CICS Transaction Server for z/OS

Using the Liberty JWT Feature with CICS

Eric Phan and Nigel Williams

IBM CICS

Table of contents

Introduction

This paper walks you through an example scenario of how a COBOL program can link to a Java program running in CICS Liberty, and how the Java program can use the JWT feature to build or consume a JSON Web Token (JWT).

JWT is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

The paper is aimed at CICS architects and application programmers who want to use a JWT to authenticate with a CICS application or want to send a JWT in a request to an external service.

We start with a look at the anatomy of a JWT, and then describe an example scenario showing how a JWT can be built by one CICS application and used to transmit information to a second CICS application.

We outline the programming and configuration steps required to build and consume a JWT using the Liberty JWT feature.

Sample code for the applications is provided in Github.

About the authors

Eric Phan and Nigel Williams work in the IBM Client Center, Montpellier, France. The center offers a wide range of client technical support across IBM systems and technologies, including IBM Z and CICS Transaction Server. Eric and Nigel work in the mainframe API enablement team.

Introducing JSON Web Tokens

JSON Web Tokens are an open, industry standard (see https://tools.ietf.org/html/rfc7519) for representing claims securely between two parties.

Claims are statements about an entity, for example what identity was used by a user login, or any other type of claims as required by business processes, for example, that the user is an administrator.

The claims in a JSON Web Token (JWT) are encoded as a JSON object and are normally digitally signed with a Message Authentication Code (MAC) and (optionally) encrypted.

A JWT is similar to a Security Assertion Markup Language (SAML) token in that they both convey claims; however, because JSON is less verbose than XML, a JWT is more compact than a SAML token. This makes JWT a good choice to be passed in HTML and HTTP environments.

Some advantages of using JWTs are:

- They are lightweight and easy to use by applications.
- They are signed.
- They have a built-in expiry mechanism.
- They can be extended to contain custom claims.

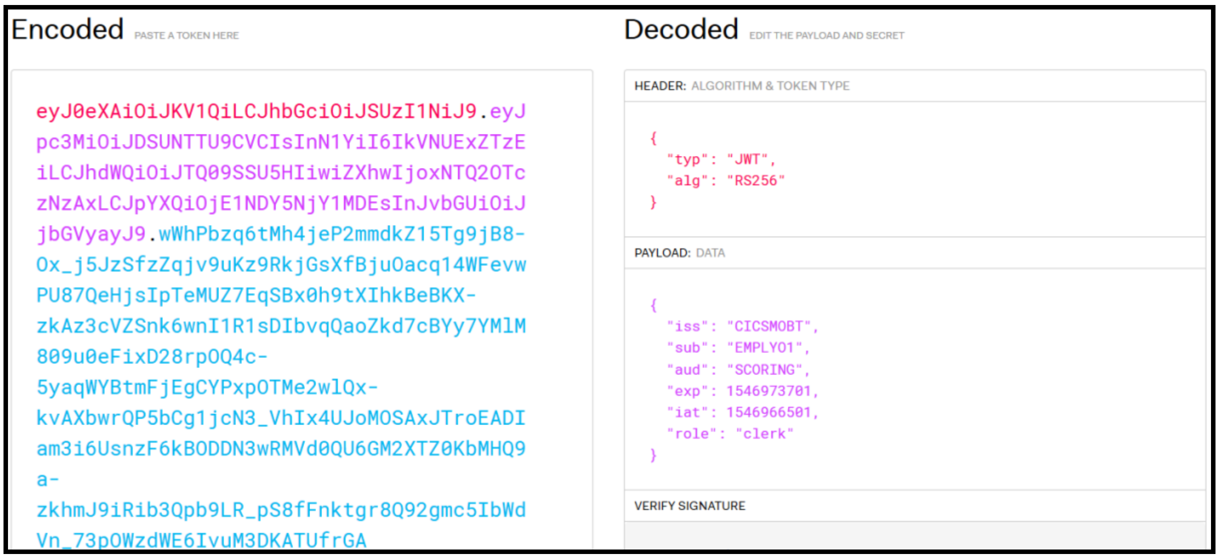
A common scenario for using a JWT is **authentication**. When the user logs in using their credentials, a JSON Web Token is returned. Whenever the user wants to access a protected resource, the client application sends the JWT, typically in the HTTP Authorization header. The JWT is then used by the resource server to identify the user and permit access to the resource. A JWT has several other advantages when used as an authentication token:

- They are self-contained, reducing the need to query a session database multiple times.
- They can be transferred between different servers and security domains that use different user registries.
- A claim in the JWT can be mapped to a specific user in the user registry.
- They do not contain a password.
- They are widely adopted by different Single Sign-On solutions and well-known standards such as OpenID Connect.

For more information on JWTs, see https://jwt.io/introduction/

Anatomy of a JWT

Figure 1 shows that a JWT consists of three parts: a header, payload and signature.



```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJ
pc3MiOiJDSUNTTU9CVCIsInN1YiI6IkVNUExZTzE
iLCJhdWQiOiJTQ09SSU5HIiwiZXhwIjoxNTQ2OTc
zNzAxLCJpYXQiOjE1NDY5NjY1MDEsInJvbGUiOiJ
jbGVyayJ9.wWhPbzq6tMh4jeP2mmdkZ15Tg9jB8-
Ox_j5JzSfzZqjv9uKz9RkjGsXfBjuOacq14WFevw
PU87QeHjsIpTeMUZ7EqSBx0h9tXIhkBeBKX-
zkAz3cVZSnk6wnI1R1sDIbvqQaoZkd7cBYy7YMlM
809u0eFixD28rpOQ4c-
5yaqWYBtmFjEgCYPxpOTMe2wlQx-
kvAXbwrQP5bCg1jcN3_VhIx4UJoMOSAxJTroEADI
am3i6UsnzF6kBODDN3wRMVd0QU6GM2XTZ0KbMHQ9
a-
zkhmJ9iRib3Qpb9LR_pS8fFnktgr8Q92gmc5IbWd
Vn_73pOWzdWE6IvuM3DKATUfrGA
```

```
{
  "typ": "JWT",
  "alg": "RS256"
}
```

```
{
  "iss": "CICSMOBT",
  "sub": "EMPLY01",
  "aud": "SCORING",
  "exp": 1546973701,
  "iat": 1546966501,
  "role": "clerk"
}
```

Figure 1 Anatomy of a JWT

The **header** typically consists of two parts: the type of the token, which is JWT, and the algorithm being used, such as HMAC SHA256 or RSA SHA256. It is Base64Url encoded to form the first part of the JWT.

The **payload** contains the claims. There is a set of predefined claims, for example: iss (issuer), exp (expiration time), sub (subject) and aud (audience). These are not mandatory but recommended, to provide a set of useful, interoperable claims. The payload may also include additional attributes that define custom claims, such as employee role. The payload is Base64Url encoded to form the second part of the JWT.

The **signature** is calculated using the header and the payload. The JWT can be signed either using a shared symmetric key or the private key of a trusted authentication server. The signature is used to verify that the sender of the JWT is who it says it is. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

A JWT can be optionally encrypted using JSON Web Encryption (JWE) which is a means of representing encrypted content using JSON data structures. Using JWE, it is possible to encrypt specific claims in the JWT. Alternatively, you can use HTTPS to encrypt the complete message.

CICS support for JWT

The easiest ways of processing JWTs in CICS all rely on CICS Liberty support, these include the use of:

- JWT feature

  The Liberty **jwt-1.0** feature provides a set of APIs that you can use to work with JWTs. You can use the feature to build or consume a JWT.

This support is available in CICS TS V5.5 and requires APAR PI91554 in CICS TS V5.3 and 5.4.

In this paper we show how to use the Liberty JWT feature to build and consume JWTs (see Using the JWT feature).

- OpenID Connect Client (OIDC) feature

  OpenID Connect is an open identity protocol, built on the OAuth 2.0 protocol, that enables client applications to rely on authentication that is performed by an OpenID Connect Provider.

  CICS Liberty supports OpenID Connect 1.0 and can play a role as a Client, OpenID Connect Provider, or Resource Server. For example, you can use the openidConnectClient-1.0 feature to configure CICS Liberty to accept a JWT as an authentication token. This support is available in CICS TS V5.5 and requires APAR PI91554 in CICS TS V5.3 and 5.4.

  For more information on configuring support for OpenID Connect with Liberty, see Using OpenID Connect in the Liberty z/OS Knowledge Center.

- Java™ Authentication Service Provider Interface for Containers (JASPIC)

  JASPIC is a Java EE standard service provider API that enables the implementation of authentication mechanisms into Java EE Web Applications. You can develop a custom JASPIC authentication provider that authenticates a request using a JWT.

  For more information on using a JASPIC implementation with CICS Liberty to validate a JWT, see the GitHub sample cics-java-liberty-jaspic-jwt.

- Trust Association Interceptor (TAI)

  You can configure CICS Liberty to integrate with a third-party security service by using a Trust Association Interceptors (TAI). The TAI can inspect the HTTP request to see whether it contains a specific security token, for example, a JWT.

  For more information on using a TAI with CICS Liberty to validate a JWT, see the GitHub sample cics-java-liberty-tai-jwt.

Note: CICS Liberty does not support JSON Web Encryption (JWE) as this is a restriction in Liberty. It is highly recommended therefore to use HTTPS to transport requests that contain a JWT.

Using the JWT feature

The JWT feature can be used to build and consume JWTs.

Building a JWT

There are several options available for building a JWT.

- You can configure a Liberty server as a JWT generation endpoint. You define a **jwtBuilder** configuration element in server.xml token which specifies attributes such as issuer and expiration time. Clients can obtain a
  JWT by sending an HTTP GET request with a Basic Authentication HTTP header to the following URL https://<host>:<port>/jwt/ibm/api/<jwtBuilderID>/token

  Use the <jwtBuilderID> to refer to a corresponding jwtBuilder entry in the server.xml file.

- You can programmatically build a JWT by creating an instance of the JwtBuilder object. You can build a JWT based on the default configuration (without specifying a configuration ID) or based on a jwtBuilder configuration defined in server.xml (by specifying the corresponding configuration ID). The JwtBuilder object has methods to set the different attributes before generating the JWT.

  The scenario that is described in the sample LOANS application below uses a jwtBuilder configuration.

Consuming a JWT

You can programmatically verify and parse a JWT by implementing the com.ibm.websphere.security.jwt.JwtConsumer and com.ibm.websphere.security.jwt.JwtToken APIs in your application.

