CICS Transaction Server for z/OS
5.4

Java Applications in CICS

Note

Before using this information and the product it supports, read the information in "Notices" on page 267.

This edition applies to the IBM CICS® Transaction Server for z/OS® Version 5 Release 4 (product number 5655-Y04) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this PDF

This PDF tells you how to develop and use Java applications and enterprise beans in CICS. It is for experienced Java application programmers with little experience of CICS, and no need to know more about CICS than is necessary to develop and run Java programs. It is also for experienced CICS users and system programmers, who need to know about CICS requirements for Java support.

For details of the terms and notation used, see Conventions and terminology used in the CICS documentation in IBM Knowledge Center.

Date of this PDF

This PDF was created on 2024-01-04 (Year-Month-Date).

Chapter 1. CICS and Java

If you are planning to use Java™ in your enterprise, CICS provides the tools and runtime environment to develop and run Java applications in a Java Virtual Machine (JVM) that is under the control of a CICS region. Java workloads that run in a JVM server are eligible to run on a zEnterprise® Application Assist Processor (zAAP).

Java application development

You can create modular and reusable Java applications that comply with the OSGi Service Platform. These applications are easier to port between CICS and other platforms and OSGi provides granularity around managing dependencies and versions.

You can use the Java CICS (JCICS) API to write applications that access CICS services, such as reading from files or temporary storage queues. Java applications can link to other CICS applications, and can access data in DB2® and IMS. Java applications run in JVM servers.

You can create a web presentation layer for the application by using the web tools supplied with the Liberty profile. CICS can run JSP pages and web servlets in the same JVM server as the application.

Web services in an Axis2 JVM server

You can create Java web services to work with service providers and service requesters in a heterogeneous environment. Java web services run in a JVM server and the SOAP processing is performed by Apache CXF in the Liberty JVM server or the Apache Axis2 web services engine in the JVM server. You can also use standard Java APIs and annotations to create Java web services and perform data conversion, handle XML, or work with structured data.

Java connectivity to CICS

You can use JCA (Java Connector Architecture) to connect external Java applications to CICS through CICS Transaction Gateway. JCA is a related technology for calling CICS applications from an external Java environment. The CICS applications that are called in this way can be implemented in Java, or in any other supported language.

Java support in CICS

CICS provides the tools and runtime environment to develop and run Java enterprise applications in a Java Virtual Machine (JVM) that is under the control of a CICS region. Java applications can interact with CICS services and applications written in other languages.

Java on z/OS provides comprehensive support for running Java applications. CICS uses the IBM® 64-bit SDK for z/OS, Java Technology Edition, Version 7, Version 7 Release 1 or Version 8. Some Liberty features require specific Java versions, and these are called out in the Liberty features table.

The SDK contains a Java Runtime Environment that supports the full set of Java APIs and a set of development tools. To help increase general purpose processor productivity and contribute to lowering the overall cost of computing for z/OS Java technology-based applications, special processors are available in certain z Systems® hardware. The IBM zEnterprise Application Assist Processor (zAAP) can provide additional processor capacity to run eligible Java workloads, including Java workloads in CICS. You can find more information about Java on the z/OS platform and download the 64-bit version of the SDK at Java Standard Edition Products on z/OS.

CICS provides a JVM server runtime environment for Java application development. You can develop applications using the IBM CICS SDK for Java, Maven modules, or Gradle modules.

JVM server

The JVM server is the strategic runtime environment for Java applications in CICS. A JVM server can handle many concurrent requests for different Java applications in a single JVM. Use of a JVM server reduces the number of JVMs that are required to run Java applications in a CICS region. To use a JVM server, Java applications must be threadsafe and must comply with the OSGi or Java EE specifications. JVM server provides the following benefits:

- Eligible Java workloads can run on specialty engine processors, reducing the cost of transactions.
- Different types of work such as threadsafe Java programs and web services, can run in a JVM server.
- Application life cycle can be managed in the OSGi framework, without restarting the JVM server.
- Java applications that are packaged using OSGi can be ported more easily between CICS and other platforms.
- Java EE applications can be deployed into the Liberty JVM server.

  Note: OSGi applications in CICS can be installed in a Liberty JVM server but cannot use any of the Liberty services or features as they are not supported.

