ISSN: 2754-6705

Journal of Mathematical & Computer Applications

SCIENTIFIC Research and Community

Review Article

Open Access

Advanced Integration of Java EE Capabilities within CICS Liberty JVM Architecture

Chandra Mouli Yalamanchili

Software Development Engineering - Sr Advisor 2, USA

ABSTRACT

With the introduction of Liberty JVM within CICS (Customer Information Control System), IBM has enabled the deployment and execution of modern computing or APIs straight in CICS and executed them next to legacy applications written in COBOL, HLASM, and other languages. Using Java EE features brings enterprise-level capabilities like scalability, transactionality, security, and manageability. This integration of Java EE features also enhances the performance of Java applications deployed in CICS environments.

This paper explores different Java EE features, such as Context and Dependency Injection (CDI), Enterprise Java Beans (EJB), Java Persistence API (JPA), Java Transaction API (JTA), Java Message Service (JMS), and more, available to Java applications deployed into the CICS Liberty JVM environment. It also explores how the Liberty JVM supports each feature and how the integration with CICS works in each case.

***Corresponding author**

Chandra Mouli Yalamanchili, Software Development Engineering - Sr Advisor 2, USA.

Received: December 03, 2024; **Accepted:** December 09, 2024, **Published:** December 17, 2024

Keywords: IBM Mainframe, z/OS, CICS TS, Java EE, Jakarta EE, Liberty JVM, Java Applications in CICS, CICS Liberty

Introduction

While there are other options for running Java applications in the CICS transaction server ecosystem, Liberty is optimal for deploying enterprise Java applications. CICS Liberty is based on Eclipse Open J9, specifically optimized for the z/OS platform to take advantage of critical z/OS features such as high-performance I/O subsystems, efficient workload management, and robust security.

Java EE, now Jakarta EE, includes many features designed to simplify and standardize enterprise application development. Some of the core features include:

- CDI (Contexts and Dependency Injection)
- EJB (Enterprise JavaBeans)
- JPA (Java Persistence API)
- JMS (Java Message Service)
- Java EE security components.

When implemented on CICS Liberty, these features enable developers to build enterprise-grade applications with modern capabilities while retaining the reliability, scalability, and transaction processing strengths of CICS and z/OS. Integrating Java EE features into CICS Liberty allows organizations to modernize their existing applications, enabling them to take advantage of the rich ecosystem of Java-based tools, frameworks, and libraries. Additionally, it provides an opportunity to offload certain legacy workloads to Java-based workloads and reduce costs by taking advantage of zIIP processing.

This paper explores different Java EE features and understands how CICS Liberty implements these features, focusing on configuration procedures, best practices, and common challenges encountered during the integration process.

CICS Liberty Overview

CICS Liberty is a modular implementation (or profile) of WebSphere Application Server (WAS) technology. Liberty provides support for most of the Java EE features using configurable components known as features. The OSGi (Open Services Gateway initiative) framework and service registry enable Liberty to dynamically add or remove services or features from the service registry when there is a configuration change without the need to restart the JVM [1].

Some benefits of running Java EE applications using Liberty include easier developer adoption, easier deployment into CICS Liberty, low latency communication with existing CICS legacy applications, and reduced application cost by taking advantage of zIIP (z Integrated Information Processor) engines to process Java workloads [1].

Below is the depiction of how CICS Liberty looks like in execution:

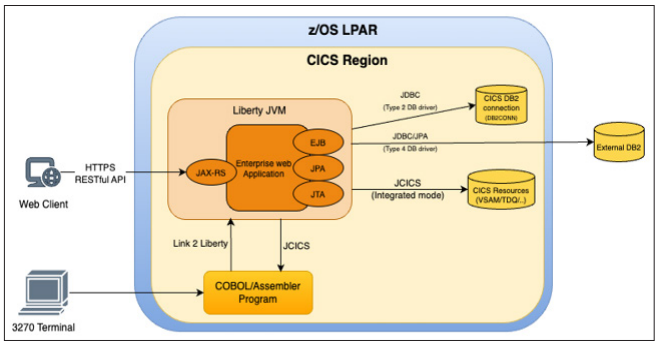

Figure 1: Depicting Liberty JVM Hosted in CICS and Implementing some of the Java EE Features [1].

