.

5.6. CONFIGURING THE APPLICATION

```
# Default tenant configuration
%prod.quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
quarkus.oidc.client-id=multi-tenant-client
quarkus.oidc.application-type=web-app

# Tenant A configuration is created dynamically in CustomTenantConfigResolver

# HTTP security configuration
quarkus.http.auth.permission.authenticated.paths=/*
quarkus.http.auth.permission.authenticated.policy=authenticated
```

The first configuration is the default tenant configuration that should be used when the tenant cannot be inferred from the request. Be aware that a **%prod** profile prefix is used with **quarkus.oidc.auth-server-url** to support testing a multitenant application with Dev Services For Keycloak. This configuration uses a Keycloak instance to authenticate users.

The second configuration, provided by **TenantConfigResolver**, is used when an incoming request is mapped to the **tenant-a** tenant.

Both configurations map to the same Keycloak server instance while using distinct **realms**.

Alternatively, you can configure the tenant **tenant-a** directly in **application.properties**:

```
# Default tenant configuration
%prod.quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
quarkus.oidc.client-id=multi-tenant-client
quarkus.oidc.application-type=web-app

# Tenant A configuration
quarkus.oidc.tenant-a.auth-server-url=http://localhost:8180/realms/tenant-a
quarkus.oidc.tenant-a.client-id=multi-tenant-client
quarkus.oidc.tenant-a.application-type=web-app

# HTTP security configuration
quarkus.http.auth.permission.authenticated.paths=/*
quarkus.http.auth.permission.authenticated.policy=authenticated
```

In that case, also use a custom **TenantConfigResolver** to resolve it:

```
package org.acme.quickstart.oidc;

import jakarta.enterprise.context.ApplicationScoped;

import io.quarkus.oidc.TenantResolver;
import io.vertx.ext.web.RoutingContext;

@ApplicationScoped
public class CustomTenantResolver implements TenantResolver {

    @Override
    public String resolve(RoutingContext context) {
        String path = context.request().path();
        String[] parts = path.split("/");

        if (parts.length == 0) {
            //Resolve to default tenant configuration
            return null;
        }

        return parts[1];
    }
}
```

You can define multiple tenants in your configuration file. To map them correctly when resolving a tenant from your **TenantResolver** implementation, ensure each has a unique alias.

However, using a static tenant resolution, which involves configuring tenants in **application.properties** and resolving them with **TenantResolver**, does not work for testing endpoints with Dev Services for Keycloak because it does not know how the requests are be mapped to individual tenants, and cannot dynamically provide tenant-specific **quarkus.oidc.<tenant-id>.auth-server-url** values. Therefore, using **%prod** prefixes with tenant-specific URLs within **application.properties** does not work in both test and development modes.

NOTE

When a current tenant represents an OIDC **web-app** application, the current **io.vertx.ext.web.RoutingContext** contains a **tenant-id** attribute by the time the custom tenant resolver has been called for all the requests completing the code authentication flow and the already authenticated requests, when either a tenant-specific state or session cookie already exists. Therefore, when working with multiple OIDC providers, you only need a path-specific check to resolve a tenant id if the **RoutingContext** does not have the **tenant-id** attribute set, for example:

```
package org.acme.quickstart.oidc;

import jakarta.enterprise.context.ApplicationScoped;

import io.quarkus.oidc.TenantResolver;
import io.vertx.ext.web.RoutingContext;

@ApplicationScoped
public class CustomTenantResolver implements TenantResolver {

    @Override
    public String resolve(RoutingContext context) {
        String tenantId = context.get("tenant-id");
        if (tenantId != null) {
            return tenantId;
        } else {
            // Initial login request
            String path = context.request().path();
            String[] parts = path.split("/");

            if (parts.length == 0) {
                //Resolve to default tenant configuration
                return null;
            }
            return parts[1];
        }
    }
}
```

This is how Quarkus OIDC resolves static custom tenants if no custom **TenantResolver** is registered.

A similar technique can be used with **TenantConfigResolver**, where a **tenant-id** provided in the context can return **OidcTenantConfig** already prepared with the previous request.

NOTE

If you also use Hibernate ORM multitenancy or MongoDB with Panache multitenancy and both tenant ids are the same and must be extracted from the Vert.x **RoutingContext**, you can pass the tenant id from the OIDC Tenant Resolver to the Hibernate ORM Tenant Resolver or MongoDB with Panache Mongo Database Resolver as a **RoutingContext** attribute, for example:

```
public class CustomTenantResolver implements TenantResolver {

    @Override
    public String resolve(RoutingContext context) {
        String tenantId = extractTenantId(context);
        context.put("tenantId", tenantId);
        return tenantId;
    }
}
```

5.7. STARTING AND CONFIGURING THE KEYCLOAK SERVER

To start a Keycloak server, you can use Docker and run the following command:

```
docker run --name keycloak -e KEYCLOAK_ADMIN=admin -e
KEYCLOAK_ADMIN_PASSWORD=admin -p 8180:8080 quay.io/keycloak/keycloak:
{keycloak.version} start-dev
```

where **keycloak.version** is set to **24.0.0** or higher.

Access your Keycloak server at localhost:8180.

Log in as the **admin** user to access the Keycloak administration console. The username and password are both **admin**.

Now, import the realms for the two tenants:

- Import the default-tenant-realm.json to create the default realm.
- Import the tenant-a-realm.json to create the realm for the tenant **tenant-a**.

For more information, see the Keycloak documentation about how to create a new realm.

5.8. RUNNING AND USING THE APPLICATION

5.8.1. Running in developer mode

To run the microservice in dev mode, use:

- Using the Quarkus CLI:

  ```
  quarkus dev
  ```

- Using Maven:

```
./mvnw quarkus:dev
```

- Using Gradle:

```
./gradlew --console=plain quarkusDev
```

5.8.2. Running in JVM mode

After exploring the application in dev mode, you can run it as a standard Java application.

First, compile it:

- Using the Quarkus CLI:

```
quarkus build
```

- Using Maven:

```
./mvnw install
```

- Using Gradle:

```
./gradlew build
```

Then run it:

```
java -jar target/quarkus-app/quarkus-run.jar
```

5.8.3. Running in native mode

This same demo can be compiled into native code; no modifications are required.

This implies that you no longer need to install a JVM on your production environment, as the runtime technology is included in the produced binary, and optimized to run with minimal resources.

Compilation takes a bit longer, so this step is turned off by default; let’s build again by enabling the native build:

- Using the Quarkus CLI:

```
quarkus build --native
```

- Using Maven:

```
./mvnw install -Dnative
```

- Using Gradle:

```
./gradlew build -Dquarkus.package.type=native
```

After a little while, you can run this binary directly:

_

```
./target/security-openid-connect-multi-tenancy-quickstart-runner
```

5.9. TEST THE APPLICATION

5.9.1. Use the browser

To test the application, open your browser and access the following URL:

- http://localhost:8080/default

If everything works as expected, you are redirected to the Keycloak server to authenticate. Be aware that the requested path defines a **default** tenant, which we don't have mapped in the configuration file. In this case, the default configuration is used.

To authenticate to the application, enter the following credentials in the Keycloak login page:

- Username: **alice**
- Password: **alice**

After clicking the **Login** button, you are redirected back to the application.

If you try now to access the application at the following URL:

- http://localhost:8080/tenant-a

You are redirected again to the Keycloak login page. However, this time, you are going to authenticate by using a different realm.

In both cases, the landing page shows the user's name and email if the user is successfully authenticated. Although **alice** exists in both tenants, the application treats them as distinct users in separate realms.

5.10. STATIC TENANT CONFIGURATION RESOLUTION

When you set multiple tenant configurations in the **application.properties** file, you only need to specify how the tenant identifier gets resolved. To configure the resolution of the tenant identifier, use one of the following options:

- Resolve with **TenantResolver**
- Default resolution
- Resolve with annotations

These tenant resolution options are tried in the order they are listed until the tenant id gets resolved. If the tenant id remains unresolved (**null**), the default (unnamed) tenant configuration is selected.

5.10.1. Resolve with TenantResolver

The following **application.properties** example shows how you can resolve the tenant identifier of two tenants named **a** and **b** by using the **TenantResolver** method:

```
# Tenant 'a' configuration
```

```
quarkus.oidc.a.auth-server-url=http://localhost:8180/realms/quarkus-a
quarkus.oidc.a.client-id=client-a
quarkus.oidc.a.credentials.secret=client-a-secret

# Tenant 'b' configuration
quarkus.oidc.b.auth-server-url=http://localhost:8180/realms/quarkus-b
quarkus.oidc.b.client-id=client-b
quarkus.oidc.b.credentials.secret=client-b-secret
```

You can return the tenant id of either **a** or **b** from **quarkus.oidc.TenantResolver**:

```
import quarkus.oidc.TenantResolver;

public class CustomTenantResolver implements TenantResolver {

    @Override
    public String resolve(RoutingContext context) {
        String path = context.request().path();
        if (path.endsWith("a")) {
            return "a";
        } else if (path.endsWith("b")) {
            return "b";
        } else {
            // default tenant
            return null;
        }
    }
}
```

In this example, the value of the last request path segment is a tenant id, but if required, you can implement a more complex tenant identifier resolution logic.

5.10.2. Default resolution

The default resolution for a tenant identifier is convention based, whereby the authentication request must include the tenant identifier in the last segment of the request path.

The following **application.properties** example shows how you can configure two tenants named **google** and **github**:

```
# Tenant 'google' configuration
quarkus.oidc.google.provider=google
quarkus.oidc.google.client-id=${google-client-id}
quarkus.oidc.google.credentials.secret=${google-client-secret}
quarkus.oidc.google.authentication.redirect-path=/signed-in

# Tenant 'github' configuration
quarkus.oidc.github.provider=google
quarkus.oidc.github.client-id=${github-client-id}
quarkus.oidc.github.credentials.secret=${github-client-secret}
quarkus.oidc.github.authentication.redirect-path=/signed-in
```

In the provided example, both tenants configure OIDC **web-app** applications to use an authorization code flow to authenticate users and require session cookies to be generated after authentication. After Google or GitHub authenticates the current user, the user gets returned to the **/signed-in** area for authenticated users, such as a secured resource path on the JAX-RS endpoint.

Finally, to complete the default tenant resolution, set the following configuration property:

```
quarkus.http.auth.permission.login.paths=/google,/github
quarkus.http.auth.permission.login.policy=authenticated
```

If the endpoint is running on **http://localhost:8080**, you can also provide UI options for users to log in to either **http://localhost:8080/google** or **http://localhost:8080/github**, without having to add specific **/google** or **/github** JAX-RS resource paths. Tenant identifiers are also recorded in the session cookie names after the authentication is completed. Therefore, authenticated users can access the secured application area without requiring either the **google** or **github** path values to be included in the secured URL.

Default resolution can also work for Bearer token authentication. Still, it might be less practical because a tenant identifier must always be set as the last path segment value.

5.10.3. Resolve with annotations

You can use the **io.quarkus.oidc.Tenant** annotation for resolving the tenant identifiers as an alternative to using **io.quarkus.oidc.TenantResolver**.

NOTE

Proactive HTTP authentication must be disabled (**quarkus.http.auth.proactive=false**) for this to work. For more information, see the Proactive authentication guide.

Assuming your application supports two OIDC tenants, the **hr** and default tenants, all resource methods and classes carrying **@Tenant("hr")** are authenticated by using the OIDC provider configured by **quarkus.oidc.hr.auth-server-url**. In contrast, all other classes and methods are still authenticated by using the default OIDC provider.

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

import io.quarkus.oidc.Tenant;
import io.quarkus.security.Authenticated;

@Authenticated
@Path("/api/hello")
public class HelloResource {

    @Tenant("hr") 1
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayHello() {
        return "Hello!";
    }
}
```

1 The **io.quarkus.oidc.Tenant** annotation must be placed on either the resource class or resource method.

5.11. DYNAMIC TENANT CONFIGURATION RESOLUTION

If you need a more dynamic configuration for the different tenants you want to support and don't want to end up with multiple entries in your configuration file, you can use the **io.quarkus.oidc.TenantConfigResolver**.

This interface allows you to dynamically create tenant configurations at runtime:

```
package io.quarkus.it.keycloak;

import jakarta.enterprise.context.ApplicationScoped;
import java.util.function.Supplier;

import io.smallrye.mutiny.Uni;
import io.quarkus.oidc.OidcRequestContext;
import io.quarkus.oidc.OidcTenantConfig;
import io.quarkus.oidc.TenantConfigResolver;
import io.vertx.ext.web.RoutingContext;

@ApplicationScoped
public class CustomTenantConfigResolver implements TenantConfigResolver {

    @Override
    public Uni<OidcTenantConfig> resolve(RoutingContext context,
OidcRequestContext<OidcTenantConfig> requestContext) {
        String path = context.request().path();
        String[] parts = path.split("/");

        if (parts.length == 0) {
            //Resolve to default tenant configuration
            return null;
        }

        if ("tenant-c".equals(parts[1])) {
            // Do 'return requestContext.runBlocking(createTenantConfig());'
            // if a blocking call is required to create a tenant config,
            return Uni.createFromItem(createTenantConfig());
        }

        //Resolve to default tenant configuration
        return null;
    }

    private Supplier<OidcTenantConfig> createTenantConfig() {
        final OidcTenantConfig config = new OidcTenantConfig();

        config.setTenantId("tenant-c");
        config.setAuthServerUrl("http://localhost:8180/realms/tenant-c");
        config.setClientId("multi-tenant-client");
        OidcTenantConfig.Credentials credentials = new OidcTenantConfig.Credentials();

        credentials.setSecret("my-secret");

        config.setCredentials(credentials);

