Using OAuth 2.0 and OpenID Connect with Caché

Version 2018.1
2024-05-02

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using OAuth 2.0 and OpenID Connect with Caché
Caché Version 2018.1 2024-05-02

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

About This Book

This book describes how to use Caché support for OAuth 2.0 and OpenID Connect. It contains the following sections:

- Introduction
- Overview of OAuth 2.0 and OpenID Connect
- Using a Caché Web Application as an OAuth 2.0 Client
- Using a Caché Web Application as an OAuth 2.0 Resource Server
- Using Caché as an OAuth 2.0 Authorization Server
- Creating Configuration Items Programmatically
- Implementing DirectLogin()
- Certificates and JWTs (JSON Web Tokens)

For general information, see the *InterSystems Documentation Guide*.

1

Overview of OAuth 2.0 and OpenID Connect

This chapter provides a brief overview of OAuth 2.0 authorization framework and OpenID Connect. The next chapter introduces Caché support for OAuth 2.0 and OpenID Connect.

1.1 Basics

The OAuth 2.0 authorization framework enables a third-party application (generally known as a client) to obtain limited access to an HTTP service (a resource). The access is limited; the client can obtain only specific information or can use only specific services. An authorization server either orchestrates an approval interaction or directly gives access. OpenID Connect extends this framework and adds authentication to it.

1.2 Roles

The OAuth 2.0 framework defines four roles:

- *Resource owner* — Usually a user.
- *Resource server* — A server that hosts protected data and/or services.
- *Client* — An application that requests limited access to a resource server. This can be a client-server application or can be an application that has no server (such as a JavaScript application or mobile application).
- *Authorization server* — A server that is responsible for issuing access tokens, with which the client can access the resource server. This server can be the same application as the authorization server but can also be a different application.

The client, resource server, and authorization server are known to each other, by prior arrangement. An authorization server has a registry of clients, which specifies the client servers and resource servers that can communicate with it. When it registers a client, an authorization server generates a *client ID* and a *client secret*, the latter of which must be kept secret. Depending on how the client will communicate with the authorization server, the client server might need the client secret; in that case, it is necessary to convey the client secret securely to the client server. In some scenarios (such as a JavaScript application), it is impossible for the client to protect the client secret; in these scenarios, the client must communicate with the authorization server in a way that does not require the client secret.

1.3 Access Tokens

An *access token* contains information about the identity of the user or client, as well as metadata such an expiration date, expected issuer name, expected audience, scope, and so on.

The general purpose of an access token is to enable a client to access specific data or services available via HTTP at a resource server. In the overall flow, the client application requests an access token from the authorization server. After receiving this token, the client uses the access token within HTTP requests to the resource server. The resource server returns the requested information only when it receives requests that contain a valid access token.

An access token can also be revoked, if the authorization server supports this.

1.3.1 Forms of Access Tokens

Caché supports two forms of access tokens:

- JSON Web Tokens (JWTs). A *JWT* is a JSON object. A JWT can be digitally signed, encrypted, or both.

  Note that one kind of JWT is an ID token; this is specific to OpenID Connect.

  A JWT can be signed, encrypted, or both.

- *Opaque access tokens* (also known as *reference tokens*). This form of access token is just the identifier of a token that is stored elsewhere, specifically on the authorization server. The identifier is a long, random string, intended to be very difficult to guess.

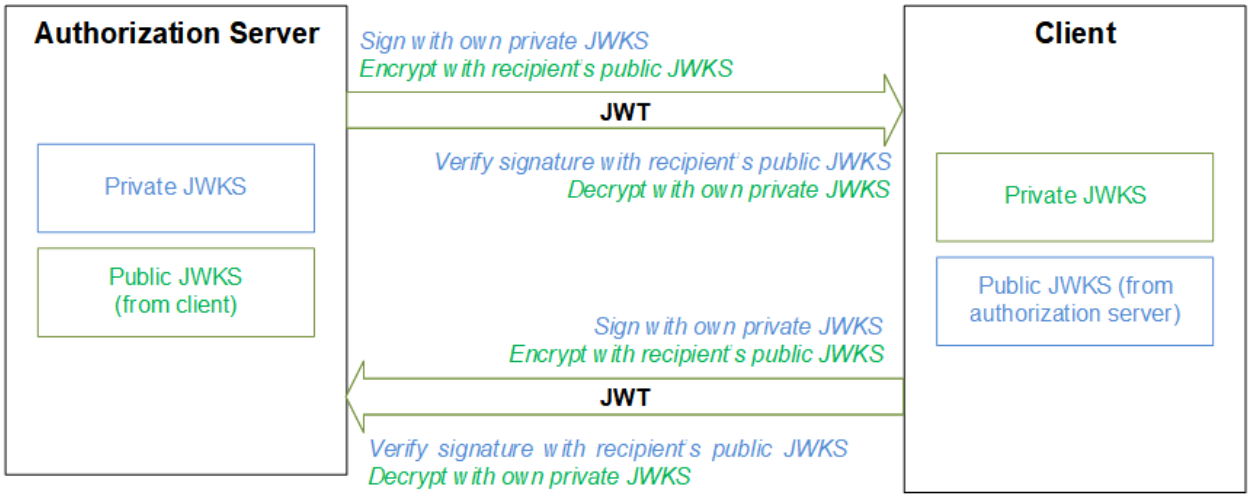
1.3.2 Claims

An access token contains a set of *claims* that communicate the identity of the user or client, or that communicate metadata such the token's expiration date, expected issuer name, expected audience, scope, and so on. The OpenID Connect Core specification defines a standard set of claims, and other claims may be used as well.

1.3.3 JWTs and JWKSs

As noted above, a JWT can be signed, encrypted, or both. In most cases, the participants in the OAuth 2.0 framework use pairs of JWKSs (JSON web key sets) for this purpose. In any pair of JWKSs, one JWKS is private and contains all the needed private keys (per algorithm) as well as the client secret for use as a symmetric key; this JWKS is never shared. The other JWKS contains the corresponding public keys and is publicly available.

Each participant has a private JWKS and provides other participants with the corresponding public JWKS. The owner of a private JWKS uses that JWKS for signing outbound JWTs and decrypting inbound JWTs. The other parties use the corresponding public JWKS to encrypt outbound JWTs and verify signatures of inbound JWTs, as shown in the following figure:



1.4 Grant Types and Flows

In the OAuth 2.0 framework, a *grant type* specifies how the authorization server should process the request for authorization. The client specifies the grant type within the initial request to the authorization server. The OAuth 2.0 specification describes four grant types, as well as an extensibility mechanism for defining additional types. In general, each grant type corresponds to a different overall flow.

The four grant types are as follows:

- *Authorization code* — This grant type can be used only with a client application that has a corresponding server. In this grant type, the authorization server displays a login page with which the user provides a username and password; these are never shared with the client. If the username and password correspond to a valid user (and if other elements of the request are in order), the authorization server first issues an authorization code, which it returns to the client. The client then uses the authorization code to obtain an access token.

  The request for the authorization code is visible in the browser, as is the response. The request for the access token, however, is a server-to-server interaction, as is that response. Thus the access token is never visible in the browser.

- *Implicit* — As with the previously listed grant type, the authorization server displays a login page, and the client never has access to the user's credentials. However, in the implicit grant type, the client directly requests and receives an access token. This grant type is useful for pure client applications such as JavaScript clients or mobile applications.

- *Resource owner password credentials* — In this grant type, the client prompts the user for a username and password and then uses those credentials to obtain an access token from the authorization server. This grant type is suitable only with trusted applications.

- *Client credentials grant type* — In this grant type, there is no user context, and the client application is unattended. The client uses its client ID and client secret to obtain an access token from the authorization server.

Note that in the OAuth 2.0 framework, in general, all HTTP requests are protected by SSL/TLS.

In addition, when a client sends a request to the authorization server, that request must be authenticated. The OAuth 2.0 specification describes the ways in which a client can authenticate the request.

1.5 Scopes

The authorization server allows the client to specify the scope of the access request using the `scope` request parameter. In turn, the authorization server uses the `scope` response parameter to inform the client of the scope of the access token issued.

OpenID Connect is an extension to the OAuth 2.0 authorization process. To request authentication, the client includes the `openid` scope value in the request to the authorization server. The authorization server returns information about the authentication in a JWT called an *ID token*. An ID token contains a specific set of claims, listed in the OpenID Connect Core specification.

1.6 Endpoints in an Authorization Server

An authorization server provides some or all of the following URLs or endpoints, which can process requests of varying kinds:

Endpoint	Purpose
Authorization endpoint	Returns an authorization code (applies only to authorization code grant type)
Token endpoint	Returns an access token
Userinfo endpoint	Returns a JSON object that contains claims about the authenticated user (applies only to OpenID Connect)
Token introspection endpoint	Returns a JSON object that contains claims determined by examining an access token
Token revocation endpoint	Revokes a token

2

How Caché Supports OAuth 2.0 and OpenID Connect

This chapter introduces Caché support for OAuth 2.0 and OpenID Connect.

2.1 Supported Scenarios

With Caché support for OAuth 2.0 and OpenID connect, you can do any or all of the following:

- Use a Caché web application as a client
- Use a Caché web application as a resource server
- Use a Caché instance as an authorization server

For example, you can use a Caché web application as a client of an authorization server that uses third-party technology. Or you can use third-party clients with an authorization server that is built on Caché. The resource server or resource servers could be implemented in Caché or in a different technology.

In all cases, the authorization server is the most complex element and is generally created first. You create clients later. When you create a client, it is generally necessary to understand the capabilities and requirements of the authorization server, such as the scopes it supports.

2.2 Caché Support for OAuth 2.0 and OpenID Connect

The Caché support for OAuth 2.0 and OpenID Connect consists of the following elements:

- Configuration pages in the Management Portal.

  If you configure a client (or a resource server), use the options at **System Administration** > **Security** > **OAuth 2.0** > **Client Configuration**.

  If you configure an authorization server, use the options at **System Administration** > **Security** > **OAuth 2.0** > **Server Configuration**.

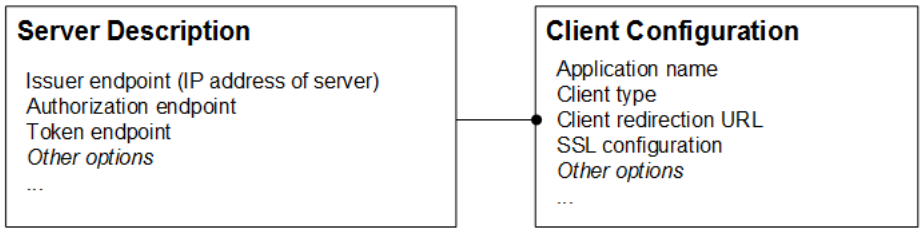
- Classes in the %SYS.OAuth2 package. These classes are the client API. If you define a Caché web application as an OAuth 2.0 client, your client uses methods in these classes.

- Classes in the %OAuth2 package. If you use a Caché instance as an OAuth 2.0 authorization server, you customize the server by subclassing one or more of the classes in the package %OAuth2.Server. Other classes in %OAuth2 provide utility methods for your code to call.
- Classes in the OAuth2 package (in the CACHESYS database). These include persistent classes for internal use by Caché, and you can ignore most of them. However, if you want to create configuration items programmatically, you would use a subset of the classes in this package.

The following subsections provide an overview of the configuration items.

2.2.1 Configuration Items on a Client

Within a Caché instance that is acting as an OAuth 2.0 client, it is necessary to define two connected configuration items for a given client application: a *server description* (which describes the authorization server) and a *client configuration* (which configures the client). A given Caché instance can have any number of server descriptions. Each server description has multiple client configurations, as shown in the following figure, which also indicates some of the information stored in these configuration items:

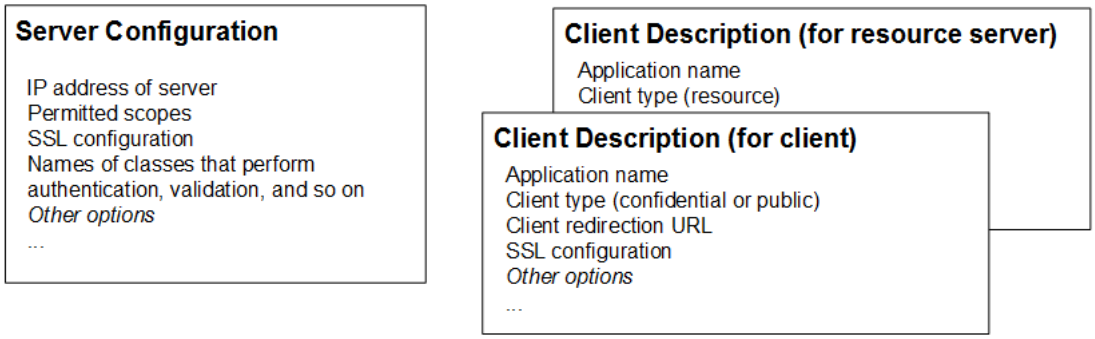


This architecture intended to simplify configuration, because it enables you to define multiple client configurations that use the same authorization server without needing to repeat the details of the authorization server.

You can create these items via the Management Portal, as described in the chapter "Using a Caché Web Application as an OAuth 2.0 Client." Or you can create them programmatically, as described in the appendix "Creating Configuration Items Programmatically."

2.2.2 Configuration Items on the Server

Within a Caché instance that is acting as an OAuth 2.0 authorization server, it is necessary to define a *server configuration* (which configures the authorization server) and a number of *client descriptions*. The following figure indicates some of the information stored in these configuration items.



A given Caché instance can have at most one server configuration and can have many client descriptions. One client description is necessary for each client application. A client description is also necessary for each resource server that uses any endpoints of the authorization server. If a resource server does not use any endpoints of the authorization server, there is no need to create a client description for it.

You can create these items via the Management Portal, as described in the chapter "Using Caché as an OAuth 2.0 Authorization Server." Or you can create them programmatically, as described in the appendix "Creating Configuration Items Programmatically."

2.3 Standards Supported in Caché

This section lists the standards that Caché supports for OAuth 2.0 and Open ID Connect:

- The OAuth 2.0 Authorization Framework (RFC 6749) — See https://datatracker.ietf.org/doc/rfc6749
- The OAuth 2.0 Authorization Framework: Bearer Token Usage (RFC 6750) — See https://datatracker.ietf.org/doc/rfc6750
- OAuth 2.0 Token Revocation (RFC 7009) — See https://datatracker.ietf.org/doc/rfc7009
- JSON Web Token (JWT) (RFC 7519) — See https://datatracker.ietf.org/doc/rfc7519
- OAuth 2.0 Token Introspection (RFC 7662) — See https://datatracker.ietf.org/doc/rfc7662
- OpenID Connect Core 1.0 — See http://openid.net/specs/openid-connect-core-1_0.html
- OAuth 2.0 Form Post Response Mode — See http://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html
- JSON Web Key (JWK) (RFC 7517) — See https://datatracker.ietf.org/doc/rfc7517
- OpenID Connect Discovery 1.0 — See https://openid.net/specs/openid-connect-discovery-1_0.html
- OpenID Connect Dynamic Client Registration — See http://openid.net/specs/openid-connect-registration-1_0-19.html

3

Using a Caché Web Application as an OAuth 2.0 Client

This chapter describes how to use a Caché web application as a client application that uses the OAuth 2.0 framework. It discusses the following:

- Prerequisites
- Configuration requirements
- Outline of client code requirements
- Obtaining tokens
- Examining tokens
- Adding an access token to an HTTP request
- Optionally defining delegated authentication
- Variations
- Revoking access tokens
- Rotating client keys
- Getting a new public JWKS from the authorization server

This chapter primarily discusses the scenario in which a Caché web application is the client of a web server/client application and uses the authorization code grant type. See the last section for details on other grant types and other variations.