As the picture illustrates, the Liberty JVM runs within the CICS address space and supports several Java EE features. Running within CICS improves unit of work management, taking benefit of access through CICS for DB2 or MQ communications. While it uses CICS integration to utilize CICS resources, the enterprise Java application can interface with external components through Java EE features like JPA and JMS.

Below are examples of a few Java EE features available in CICS Liberty that encompass several dependent features to provide Java EE support:

- Jakartaee-10.0 - Combination of Liberty features that support Jakarta EE 10.0 Platform. Support Java 11, 17, 21, and 23 versions [2].
- Jakartaee-8.0 - Combination of Liberty features that support Jakarta EE 8.0 Platform. Supports Java 8, 11, 17, 21, and 23 versions [2].
- Javaee-7.0 - Combination of Liberty features that support Java EE full 8.0 Platform. Supports Java 8, 11, 17, 21, and 23 versions [2].

The rest of the paper briefly explores different Java EE features and how applications can implement them using CICS Liberty.

Contexts and Dependency Injection (CDI)

CDI is a powerful Java EE feature that provides a common mechanism to inject components such as Enterprise JavaBeans (EJBs) or managed beans into other components such as JavaServer Pages (JSPs) or other EJBs [3].

Below are the main services provided by CDI:

- Contexts: CDI binds the lifecycle and interactions of stateful components to well-defined but extensible lifecycle contexts [4].
- Dependency Injection: CDI allows components to be injected into an application in a type-safe way and to choose at deployment time which implementation of a particular interface to inject [4].

CDI Example

- Enabling CDI in the CICS Liberty JVM setup:
- The CDI feature must be added to the server.xml file, as shown below

```
<featureManager>
<feature>cdi-2.0</feature>
</featureManager>
```

- The application needs to have either of the following to enable CDI discovery [3].

a) Explicit bean archives must contain a beans.xml file in the respective location based on archive type [3].

b) Implicit bean archives must contain one or bean classes with bean-defining annotations or session beans [3].

c) Liberty provides a configuration option to enable implicit bean archives, but it is recommended to use explicit bean archives in CICS to avoid CPU overhead. Liberty has to scan each application for the presence of CDI beans with an implicit bean archive [3].

- Code example - The code example below illustrates the usage of a simple CDI in the form of dependency injection for a RESTful API service [5].
- Since the MyBean class has the @ApplicationScoped annotation, only one instance is created and used for the application's lifetime [5].

```
@ApplicationScoped
public class MyBean {
int i=0;
public String sayHello() {
return "MyBean hello " + i++;
}
}
```

- The MyRestEndPoint class specifies the @RequestScoped annotation, which means CDI creates an instance for each API request and injects the same MyBean instance into each API request [5].

```
@RequestScoped
@Path("/hello")
public class MyRestEndPoint {
@Inject MyBean bean;
@GET
@Produces (MediaType.TEXT_PLAIN)
public String sayHello() {
        return bean.sayHello();
}
}
```

Enterprise JavaBeans (EJB)

Enterprise JavaBeans (EJB) is a Java API and a subset of the Java EE specification that simplifies the development of secure, transactional, and portable applications. EJBs contain the business logic of a server side application, and CICS Liberty fully supports EJBs, including the Lite subset [6, 7].

Enterprise beans can be either session beans or message-driven beans [7].

- A session bean contains business logic executed when a web service client requests a new request.
- A message driven bean combines the features of a session bean and a message listener, allowing a business component to receive messages asynchronously. Commonly, these are Java Message Service (JMS) messages [7].

Below are the features available in CICS Liberty to provide full support to EJB applications:

- ejbLite-3.1 - Enables the subset of EJB features that support the local session beans.
- mdb-3.1 - Enables the subset of EJB features related to message-driven bean.
- ejbHome-3.2 - Enables support for the EJBLocalHome interface.
- ejbRemote-3.2 - Enables support for remote EJB interfaces.
- ejbPersistentTimers-3.2 - Enables support for persistent EJB timers.
- ejb-3.2 - Enables complete EJB 3.2 support.

EJB Example

Below is a quick example showcasing the usage of EJB bean.