        // Any other setting supported by the quarkus-oidc extension
        return () -> config;
    }
}
```

The **OidcTenantConfig** returned by this method is the same one used to parse the **oidc** namespace configuration from the **application.properties**. You can populate it by using any settings supported by the **quarkus-oidc** extension.

If the dynamic tenant resolver returns **null**, a Static tenant configuration resolution is attempted next.

5.11.1. Tenant resolution for OIDC web-app applications

The simplest option for resolving the OIDC **web-app** application configuration is to follow the steps described in the Default resolution section.

Try one of the options below if the default resolution strategy does not work for your application setup.

Several options are available for selecting the tenant configuration that should be used to secure the current HTTP request for both **service** and **web-app** OIDC applications, such as:

- Check the URL paths. For example, a **tenant-service** configuration must be used for the /**service** paths, while a **tenant-manage** configuration must be used for the /**management** paths.
- Check the HTTP headers. For example, with a URL path always being /**service**, a header such as **Realm: service** or **Realm: management** can help to select between the **tenant-service** and **tenant-manage** configurations.
- Check the URL query parameters. It can work similarly to the way the headers are used to select the tenant configuration.

All these options can be easily implemented with the custom **TenantResolver** and **TenantConfigResolver** implementations for the OIDC **service** applications.

However, due to an HTTP redirect required to complete the code authentication flow for the OIDC **web-app** applications, a custom HTTP cookie might be needed to select the same tenant configuration before and after this redirect request because:

- The URL path might not be the same after the redirect request if a single redirect URL has been registered in the OIDC provider. The original request path can be restored after the tenant configuration has been resolved.
- The HTTP headers used during the original request are unavailable after the redirect.
- The custom URL query parameters are restored after the redirect but only after the tenant configuration is resolved.

One option to ensure the information for resolving the tenant configurations for **web-app** applications is available before and after the redirect is to use a cookie, for example:

```
package org.acme.quickstart.oidc;

import java.util.List;

import jakarta.enterprise.context.ApplicationScoped;
import io.quarkus.oidc.TenantResolver;
import io.vertx.core.http.Cookie;
import io.vertx.ext.web.RoutingContext;

@ApplicationScoped
public class CustomTenantResolver implements TenantResolver {

    @Override
    public String resolve(RoutingContext context) {
        List<String> tenantIdQuery = context.queryParam("tenantId");
        if (!tenantIdQuery.isEmpty()) {
            String tenantId = tenantIdQuery.get(0);
            context.response().addCookie(Cookie.cookie("tenant", tenantId));
            return tenantId;
        } else if (!context.request().cookies("tenant").isEmpty()) {
            return context.request().getCookie("tenant").getValue();
        }