The JwtConsumer Java class can either be instantiated with a default configuration or with a **jwtConsumer** configuration defined in server.xml. The scenario that is described in the sample SCORING application below uses a jwtConsumer configuration.

For more details on on using the Liberty JWT feature, see Building and consuming JSON Web Token (JWT) tokens in Liberty in the Liberty z/OS Knowledge Center.

Example CICS JWT scenario

The two common CICS JWT scenarios are:

i. Build a JWT containing a set of claims and send the JWT to an external service provider. This scenario is shown in Figure 2.

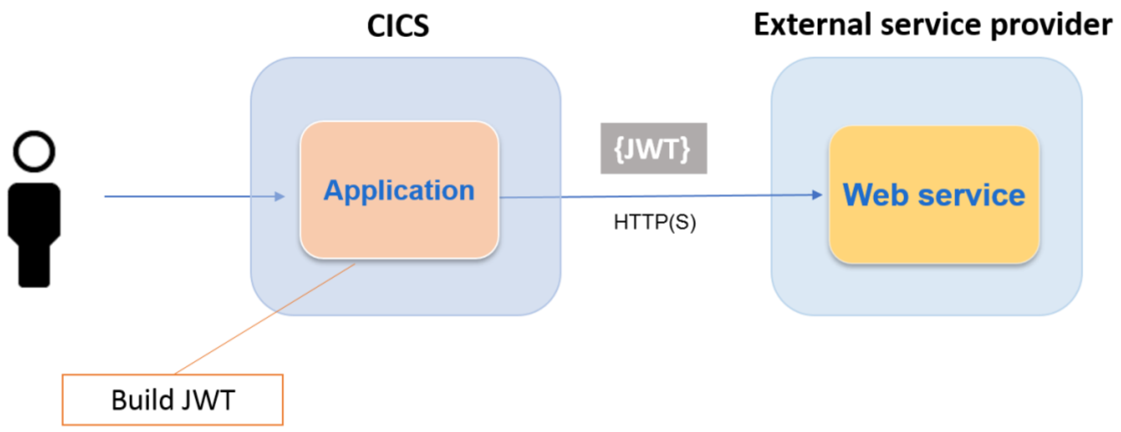


Figure 2 Build a JWT

ii. Consume a JWT containing a set of claims that has been built and sent by an external service requester. This scenario is shown in Figure 3.

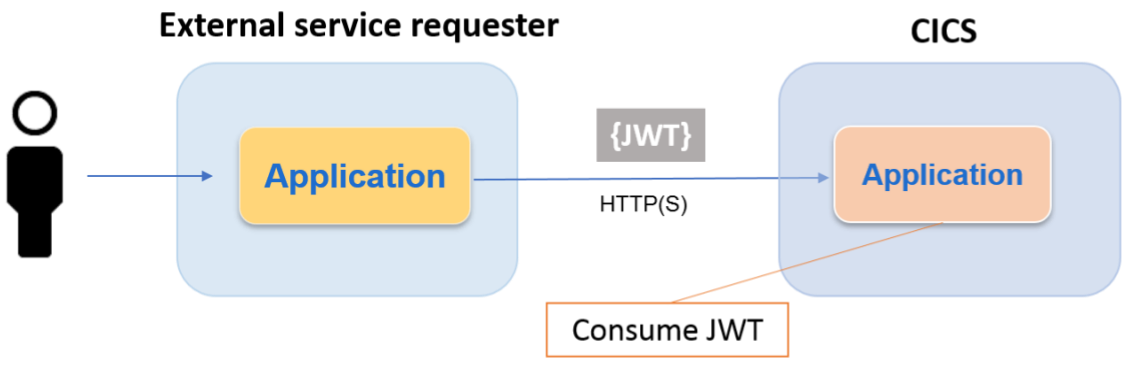


Figure 3 Consume a JWT

For the sake of simplicity, in this paper we combine these scenarios into a single sample scenario in which one CICS application builds a JWT, and then sends the JWT in an HTTP request to a second CICS application.

Sample applications used in the example scenario

Figure 4 shows an overview of our sample scenario based on the LOANS and SCORING applications.

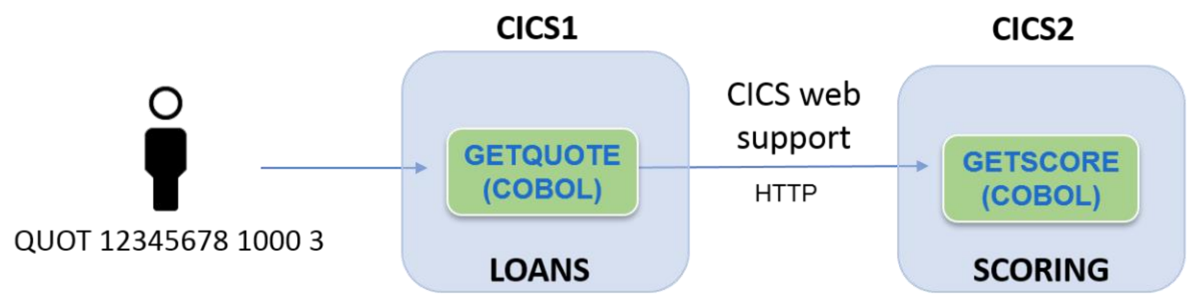


Figure 4 Sample LOANS and SCORING applications

In Figure 4.

i. The end user (a clerk or supervisor) uses the **QUOT** transaction to get a quote for a loan of 1000 dollars over 3 years for customer 12345678.
ii. The **GETQUOTE** COBOL program is part of the LOANS application.
iii. To process the quote request, the GETQUOTE program makes a web request to the **SCORING** application (COBOL program **GETSCORE**) which returns a credit score for customer 12345678.
iv. The **LOANS** application then provides a quote to the end user.

The code for these applications is provided in Github. See Downloading the sample applications.

Tested scenario for JWT with CICS

In our tested scenario, we assume that there is a new requirement to pass a JWT from the LOANS application to the SCORING application.

The different ways of processing JWTs in CICS all rely on CICS Liberty support. However, a COBOL application (or other non-Java language) can link to a Java EE application running in a Liberty JVM server, thus leveraging JWT APIs from the COBOL program.

Our tested scenario is shown in Figure 5.

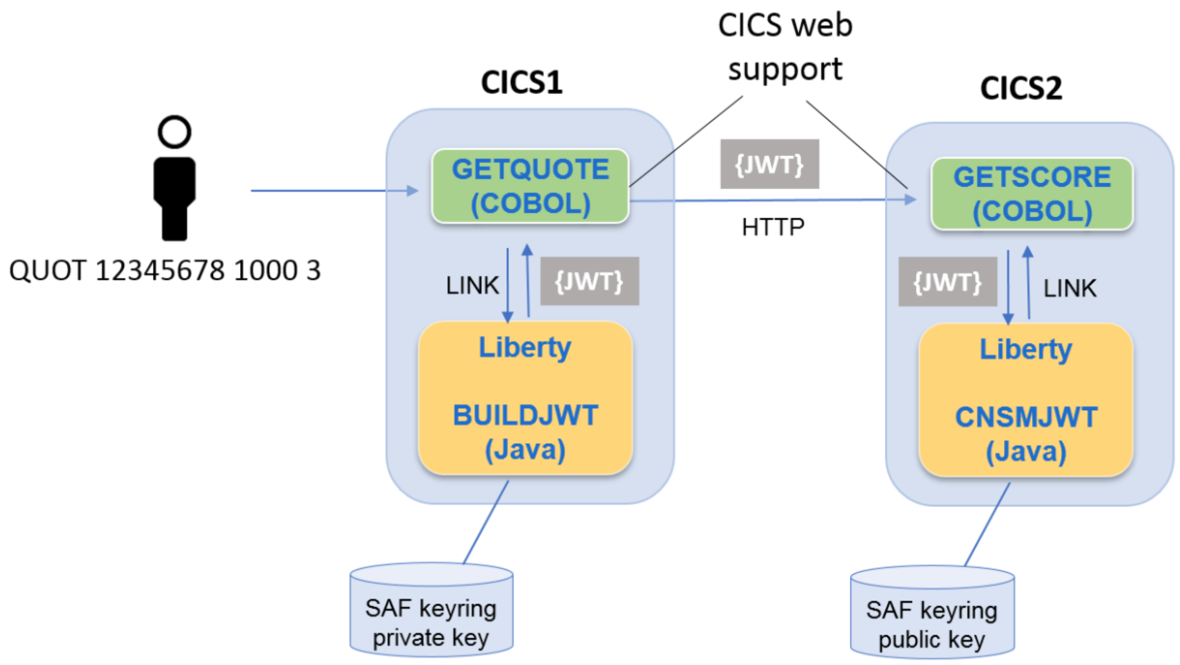


Figure 5 Tested scenario

In Figure 5.

1. Before sending a scoring request, the GETQUOTE program links to a Java program that builds a JWT containing the following standard (predefined) claims:

- **iss** is the issuer which is set to CICS1.
- **sub** is the subject which is set to the CICS task user ID (the ID of the clerk or supervisor).
- **aud** is the audience which is set to SCORING.
- **iat** identifies the time at which the JWT was issued.
- **exp** is the expiration time which is set to allow the JWT to be valid for up to 1 hour. Expired tokens should be rejected.

  Note: use of the **iat** and **exp** claims are optional but highly recommended in order to protect against replay attacks. A **jti** claim can be used for additional protection. A jti is a unique identifier for the JWT. A server normally rejects a JWT that contains the same jti as a previous request.

  In addition to the standard claims, a custom claim **role** is added to the JWT. The role of the user is set to clerk or supervisor.

  The JWT is signed with the RS256 algorithm using a private key from a X.509 certificate stored in RACF.

2. The GETQUOTE COBOL program adds the JWT as an HTTP Authorization request header in the web request to the SCORING application.

3. The GETSCORE program runs under the CICS2 default user (or URIMAP user). It links to a Java program that validates the JWT (signature, expiration time, issuer and audience). The JWT signature is verified using a public key from a X.509 certificate stored in RACF.

   Note that JWT claims can be used by applications for access control or other processing that is dependent on the claims. In our scenario, the GETSCORE program writes a message recording the fact that the end user (from the **sub** claim) in a specific role (from the **role** claim) has run the scoring request. As an alternative, we could check a custom authorization table if the user or role is authorized to call a specific function or use the EXEC CICS QUERY SECURITY command to determine the level of access that the user has to a resource.