1. Define the EJB:

```
@Stateless // Marks this class as a Stateless EJB
public class GreetingService {
 public String getGreeting(String name) {
 return "Hello, " + name + "!";
 }
```

2. Inject and use the EJB bean to execute client requests:

```
public class GreetingServlet extends HttpServlet {
 @EJB
 private GreetingService greetingService; // EJB injection
 @Override
 protected void doGet(HttpServletRequest req, HttpServletResponse
resp) throws IOException {
 String name = req.getParameter("name");
 String greeting = greetingService.getGreeting(name != null ? name
: "World");
 resp.getWriter().write(greeting);
 }
}
```

Enabling EJB Feature in Liberty

- Like CDI, the EJB feature must be added to the feature manager component in server.xml to enable it in CICS Liberty.

```
<featureManager>
<feature>ejb-3.2</feature>
</featureManager>
```

EJB Transactionality & CICS UOW (Unit of Work)

EJBs support two types of transaction management: container-managed and bean-managed. Container managed transactions provide a transactional context for calls to bean methods and are defined using Java annotations or the deployment descriptor file ejbjar.xml. Bean-managed transactions are controlled directly through the Java Transaction API (JTA). In both cases, the CICS unit of work (UOW) remains subordinate to the outcome of the Liberty JTA transaction unless the CICS JTA is disabled through <cicsts_jta Integration="false"/> server.xml element [8].

The picture below illustrates how CICS Liberty supports different JTA transaction attributes [8].

Transaction Attribute	No JTA transaction	Pre-existing JTA transaction	Accessing a remote EJB with a pre-existing JTA transaction	Exception behavior
Mandatory	Throws exception EJBTransactionRequiredException.	Inherits the existing JTA transaction.	Throws exception com.ibm.websphere.csi.CSITransactionMandatoryException.	Rollback
Required This is the default transaction attribute.	EJB container creates new JTA transaction.	Inherits the existing JTA transaction.	Throws exception com.ibm.websphere.csi.CSITransactionRequiredException.	Rollback
RequiresNew	EJB container creates new JTA transaction.	Throws exception javax.ejb.EJBException.	EJB container creates a new JTA transaction that is managed by the remote server.	Rollback
Supports	Continues without a JTA transaction.	Inherits the existing JTA transaction.	Throws exception com.ibm.websphere.csi.CSITransactionSupportedException.	Rollback if called from JTA
NotSupported	Continues without a JTA transaction.	Suspends the JTA transaction but not the CICS UOW.	The remote server continues without a JTA transaction.	No rollback
Never	Continues without a JTA transaction.	Throws exception javax.ejb.EJBException.	The remote server continues without a JTA transaction.	No rollback

Figure 2: EJB Transaction Support [8].

Java Persistence API (JPA)

The Java Persistence API (JPA) is a Java EE feature that simplifies data persistence between Java objects and relational databases. It abstracts the direct SQL interaction through the ORM (Object-Relational Mapping) mechanism. Several ORM and JPA implementations, such as EclipseLink, Hibernate, etc., can connect applications to relational databases. CICS Liberty uses EclipseLink as the default JPA implementation.

CICS Liberty provides two options to interact with the relational databases, as mentioned below:

- JPA - This option uses the Java EE standard API to implement the database operations by abstracting the actual SQL queries or DB communications. It uses type 4 DB2 drivers that use the distributed service of DB2, providing better resiliency. It is platform-agnostic and can be migrated to other platforms if needed, as it does not rely on native code.
- JDBC - It's a low-level Java API that provides a client interface for database query execution. With this option, the Java application must handle DB operations such as SQL query construction, handling results, etc. As this is a low-level API, it gives more control to the application to build the queries that would improve its performance. The application must manage most of the functionality around database communication directly, like coming up with SQL queries to execute and connection management. Using JDBC API within CICS Liberty provides some advantages in the form of optimized z/OS connection pooling and native DB code, which offers better performance when interacting with DB2 on z/OS. There are two DB driver options to implement JDBC API:
- Type 4 mode: In this mode, the driver is independent of the DB client. There is no client, and communication happens through the DB server's TCP/IP name and port; it provides more resiliency. Java directly handles all TCP/IP command execution and routing to DB2 [9].
- Type 2 mode: In this mode, the driver depends on the DB2 client to establish connections via OS API calls to TCP/IP, and the DB2 client location must be in the library path. DB2 client uses the entries in the local catalogs and only requires the database name. In type 2 mode, the DB2 client reroutes and Sysplex mechanisms are used, and client-side connection concentration is unavailable [9].