        return null;
    }
}
```

5.12. DISABLING TENANT CONFIGURATIONS

Custom **TenantResolver** and **TenantConfigResolver** implementations might return **null** if no tenant can be inferred from the current request and a fallback to the default tenant configuration is required.

If you expect the custom resolvers always to resolve a tenant, you do not need to configure the default tenant resolution.

- To turn off the default tenant configuration, set **quarkus.oidc.tenant-enabled=false**.

NOTE

The default tenant configuration is automatically disabled when **quarkus.oidc.auth-server-url** is not configured, but either custom tenant configurations are available or **TenantConfigResolver** is registered.

Be aware that tenant-specific configurations can also be disabled, for example: **quarkus.oidc.tenant-a.tenant-enabled=false**.

5.13. REFERENCES

- OIDC configuration properties
- Keycloak Documentation
- OpenID Connect
- JSON Web Token
- Google OpenID Connect
- Quarkus Security overview

CHAPTER 6. OPENID CONNECT (OIDC) CONFIGURATION PROPERTIES

As a Quarkus developer, you configure the Quarkus OpenID Connect (OIDC) extension by setting the following properties in the **src/main/resources/application.properties** file.

🔒 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
🔒 quarkus.keycloak.devservices.enabled Flag to enable (default) or disable Dev Services. When enabled, Dev Services for Keycloak automatically configures and starts Keycloak in Dev or Test mode, and when Docker is running. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_ENABLED**	boolean	**true**
🔒 quarkus.keycloak.devservices.image-name The container image name for Dev Services providers. Defaults to a Quarkus-based Keycloak image. For a WildFly-based distribution, use an image like **quay.io/keycloak/keycloak:19.0.3-legacy**. Keycloak Quarkus and WildFly images are initialized differently. Dev Services for Keycloak will assume it is a Keycloak Quarkus image unless the image version ends with **-legacy**. Override with **quarkus.keycloak.devservices.keycloak-x-image**. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_IMAGE_NAME**	string	**quay.io/keycloak/keycloak:24.0.4**
🔒 quarkus.keycloak.devservices.keycloak-x-image Indicates if a Keycloak-X image is used. By default, the image is identified by **keycloak-x** in the image name. For custom images, override with **quarkus.keycloak.devservices.keycloak-x-image**. You do not need to set this property if the default check works. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_KEYCLOAK_X_IMAGE**	boolean	
🔒 quarkus.keycloak.devservices.shared Determines if the Keycloak container is shared. When shared, Quarkus uses label-based service discovery to find and reuse a running Keycloak container, so a second one is not started. Otherwise, if a matching container is not is found, a new container is started. The service discovery uses the **quarkus-dev-service-label** label, whose value is set by the **service-name** property. Container sharing is available only in dev mode. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_SHARED**	boolean	**true**

🔒 **quarkus.keycloak.devservices.service-name** The value of the **quarkus-dev-service-keycloak** label for identifying the Keycloak container. Used in shared mode to locate an existing container with this label. If not found, a new container is initialized with this label. Applicable only in dev mode. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_SERVICE_NAME**	string	**quarkus**
🔒 **quarkus.keycloak.devservices.realm-path** A comma-separated list of class or file system paths to Keycloak realm files. This list is used to initialize Keycloak. The first value in this list is used to initialize default tenant connection properties. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_REALM_PATH**	list of string	
🔒 **quarkus.keycloak.devservices.java-opts** The **JAVA_OPTS** passed to the keycloak JVM Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_JAVA_OPTS**	string	
🔒 **quarkus.keycloak.devservices.show-logs** Show Keycloak log messages with a "Keycloak:" prefix. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_SHOW_LOGS**	boolean	**false**
🔒 **quarkus.keycloak.devservices.start-command** Keycloak start command. Use this property to experiment with Keycloak start options, see **https://www.keycloak.org/server/all-config**. Note, it is ignored when loading legacy Keycloak WildFly images. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_START_COMMAND**	string	
🔒 **quarkus.keycloak.devservices.realm-name** The name of the Keycloak realm. This property is used to create the realm if the realm file pointed to by the **realm-path** property does not exist. The default value is **quarkus** in this case. It is recommended to always set this property so that Dev Services for Keycloak can identify the realm name without parsing the realm file. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_REALM_NAME**	string	

🔒 quarkus.keycloak.devservices.create-realm Specifies whether to create the Keycloak realm when no realm file is found at the **realm-path**. Set to **false** if the realm is to be created using either the Keycloak Administration Console or the Keycloak Admin API provided by **io.quarkus.test.common.QuarkusTestResourceLifecycleManager**. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_CREATE_REALM**	boolean	**true**
🔒 quarkus.keycloak.devservices.port The specific port for the dev service to listen on. If not specified, a random port is selected. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_PORT**	int	
🔒 quarkus.keycloak.devservices.resource-aliases Aliases to additional class or file system resources that are used to initialize Keycloak. Each map entry represents a mapping between an alias and a class or file system resource path. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_RESOURCE_ALIASES**	**Map<String,String>**	
🔒 quarkus.keycloak.devservices.resource-mappings Additional class or file system resources that are used to initialize Keycloak. Each map entry represents a mapping between a class or file system resource path alias and the Keycloak container location. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_RESOURCE_MAPPINGS**	**Map<String,String>**	
🔒 quarkus.keycloak.devservices.users A map of Keycloak usernames to passwords. If empty, default users **alice** and **bob** are created with their names as passwords. This map is used for user creation when no realm file is found at the **realm-path**. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_USERS**	**Map<String,String>**	
🔒 quarkus.keycloak.devservices.roles A map of roles for Keycloak users. If empty, default roles are assigned: **alice** receives **admin** and **user** roles, while other users receive **user** role. This map is used for role creation when no realm file is found at the **realm-path**. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_ROLES**	**Map<String,List<String>>**	

🔒 quarkus.keycloak.devservices.container-env Environment variables to be passed to the container. Environment variable: **QUARKUS_KEYCLOAK_DEVSERVICES_CONTAINER_ENV**	**Map<String,String>**	
🔒 quarkus.oidc.enabled If the OIDC extension is enabled. Environment variable: **QUARKUS_OIDC_ENABLED**	boolean	**true**
🔒 quarkus.oidc.devui.grant.type Grant type which will be used to acquire a token to test the OIDC 'service' applications Environment variable: **QUARKUS_OIDC_DEVUI_GRANT_TYPE**	tooltip:client['client_credentials' grant], tooltip:password['password' grant], tooltip:code['authorization_code' grant], tooltip:implicit['implicit' grant]	
🔒 quarkus.oidc.devui.web-client-timeout The WebClient timeout. Use this property to configure how long an HTTP client used by Dev UI handlers will wait for a response when requesting tokens from OpenId Connect Provider and sending them to the service endpoint. This timeout is also used by the OIDC dev service admin client. Environment variable: **QUARKUS_OIDC_DEVUI_WEB_CLIENT_TIMEOUT**	Duration ⓘ	**4S**

🔒 **quarkus.oidc.default-token-cache-enabled** Enable the registration of the Default TokenIntrospection and UserInfo Cache implementation bean. Note: This only enables the default implementation. It requires configuration to be activated. See **OidcConfig#tokenCache**. Environment variable: **QUARKUS_OIDC_DEFAULT_TOKEN_CACHE_ENABLED**	boolean	**true**
quarkus.oidc.auth-server-url The base URL of the OpenID Connect (OIDC) server, for example, **https://host:port/auth**. Do not set this property if the public key verification (**public-key**) or certificate chain verification only (**certificate-chain**) is required. The OIDC discovery endpoint is called by default by appending a **.well-known/openid-configuration** path to this URL. For Keycloak, use **https://host:port/realms/{realm}**, replacing **{realm}** with the Keycloak realm name. Environment variable: **QUARKUS_OIDC_AUTH_SERVER_URL**	string	
quarkus.oidc.discovery-enabled Discovery of the OIDC endpoints. If not enabled, you must configure the OIDC endpoint URLs individually. Environment variable: **QUARKUS_OIDC_DISCOVERY_ENABLED**	boolean	**true**
quarkus.oidc.token-path The OIDC token endpoint that issues access and refresh tokens; specified as a relative path or absolute URL. Set if **discovery-enabled** is **false** or a discovered token endpoint path must be customized. Environment variable: **QUARKUS_OIDC_TOKEN_PATH**	string	
quarkus.oidc.revoke-path The relative path or absolute URL of the OIDC token revocation endpoint. Environment variable: **QUARKUS_OIDC_REVOKE_PATH**	string	
quarkus.oidc.client-id The client id of the application. Each application has a client id that is used to identify the application. Setting the client id is not required if **application-type** is **service** and no token introspection is required. Environment variable: **QUARKUS_OIDC_CLIENT_ID**	string	

quarkus.oidc.connection-delay The duration to attempt the initial connection to an OIDC server. For example, setting the duration to **20S** allows 10 retries, each 2 seconds apart. This property is only effective when the initial OIDC connection is created. For dropped connections, use the **connection-retry-count** property instead. Environment variable: **QUARKUS_OIDC_CONNECTION_DELAY**	Duration ?	
quarkus.oidc.connection-retry-count The number of times to retry re-establishing an existing OIDC connection if it is temporarily lost. Different from **connection-delay**, which applies only to initial connection attempts. For instance, if a request to the OIDC token endpoint fails due to a connection issue, it will be retried as per this setting. Environment variable: **QUARKUS_OIDC_CONNECTION_RETRY_COUNT**	int	**3**
quarkus.oidc.connection-timeout The number of seconds after which the current OIDC connection request times out. Environment variable: **QUARKUS_OIDC_CONNECTION_TIMEOUT**	Duration ?	**10S**
quarkus.oidc.use-blocking-dns-lookup Whether DNS lookup should be performed on the worker thread. Use this option when you can see logged warnings about blocked Vert.x event loop by HTTP requests to OIDC server. Environment variable: **QUARKUS_OIDC_USE_BLOCKING_DNS_LOOKUP**	boolean	**false**
quarkus.oidc.max-pool-size The maximum size of the connection pool used by the WebClient. Environment variable: **QUARKUS_OIDC_MAX_POOL_SIZE**	int	
quarkus.oidc.credentials.secret The client secret used by the **client_secret_basic** authentication method. Must be set unless a secret is set in **client-secret** or **jwt** client authentication is required. You can use **client-secret.value** instead, but both properties are mutually exclusive. Environment variable: **QUARKUS_OIDC_CREDENTIALS_SECRET**	string	
quarkus.oidc.credentials.client-secret.value The client secret value. This value is ignored if **credentials.secret** is set. Must be set unless a secret is set in **client-secret** or **jwt** client authentication is required. Environment variable: **QUARKUS_OIDC_CREDENTIALS_CLIENT_SECRET_VALUE**	string	

quarkus.oidc.credentials.client-secret.provider.name The CredentialsProvider name, which should only be set if more than one CredentialsProvider is registered Environment variable: **QUARKUS_OIDC_CREDENTIALS_CLIENT_SECRET_PROVIDER_NAME**	string	
quarkus.oidc.credentials.client-secret.provider.key The CredentialsProvider client secret key Environment variable: **QUARKUS_OIDC_CREDENTIALS_CLIENT_SECRET_PROVIDER_KEY**	string	
quarkus.oidc.credentials.client-secret.method The authentication method. If the **clientSecret.value** secret is set, this method is **basic** by default. Environment variable: **QUARKUS_OIDC_CREDENTIALS_CLIENT_SECRET_METHOD**	tooltip: basic[**client_secret_basic** (default): The client id and secret are submitted with the HTTP Authorization Basic scheme.], tooltip: post[**client_secret_post**: The client id and secret are submitted as the **client_id** and **client_secret** form parame	

	ters.], tooltip: post-jwt[**client_secret_jwt**: The client id and generated JWT secret are submitted as the **client_id** and **client_secret** form parameters.], tooltip: query[client id and secret are submitted as HTTP query parameters. This option is only supported for the OIDC extension.]	
quarkus.oidc.credentials.jwt.secret If provided, indicates that JWT is signed using a secret key. Environment variable: **QUARKUS_OIDC_CREDENTIALS_JWT_SECRET**	string	

quarkus.oidc.credentials.jwt.secret-provider.name The CredentialsProvider name, which should only be set if more than one CredentialsProvider is registered Environment variable: **QUARKUS_OIDC_CREDENTIALS_JWT_SECRET_PROVIDER_NAME**	string	
quarkus.oidc.credentials.jwt.secret-provider.key The CredentialsProvider client secret key Environment variable: **QUARKUS_OIDC_CREDENTIALS_JWT_SECRET_PROVIDER_KEY**	string	
quarkus.oidc.credentials.jwt.key-file If provided, indicates that JWT is signed using a private key in PEM or JWK format. You can use the **signature-algorithm** property to override the default key algorithm, **RS256**. Environment variable: **QUARKUS_OIDC_CREDENTIALS_JWT_KEY_FILE**	string	
quarkus.oidc.credentials.jwt.key-store-file If provided, indicates that JWT is signed using a private key from a keystore. Environment variable: **QUARKUS_OIDC_CREDENTIALS_JWT_KEY_STORE_FILE**	string	
quarkus.oidc.credentials.jwt.key-store-password A parameter to specify the password of the keystore file. Environment variable: **QUARKUS_OIDC_CREDENTIALS_JWT_KEY_STORE_PASSWORD**	string	
quarkus.oidc.credentials.jwt.key-id The private key id or alias. Environment variable: **QUARKUS_OIDC_CREDENTIALS_JWT_KEY_ID**	string	
quarkus.oidc.credentials.jwt.key-password The private key password. Environment variable: **QUARKUS_OIDC_CREDENTIALS_JWT_KEY_PASSWORD**	string	

quarkus.oidc.credentials.jwt.audience The JWT audience (**aud**) claim value. By default, the audience is set to the address of the OpenId Connect Provider's token endpoint. Environment variable: **QUARKUS_OIDC_CREDENTIALS_JWT_AUDIENCE**	string	
quarkus.oidc.credentials.jwt.token-key-id The key identifier of the signing key added as a JWT **kid** header. Environment variable: **QUARKUS_OIDC_CREDENTIALS_JWT_TOKEN_KEY_ID**	string	
quarkus.oidc.credentials.jwt.issuer The issuer of the signing key added as a JWT **iss** claim. The default value is the client id. Environment variable: **QUARKUS_OIDC_CREDENTIALS_JWT_ISSUER**	string	
quarkus.oidc.credentials.jwt.subject Subject of the signing key added as a JWT **sub** claim The default value is the client id. Environment variable: **QUARKUS_OIDC_CREDENTIALS_JWT_SUBJECT**	string	
quarkus.oidc.credentials.jwt.signature-algorithm The signature algorithm used for the **key-file** property. Supported values: **RS256** (default), **RS384**, **RS512**, **PS256**, **PS384**, **PS512**, **ES256**, **ES384**, **ES512**, **HS256**, **HS384**, **HS512**. Environment variable: **QUARKUS_OIDC_CREDENTIALS_JWT_SIGNATURE_ALGORITHM**	string	
quarkus.oidc.credentials.jwt.lifespan The JWT lifespan in seconds. This value is added to the time at which the JWT was issued to calculate the expiration time. Environment variable: **QUARKUS_OIDC_CREDENTIALS_JWT_LIFESPAN**	int	**10**
quarkus.oidc.proxy.host The host name or IP address of the Proxy. Note: If the OIDC adapter requires a Proxy to talk with the OIDC server (Provider), set this value to enable the usage of a Proxy. Environment variable: **QUARKUS_OIDC_PROXY_HOST**	string	

quarkus.oidc.proxy.port The port number of the Proxy. The default value is **80**. Environment variable: **QUARKUS_OIDC_PROXY_PORT**	int	**80**
quarkus.oidc.proxy.username The username, if the Proxy needs authentication. Environment variable: **QUARKUS_OIDC_PROXY_USERNAME**	string	
quarkus.oidc.proxy.password The password, if the Proxy needs authentication. Environment variable: **QUARKUS_OIDC_PROXY_PASSWORD**	string	

quarkus.oidc.tls.verification Certificate validation and hostname verification, which can be one of the following **Verification** values. Default is **required**. Environment variable: **QUARKUS_OIDC_TLS_VERIFICATION**	tooltip: required[Certificates are validated and hostname verification is enabled. This is the default value.], tooltip: certificate-validation[Certificates are validated but hostname verification is disabled.], tooltip: none[All certificates are trusted and hostname verification is disabled.]	
quarkus.oidc.tls.key-store-file An optional keystore that holds the certificate information instead of specifying separate files. Environment variable: **QUARKUS_OIDC_TLS_KEY_STORE_FILE**	path	

quarkus.oidc.tls.key-store-file-type The type of the keystore file. If not given, the type is automatically detected based on the file name. Environment variable: **QUARKUS_OIDC_TLS_KEY_STORE_FILE_TYPE**	string	
quarkus.oidc.tls.key-store-provider The provider of the keystore file. If not given, the provider is automatically detected based on the keystore file type. Environment variable: **QUARKUS_OIDC_TLS_KEY_STORE_PROVIDER**	string	