3.1 Prerequisites for the Caché Client

Before starting the tasks described in this chapter, make sure the following items are available:

- An OAuth 2 authorization server. Later you will need to know specific details about this server. Some of the details apply when you configure the client within Caché:
  - Location of the authorization server (issuer endpoint)
  - Location of the authorization endpoint
  - Location of the token endpoint

  - Location of the Userinfo endpoint (if supported; see OpenID Connect Core)
  - Location of the token introspection endpoint (if supported; see RFC 7662)
  - Location of the token revocation endpoint (if supported; see RFC 7009)
  - Whether the authorization server supports dynamic registration

  Other details apply when you write the client code:

  - Grant types supported by this server
  - Scopes supported by this server. For example, the server may or may not support `openid` and `profile`, which are special scopes defined by OpenID Connect Core.
  - Other requirements for requests made to this server

- If the authorization server does not support dynamic client registration, the Caché application must be registered as a client of the OAuth 2.0 authorization server, and you must have the client ID and client secret for this client. The details depend upon the implementation of the authorization server. (If the server does support dynamic registration, you can register the client while configuring it as described in this chapter.)

3.2 Configuration Requirements

To use a Caché web application as an OAuth 2.0 client, perform the following configuration tasks:

- For the web server that is serving Caché, configure that web server to use SSL. It is beyond the scope of this documentation to describe how to configure a web server to use SSL.
- Create a Caché SSL configuration for use by the client.

  This should be a client SSL configuration; no certificate is needed. The configuration is used to connect to a web server. Via this connection, the client communicates with the authorization server to obtain access tokens, call the Userinfo endpoint, call the introspection endpoint, and so on.

  For details on creating SSL configurations, see the chapter "Using SSL/TLS with Caché" in the *Caché Security Administration Guide*.

  Each SSL configuration has a unique name. For reference, the documentation refers to this one as `sslconfig`, but you can use any unique name.
- Create the OAuth 2.0 configuration items for the client. To do so, first create the server description and then create the client configuration, as described in the subsections.

  For both items, to find the needed options in the Management Portal, select **System Administration** > **Security** > **OAuth 2.0** > **Client Configuration**. This page provides the options needed when you create an OAuth 2.0 configuration on a client machine (that is, on any machine other than one being used as an authorization server).

  On a client machine, do not use the menu **System Administration** > **Security** > **OAuth 2.0** > **Server Configuration**.

3.2.1 Creating a Server Description (Using Discovery)

1. In the Management Portal, select **System Administration** > **Security** > **OAuth 2.0** > **Client Configuration**.

   This displays a page that lists any server descriptions that are available on this instance. In any given row, the **Issuer endpoint** column indicates the issuer endpoint for the server description. The **Client Count** column indicates the number

of client configurations associated with the given server description. In the last column, the **Client Configurations** link enables you to create, view, edit, and delete the associated client configurations.

2. Select **Create Server Configuration**.

   The Management Portal then displays a new page where you can enter details for the server description.

3. Specify the following details:

   - **Issuer endpoint** (required) — Enter the endpoint URL to be used to identify the authorization server.
   - **SSL/TLS configuration** (required) — Select the SSL/TLS configuration to use when making the dynamic client registration request.
   - **Registration access token** — Optionally enter the initial registration access token to use as a bearer token to authorize the dynamic client registration request.

4. Select **Discover and Save**.

   Caché then communicates with the given authorization server, retrieves information needed in the server description, and then saves that information.

   The Management Portal then redisplays the list of server descriptions.

3.2.1.1 Manually Creating a Server Description (No Discovery)

To manually create a server description (rather than using discovery), first display the server description page (steps 1 and 2 above) and then select **Manual**. Then the page displays a larger set of options, as follows:

- **Issuer endpoint** (required) — Enter the endpoint URL to be used to identify the authorization server.
- **Authorization endpoint** (required) — Enter the endpoint URL to be used when requesting an authorization code from the authorization server.
- **Token endpoint** (required) — Enter the endpoint URL to be used when requesting an access token from the authorization server.
- **Userinfo endpoint** — Enter the endpoint URL to be used when making a Userinfo request using an access token from the authorization server for authorization.
- **Token introspection endpoint** — Enter the endpoint URL to be used when making a token introspection request using the `client_id` and `client_secret` for authorization. See RFC 7662.
- **Token revocation endpoint**— Enter the endpoint URL to be used when making a token revocation request using the `client_id` and `client_secret` for authorization. See RFC 7009.
- **JSON Web Token (JWT) Settings** — Specifies the source of the public keys that the client should use for signature verification and decryption of JWTs from the authorization server.

  By default, the authorization server generates a pair of JWKSs (JSON web key sets). One JWKS is private and contains all the needed private keys (per algorithm) as well as the client secret for use as a symmetric key; this JWKS is never shared. The other JWKS contains the corresponding public keys and is publicly available. The process of creating the server definition also copies the public JWKS from the authorization server to the client for its use in signature verification and encryption of JWTs.

  – **JWKS from URL** — Specify a URL that points to a public JWKS and then load the JWKS into Caché.
  – **JWKS from file** — Select a file that contains a public JWKS and then load that file into Caché.
  – **X509 certificate** — For details, see "Using Certificates for an OAuth 2.0 Authorization Server," in the appendix "Certificates and JWTs (JSON Web Tokens)."

To access any of these options, first select **Source other than dynamic registration**.

Specify these values and then select **Save**.

3.2.2 Configuring and Dynamically Registering a Client

This section describes how to create a client configuration and dynamically register the client.

1. In the Management Portal, select **System Administration** > **Security** > **OAuth 2.0** > **Client Configuration**.

   The Management Portal displays the list of server descriptions.

2. Click the **Client Configurations** link in the row for the server description with which this client configuration should be associated.

   The Management Portal then displays the list of client configurations associated with the server description. This list is initially empty.

3. Click **Create Client Configuration**.

   The Management Portal then displays a new page where you can enter details.

4. On the **General** tab, specify the following details:

   - **Application name** — Specify a short name for the application.
   - **Client name** — Specify the client name to display to the end user.
   - **Description** — Specify an optional description of the application.
   - **Enabled** — Optionally clear this check box if you want to prevent this application from being used.
   - **Client Type** — Select one of the following:
     - **Confidential** — Specifies that the client is a confidential client, per RFC 6749.

       This chapter primarily discusses the scenario in which the client uses the authorization code grant type. For this scenario, specify **Client Type** as **Confidential**. For other grant types, see "Variations," later in this chapter.
     - **Public** — Specifies that the client is a public client, per RFC 6749.
     - **Resource Server** — Specifies that the client is a resource server which is not also a client.
   - **SSL/TLS configuration** — Select the SSL configuration you created for use by the client (for example, `sslconfig`).
   - **The client URL to be specified to the authorization server to receive responses** — Specify the URL of the internal destination required for a Caché OAuth 2.0 client. At this destination, the access token is saved and then the browser is further redirected back to the client application.

     To specify this URL, enter values for the following options:

     - **Host name** — Specify the host name or IP address of the authorization server.
     - **Port** — Specify this if needed to accommodate any changes in the CSP Gateway configuration.
     - **Prefix** — Specify this if needed to accommodate any changes in the CSP Gateway configuration.

     The resulting URL has the following form:

     ```
     https://hostname:port/prefix/csp/sys/oauth2/OAuth2.Response.cls
     ```

     If you omit **Port**, the colon is omitted. Similarly, if you omit **Prefix**, there is only one slash between `hostname:port` and `csp`. (Also if the **Use TLS/SSL** option is cleared, the URL starts with `http` rather than `https`.)

- **Use TLS/SSL** — Select this option, unless there is a good reason not to use TLS/SSL when opening the redirect page.
- **Required grant types** — Specify the OAuth 2.0 grant types that the client will restrict itself to using.
- **Authentication type** — Select the type of authentication (as specified in RFC 6749 or OpenID Connect Core section 9) to be used for HTTP requests to the authorization server. Select one of the following: **none**, **basic**, **form encoded body**, **client secret JWT**, or **private key JWT**.
- **Authentication signing algorithm** — Select the algorithm that must be used for signing the JWTs used to authenticate this client at the token endpoint (if the authentication type is **client secret JWT** or **private key JWT**). If you do not select an option, any algorithm supported by the OpenID provider and the relying party may be used.

5. On the **Client Information** tab, specify the following details:
   - **Logo URL** — URL of the logo for the client application.
   - **Client home page URL** — URL of the home page for the client application.
   - **Policy URL** — URL of the policy document for the client application.
   - **Terms of service URL** — URL of the terms of service document for the client application.
   - **Default scope** — Specify the default scope, as a blank separated list, for access token requests. This default should be consistent with the scopes permitted by the authorization server.
   - **Contact emails** — Comma-separated list of email addresses suitable for use in contacting those responsible for the client application.
   - **Default max age** — Specify the default maximum authentication age, in seconds. If you specify this option, the end user must be actively re-authenticated when the maximum authentication age is reached. The max_age request parameter overrides this default value. If you omit this option, there is no default maximum authentication age.
6. On the **JWT Settings** tab, specify the following details:
   - **Create JWT Settings from X509 credentials** — Select this option if, for signing and encryption, you want to use the private key associated with a certificate; in this case, also see "Using Certificates for an OAuth 2.0 Client," in the appendix "Certificates and JWTs (JSON Web Tokens)."

     Note: InterSystems expects that the option **Create JWT Settings from X509 credentials** will rarely be used, and that instead customers use the default behavior described next.

     If you leave this option clear, the system generates a pair of JWKSs (JSON web key sets). One JWKS is private and contains all the needed private keys (per algorithm) as well as the client secret for use as a symmetric key; this JWKS is never shared. The other JWKS contains the corresponding public keys and is publicly available. The dynamic registration process also copies the public JWKS to the authorization server, so that the authorization server can encrypt and verify signatures of JWTs from this client.
   - **Signing algorithm** — Select the signing algorithm used to create signed JWTs. Or leave this blank if JWTs are not to be signed.
   - **Encryption algorithm** — Select the encryption algorithm used to create encrypted JWTs. Or leave this blank if JWTs are not to be encrypted. If you select a value, you must also specify **Key algorithm**.
   - **Key algorithm** — Select the key management algorithm used to create encrypted JWTs. Or leave this blank if JWTs are not to be encrypted.
7. If the authorization server supports dynamic registration, double-check all the data you have entered and then press **Dynamic Registration and Save**. Caché then contacts the authorization server, registers the client, and obtains the client ID and client secret.

If the authorization server does not support dynamic registration, see the following subsection.

3.2.2.1 Configuring a Client (No Dynamic Registration)

If the authorization server does not support dynamic registration, then do the following instead of the last step above:

1. Select the **Client Credentials** tab and specify the following details:
   - **Client ID** — Enter the client ID as provided by the authorization server.
   - **Client secret** — Enter the client secret as provided by the authorization server. This value is required if the **Client Type** is **Confidential**.

     This chapter primarily discusses the scenario in which the client uses the authorization code grant type. For this scenario, specify a value for **Client secret**. For other grant types, see "Variations," later in this chapter.

   Do not enter values for **Client ID Issued At**, **Client Secret Expires At**, and **Registration Client Uri**.
2. Select **Save**.

3.3 Outline of Code Requirements

Note: This section describes the code needed when the client uses the authorization code grant type when requesting tokens. For other grant types, see "Variations," later in this chapter.

In order for a Caché web application to act as OAuth 2.0 client, this web application must use logic like the following:

1. Obtain an access token (and if needed, and ID token). See "Obtaining Tokens."
2. Examine the access token and (optionally, an ID token) to determine whether the user has the necessary permissions to use the requested resource. See "Examining the Tokens," later in this chapter.
3. If appropriate, call the resource server as described in "Adding an Access Token to an HTTP Request," later in this chapter.

The following sections provide information on these steps.

Note that the pages of this web application must ultimately be based on %CSP.Page; you can create them with CSP, Zen, or Zen Mojo. It is useful to remember that all these kinds of pages can execute code on the client or on the server. The Caché OAuth 2.0 client API is a set of methods that run on the server.

3.4 Obtaining Tokens

Note: This section provides information on the code needed when the client uses the authorization code grant type when requesting tokens. For other grant types, see "Variations," later in this chapter.

To obtain tokens, use steps like the following to obtain tokens. The subsection provides details on the methods discussed here.

1. Call the **IsAuthorized()** method of the %SYS.OAuth2.AccessToken class. For this, you will need to first determine the desired scope or scopes for the access token.

   For example:

ObjectScript

```
set myscopes="openid profile scope1 scope2"
set isAuth=##class(%SYS.OAuth2.AccessToken).IsAuthorized("myclient",,myscopes,
                .accessToken,.idtoken,.responseProperties,.error)
```

This method checks to see whether an access token has already been saved locally.

2. Check to see if the *error* argument has returned an error and then handle that error appropriately. Note that if this argument contains an error, the function *$ISOBJECT()* will return 1; otherwise *$ISOBJECT()* will return 0.

   ObjectScript

   ```
    if $isobject(error) {
   //error handling here
  }
   ```

3. If **IsAuthorized()** returns 1, skip to "Examining the Tokens."
4. Otherwise, call the **GetAuthorizationCodeEndpoint()** method of the %SYS.OAuth2.Authorization class. For this, you will need the following information:

   - Complete URL that the authorization server should redirect to, after it returns an access token. This is the *client's redirect page* (which can be the same as the original page, or can be different).
   - The scope or scopes of the request.
   - Any parameters to be included with the request. For example, you may need to pass the `claims` parameter.

   For example:

   ObjectScript

   ```
   set scope="openid profile scope1 scope2"
   set redirect="http://localhost/csp/openid/SampleClientResult.csp"

   set url=##class(%SYS.OAuth2.Authorization).GetAuthorizationCodeEndpoint("myclient",
         scope,redirect,.properties,.isAuthorized,.sc)
   if $$$ISERR(sc) {
     //error handling here
   }
   ```

   This method returns the full URL, including query parameters, of the internal destination required for a Caché OAuth 2.0 client.

5. Provide an option (such as a button) that opens the URL returned by **GetAuthorizationCodeEndpoint()**, thus enabling the user to authorize the request.

   At this internal URL, which is never visible to users, Caché obtains an authorization code, exchanges that for an access token, and then redirects the browser to the client's redirect page.

3.4.1 Method Details

This subsection provides the details on the methods described in the previous subsection.

IsAuthorized()

Location: This method is in the class %SYS.OAuth2.AccessToken.

```
ClassMethod IsAuthorized(applicationName As %String,
                         sessionId As %String,
                         scope As %String = "",
                         Output accessToken As %String,
                         Output IDToken As %String,
                         Output responseProperties,
                         Output error As %OAuth2.Error) As %Boolean
```

This method returns 1 if there is a locally stored access token for this client and this session, *and* if that access token authorizes all the scopes given by the *scope* argument. (Note that this method looks for the access token in the CACHESYS database, and that tokens are removed automatically after they have expired.)

Otherwise the method returns 0.

The arguments are as follows:

- *applicationName* is the name of the client application.
- *sessionId* specifies the session ID. Specify this only if you want to override the default session (*%session*.SessionId).
- *scope* is a space-delimited list of scopes, for example: `"openid profile scope1 scope2"`

  Note that `openid` and `profile` are special scopes defined by OpenID Connect Core.
- *accessToken*, which is returned as output, is the access token, if any.
- *IDToken*, which is returned as output, is the ID token, if any. (This applies only if you are using OpenId Connect, specifically if the request used the scope `openid`.) Note that an ID token is a JWT.
- *responseProperties*, which is returned as output, is a multidimensional array that contains any parameters of the response. This array has the following structure:

Array node	Array value
responseProperties(*parametername*) where *parametername* is the name of a parameter (such as `token_type` or `expires_in`)	Value of the given parameter.

- *error*, which is returned as output, is either (when there is no error) an empty string or (in the case of error) an instance of %OAuth2.Error containing error information.

  %OAuth2.Error has three string properties: Error, ErrorDescription, and ErrorUri.

GetAuthorizationCodeEndpoint()

Location: This method is in the class %SYS.OAuth2.Authorization.

```
ClassMethod GetAuthorizationCodeEndpoint(applicationName As %String,
                                         scope As %String,
                                         redirectURL As %String,
                                         ByRef properties As %String,
                                         Output isAuthorized As %Boolean,
                                         Output sc As %Status,
                                         responseMode As %String
                                         sessionId As %String = "") As %String
```

This method returns the URL, with all needed query parameters, of the local, internal page that Caché uses to request the authorization code. (Note that this page is never visible to users.)

The arguments are as follows:

- *applicationName* is the name of the client application.

- *scope* is a space-delimited list of scopes for which access is requested, for example: `"scope1 scope2 scope3"`

  The default is determined by the client configuration for the given *applicationName*.