The sample applications use the CICS web support and Link to Liberty capabilities:

- CICS web support enables CICS to be used as a web client or a web server.
  For information on CICS web support, see Configuring CICS web support components in the CICS TS Knowledge Center.

- Link to Liberty allows a non-Java CICS program to link to a Java program running in a Liberty JVM server. For information on the CICS Link to Liberty support, see Linking to a Java EE application from a CICS program in the CICS TS Knowledge Center.

Downloading the sample applications

In this section, we describe how to download the sample applications from GitHub.

Prerequisites

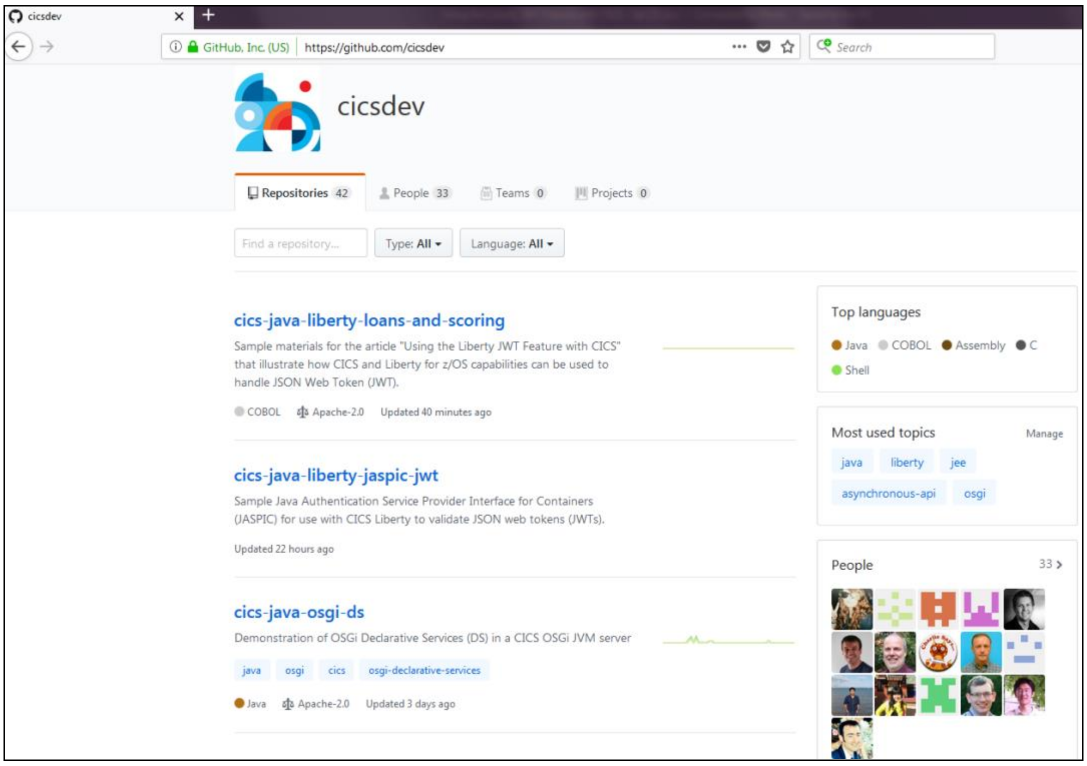
To deploy the sample applications, the following minimum prerequisites must be met:

- CICS TS V5.5; or CICS TS V5.4 or CICS TS V5.3 with APAR PI91554 applied (which is part of RSU 1803)
- COBOL V6.2
- A configured CICS Liberty JVM server that is running in integrated mode
- Eclipse environment with the CICS Explorer SDK for Java EE and Liberty

We used CICS TS V5.4 on z/OS 2.2 that is running Java 1.8 SR5 (64bit), WebSphere Liberty Profile (WLP) 18.0.0.1, IBM CICS Explorer Version 5.4.11 and the COBOL V6.2 compiler. Two CICS regions were used, one for the LOANS application and one for the SCORING application.

Obtaining the sample code

The source project for cics-java-liberty-loans-and-scoring is available at the cicsdev GitHub website. The results for matching repositories are shown in Figure 6.



cics-java-liberty-loans-and-scoring
Sample materials for the article "Using the Liberty JWT Feature with CICS" that illustrate how CICS and Liberty for z/OS capabilities can be used to handle JSON Web Token (JWT).
COBOL Apache-2.0 Updated 40 minutes ago

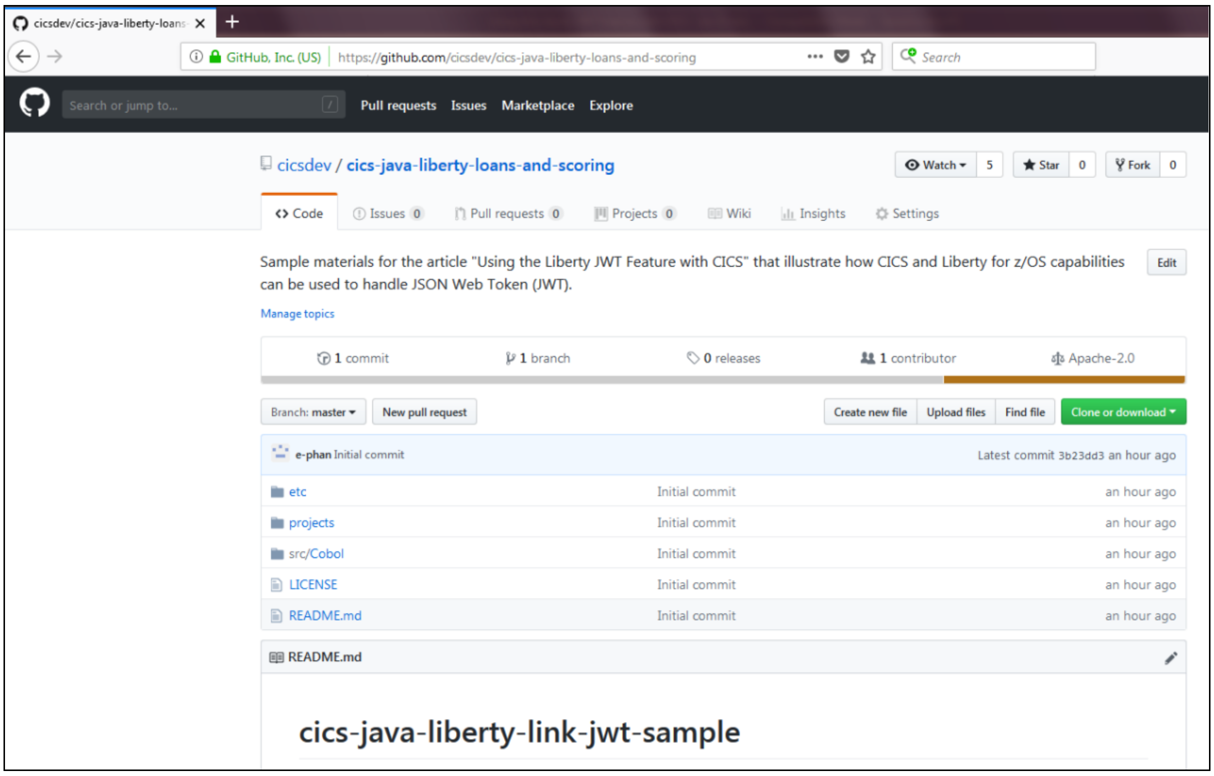
cics-java-liberty-jaspic-jwt
Sample Java Authentication Service Provider Interface for Containers (JASPIC) for use with CICS Liberty to validate JSON web tokens (JWTs).
Updated 22 hours ago

cics-java-osgi-ds
Demonstration of OSGi Declarative Services (DS) in a CICS OSGi JVM server
java osgi cics osgi-declarative-services
Java Apache-2.0 Updated 3 days ago

Figure 6 GitHub cicsdev repositories view

Complete the following steps to download cics-java-liberty-loans-and-scoring:

1. Select the cics-java-liberty-loans-and-scoring repository, click "Clone or download" → "Download ZIP" (see Figure 7).
   The master branch of the repository is downloaded as cics-java-liberty-loans-and-scoring -master.zip into your download directory.
2. Unzip the archive



Sample materials for the article "Using the Liberty JWT Feature with CICS" that illustrate how CICS and Liberty for z/OS capabilities can be used to handle JSON Web Token (JWT).

e-phan Initial commit Latest commit 3b23dd3 an hour ago

etc	Initial commit	an hour ago
projects	Initial commit	an hour ago
src/Cobol	Initial commit	an hour ago
LICENSE	Initial commit	an hour ago
README.md	Initial commit	an hour ago

cics-java-liberty-link-jwt-sample

Figure 7 GitHub cics-java-liberty-loans-and-scoring repository view

For information on enabling the LOANS application to build a JWT, see LOANS application.

For information on enabling the SCORING application to consume a JWT, see SCORING application.

LOANS application

In this section, we describe how to build and deploy the LOANS application in CICS1. The LOANS application has two components:

1. A COBOL program GETQUOTE
2. A Java program running in a Liberty JVM server

The steps that you will go through in this section are as follows:

- Understand the flow of the LOANS application
- Review the resource names used in the LOANS application
- Build the LOANS application
- Deploy the LOANS application

Application flow

Figure 8 shows the LOANS application flow.