JPA Implementation Example

Implementing the JPA feature in CICS Liberty would be similar to open Liberty. Below is a simple example of the same:

- Entity class - Java persistent object mapped to a table in a relational database.

```
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
 @Id
 private int id;
 private String name;

 // Getters and Setters
 public int getId() {
 return id;
 }
```

```
public void setId(int id) {
 this.id = id;
 }

 public String getName() {
 return name;
 }

 public void setName(String name) {
 this.name = name;
 }
}
```

- DAO (Data Access Object) is the abstraction layer between the application and the database functions.

```
public class EmployeeDAO {
 private EntityManagerFactory emf = Persistence.createEntity
ManagerFactory("CICSJPA");

 public void saveEmployee(Employee employee) {
 EntityManager em = emf.createEntityManager();
 em.getTransaction().begin();
 em.persist(employee);
 em.getTransaction().commit();
 em.close();
 }
}
```

- CICS Liberty configurations server.xml, need to add a JPA feature to the Liberty JVM, as shown below.
- Include the JPA feature.
- Data source element needs to be defined to access DB2 through data source.
- Need to provide the path to JDBC drivers as well through library element.

```
<server>
 <!-- Enable JPA Feature -->
 <featureManager>
 <feature>jpa-3.0</feature>
 </featureManager>

 <!-- Define the Datasource -->
  <dataSource id="DefaultDataSource" jndiName="jdbc/
DefaultDS">
 <jdbcDriver libraryRef="DB2Lib"/>
   <properties.db2.jcc databaseName="DBNAME"
serverName="hostname" portNumber="port" user="dbuser"
password="dbpassword"/>
 </dataSource>

 <library id="DB2Lib">
 <file path="/path-to-db2-driver/db2jcc4.jar"/>
 </library>
</server>
```

- Peristence.xml is the configuration file that provides the JPA API's configuration parameters and defines the DB's data source.

```
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
version="3.0">
 <persistence-unit name="CICSJPA">
 <class>com.example.Employee</class>
 <properties>
    <property name="jakarta.persistence.jdbc.url"
value="jdbc:db2://hostname:port/DBNAME"/>
   <property name="jakarta.persistence.jdbc.user"
value="dbuser"/>
  <property name="jakarta.persistence.jdbc.password"
value="dbpassword"/>
 <property name="jakarta.persistence.jdbc.driver" value="com.
ibm.db2.jcc.DB2Driver"/>
 </properties>
 </persistence-unit>
</persistence>
```

JDBC Implementation Example for CICS Liberty

- Java class with JDBC code:

```
public class EmployeeJDBC {
 public void fetchEmployee(int employeeId) {
 String sql = "SELECT NAME FROM EMPLOYEE WHERE
ID = ?";
 try (Connection conn = DriverManager.getConnection("jdbc:db2://
hostname:port/DBNAME", "dbuser", "dbpassword");
 PreparedStatement stmt = conn.prepareStatement(sql)) {

 stmt.setInt(1, employeeId);
 ResultSet rs = stmt.executeQuery();

 while (rs.next()) {
System.out.println("Employee Name: " + rs.getString("NAME"));
 }
 } catch (Exception e) {
 e.printStackTrace();
 }
 }
}
```

- Liberty server.xml configuration for type 4 mode doesn't use CICS DB2 connection for communication with DB2.

```
<server>
 <featureManager>
 <feature>jdbc-4.3</feature>
 </featureManager>

  <dataSource id="DefaultDataSource" jndiName="jdbc/
DefaultDS">
 <jdbcDriver libraryRef="DB2Lib"/>
   <properties.db2.jcc databaseName="DBNAME"
serverName="hostname" portNumber="50000" user="dbuser"
password="dbpassword"/>
 </dataSource>

 <library id="DB2Lib">
 <file path="/path-to-db2-driver/db2jcc4.jar"/>
 </library>
</server>
```