quarkus.oidc.tls.key-store-password The password of the keystore file. If not given, the default value, **password**, is used. Environment variable: **QUARKUS_OIDC_TLS_KEY_STORE_PASSWORD**	string	
quarkus.oidc.tls.key-store-key-alias The alias of a specific key in the keystore. When SNI is disabled, if the keystore contains multiple keys and no alias is specified, the behavior is undefined. Environment variable: **QUARKUS_OIDC_TLS_KEY_STORE_KEY_ALIAS**	string	
quarkus.oidc.tls.key-store-key-password The password of the key, if it is different from the **key-store-password**. Environment variable: **QUARKUS_OIDC_TLS_KEY_STORE_KEY_PASSWORD**	string	
quarkus.oidc.tls.trust-store-file The truststore that holds the certificate information of the certificates to trust. Environment variable: **QUARKUS_OIDC_TLS_TRUST_STORE_FILE**	path	
quarkus.oidc.tls.trust-store-password The password of the truststore file. Environment variable: **QUARKUS_OIDC_TLS_TRUST_STORE_PASSWORD**	string	
quarkus.oidc.tls.trust-store-cert-alias The alias of the truststore certificate. Environment variable: **QUARKUS_OIDC_TLS_TRUST_STORE_CERT_ALIAS**	string	

quarkus.oidc.tls.trust-store-file-type The type of the truststore file. If not given, the type is automatically detected based on the file name. Environment variable: **QUARKUS_OIDC_TLS_TRUST_STORE_FILE_TYPE**	string	
quarkus.oidc.tls.trust-store-provider The provider of the truststore file. If not given, the provider is automatically detected based on the truststore file type. Environment variable: **QUARKUS_OIDC_TLS_TRUST_STORE_PROVIDER**	string	
quarkus.oidc.tenant-id A unique tenant identifier. It can be set by **TenantConfigResolver** providers, which resolve the tenant configuration dynamically. Environment variable: **QUARKUS_OIDC_TENANT_ID**	string	
quarkus.oidc.tenant-enabled If this tenant configuration is enabled. The default tenant is disabled if it is not configured but a **TenantConfigResolver** that resolves tenant configurations is registered, or named tenants are configured. In this case, you do not need to disable the default tenant. Environment variable: **QUARKUS_OIDC_TENANT_ENABLED**	boolean	**true**
quarkus.oidc.application-type The application type, which can be one of the following **ApplicationType** values. Environment variable: **QUARKUS_OIDC_APPLICATION_TYPE**	tooltip: web-app[A **WEB_APP** is a client that serves pages, usually a front-end application. For this type of client the Authorization Code Flow is defined as the preferr	**service**

	ed method for authenticating users.], tooltip: service [A **SERVICE** is a client that has a set of protected HTTP resources, usually a backend application following the RESTful Architectural Design. For this type of client, the Bearer Authorization method is defined as the preferred method for authenticating and authorizing users.], tooltip: hybrid[	

	A combined **SERVICE** and **WEB_APP** client. For this type of client, the Bearer Authorization method is used if the Authorization header is set and Authorization Code Flow - if not.]	
quarkus.oidc.authorization-path The relative path or absolute URL of the OpenID Connect (OIDC) authorization endpoint, which authenticates users. You must set this property for **web-app** applications if OIDC discovery is disabled. This property is ignored if OIDC discovery is enabled. Environment variable: **QUARKUS_OIDC_AUTHORIZATION_PATH**	string	
quarkus.oidc.user-info-path The relative path or absolute URL of the OIDC UserInfo endpoint. You must set this property for **web-app** applications if OIDC discovery is disabled and the **authentication.user-info-required** property is enabled. This property is ignored if OIDC discovery is enabled. Environment variable: **QUARKUS_OIDC_USER_INFO_PATH**	string	

quarkus.oidc.introspection-path Relative path or absolute URL of the OIDC RFC7662 introspection endpoint which can introspect both opaque and JSON Web Token (JWT) tokens. This property must be set if OIDC discovery is disabled and 1) the opaque bearer access tokens must be verified or 2) JWT tokens must be verified while the cached JWK verification set with no matching JWK is being refreshed. This property is ignored if the discovery is enabled. Environment variable: **QUARKUS_OIDC_INTROSPECTION_PATH**	string	
quarkus.oidc.jwks-path Relative path or absolute URL of the OIDC JSON Web Key Set (JWKS) endpoint which returns a JSON Web Key Verification Set. This property should be set if OIDC discovery is disabled and the local JWT verification is required. This property is ignored if the discovery is enabled. Environment variable: **QUARKUS_OIDC_JWKS_PATH**	string	
quarkus.oidc.end-session-path Relative path or absolute URL of the OIDC end_session_endpoint. This property must be set if OIDC discovery is disabled and RP Initiated Logout support for the **web-app** applications is required. This property is ignored if the discovery is enabled. Environment variable: **QUARKUS_OIDC_END_SESSION_PATH**	string	
quarkus.oidc.public-key The public key for the local JWT token verification. OIDC server connection is not created when this property is set. Environment variable: **QUARKUS_OIDC_PUBLIC_KEY**	string	
quarkus.oidc.introspection-credentials.name Name Environment variable: **QUARKUS_OIDC_INTROSPECTION_CREDENTIALS_NAME**	string	
quarkus.oidc.introspection-credentials.secret Secret Environment variable: **QUARKUS_OIDC_INTROSPECTION_CREDENTIALS_SECRET**	string	
quarkus.oidc.introspection-credentials.include-client-id Include OpenId Connect Client ID configured with **quarkus.oidc.client-id**. Environment variable: **QUARKUS_OIDC_INTROSPECTION_CREDENTIALS_INCLUDE_CLIENT_ID**	boolean	**true**

quarkus.oidc.roles.role-claim-path A list of paths to claims containing an array of groups. Each path starts from the top level JWT JSON object and can contain multiple segments. Each segment represents a JSON object name only; for example: "realm/groups". Use double quotes with the namespace-qualified claim names. This property can be used if a token has no **groups** claim but has the groups set in one or more different claims. Environment variable: **QUARKUS_OIDC_ROLES_ROLE_CLAIM_PATH**	list of string	
quarkus.oidc.roles.role-claim-separator The separator for splitting strings that contain multiple group values. It is only used if the "role-claim-path" property points to one or more custom claims whose values are strings. A single space is used by default because the standard **scope** claim can contain a space-separated sequence. Environment variable: **QUARKUS_OIDC_ROLES_ROLE_CLAIM_SEPARATOR**	string	

quarkus.oidc.roles.source Source of the principal roles. Environment variable: **QUARKUS_OIDC_ROLES_SOURCE**	tooltip:idtoken[ID Token - the default value for the **web-app** applications.], tooltip:accesstoken[Access Token - the default value for the **service** applications; can also be used as the source of roles for the **web-app** applications.], tooltip:userinfo[User Info]	
quarkus.oidc.token.issuer The expected issuer **iss** claim value. This property overrides the**issuer** property, which might be set in OpenId Connect provider’s well-known configuration. If the **iss** claim value varies depending on the host, IP address, or tenant id of the provider, you can skip the issuer verification by setting this property to **any**, but it should be done only when other options (such as configuring the provider to use the fixed **iss** claim value) are not possible. Environment variable: **QUARKUS_OIDC_TOKEN_ISSUER**	string	

quarkus.oidc.token.audience The expected audience **aud** claim value, which can be a string or an array of strings. Note the audience claim is verified for ID tokens by default. ID token audience must be equal to the value of **quarkus.oidc.client-id** property. Use this property to override the expected value if your OpenID Connect provider sets a different audience claim value in ID tokens. Set it to **any** if your provider does not set ID token audience` claim. Audience verification for access tokens is only done if this property is configured. Environment variable: **QUARKUS_OIDC_TOKEN_AUDIENCE**	list of string	
quarkus.oidc.token.subject-required Require that the token includes a **sub** (subject) claim which is a unique and never reassigned identifier for the current user. Note that if you enable this property and if UserInfo is also required, both the token and UserInfo **sub** claims must be present and match each other. Environment variable: **QUARKUS_OIDC_TOKEN_SUBJECT_REQUIRED**	boolean	**false**
quarkus.oidc.token.token-type Expected token type Environment variable: **QUARKUS_OIDC_TOKEN_TOKEN_TYPE**	string	
quarkus.oidc.token.lifespan-grace Life span grace period in seconds. When checking token expiry, current time is allowed to be later than token expiration time by at most the configured number of seconds. When checking token issuance, current time is allowed to be sooner than token issue time by at most the configured number of seconds. Environment variable: **QUARKUS_OIDC_TOKEN_LIFESPAN_GRACE**	int	
quarkus.oidc.token.age Token age. It allows for the number of seconds to be specified that must not elapse since the **iat** (issued at) time. A small leeway to account for clock skew which can be configured with **quarkus.oidc.token.lifespan-grace** to verify the token expiry time can also be used to verify the token age property. Note that setting this property does not relax the requirement that Bearer and Code Flow JWT tokens must have a valid (**exp**) expiry claim value. The only exception where setting this property relaxes the requirement is when a logout token is sent with a back-channel logout request since the current OpenId Connect Back-Channel specification does not explicitly require the logout tokens to contain an **exp** claim. However, even if the current logout token is allowed to have no **exp** claim, the **exp** claim is still verified if the logout token contains it. Environment variable: **QUARKUS_OIDC_TOKEN_AGE**	Duration ⓘ	

quarkus.oidc.token.principal-claim Name of the claim which contains a principal name. By default, the **upn**, **preferred_username** and **sub** claims are checked. Environment variable: **QUARKUS_OIDC_TOKEN_PRINCIPAL_CLAIM**	string	
quarkus.oidc.token.refresh-expired Refresh expired authorization code flow ID or access tokens. If this property is enabled, a refresh token request is performed if the authorization code ID or access token has expired and, if successful, the local session is updated with the new set of tokens. Otherwise, the local session is invalidated and the user redirected to the OpenID Provider to re-authenticate. In this case, the user might not be challenged again if the OIDC provider session is still active. For this option be effective the **authentication.session-age-extension** property should also be set to a nonzero value since the refresh token is currently kept in the user session. This option is valid only when the application is of type **ApplicationType#WEB_APP**}. This property is enabled if **quarkus.oidc.token.refresh-token-time-skew** is configured, you do not need to enable this property manually in this case. Environment variable: **QUARKUS_OIDC_TOKEN_REFRESH_EXPIRED**	boolean	**false**
quarkus.oidc.token.refresh-token-time-skew The refresh token time skew, in seconds. If this property is enabled, the configured number of seconds is added to the current time when checking if the authorization code ID or access token should be refreshed. If the sum is greater than the authorization code ID or access token's expiration time, a refresh is going to happen. Environment variable: **QUARKUS_OIDC_TOKEN_REFRESH_TOKEN_TIME_SKEW**	Duration ?	
quarkus.oidc.token.forced-jwk-refresh-interval The forced JWK set refresh interval in minutes. Environment variable: **QUARKUS_OIDC_TOKEN_FORCED_JWK_REFRESH_INTERVAL**	Duration ?	**10M**
quarkus.oidc.token.header Custom HTTP header that contains a bearer token. This option is valid only when the application is of type **ApplicationType#SERVICE**}. Environment variable: **QUARKUS_OIDC_TOKEN_HEADER**	string	
quarkus.oidc.token.authorization-scheme HTTP Authorization header scheme. Environment variable: **QUARKUS_OIDC_TOKEN_AUTHORIZATION_SCHEME**	string	**Bearer**

quarkus.oidc.token.signature-algorithm Required signature algorithm. OIDC providers support many signature algorithms but if necessary you can restrict Quarkus application to accept tokens signed only using an algorithm configured with this property. Environment variable: **QUARKUS_OIDC_TOKEN_SIGNATURE_ALGORITHM**	**rs256**, **rs384**, **rs512**, **ps256**, **ps384**, **ps512**, **es256**, **es384**, **es512**, **eddsa**	
quarkus.oidc.token.decryption-key-location Decryption key location. JWT tokens can be inner-signed and encrypted by OpenId Connect providers. However, it is not always possible to remotely introspect such tokens because the providers might not control the private decryption keys. In such cases set this property to point to the file containing the decryption private key in PEM or JSON Web Key (JWK) format. If this property is not set and the **private_key_jwt** client authentication method is used, the private key used to sign the client authentication JWT tokens are also used to decrypt the encrypted ID tokens. Environment variable: **QUARKUS_OIDC_TOKEN_DECRYPTION_KEY_LOCATION**	string	
quarkus.oidc.token.allow-jwt-introspection Allow the remote introspection of JWT tokens when no matching JWK key is available. This property is set to **true** by default for backward-compatibility reasons. It is planned that this default value will be changed to **false** in an upcoming release. Also note this property is ignored if JWK endpoint URI is not available and introspecting the tokens is the only verification option. Environment variable: **QUARKUS_OIDC_TOKEN_ALLOW_JWT_INTROSPECTION**	boolean	**true**
quarkus.oidc.token.require-jwt-introspection-only Require that JWT tokens are only introspected remotely. Environment variable: **QUARKUS_OIDC_TOKEN_REQUIRE_JWT_INTROSPECTION_ONLY**	boolean	**false**
quarkus.oidc.token.allow-opaque-token-introspection Allow the remote introspection of the opaque tokens. Set this property to **false** if only JWT tokens are expected. Environment variable: **QUARKUS_OIDC_TOKEN_ALLOW_OPAQUE_TOKEN_INTROSPECTION**	boolean	**true**

quarkus.oidc.token.customizer-name Token customizer name. Allows to select a tenant specific token customizer as a named bean. Prefer using **TenantFeature** qualifier when registering custom **TokenCustomizer**. Use this property only to refer to **TokenCustomizer** implementations provided by this extension. Environment variable: **QUARKUS_OIDC_TOKEN_CUSTOMIZER_NAME**	string	
quarkus.oidc.token.verify-access-token-with-user-info Indirectly verify that the opaque (binary) access token is valid by using it to request UserInfo. Opaque access token is considered valid if the provider accepted this token and returned a valid UserInfo. You should only enable this option if the opaque access tokens must be accepted but OpenId Connect provider does not have a token introspection endpoint. This property has no effect when JWT tokens must be verified. Environment variable: **QUARKUS_OIDC_TOKEN_VERIFY_ACCESS_TOKEN_WITH_USER_INFO**	boolean	**false**
quarkus.oidc.logout.path The relative path of the logout endpoint at the application. If provided, the application is able to initiate the logout through this endpoint in conformance with the OpenID Connect RP-Initiated Logout specification. Environment variable: **QUARKUS_OIDC_LOGOUT_PATH**	string	
quarkus.oidc.logout.post-logout-path Relative path of the application endpoint where the user should be redirected to after logging out from the OpenID Connect Provider. This endpoint URI must be properly registered at the OpenID Connect Provider as a valid redirect URI. Environment variable: **QUARKUS_OIDC_LOGOUT_POST_LOGOUT_PATH**	string	
quarkus.oidc.logout.post-logout-uri-param Name of the post logout URI parameter which is added as a query parameter to the logout redirect URI. Environment variable: **QUARKUS_OIDC_LOGOUT_POST_LOGOUT_URI_PARAM**	string	**post_logout_redirect_uri**
quarkus.oidc.logout.backchannel.path The relative path of the Back-Channel Logout endpoint at the application. Environment variable: **QUARKUS_OIDC_LOGOUT_BACKCHANNEL_PATH**	string	

quarkus.oidc.logout.backchannel.token-cache-size Maximum number of logout tokens that can be cached before they are matched against ID tokens stored in session cookies. Environment variable: **QUARKUS_OIDC_LOGOUT_BACKCHANNEL_TOKEN_CACHE_SIZE**	int	**10**
quarkus.oidc.logout.backchannel.token-cache-time-to-live Number of minutes a logout token can be cached for. Environment variable: **QUARKUS_OIDC_LOGOUT_BACKCHANNEL_TOKEN_CACHE_TIME_TO_LIVE**	Duration ?	**10M**
quarkus.oidc.logout.backchannel.clean-up-timer-interval Token cache timer interval. If this property is set, a timer checks and removes the stale entries periodically. Environment variable: **QUARKUS_OIDC_LOGOUT_BACKCHANNEL_CLEAN_UP_TIMER_INTERVAL**	Duration ?	
quarkus.oidc.logout.backchannel.logout-token-key Logout token claim whose value is used as a key for caching the tokens. Only **sub** (subject) and **sid** (session id) claims can be used as keys. Set it to**sid** only if ID tokens issued by the OIDC provider have no **sub** but have**sid** claim. Environment variable: **QUARKUS_OIDC_LOGOUT_BACKCHANNEL_LOGOUT_TOKEN_KEY**	string	**sub**
quarkus.oidc.logout.frontchannel.path The relative path of the Front-Channel Logout endpoint at the application. Environment variable: **QUARKUS_OIDC_LOGOUT_FRONTCHANNEL_PATH**	string	