- *redirectURL* is the full URL of the client's redirect page, the page to which the authorization server should redirect the browser after returning the access token to the client.

- *properties*, which is passed by reference, is a multidimensional array that contains any parameters to be added to the request. This array must have the following structure:

Array node	Array value
properties(*parametername*) where *parametername* is the name of a parameter	Value of the given parameter. The value can be a scalar value, an instance of a dynamic object, or the UTF-8 encoded serialized form of a dynamic object. Use a dynamic object if you want the request to include a parameter whose value is a JSON object; a scenario is the `claims` parameter that is defined by OpenID Connect. For details on dynamic objects, see *Using JSON in Caché*. To use the `request` or `request_uri` parameter, see the section "Passing Request Objects as JWTs."

- *isAuthorized*, which is returned as output, equals 1 if there is a locally stored access token for this client and this session (scope is not checked). This parameter equals 0 otherwise. There is no need to check this output argument, because we have just called the **IsAuthorized()** method.

- *sc*, which is returned as output, contains the status code set by this method.

- *responseMode* specifies the mode of the response from the authorization server. This can be `"query"` (the default), `"fragment"` or `"form_post"`. The default is almost always appropriate.

- *sessionId* specifies the session ID. Specify this only if you want to override the default session (*%session*.SessionId).

Also see "Variation: Performing the Redirect within OnPreHTTP."

3.5 Examining the Token(s)

After the client receives an access token (and, optionally, an ID token), the client should perform additional checks to determine whether the user has the necessary permissions to use the requested resource. To perform this examination, the client can use the methods described here to obtain additional information.

Then examine the dynamic objects returned by reference from these methods. For example, **GetIntrospection()** returns a dynamic object that contains the claims made by the introspection endpoint. Use these claims as needed to determine whether to proceed.

ValidateJWT()

Location: This method is in the class %SYS.OAuth2.Validation.

```
ClassMethod ValidateJWT(applicationName As %String,
                        accessToken As %String,
                        scope As %String,
                        aud As %String,
                        Output jsonObject As %RegisteredObject,
                        Output securityParameters As %String,
                        Output sc As %Status) As %Boolean
```

Use this method only if the access token is a JWT (rather than an opaque token).

This method decrypts the JWT if necessary, validates the JWT, and creates an object (*jsonObject*) to contain the JWT properties. To validate the JWT, the method checks the audience (if *aud* is specified), issuer endpoint (must match that specified in server definition), and scope (if *scope* is specified). The method also makes sure the access token has not expired. Both signed and unsigned JWTs are accepted. If the JWT is signed, the method checks the signature.

This method returns 1 if the JWT is valid or returns 0 otherwise. It also returns several arguments as output.

The arguments are as follows:

- *applicationName* is the name of the client application.
- *accessToken* is the JWT to be validated.
- *scope* is a space-delimited list of scopes, for example: `"scope1 scope2 scope3"`

  If *scope* is specified, the JWT must contain a scope claim that includes this scope.
- *aud* specifies the audience that is using the token. If the token has an associated `aud` property (usually because the audience was specified when requesting the token), then `aud` is matched to the token audience. If `aud` is not specified, then no audience checking takes place.
- *jsonObject*, which is returned as output, is a dynamic object that contains the claims in the JWT. This dynamic object contains properties such as aud, exp, iss, and so on. For details on dynamic objects, see *Using JSON in Caché*.
- *securityParameters*, which is returned as output, is a multidimensional array that contains security information taken from the header, for optional additional use in verifying signatures, decrypting, or both.

  This array contains the following nodes:

Node	Value
securityParameters`("sigalg")`	The signature or MAC algorithm. Set only if the JWT is signed
securityParameters`("keyalg")`	The key management algorithm
securityParameters`("encalg")`	The content encryption algorithm

  The `keyalg` and `encalg` nodes are either both specified or both null.
- *sc*, which is returned as output, contains the status code set by this method.

If this method returns success (1), examine *jsonObject*, and use the contained claims as needed to determine whether to allow access to the requested resource. Use *securityParameters* if needed.

Because the OAuth specification allows an application to accept both signed and unsigned JWTs, the `ValidateJWT` method does not reject an unsigned JWT. However, in many cases it is strongly recommended that your application implement stricter security by rejecting an unsigned JWT. You can determine whether the token passed into

`ValidateJWT` was unsigned by inspecting the *securityParameters* array that is returned by the method. If *securityParamters* (`"sigalg"`) was not set, the token was unsigned. For example, the following code determines whether the token was unsigned and rejects it if it was:

```
Set tInitialValidationPassed = ##class(%SYS.OAuth2.Validation).ValidateJWT(tClientName,
tAccessToken, "", "", .tJsonObj,.tSecurityParams, .tValidateStatus)
// the “sigalg” subscript is set only if the JWT was signed
Set tIsTokenSigned = $Data(tSecurityParams("sigalg"))#2
If 'tIsTokenSigned {
   $$$ThrowStatus($System.Status.Error($$$AccessDenied))
}
```

ValidateIDToken()

Location: This method is in the class %SYS.OAuth2.Validation.

```
ClassMethod ValidateIDToken(applicationName As %String,
                            IDToken As %String,
                            accessToken As %String,
                            scope As %String,
                            aud As %String,
                            Output jsonObject As %RegisteredObject,
                            Output securityParameters As %String,
                            Output sc As %Status) As %Boolean
```

This method validates the signed OpenID Connect ID token (*IDToken*) and creates an object (*jsonObject*) to contain the properties of the ID token. To validate the ID token, the method checks the audience (if *aud* is specified), endpoint (must match that specified in server definition), and scope (if *scope* is specified), and signature. The method also makes sure the ID token has not expired.

This method also validates the access token (*accessToken*) based on the `at_hash` property of the ID token.

This method returns 1 if the ID token is valid or returns 0 otherwise. It also returns several arguments as output.

The arguments are as follows:

- *applicationName* is the name of the client application.
- *IDToken* is the ID token.
- *accessToken* is the access token.
- *scope* is a space-delimited list of scopes, for example: `"scope1 scope2 scope3"`
- *aud* specifies the audience that is using the token. If the token has an associated `aud` property (usually because the audience was specified when requesting the token), then `aud` is matched to the token audience. If `aud` is not specified, then no audience checking takes place.
- *jsonObject*, which is returned as output, is a dynamic object that contains the properties of the *IDToken*. Note that an ID token is a JWT. For details on dynamic objects, see *Using JSON in Caché*.
- *securityParameters*, which is returned as output, is a multidimensional array that contains security information taken from the header, for optional additional use in verifying signatures, decrypting, or both. See the *securityParameters* argument for **ValidateJWT()**.
- *sc*, which is returned as output, contains the status code set by this method.

If this method returns success (1), examine *jsonObject*, and use the contained claims as needed to determine whether to allow access to the requested resource. Use *securityParameters* if needed.

GetIntrospection()

Location: This method is in the class %SYS.OAuth2.AccessToken.

```
ClassMethod GetIntrospection(applicationName As %String,
                             accessToken As %String,
                             Output jsonObject As %RegisteredObject) As %Status
```

This method sends the access token to the introspection endpoint, receives a response that contains claims, and creates an object (*jsonObject*) that contains the claims returned by that endpoint.

The request is authorized using the basic authorization HTTP header with the `client_id` and `client_secret` associated with *applicationName*.

The arguments are as follows:

- *applicationName* is the name of the client application.
- *accessToken* is the access token.
- *jsonObject*, which is returned as output, is a dynamic object that contains the claims returned by introspection endpoint. For details on dynamic objects, see *Using JSON in Caché*.

Note that you cannot use this method if the server does not specify an introspection endpoint or if **Client secret** is not specified.

If this method returns success (1), examine *jsonObject*, and use the contained claims as needed to determine whether to allow access to the requested resource.

GetUserinfo()

Location: This method is in the class %SYS.OAuth2.AccessToken.

```
ClassMethod GetUserinfo(applicationName As %String,
                        accessToken As %String,
                        IDTokenObject As %RegisteredObject,
                        Output jsonObject As %RegisteredObject,
                        Output securityParameters As %String) As %Status
```

This method sends the access token to the Userinfo endpoint, receives a response that contains claims, and creates an object (*jsonObject*) that contains the claims returned by that endpoint. If the response returns a JWT, then the response is decrypted and the signature is checked before *jsonObject* is created. If the argument *IDTokenObject* is specified, the method also verifies that the `sub` claim from the User info endpoint matches the `sub` claim in *IDTokenObject*.

The request is authorized using the specified access token.

The arguments are as follows:

- *applicationName* is the name of the client application.
- *accessToken* is the access token.
- *IDTokenObject* (optional), is a dynamic object containing an ID token. For details on dynamic objects, see *Using JSON in Caché*.
- *jsonObject*, which is returned as output, is a dynamic object that contains the claims returned by Userinfo endpoint.
- *securityParameters*, which is returned as output, is a multidimensional array that contains security information taken from the header, for optional additional use in verifying signatures, decrypting, or both. See the *securityParameters* argument for **ValidateJWT()**.

If this method returns success (1), examine *jsonObject*, and use the contained claims as needed to determine whether to allow access to the requested resource. Use *securityParameters* if needed.

3.6 Adding an Access Token to an HTTP Request

After the client application has received and examined an access token, the application can make HTTP requests to the resource server. Depending on the application, those HTTP requests may need the access token.

To add an access token to an HTTP request (as a bearer token HTTP authorization header), do the following:

1. Create an instance of %Net.HttpRequest and set properties as needed.

   For details on this class, see "Sending HTTP Requests" in *Using Caché Internet Utilities*.

2. Call the AddAccessToken() method of %SYS.OAuth2.AccessToken, which adds the access token to the HTTP request. This method is as follows:

   ```
   ClassMethod AddAccessToken(httpRequest As %Net.HttpRequest,
                              type As %String = "header",
                              sslConfiguration As %String,
                              applicationName As %String,
                              sessionId As %String) As %Status
   ```

   This method adds the bearer access token associated with the given application and session to the resource server request as defined by RFC 6750. The arguments are as follows:

   - *httpRequest* is the instance of %Net.HttpRequest that you want to modify.
   - *type* specifies how to include the access token in the HTTP request:
     - `"header"` — Use the bearer token HTTP header.
     - `"body"` — Use form-encoded body. In this case, the request must be a POST with a form-encoded body.
     - `"query"` — Use a query parameter.
   - *sslConfiguration* is the Caché SSL configuration to use for this HTTP request. If you omit this, Caché uses the SSL configuration associated with the client configuration.
   - *applicationName* is the name of the client application.
   - *sessionId* specifies the session ID. Specify this only if you want to override the default session (*%session*.SessionId).

   This method returns a status code, which your code should check.

3. Send the HTTP request (as described in "Sending HTTP Requests" in *Using Caché Internet Utilities*. To do so, you call a method such as **Get()** or **Put()**.
4. Check the status returned by the previous step.
5. Optionally examine the HTTP response, which is available as the HttpResponse property of the HTTP request.

   See "Sending HTTP Requests" in *Using Caché Internet Utilities*.

For example:

ObjectScript

```
set httpRequest=##class(%Net.HttpRequest).%New()
// AddAccessToken adds the current access token to the request.
set sc=##class(%SYS.OAuth2.AccessToken).AddAccessToken(httpRequest,,"sslunittest",applicationName)
if $$$ISOK(sc) {
   set sc=httpRequest.Get("https://myresourceserver/csp/openid/openid.SampleResource.cls")
}
```

3.7 Optionally Defining Delegated Authentication for the Web Client

You can optionally define delegated authentication for a Caché web client that is used as an OAuth 2.0 client. Caché provides two ways that you can do this:

- By creating and using a **ZAUTHENTICATE** routine, starting from a sample that is provided for use with OAuth 2.0. Your client code must also call *%session*.**Login()**.
- By creating and using a custom login page, starting from a superclass provided by Caché. It is also necessary to create and use a **ZAUTHENTICATE** routine (starting from the same sample that is provided for use with OAuth 2.0), but your client code does not need to call *%session*.**Login()**.

The following subsections give the details. A final subsection discusses the **ZAUTHENTICATE** sample.

3.7.1 Creating and Using a ZAUTHENTICATE Routine for an OAuth 2.0 Client

To create and use a **ZAUTHENTICATE** routine for a Caché web client that is used as an OAuth 2.0 client, do all of the following:

- In your client code, after you call the **IsAuthorized()** method of the %SYS.OAuth2.AccessToken class and successfully obtain an access token, call the **Login()** method of the *%session* variable. For the username, specify the OAuth 2.0 application name; for the password, specify the CSP session ID.
- Create the **ZAUTHENTICATE** routine. This routine must perform basic setup for a user account, such as specifying roles and other user properties.

  To create this routine, copy OAUTH2.ZAUTHENTICATE.mac from the SAMPLES namespace to the %SYS namespace and rename it to ZAUTHENTICATE.mac. Then modify this routine as needed. See "Notes about the OAUTH2.ZAUTHENTICATE.mac Sample," later in this section. For more general information, see "Creating Delegated (User-Defined) Authentication Code" in the chapter "Using Delegated Authentication" in the *Caché Security Administration Guide*.
- Enable delegated authentication for the Caché instance on the Authentication Options page.

  For information on this step and the next step, see the chapter "Using Delegated Authentication" in *Caché Security Administration Guide*.
- Enable delegated authentication for the relevant web application.

3.7.2 Creating and Using a Custom Login Page for an OAuth 2.0 Client

To create and use a custom login page for a Caché web client that is used as an OAuth 2.0 client, do all of the following:

- Create a subclass of %OAuth2.Login. In your subclass:
  - Specify the application name, scope list, and (optionally) response mode. You can specify these items in either or both of the following ways:
    - By specifying parameters of your subclass of %OAuth2.Login.
    - By overriding the **DefineParameters()** class method. In contrast to specifying parameters, this technique enables you to set these values at runtime.

    The parameters are as follows:

- *APPLICATION* — This must be the application name for the application being logged into.
- *SCOPE* — This specifies the scope list to be used for the access token request. This must be a blank-separated list of strings.
- *RESPONSEMODE* — This specifies the mode of the response. The allowed values are `"query"` (the default), `"fragment"` or `"form_post"`.

The **DefineParameters()** class method has the following signature:

```
ClassMethod DefineParameters(Output application As %String, Output scope As %String, Output
responseMode As %String)
```

This method returns the application name, scope list, and response mode as output arguments. The default implementation of this method returns the values of the *APPLICATION*, *SCOPE*, and *RESPONSEMODE* class parameters.

– In your subclass of %OAuth2.Login, also specify the properties list for the **GetAccessTokenAuthorizationCode()** call. To do so, override the **DefineProperties()** class method. This method has the following signature:

```
ClassMethod DefineProperties(Output properties As %String)
```

This method returns (as output) the *properties* array, which is a local array specifying additional properties to be included in a token request. The *properties* array has the following form:

Node	Value
properties(name), where *name* is the name of a parameter.	Value of the given parameter.

To add a request parameter that is a JSON object, create a properties element which is an instance of %DynamicObject. Or create a string that is the UTF-8 encoded serialized object.

To add the `request` or `request_uri` request parameters, use the %SYS.OAuth2.Request class to create the JWT. Then, as appropriate, set *properties*`("request")` equal to the JWT or set *properties*`("request_uri")` equal to the URL of the JWT.

- Configure the relevant web application to use the custom login page.
- Create and use a **ZAUTHENTICATE** routine as described in the previous section, except for the first bullet item. (In this scenario, there is no need for the client code to call *%session*.**Login()**.)

3.7.3 Notes about the OAUTH2.ZAUTHENTICATE.mac Sample

The OAUTH2.ZAUTHENTICATE.mac sample supports both scenarios described in the previous subsections. In this sample, the **GetCredentials()** subroutine looks like this:

```
GetCredentials(ServiceName,Namespace,Username,Password,Credentials) Public {
    If ServiceName="%Service_CSP" {
        // Supply user name and password for authentication via a subclass of %OAuth2.Login
        Set Username="OAuth2"
        Set Password=$c(1,2,3)
    }
    Quit $$$OK
}
```

This subroutine is called if no username and password are provided (which is the case when the custom login page is being used). For the service **`%Service_CSP`**, this sample sets the username and password to specific values that are also used in the **ZAUTHENTICATE()** subroutine (which is called in later processing).