IBM CICS SDK for Java

CICS Explorer® is a freely available download for Eclipse-based Integrated Development Environments (IDEs). The IBM CICS SDK for Java that is included with CICS Explorer provides support for developing and deploying applications that comply with the OSGi Service Platform specification.

The OSGi Service Platform provides a mechanism for developing applications using a component model and deploying those applications into a framework as OSGi bundles. An *OSGi bundle* is the unit of deployment for an application component and contains version control information, dependencies, and application code. The main benefit of OSGi is that you can create applications from reusable components that are accessed only though well-defined interfaces called *OSGi services*. You can also manage the life cycle and dependencies of Java applications in a granular way.

The IBM CICS SDK for Java allows development of Java applications for any supported release of CICS. The SDK includes the Java CICS library (JCICS) to access CICS services along with examples to get started with developing applications for CICS. You can also use the tool to convert existing Java applications to OSGi.

The IBM CICS SDK for Java EE and Liberty is included as an option with CICS Explorer and supports packaging of Liberty applications into CICS bundles that can be deployed to CICS.

Maven and Gradle modules

As an alternative to the IBM CICS SDK for Java, you can define your projects as Maven or Gradle modules, and express dependencies by referencing the coordinates of the artifacts provided by CICS on Maven Central. Maven and Gradle are popular build tools that support dependency management. Using them has the following benefits:

- Easy management of dependencies. Java developers can easily add the required versions of the Java CICS APIs and the CICS annotation processor to the Java dependencies with just a few lines of configurations.
- More freedom when choosing the development environment. Maven and Gradle support most Java IDEs, such as Eclipse, IntelliJ IDEA, and Visual Studio Code. Java developers can write application code in a familiar IDE.
- Better integration into a build toolchain. Maven and Gradle are, in nature, build tools, and they integrate smoothly with other automation tools such as Jenkins and Travis CI. The dependencies defined through Maven or Gradle are integrated into the build toolchain automatically.

The following artifacts are available on Maven Central:

The Java CICS class library (JCICS)
Provides the **EXEC CICS** API support for Java applications in CICS TS.

The CICS annotations library and the CICS annotation processor
Provides support that enables CICS programs to invoke Java applications in a Liberty JVM server.

A bill of materials (BOM)
Defines the versions of the other artifacts to ensure that they are at the same CICS TS level.

For more information about the artifacts and how to use them, see Developing applications using other tools.

These artifacts, along with other existing dependencies on Maven Central such as the following, contain the APIs that are provided in CICS JVM servers.

- org.osgi for OSGi framework
- net.wasdev.maven.tools.targets for Liberty APIs, SPIs, and Java EE specifications

You can either use these artifacts directly from Maven Central, or have them referenced as whitelisted dependencies from an enterprise repository by using repository managers such as Artifactory or Nexus.

The OSGi Service Platform

The OSGi Service Platform provides a mechanism for developing applications by using a component model and deploying those applications into an OSGi framework. The OSGi architecture is separated into a number of layers that provide benefits to creating and managing Java applications.

The OSGi framework is at the core of the OSGi Service Platform specification. CICS uses the Equinox implementation of the OSGi framework. The OSGi framework is initialized when a JVM server starts. Using OSGi for Java applications provides the following major benefits:

- New Java applications, and new versions of Java applications, can be deployed into a live production system without having to restart the JVM, and without impacting the other Java applications deployed in that JVM.
- Java applications are more portable, easier to re-engineer, and more adaptable to changing requirements.
- You can follow the Plain Old Java Object (POJO) programming model, giving you the option of deploying an application as a set of OSGi bundles with dynamic life cycles.
- You can more easily manage and administer application bundle dependencies and versions.

The OSGi architecture has the following layers:

- Modules layer
- Life cycle layer
- Services layer

Modules layer

The unit of deployment is an OSGi bundle. The modules layer is where the OSGi framework processes the modular aspects of a bundle. The metadata that enables the OSGi framework to do this processing is provided in a bundle manifest file.

One key advantage of OSGi is its class loader model, which uses the metadata in the manifest file. There is no global class path in OSGi. When bundles are installed into the OSGi framework, their metadata is processed by the module layer and their declared external dependencies are reconciled against the exports and version information declared by other installed modules. The OSGi framework works out all the dependencies and calculates the independent required class path for each bundle. This approach resolves the shortcomings of plain Java class loading by ensuring that the following requirements are met:

- Each bundle provides visibility only to Java packages that it explicitly exports.
- Each bundle declares its package dependencies explicitly.
- Packages can be exported at specific versions, and imported at specific versions or from a specific range of versions.