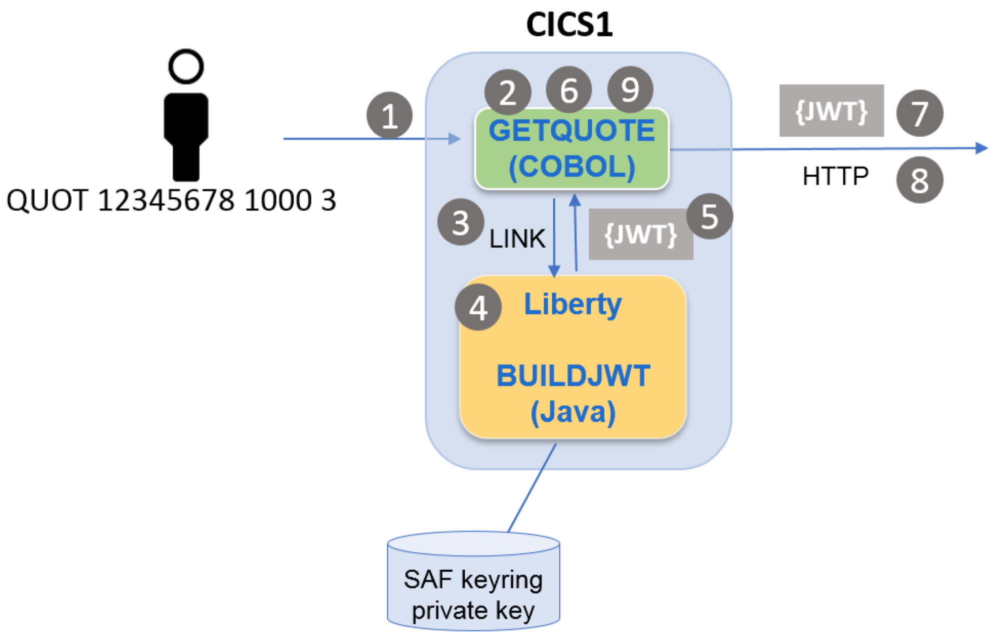


Figure 8 LOANS application flow

In Figure 8:

1. The end user (a clerk or supervisor) uses the QUOT transaction to get a quote for a loan of 1000 dollars over 3 years for customer 12345678.
2. The **GETQUOTE** COBOL program validates the user input and proceeds if the input is valid.
3. The GETQUOTE program creates a container with JWT claims
   a. The EXEC CICS ASSIGN USERID command is used to retrieve the user ID that is being used for the task. The **sub** claim is set to this user ID.
   b. The **aud** claim is set to SCORING.
   c. The **role** claim can be set to clerk or supervisor. Normally, the user role should be retrieved from a registry, but to make it simpler in our example, the role claim is hardcoded to clerk.

   It then links to the **BUILDJWT** java program.
4. The invoked Java method retrieves the provided claims and builds a JWT using the com.ibm.websphere.security.jwt.JwtBuilder and com.ibm.websphere.security.jwt.JwtToken APIs. Additional claims used in the JWT are configured using the **jwtBuilder element** in the CICS Liberty configuration (server.xml).
5. If the JWT generation succeeds, the Java method sends the JWT back to the GETQUOTE program.
6. GETQUOTE builds a JSON message based on the end user input.
7. GETQUOTE uses the CICS WEB API to establish a connection to the remote endpoint. It builds the HTTP POST request with the generated JSON message in the request body and the JWT in the HTTP Authorization header.
8. GETQUOTE receives the response back from the SCORING application. See SCORING application for information about the processing of that application.
9. If there is a credit score in the response, then GETQUOTE accepts the loan request if the score is higher than 75, otherwise it rejects the request.

Definition checklist

The resource names we use in region CICS1 are listed in Table 1.

Table 1. CICS1 resource definitions for LOANS application

Name	**Value**	**Comments**
CSD definitions (group LOANS)		
COBOL program	GETQUOTE	COBOL program to get quotes
Transaction	QUOT	Transaction identifier
Liberty JVM server	JWTB	Liberty JVM server used for building the JWT

Bundle	JWTBBND	LOANS bundle name
URIMAP	SCORECLT	URIMAP used in WEB OPEN and WEB CONVERSE commands
Other definitions		
CICS APPLID	CICSMOBT	CICS region that runs the LOANS application (used as the **issuer** claim in the JWT)
CICS Java Program	BUILDJWT	Program auto-installed based on the @CICSProgram annotation
Eclipse project	com.ibm.cicsdev.cicsjwt.builder	JWT builder project
Class name	CICSJwtBuilder	Java class that uses JwtBuilder
CICS bundle project	com.ibm.cicsdev.cicsjwt.builder.bundle_1.0.0	Bundle that contains the CICSJwtBuilder class
RACF keyring	SignJWT	Certificate used to sign the JWT is connected to this keyring
RACF certificate label	JWT signer	Label of certificate used to sign the JWT

Note: You will probably use different names for some of these resources.

Building the LOANS application

The steps that you will go through in this section are as follows:

- Compile the LOANS COBOL program
- Import the LOANS Eclipse project

Compiling the LOANS COBOL program

The GETQUOTE COBOL program uses the support for JSON, so it requires the **COBOL V6.2** compiler. The COBOL material is provided in the src/Cobol folder downloaded from GitHub.

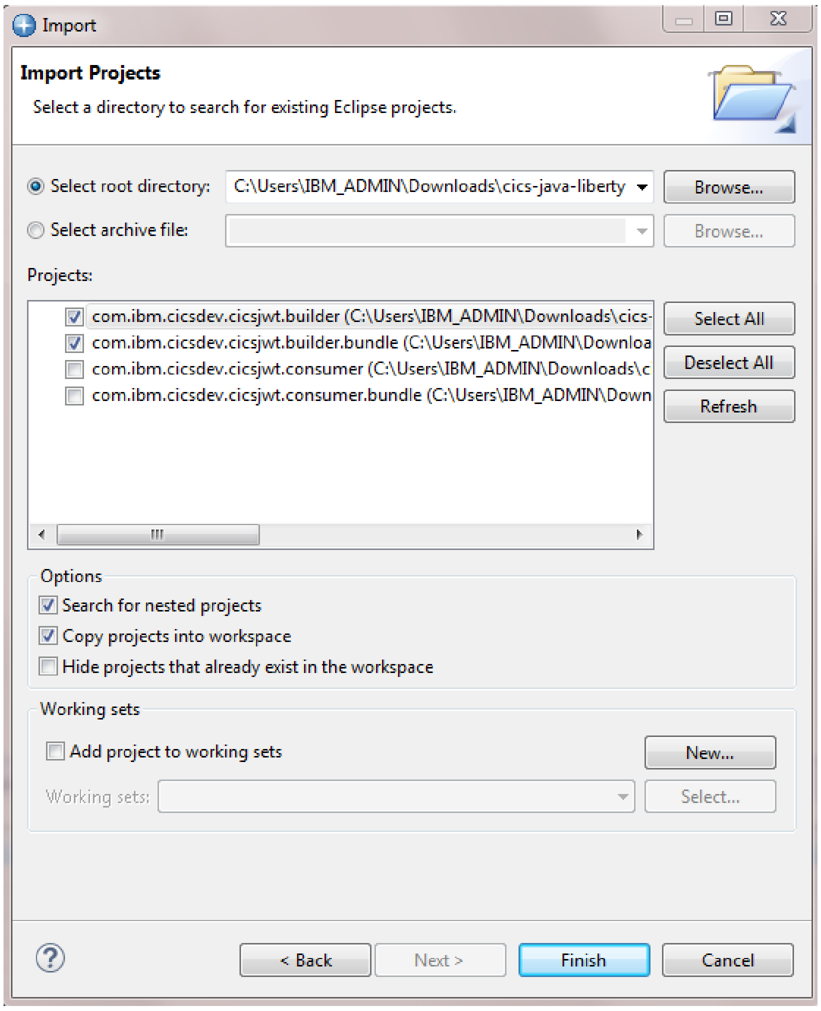
For compiling the LOANS application, you will use GETQUOTE.cbl and the four copybooks (*.cpy). Upload these source files to the appropriate data set members on z/OS and compile GETQUOTE into a load library in DFHRPL or a dynamic load library used by CICS1.

Importing the LOANS Eclipse projects

The steps outlined in this section are executed in the Web perspective of Eclipse with the CICS Explorer SDK for Java EE and Liberty installed.

1. In CICS Explorer, import the LOANS Eclipse projects by clicking on **File** → **Import**. Then select **Existing Projects into Workspace**.
   Click on **Select root directory** and browse to the folder where the archive has been unzipped.

   Check the com.ibm.cicsdev.cicsjwt.builder and com.ibm.cicsdev.cicsjwt.builder.bundle projects, as shown in Figure 9.
   Then click on **Finish**.



Import

Import Projects

Select a directory to search for existing Eclipse projects.

[✓ Select root directory:] C:\Users\IBM_ADMIN\Downloads\cics-java-liberty

Select archive file:

Projects:

- [✓] com.ibm.cicsdev.cicsjwt.builder (C:\Users\IBM_ADMIN\Downloads\cics-
- [✓] com.ibm.cicsdev.cicsjwt.builder.bundle (C:\Users\IBM_ADMIN\Downloa
- [ ] com.ibm.cicsdev.cicsjwt.consumer (C:\Users\IBM_ADMIN\Downloads\c
- [ ] com.ibm.cicsdev.cicsjwt.consumer.bundle (C:\Users\IBM_ADMIN\Down

Options

- [✓] Search for nested projects
- [✓] Copy projects into workspace
- [ ] Hide projects that already exist in the workspace

Working sets

- [ ] Add project to working sets

Working sets:

Figure 9. Import LOANS Eclipse projects

2. This step does not require any user action, it aims at providing some explanation on the com.ibm.cicsdev.cicsjwt.builder project and on how the JwtBuilder API is used.

   In the com.ibm.cicsdev.cicsjwt.builder project, you should see the following :
   - The Java class CICSJwtBuilder.
   - The list of Java libraries needed for the compilation.
   - The com.ibm.cicsdev.cicsjwt.datastructures.jar JAR file.

The IBM Record Generator for Java V3.0 has been used to generate the Java helper classes from the associated-data (ADATA) files that are produced from compiling COBOL copybooks; these Java helper classes are used in CICSJwtBuilder to access the COBOL record structures. For more information on using the IBM Record Generator for Java v3.0 with CICS, see the article IBM Record Generator for Java V3.0 now available.

There are two generated Java helper classes: JwtClaimsData for the JWTCLAIM copybook and JwtTokenData for the JWTTOKEN copybook. Both have been compiled and archived into com.ibm.cicsdev.cicsjwt.datastructures.jar.

The buildJwt method retrieves the claims provided by GETQUOTE and then uses the JwtBuilder API to instantiate an instance of a JWT generator based on the jwtBuilder configuration element named **myJWTBuilder**. This jwtBuilder configuration is defined in the Liberty server.xml file, see Configuring the Liberty server in CICS1. The id attribute of the jwtBuilder configuration element is what identifies it and makes it usable from the JwtBuilder API.

Creating a JwtBuilder object based on a configuration element allows to directly load the configured parameters instead of setting them. More parameters can also be set after the JwtBuilder object has been created. But defining a jwtBuilder configuration in server.xml is not mandatory, the parameters that can be set in the configuration element can also be set by the JwtBuilder API. However, it is easier to update a parameter in server.xml and trigger a dynamic update than to modify the Java code, recompile it, redeploy it and trigger a dynamic update (if available).

Thus this sample sets:

- The JWT provider specific parameters in the jwtBuilder configuration element, this includes the signing algorithm, the signing certificate, the token life duration and the issuer claim;
- The user and application related claims through the JwtBuilder API, these values are passed from the GETQUOTE program;

Note: See Appendix 1 for a walkthrough of the Java code.