The **ZAUTHENTICATE()** subroutine includes the following:

```
If Username="OAuth2",Password=$c(1,2,3) {
    // Authentication is via a subclass of %OAuth2.Login that sets the query parameter CSPOAUTH2
    // with a hash value that allows GetCurrentApplication to determine the application --
    // username/password is supplied by GetCredentials.
    Set sc=##class(OAuth2.Response).GetCurrentApplication(.applicationName)
    Set sessionId=%session.SessionId
} Else {
    // If authentication is based on %session.Login, then application and session id are passed in.
    Set applicationName=Username
    Set sessionId=Password
}
```

A later step calls the **isAuthorized()** method like this:

```
Set
isAuthorized=##class(%SYS.OAuth2.AccessToken).IsAuthorized(applicationName,sessionId,,.accessToken,,,.error)
```

If **isAuthorized()** returns 1, then later code calls the introspection endpoint and uses the information obtained there to define a user:

```
Set sc=##class(%SYS.OAuth2.AccessToken).GetIntrospection(applicationName,accessToken,.jsonObject)
...
Set Username="OAuth2"_jsonObject.sub
Set Properties("FullName")="OAuth account "_Username
Set Properties("Username")=Username
Set Properties("Password")=""    // we don't really care about oauth2 account password
// Set the roles and other Properties as appropriate.
Set Properties("Roles")=roles
```

Your code could use different logic to obtain the information needed to define the user. You could instead obtain this information in the following ways:

- From the JWT if the access token is a JWT. In this case, call the **ValidateJWT()** method of %SYS.OAuth2.Validate.
- From IDToken if you are using OpenID Connect. In this case, call the **ValidateIDToken()** of %SYS.OAuth2.Validate.
- From Userinfo endpoint if you are OpenID Connect. In this case, call the **GetUserinfo()** of %SYS.OAuth2.AccessToken.

In any case, it is necessary to define a user whose username does not match a normal Caché username.

Your routine would also need to set roles and other parts of the *Properties* array as needed for your application. See "Creating Delegated (User-Defined) Authentication Code" in the chapter "Using Delegated Authentication" in *Caché Security Administration Guide*.

3.8 Variations

This chapter primarily discusses the scenario in which a Caché web application uses the authorization code grant type. This section discusses some variations:

- Implicit grant type
- Password credentials grant type
- Client credentials grant type
- Performing the redirect within OnPreHttp (applies to authorization code and implicit grant types)
- Passing request objects as JWTs
- Calling other endpoints

In the basic scenario described earlier in this chapter, the client receives an access token from the authorization server and then optionally calls additional endpoints in the authorization server: the introspection endpoint, the Userinfo endpoint, or both. After that, the client calls the resource server. Notice that it is also possible for the resource server to independently call these endpoints.

3.8.1 Variation: Implicit Grant Type

In this variation, the client uses the implicit grant type when requesting tokens.

Configuration requirements: See the instructions in "Configuring a Client," but specify **Client Type** as appropriate for your use case.

Code requirements: The overall flow is similar to the one for the authorization code grant type, but do not call **GetAuthorizationCodeEndpoint()**. Instead call the **GetImplicitEndpoint()** method of the %SYS.OAuth2.Authorization class:

```
ClassMethod GetImplicitEndpoint(applicationName As %String,
                                scope As %String,
                                redirectURL As %String,
                                idtokenOnly As %Boolean = 0,
                                responseMode As %String,
                                ByRef properties As %String,
                                Output isAuthorized As %Boolean,
                                Output sc As %Status
                                sessionId as %String="") As %String
```

The arguments are as follows:

- *applicationName* is the name of the client application.
- *scope* is a space-delimited list of scopes for which access is requested, for example: `"openid profile scope3 scope4"`

  The default is determined by the client configuration for the given *applicationName*.
- *redirectURL* is the URL of the page to which the authorization server should redirect the browser after returning the access token to the client server.
- *idtokenOnly* enables you to obtain only an ID token. If this argument is 0, the method obtains both an access token and (if the request includes the appropriate scope) an ID token. If this argument is 1, the method does not obtain an access token.
- *responseMode* specifies the mode of the response from the authorization server. This can be `"query"` (the default), `"fragment"` or `"form_post"`.
- *properties*, which is passed by reference, is a multidimensional array that contains any parameters to be added to the request. This array must have the following structure:

Array node	Array value
properties(*parametername*) where *parametername* is the name of a parameter	Value of the given parameter. The value can be a scalar value, an instance of a dynamic object, or the UTF-8 encoded serialized form of a dynamic object. Use a dynamic object if you want the request to include a parameter whose value is a JSON object; a scenario is the `claims` parameter that is defined by OpenID Connect. For details on dynamic objects, see *Using JSON in Caché*. To use the `request` or `request_uri` parameter, see the section "Passing Request Objects as JWTs."

- *isAuthorized*, which is returned as output, equals 1 if there is a locally stored access token for this client and this session, *and* if that access token authorizes all the scopes given by the *scope* argument. This parameter equals 0 otherwise.
- *sc*, which is returned as output, contains the status code set by this method.
- *sessionId* specifies the session ID. Specify this only if you want to override the default session (*%session*.SessionId).

Also see "Variation: Performing the Redirect within OnPreHTTP."

3.8.2 Variation: Password Credentials Grant Type

In this variation, the client uses the password credentials grant type when requesting tokens. You can use this grant type when the client has the password belonging to the resource owner. The client application can simply perform an HTTP POST operation to the token endpoint, without any page redirection; Caché provides a method to do this.

Configuration requirements: See the instructions in "Configuring a Client," but note that you do not need to specify **Client Secret**. (In general, you should use the client secret only when the client secret is needed *and* it is possible to protect the client secret.)

Code requirements: Your application should do the following:

1. Call the **IsAuthorized()** method of %SYS.OAuth2.AccessToken and check the returned value (and possible error), as described in "Obtaining Tokens," earlier in this chapter.
2. If **IsAuthorized()** returned 0, call the **GetAccessTokenPassword()** method of %SYS.OAuth2.Authorization.

```
ClassMethod GetAccessTokenPassword(applicationName As %String,
                                   username As %String,
                                   password As %String,
                                   scope As %String,
                                   ByRef properties As %String,
                                   Output error As %OAuth2.Error) As %Status
```

   The arguments are as follows:

   - *applicationName* is the name of the client application.
   - *username* is a username.
   - *password* is the corresponding password.
   - *scope* is a space-delimited list of scopes for which access is requested, for example: `"scope1 scope2 scope3"`

     The default is determined by the client configuration for the given *applicationName*.
   - *properties*, which is passed by reference, is a multidimensional array that contains any parameters to be added to the request. This array must have the following structure:

Array node	Array value
properties(*parametername*) where *parametername* is the name of a parameter	Value of the given parameter. The value can be a scalar value, an instance of a dynamic object, or the UTF-8 encoded serialized form of a dynamic object. Use a dynamic object if you want the request to include a parameter whose value is a JSON object; a scenario is the `claims` parameter that is defined by OpenID Connect. For details on dynamic objects, see *Using JSON in Caché*.

   - *error*, which is returned as output, is either null or is an instance of OAuth2.Error that contains error information.

This method performs an HTTP POST operation to the token endpoint, and then receives and saves the access token (if any).

3. Check the *error* argument and proceed accordingly.
4. Continue as described in "Examining the Token(s)" and "Adding an Access Token to an HTTP Request."

3.8.3 Variation: Client Credentials Grant Type

In this variation, the client uses the client credentials grant type when requesting tokens. This grant type enables the client application to communicate with the resource server independently from any user. There is no user context. The client application can simply perform an HTTP POST operation to the token endpoint, without any page redirection; Caché provides a method to do this.

Configuration requirements: See the instructions in "Configuring a Client." Make sure to specify the **Client Type** as **Private** and specify **Client Secret**.

Code requirements: Your application should do the following:

1. Call the **IsAuthorized()** method of %SYS.OAuth2.AccessToken and check the returned value (and possible error), as described in "Obtaining Tokens," earlier in this chapter.
2. If **IsAuthorized()** returned 0, call the **GetAccessTokenClient()** method of %SYS.OAuth2.Authorization.

```
ClassMethod GetAccessTokenClient(applicationName As %String,
                                 scope As %String,
                                 ByRef properties As %String,
                                 Output error As %OAuth2.Error) As %Status
```

The arguments are as follows:

- *applicationName* is the name of the client application.
- *scope* is a space-delimited list of scopes for which access is requested, for example: `"scope1 scope2 scope3"`

  The default is determined by the client configuration for the given *applicationName*.
- *properties*, which is passed by reference, is a multidimensional array that contains any parameters to be added to the request. See the *properties* argument for **GetAccessTokenPassword()**, in the previous subsection.
- *error*, which is returned as output, is either null or is an instance of OAuth2.Error that contains error information.

This method performs an HTTP POST operation to the token endpoint, and then receives and saves the access token (if any).

3. Check the *error* argument and proceed accordingly.
4. Continue as described in "Examining the Token(s)" and "Adding an Access Token to an HTTP Request."

3.8.4 Variation: Performing the Redirect within OnPreHTTP

For the authorization code and implicit grant types, the previous instructions use the following steps:

1. Call the **IsAuthorized()** method of %SYS.OAuth2.AccessToken.
2. Call the **GetAuthorizationCodeEndpoint()** method (for the authorization code grant type) or call the **GetImplicitEndpoint()** method (for the implicit grant type).
3. Provide an option (such as a button) that opens the URL returned by the previous step, thus enabling the user to authorize the request

An alternative is to modify the **OnPreHttp()** method of the page class (in your application), so that it calls either the **GetAccessTokenAuthorizationCode()** method (for the authorization code grant type) or call the **GetAccessTokenImplicit()** method (for the implicit grant type). These methods cause the browser to navigate directly (if needed) to the authentication form of the authorization server, without first displaying any content of your page.

3.8.5 Variation: Passing Request Objects as JWTs

Caché also supports passing the request object as a JWT, as specified in section 6 of the OpenID Connect Core specification. You can pass the request object by value or by reference.

In both cases, you use methods of the %SYS.OAuth2.Request class. See the class reference for additional methods not described in this section.

3.8.5.1 Passing a Request Object by Value

To use the `request` parameter to pass the request object as a JWT:

1. Call the **MakeRequestJWT()** method of the %SYS.OAuth2.Request class:

   ```
   ClassMethod MakeRequestJWT(applicationName As %String,
                              ByRef properties As %String,
                              Output sc As %Status) As %String
   ```

   Where:

   - *applicationName* is the name of the client application.
   - *properties*, which is passed by reference, is a multidimensional array that contains any parameters to be added to the request. This array must have the following structure:

Array node	Array value
properties(*parametername*) where *parametername* is the name of a parameter	Value of the given parameter. The value can be a scalar value, an instance of a dynamic object, or the UTF-8 encoded serialized form of a dynamic object.

   - *sc*, which is returned as output, contains the status code set by this method.

   This method returns a string, which is the JWT. For example:

   ObjectScript

   ```
   // create jwt
   set jwt=##class(%SYS.OAuth2.Request).MakeRequestJWT("myapp",.properties,.sc)
   ```

2. Modify the *properties* array that you will use as the argument for **GetAuthorizationCodeEndpoint()** or **GetImplicitEndpoint()**. Set the node *properties*`("request")` equal to the JWT that you created in the previous step. For example:

   ObjectScript

   ```
   set properties("request")=jwt
   ```

3. When you call **GetAuthorizationCodeEndpoint()** or **GetImplicitEndpoint()**, include the *properties* array. For example:

ObjectScript

```
set url=##class(%SYS.OAuth2.Authorization).GetAuthorizationCodeEndpoint("myapp",
      scope,redirect,.properties,.isAuthorized,.sc, responseMode)
```

3.8.5.2 Passing a Request Object by Reference

To use the `request_uri` parameter to pass the request object as a JWT:

1. Call the **UpdateRequestObject()** method of the %SYS.OAuth2.Request class:

   ```
   ClassMethod UpdateRequestObject(applicationName As %String,
                                   requestName As %String,
                                   ByRef properties As %String,
                                   Output sc As %Status) As %SYS.OAuth2.Request
   ```

   Where:

   - *applicationName* is the name of the client application.
   - *requestName* is the name of the request.
   - *properties*, which is passed by reference, is a multidimensional array that contains any parameters to be added to the request. This array must have the following structure:

Array node	Array value
properties(*parametername*) where *parametername* is the name of a parameter	Value of the given parameter. The value can be a scalar value, an instance of a dynamic object, or the UTF-8 encoded serialized form of a dynamic object.

   - *sc*, which is returned as output, contains the status code set by this method.

   This method creates, saves, and returns an instance of %SYS.OAuth2.Request.

   #### ObjectScript

   ```
    // create requestobject
    set
   requestobject=##class(%SYS.OAuth2.Request).UpdateRequestObject("myapp","myrequest",.properties,.sc)
   ```

2. Get the URL of the saved request object. To do so, call the **GetURL()** method of the instance. Note that **GetURL()** returns a status code as output in the first argument; your code should check that.

   #### ObjectScript

   ```
    Set requesturl=requestobject.GetURL()
   ```

3. Modify the *properties* array that you will use as the argument for **GetAuthorizationCodeEndpoint()** or **GetImplicitEndpoint()**. Set the node *properties*`("request_uri")` equal to the URL obtained in the previous step. For example:

   ```
   set properties("request_uri")=requesturl
   ```

4. When you call **GetAuthorizationCodeEndpoint()** or **GetImplicitEndpoint()**, include the *properties* array. For example:

   ```
   set url=##class(%SYS.OAuth2.Authorization).GetAuthorizationCodeEndpoint("myapp",
         scope,redirect,.properties,.isAuthorized,.sc, responseMode)
   ```

3.8.6 Variation: Calling Other Endpoints of the Authorization Server

The methods in %SYS.OAuth2.Authorization enable you to call a specific set of endpoints in the authorization server. If the authorization server has other endpoints, use the following general process to call them:

1. Create an instance of %Net.HttpRequest, set its properties as needed, and call methods as needed, in order to define the request.

   ```
   Set httpRequest=##class(%Net.HttpRequest).%New()
   Set httpRequest.ContentType="application/x-www-form-urlencoded"
   ...
   ```

   For details on this class, see "Sending HTTP Requests" in *Using Caché Internet Utilities*.

2. To add authentication to the request, call the **AddAuthentication()** method of %SYS.OAuth2.AccessToken.

   ```
   ClassMethod AddAuthentication(applicationName As %String, httpRequest As %Net.HttpRequest) As
   %Status
   ```

   Where:

   - *applicationName* is the name of the OAuth 2.0 client.
   - *httpRequest* is the instance of %Net.HttpRequest

   Caché looks up the given client and uses its **Authentication type**, **SSL configuration**, and other information to add the appropriate authentication to the request.

3. Optionally open the client configuration so that you can use properties contained in it. To do so, switch to the %SYS namespace and call the **Open()** method of OAuth2.Client, passing the client name as the argument:

   ObjectScript

   ```
   New $NAMESPACE
   set $NAMESPACE="%SYS"
   Set client=##class(OAuth2.Client).Open(applicationName,.sc)
   If client="" Quit
   ```

4. Call the **Post()**, **Get()**, or **Put()** method (as appropriate) of the HTTP request object, providing the authorization server's token endpoint as the argument. For example:

   ObjectScript

   ```
   set sc=httpRequest.Post(client.ServerDefinition.TokenEndpoint)
   ```

5. Perform additional processing as needed.

3.9 Revoking Access Tokens

If the authorization server supports token revocation, you can revoke access tokens via the Management Portal or programmatically.

3.9.1 Revoking a User's Access Tokens

To revoke all the access tokens for a given user, do the following:

1. Select **System Administration** > **Security** > **OAuth 2.0** > **Administration**.

2. Type the user ID into the field **Revoke tokens for user**.
3. Select **Revoke**.

To perform this task, you must be logged in as a user who has USE permission on the %Admin_Secure resource.

3.9.2 Revoking Access Tokens Programmatically

If it is necessary for the client to revoke an access token, use the **RevokeToken()** method of %SYS.OAuth2.AccessToken. Note that when the session holding a given token is deleted, the system automatically calls this method (if a revocation endpoint is specified).

```
ClassMethod RevokeToken(applicationName As %String, accessToken As %String) As %Status
```

The arguments are as follows:

- *applicationName* is the name of the client application.
- *accessToken* is the access token.