- Multiple versions of a package can be available concurrently to different clients.

Life cycle layer

The bundle life cycle management layer in OSGi enables bundles to be dynamically installed, started, stopped, and uninstalled, independently from the life cycle of the JVM. The life cycle layer ensures that bundles are started only if all their dependencies are resolved, reducing the occurrence of `ClassNotFoundException` exceptions at run time. If there are unresolved dependencies, the OSGi framework reports the problem and does not start the bundle.

Each bundle can provide a bundle activator class, which is identified in the bundle manifest, that the framework calls as part of bundle start and stop events.

Services layer

The services layer in OSGi intrinsically supports a service-oriented architecture through its non-durable service registry component. Bundles publish services to the service registry, and other bundles can discover these services from the service registry. These services are the primary means of collaboration between bundles. An OSGi service is a Plain Old Java Object (POJO), published to the service registry under one or more Java interface names, with optional metadata stored as custom properties (name/value pairs). A discovering bundle can look up a service in the service registry by an interface name, and can potentially filter the services that are being looked up based on the custom properties.

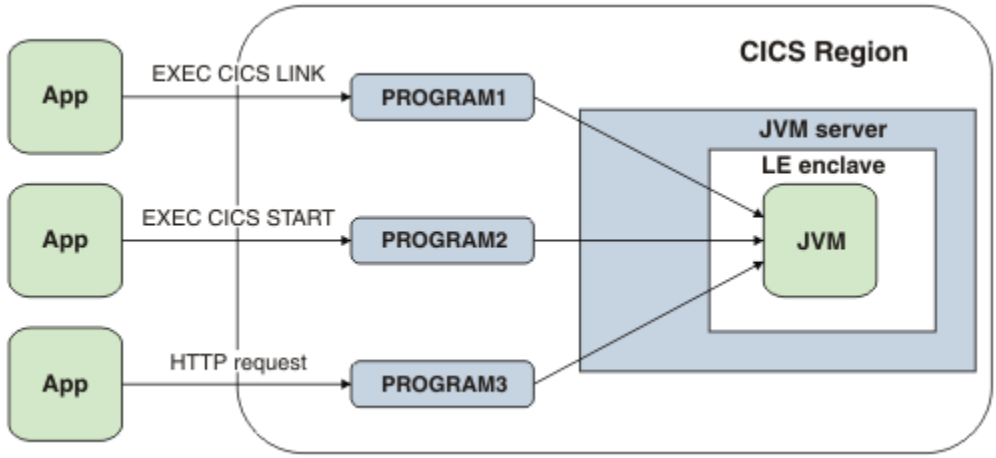
Services are fully dynamic and typically have the same life cycle as the bundle that provides them.

JVM server runtime environment

A *JVM server* is a runtime environment that can handle many concurrent requests for different Java applications in a single JVM. You can use a JVM server to run threadsafe Java applications in an OSGi framework, run web applications in a Liberty profile, and process web service requests in the Axis2 web services engine.

A JVM server is represented by the JVMSERVER resource. When you enable a JVMSERVER resource, CICS requests storage from MVS™, sets up a Language Environment® enclave, and launches the 64-bit JVM in the enclave. CICS uses a JVM profile that is specified on the JVMSERVER resource to create the JVM with the correct options. In this profile, you can specify JVM options and system properties, and add native libraries; for example, you can add native libraries to access DB2 or IBM MQ from Java applications.

One of the advantages of using JVM servers is that you can run many requests for different applications in the same JVM. In the following diagram, three applications are calling three Java programs in a CICS region concurrently using different access methods. Each Java program runs in the same JVM server.



Java applications

To run a Java application in a JVM server, it must be threadsafe and packaged as one or more OSGi bundles in a CICS bundle. The JVM server implements an OSGi framework in which you can run OSGi bundles and services. The OSGi framework registers the services and manages the dependencies and versions between the bundles. OSGi handles all the class path management in the framework, so you can add, update, and remove Java applications without stopping and restarting the JVM server.