3. In the com.ibm.cicsdev.cicsjwt.builder.bundle project, you should see:
   - The META-INF/cics.xml file that defines the com.ibm.cicsdev.cicsjwt.builder project.

   - The com.ibm.cicsdev.cicsjwt.builder.warbundle which specifies the JVM server name (by default JWTB).

   Open the com.ibm.cicsdev.cicsjwt.builder.warbundle by double-clicking it in the **Project Explorer** window (see Figure 10).

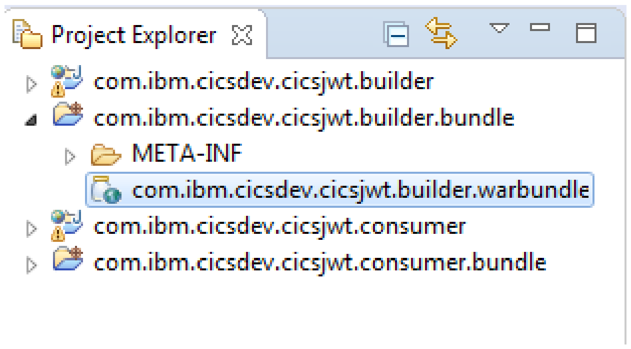
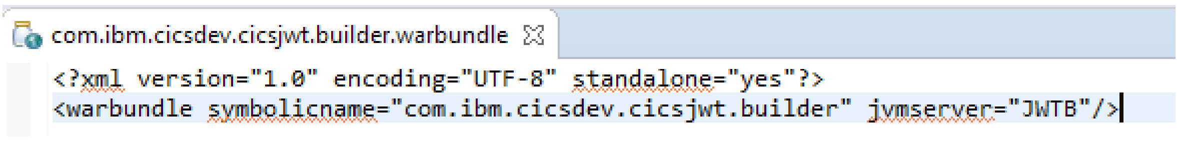

Project Explorer
com.ibm.cicsdev.cicsjwt.builder
com.ibm.cicsdev.cicsjwt.builder.bundle
META-INF
com.ibm.cicsdev.cicsjwt.builder.warbundle
com.ibm.cicsdev.cicsjwt.consumer
com.ibm.cicsdev.cicsjwt.consumer.bundle

Figure 10. Project Explorer window

Update the jvmserver parameter in the warbundle file to match your JVM Server name (see Figure 11).