- Liberty server.xml configuration for type 2 mode, this can be done either through a DB2 CICS connection or a CICS aware connection, using the DB2CONN resource defined in CICS [10].
- The dataSource definition should have transactionality disabled to allow CICS to manage the transactions.
- Liberty JDBC using CICS DB2CONN resource:

```
<server>
        <featureManager>
                <feature>jdbc-4.2</feature>
        </featureManager>

        <library id="db2Type2Driver">
                <fileset dir="/usr/lpp/db2v12/jdbc/classes"
includes="db2jcc4.jar db2jcc_license_cisuz.jar"/>
                <fileset dir="/usr/lpp/db2v12/jdbc/lib"
includes="libdb2jcct2zos4_64.so"/>
        </library>

        <dataSource id="db2Type2" jndiName="jdbc/
db2Type2" transactional="false">
                <jdbcDriver libraryRef="db2Type2Driver"/>
                <properties.db2.jcc driverType="2"/>
                <connectionManager agedTimeout="0"/>
        </dataSource>

        <library id="global">
                <fileset dir="/usr/lpp/db2v12/jdbc/classes/"
includes=”db2jcc4.jar”/>
        </library>
</server>
```

- CICS JDBC using CICS DB2CONN resource:

```
<server>
        <featureManager>
                <feature>cicsts:jdbc-1.0</feature>
        </featureManager>

        <library id="db2Library">
                <fileset dir="/usr/lpp/db2v12/jdbc/classes"
includes="db2jcc4.jar db2jcc_license_cisuz.jar"/>
        <fileset dir="/usr/lpp/db2v12/jdbc/lib"
includes="libdb2jcct2zos4_64.so"/>
        </library>

        <cicsts_jdbcDriver libraryRef="db2Library"/>

        <cicsts_dataSource jndiName="jdbc/
cicsDb2DataSource"/>
</server>
```

As discussed, there are several options and different ways to implement Java applications with DB functionality using CICS Liberty. Using JPA API would help make the application platform agnostic and easier to maintain. In contrast, JDBC API gives more control over the implementation to build efficient SQL queries, manage connection pools, manage transactionality, etc.

Java Message Service (JMS)

Java Message Service (JMS) API allows enterprise Java applications to create, send, receive, and read messages. CICS Liberty provides support to multiple JMS clients through the features listed below [11]:

- WebSphere MQ JMS 2.0 (wmqJmsClient-2.0) client
- WebSphere Application Server JMS 2.0 (wasJmsClient-2.0) client

CICS Liberty also supports hosting the JMS server through the wasJmsServer-1.0 feature, but this paper further explores the JMS client support and details the configuration [11].

From Java applications (including CICS Liberty JVM), we can access IBM MQ through IBM MQ classes for Java or IBM MQ classes for JMS (the latter is the preferred approach) [12]. Application using MQ classes for JMS can connect to queue manager in either client or bindings mode [13].

- In client mode, IBM MQ classes for JMS connect to the queue manager over TCP/IP [13].
- In bindings mode, IBM MQ classes for JMS connect directly to the queue manager using the Java Native Interface (JNI) [13].

CICS Liberty JVM, running in standard mode, can connect to a queue manager using bindings or client-mode connections. Binding mode will work only when there is no CICS connection to the same queue manager from the same CICS regions. CICS Liberty, running in integrated mode, can connect to a queue manager only using a client mode connection [14].

Below are the steps to enable JMS in CICS Liberty JVM [15]:

- Add the wmqJmsClient-2.0 feature to the server.xml file to allow Liberty to load the IBM MQ bundles needed to define the JMS resources, such as connection factory and activation specification properties.

```
<featureManager>
<feature>wmqJmsClient-2.0</feature>
<feature>jndi-1.0</feature>
</featureManager>
```

- We need to add the zosTransaction-1.0 feature to the feature list if CICS Liberty will be connecting to the queue manager in binding mode.

```
<featureManager>
        <feature>zosTransaction-1.0</feature>
</featureManager>
```

- We must provide the zFS path to the IBM MQ RA (Resource Adapter).

```
<variable name="wmqJmsClient.rar.location" value="/path/to/
wmq/rar/wmq.jmsra.rar"/>
```

- We must add the JMS connection factory to server.xml to connect to the IBM MQ queue manager.
- Client mode example

```
<jmsConnectionFactory jndiName="jms/wmqCF"
connectionManagerRef="ConMgr6"> <properties.
wmqJms transportType="CLIENT" hostName="localhost"
port="1414" channel="SYSTEM.DEF.SVRCONN"
queueManager="QM1"/>
</jmsConnectionFactory>
<connectionManager id="ConMgr6" maxPoolSize="10"/>
```