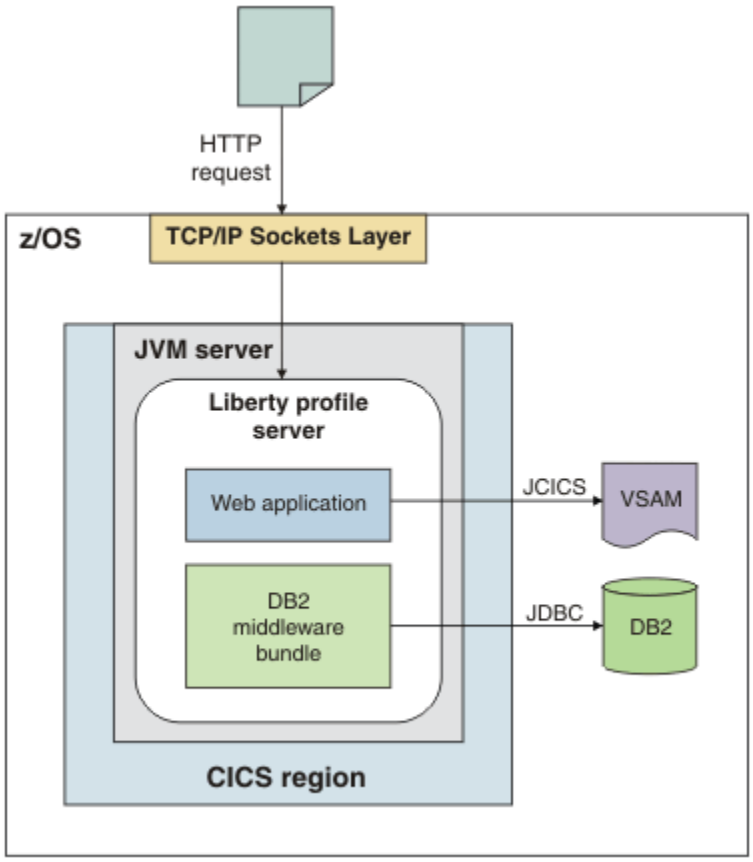
The unit of deployment for a Java application that is packaged using OSGi is a CICS bundle. The BUNDLE resource represents the application to CICS and you can use it to manage the lifecycle of the application. The IBM CICS SDK for Java provides support for deploying OSGi bundles in a CICS bundle project to zFS.

To access the Java application from outside the OSGi framework, use a PROGRAM resource to identify the JVM server in which the application is running and the name of the OSGi service. The OSGi service points to the CICS main class.

For more information about using the OSGi framework in a JVM server, see "Java applications that comply with OSGi" on page 12.

Java web applications

In addition to running Java applications in an OSGi framework, the JVM server also supports running WebSphere® Application Server Liberty. Liberty is a lightweight application server for running web applications. Web applications can use JCICS to access resources and services in CICS, and to access data in DB2. Applications running in Liberty are accessed through the TCP/IP sockets layer in z/OS rather than through web support in CICS.



Java web applications can follow the Liberty model for deployment, where developers can deploy web archive (WAR) files or enterprise application archive (EAR) files directly into the drop-in directory of Liberty, or use the CICS application model of creating CICS bundles. CICS bundles provide lifecycle

management and can package an application that contains many components, including OSGi bundles and WAR files, together.

To access OSGi bundles from a web application, you must deploy your application as an Enterprise Bundle Archive (EBA) file. To develop EBAs, you can use Rational® Application Developer, or you can use a combination of the Eclipse IDE, the IBM CICS SDK for Java, and WebSphere Application Server Developer Tools for Eclipse. The latter set of tools is free to use but, apart from the IBM CICS SDK for Java, IBM support is not available for them.

For more information about using Liberty, see "Java applications in a Liberty JVM server" on page 14.

Web services

You can use a JVM server to run the SOAP processing for web service requester and provider applications. If a pipeline uses Axis2, a SOAP engine that is based on Java, the SOAP processing occurs in a JVM server. The advantage of using a JVM server for web services is that you can offload the work to a zAAP processor.

For more information about using a JVM server for web services, see "Java web services" on page 16.

JVM profiles

JVM profiles are text files that contain Java launcher options and system properties, which determine the characteristics of JVMs. You can edit JVM profiles using any standard text editor.

When CICS receives a request to run a Java program, the name of the JVM profile is passed to the Java launcher. The Java program runs in a JVM, which was created using the options in the JVM profile.

CICS uses JVM profiles that are in the z/OS UNIX System Services directory specified by the JVMPROFILEDIR system initialization parameter. This directory must have the correct permissions for CICS to read the JVM profiles.