The request is authorized using the basic authorization HTTP header with the `client_id` and `client_secret` associated with *applicationName*.

For example:

```
set sc=##class(%SYS.OAuth2.AccessToken).RevokeToken("myclient",accessToken)
if $$$ISERR(sc) {
    //error handling here
}
```

Note that you cannot use this method if the server does not specify a revocation endpoint or if **Client secret** is not specified.

%SYS.OAuth2.AccessToken also provides the method **RemoveAccessToken()**, which removes the access token from the client but does not remove the token from the server.

3.10 Rotating Keys Used for JWTs

In most cases, you can cause the client to generate new public/private key pairs; this applies only to the RSA keys used for the asymmetric RS256, RS384, and RS512 algorithms. (The exception is if you specify **Source other than dynamic registration** as **X509 certificate**. In this case, it is not possible to generate new keys.)

Generating new public/private key pairs is known as *key rotation*; this process adds new private RSA keys and associated public RSA keys to the private and public JWKSs.

When you perform key rotation on the client, the client uses the new private RSA keys to sign JWTs to be sent to the authorization server. Similarly, the client uses the new public RSA keys to encrypt JWTs to be sent to the authorization server. To decrypt JWTs received from the authorization server, the client uses the new RSA keys, and if that fails, uses the old RSA keys; thus the client can decrypt a JWT that was created using its old public RSA keys. Last, if the client cannot verify a signed JWT received from the authorization server, then if the client has the URL for the authorization server public JWKS, the client obtains a new public JWKS and tries again to verify the signature. (Note that the client has a URL for the authorization server public JWKS if the client was registered dynamically or if the configuration specified the **JWKS from URL** option; otherwise, the client does not have this URL.)

To rotate keys for a given client configuration:

1. In the Management Portal, select **System Administration** > **Security** > **OAuth 2.0** > **Client Configuration**.

2. Select the server description with which the client configuration is associated.

   The system then displays all client configurations associated with that server description.

3. Select the configuration of the client whose keys you want to rotate.
4. Select the **Rotate Keys** button.

Note: The symmetric HS256, HS384, and HS512 algorithms always use the client secret as the symmetric key.

3.10.1 API for Key Rotation on the Client

To rotate keys programmatically on the client, call the **RotateKeys()** method of OAuth2.Client.

To obtain a new authorization server public JWKS, call the **UpdateJWKS()** method of OAuth2.ServerDefinition.

For details on these methods, see the class reference.

3.11 Getting a New Public JWKS from the Authorization Server

In most cases, the authorization server generates a public/private pair of JWKSs. There are different ways in which the client can receive the public JWKS. One way is for the authorization server to provide the public JWKS at a URL; see the **JWKS from URL** option in "Manually Creating a Server Description (No Discovery)."

If the authorization server was defined with **JWKS from URL** and if the authorization server generates a new pair of JWKSs, you can cause the client to obtain the new public JWKS from the same URL. To do so:

1. In the Management Portal, select **System Administration** > **Security** > **OAuth 2.0** > **Client Configuration**.
2. Select the server description with which the client configuration is associated.

   The system then displays all client configurations associated with that server description.

3. Select the configuration of the client.
4. Select the **Update JWKS** button.

If the authorization server was not defined with **JWKS from URL** and if the authorization server generates a new pair of JWKSs, it is necessary to obtain the public JWKS, send it to the client, and load it from a file.

4

Using a Caché Web Application as an OAuth 2.0 Resource Server

This chapter describes how to use a Caché web application as a resource server that uses the OAuth 2.0 framework. It discusses the following:

- Prerequisites
- Configuration requirements
- Code requirements
- Variations

This chapter primarily discusses the scenario in which the resource server uses the introspection endpoint of the authorization server. See the last section for details on variations.

See the previous chapter for information on rotating keys used for signing, encryption, signature verification, and decryption of JWTs.

4.1 Prerequisites for the Caché Resource Server

Before starting the tasks described in this chapter, make sure the following items are available:

- An OAuth 2 authorization server.
- If the resource server uses any endpoint of the authorization server, the resource server may be registered as a client of the OAuth 2.0 authorization server. The details depend upon the implementation of the authorization server.

  In this case, you will also later need to know specific details about this server:

  - Location of the authorization server (issuer endpoint)
  - Location of the token endpoint
  - Location of the Userinfo endpoint (if supported; see OpenID Connect Core)
  - Location of the token introspection endpoint (if supported; see RFC 7662)
  - Location of the token revocation endpoint (if supported; see RFC 7009)
  - Whether the authorization server supports dynamic registration

- If the authorization server does not support dynamic registration, you will need the client ID and client secret for the resource server. The authorization server generates these two pieces of information (on a one-time basis) and you need to get them securely to the resource server machine.

4.2 Configuration Requirements

See "Configuration Requirements" in the previous chapter, with the following changes when you create the client configuration:

- For **Application name**, specify the application name of the resource server.
- For **Client Type**, specify **Resource Server**.

  Note that when you specify **Resource Server** as the type, the configuration page displays only the options that are applicable to a resource server.
- For **clientID**, use the client ID of the resource server.
- For **clientSecret**, use the client secret of the resource server.

4.3 Code Requirements

An OAuth 2.0 resource server receives a request, examines the access token that it contains, and (depending on the access token) returns the requested information.

To create a Caché resource server, create a CSP page in the namespace used by the resource server's web application. (Because this page needs only to define a callback method and the page is never visible to users, there is no reason to use Zen or Zen Mojo.) Specifically, InterSystems suggests creating a subclass of %CSP.REST.

In this CSP page, implement the **OnPage()** method. (Or if you have created a subclass of %CSP.REST, create a URL map and the corresponding methods.)

In the **OnPage()** method (or in the applicable REST method), do the following:

1. Call the method **GetAccessTokenFromRequest()** of %SYS.OAuth2.AccessToken. This method is as follows:

   ```
   ClassMethod GetAccessTokenFromRequest(Output sc As %Status) As %String
   ```

   The method returns the access token, if any, found in the HTTP request received by this page. It uses one of the three RFC 6750 formats. The parameter *sc*, returned as output, is a status code that indicates whether an error was detected. If the request did not use SSL/TLS, that is an error condition. Also, if the request did not include a valid bearer header, that is an error condition.
2. Check to see whether the status code is an error.

   If the status is an error, the method should return a suitable error (and not return the requested information).
3. If the status code is not an error, validate the access token. To do so, use **ValidateJWT()** or your own custom method. See "Method Details," in the previous chapter.

4. Optionally call the **GetIntrospection()** method for additional information. This method calls the introspection endpoint of the authorization server and obtains claims about the access token. This method is as follows:

```
ClassMethod GetIntrospection(applicationName As %String,
                             accessToken As %String,
                             Output jsonObject As %RegisteredObject) As %Status
```

   The arguments are as follows:

   - *applicationName* is the name of the client application.
   - *accessToken* is the access token previously returned.
   - *jsonObject*, which is returned as output, is a JSON object that contains the claims that the authorization server makes about this access token.

5. If the preceding steps indicate that the user's request for information should be granted, perform the requested processing and return the requested information.

For example:

```
// This is a dummy resource server which just gets the access token from the request
// and uses the introspection endpoint to ensure that the access token is valid.
// Normally the response would not be security related, but would ocntain some interesting
// data based on the request parameters.
set accessToken=##class(%SYS.OAuth2.AccessToken).GetAccessTokenFromRequest(.sc)
if $$$ISOK(sc) {
   set sc=##class(%SYS.OAuth2.AccessToken).GetIntrospection("demo resource",accessToken,.jsonObject)

    if $$$ISOK(sc) {
        write "OAuth 2.0 access token used to authorize resource server (RFC 6749)<br>"
        write "Access token validated using introspection endpoint (RFC 7662)<br>"
        write "   scope='"_jsonObject.scope_"'<br>"
        write "   user='"_jsonObject.username_"'",!
    } else {
        write "Introspection Error="_..EscapeHTML($system.Status.GetErrorText(sc)),!
    }
} else {
    write "Error Getting Access Token="_$system.Status.GetErrorText(sc),!
}

Quit $$$OK
```

4.4 Variations

This chapter primarily discusses the scenario in which the Caché resource server uses the introspection endpoint of the authorization server. This section discusses some possible variations.

4.4.1 Variation: Resource Server Calls Userinfo Endpoint

The resource server can also call the Userinfo endpoint. To do so, the resource server code must use the **GetUserinfo()** method, discussed in the previous chapter.

4.4.2 Variation: Resource Server Does Not Call Endpoints

If the Caché resource server does not use any endpoints of the authorization server, it is not necessary to create an OAuth 2.0 configuration on this machine.

Also, the resource server does not need to use **GetAccessTokenFromRequest()**. Instead, it can get the access token directly from the HTTP authorization header and use it as needed.

5

Using Caché as an OAuth 2.0 Authorization Server

This chapter describes how to use a Caché instance as an OAuth 2.0 authorization server. It discusses the following:

- Configuration requirements for the authorization server
- Code customization options and overall flow
- Implementing the custom methods
- Details for the %OAuth2.Server.Properties object
- Locations of the endpoints
- Creating client definitions
- Rotating authorization server keys
- Getting a new public JWKS from a client

Note that this chapter is organized differently from the other chapters in this book. It is likely that the person who creates client definitions will not be the same person who set up the server. Moreover, it may be necessary to create client definitions on an ongoing basis. For this reason, the task of creating client definitions is included as a stand-alone section, at the end of the chapter.

5.1 Configuration Requirements for the Caché Authorization Server

To use a Caché instance as an OAuth 2.0 authorization server, perform the following configuration tasks:

- For the web server that is serving Caché, configure that web server to use SSL. It is beyond the scope of this documentation to describe how to configure a web server to use SSL.
- Create a Caché SSL configuration for use by the server.

  This should be a client SSL configuration; no certificate is needed. The configuration is used to connect to a web server. Via this connection, the authorization server accesses the request object specified by the `request_uri` parameter. Via this connection, the authorization server also accesses the `jwks_uri` when updating a client JWKS. If the client

does not send requests using the `request_uri` parameter and if the authorization server does not update the client JWKSs via the `jwks_uri` parameter, then the authorization server does not need an SSL configuration.

For details on creating SSL configurations, see the chapter "Using SSL/TLS with Caché" in the *Caché Security Administration Guide*.

Each SSL configuration has a unique name. For reference, the documentation refers to this one as `sslconfig`, but you can use any unique name.

- Create the server configuration as described in the following subsection.
- Later, create client definitions as needed; see the last section in this chapter.

5.1.1 Configuring the Authorization Server

In order to perform this task, you must be logged in as a user who has USE permission on the %Admin_Secure resource.

1. In the Management Portal, select **System Administration** > **Security** > **OAuth 2.0** > **Server Configuration**.
2. On the **General** tab, specify the following details:
   - **Description** — Enter an optional description.
   - **Issuer endpoint** — Specify the endpoint URL of the authorization server. To specify this URL, enter values for the following options:
     - **Host name** — Specify the host name or IP address of the authorization server.
     - **Port** — Specify this if needed to accommodate any changes in the CSP Gateway configuration.
     - **Prefix** — Specify this if needed to accommodate any changes in the CSP Gateway configuration.

     The resulting issuer endpoint has the following form:

     ```
     https://hostname:port/prefix/oauth2
     ```

     If you omit **Port**, the colon is omitted. Similarly, if you omit **Prefix**, there is only one slash between `hostname:port` and `oauth2`.
   - **Audience required** — Specify whether the authorization server requires the `aud` parameter in authorization code and implicit requests. If this check box is clear, `aud` is not required.
   - **Support user session** — Specify whether the authorization server supports user sessions. A Caché authorization server can support user sessions (note that these are not CSP sessions). To do this, Caché uses a session maintenance class (**Session maintenance class**); a default is provided. See "Code Customization Options," later in this chapter. If this check box is clear, sessions are not supported.
   - **Return refresh token** — Specify the conditions under which a refresh token is returned along with the access token. Select the option appropriate for your business case.
   - **Supported grant types** — Specify the grant types that this authorization server allows to be used to create an access token. Select at least one.
   - **OpenID provider documentation** — Specify URLs provided by the OpenID provider as follows:
     - **Service Documentation URL** — URL of a web page that provides human-readable information that developers might want or need to know when using the OpenID provider.
     - **Policy URL** — URL of a web page that describes the OpenID provider's policy on how a relying party can use the data provided by the provider.
     - **Terms of Service URL** — URL of a web page that describes the OpenID provider's terms of service.

- **SSL configuration** — Select the SSL configuration you created for use by the authorization server (for example, `sslconfig`).

3. On the **Scopes** tab, specify the following details:
   - Table with **Scope** and **Description** columns — Specify all scopes supported by this authorization server.
   - **Allow unsupported scope** — Specify whether the authorization server ignores scope values that it does not support. If this check box is clear, the authorization server returns an error when a request contains an unsupported scope value; in this case the request is not processed. If this check box is selected, the authorization server ignores the scope and processes the request.
   - **Default scope** — Specify the default for access token scope if scope is not specified in the access token request or in the client configuration.
4. On the **Intervals** tab, specify the following details:
   - **Access token interval** — Specify the number of seconds after which an access token issued by this server will expire. The default is 3600 seconds.
   - **Authorization code interval** — Specify the number of seconds after which an authorization code issued by this server will expire. The default is 60 seconds.
   - **Refresh token interval** — Specify the number of seconds after which a refresh token issued by this server will expire. The default is 24 hours (86400 seconds).
   - **Session termination interval** — Specify the number of seconds after which a user session will be automatically terminated. The value 0 means the session will not be automatically terminated. The default is 24 hours (86400 seconds).
   - **Client secret expiration interval** — Specify the number of seconds after which a client secret issued by this server will expire. The default value (0) means that the client secrets do not expire.
5. For the **JWT Settings** tab, specify the following details:
   - **Create JWT Settings from X509 credentials** — Select this option if, for signing and encryption, you want to use the private key associated with a certificate; in this case, also see "Using Certificates for an OAuth 2.0 Authorization Server," in the appendix "Certificates and JWTs (JSON Web Tokens)."

     Note: InterSystems expects that the option **Create JWT Settings from X509 credentials** will rarely be used, and that instead customers use the default behavior described next.

     If you leave this option clear, the system generates a pair of JWKSs (JSON web key sets). One JWKS is private and contains all the needed private keys (per algorithm) as well as the client secret for use as a symmetric key; this JWKS is never shared. The other JWKS contains the corresponding public keys and is publicly available. Caché also copies the public JWKS to the client, so that the client can encrypt and verify signatures of JWTs from the authorization server.
   - **Signing algorithm** — Select the algorithm to use when signing JWTs. Or leave this blank if JWTs are not to be signed.
   - **Key management algorithm** — Select the algorithm to use for key management when encrypting JWTs.

     Do this only if you select a content encryption algorithm.
   - **Content encryption algorithm** — Select the algorithm to use when encrypting JWTs. Or leave this blank if JWTs are not to be encrypted. If you select an algorithm, you must also select an algorithm for key management.
6. On the **Customization** tab, specify details as described in "Code Customization Options," later in this chapter.

7. Select **Save**.

When you save this configuration, the system creates a web application (`/oauth2`) for use by the authorization server. Do not modify this web application.

5.2 Code Customization Options and Overall Flow

This section describes the items in the **Customization Options** section of the configuration options of the authorization server. Subsections describe the overall flow and the default classes.

- **Authenticate class** — Use %OAuth2.Server.Authenticate (the default) or a custom subclass of that class.

  If you define a custom subclass, implement some or all of the following methods, depending on your needs:

  – **BeforeAuthenticate()** — Optionally implement this method to perform custom processing before authentication.

  – **DisplayLogin()** — Optionally implement this method to display a login page to identify the user. (For a less common alternative, see the appendix "Implementing DirectLogin().")

  – **DisplayPermissions()** — Optionally implement this method to display the requested permissions to the user.

  – **AfterAuthenticate()** — Optionally implement this method to perform custom processing after authentication.