quarkus.oidc.certificate-chain.leaf-certificate-name Common name of the leaf certificate. It must be set if the **trust-store-file** does not have this certificate imported. Environment variable: **QUARKUS_OIDC_CERTIFICATE_CHAIN_LEAF_CERTIFICATE_NAME**	string	
quarkus.oidc.certificate-chain.trust-store-file Truststore file which keeps thumbprints of the trusted certificates. Environment variable: **QUARKUS_OIDC_CERTIFICATE_CHAIN_TRUST_STORE_FILE**	path	

quarkus.oidc.certificate-chain.trust-store-password A parameter to specify the password of the truststore file if it is configured with **trust-store-file**. Environment variable: **QUARKUS_OIDC_CERTIFICATE_CHAIN_TRUST_STORE_PASSWORD**	string	
quarkus.oidc.certificate-chain.trust-store-cert-alias A parameter to specify the alias of the truststore certificate. Environment variable: **QUARKUS_OIDC_CERTIFICATE_CHAIN_TRUST_STORE_CERT_ALIAS**	string	
quarkus.oidc.certificate-chain.trust-store-file-type An optional parameter to specify type of the truststore file. If not given, the type is automatically detected based on the file name. Environment variable: **QUARKUS_OIDC_CERTIFICATE_CHAIN_TRUST_STORE_FILE_TYPE**	string	
quarkus.oidc.authentication.response-mode Authorization code flow response mode. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_RESPONSE_MODE**	tooltip: query[Authorization response parameters are encoded in the query string added to the **redirect_uri**], tooltip: form-post[Authorization response parameters are encoded as HTML form values that	**query**

	are auto-submitted in the browser and transmitted by the HTTP POST method using the application/x-www-form-urlencoded content type]	
quarkus.oidc.authentication.redirect-path The relative path for calculating a **redirect_uri** query parameter. It has to start from a forward slash and is appended to the request URI's host and port. For example, if the current request URI is **https://localhost:8080/service**, a **redirect_uri** parameter is set to **https://localhost:8080/** if this property is set to / and be the same as the request URI if this property has not been configured. Note the original request URI is restored after the user has authenticated if **restorePathAfterRedirect** is set to **true**. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_REDIRECT_PATH**	string	
quarkus.oidc.authentication.restore-path-after-redirect If this property is set to **true**, the original request URI which was used before the authentication is restored after the user has been redirected back to the application. Note if **redirectPath** property is not set, the original request URI is restored even if this property is disabled. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_RESTORE_PATH_AFTER_REDIRECT**	boolean	**false**

quarkus.oidc.authentication.remove-redirect-parameters Remove the query parameters such as **code** and **state** set by the OIDC server on the redirect URI after the user has authenticated by redirecting a user to the same URI but without the query parameters. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_REMOVE_REDIRECT_PARAMETERS**	boolean	**true**
quarkus.oidc.authentication.error-path Relative path to the public endpoint which processes the error response from the OIDC authorization endpoint. If the user authentication has failed, the OIDC provider returns an **error** and an optional **error_description** parameters, instead of the expected authorization **code**. If this property is set, the user is redirected to the endpoint which can return a user-friendly error description page. It has to start from a forward slash and is appended to the request URI's host and port. For example, if it is set as **/error** and the current request URI is **https://localhost:8080/callback?error=invalid_scope**, a redirect is made to **https://localhost:8080/error?error=invalid_scope**. If this property is not set, HTTP 401 status is returned in case of the user authentication failure. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_ERROR_PATH**	string	
quarkus.oidc.authentication.verify-access-token Both ID and access tokens are fetched from the OIDC provider as part of the authorization code flow. ID token is always verified on every user request as the primary token which is used to represent the principal and extract the roles. Access token is not verified by default since it is meant to be propagated to the downstream services. The verification of the access token should be enabled if it is injected as a JWT token. Access tokens obtained as part of the code flow are always verified if **quarkus.oidc.roles.source** property is set to **accesstoken** which means the authorization decision is based on the roles extracted from the access token. Bearer access tokens are always verified. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_VERIFY_ACCESS_TOKEN**	boolean	**false**
quarkus.oidc.authentication.force-redirect-https-scheme Force **https** as the **redirect_uri** parameter scheme when running behind an SSL/TLS terminating reverse proxy. This property, if enabled, also affects the logout **post_logout_redirect_uri** and the local redirect requests. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_FORCE_REDIRECT_HTTPS_SCHEME**	boolean	**false**

quarkus.oidc.authentication.scopes List of scopes Environment variable: **QUARKUS_OIDC_AUTHENTICATION_SCOPES**	list of string	
quarkus.oidc.authentication.nonce-required Require that ID token includes a **nonce** claim which must match**nonce** authentication request query parameter. Enabling this property can help mitigate replay attacks. Do not enable this property if your OpenId Connect provider does not support setting **nonce** in ID token or if you work with OAuth2 provider such as **GitHub** which does not issue ID tokens. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_NONCE_REQUIRED**	boolean	**false**
quarkus.oidc.authentication.add-openid-scope Add the **openid** scope automatically to the list of scopes. This is required for OpenId Connect providers, but does not work for OAuth2 providers such as Twitter OAuth2, which do not accept this scope and throw errors. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_ADD_OPENID_SCOPE**	boolean	**true**
quarkus.oidc.authentication.forward-params Request URL query parameters which, if present, are added to the authentication redirect URI. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_FORWARD_PARAMS**	list of string	
quarkus.oidc.authentication.cookie-force-secure If enabled the state, session, and post logout cookies have their **secure** parameter set to **true** when HTTP is used. It might be necessary when running behind an SSL/TLS terminating reverse proxy. The cookies are always secure if HTTPS is used, even if this property is set to false. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_COOKIE_FORCE_SECURE**	boolean	**false**
quarkus.oidc.authentication.cookie-suffix Cookie name suffix. For example, a session cookie name for the default OIDC tenant is **q_session** but can be changed to**q_session_test** if this property is set to**test**. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_COOKIE_SUFFIX**	string	

quarkus.oidc.authentication.cookie-path Cookie path parameter value which, if set, is used to set a path parameter for the session, state and post logout cookies. The **cookie-path-header** property, if set, is checked first. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_COOKIE_PATH**	string	/
quarkus.oidc.authentication.cookie-path-header Cookie path header parameter value which, if set, identifies the incoming HTTP header whose value is used to set a path parameter for the session, state and post logout cookies. If the header is missing, the **cookie-path** property is checked. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_COOKIE_PATH_HEADER**	string	
quarkus.oidc.authentication.cookie-domain Cookie domain parameter value which, if set, is used for the session, state and post logout cookies. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_COOKIE_DOMAIN**	string	
quarkus.oidc.authentication.cookie-same-site SameSite attribute for the session cookie. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_COOKIE_SAME_SITE**	**strict**, **lax**, **none**	**lax**
quarkus.oidc.authentication.allow-multiple-code-flows If a state cookie is present, a **state** query parameter must also be present and both the state cookie name suffix and state cookie value must match the value of the **state** query parameter when the redirect path matches the current path. However, if multiple authentications are attempted from the same browser, for example, from the different browser tabs, then the currently available state cookie might represent the authentication flow initiated from another tab and not related to the current request. Disable this property to permit only a single authorization code flow in the same browser. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_ALLOW_MULTIPLE_CODE_FLOWS**	boolean	**true**

quarkus.oidc.authentication.fail-on-missing-state-param Fail with the HTTP 401 error if the state cookie is present but no state query parameter is present. When either multiple authentications are disabled or the redirect URL matches the original request URL, the stale state cookie might remain in the browser cache from the earlier failed redirect to an OpenId Connect provider and be visible during the current request. For example, if Single-page application (SPA) uses XHR to handle redirects to the provider which does not support CORS for its authorization endpoint, the browser blocks it and the state cookie created by Quarkus remains in the browser cache. Quarkus reports an authentication failure when it detects such an old state cookie but find no matching state query parameter. Reporting HTTP 401 error is usually the right thing to do in such cases, it minimizes a risk of the browser redirect loop but also can identify problems in the way SPA or Quarkus application manage redirects. For example, enabling **java-script-auto-redirect** or having the provider redirect to URL configured with **redirect-path** might be needed to avoid such errors. However, setting this property to **false** might help if the above options are not suitable. It causes a new authentication redirect to OpenId Connect provider. Doing so might increase the risk of browser redirect loops. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_FAIL_ON_MISSING_STATE_PARAM**	boolean	**false**
quarkus.oidc.authentication.user-info-required If this property is set to **true**, an OIDC UserInfo endpoint is called. This property is enabled if **quarkus.oidc.roles.source** is **userinfo**. or **quarkus.oidc.token.verify-access-token-with-user-info** is **true** or **quarkus.oidc.authentication.id-token-required** is set to **false**, you do not need to enable this property manually in these cases. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_USER_INFO_REQUIRED**	boolean	**false**
quarkus.oidc.authentication.session-age-extension Session age extension in minutes. The user session age property is set to the value of the ID token life-span by default and the user is redirected to the OIDC provider to re-authenticate once the session has expired. If this property is set to a nonzero value, then the expired ID token can be refreshed before the session has expired. This property is ignored if the **token.refresh-expired** property has not been enabled. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_SESSION_AGE_EXTENSION**	Duration ?	**5M**

quarkus.oidc.authentication.java-script-auto-redirect If this property is set to **true**, a normal 302 redirect response is returned if the request was initiated by a JavaScript API such as XMLHttpRequest or Fetch and the current user needs to be (re)authenticated, which might not be desirable for Single-page applications (SPA) since it automatically following the redirect might not work given that OIDC authorization endpoints typically do not support CORS. If this property is set to **false**, a status code of**499** is returned to allow SPA to handle the redirect manually if a request header identifying current request as a JavaScript request is found. **X-Requested-With** request header with its value set to either **JavaScript** or **XMLHttpRequest** is expected by default if this property is enabled. You can register a custom **JavaScriptRequestChecker** to do a custom JavaScript request check instead. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_JAVA_SCRIPT_AUTO_REDIRECT**	boolean	**true**
quarkus.oidc.authentication.id-token-required Requires that ID token is available when the authorization code flow completes. Disable this property only when you need to use the authorization code flow with OAuth2 providers which do not return ID token - an internal IdToken is generated in such cases. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_ID_TOKEN_REQUIRED**	boolean	**true**
quarkus.oidc.authentication.internal-id-token-lifespan Internal ID token lifespan. This property is only checked when an internal IdToken is generated when Oauth2 providers do not return IdToken. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_INTERNAL_ID_TOKEN_LIFESPAN**	Duration ⓘ	**5M**
quarkus.oidc.authentication.pkce-required Requires that a Proof Key for Code Exchange (PKCE) is used. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_PKCE_REQUIRED**	boolean	**false**

quarkus.oidc.authentication.state-secret Secret used to encrypt Proof Key for Code Exchange (PKCE) code verifier and/or nonce in the code flow state. This secret should be at least 32 characters long. If this secret is not set, the client secret configured with either **quarkus.oidc.credentials.secret** or **quarkus.oidc.credentials.client-secret.value** is checked. Finally, **quarkus.oidc.credentials.jwt.secret** which can be used for **client_jwt_secret** authentication is checked. A client secret is not be used as a state encryption secret if it is less than 32 characters long. The secret is auto-generated if it remains uninitialized after checking all of these properties. Error is reported if the secret length is less than 16 characters. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_STATE_SECRET**	string	
quarkus.oidc.token-state-manager.strategy Default TokenStateManager strategy. Environment variable: **QUARKUS_OIDC_TOKEN_STATE_MANAGER_STRATEGY**	tooltip: keep-all-tokens[Keep ID, access and refresh tokens.], tooltip:id-token[Keep ID token only], tooltip:id-refresh-tokens[Keep ID and refresh tokens only]	**keep-all-tokens**
quarkus.oidc.token-state-manager.split-tokens Default TokenStateManager keeps all tokens (ID, access and refresh) returned in the authorization code grant response in a single session cookie by default. Enable this property to minimize a session cookie size Environment variable: **QUARKUS_OIDC_TOKEN_STATE_MANAGER_SPLIT_TOKENS**	boolean	**false**

quarkus.oidc.token-state-manager.encryption-required Mandates that the Default TokenStateManager encrypt the session cookie that stores the tokens. Environment variable: **QUARKUS_OIDC_TOKEN_STATE_MANAGER_ENCRYPTION_REQUIRED**	boolean	**true**
quarkus.oidc.token-state-manager.encryption-secret The secret used by the Default TokenStateManager to encrypt the session cookie storing the tokens when **encryption-required** property is enabled. If this secret is not set, the client secret configured with either **quarkus.oidc.credentials.secret** or **quarkus.oidc.credentials.client-secret.value** is checked. Finally, **quarkus.oidc.credentials.jwt.secret** which can be used for **client_jwt_secret** authentication is checked. The secret is auto-generated if it remains uninitialized after checking all of these properties. The length of the secret used to encrypt the tokens should be at least 32 characters long. A warning is logged if the secret length is less than 16 characters. Environment variable: **QUARKUS_OIDC_TOKEN_STATE_MANAGER_ENCRYPTION_SECRET**	string	
quarkus.oidc.allow-token-introspection-cache Allow caching the token introspection data. Note enabling this property does not enable the cache itself but only permits to cache the token introspection for a given tenant. If the default token cache can be used, see **OidcConfig.TokenCache** to enable it. Environment variable: **QUARKUS_OIDC_ALLOW_TOKEN_INTROSPECTION_CACHE**	boolean	**true**
quarkus.oidc.allow-user-info-cache Allow caching the user info data. Note enabling this property does not enable the cache itself but only permits to cache the user info data for a given tenant. If the default token cache can be used, see **OidcConfig.TokenCache** to enable it. Environment variable: **QUARKUS_OIDC_ALLOW_USER_INFO_CACHE**	boolean	**true**
quarkus.oidc.cache-user-info-in-idtoken Allow inlining UserInfo in IdToken instead of caching it in the token cache. This property is only checked when an internal IdToken is generated when Oauth2 providers do not return IdToken. Inlining UserInfo in the generated IdToken allows to store it in the session cookie and avoids introducing a cached state. Environment variable: **QUARKUS_OIDC_CACHE_USER_INFO_IN_IDTOKEN**	boolean	**false**

quarkus.oidc.jwks.resolve-early If JWK verification keys should be fetched at the moment a connection to the OIDC provider is initialized. Disabling this property delays the key acquisition until the moment the current token has to be verified. Typically it can only be necessary if the token or other telated request properties provide an additional context which is required to resolve the keys correctly. Environment variable: **QUARKUS_OIDC_JWKS_RESOLVE_EARLY**	boolean	**true**
quarkus.oidc.jwks.cache-size Maximum number of JWK keys that can be cached. This property is ignored if the **resolve-early** property is set to true. Environment variable: **QUARKUS_OIDC_JWKS_CACHE_SIZE**	int	**10**