- Binding mode example

```
<jmsConnectionFactory jndiName="jms/qm1"
connectionManagerRef="ConMgr6">          <properties.
wmqJms transportType="BINDINGS" queueManager="QM1"/>
</jmsConnectionFactory>
<connectionManager id="ConMgr6" maxPoolSize="10"/>
```

- Add queue definitions to server.xml

```
<jmsQueue id="jms/queue1" jndiName="jms/queue1">
<properties.wmqJms baseQueueName="QUEUE1"
baseQueueManagerName="QM1"/>
</jmsQueue>
```

- If we used binding mode, we must provide the native library to connect to the queue manager

```
<wmqJmsClient nativeLibraryPath="/opt/mqm/java/lib64"/>
```

- If the inbound messages need to be supported, we must add the MDB feature in server.xml (mdb-3.2) and the action spec in either client or binding mode to handle the incoming messages and deliver them to message-driven beans.

As with other features, CICS Liberty has some unique features for implementing JMS, and it needs special consideration to choose the right connection modes to achieve efficient performance from JMS usage to integrate with the MQ queue manager.

CICS Liberty & Security

While this paper doesn't explore the details of setting up security, below is an illustration of how different security components work to help secure the enterprise Java applications running in CICS Liberty JVM [16].

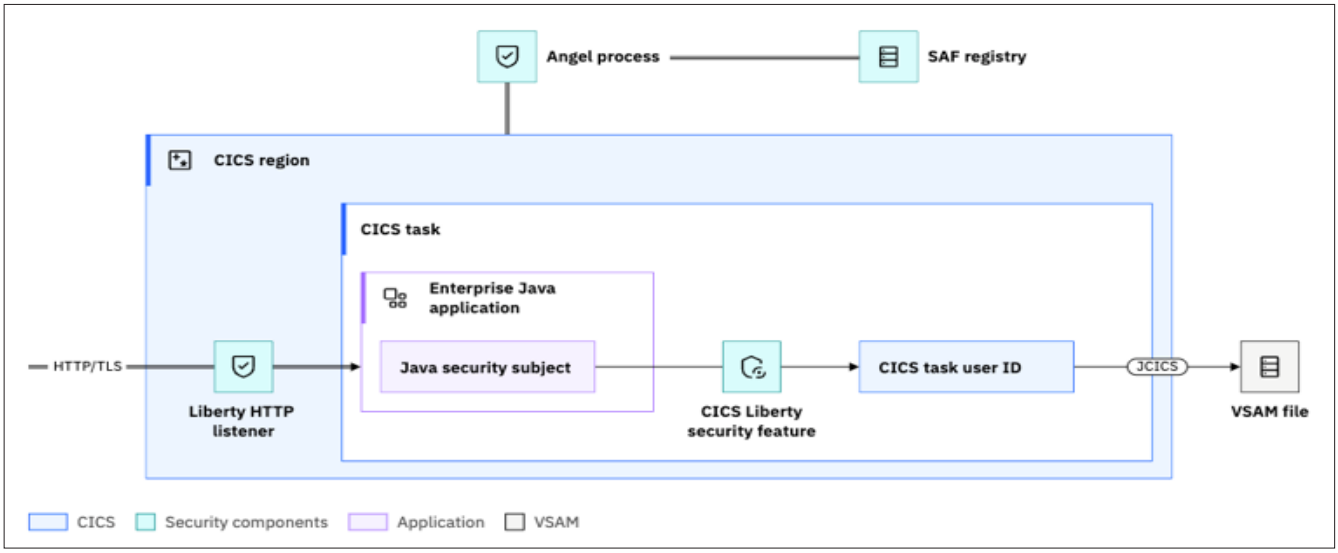

Figure 3: Illustrating Different Security Components that will help in Securing Enterprise Applications Running in CICS Liberty [16].