- **Validate user class** — Use %OAuth2.Server.Validate (the default) or a custom class that defines the following methods:

  – **ValidateUser()** (used by all grant types other than client credentials)

  – **ValidateClient()** (used by the client credentials grant type)

  InterSystems highly recommends that you define and use a custom class. The %OAuth2.Server.Validate class is provided for demonstration purposes and is very unlikely to be suitable for production use.

- **Session maintenance class** — Use OAuth2.Server.Session (the default) or a custom subclass of that class. The default session class maintains user sessions via an HTTP-only cookie.

- **Generate token class** — Use %OAuth2.Server.Generate (the default), %OAuth2.Server.JWT, or a custom class that defines the method **GenerateAccessToken()**. If you create a custom class, you might find it useful to subclass one of the classes listed here, because they provide methods you may want to use.

- **Customization namespace** — Specify the namespace in which the customization code should run.

- **Customization roles** — Specify the role or roles to use when running the customization code.

If you use any custom subclasses, see "Implementing the Custom Methods."

5.2.1 How a Caché Authorization Server Processes Requests

This section describes what a Caché authorization server does when it receives an authorization code request or an implicit request for a token.

1. Calls the **BeforeAuthenticate()** method of the class specified via the **Authenticate class** option. The purpose of this method is to make any modifications to the request before user identification starts.

   In the default class, this method is a stub.

2. Next, if the grant type is authorization code or implicit grant, Caché does the following:

a. Calls the **DisplayLogin()** of the class specified via the **Authenticate class** option. (But also see the appendix "Implementing DirectLogin().")

   In the default class, **DisplayLogin()** displays a simple HTML login page.

b. If the username is not null, calls the **ValidateUser()** method of the class specified via the **Validate user class** option. The purpose of this method is to validate the user and (by modifying the *properties* array) to prepare any claims to be returned by the token, Userinfo, and token introspection endpoints.

   In the default class, this method is only a sample and is very unlikely to be suitable for production use.

c. If the user is validated, calls the **DisplayPermissions()** method of the class specified via the **Authenticate class** option. The purpose of this method is to display a page to the user that lists the requested permissions.

   In the default class, this method displays a simple HTML page with the permissions.

Or if the grant type is password credentials, Caché just calls the **ValidateUser()** method of the class specified via the **Validate user class** option.

Or if the grant type is client credentials, Caché just calls the **ValidateClient()** method of the class specified via the **Validate user class** option.

3. If the user accepts the permissions, calls the **AfterAuthenticate()** method of the class specified via the **Authenticate class** option. The purpose of this method is to perform any custom processing before generating an access token.

   In the default class, this method is a stub.

4. Calls the **GenerateAccessToken()** method of the class specified via the **Generate token class** option. The purpose of this method is to generate an access token to return to the user.

   In the default class (%OAuth2.Server.Generate), this method generates an access token that is an opaque string. Caché also provides an alternative class (%OAuth2.Server.JWT), in which **GenerateAccessToken()** generates an access token that is a JWT.

5.2.2 Default Classes

This section describes the default classes in a Caché authorization server, as well as the class %OAuth2.Server.JWT, which is provided as another option for the **Generate token class**.

5.2.2.1 %OAuth2.Server.Authenticate (Default for Authenticate Class)

The class %OAuth2.Server.Authenticate defines the following methods, listed in the order in which they are called:

- **BeforeAuthenticate()** is a stub. It simply quits with an OK status.
- **DisplayLogin()** writes the HTML that creates a simple login page with **Login** and **Cancel** buttons.
- **DisplayPermissions()** writes the HTML that creates a simple page that displays the requested permissions. This page includes the buttons **Accept** and **Cancel**.
- **AfterAuthenticate()** is a stub. It simply quits with an OK status.

5.2.2.2 %OAuth2.Server.Validate (Default for Validate User Class)

The %OAuth2.Server.Validate class is the default class for the **Validate user class** option.

Note: This class is provided for sample purposes and is very unlikely to be suitable for production use. That is, InterSystems expects that customers will replace or subclass this class for their own needs.

This class defines the following sample methods:

- **ValidateUser()** does the following:
  1. Looks for the given user in the CACHESYS database.
  2. Verifies the password for the user.
  3. Gets a multidimensional array that contains information about the user.
  4. Uses this array to add additional claims to the *properties* object.
- **SupportedClaims()** returns a $LIST of claims that are supported by this authorization server. By default, this method specifically returns the list of claims defined by OpenID Connect Core.
- **ValidateClient()** (used by the client credentials grant type) accepts all clients and adds no properties.

You can override all these methods in your subclass.

5.2.2.3 OAuth2.Server.Session (Default for Session Class)

The %OAuth2.Server.Session class is the default class for the **Session maintenance class** option. This class maintains sessions via an HTTP-only cookie.

In this class, the **GetUser()** method tries to access the current session. If there is a session, the method obtains the username from that session and returns that. If there is no session, the method returns the username as an empty string and also returns an error status as output.

For additional information on this class, see the class reference.

5.2.2.4 %OAuth2.Server.Generate (Default for Generate Token Class)

The %OAuth2.Server.Generate class is the default class for the **Generate token class** option. This class defines the following methods:

- **GenerateAccessToken()** generates a random string as the opaque access token.
- **IsJWT()** returns 0.

5.2.2.5 %OAuth2.Server.JWT (Another Option for Generate Access Token Class)

The %OAuth2.Server.JWT class is another class you can use (or subclass) for the **Generate token class** option. This class defines the following methods:

- **GenerateAccessToken()** returns a JWT. Before returning the JWT, Caché signs it, encrypts it, or both, according to the **JSON Web Token (JWT) Settings** in the authorization server configuration.
- **IsJWT()** returns 1.
- **CreateJWT()** creates a JWT based on a JSON object containing the claims; signs and encrypts the JWT as specified in the authorization server configuration. This method follows the specifications for OAuth 2.0 and OpenID Connect usage and should not be overridden in a subclass.
- **AddClaims()** — Adds the requested claims to the JWT. This method is as follows:

  ```
  ClassMethod AddClaims(claims As %ArrayOfObjects,
                        properties As %OAuth2.Server.Properties,
                        json As %DynamicObject)
  ```

  Where:

  – *claims* is an array of %OAuth2.Server.Claim instances.

- *properties* is an instance of %OAuth2.Server.Properties that contains properties and claims that are used by the authorization server.
- *json* is a dynamic object that represents the JWT. The method modifies this object.

5.3 Implementing the Custom Methods for the Caché Authorization Server

To customize the behavior of the authorization server, define classes as described in "Code Customization Options." Then use this section for information on defining methods in those classes, depending on the processing steps that you want to customize.

1. Optional custom processing before authentication
2. Identifying the user
3. Validating the user and specifying claims
4. Optionally displaying permissions to the user
5. Optional custom processing after authentication
6. Generating the access token

After these subsections, a final subsection describes how to validate the client, in the case when this server must support the client credentials grant type. The client credentials grant type does not use steps 2 – 4 of the preceding list.

5.3.1 Optional Custom Processing Before Authentication

The information here applies to all grant types.

To perform custom processing before authenticating the user, implement the **BeforeAuthenticate()** method of the Authenticate class. This method has the following signature:

```
ClassMethod BeforeAuthenticate(scope As %ArrayOfDataTypes,
                               properties As %OAuth2.Server.Properties) As %Status
```

Where:

- *scope* is an instance of %ArrayOfDataTypes that contains the scopes contained in the original client request. The array keys are the scope values and the array values are the corresponding display forms of those values.
- *properties* is an instance of %OAuth2.Server.Properties that contains properties and claims that are used by the authorization server. See "Details for the %OAuth2.Server.Properties Object."

In your method, optionally modify either or both of these arguments, both of which are later passed to the methods used to identify the user. The method must return a %Status.

Normally, there is no need to implement this method. However, one use case is to implement the `launch` and `launch/patient` scopes used by FHIR®, where the scope needs to be adjusted to include a specific patient.

5.3.2 Identifying the User

The information here applies only to the authorization code and implicit grant types.

To identify the user, implement the **DisplayLogin()** method of the Authenticate class. The **DisplayLogin()** method has the following signature:

```
ClassMethod DisplayLogin(authorizationCode As %String,
                         scope As %ArrayOfDataTypes,
                         properties As %OAuth2.Server.Properties,
                         loginCount As %Integer = 1) As %Status
```

Where:

- *authorizationCode*
- *scope* is an instance of %ArrayOfDataTypes that contains the scopes contained in the original client request, possibly modified by the **BeforeAuthenticate()** method. The array keys are the scope values and the array values are the corresponding display forms of the scope values.
- *properties* is an instance of %OAuth2.Server.Properties that contains properties and claims received by the authorization server and modified by methods earlier in the processing. See "Details for the %OAuth2.Server.Properties Object."
- *loginCount* is the integer count of which login attempt is taking place.

This method is responsible for writing the HTML to display the user login form. The login form must contain a Username field, a Password field, and an AuthorizationCode field (which should be hidden). The default **DisplayLogin()** method uses of the **InsertHiddenField()** method of %CSP.Page to add the AuthorizationCode hidden field.

Typically, the form also has buttons with the values `Login` and `Cancel`. These buttons should submit the form. If the user submits the form with the `Login` button, the method will accept the username and password. If the user submits the form with the `Cancel` button, the authorization process will terminate with an error return of `access_denied`.

In your implementation, you might choose to display permissions on the same page. In that case, your method would display the scopes and would use a button named `Accept` to submit the page.

The method must return a %Status.

5.3.2.1 Updating properties.CustomProperties

If the form contains elements with names that start `p_`, such elements receive special handling. After the **DisplayLogin()** method returns, Caché adds values of those elements to the *properties*.CustomProperties array, first removing the `p_` prefix from the names. For example, if the form contains an element named `p_addme`, then Caché adds `addme` (and the value of the `p_addme` element) to the *properties*.CustomProperties array.

Your method can also directly set other properties of *properties* as needed.

5.3.3 Validating the User and Specifying Claims

The information here applies to all grant types other than the client credentials grant type. (For that grant type, see "Validating the Client.")

To validate the user and specify any claims to be returned by the token, Userinfo, and token introspection endpoints, define the **ValidateUser()** method of the Validate user class. This method has the following signature:

```
ClassMethod ValidateUser(username As %String,
                         password As %String,
                         scope As %ArrayOfDataTypes,
                         properties As %OAuth2.Server.Properties,
                         Output sc As %Status) As %Boolean
```

Where:

- *username* is the username provided by the user.

- *password* is the password provided by the user. Note that if the user has already logged in, Caché calls this method with *password* as an empty string. This means that your method should detect when *password* is an empty string and not attempt to check the password in that case.
- *scope* is an instance of %ArrayOfDataTypes that contains the scopes contained in the original client request, possibly modified by the **BeforeAuthenticate()** method. The array keys are the scope values and the array values are the corresponding display forms of the scope values.
- *properties* is an instance of %OAuth2.Server.Properties that contains properties and claims received by the authorization server and modified by methods earlier in the processing. See "Details for the %OAuth2.Server.Properties Object."
- *sc* is the status code set by this method. Use this to communicate details of any errors.

Your method should do the following:

- Make sure that the password applies to the given username.
- Use the *scope* and *properties* arguments as needed for your business needs.
- Modify the *properties* object to specify any claim values, as needed, or to add new claims. For example:

```
// Setup claims for profile and email OpenID Connect scopes.
Do properties.SetClaimValue("sub",username)
Do properties.SetClaimValue("preferred_username",username)
Do properties.SetClaimValue("email",email)
Do properties.SetClaimValue("email_verified",0,"boolean")
Do properties.SetClaimValue("name",fullname)
```

- In the case of any errors, set the *sc* variable.
- Return 1 if the user is considered valid; return 0 in all other cases.

Note that after the return from **ValidateUser()**, the authorization server automatically sets the following values in the *properties* object, if these values are missing:

- In *properties*.ClaimValues:
  - iss — URL of authorization server
  - sub — `client_id`
  - exp — expiration time in seconds since December 31, 1840
- In *properties*.CustomProperties:
  - client_id — `client_id` of the requesting client

5.3.4 Displaying Permissions

The information here applies only to the authorization code and implicit grant types.

To display permissions after validating the user, implement the **DisplayPermissions()** method of the Authenticate class. This method has the following signature:

```
ClassMethod DisplayPermissions(authorizationCode As %String,
                               scopeArray As %ArrayOfDataTypes,
                               currentScopeArray As %ArrayOfDataTypes,
                               properties As %OAuth2.Server.Properties) As %Status
```

Where:

- *authorizationCode* is the authorization code.

- *scopeArray* represents the newly requested scopes, for which the user has not yet granted permission. This argument is an instance of %ArrayOfDataTypes.

  The array keys are the scope values and the array values are the corresponding display forms of the scope values.

- *currentScopeArray* represents the scopes for which the user has previously granted permission. This argument is an instance of %ArrayOfDataTypes.

  The array keys are the scope values and the array values are the corresponding display forms of the scope values.

- *properties* is an instance of %OAuth2.Server.Properties that contains properties and claims received by the authorization server and modified by methods earlier in the processing. See "Details for the %OAuth2.Server.Properties Object."

This form must have buttons with the values `Accept` and `Cancel`. These buttons should submit the form. If the user submits the form with the `Accept` button, the method should continue with authorization. If the user submits the form with the `Cancel` button, the authorization process should terminate.

5.3.5 Optional Custom Processing After Authentication

The information here applies to all grant types.

To perform custom processing after authentication, implement the **AfterAuthenticate()** method of the Authenticate class. This method has the following signature:

```
ClassMethod AfterAuthenticate(scope As %ArrayOfDataTypes, properties As %OAuth2.Server.Properties) As
 %Status
```

Where:

- *scope* is an instance of %ArrayOfDataTypes that contains the scopes as set by the authorization request and all processing before this method was called. The array keys are the scope values and the array values are the corresponding display forms of the scope values.
- *properties* is an instance of %OAuth2.Server.Properties that contains properties and claims as set by the authorization request and all processing before this method was called. See "Details for the %OAuth2.Server.Properties Object."

In your method, optionally modify either or both of these arguments. In particular, you may want to may add properties to the authentication HTTP response; to do so add properties to *properties*.ResponseProperties.

Normally, there is no need to implement this method. However, one use case is to implement the launch and launch/patient scopes used by FHIR®, where it is necessary to adjust the scope to include a specific patient.

5.3.6 Generating the Access Token

The information here applies to all grant types.

To generate access tokens, implement the **GenerateAccessToken()** method of the Generate token class. This method has the following signature:

```
ClassMethod GenerateAccessToken(properties As %OAuth2.Server.Properties, Output sc As %Status) As
%String
```

Where:

- *properties* is an instance of %OAuth2.Server.Properties that contains properties and claims received by the authorization server and modified by methods earlier in the processing. See "Details for the %OAuth2.Server.Properties Object."
- *sc*, which is returned as output, is the status code set by this method. Set this variable to communicate details of any errors.

The method should return the access token. The access token may be based on the *properties* argument. In your method, you might also want to add claims to the JSON response object. To do so, set the ResponseProperties array property of the *properties* object.

5.3.7 Validating the Client

The information here applies only to the client credentials type.

To validate the client credentials and specify any claims to be returned by the token, Userinfo, and token introspection endpoints, define the **ValidateClient()** method of the Validate user class. This method has the following signature:

```
ClassMethod ValidateClient(clientId As %String,
                           clientSecret As %String,
                           scope As %ArrayOfDataTypes,
                           Output properties As %OAuth2.Server.Properties,
                           Output sc As %Status) As %Boolean
```

Where:

- *clientId* is the client ID.
- *clientSecret* is the client secret.
- *scope* is an instance of %ArrayOfDataTypes that contains the scopes contained in the original client request, possibly modified by the **BeforeAuthenticate()** method. The array keys are the scope values and the array values are the corresponding display forms of the scope values.
- *properties* is an instance of %OAuth2.Server.Properties that contains properties and claims received by the authorization server and modified by methods earlier in the processing. See "Details for the %OAuth2.Server.Properties Object."
- *sc* is the status code set by this method. Use this to communicate details of any errors.

Your method should do the following:

- Make sure that the client secret applies to the given client ID.
- Use the *scope* and *properties* arguments as needed for your business needs.
- Modify the *properties* object to specify any claim values, as needed. For example:

```
// Setup claims for profile and email OpenID Connect scopes.
Do properties.SetClaimValue("sub",username)
Do properties.SetClaimValue("preferred_username",username)
Do properties.SetClaimValue("email",email)
Do properties.SetClaimValue("email_verified",0,"boolean")
Do properties.SetClaimValue("name",fullname)
```

- In the case of any errors, set the *sc* variable.
- Return 1 if the user is considered valid; return 0 in all other cases.