quarkus.oidc.jwks.cache-time-to-live Number of minutes a JWK key can be cached for. This property is ignored if the **resolve-early** property is set to true. Environment variable: **QUARKUS_OIDC_JWKS_CACHE_TIME_TO_LIVE**	Duration ?	**10M**
quarkus.oidc.jwks.clean-up-timer-interval Cache timer interval. If this property is set, a timer checks and removes the stale entries periodically. This property is ignored if the **resolve-early** property is set to true. Environment variable: **QUARKUS_OIDC_JWKS_CLEAN_UP_TIMER_INTERVAL**	Duration ?	
quarkus.oidc.provider Well known OpenId Connect provider identifier Environment variable: **QUARKUS_OIDC_PROVIDER**	**apple**, **discord**, **facebook**, **github**, **google**, **linkedin**, **mastodon**, **microsoft**, **spotify**, **strava**, **twitch**, **twitter**, **x**	

quarkus.oidc.token-cache.max-size Maximum number of cache entries. Set it to a positive value if the cache has to be enabled. Environment variable: **QUARKUS_OIDC_TOKEN_CACHE_MAX_SIZE**	int	**0**
quarkus.oidc.token-cache.time-to-live Maximum amount of time a given cache entry is valid for. Environment variable: **QUARKUS_OIDC_TOKEN_CACHE_TIME_TO_LIVE**	Duration ?	**3M**
quarkus.oidc.token-cache.clean-up-timer-interval Clean up timer interval. If this property is set then a timer will check and remove the stale entries periodically. Environment variable: **QUARKUS_OIDC_TOKEN_CACHE_CLEAN_UP_TIMER_INTERVAL**	Duration ?	
🔒 **quarkus.oidc.devui.grant-options** Grant options Environment variable: **QUARKUS_OIDC_DEVUI_GRANT_OPTIONS**	**Map<String,Map<String,String>>**	
quarkus.oidc.credentials.jwt.claims Additional claims. Environment variable: **QUARKUS_OIDC_CREDENTIALS_JWT_CLAIMS**	**Map<String,String>**	
quarkus.oidc.token.required-claims A map of required claims and their expected values. For example, **quarkus.oidc.token.required-claims.org_id = org_xyz** would require tokens to have the **org_id** claim to be present and set to**org_xyz**. Strings are the only supported types. Use **SecurityIdentityAugmentor** to verify claims of other types or complex claims. Environment variable: **QUARKUS_OIDC_TOKEN_REQUIRED_CLAIMS**	**Map<String,String>**	
quarkus.oidc.logout.extra-params Additional properties which is added as the query parameters to the logout redirect URI. Environment variable: **QUARKUS_OIDC_LOGOUT_EXTRA_PARAMS**	**Map<String,String>**	

quarkus.oidc.authentication.extra-params Additional properties added as query parameters to the authentication redirect URI. Environment variable: **QUARKUS_OIDC_AUTHENTICATION_EXTRA_PARAMS**	**Map<String,String>**	
quarkus.oidc.code-grant.extra-params Additional parameters, in addition to the required **code** and **redirect-uri** parameters, which must be included to complete the authorization code grant request. Environment variable: **QUARKUS_OIDC_CODE_GRANT_EXTRA_PARAMS**	**Map<String,String>**	
quarkus.oidc.code-grant.headers Custom HTTP headers which must be sent to complete the authorization code grant request. Environment variable: **QUARKUS_OIDC_CODE_GRANT_HEADERS**	**Map<String,String>**	
Additional named tenants	Type	Default
quarkus.oidc."tenant".auth-server-url The base URL of the OpenID Connect (OIDC) server, for example, **https://host:port/auth**. Do not set this property if the public key verification (**public-key**) or certificate chain verification only (**certificate-chain**) is required. The OIDC discovery endpoint is called by default by appending a **.well-known/openid-configuration** path to this URL. For Keycloak, use **https://host:port/realms/{realm}**, replacing **{realm}** with the Keycloak realm name. Environment variable: **QUARKUS_OIDC__TENANT__AUTH_SERVER_URL**	string	
quarkus.oidc."tenant".discovery-enabled Discovery of the OIDC endpoints. If not enabled, you must configure the OIDC endpoint URLs individually. Environment variable: **QUARKUS_OIDC__TENANT__DISCOVERY_ENABLED**	boolean	**true**
quarkus.oidc."tenant".token-path The OIDC token endpoint that issues access and refresh tokens; specified as a relative path or absolute URL. Set if **discovery-enabled** is **false** or a discovered token endpoint path must be customized. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_PATH**	string	

quarkus.oidc."tenant".revoke-path The relative path or absolute URL of the OIDC token revocation endpoint. Environment variable: **QUARKUS_OIDC__TENANT__REVOKE_PATH**	string	
quarkus.oidc."tenant".client-id The client id of the application. Each application has a client id that is used to identify the application. Setting the client id is not required if **application-type** is **service** and no token introspection is required. Environment variable: **QUARKUS_OIDC__TENANT__CLIENT_ID**	string	
quarkus.oidc."tenant".connection-delay The duration to attempt the initial connection to an OIDC server. For example, setting the duration to **20S** allows 10 retries, each 2 seconds apart. This property is only effective when the initial OIDC connection is created. For dropped connections, use the **connection-retry-count** property instead. Environment variable: **QUARKUS_OIDC__TENANT__CONNECTION_DELAY**	Duration ?	
quarkus.oidc."tenant".connection-retry-count The number of times to retry re-establishing an existing OIDC connection if it is temporarily lost. Different from **connection-delay**, which applies only to initial connection attempts. For instance, if a request to the OIDC token endpoint fails due to a connection issue, it will be retried as per this setting. Environment variable: **QUARKUS_OIDC__TENANT__CONNECTION_RETRY_COUNT**	int	**3**
quarkus.oidc."tenant".connection-timeout The number of seconds after which the current OIDC connection request times out. Environment variable: **QUARKUS_OIDC__TENANT__CONNECTION_TIMEOUT**	Duration ?	**10S**
quarkus.oidc."tenant".use-blocking-dns-lookup Whether DNS lookup should be performed on the worker thread. Use this option when you can see logged warnings about blocked Vert.x event loop by HTTP requests to OIDC server. Environment variable: **QUARKUS_OIDC__TENANT__USE_BLOCKING_DNS_LOOKUP**	boolean	**false**
quarkus.oidc."tenant".max-pool-size The maximum size of the connection pool used by the WebClient. Environment variable: **QUARKUS_OIDC__TENANT__MAX_POOL_SIZE**	int	

quarkus.oidc."tenant".credentials.secret The client secret used by the **client_secret_basic** authentication method. Must be set unless a secret is set in **client-secret** or **jwt** client authentication is required. You can use **client-secret.value** instead, but both properties are mutually exclusive. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_SECRET**	string	
quarkus.oidc."tenant".credentials.client-secret.value The client secret value. This value is ignored if **credentials.secret** is set. Must be set unless a secret is set in **client-secret** or **jwt** client authentication is required. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_CLIENT_SECRET_VALUE**	string	
quarkus.oidc."tenant".credentials.client-secret.provider.name The CredentialsProvider name, which should only be set if more than one CredentialsProvider is registered Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_CLIENT_SECRET_PROVIDER_NAME**	string	
quarkus.oidc."tenant".credentials.client-secret.provider.key The CredentialsProvider client secret key Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_CLIENT_SECRET_PROVIDER_KEY**	string	
quarkus.oidc."tenant".credentials.client-secret.method The authentication method. If the **clientSecret.value** secret is set, this method is **basic** by default. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_CLIENT_SECRET_METHOD**	tooltip: basic[**client_secret_basic** (default): The client id and secret are submitted with the HTTP Authorization Basic scheme.], tooltip: post[**cl**	

	ient_secret_post: The client id and secret are submitted as the **client_id** and **client_secret** form parameters.], tooltip: post-jwt[**client_secret_jwt**: The client id and generated JWT secret are submitted as the **client_id** and **client_secret** form parameters.], tooltip: query[client id and secret are submitted as HTTP query parameters.	

	This option is only supported for the OIDC extension.]	
quarkus.oidc."tenant".credentials.jwt.secret If provided, indicates that JWT is signed using a secret key. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_JWT_SECRET**	string	
quarkus.oidc."tenant".credentials.jwt.secret-provider.name The CredentialsProvider name, which should only be set if more than one CredentialsProvider is registered Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_JWT_SECRET_PROVIDER_NAME**	string	
quarkus.oidc."tenant".credentials.jwt.secret-provider.key The CredentialsProvider client secret key Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_JWT_SECRET_PROVIDER_KEY**	string	
quarkus.oidc."tenant".credentials.jwt.key-file If provided, indicates that JWT is signed using a private key in PEM or JWK format. You can use the **signature-algorithm** property to override the default key algorithm, **RS256**. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_JWT_KEY_FILE**	string	
quarkus.oidc."tenant".credentials.jwt.key-store-file If provided, indicates that JWT is signed using a private key from a keystore. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_JWT_KEY_STORE_FILE**	string	

quarkus.oidc."tenant".credentials.jwt.key-store-password A parameter to specify the password of the keystore file. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_JWT_KEY_STORE_PASSWORD**	string	
quarkus.oidc."tenant".credentials.jwt.key-id The private key id or alias. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_JWT_KEY_ID**	string	
quarkus.oidc."tenant".credentials.jwt.key-password The private key password. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_JWT_KEY_PASSWORD**	string	
quarkus.oidc."tenant".credentials.jwt.audience The JWT audience (**aud**) claim value. By default, the audience is set to the address of the OpenId Connect Provider's token endpoint. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_JWT_AUDIENCE**	string	
quarkus.oidc."tenant".credentials.jwt.token-key-id The key identifier of the signing key added as a JWT **kid** header. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_JWT_TOKEN_KEY_ID**	string	
quarkus.oidc."tenant".credentials.jwt.issuer The issuer of the signing key added as a JWT **iss** claim. The default value is the client id. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_JWT_ISSUER**	string	
quarkus.oidc."tenant".credentials.jwt.subject Subject of the signing key added as a JWT **sub** claim The default value is the client id. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_JWT_SUBJECT**	string	

quarkus.oidc."tenant".credentials.jwt.claims Additional claims. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_JWT_CLAIMS**	**Map<String,String>**	
quarkus.oidc."tenant".credentials.jwt.signature-algorithm The signature algorithm used for the **key-file** property. Supported values: **RS256** (default), **RS384**, **RS512**, **PS256**, **PS384**, **PS512**, **ES256**, **ES384**, **ES512**, **HS256**, **HS384**, **HS512**. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_JWT_SIGNATURE_ALGORITHM**	string	
quarkus.oidc."tenant".credentials.jwt.lifespan The JWT lifespan in seconds. This value is added to the time at which the JWT was issued to calculate the expiration time. Environment variable: **QUARKUS_OIDC__TENANT__CREDENTIALS_JWT_LIFESPAN**	int	**10**
quarkus.oidc."tenant".proxy.host The host name or IP address of the Proxy. Note: If the OIDC adapter requires a Proxy to talk with the OIDC server (Provider), set this value to enable the usage of a Proxy. Environment variable: **QUARKUS_OIDC__TENANT__PROXY_HOST**	string	
quarkus.oidc."tenant".proxy.port The port number of the Proxy. The default value is **80**. Environment variable: **QUARKUS_OIDC__TENANT__PROXY_PORT**	int	**80**
quarkus.oidc."tenant".proxy.username The username, if the Proxy needs authentication. Environment variable: **QUARKUS_OIDC__TENANT__PROXY_USERNAME**	string	
quarkus.oidc."tenant".proxy.password The password, if the Proxy needs authentication. Environment variable: **QUARKUS_OIDC__TENANT__PROXY_PASSWORD**	string	

quarkus.oidc."tenant".tls.verification Certificate validation and hostname verification, which can be one of the following **Verification** values. Default is **required**. Environment variable: **QUARKUS_OIDC__TENANT__TLS_VERIFICATION**	tooltip: required[Certificates are validated and hostname verification is enabled. This is the default value.], tooltip: certificate-validation[Certificates are validated but hostname verification is disabled.], tooltip: none[All certificates are trusted and hostname verification is disabled.]	
quarkus.oidc."tenant".tls.key-store-file An optional keystore that holds the certificate information instead of specifying separate files. Environment variable: **QUARKUS_OIDC__TENANT__TLS_KEY_STORE_FILE**	path	

quarkus.oidc."tenant".tls.key-store-file-type The type of the keystore file. If not given, the type is automatically detected based on the file name. Environment variable: **QUARKUS_OIDC__TENANT__TLS_KEY_STORE_FILE_TYPE**	string	
quarkus.oidc."tenant".tls.key-store-provider The provider of the keystore file. If not given, the provider is automatically detected based on the keystore file type. Environment variable: **QUARKUS_OIDC__TENANT__TLS_KEY_STORE_PROVIDER**	string	
quarkus.oidc."tenant".tls.key-store-password The password of the keystore file. If not given, the default value, **password**, is used. Environment variable: **QUARKUS_OIDC__TENANT__TLS_KEY_STORE_PASSWORD**	string	
quarkus.oidc."tenant".tls.key-store-key-alias The alias of a specific key in the keystore. When SNI is disabled, if the keystore contains multiple keys and no alias is specified, the behavior is undefined. Environment variable: **QUARKUS_OIDC__TENANT__TLS_KEY_STORE_KEY_ALIAS**	string	
quarkus.oidc."tenant".tls.key-store-key-password The password of the key, if it is different from the **key-store-password**. Environment variable: **QUARKUS_OIDC__TENANT__TLS_KEY_STORE_KEY_PASSWORD**	string	
quarkus.oidc."tenant".tls.trust-store-file The truststore that holds the certificate information of the certificates to trust. Environment variable: **QUARKUS_OIDC__TENANT__TLS_TRUST_STORE_FILE**	path	
quarkus.oidc."tenant".tls.trust-store-password The password of the truststore file. Environment variable: **QUARKUS_OIDC__TENANT__TLS_TRUST_STORE_PASSWORD**	string	

quarkus.oidc."tenant".tls.trust-store-cert-alias The alias of the truststore certificate. Environment variable: **QUARKUS_OIDC__TENANT__TLS_TRUST_STORE_CERT_ALIAS**	string	
quarkus.oidc."tenant".tls.trust-store-file-type The type of the truststore file. If not given, the type is automatically detected based on the file name. Environment variable: **QUARKUS_OIDC__TENANT__TLS_TRUST_STORE_FILE_TYPE**	string	
quarkus.oidc."tenant".tls.trust-store-provider The provider of the truststore file. If not given, the provider is automatically detected based on the truststore file type. Environment variable: **QUARKUS_OIDC__TENANT__TLS_TRUST_STORE_PROVIDER**	string	
quarkus.oidc."tenant".tenant-id A unique tenant identifier. It can be set by **TenantConfigResolver** providers, which resolve the tenant configuration dynamically. Environment variable: **QUARKUS_OIDC__TENANT__TENANT_ID**	string	
quarkus.oidc."tenant".tenant-enabled If this tenant configuration is enabled. The default tenant is disabled if it is not configured but a **TenantConfigResolver** that resolves tenant configurations is registered, or named tenants are configured. In this case, you do not need to disable the default tenant. Environment variable: **QUARKUS_OIDC__TENANT__TENANT_ENABLED**	boolean	**true**
quarkus.oidc."tenant".application-type The application type, which can be one of the following **ApplicationType** values. Environment variable: **QUARKUS_OIDC__TENANT__APPLICATION_TYPE**	tooltip: web-app[A **WEB_APP** is a client that serves pages, usually a front-end application. For this type of	**service**

	client the Authorization Code Flow is defined as the preferred method for authenticating users.], tooltip: service [A **SERVICE** is a client that has a set of protected HTTP resources, usually a backend application following the RESTful Architectural Design. For this type of client, the Bearer Authorization method is defined as the preferred metho	

	d for authenticating and authorizing users.], tooltip: hybrid[A combined **SERVICE** and **WEB_APP** client. For this type of client, the Bearer Authorization method is used if the Authorization header is set and Authorization Code Flow - if not.]	
quarkus.oidc."tenant".authorization-path The relative path or absolute URL of the OpenID Connect (OIDC) authorization endpoint, which authenticates users. You must set this property for **web-app** applications if OIDC discovery is disabled. This property is ignored if OIDC discovery is enabled. Environment variable: **QUARKUS_OIDC__TENANT__AUTHORIZATION_PATH**	string	
quarkus.oidc."tenant".user-info-path The relative path or absolute URL of the OIDC UserInfo endpoint. You must set this property for **web-app** applications if OIDC discovery is disabled and the **authentication.user-info-required** property is enabled. This property is ignored if OIDC discovery is enabled. Environment variable: **QUARKUS_OIDC__TENANT__USER_INFO_PATH**	string	

quarkus.oidc."tenant".introspection-path Relative path or absolute URL of the OIDC RFC7662 introspection endpoint which can introspect both opaque and JSON Web Token (JWT) tokens. This property must be set if OIDC discovery is disabled and 1) the opaque bearer access tokens must be verified or 2) JWT tokens must be verified while the cached JWK verification set with no matching JWK is being refreshed. This property is ignored if the discovery is enabled. Environment variable: **QUARKUS_OIDC__TENANT__INTROSPECTION_PATH**	string	