Sample JVM profiles

CICS includes several sample JVM profiles to help you configure your Java environment. They are customized during the CICS installation process. These files are used by CICS as defaults or for system programs.

A JVM profile lists the options that are used by the CICS launcher for Java. Some of the options are specific to CICS and others are standard for the JVM runtime environment. For example, the JVM profile controls the initial size of the storage heap and how far it can expand. The profile can also define the destinations for messages and dump output produced by the JVM. The JVM profile is named in the JVMPROFILE attribute in a JVMSERVER resource definition.

You can copy the samples and customize them for your own applications. The sample JVM profiles supplied with CICS are in the directory `/usr/lpp/cicsts/cicsts54/JVMProfiles` on z/OS UNIX.

Note: If you are unable to find the sample JVM profiles in `/usr/lpp/cicsts/cicsts54/JVMProfiles`, or the directory does not exist, then it is likely you have not run the DFHIJVMS job to populate the directories and create the JVM profiles. You can find this job in SDFHINST and it is documented in the Program Directory for CICS Transaction Server for z/OS. See Program Directories for more information.

Copy the samples from the installation directory to the directory that you specified in the **JVMPROFILEDIR** system initialization parameter. The sample JVM profiles in the installation location are overwritten if you apply an APAR that includes changes to these files. To avoid losing your modifications, always copy the samples to a different location before adding your own application classes or changing any options.

The following table summarizes the key characteristics of each sample JVM profile.

Table 1. Sample JVM profiles supplied with CICS	
JVM profile	**Characteristics**
DFHJVMAX.jvmprofile	The supplied sample profile for an Axis2 JVM server. The JVM profile is specified on the JVMSERVER resource. CICS uses DFHJVMAX.jvmprofile to initialize the JVM server.
DFHJVMST.jvmprofile	The supplied sample profile for a JVM server for a Security Token Service. The JVM profile is specified on the JVMSERVER resource. CICS uses DFHJVMST.jvmprofile to initialize the JVM server.
DFHOSGI.jvmprofile	The supplied sample profile for an OSGi JVM server. The JVM profile is specified on the JVMSERVER resource. CICS uses DFHOSGI.jvmprofile to initialize the JVM server.
DFHWLP.jvmprofile	The supplied sample profile for a Liberty JVM server. The JVM profile is specified on the JVMSERVER resource. CICS uses DFHWLP.jvmprofile to initialize the Liberty JVM server.

Structure of a JVM

JVMs that run under CICS use a set of classes and class paths that are defined in JVM profiles and use 64-bit storage. Each JVM runs in a Language Environment enclave that you can tune to make the most efficient use of MVS storage.

For further information about the IBM 64-bit SDK for z/OS, Java Technology Edition, see z/OS User Guide for IBM SDK, Java Technology Edition, Version 7 or z/OS User Guide for IBM SDK, Java Technology Edition, Version 8.

Classes and class paths in JVMs

A JVM running under CICS can use different types of class or library files: primordial classes (system and standard extension classes), native C DLL library files, and application classes.

The JVM recognizes the purpose of each of these components, determines how to load them, and determines where to store them. The class paths for a JVM are defined by options in the JVM profile, and (optionally) are referenced in JVM properties files.

- *Primordial classes* are the z/OS JVM code that provide the base services in the JVM. Primordial classes can be categorized as system classes and standard extension classes.
- *Native C dynamic link library (DLL) files* have the extension `.so` in z/OS UNIX. Some libraries are required for the JVM to run, and additional native libraries can be loaded by application code or services. For example, the additional native libraries might include the DLL files to use the DB2 JDBC drivers.
- *Application classes* are the classes for applications that run in the JVM, and include classes that belong to user-written applications. Java application classes also include those supplied by IBM or by other vendors, to provide services that access resources, such as the JCICS interfaces classes, JDBC and JNDI, which are not included in the standard JVM setup for CICS. When Java application classes are loaded into the class cache the are kept and can be reused by other applications running in the same JVM.

The class paths on which classes or native libraries can be specified are the library path, and the standard class path.

- The *Library path* specifies the native C dynamic link library (DLL) files that are used by the JVM, including the files required to run the JVM and additional native libraries loaded by application code or services. Only one copy of each DLL file is loaded, and all the JVMs share it, but each JVM has its own copy of the static data area for the DLL.