Note that after the return from **ValidateClient()**, the authorization server automatically sets the following values in the *properties* object, if these values are missing:

- In *properties*.ClaimValues:
  - iss — URL of authorization server
  - sub — `client_id`
  - exp — expiration time in seconds since December 31, 1840
- In *properties*.CustomProperties:
  - client_id — `client_id` of the requesting client

5.4 Details for the %OAuth2.Server.Properties Object

The methods described in the previous section use the argument *properties*, which is an instance of %OAuth2.Server.Properties. The %OAuth2.Server.Properties class is intended to hold information that needs to be passed from method to method within the authorization server code. This section describes the basic properties in this class, as well as the properties related to claims. The class also has methods for working with claims; the last subsection describes them.

5.4.1 Basic Properties

The %OAuth2.Server.Properties class has the following basic properties, used to convey information for any internal processing of your custom code:

RequestProperties

```
Property RequestProperties as array of %String (MAXLEN=16384);
```

Contains the query parameters from the authorization request.

Because this property is an array, use the usual array interface to work with it. (The same comment applies to the other properties of this class.) For example, to get the value of a query parameter, use RequestProperties.**GetAt(***parmname***)**, where *parmname* is the name of the query parameter.

ResponseProperties

```
Property ResponseProperties as array of %String (MAXLEN=1024);
```

Contains any properties to be added to the JSON response object to a token request. Set this property as needed.

CustomProperties

```
Property CustomProperties as array of %String (MAXLEN=1024);
```

Contains any custom properties to be used to communicate between various pieces of customization code. See the section "Updating properties.CustomProperties."

ServerProperties

```
Property ServerProperties as array of %String (MAXLEN=1024);
```

Contains any properties that the authorization server chooses to share with the customization code. The `logo_uri`, `client_uri`, `policy_uri` and `tos_uri` client properties are shared in this way for use by the Authentication Class.

5.4.2 Properties Related to Claims

The %OAuth2.Server.Properties class contains the IntrospectionClaims, IDTokenClaims, UserinfoClaims, and JWTClaims properties, which carry information about required claims, specifically custom claims.

The class also contains the ClaimValues property, which carries the actual claim values. Your customization code should set the values of the claims (typically in the ValidateUser class).

The following list describes these properties:

IntrospectionClaims

```
Property IntrospectionClaims as array of %OAuth2.Server.Claim;
```

Specifies the claims to be returned by the Introspection endpoint (beyond the base required claims). The authorization server will return the `scope`, `client_id`, `username`, `token_type`, `exp`, `iat`, `nbf`, `sub`, `aud`, `iss`, and `jti` claims even if they are not in this property.

In most cases, the value of this property can be an empty string; this property is included to support the `claims` request parameter (see OpenID Connect Core section 5.5 for details).

Formally, this property is an array in which the array key is the claim name (which matches the name in the ClaimValues property) and the array value is an instance of %OAuth2.Server.Claim. The %OAuth2.Server.Claim class has the following properties:

- Essential

  ```
  property Essential as %Boolean [ InitialExpression = 0 ];
  ```

  Specifies whether the claim is essential or voluntary. The value 1 means essential and the value 0 means voluntary.

- Values

  ```
  property Values as list of %String(MAXLEN=2048);
  ```

  Specifies the list of permissible values for this claim.

The value of the claims will usually be set by the ValidateUser class.

IDTokenClaims

```
Property IDTokenClaims as array of %OAuth2.Server.Claim;
```

Specifies the claims that the authorization server requires in the IDToken (beyond the base set of required claims). The authorization server requires the `iss`, `sub`, `exp`, `aud`, and `azp` claims even if these claims are not in this property.

This property is an array of objects; for details, see the entry for the IntrospectionClaims property.

In most cases, the value of this property can be an empty string; this property is included to support the `claims` request parameter (see OpenID Connect Core section 5.5 for details).

UserinfoClaims

```
Property UserinfoClaims as array of %OAuth2.Server.Claim;
```

Specifies the claims to be returned by the Userinfo endpoint (beyond the base required claims). The authorization server will return the `sub` claim even if that claim is not in this property.

In most cases, the value of this property can be an empty string; this property is provided to support section 5.5 of OpenID Connect Core.

This property is an array of objects; for details, see the entry for the IntrospectionClaims property.

The claims are defined based on the scope and request claims parameter. The value to be returned for the claim will have the same key in the ClaimValues property. The value of the claims will usually be set by the ValidateUser class.

JWTClaims

```
Property JWTClaims as array of %OAuth2.Server.Claim;
```

Specifies the claims that are needed for the JWT access token that is returned by the default JWT-based access token class (%OAuth2.Server.JWT) beyond the base set of required claims. The authorization server will return the `iss`, `sub`, `exp`, `aud`, and `jti` claims even if they are not in this property.

This property is an array of objects; for details, see the entry for the IntrospectionClaims property.

The claims are defined by the customization code. The value of the claims will usually be set by the ValidateUser class.

ClaimValues

```
property ClaimValues as array of %String(MAXLEN=1024);
```

Specifies the actual claim values and their types. To work with this property, use the methods in the next section.

If you need to work with this property directly, note that this property is an array in which:

- The array key is the claim name.
- The array value has the form **$LISTBUILD**(*type*,*value*), where *type* holds the type of the value, and *value* holds the actual value. The *type* can be `"string"`, `"boolean"`, `"number"`, or `"object"`. If *type* is `"object"`, then *value* is a JSON object serialized as a string.

  Note that *value* can be a $LIST structure. In this case, when the claim value is serialized, it is serialized as a JSON array, in which each array item has the given *type*.

5.4.3 Methods for Working with Claims

The %OAuth2.Server.Properties class also provides instance methods that you can use to work with that simplify working with the ClaimValues property.

SetClaimValue()

```
Method SetClaimValue(name As %String, value As %String, type As %String = "string")
```

Updates the ClaimValues property by setting the value of the claim named by the *name* argument. The *type* argument indicates the type of the claim: `"string"` (the default) , `"boolean"`, `"number"`, or `"object"`. If *type* is `"object"`, then *value* must be a JSON object serialized as a string.

Note that *value* can be a $LIST structure. In this case, when the claim value is serialized, it is serialized as a JSON array, in which each array item has the given *type*.

RemoveClaimValue()

```
Method RemoveClaimValue(name As %String)
```

Updates the ClaimValues property by removing the claim named by the *name* argument.

GetClaimValue()

```
Method GetClaimValue(name As %String, output type) As %String
```

Examines the ClaimValues property and returns the value of the claim named by the *name* argument. The *type* argument, which is returned as output, indicates the type of the claim; see **SetClaimValue()**.

NextClaimValue()

```
Method NextClaimValue(name As %String) As %String
```

Returns the name of the next claim (in the ClaimValues property) after the given claim.

5.5 Locations of the Authorization Server Endpoints

When you use a Caché instance as an OAuth 2.0 authorization server, the URLs for the authorization endpoints are as follows:

Endpoint	URL
Issuer endpoint	`https://`*serveraddress*`/oauth2`
Authorization endpoint	`https://`*serveraddress*`/oauth2/authorize`
Token endpoint	`https://`*serveraddress*`/oauth2/token`
Userinfo endpoint	`https://`*serveraddress*`/oauth2/userinfo`
Token introspection endpoint	`https://`*serveraddress*`/oauth2/introspection`
Token revocation endpoint	`https://`*serveraddress*`/oauth2/revocation`

In all cases, *serveraddress* is the IP address or host name of the server on which the Caché instance is running.

5.6 Creating Client Definitions on a Caché OAuth 2.0 Authorization Server

This section describes how to create a client definition on a Caché OAuth 2.0 authorization server, if you have not registered the client dynamically. First, set up the Caché OAuth 2.0 authorization server as described earlier in this chapter. Then use the Management Portal to do the following:

1. Select **System Administration** > **Security** > **OAuth 2.0** > **Server Configuration**.
2. Click the **Client Configurations** button to view the client descriptions. This table is initially empty.
3. On the **General** tab, specify the following details:
   - **Name** — Specify the unique name of this client.
   - **Description** — Specify an optional description.
   - **Client type** — Specify the type of this client. The choices are **public** (a public client, per RFC 6749), **confidential** (a confidential client, per RFC 6749), and **resource** (a resource server which is not also a client).
   - **Redirect URLs** — One or more expected redirect URLs for this client.
   - **Supported grant types** — Specify the grant types that this client can use to create an access token. Select at least one.
   - **Supported response types** — Select the OAuth 2.0 response_type values that the Client will restrict itself to using.
   - **Authentication type** — Select the type of authentication (as specified in RFC 6749 or OpenID Connect Core section 9) to be used for HTTP requests to the authorization server. Select one of the following:

  - **none**
  - **basic**
  - **form encoded body**
  - **client secret JWT**
  - **private key JWT**

- **Authentication signing algorithm** — Select the algorithm that must be used for signing the JWTs used to authenticate this client at the token endpoint (if the authentication type is **client secret JWT** or **private key JWT**). If you do not select an option, any algorithm supported by the OpenID provider and the relying party may be used.

4. If needed, select the **Client Credentials** tab and view the following details:
   - **Client ID** — Client ID as specified in RFC 6749. Caché generates this string.
   - **Client secret** — Client secret as specified in RFC 6749. Caché generates this string.
5. On the **Client Information** tab, specify the following details:
   - **Launch URL** — Specify the URL used to launch this client. In some circumstances, this value can be used to identify the client and can be used as the value of the `aud` claim.
   - **Authorization display** section:
     - **Client name** — Specify the name of the client to be presented to the end user.
     - **Logo URL** — Specify a URL that references a logo for the client application. If you specify this option, the authorization server displays this image to the end user during approval. The value of this field must point to a valid image file.
     - **Client home page** — Specify the URL of the home page of the client. The value of this field must point to a valid web page. If you specify this option, the authorization server displays this URL to the end user in a followable fashion.
     - **Policy URL** — Specify the URL that the Relying Party Client provides to the end user to read about the how the profile data will be used. The value of this field must point to a valid web page.
     - **Terms of service URL** — Specify the URL that the Relying Party Client provides to the end user to read about the Relying Party's terms of service. The value of this field must point to a valid web page.
   - **Contact emails** — Comma-separated list of email addresses suitable for use in contacting those responsible for the client application.
   - **Default max age** — Specify the default maximum authentication age, in seconds. If you specify this option, the end user must be actively re-authenticated when the maximum authentication age is reached. The max_age request parameter overrides this default value. If you omit this option, there is no default maximum authentication age.
   - **Default scope** — Specify the default scope, as a blank separated list, for access token requests.
6. On the **JWT Settings** tab, specify the following details:
   - **JSON Web Token (JWT) Settings** — Specifies the source of the public keys that the client uses to verify signatures of JWTs from the authorization server and to encrypt JWTs sent to the authorization server.

     By default, the dynamic registration process generates a pair of JWKSs (JSON web key sets). One JWKS is private and contains all the needed private keys (per algorithm) as well as the client secret for use as a symmetric key; this JWKS is never shared. The other JWKS contains the corresponding public keys and is publicly available. The dynamic registration process also copies the public JWKS to the client.

The other options are as follows:

- **JWKS from URL** — Specify a URL that points to a public JWKS and then load the JWKS into Caché.
- **JWKS from file** — Select a file that contains a public JWKS and then load that file into Caché.
- **X509 certificate** — For details, see "Using Certificates for an OAuth 2.0 Authorization Server," in the appendix "Certificates and JWTs (JSON Web Tokens)."

To access any of these options, first select **Source other than dynamic registration**.

7. Select **Save**.

5.7 Rotating Keys Used for JWTs

In most cases, you can cause the authorization server to generate new public/private key pairs; this applies only to the RSA keys used for the asymmetric RS256, RS384, and RS512 algorithms. (The exception is if you specify **Source other than dynamic registration** as **X509 certificate**. In this case, it is not possible to generate new keys.)

Generating new public/private key pairs is known as *key rotation*; this process adds new private RSA keys and associated public RSA keys to the private and public JWKSs.

When you perform key rotation on the authorization server, the authorization server uses the new private RSA keys to sign JWTs to be sent to the clients. Similarly, the authorization server uses the new public RSA keys to encrypt JWTs to be sent to the clients. To decrypt JWTs received from the clients, the authorization server uses the new RSA keys, and if that fails, uses the old RSA keys; thus the server can decrypt a JWT that was created using its old public RSA keys.

Last, if the authorization server cannot verify a signed JWT received from a client, then if the authorization server has the URL for the client public JWKS, the authorization server obtains a new public JWKS and tries again to verify the signature. (Note that the authorization server has a URL for the client public JWKS if you used dynamic discovery or if the configuration specified the **JWKS from URL** option; otherwise, the authorization server does not have this URL.)

To rotate keys for the authorization server:

1. Select **System Administration** > **Security** > **OAuth 2.0** > **Server Configuration**.

   The system displays the configuration for the authorization server.

2. Select the **Rotate Keys** button.

Note: The symmetric HS256, HS384, and HS512 algorithms always use the client secret as the symmetric key.

5.7.1 API for Key Rotation on the Authorization Server

To rotate keys programmatically on the authorization server, call the **RotateKeys()** method of OAuth2.Server.Configuration.

To obtain a new client JWKS, call the **UpdateJWKS()** method of OAuth2.Server.Client.

For details on these methods, see the class reference.

5.8 Getting a New Public JWKS from a Client

In most cases, a client generates a public/private pair of JWKSs. There are different ways in which the authorization server can receive the public JWKS. One way is for the client to provide the public JWKS at a URL; see the **JWKS from URL** option in "Creating Client Definitions on a Caché OAuth 2.0 Authorization Server."

If the client was defined with **JWKS from URL** and if the client generates a new pair of JWKSs, you can cause the authorization server to obtain the new public JWKS from the same URL. To do so:

1. In the Management Portal, select **System Administration** > **Security** > **OAuth 2.0** > **Server Configuration**.

   The system displays the configuration for the authorization server.

2. Select the **Update JWKS** button.

If the client was not defined with **JWKS from URL** and if the client generates a new pair of JWKSs, it is necessary to obtain the public JWKS, send it to the authorization server, and load it from a file.

A

Creating Configuration Items Programmatically

The earlier chapters of this book describe how to use the Management Portal to configure OAuth 2.0 clients, resource servers, and authorization servers. InterSystems also supports creating these configuration items programmatically. The following subsections provide the details for clients (including resource servers) and for the authorization server.

A.1 Creating the Client Configuration Items Programmatically

To programmatically create the configuration items for an OAuth 2.0 client or an OAuth 2.0 resource server:

1. Create a server description.
2. Create an associated client configuration.

A.1.1 Creating a Server Description

A server description is an instance of OAuth2.ServerDefinition. To create a server definition:

1. Switch to the %SYS namespace.
2. If the authorization server supports discovery, call the **Discover()** method of %SYS.OAuth2.Registration. This method is as follows:

   ```
   ClassMethod Discover(issuerEndpoint As %String,
                        sslConfiguration As %String,
                        Output server As OAuth2.ServerDefinition) As %Status
   ```

   Where:

   - *issuerEndpoint* specifies the endpoint URL to be used to identify the authorization server.
   - *sslConfiguration* specifies the alias of the Caché SSL/TLS configuration to use calling the **Discover()** method.
   - *server*, which is returned as output, is an instance of OAuth2.ServerDefinition,

3. Then save the returned instance of OAuth2.ServerDefinition.

Or, if the authorization server does not support discovery:

1. Switch to the %SYS namespace.
2. Create an instance of OAuth2.ServerDefinition.
3. Set its properties. In most cases, the names of the properties match the labels shown in the Management Portal (apart from spaces and capitalization). For reference, see "Manually Creating a Server Description." The properties are as follows:
   - IssuerEndpoint
   - SSLConfiguration
   - InitialAccessToken, which corresponds to the **Registration access token** field.
   - Metadata, which is an instance of **OAuth2.Server.Metadata**, and which includes many properties. See "OpenID Provider Metadata" in https://openid.net/specs/openid-connect-discovery-1_0.html.