quarkus.oidc."tenant".jwks-path Relative path or absolute URL of the OIDC JSON Web Key Set (JWKS) endpoint which returns a JSON Web Key Verification Set. This property should be set if OIDC discovery is disabled and the local JWT verification is required. This property is ignored if the discovery is enabled. Environment variable: **QUARKUS_OIDC__TENANT__JWKS_PATH**	string	
quarkus.oidc."tenant".end-session-path Relative path or absolute URL of the OIDC end_session_endpoint. This property must be set if OIDC discovery is disabled and RP Initiated Logout support for the **web-app** applications is required. This property is ignored if the discovery is enabled. Environment variable: **QUARKUS_OIDC__TENANT__END_SESSION_PATH**	string	
quarkus.oidc."tenant".public-key The public key for the local JWT token verification. OIDC server connection is not created when this property is set. Environment variable: **QUARKUS_OIDC__TENANT__PUBLIC_KEY**	string	
quarkus.oidc."tenant".introspection-credentials.name Name Environment variable: **QUARKUS_OIDC__TENANT__INTROSPECTION_CREDENTIALS_NAME**	string	
quarkus.oidc."tenant".introspection-credentials.secret Secret Environment variable: **QUARKUS_OIDC__TENANT__INTROSPECTION_CREDENTIALS_SECRET**	string	
quarkus.oidc."tenant".introspection-credentials.include-client-id Include OpenId Connect Client ID configured with **quarkus.oidc.client-id**. Environment variable: **QUARKUS_OIDC__TENANT__INTROSPECTION_CREDENTIALS_INCLUDE_CLIENT_ID**	boolean	**true**

quarkus.oidc."tenant".roles.role-claim-path A list of paths to claims containing an array of groups. Each path starts from the top level JWT JSON object and can contain multiple segments. Each segment represents a JSON object name only; for example: "realm/groups". Use double quotes with the namespace-qualified claim names. This property can be used if a token has no **groups** claim but has the groups set in one or more different claims. Environment variable: **QUARKUS_OIDC__TENANT__ROLES_ROLE_CLAIM_PATH**	list of string	
quarkus.oidc."tenant".roles.role-claim-separator The separator for splitting strings that contain multiple group values. It is only used if the "role-claim-path" property points to one or more custom claims whose values are strings. A single space is used by default because the standard **scope** claim can contain a space-separated sequence. Environment variable: **QUARKUS_OIDC__TENANT__ROLES_ROLE_CLAIM_SEPARATOR**	string	

quarkus.oidc."tenant".roles.source Source of the principal roles. Environment variable: **QUARKUS_OIDC__TENANT__ROLES_SOURCE**	tooltip:idtoken[ID Token - the default value for the **web-app** applications.], tooltip:accesstoken[Access Token - the default value for the **service** applications; can also be used as the source of roles for the **web-app** applications.], tooltip:userinfo[User Info]	
quarkus.oidc."tenant".token.issuer The expected issuer **iss** claim value. This property overrides the**issuer** property, which might be set in OpenId Connect provider's well-known configuration. If the **iss** claim value varies depending on the host, IP address, or tenant id of the provider, you can skip the issuer verification by setting this property to **any**, but it should be done only when other options (such as configuring the provider to use the fixed **iss** claim value) are not possible. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_ISSUER**	string	

quarkus.oidc."tenant".token.audience The expected audience **aud** claim value, which can be a string or an array of strings. Note the audience claim is verified for ID tokens by default. ID token audience must be equal to the value of **quarkus.oidc.client-id** property. Use this property to override the expected value if your OpenID Connect provider sets a different audience claim value in ID tokens. Set it to **any** if your provider does not set ID token audience` claim. Audience verification for access tokens is only done if this property is configured. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_AUDIENCE**	list of string	
quarkus.oidc."tenant".token.subject-required Require that the token includes a **sub** (subject) claim which is a unique and never reassigned identifier for the current user. Note that if you enable this property and if UserInfo is also required, both the token and UserInfo **sub** claims must be present and match each other. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_SUBJECT_REQUIRED**	boolean	**false**
quarkus.oidc."tenant".token.required-claims A map of required claims and their expected values. For example, **quarkus.oidc.token.required-claims.org_id = org_xyz** would require tokens to have the **org_id** claim to be present and set to**org_xyz**. Strings are the only supported types. Use **SecurityIdentityAugmentor** to verify claims of other types or complex claims. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_REQUIRED_CLAIMS**	**Map<String,String>**	
quarkus.oidc."tenant".token.token-type Expected token type Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_TOKEN_TYPE**	string	
quarkus.oidc."tenant".token.lifespan-grace Life span grace period in seconds. When checking token expiry, current time is allowed to be later than token expiration time by at most the configured number of seconds. When checking token issuance, current time is allowed to be sooner than token issue time by at most the configured number of seconds. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_LIFESPAN_GRACE**	int	

quarkus.oidc."tenant".token.age Token age. It allows for the number of seconds to be specified that must not elapse since the **iat** (issued at) time. A small leeway to account for clock skew which can be configured with **quarkus.oidc.token.lifespan-grace** to verify the token expiry time can also be used to verify the token age property. Note that setting this property does not relax the requirement that Bearer and Code Flow JWT tokens must have a valid (**exp**) expiry claim value. The only exception where setting this property relaxes the requirement is when a logout token is sent with a back-channel logout request since the current OpenId Connect Back-Channel specification does not explicitly require the logout tokens to contain an **exp** claim. However, even if the current logout token is allowed to have no **exp** claim, the **exp** claim is still verified if the logout token contains it. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_AGE**	Duration ?	
quarkus.oidc."tenant".token.principal-claim Name of the claim which contains a principal name. By default, the **upn**, **preferred_username** and **sub** claims are checked. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_PRINCIPAL_CLAIM**	string	
quarkus.oidc."tenant".token.refresh-expired Refresh expired authorization code flow ID or access tokens. If this property is enabled, a refresh token request is performed if the authorization code ID or access token has expired and, if successful, the local session is updated with the new set of tokens. Otherwise, the local session is invalidated and the user redirected to the OpenID Provider to re-authenticate. In this case, the user might not be challenged again if the OIDC provider session is still active. For this option be effective the **authentication.session-age-extension** property should also be set to a nonzero value since the refresh token is currently kept in the user session. This option is valid only when the application is of type **ApplicationType#WEB_APP**}. This property is enabled if **quarkus.oidc.token.refresh-token-time-skew** is configured, you do not need to enable this property manually in this case. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_REFRESH_EXPIRED**	boolean	**false**
quarkus.oidc."tenant".token.refresh-token-time-skew The refresh token time skew, in seconds. If this property is enabled, the configured number of seconds is added to the current time when checking if the authorization code ID or access token should be refreshed. If the sum is greater than the authorization code ID or access token's expiration time, a refresh is going to happen. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_REFRESH_TOKEN_TIME_SKEW**	Duration ?	

quarkus.oidc."tenant".token.forced-jwk-refresh-interval The forced JWK set refresh interval in minutes. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_FORCED_JWK_REFRESH_INTERVAL**	Duration ?	**10M**
quarkus.oidc."tenant".token.header Custom HTTP header that contains a bearer token. This option is valid only when the application is of type **ApplicationType#SERVICE**}. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_HEADER**	string	
quarkus.oidc."tenant".token.authorization-scheme HTTP Authorization header scheme. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_AUTHORIZATION_SCHEME**	string	**Bearer**
quarkus.oidc."tenant".token.signature-algorithm Required signature algorithm. OIDC providers support many signature algorithms but if necessary you can restrict Quarkus application to accept tokens signed only using an algorithm configured with this property. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_SIGNATURE_ALGORITHM**	**rs256**, **rs384**, **rs512**, **ps256**, **ps384**, **ps512**, **es256**, **es384**, **es512**, **eddsa**	
quarkus.oidc."tenant".token.decryption-key-location Decryption key location. JWT tokens can be inner-signed and encrypted by OpenId Connect providers. However, it is not always possible to remotely introspect such tokens because the providers might not control the private decryption keys. In such cases set this property to point to the file containing the decryption private key in PEM or JSON Web Key (JWK) format. If this property is not set and the **private_key_jwt** client authentication method is used, the private key used to sign the client authentication JWT tokens are also used to decrypt the encrypted ID tokens. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_DECRYPTION_KEY_LOCATION**	string	

quarkus.oidc."tenant".token.allow-jwt-introspection Allow the remote introspection of JWT tokens when no matching JWK key is available. This property is set to **true** by default for backward-compatibility reasons. It is planned that this default value will be changed to **false** in an upcoming release. Also note this property is ignored if JWK endpoint URI is not available and introspecting the tokens is the only verification option. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_ALLOW_JWT_INTROSPECTION**	boolean	**true**
quarkus.oidc."tenant".token.require-jwt-introspection-only Require that JWT tokens are only introspected remotely. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_REQUIRE_JWT_INTROSPECTION_ONLY**	boolean	**false**
quarkus.oidc."tenant".token.allow-opaque-token-introspection Allow the remote introspection of the opaque tokens. Set this property to **false** if only JWT tokens are expected. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_ALLOW_OPAQUE_TOKEN_INTROSPECTION**	boolean	**true**
quarkus.oidc."tenant".token.customizer-name Token customizer name. Allows to select a tenant specific token customizer as a named bean. Prefer using **TenantFeature** qualifier when registering custom **TokenCustomizer**. Use this property only to refer to **TokenCustomizer** implementations provided by this extension. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_CUSTOMIZER_NAME**	string	
quarkus.oidc."tenant".token.verify-access-token-with-user-info Indirectly verify that the opaque (binary) access token is valid by using it to request UserInfo. Opaque access token is considered valid if the provider accepted this token and returned a valid UserInfo. You should only enable this option if the opaque access tokens must be accepted but OpenId Connect provider does not have a token introspection endpoint. This property has no effect when JWT tokens must be verified. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_VERIFY_ACCESS_TOKEN_WITH_USER_INFO**	boolean	**false**

quarkus.oidc."tenant".logout.path The relative path of the logout endpoint at the application. If provided, the application is able to initiate the logout through this endpoint in conformance with the OpenID Connect RP-Initiated Logout specification. Environment variable: **QUARKUS_OIDC__TENANT__LOGOUT_PATH**	string	
quarkus.oidc."tenant".logout.post-logout-path Relative path of the application endpoint where the user should be redirected to after logging out from the OpenID Connect Provider. This endpoint URI must be properly registered at the OpenID Connect Provider as a valid redirect URI. Environment variable: **QUARKUS_OIDC__TENANT__LOGOUT_POST_LOGOUT_PATH**	string	
quarkus.oidc."tenant".logout.post-logout-uri-param Name of the post logout URI parameter which is added as a query parameter to the logout redirect URI. Environment variable: **QUARKUS_OIDC__TENANT__LOGOUT_POST_LOGOUT_URI_PARAM**	string	**post_logout_redirect_uri**
quarkus.oidc."tenant".logout.extra-params Additional properties which is added as the query parameters to the logout redirect URI. Environment variable: **QUARKUS_OIDC__TENANT__LOGOUT_EXTRA_PARAMS**	**Map<String,String>**	
quarkus.oidc."tenant".logout.backchannel.path The relative path of the Back-Channel Logout endpoint at the application. Environment variable: **QUARKUS_OIDC__TENANT__LOGOUT_BACKCHANNEL_PATH**	string	
quarkus.oidc."tenant".logout.backchannel.token-cache-size Maximum number of logout tokens that can be cached before they are matched against ID tokens stored in session cookies. Environment variable: **QUARKUS_OIDC__TENANT__LOGOUT_BACKCHANNEL_TOKEN_CACHE_SIZE**	int	**10**
quarkus.oidc."tenant".logout.backchannel.token-cache-time-to-live Number of minutes a logout token can be cached for. Environment variable: **QUARKUS_OIDC__TENANT__LOGOUT_BACKCHANNEL_TOKEN_CACHE_TIME_TO_LIVE**	Duration ⓘ	**10M**

quarkus.oidc."tenant".logout.backchannel.clean-up-timer-interval Token cache timer interval. If this property is set, a timer checks and removes the stale entries periodically. Environment variable: **QUARKUS_OIDC__TENANT__LOGOUT_BACKCHANNEL_CLEAN_UP_TIMER_INTERVAL**	Duration ?	
quarkus.oidc."tenant".logout.backchannel.logout-token-key Logout token claim whose value is used as a key for caching the tokens. Only **sub** (subject) and **sid** (session id) claims can be used as keys. Set it to**sid** only if ID tokens issued by the OIDC provider have no **sub** but have**sid** claim. Environment variable: **QUARKUS_OIDC__TENANT__LOGOUT_BACKCHANNEL_LOGOUT_TOKEN_KEY**	string	**sub**
quarkus.oidc."tenant".logout.frontchannel.path The relative path of the Front-Channel Logout endpoint at the application. Environment variable: **QUARKUS_OIDC__TENANT__LOGOUT_FRONTCHANNEL_PATH**	string	
quarkus.oidc."tenant".certificate-chain.leaf-certificate-name Common name of the leaf certificate. It must be set if the **trust-store-file** does not have this certificate imported. Environment variable: **QUARKUS_OIDC__TENANT__CERTIFICATE_CHAIN_LEAF_CERTIFICATE_NAME**	string	
quarkus.oidc."tenant".certificate-chain.trust-store-file Truststore file which keeps thumbprints of the trusted certificates. Environment variable: **QUARKUS_OIDC__TENANT__CERTIFICATE_CHAIN_TRUST_STORE_FILE**	path	
quarkus.oidc."tenant".certificate-chain.trust-store-password A parameter to specify the password of the truststore file if it is configured with **trust-store-file**. Environment variable: **QUARKUS_OIDC__TENANT__CERTIFICATE_CHAIN_TRUST_STORE_PASSWORD**	string	

quarkus.oidc."tenant".certificate-chain.trust-store-cert-alias A parameter to specify the alias of the truststore certificate. Environment variable: **QUARKUS_OIDC__TENANT__CERTIFICATE_CHAIN_TRUST_STORE_CERT_ALIAS**	string	
quarkus.oidc."tenant".certificate-chain.trust-store-file-type An optional parameter to specify type of the truststore file. If not given, the type is automatically detected based on the file name. Environment variable: **QUARKUS_OIDC__TENANT__CERTIFICATE_CHAIN_TRUST_STORE_FILE_TYPE**	string	
quarkus.oidc."tenant".authentication.response-mode Authorization code flow response mode. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_RESPONSE_MODE**	tooltip: query[Authorization response parameters are encoded in the query string added to the **redirect_uri**], tooltip: form-post[Authorization response parameters are encoded as HTML form values that are auto-submitted in the browse	**query**

	r and transmitted by the HTTP POST method using the application/x-www-form-urlencoded content type]	
quarkus.oidc."tenant".authentication.redirect-path The relative path for calculating a **redirect_uri** query parameter. It has to start from a forward slash and is appended to the request URI's host and port. For example, if the current request URI is **https://localhost:8080/service**, a **redirect_uri** parameter is set to **https://localhost:8080/** if this property is set to / and be the same as the request URI if this property has not been configured. Note the original request URI is restored after the user has authenticated if **restorePathAfterRedirect** is set to **true**. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_REDIRECT_PATH**	string	
quarkus.oidc."tenant".authentication.restore-path-after-redirect If this property is set to **true**, the original request URI which was used before the authentication is restored after the user has been redirected back to the application. Note if **redirectPath** property is not set, the original request URI is restored even if this property is disabled. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_RESTORE_PATH_AFTER_REDIRECT**	boolean	**false**
quarkus.oidc."tenant".authentication.remove-redirect-parameters Remove the query parameters such as **code** and **state** set by the OIDC server on the redirect URI after the user has authenticated by redirecting a user to the same URI but without the query parameters. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_REMOVE_REDIRECT_PARAMETERS**	boolean	**true**