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<warbundle symbolicname="com.ibm.cicsdev.cicsjwt.builder" jvmserver="JWTB"/>
```

Figure 11. com.ibm.cicsdev.cicsjwt.builder.warbundle file

Note: At this point some of the necessary CICS resources (PROGRAM, TRANSACTION, URIMAP) could be defined in the bundle instead of using the DFHCSDQT file as shown later. And the CICS bundle needs to be defined in one of the following ways: CEDA, DFHCSDUP, CICS Explorer.

Deploying the LOANS application

The steps that you will go through in this section are as follows:

- Add the JWT signing certificate to the CICS1 keyring
- Configure the Liberty server
- Deploy the LOANS CICS bundle
- Install the CICS resources

Adding the JWT signing certificate to the CICS1 keyring

Liberty supports the following types of keystore: Java KeyStore (JKS), SAF keyring and PKCS12. In this sample we used a SAF keyring, however, you may also choose to use a JKS.

1. Create the JWT signing certificate.
   To generate a RACF certificate signed by a Certificate Authority or a self-signed certificate, the RACDCERT GENCERT command can be used.

   A template for the command that we used to create a self-signed certificate is provided in the **etc/GENCERT.jcl** file downloaded from GitHub.

   Replace the **<certOwner>**, Distinguished Name, and NOTAFTER values with your values.

   For more information on creating RACF certificates, see the z/OS Knowledge Center:
   https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.icha400/le-gencert.htm#le-gencert

   This example uses RS256 as the signing algorithm, so the generated certificate needs to use an **RSA** key type.

   Note: The CICS region ID must be authorized to access the certificate.

2. Connect the JWT signing certificate to the CICS1 keyring.
   Create a keyring owned by the CICS region CICS1, and connect the certificate generated in the previous step as **PERSONAL**.

   A template for the commands that we used to create a keyring and to connect the certificate to the keyring is also provided in the etc/GENCERT.jcl file downloaded from GitHub.

   Replace the **<certOwner>** and **<ringOwner>** placeholders with your values and submit the JCL.

   For more information on creating keyrings and connecting certificates to keyrings see the z/OS Knowledge Center:
   https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.icha400/le-addring.htm#le-addring
   https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.icha400/le-connect.htm#le-connect

Configuring the Liberty server in CICS1

A Liberty JVM server is assumed to be available. For more information on setting up a Liberty JVM server in CICS, see section "1.3 Setting up a Liberty JVM Server" in the RedBook Liberty in IBM CICS.

Since the Liberty server does not need to be accessed through HTTP, the server ports can be disabled by specifying:
<httpEndpoint id="defaultHttpEndpoint" host="" httpPort="-1" httpsPort="-1"/>*

1. Add features to the Liberty server.
   In the Liberty server.xml, add the **cicsts:link-1.0** and the **jwt-1.0** features, as shown below:

   ```
           <featureManager>
               <feature>cicsts:core-1.0</feature>
               <feature>jsp-2.3</feature>
               <feature>cicsts:link-1.0</feature>
               <feature>jwt-1.0</feature>
            </featureManager>
   ```

   These features enable the use of the Link to Liberty functionality and the use the Liberty JWT APIs.

2. Configure a keystore.
   In server.xml, add the **keyStore** element which references the keyring that contains the signing certificate:

   ```
   <keyStore id="JWTsigner" fileBased="false" location="safkeyring:///SignJWT"
   password="password" readOnly="true" type="JCERACFKS"/>
   ```

3. Configure a JWT builder.
   In server.xml, add the **jwtBuilder** element which contains the pre-defined claims (expiry and issuer), the keystore reference and the label of the signing certificate:

   ```
   <jwtBuilder id="myJWTBuilder" expiry="1h" issuer="CICSMOBT" keyAlias="JWT
   signer" keyStoreRef="JWTsigner"/>
   ```

   Note: the other claims (aud, sub, role) are passed by the COBOL program and set in the CICSJwtBuilder Java class.

   For more information on the jwtBuilder configuration element see JWT Builder in the Liberty z/OS Knowledge Center.

Deploying the LOANS CICS bundle in CICS1

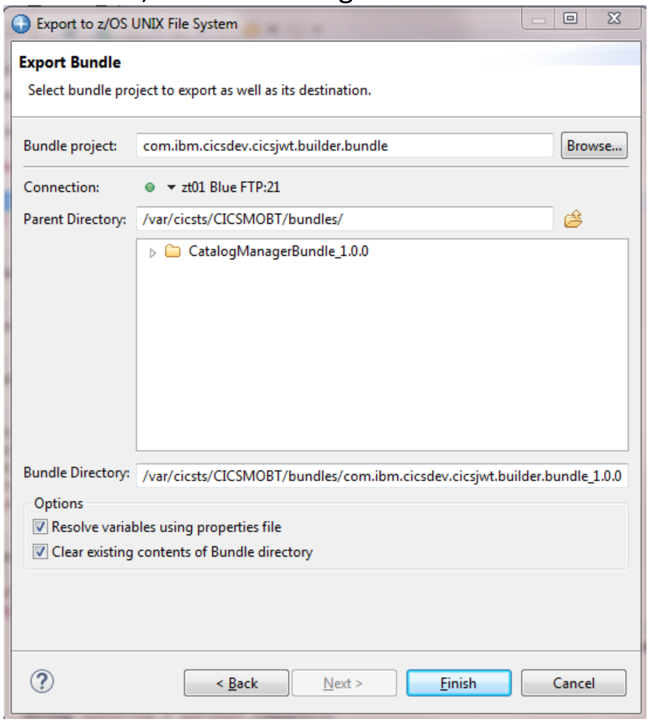
Export the CICS bundle to zFS.

1. In CICS Explorer, right-click on the **com.ibm.cicsdev.cicsjwt.builder.bundle** bundle project, and select **Export Bundle Project to z/OS Unix File System…**

2. Choose **Export to a specific location in the file system**. Click **Next**.

3. In the **Connection** drop-down list choose the **z/OS FTP** connection to use. Otherwise create a **New z/OS FTP Connection…**, by entering the hostname and port number; then click **Save and Connect** and fill in your SAF credentials.

4. Once the connection is established, type the path to the **Parent Directory** for the bundle, as shown in Figure 12.



Export to z/OS UNIX File System

Export Bundle

Select bundle project to export as well as its destination.

Bundle project: com.ibm.cicsdev.cicsjwt.builder.bundle

Connection: zt01 Blue FTP:21

Parent Directory: /var/cicsts/CICSMOBT/bundles/

CatalogManagerBundle_1.0.0

Bundle Directory: /var/cicsts/CICSMOBT/bundles/com.ibm.cicsdev.cicsjwt.builder.bundle_1.0.0

Options

☑ Resolve variables using properties file

☑ Clear existing contents of Bundle directory

Figure 12 Export the CICS bundle

5. Click **Finish**.

Installing the LOANS application CICS resources

A list of the required CICS resources is provided in the etc/DFHCSDQT.txt file.
Some modifications are required for the following resources:

- URIMAP
  Insert the correct value for the **host** and **port** attributes to match the hostname of the CICS region identified as CICS2 and the port number used by CICS2 to receive scoring requests.

- BUNDLE
  Set the **bundledir** attribute to the absolute path (needs to begin with /) of the folder that was exported in the previous section.

Note: When using DFHCSDUP with an in-stream SYSIN to define the bundle, an asterisk can be put in column 72 to indicate continuation, the line is continued on the 1st column of the next line.

Install the **LOANS** group in CICS1.
Note: A JVM server resource definition is not provided in the etc/DFHCSDQT.txt file as it is assumed to be installed prior to running this sample.

SCORING application

In this section, we describe how to build and deploy the SCORING application in CICS2. The SCORING application has two components:

1. A COBOL program GETSCORE
2. A Java program running in a Liberty JVM server

The steps that you will go through in this section are as follows:

- Understand the flow of the SCORING application
- Review the resource names used in the SCORING application
- Build the SCORING application
- Deploy the SCORING application

Application flow

Figure 13 shows the SCORING application flow.

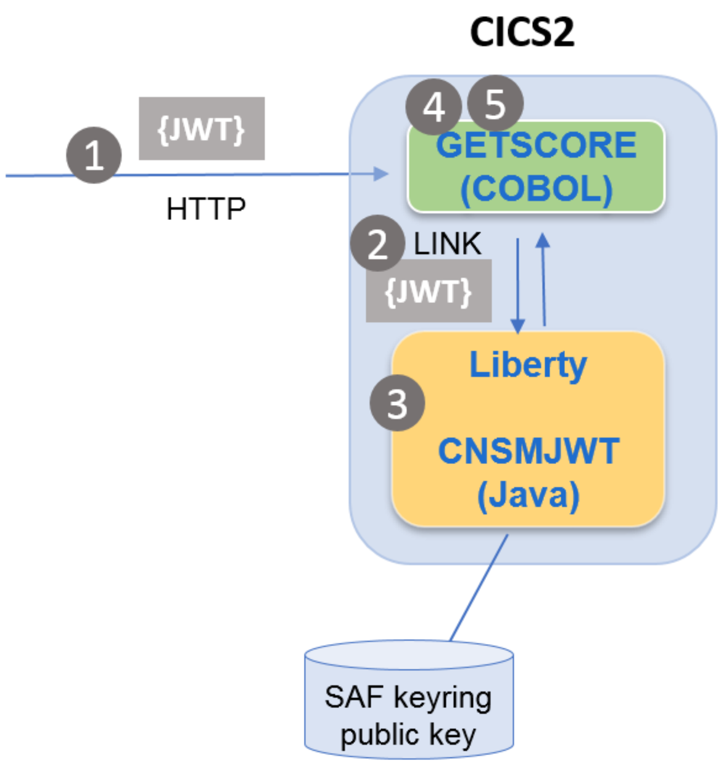


Figure 13 SCORING application flow

In Figure 13:

1. The client application sends a HTTP request that contains a JSON message and a JWT in the HTTP Authorization header. The CICS2 region searches for the URIMAP with the matching path to know which CICS program to run.
2. The **GETSCORE** COBOL program retrieves the JWT from the HTTP request and links to the **CNSMJWT** java program to validate the JWT.

3. The CNSMJWT java program validates the JWT using the com.ibm.websphere.security.jwt.JwtConsumer and com.ibm.websphere.security.jwt.JwtToken APIs. JWT consumer attributes are configured using the **jwtConsumer element** in the CICS Liberty configuration (server.xml). CNSMJWT then returns the claims from the JWT to the GETSCORE program.
4. If the JWT is valid, GETSCORE replies with a credit score in the range of 1-100 (default 75).
5. GETSCORE sends the response back to the client application.

Definition checklist

The resource names we use in CICS2 are listed in Table 2.

Table 2. CICS2 resource definitions for SCORING application

Name	**Value**	**Comments**
CSD definitions (group SCORING)		
COBOL program	GETSCORE	COBOL program to get credit scores
Liberty JVM server	JWTC	Liberty JVM server used for consuming JWTs
Bundle	JWTCBND	SCORING bundle name
URIMAP	SCORESVR	URIMAP for scoring web requests
TCPIPSERVICE	SCORETCP	TCIPSERVICE used for web connections from CICS1
Other definitions		
CICS APPLID	CICSMOB1	CICS region that runs the SCORING application
CICS Java Program	CNSMJWT	Program auto-installed based on the @CICSProgram annotation
Eclipse project	com.ibm.cicsdev.cicsjwt.consumer	JWT consumer project

Class name	CICSJwtConsumer	Java class that uses JwtConsumer
CICS bundle project	com.ibm.cicsdev.cicsjwt.consumer.bundle_1.0.0	Bundle that contains the CICSJwtConsumer class
RACF keyring	TrustJWT	Certificate used to verify the JWT signature is connected to this keyring
RACF certificate label	JWT signer	Certificate used to verify the JWT signature
Port in TCPIPSERVICE	50583	Port defined in TCPIPSERVICE

Note: You will probably use different names for some of these resources.

Building the SCORING application

The steps that you will go through in this section are as follows:

- Compile the SCORING COBOL program
- Import the SCORING Eclipse project

Compiling the SCORING COBOL program

The GETSCORE COBOL program uses the support for JSON, so it requires the **Cobol V6.2** compiler. The Cobol material is provided in the src/Cobol folder downloaded from GitHub.

For compiling the SCORING application, you will use GETSCORE.cbl and the four copybooks (*.cpy). Upload these source files to the appropriate data set members on z/OS and compile GETSCORE into a load library in DFHRPL or a dynamic load library used by CICS2.

Importing the SCORING Eclipse projects

The steps to import the SCORING Eclipse projects are similar to the ones for the LOANS Eclipse projects (see Importing the LOANS Eclipse projects), so we only list the key differences below.

1. The projects to check are now: **com.ibm.cicsdev.cicsjwt.consumer** and **com.ibm.cicsdev.cicsjwt.consumer.bundle**. They are located in the same folder as the LOANS Eclipse projects.

2. This step does not require any user action, it aims at providing an explanation of the com.ibm.cicsdev.cicsjwt.consumer project and on how the JwtConsumer API is used.

   In the com.ibm.cicsdev.cicsjwt.consumer project, you should see the following:
   - The Java class CICSJwtConsumer
   - The list of Java libraries needed for the compilation (Java 1.8, CICS TS V5.4 & Liberty 17.0.0.4 libraries, com.ibm.cicsdev.cicsjwt.datastructures.jar)
   - The com.ibm.cicsdev.cicsjwt.datastructures.jar JAR file.

   As described for the LOANS application, the necessary Java helper classes have been compiled and archived into com.ibm.cicsdev.cicsjwt.datastructures.jar.

   The consumeJwt method retrieves the JWT provided by GETSCORE.
   It then uses the JwtConsumer API to instantiate an instance of a JWT validator based on the jwtConsumer configuration element named **myJWTConsumer**. This jwtConsumer configuration is defined in the Liberty server.xml file, see Configuring the Liberty server in CICS2. The id attribute of the jwtConsumer configuration element is what identifies it and makes it usable from the JwtConsumer API.

   Creating a JwtConsumer object based on a configuration element allows to directly load the configured parameters instead of setting them.
   Like for the jwtBuilder configuration element, defining a jwtConsumer configuration in server.xml is not mandatory. For flexibility, in this sample the parameters are set in the configuration element instead of being set with the JwtConsumer API.

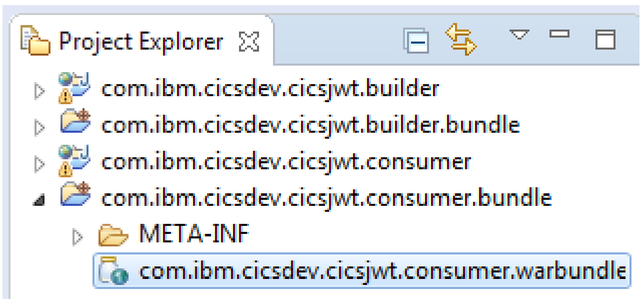
   Unlike the jwtBuilder configuration element, all the required parameters (issuer, signing certificate, audiences) can be directly set in server.xml.
   The CICSJwtConsumer class only creates an instance of JwtConsumer with the myJWTConsumer configuration, and uses it (without modifying it) to validate the JWT.

   Note: See Appendix 2 for a walkthrough of the Java code.

3. In the com.ibm.cicsdev.cicsjwt.consumer.bundle project, you should see:
   - The META-INF/cics.xml file that defines the com.ibm.cicsdev.cicsjwt.consumer project;

   - the com.ibm.cicsdev.cicsjwt.consumer.warbundle which specifies the JVM server name (by default JWTC);

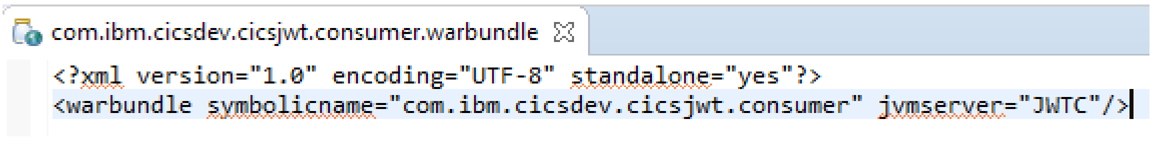
   Open the com.ibm.cicsdev.cicsjwt.consumer.warbundle by double-clicking it in the **Project Explorer** window (see Figure 14).


Project Explorer

- com.ibm.cicsdev.cicsjwt.builder
- com.ibm.cicsdev.cicsjwt.builder.bundle
- com.ibm.cicsdev.cicsjwt.consumer
- com.ibm.cicsdev.cicsjwt.consumer.bundle
  - META-INF
  - com.ibm.cicsdev.cicsjwt.consumer.warbundle

Figure 14. Project Explorer window

Update the jvmserver parameter in the warbundle file to match your JVM Server name (see Figure 15).


com.ibm.cicsdev.cicsjwt.consumer.warbundle

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<warbundle symbolicname="com.ibm.cicsdev.cicsjwt.consumer" jvmserver="JWTC"/>
```

Figure 15. com.ibm.cicsdev.cicsjwt.consumer.warbundle file

Note: At this point some of the necessary CICS resources (PROGRAM, TCPIPSERVICE, URIMAP) can be defined in the bundle instead of using the DFHCSDSC file as shown later. And the CICS bundle needs to be defined in one of the following ways: CEDA, DFHCSDUP, CICS Explorer.

Deploying the SCORING application

The steps that you will go through in this section are as follows:
- Add the JWT trust certificate to the CICS2 keyring
- Configure the Liberty server
- Deploy the SCORING CICS bundle
- Install the CICS resources

Adding the JWT trust certificate to the CICS2 keyring

In the same way that we added the JWT signer certificate to the CICS1 keyring (see Adding the JWT signing certificate to the CICS1 keyring), we now need to add the certificate to the CICS2 keyring.

It is assumed that the two CICS regions are running on the same z/OS system, thus the SAF registry would be the same. If this is not your case, you would need to export the **public** part of the certificate labelled **JWT signer** and import it into the SAF registry of the second z/OS system.

A template for the commands that we used to create a keyring owned by the CICS region CICS2 and to connect the JWT signer certificate as **CERTAUTH** is provided in the etc/TRUSTCRT.jcl file downloaded from GitHub.

Replace the **<certOwner>** and **<ringOwner>** placeholder with your values and submit the JCL.