   For information on ServerCredentials, see "Using Certificates for an OAuth 2.0 Authorization Server," in the appendix "Certificates and JWTs (JSON Web Tokens)."
4. Save the instance.

A.1.2 Creating a Client Configuration

A client configuration is an instance of OAuth2.Client. To create a client configuration:

1. Switch to the %SYS namespace.
2. Create an instance of OAuth2.Client.
3. Set its properties. In most cases, the names of the properties match the labels shown in the Management Portal (apart from spaces and capitalization). For reference, see "Configuring and Dynamically Registering a Client." The properties are as follows:
   - ApplicationName
   - ClientId, which you do not need to set manually if you will register the client dynamically.
   - ClientSecret, which you do not need to set manually if you will register the client dynamically.
   - DefaultScope
   - Description
   - Enabled
   - JWTInterval
   - Metadata, which is an instance of **OAuth2.Client.Metadata**, and which includes many properties. For information, see "Client Metadata" in http://openid.net/specs/openid-connect-registration-1_0-19.html.
   - RedirectionEndpoint, which corresponds to the option **The client URL to be specified to the authorization server to receive responses**. This property is of type %OAuth2.Endpoint. The class OAuth2.Endpoint is a serial class with the properties UseSSL, Host, Port, and Prefix.
   - SSLConfiguration
   - ServerDefinition, which must be an instance of OAuth2.ServerDefinition that you created previously.

   For information on ClientCredentials, see "Using Certificates for an OAuth 2.0 Client," in the appendix "Certificates and JWTs (JSON Web Tokens)."

4. If the authorization server supports dynamic client registration, call the **RegisterClient()** method of %SYS.OAuth2.Registration. This method is as follows:

   ```
   ClassMethod RegisterClient(applicationName As %String) As %Status
   ```

   Where *applicationName* is the name of the client application.

   This method registers the client, retrieves client metadata (including the client ID and client secret), and then updates the instance of OAuth2.Client.

A.2 Creating the Server Configuration Items Programmatically

To programmatically create the configuration items for an OAuth 2.0 authorization server:

1. Create an authorization server configuration.

   Note that you cannot define more than one authorization server configuration on any given Caché instance. Also, to create this configuration, you must be logged in as a user who has USE permission on the %Admin_Secure resource.

2. Create the associated client descriptions.

A.2.1 Creating the Authorization Server Configuration

An authorization server configuration is an instance of OAuth2.Server.Configuration. To create an authorization server configuration:

1. Switch to the %SYS namespace.
2. Create an instance of OAuth2.Server.Configuration
3. Set its properties. In most cases, the names of the properties match the labels shown in the Management Portal (apart from spaces and capitalization). For reference, see "Configuring the Authorization Server." The properties are as follows:
   - AccessTokenInterval
   - AllowUnsupportedScope
   - AudRequired, which corresponds to the **Audience required** option
   - AuthenticateClass
   - AuthorizationCodeInterval
   - ClientSecretInterval
   - CustomizationNamespace
   - CustomizationRoles
   - DefaultScope
   - Description
   - EncryptionAlgorithm
   - GenerateTokenClass

- IssuerEndpoint, which corresponds to the **Issuer endpoint** option, is of type OAuth2.Endpoint. The class OAuth2.Endpoint is a serial class with the properties UseSSL, Host, Port, and Prefix.
- JWKSFromCredentials
- KeyAlgorithm
- Metadata, which is an instance of **OAuth2.Server.Metadata**, and which includes many properties. See "OpenID Provider Metadata" in https://openid.net/specs/openid-connect-discovery-1_0.html.
- RefreshTokenInterval
- ReturnRefreshToken
- SSLConfiguration
- SessionClass
- SessionInterval, which corresponds to the **Session termination interval** option
- SigningAlgorithm
- SupportSession, which corresponds to the **Support user session** option
- SupportedScopes, which corresponds to the table with **Scope** and **Description** columns. This property is an array of strings, and thus uses the usual array interface: **SetAt()**, **GetAt()**, and so on.
- ValidateUserClass

For allowed values for algorithms for signing, key management, and encryption, the class reference for %OAuth2.JWT.

For information on ServerCredentials and ServerPassword, see "Using Certificates for an OAuth 2.0 Authorization Server," in the appendix "Certificates and JWTs (JSON Web Tokens)."

4. Save the instance.

Note that Caché does not support having more than one instance of this class.

Also note that in order to save this instance, you must be logged in as a user who has USE permission on the %Admin_Secure resource.

A.2.2 Creating a Client Description

A client description is an instance of OAuth2.Server.Client. To create a client description:

1. Switch to the %SYS namespace.
2. Create an instance of OAuth2.Server.Client.
3. Set its properties. In most cases, the names of the properties match the labels shown in the Management Portal (apart from spaces and capitalization). For reference, see "Creating a Client Description." The properties are as follows:
   - ClientCredentials
   - ClientType
   - DefaultScope
   - Description
   - LaunchURL
   - Metadata, which is an instance of **OAuth2.Client.Metadata**, and which includes many properties. For information, see "Client Metadata" in http://openid.net/specs/openid-connect-registration-1_0-19.html.

- Name
- RedirectURL, which corresponds to the **Redirect URLs** option. This property is an array of strings, and thus uses the usual array interface: **SetAt()**, **GetAt()**, and so on.

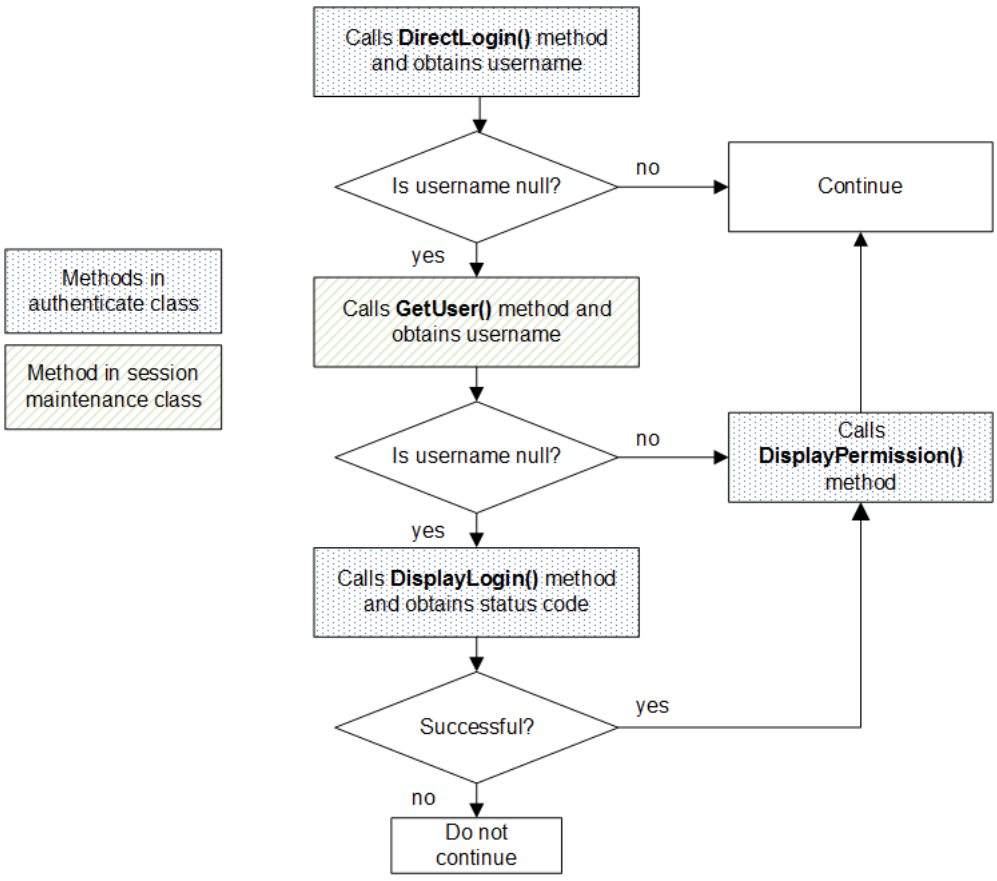
4. Save the instance.

   The system generates values for the ClientId and ClientSecret properties.

B

Implementing DirectLogin()

When you use Caché as an OAuth 2.0 authorization server, normally you implement the **DisplayLogin()** method of the Authenticate class, which displays a page where the user enters a username and password and logs in. If you instead want the server to authenticate without displaying a login form and without using the current session, then implement the **DirectLogin()** method of the Authenticate class. The following flowchart shows how a Caché authorization server identifies the user, when processing a request for an access token:



By default, the **GetUser()** method gets the username that was entered in the previous login.

Note that **DisplayPermissions()** is not called if you implement **DirectLogin()**, because **DirectLogin()** takes responsibility for displaying permissions.

The **DirectLogin()** method has the following signature:

```
ClassMethod DirectLogin(scope As %ArrayOfDataTypes,
                        properties As %OAuth2.Server.Properties,
                        Output username As %String,
                        Output password As %String) As %Status
```

Where:

- *scope* is an instance of %ArrayOfDataTypes that contains the scopes contained in the original client request, possibly modified by the **BeforeAuthenticate()** method. The array keys are the scope values and the array values are the corresponding display forms of the scope values.
- *properties* is an instance of %OAuth2.Server.Properties that contains properties and claims received by the authorization server and modified by methods earlier in the processing. See "Details for the %OAuth2.Server.Properties Object."
- *username*, returned as output, is a username.
- *password*, returned as output, is the corresponding password.

In your implementation, use your own logic to set the *username* and *password* arguments. To do so, use the *scope* and *properties* arguments as needed. To deny access, your method can set the *username* argument to $char(0). In this case, the authorization server will return the `access_denied` error.

The method can also set properties of *properties*; this object is available in later processing.

The method must return a %Status.

C

Certificates and JWTs (JSON Web Tokens)

Each party in OAuth 2.0 requires public/private key pairs. In release 2016.2, it was necessary to use certificates (and their private keys). For reasons of compatibility, this release still contains the options related to certificates. This appendix provides the details for each of the following scenarios.

- OAuth 2.0 client
- OAuth 2.0 resource server
- OAuth 2.0 authorization server

In each case, to generate the private keys and corresponding certificates, you can use the InterSystems public key infrastructure, which is discussed in the *Caché Security Administration Guide*.

Note: As of release 2017.1, Caché can generate a pair of JWKSs (JSON web key sets). One JWKS is private and contains all the needed private keys (per algorithm) as well as the client secret for use as a symmetric key; this JWKS is never shared. The other JWKS contains the corresponding public keys and is publicly available. If you want to use the option of generating JWKSs, ignore this appendix.

C.1 Using Certificates for an OAuth 2.0 Client

An OAuth 2.0 client can receive JWTs (which might be encrypted, signed, or both) from the authorization server. Similarly, the client can send JWTs (which might be encrypted, signed, or both) to the authorization server. If you would like to use certificate/private key pairs for these purposes, consult the table below to determine which certificates you need:

Scenario	Requirement for Client Configuration
Client needs to verify signatures of JWTs sent by authorization server	Obtain a certificate owned by the authorization server, as well as the CA (certificate authority) certificate that signs the server certificate. The public key in this certificate is used for signature verification and encryption.
Client needs to encrypt JWTs sent to authorization server	
Client needs to sign JWTs before sending to authorization server	Obtain a private key for the client, as well as the corresponding certificate and the CA certificate that signs the certificate. The private key is used for signing and decryption.
Client needs to decrypt JWTs sent by authorization server	

In each case, it is also necessary to do the following on the same instance that contains the client web application:

- Provide trusted certificates for Caché to use. See "Providing Trusted Certificates for Caché to Use" in the chapter "Setup and Other Common Activities" in *Securing Caché Web Services*. The trusted certificates must include the certificates that sign the client's certificate and the authorization server's certificate (either or both, depending on which certificates you need).
- Create a Caché credential set that enables Caché to use the certificate. See "Creating and Editing Caché Credential Sets" in the chapter "Setup and Other Common Activities" in *Securing Caché Web Services*.

  For the client certificate, when you create the credential set, be sure to load the private key and provide the password for the private key.
- When you configure the client, select the option **Create JWT Settings from X509 credentials**. Also specify the following:
  - **X509 credentials** — Select the credential set that uses the client's certificate and that contains the corresponding private key (for example, `ClientConfig`).
  - **Private key password** — Enter the password for the private key for this certificate.

C.2 Using Certificates for an OAuth 2.0 Resource Server

An OAuth 2.0 resource server can receive JWTs (which might be encrypted, signed, or both) from the authorization server. Similarly, the resource server can send JWTs (which might be encrypted, signed, or both) to the authorization server. If you would like to use certificate/private key pairs for these purposes, consult the table below to determine which certificates you need:

Scenario	Requirement for Resource Server Configuration
Resource server needs to verify signatures of JWTs sent by authorization server	Obtain a certificate owned by the authorization server, as well as the CA certificate that signs the server certificate. The public key in this certificate is used for signature verification and encryption.
Resource server needs to encrypt JWTs sent to authorization server	
Resource server needs to sign JWTs before sending to authorization server	Obtain a private key for the resource server, as well as the corresponding certificate and the CA certificate that signs the certificate. The private key is used for signing and decryption.
Resource server needs to decrypt JWTs sent by authorization server	

In each case, it is also necessary to do the following on the same instance that contains the resource server web application:

- Provide trusted certificates for Caché to use. See "Providing Trusted Certificates for Caché to Use" in the chapter "Setup and Other Common Activities" in *Securing Caché Web Services*. The trusted certificates must include the certificates that sign the resource server's certificate and the authorization server's certificate (either or both, depending on which certificates you need).
- Create a Caché credential set that enables Caché to use the certificate. See "Creating and Editing Caché Credential Sets" in the chapter "Setup and Other Common Activities" in *Securing Caché Web Services*.

  For the resource server's certificate, when you create the credential set, be sure to load the private key and provide the password for the private key.
- When you configure the resource server, select the option **Create JWT Settings from X509 credentials**. Also specify the following:

- **X509 credentials** — Select the credential set that uses the resource server's certificate and that contains the corresponding private key (for example, `ResourceConfig`).
- **Private key password** — Enter the password for the private key for this certificate.

C.3 Using Certificates for an OAuth 2.0 Authorization Server

An OAuth 2.0 authorization server can receive JWTs (which might be encrypted, signed, or both) from its clients. Similarly, it can send JWTs (which might be encrypted, signed, or both) to its clients. If you would like to use certificate/private key pairs for these purposes, consult the table below to determine which certificates you need:

Scenario	Requirement for Authorization Server Configuration
Authorization server needs to verify signatures of JWTs sent by a client	Obtain a certificate owned by that client, as well as the CA certificate that signs the certificate. The public key in this certificate is used for signature verification and encryption.
Authorization server needs to encrypt JWTs sent to a client	
Authorization server needs to sign JWTs before sending to its clients	Obtain a private key for the authorization server, as well as the corresponding certificate and the CA certificate that signs the certificate. The private key is used for signing and decryption.
Authorization server needs to decrypt JWTs sent by its clients	

In each case, it is also necessary to do the following on the same instance that contains the authorization server:

- Provide trusted certificates for Caché to use. See "Providing Trusted Certificates for Caché to Use" in the chapter "Setup and Other Common Activities" in *Securing Caché Web Services*. The trusted certificates must include the certificates that sign the clients' certificates and the authorization server's certificate (either or both, depending on which certificates you need).
- Create a Caché credential set that enables Caché to use the certificate. See "Creating and Editing Caché Credential Sets" in the chapter "Setup and Other Common Activities" in *Securing Caché Web Services*.

  For the authorization server certificate, when you create the credential set, be sure to load the private key and provide the password for the private key.
- When you configure the server, select the **JWT Settings** tab. On that tab, select the option **Create JWT Settings from X509 credentials**. Also specify the following:
  - **X509 credentials** — Select the credential set that uses the authorization server's certificate and that contains the corresponding private key (for example, `AuthConfig`).
  - **Private key password** — Enter the password for the private key for this certificate.
- When you create client definitions on the server, select the **JWT Settings** tab. On that tab, for **Source other than dynamic registration**, select **X509 certificate**. Also, for **Client credentials** — Select the credential set that uses the client's certificate (for example, `ClientConfig`).