quarkus.oidc."tenant".authentication.error-path Relative path to the public endpoint which processes the error response from the OIDC authorization endpoint. If the user authentication has failed, the OIDC provider returns an **error** and an optional **error_description** parameters, instead of the expected authorization **code**. If this property is set, the user is redirected to the endpoint which can return a user-friendly error description page. It has to start from a forward slash and is appended to the request URI's host and port. For example, if it is set as **/error** and the current request URI is **https://localhost:8080/callback?error=invalid_scope**, a redirect is made to **https://localhost:8080/error?error=invalid_scope**. If this property is not set, HTTP 401 status is returned in case of the user authentication failure. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_ERROR_PATH**	string	
quarkus.oidc."tenant".authentication.verify-access-token Both ID and access tokens are fetched from the OIDC provider as part of the authorization code flow. ID token is always verified on every user request as the primary token which is used to represent the principal and extract the roles. Access token is not verified by default since it is meant to be propagated to the downstream services. The verification of the access token should be enabled if it is injected as a JWT token. Access tokens obtained as part of the code flow are always verified if **quarkus.oidc.roles.source** property is set to **accesstoken** which means the authorization decision is based on the roles extracted from the access token. Bearer access tokens are always verified. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_VERIFY_ACCESS_TOKEN**	boolean	**false**
quarkus.oidc."tenant".authentication.force-redirect-https-scheme Force **https** as the **redirect_uri** parameter scheme when running behind an SSL/TLS terminating reverse proxy. This property, if enabled, also affects the logout **post_logout_redirect_uri** and the local redirect requests. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_FORCE_REDIRECT_HTTPS_SCHEME**	boolean	**false**
quarkus.oidc."tenant".authentication.scopes List of scopes Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_SCOPES**	list of string	

quarkus.oidc."tenant".authentication.nonce-required Require that ID token includes a **nonce** claim which must match**nonce** authentication request query parameter. Enabling this property can help mitigate replay attacks. Do not enable this property if your OpenId Connect provider does not support setting **nonce** in ID token or if you work with OAuth2 provider such as **GitHub** which does not issue ID tokens. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_NONCE_REQUIRED**	boolean	**false**
quarkus.oidc."tenant".authentication.add-openid-scope Add the **openid** scope automatically to the list of scopes. This is required for OpenId Connect providers, but does not work for OAuth2 providers such as Twitter OAuth2, which do not accept this scope and throw errors. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_ADD_OPENID_SCOPE**	boolean	**true**
quarkus.oidc."tenant".authentication.extra-params Additional properties added as query parameters to the authentication redirect URI. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_EXTRA_PARAMS**	**Map<String,String>**	
quarkus.oidc."tenant".authentication.forward-params Request URL query parameters which, if present, are added to the authentication redirect URI. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_FORWARD_PARAMS**	list of string	
quarkus.oidc."tenant".authentication.cookie-force-secure If enabled the state, session, and post logout cookies have their **secure** parameter set to **true** when HTTP is used. It might be necessary when running behind an SSL/TLS terminating reverse proxy. The cookies are always secure if HTTPS is used, even if this property is set to false. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_COOKIE_FORCE_SECURE**	boolean	**false**
quarkus.oidc."tenant".authentication.cookie-suffix Cookie name suffix. For example, a session cookie name for the default OIDC tenant is **q_session** but can be changed to**q_session_test** if this property is set to**test**. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_COOKIE_SUFFIX**	string	

quarkus.oidc."tenant".authentication.cookie-path Cookie path parameter value which, if set, is used to set a path parameter for the session, state and post logout cookies. The **cookie-path-header** property, if set, is checked first. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_COOKIE_PATH**	string	/
quarkus.oidc."tenant".authentication.cookie-path-header Cookie path header parameter value which, if set, identifies the incoming HTTP header whose value is used to set a path parameter for the session, state and post logout cookies. If the header is missing, the **cookie-path** property is checked. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_COOKIE_PATH_HEADER**	string	
quarkus.oidc."tenant".authentication.cookie-domain Cookie domain parameter value which, if set, is used for the session, state and post logout cookies. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_COOKIE_DOMAIN**	string	
quarkus.oidc."tenant".authentication.cookie-same-site SameSite attribute for the session cookie. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_COOKIE_SAME_SITE**	**strict**, **lax**, **none**	**lax**
quarkus.oidc."tenant".authentication.allow-multiple-code-flows If a state cookie is present, a **state** query parameter must also be present and both the state cookie name suffix and state cookie value must match the value of the **state** query parameter when the redirect path matches the current path. However, if multiple authentications are attempted from the same browser, for example, from the different browser tabs, then the currently available state cookie might represent the authentication flow initiated from another tab and not related to the current request. Disable this property to permit only a single authorization code flow in the same browser. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_ALLOW_MULTIPLE_CODE_FLOWS**	boolean	**true**

quarkus.oidc."tenant".authentication.fail-on-missing-state-param Fail with the HTTP 401 error if the state cookie is present but no state query parameter is present. When either multiple authentications are disabled or the redirect URL matches the original request URL, the stale state cookie might remain in the browser cache from the earlier failed redirect to an OpenId Connect provider and be visible during the current request. For example, if Single-page application (SPA) uses XHR to handle redirects to the provider which does not support CORS for its authorization endpoint, the browser blocks it and the state cookie created by Quarkus remains in the browser cache. Quarkus reports an authentication failure when it detects such an old state cookie but find no matching state query parameter. Reporting HTTP 401 error is usually the right thing to do in such cases, it minimizes a risk of the browser redirect loop but also can identify problems in the way SPA or Quarkus application manage redirects. For example, enabling **java-script-auto-redirect** or having the provider redirect to URL configured with **redirect-path** might be needed to avoid such errors. However, setting this property to **false** might help if the above options are not suitable. It causes a new authentication redirect to OpenId Connect provider. Doing so might increase the risk of browser redirect loops. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_FAIL_ON_MISSING_STATE_PARAM**	boolean	**false**
quarkus.oidc."tenant".authentication.user-info-required If this property is set to **true**, an OIDC UserInfo endpoint is called. This property is enabled if **quarkus.oidc.roles.source** is **userinfo**. or **quarkus.oidc.token.verify-access-token-with-user-info** is **true** or **quarkus.oidc.authentication.id-token-required** is set to **false**, you do not need to enable this property manually in these cases. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_USER_INFO_REQUIRED**	boolean	**false**
quarkus.oidc."tenant".authentication.session-age-extension Session age extension in minutes. The user session age property is set to the value of the ID token life-span by default and the user is redirected to the OIDC provider to re-authenticate once the session has expired. If this property is set to a nonzero value, then the expired ID token can be refreshed before the session has expired. This property is ignored if the **token.refresh-expired** property has not been enabled. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_SESSION_AGE_EXTENSION**	Duration ⓘ	**5M**

quarkus.oidc."tenant".authentication.java-script-auto-redirect If this property is set to **true**, a normal 302 redirect response is returned if the request was initiated by a JavaScript API such as XMLHttpRequest or Fetch and the current user needs to be (re)authenticated, which might not be desirable for Single-page applications (SPA) since it automatically following the redirect might not work given that OIDC authorization endpoints typically do not support CORS. If this property is set to **false**, a status code of**499** is returned to allow SPA to handle the redirect manually if a request header identifying current request as a JavaScript request is found. **X-Requested-With** request header with its value set to either **JavaScript** or **XMLHttpRequest** is expected by default if this property is enabled. You can register a custom **JavaScriptRequestChecker** to do a custom JavaScript request check instead. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_JAVA_SCRIPT_AUTO_REDIRECT**	boolean	**true**
quarkus.oidc."tenant".authentication.id-token-required Requires that ID token is available when the authorization code flow completes. Disable this property only when you need to use the authorization code flow with OAuth2 providers which do not return ID token - an internal IdToken is generated in such cases. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_ID_TOKEN_REQUIRED**	boolean	**true**
quarkus.oidc."tenant".authentication.internal-id-token-lifespan Internal ID token lifespan. This property is only checked when an internal IdToken is generated when Oauth2 providers do not return IdToken. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_INTERNAL_ID_TOKEN_LIFESPAN**	Duration ⓘ	**5M**
quarkus.oidc."tenant".authentication.pkce-required Requires that a Proof Key for Code Exchange (PKCE) is used. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_PKCE_REQUIRED**	boolean	**false**

quarkus.oidc."tenant".authentication.state-secret Secret used to encrypt Proof Key for Code Exchange (PKCE) code verifier and/or nonce in the code flow state. This secret should be at least 32 characters long. If this secret is not set, the client secret configured with either **quarkus.oidc.credentials.secret** or **quarkus.oidc.credentials.client-secret.value** is checked. Finally, **quarkus.oidc.credentials.jwt.secret** which can be used for **client_jwt_secret** authentication is checked. A client secret is not be used as a state encryption secret if it is less than 32 characters long. The secret is auto-generated if it remains uninitialized after checking all of these properties. Error is reported if the secret length is less than 16 characters. Environment variable: **QUARKUS_OIDC__TENANT__AUTHENTICATION_STATE_SECRET**	string	
quarkus.oidc."tenant".code-grant.extra-params Additional parameters, in addition to the required **code** and **redirect-uri** parameters, which must be included to complete the authorization code grant request. Environment variable: **QUARKUS_OIDC__TENANT__CODE_GRANT_EXTRA_PARAMS**	**Map<String,String>**	
quarkus.oidc."tenant".code-grant.headers Custom HTTP headers which must be sent to complete the authorization code grant request. Environment variable: **QUARKUS_OIDC__TENANT__CODE_GRANT_HEADERS**	**Map<String,String>**	

quarkus.oidc."tenant".token-state-manager.strategy Default TokenStateManager strategy. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_STATE_MANAGER_STRATEGY**	tooltip: keep-all-tokens[Keep ID, access and refresh tokens.], tooltip:id-token[Keep ID token only], tooltip:id-refresh-tokens[Keep ID and refresh tokens only]	**keep-all-tokens**
quarkus.oidc."tenant".token-state-manager.split-tokens Default TokenStateManager keeps all tokens (ID, access and refresh) returned in the authorization code grant response in a single session cookie by default. Enable this property to minimize a session cookie size Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_STATE_MANAGER_SPLIT_TOKENS**	boolean	**false**
quarkus.oidc."tenant".token-state-manager.encryption-required Mandates that the Default TokenStateManager encrypt the session cookie that stores the tokens. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_STATE_MANAGER_ENCRYPTION_REQUIRED**	boolean	**true**

quarkus.oidc."tenant".token-state-manager.encryption-secret The secret used by the Default TokenStateManager to encrypt the session cookie storing the tokens when **encryption-required** property is enabled. If this secret is not set, the client secret configured with either **quarkus.oidc.credentials.secret** or **quarkus.oidc.credentials.client-secret.value** is checked. Finally, **quarkus.oidc.credentials.jwt.secret** which can be used for **client_jwt_secret** authentication is checked. The secret is auto-generated if it remains uninitialized after checking all of these properties. The length of the secret used to encrypt the tokens should be at least 32 characters long. A warning is logged if the secret length is less than 16 characters. Environment variable: **QUARKUS_OIDC__TENANT__TOKEN_STATE_MANAGER_ENCRYPTION_SECRET**	string	
quarkus.oidc."tenant".allow-token-introspection-cache Allow caching the token introspection data. Note enabling this property does not enable the cache itself but only permits to cache the token introspection for a given tenant. If the default token cache can be used, see **OidcConfig.TokenCache** to enable it. Environment variable: **QUARKUS_OIDC__TENANT__ALLOW_TOKEN_INTROSPECTION_CACHE**	boolean	**true**
quarkus.oidc."tenant".allow-user-info-cache Allow caching the user info data. Note enabling this property does not enable the cache itself but only permits to cache the user info data for a given tenant. If the default token cache can be used, see **OidcConfig.TokenCache** to enable it. Environment variable: **QUARKUS_OIDC__TENANT__ALLOW_USER_INFO_CACHE**	boolean	**true**
quarkus.oidc."tenant".cache-user-info-in-idtoken Allow inlining UserInfo in IdToken instead of caching it in the token cache. This property is only checked when an internal IdToken is generated when Oauth2 providers do not return IdToken. Inlining UserInfo in the generated IdToken allows to store it in the session cookie and avoids introducing a cached state. Environment variable: **QUARKUS_OIDC__TENANT__CACHE_USER_INFO_IN_IDTOKEN**	boolean	**false**

quarkus.oidc."tenant".jwks.resolve-early If JWK verification keys should be fetched at the moment a connection to the OIDC provider is initialized. Disabling this property delays the key acquisition until the moment the current token has to be verified. Typically it can only be necessary if the token or other telated request properties provide an additional context which is required to resolve the keys correctly. Environment variable: **QUARKUS_OIDC__TENANT__JWKS_RESOLVE_EARLY**	boolean	**true**
quarkus.oidc."tenant".jwks.cache-size Maximum number of JWK keys that can be cached. This property is ignored if the **resolve-early** property is set to true. Environment variable: **QUARKUS_OIDC__TENANT__JWKS_CACHE_SIZE**	int	**10**
quarkus.oidc."tenant".jwks.cache-time-to-live Number of minutes a JWK key can be cached for. This property is ignored if the **resolve-early** property is set to true. Environment variable: **QUARKUS_OIDC__TENANT__JWKS_CACHE_TIME_TO_LIVE**	Duration ⓘ	**10M**
quarkus.oidc."tenant".jwks.clean-up-timer-interval Cache timer interval. If this property is set, a timer checks and removes the stale entries periodically. This property is ignored if the **resolve-early** property is set to true. Environment variable: **QUARKUS_OIDC__TENANT__JWKS_CLEAN_UP_TIMER_INTERVAL**	Duration ⓘ	
quarkus.oidc."tenant".provider Well known OpenId Connect provider identifier Environment variable: **QUARKUS_OIDC__TENANT__PROVIDER**	**apple**, **discord**, **facebook**, **github**, **google**, **linkedin**, **mastodon**, **microsoft**, **spotify**, **strava**, **twitch**, **twitter**, **x**	

ABOUT THE DURATION FORMAT

To write duration values, use the standard **java.time.Duration** format. See the Duration#parse() Java API documentation for more information.

You can also use a simplified format, starting with a number:

- If the value is only a number, it represents time in seconds.
- If the value is a number followed by **ms**, it represents time in milliseconds.

In other cases, the simplified format is translated to the **java.time.Duration** format for parsing:

- If the value is a number followed by **h**, **m**, or **s**, it is prefixed with **PT**.
- If the value is a number followed by **d**, it is prefixed with **P**.

6.1. REFERENCES

- OIDC Bearer token authentication
- Protect a service application by using OpenID Connect (OIDC) Bearer token authentication
- OpenID Connect
- OpenID Connect and OAuth2 Client and Filters Reference Guide
- Choosing between OpenID Connect, SmallRye JWT, and OAuth2 authentication mechanisms
- Combining authentication mechanisms
- Quarkus Security