Configuring the Liberty server in CICS2

The Liberty JVM server is assumed to be configured. Since the Liberty server does not need to be accessed through HTTP, the server ports can be disabled by specifying:
<httpEndpoint id="defaultHttpEndpoint" host="" httpPort="-1" httpsPort="-1"/>*

1. Add features to the Liberty server.
   In the Liberty server.xml, add the **cicsts:link-1.0** and the **jwt-1.0** features, as shown below:

```
          <featureManager>
              <feature>cicsts:core-1.0</feature>
              <feature>jsp-2.3</feature>
              <feature>cicsts:link-1.0</feature>
              <feature>jwt-1.0</feature>
           </featureManager>
```
   These features enable the use of the Link to Liberty functionality and the use the Liberty JWT APIs.

2. Configure a keystore.

In server.xml, add the **keyStore** element which references the keyring that contains the trust certificate:

```
<keyStore id="TrustJWT" fileBased="false" location="safkeyring:///TrustJWT"
password="password" readOnly="true" type="JCERACFKS"/>
```

3. Configure a JWT consumer.
   In server.xml, add the **jwtConsumer** element which contains the keystore reference and the label of the trust certificate:

   ```
   <jwtConsumer id="myJWTConsumer" audiences="SCORING" issuer="CICSMOBT"
   trustStoreRef="TrustJWT" trustedAlias="JWT signer"/>
   ```

   Note: the values specified for the **audiences** and **issuer** attributes in the jwtConsumer element are consistent with the **aud** and **issuer** claims in the JWT.

For more information on the jwtConsumer configuration element see JWT Consumer in the Liberty z/OS Knowledge Center.

Deploying the SCORING CICS bundle in CICS2

The steps to deploy the SCORING CICS bundle are similar to the ones for the LOANS CICS bundle, so only the key differences are listed below.

Export the **com.ibm.cicsdev.cicsjwt.consumer.bundle** CICS bundle instead of the **com.ibm.cicsdev.cicsjwt.builder.bundle** CICS bundle.

Installing the SCORING application CICS resources

A list of the required CICS resources is provided in the etc/DFHCSDSC.txt file.
Some modifications are required for:

- TCPIPSERVICE
  Insert the correct value for the **portnumber** attribute this is the port number on which CICS2 is listening to, it needs to match the port number used by CICS1 to send scoring requests.

- BUNDLE
  Set the **bundledir** attribute to the absolute path (needs to begin with /) of the folder that was exported in the previous section.

  Note: When using DFHCSDUP with an in-stream SYSIN to define the bundle, an asterisk can be put in column 72 to indicate continuation, the line is continued on the 1st column of the next line.

Install the **SCORING** group in CICS2.

One resource definition is missing from the etc/DFHCSDSC.txt file: the JVM server. The JVM server resource definition, which is assumed to be installed prior to running this sample. For more information on setting up a Liberty JVM server in CICS, see section "1.3 Setting up a Liberty JVM Server" in the RedBook Liberty in IBM CICS.

Testing the scenario

In this section, we show the results of testing the scenario and show some potential failure scenarios that might occur in JWT processing.

Using the sample

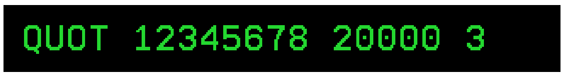
To start using this sample, logon to the CICS1 region with a 3270 emulator.
Then start the transaction **QUOT**:

QUOT <customerID> <amount> <duration>

where:

- customerID - is an alphanumeric identifier with a length of 8 characters;
- amount - is an integer number with a maximum of 6 digits;
- duration - is the duration in years with a maximum of 2 digits;

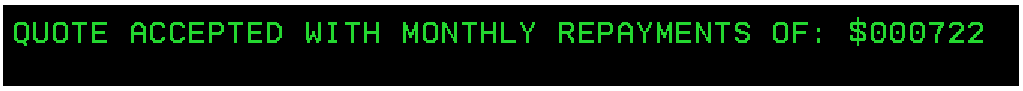
An example is shown on Figure 16.



```
QUOT 12345678 20000 3
```

Figure 16 Running the QUOT transaction

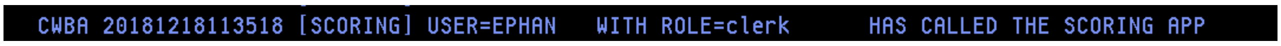
If the sample was correctly configured, the generated JWT should be valid and the quote should be accepted (see Figure 17).



```
QUOTE ACCEPTED WITH MONTHLY REPAYMENTS OF: $000722
```

Figure 17 QUOT transaction runs successfully

The message shown in Figure 18 is written to the CICS2 MSGUSR log.



```
CWBA 20181218113518 [SCORING] USER=EPHAN   WITH ROLE=clerk      HAS CALLED THE SCORING APP
```

Figure 18 Message written to CICS2 MSGUSR log

Failure scenarios

If instead of the message shown in Figure 17, the QUOT transaction issues a "QUOTE REJECTED" message, then it is likely that there is a configuration error.

Below is a list of errors that may occur, and there is more information about each of these potential errors further below.

- The private key of the JWT signer certificate is not accessible to the CICSJwtBuilder in CICS1.
- The JWT trust certificate is not accessible to the CICSJwtConsumer in CICS2.
- The JWT signature cannot be validated by the CICSJwtConsumer.
- The issuer claim is invalid.
- The audience claim is invalid.

Private key of JWT signer certificate is not accessible to CICSJwtBuilder

Example 1 shows the error message written to the CICS1 Liberty messages log when the private key used to sign the JWT cannot be accessed by the Liberty server.

Example 1 Private key of JWT signer certificate is not accessible to CICSJwtBuilder

```
CWWKS6016E: The signing key that is required by the signature algorithm [RS256] is not available. Verify that
the signature algorithm and the jwkEnabled [false] are configured properly. The alias [JWT signer] is not
present in the KeyStore as a key entry.
```

Example 2 shows the associated FFDC exception.

Example 2 FFDC - Private key is not accessible to CICSJwtBuilder

```
FFDC1015I: An FFDC Incident has been created: "java.io.IOException: The private key of JWT signer is not
available or no authority to access the private key com.ibm.ws.ssl.config.WSKeyStore$1 do_getKeyStore" at
ffdc_18.12.18_12.02.23.0.log
```

One reason this error may occur is if CICS1 is not authorized to access the private key. The easiest way to solve this issue is to generate a self-signed certificate owned by the CICS1 started task ID. Alternatively, the CICS1 task ID should be granted access to the private key using the RACF RDATALIB class (see https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.ichd100/usgntrdata.htm#usgntrdata).

JWT trust certificate is not accessible to CICSJwtConsumer

Example 3 shows the FFDC exceptions written when the JWT trust certificate cannot be accessed by the CICS2 Liberty server.

Example 3 FFDCs – JWT trust certificate is not accessible to CICSJwtConsumer

```
FFDC1015I: An FFDC Incident has been created: "java.security.cert.CertificateException: The alias [JWT signer]
is not present in the KeyStore as a certificate entry com.ibm.ws.security.jwt.internal.ConsumerUtil 263" at
ffdc_18.12.18_11.38.02.0.log

FFDC1015I: An FFDC Incident has been created: "com.ibm.websphere.security.jwt.KeyException:
CWWKS6007E: The signing key that is required by the signature algorithm [RS256] and the key type [x509] is
not available. Verify that the signature algorithm and key are configured properly. The alias [JWT signer] is not
present in the KeyStore as a certificate entry com.ibm.ws.security.jwt.internal.ConsumerUtil 233" at
ffdc_18.12.18_11.38.02.1.log

FFDC1015I: An FFDC Incident has been created: "com.ibm.websphere.security.jwt.KeyException:
CWWKS6033E: The public key that matches the alias [JWT signer] within the [TrustJWT] truststore cannot be
retrieved. CWWKS6007E: The signing key that is required by the signature algorithm [RS256] and the key type
[x509] is not available. Verify that the signature algorithm and key are configured properly. The alias [JWT
signer] is not present in the KeyStore as a certificate entry com.ibm.ws.security.jwt.internal.ConsumerImpl
189" at
ffdc_18.12.18_11.38.02.2.log
```

Check that certificate labelled "JWT signer" has been connected to the TrustJWT SAF keyring as **CERTAUTH**.

JWT signature cannot be validated by CICSJwtConsumer

Example 4 shows the FFDC exceptions written when the JWT signature cannot be validated by the CICS2 Liberty server.

Example 4 FFDCs – JWT signature cannot be validated by CICSJwtConsumer

```
FFDC1015I: An FFDC Incident has been created: "org.jose4j.jwt.consumer.InvalidJwtSignatureException: JWS
signature is invalid: JsonWebSignature{"typ":"JWT","alg":"RS256"}-
>eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJ0b2tlbl90eXBlIjoiQmVhcmVyIiwiYXVkIjoiU0NPUklORyIsInN1YiI6Ik
VQSEFOIiwicm9sZSI6ImNsZXJrIiwiaXNzIjoiQ0lDU01PQlQiLCJleHAiOjE1NDUxMzc0NjYsImlhdCI6MTU0NTEzMzg2
Nn0.WzvZqwkkAQuvIQTacvZNv_MHoDW9SE8Pwt_DeEjCwKW3-
TXYiOMB8JLs4wJFSW8i_1wisi81pVNafiNU860hUlkqFpsr3UTjTTQQnTBU8e-
Wk7DHUAuv6A7uNlRfwx9blwPNEhyatLkjg2-
0jsWhcXOGUVSpo6ttL2VztjhnlZZRA0h5twn4eBzsM8kgL7oPRssRc4_AjNjLDSyiOmEhYrNbW4-
EY7AoY_Kd0sLOv2pVXfdD_YMBfjpnyGXjDbQW8y-hyv_-oHdVzgSw2EzouvLRYDXMaVywb3-
2q6_xdTbb4BOa2DbSEqxqFq_PBDOE6BwLcxoNxXfv4iTEyTSGRQ
com.ibm.ws.security.jwt.internal.ConsumerUtil 552" at ffdc_18.12.18_11.51.06.0.log

FFDC1015I: An FFDC Incident has been created: "com.ibm.websphere.security.jwt.InvalidTokenException:
CWWKS6041E: The JSON Web Token (JWT) signature is not valid. JWS signature is invalid:
JsonWebSignature{"typ":"JWT","alg":"RS256"}-
>eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJ0b2tlbl90eXBlIjoiQmVhcmVyIiwiYXVkIjoiU0NPUklORyIsInN1YiI6Ik
VQSEFOIiwicm9sZSI6ImNsZXJrIiwiaXNzIjoiQ0lDU01PQlQiLCJleHAiOjE1NDUxMzc0NjYsImlhdCI6MTU0NTEzMzg2
Nn0.WzvZqwkkAQuvIQTacvZNv_MHoDW9SE8Pwt_DeEjCwKW3-
TXYiOMB8JLs4wJFSW8i_1wisi81pVNafiNU860hUlkqFpsr3UTjTTQQnTBU8e-
Wk7DHUAuv6A7uNlRfwx9blwPNEhyatLkjg2-
0jsWhcXOGUVSpo6ttL2VztjhnlZZRA0h5twn4eBzsM8kgL7oPRssRc4_AjNjLDSyiOmEhYrNbW4-
EY7AoY_Kd0sLOv2pVXfdD_YMBfjpnyGXjDbQW8y-hyv_-oHdVzgSw2EzouvLRYDXMaVywb3-
2q6_xdTbb4BOa2DbSEqxqFq_PBDOE6BwLcxoNxXfv4iTEyTSGRQ
com.ibm.ws.security.jwt.internal.ConsumerImpl 189" at
ffdc_18.12.18_11.51.06.1.log
```

Validate that the jwtConsumer configuration element contains the right attributes and values.
Validate that the trust certificate that is used to validate the signature contains the public key that matches the private key that was used to sign the JWT.