Below is the high level summary of key security features provided by CICS Liberty for Java EE applications [16]:

- **Role Based Access Control:** Ensures granular access control by assigning roles and permissions at the application level. Enterprise applications can implement role-based access control using features of EJB roles.
- **SAF (System Authorization Facility) Integration:** CICS Liberty can integrate with z/OS's SAF registry for identity management and resource protection using the `cicsts:security-1.0` feature. SAF registry integration through CICS security features relies on the angel service behind the scenes.
- **OAuth Based User Authentication:** Enables token based user authentication for API based applications.
- **Secure Session Management:** Manages user sessions securely with expiration policies and secure cookies.
- **TLS Encryption:** CICS Liberty supports the TLS by using the RACF key ring as a security manager to store the key stores needed. We can also use AT-TLS to secure the transport layer for client to server communication.
- **Secure Session Management:** Manages user sessions securely with expiration policies and secure cookies.

Conclusion

Integrating Java EE features into CICS Liberty enables organizations to modernize their mainframe applications, taking advantage of powerful features like CDI, EJB, and JPA while maintaining the stability and performance of their legacy systems. CICS and z/OS customization and optimal implementation of these features allow Java enterprise applications to benefit from the high processing power of CPUs much more efficiently. Following best practices, developers can build scalable, maintainable, and efficient enterprise applications in CICS Liberty.

References

1. IBM Corporation (2016) Modernizing Your Business Applications with IBM CICS and Liberty. IBM Redbooks https://www.redbooks.ibm.com/redpapers/pdfs/redp5334.pdf.
2. IBM Corporation (2024) CICS Liberty features. IBM https://www.ibm.com/docs/en/cics-ts/6.x?topic=server-liberty-features#features__cics-features.
3. IBM Corporation (2024) Context and Dependency Injection (CDI). IBM https://www.ibm.com/docs/en/cics-ts/6.x?topic=server-context-dependency-injection-cdi.
4. Oracle Corporation (2017) Overview of CDI. Oracle https://javaee.github.io/tutorial/cdi-basic002.html#GIWHL.

5. Open Liberty, IBM Corporation (2024) Context and Dependency Injection (CDI) Open Liberty https://openliberty.io/docs/latest/cdi-beans.html.
6. IBM Corporation (2024) Enterprise JavaBeans (EJB) IBM https://www.ibm.com/docs/en/cics-ts/6.x?topic=server-enterprise-javabeans-ejb.
7. Oracle Corporation (2024) Enterprise JavaBeans Technology. Oracle https://javaee.github.io/tutorial/overview008.html#BNACL.
8. IBM Corporation (2024) Using JTA transactions in EJBs. IBM https://www.ibm.com/docs/en/cics-ts/6.x?topic=ejb-using-jta-transactions-in-ejbs.
9. IBM Corporation (2024) 50 DB2 Nuggets #32 : Resource Info - Comparing JDBC Type 4 and type 2 connections. IBM https://www.ibm.com/support/pages/50-db2-nuggets-32-resource-info-comparing-jdbc-type-4-and-type-2-connections.
10. IBM Corporation (2024) Configuring database connectivity with JDBC. IBM https://www.ibm.com/docs/en/cics-ts/6.x?topic=server-configuring-database-connectivity-jdbc.
11. IBM Corporation (2024) Java Message Service (JMS). IBM https://www.ibm.com/docs/en/cics-ts/6.x?topic=server-java-message-service-jms.
12. IBM Corporation (2024) CICS and IBM MQ IBM https://www.ibm.com/docs/en/cics-ts/6.x?topic=fundamentals-cics-mq .
13. IBM Corporation (2024) Connection modes for IBM MQ classes for JMS. IBM https://www.ibm.com/docs/en/ibm-mq/9.4?topic=applications-connection-modes-mq-classes-jms.
14. IBM Corporation, "How it works: IBM MQ classes for JMS with CICS.", IBM 2024. Retrieved November 2024 from https://www.ibm.com/docs/en/cics-ts/6.x?topic=mq-how-it-works-classes-jms-cics
15. IBM Corporation (2024) Configuring a Liberty JVM server to support JMS. IBM https://www.ibm.com/docs/en/cics-ts/6.x?topic=server-configuring-liberty-jvm-support-jms.
16. IBM Corporation (2024) How it works: CICS Liberty security. IBM https://www.ibm.com/docs/en/cics-ts/6.x?topic=applications-how-it-works-cics-liberty-security.