Issuer claim is invalid

Example 5 shows the FFDC exceptions written by the CICS2 Liberty server when the JWT issuer claim is invalid.

Example 5 FFDCs – JWT issuer claim is invalid

```
FFDC1015I: An FFDC Incident has been created: "com.ibm.websphere.security.jwt.InvalidClaimException:
CWWKS6022E: The issuer [CICS1] of the provided JSON web token (JWT) is not listed as a trusted issuer in the
[myJWTConsumer] JWT configuration. The trusted issuers are [CICSMOBT].
com.ibm.ws.security.jwt.internal.ConsumerImpl 189" at ffdc_18.12.18_11.18.22.0.log

com.ibm.websphere.security.jwt.InvalidTokenException: CWWKS6031E: The JSON Web Token (JWT) consumer
[myJWTConsumer] cannot process the token string. CWWKS6022E: The issuer [CICS1] of the provided JSON
web token (JWT) is not listed as a trusted issuer in the [myJWTConsumer] JWT configuration. The trusted
issuers are
[CICSMOBT].
```

Audience claim is invalid

Example 6 shows the FFDC exceptions written by the CICS2 Liberty server when the JWT audience claim is invalid.

Example 6 FFDCs – JWT audience claim is invalid

```
FFDC1015I: An FFDC Incident has been created: "com.ibm.websphere.security.jwt.InvalidClaimException:
CWWKS6023E: The audience [[SCORING]] of the provided JSON web token (JWT) is not listed as a trusted
audience in the [myJWTConsumer] JWT configuration. The trusted audiences are [[SCORES]].
com.ibm.ws.security.jwt.internal.ConsumerImpl 189" at ffdc_18.12.18_11.26.15.0.log

com.ibm.websphere.security.jwt.InvalidTokenException: CWWKS6031E: The JSON Web Token (JWT) consumer
[myJWTConsumer] cannot process the token string. CWWKS6023E: The audience [[SCORING]] of the provided
JSON web token (JWT) is not listed as a trusted audience in the [myJWTConsumer] JWT configuration. The
trusted audiences are
[[SCORES]].
```

Appendix 1 – CicsJwtBuilder

This appendix describes the CICSJwtBuilder class.

The **buildJwt** method is annotated with **@**CICSProgram to make it linkable from a CICS program. The method is seen as being a program called **BUILDJWT**.

```
      public class CICSJwtBuilder {

          @CICSProgram("BUILDJWT")
          public void buildJwt() throws CicsConditionException
          {
```

This method expects to retrieve the **JWT-REQ** container from the current channel, which is **L2LCHANNEL**. The record structure of JWT-REQ is described in the **JWTCLAIM** copybook.

```
       // Retrieve the current channel
        Channel ch = Task.getTask().getCurrentChannel();
       // Get the request container
        Container requestContainer = ch.getContainer("JWT-REQ");
```

An instance of JwtClaimsData, which represents the record structure of JWTCLAIM, is created with the byte array taken from the container.

```
        // Convert container data to an instance of JwtClaimsData
        JwtClaimsData jwtClaimsData = new JwtClaimsData(requestContainer.get());
```

The application and user specific claims are then retrieved from the Java helper class.

```
        // Extract the subject and audience claims
        String subject = jwtClaimsData.getSubject().trim();
        String audience = jwtClaimsData.getAudience().trim();
        String role = jwtClaimsData.getRole().trim();
        ArrayList<String> audiences = new ArrayList<String>();
        audiences.add(audience);
```

An empty instance of JwtTokenData is created, it is the record structure for the response container. The record structure is described in the **JWTTOKEN** copybook.

```
        // Create an instance of JwtTokenData
        JwtTokenData jwtTokenData = new JwtTokenData();
```

The following code snippet sets the return code to 0, which means the JWT generation succeeded. If the generation fails, the return code will be set to 1.

```
        // If returnCode equals 0 then everything went okay
      jwtTokenData.setBuildReturnCode(0);
```

The following snippet creates an instance of JwtBuilder based on the myJWTBuilder configuration (defined in server.xml). The jwtBuilder object will load the issuer value, the signing certificate reference and the token life time from the configuration element. After creating the jwtBuilder instance, the subject, audiences and role claims are set.

```
      // Create an instance of JwtBuilder based on "myJWTBuilder" configuration &
set subject,audiences
      JwtBuilder jwtBuilder =
JwtBuilder.create("myJWTBuilder").subject(subject).audience(audiences).claim("role
", role);
```

Then the JwtToken object is created from jwtBuilder.

```
      // Build token
      JwtToken jwtToken = jwtBuilder.buildJwt();
```

The String value of the token is extracted from jwtToken, and passed into the data structure for the response container.

```
      // Set JWT and length in response container
      String jwtString = jwtToken.compact();
      jwtTokenData.setJwtString(jwtString);
      jwtTokenData.setJwtStringLen(jwtString.length());
```

Finally, the response container **JWT-REP** is created. The response container is set with the byte array representing the JWTTOKEN data structure. The program then returns to GETQUOTE.

```
        // Create response container and put data inside
        Container replyContainer = ch.createContainer("JWT-REP");
        replyContainer.put(jwtTokenData.getByteBuffer());
```

For the Javadoc on the JwtBuilder class, see https://www.ibm.com/support/knowledgecenter/en/SS7K4U_liberty/com.ibm.websphere.javadoc.liberty.doc/com.ibm.websphere.appserver.api.jwt_1.1-javadoc/index.html

Appendix 2 – CicsJwtConsumer

This appendix describes the CICSJwtConsumer class.

The **consumeJwt** method is annotated with **@**CICSProgram to make it linkable from a CICS program. The method is seen as being a program called **CNSMJWT**.

```
public class CICSJwtConsumer {

    @CICSProgram("CNSMJWT")
    public void consumeJwt() throws CicsConditionException
    {
```

This method expects to retrieve the **JWT-REQ** container from the current channel, which is **L2LCHANNEL**. The record structure of **JWT-REQ** is described in the JWTTOKEN copybook.

```
       // Retrieve the current channel
        Channel ch = Task.getTask().getCurrentChannel();
       // Get the request container
        Container requestContainer = ch.getContainer("JWT-REQ");
```

An instance of JwtTokenData, which represents the record structure of JWTTOKEN, is created with the byte array taken from the container.

```
        // Convert container data to an instance of JwtTokenData
        JwtTokenData jwtTokenData = new JwtTokenData(requestContainer.get());
```

The GETSCORE program sends the HTTP Authorization header (without the Bearer prefix) to CNSMJWT in the container. The String value of the JWT needs to be extracted.

```
        // Extract the JWT string and trim trailing space
        String jwtString = jwtTokenData.getJwtString().trim();
```

An empty instance of JwtClaimsData is created, this is the record structure for the response container The record structure is described in the JWTCLAIM copybook.

```
        // Create an instance of JwtClaimsData
        JwtClaimsData jwtClaimsData = new JwtClaimsData();
```

The following code snippet sets the return code to 0, which means the JWT is valid. If the validation fails, the return code will be set to 1.

```
        // If returnCode equals 0 then everything went okay
      jwtClaimsData.setBuildReturnCode(0);
```

The following snippet creates an instance of JwtConsumer based on the myJWTConsumer configuration (defined in server.xml). The jwtConsumer object will load the issuer value, the signing certificate reference and the audiences list from the configuration element.

```
      // Create an instance of JwtConsumer based on "myJWTConsumer" configuration
      JwtConsumer jwtConsumer = JwtConsumer.create("myJWTConsumer");
```

Then the String representing the JWT is validated and parsed with jwtConsumer.

```
      // Validate and parse JWT
      JwtToken jwtToken = jwtConsumer.createJwt(jwtString);
```

The parsed claims are retrieved and passed into the data structure for the response container, so that the GETSCORE program can access the claims.

```
        // Retrieve the claims and set the values in the JwtClaimsData
        Claims claims = jwtToken.getClaims();
        jwtClaimsData.setAudience(claims.get("aud").toString());
        jwtClaimsData.setExpire(claims.getExpiration());
        jwtClaimsData.setIssuer(claims.getIssuer());
        jwtClaimsData.setSubject(claims.getSubject());
        jwtClaimsData.setRole(claims.get("role").toString());
```

Finally, the response container **JWT-REP** is created, and is set with the byte array representing the JWTCLAIM data structure. The program then returns to GETSCORE.

```
          // Create response container and put data inside
          Container replyContainer = ch.createContainer("JWT-REP");
          replyContainer.put(jwtTokenData.getByteBuffer());
```

For the Javadoc on the JwtConsumer class, see https://www.ibm.com/support/knowledgecenter/en/SS7K4U_liberty/com.ibm.websphere.javadoc.liberty.doc/com.ibm.websphere.appserver.api.jwt_1.1-javadoc/index.html.

More information

The following is a list of resources that might be useful. There are lots of resources available that cover aspects of the subject material in this document, but this list contains a few references we considered to be particularly useful or relevant.

RFC 7519 – JSON Web Token (JWT)
https://tools.ietf.org/html/rfc7519

JWT Introduction
https://jwt.io/introduction/

Liberty in IBM CICS
http://www.redbooks.ibm.com/redbooks/pdfs/sg248418.pdf

IBM CICS