The base library path for the JVM is built automatically using the directories specified by the **USSHOME** system initialization parameter and the **JAVA_HOME** option in the JVM profile. The base library path is not visible in the JVM profile. It includes all the DLL files required to run the JVM and the native libraries used by CICS. You can extend the library path using the **LIBPATH_SUFFIX** option or the **LIBPATH_PREFIX** option. **LIBPATH_SUFFIX** adds items to the end of the library path, after the IBM-supplied libraries. **LIBPATH_PREFIX** adds items to the beginning, which are loaded in place of the IBM-supplied libraries if they have the same name. You might have to do this for problem determination purposes.

Compile and link with the LP64 option any DLL files that you include on the library path . The DLL files supplied on the base library path and the DLL files used by services such as the DB2 JDBC drivers are built with the LP64 option.

- The *Standard class path* must not be used for OSGi enabled JVM servers because the OSGi framework automatically determines the class path for an application from information in the OSGi bundle that contains the application. The standard class path is retained for use by JVM servers that are not configured for OSGi (for example the Axis2 environment in CICS). For exceptional scenarios, such as Axis2, in which the standard class path is used, you can use a wildcard suffix on the class path entries to specify all JAR files in a particular directory.

  CICS also builds a base class path automatically for the JVM using the `/lib` subdirectories of the directories specified by the **USSHOME** system initialization parameter. This class path contains the JAR files supplied by CICS and by the JVM. It is not visible in the JVM profile.

You do not have to include the system classes and standard extension classes (the primordial classes) on a class path, because they are already included on the boot class path in the JVM.

Storage heap in JVMs

The runtime JVM storage is managed by a single 64-bit storage heap.

The heap for each JVM is allocated from 64-bit storage in the Language Environment enclave for the JVM. The size of each heap is determined by options in the JVM profile.

The single storage heap is known as the *heap*, or sometimes as the *garbage-collected heap*. Its initial storage allocation is set by the **-Xms** option in a JVM profile, and its maximum size is set by the **-Xmx** option.

You can tune the size of a heap to achieve optimum performance for your JVMs. See Tuning JVM server heap and garbage collection.

Where JVMs are constructed

When a JVM is required, the CICS launcher program for JVMs requests storage from MVS, sets up a Language Environment enclave, and launches the JVM in the Language Environment enclave. Each JVM is constructed in its own Language Environment enclave, to ensure isolation between JVMs running in parallel.

The Language Environment enclave is created using the Language Environment preinitialization module, CELQPIPI, and the JVM runs as a z/OS UNIX process. The JVM therefore uses MVS Language Environment services rather than CICS Language Environment services. The storage used for a JVM is MVS 64-bit storage, obtained by calls to MVS Language Environment services. This storage resides in the CICS address space, but is not included in the CICS dynamic storage areas (DSAs).

The Language Environment enclave for a JVM can expand, depending on the storage requirements of the JVM. The Language Environment runtime options used by CICS for a Language Environment enclave control the initial size of, and incremental additions to, the Language Environment enclave heap storage.

You can tune the runtime options that CICS uses for a Language Environment enclave, so that the amount of storage CICS requests for the enclave is as close as possible to the amount of storage specified by your JVM profiles. You can therefore make the most efficient use of MVS storage. For more information about tuning storage, see Language Environment enclave storage for JVMs.

JVMs and the z/OS shared library region

The shared library region is a z/OS feature that enables address spaces to share dynamic link library (DLL) files.

This feature enables your CICS regions to share the DLLs that are needed for JVMs, rather than each region having to load them individually. This can greatly reduce the amount of real storage used by MVS, and the time it takes for the regions to load the files.

The storage that is reserved for the shared library region is allocated in each CICS region when the first JVM is started in the region. The amount of storage that is allocated is controlled by the **SHRLIBRGNSIZE** parameter in z/OS. For more information about tuning the amount of storage that is allocated for the shared library region, see Tuning the z/OS shared library region.

CICS task and thread management

CICS uses the open transaction environment (OTE) to run JVM server work. Each task runs as a thread in the JVM server and is attached by using a T8 TCB. A major benefit of using OSGi is that applications in an OSGi framework can use an `ExecutorService` to create threads that run extra tasks in CICS asynchronously. CICS takes special measures to deal with runaway tasks.

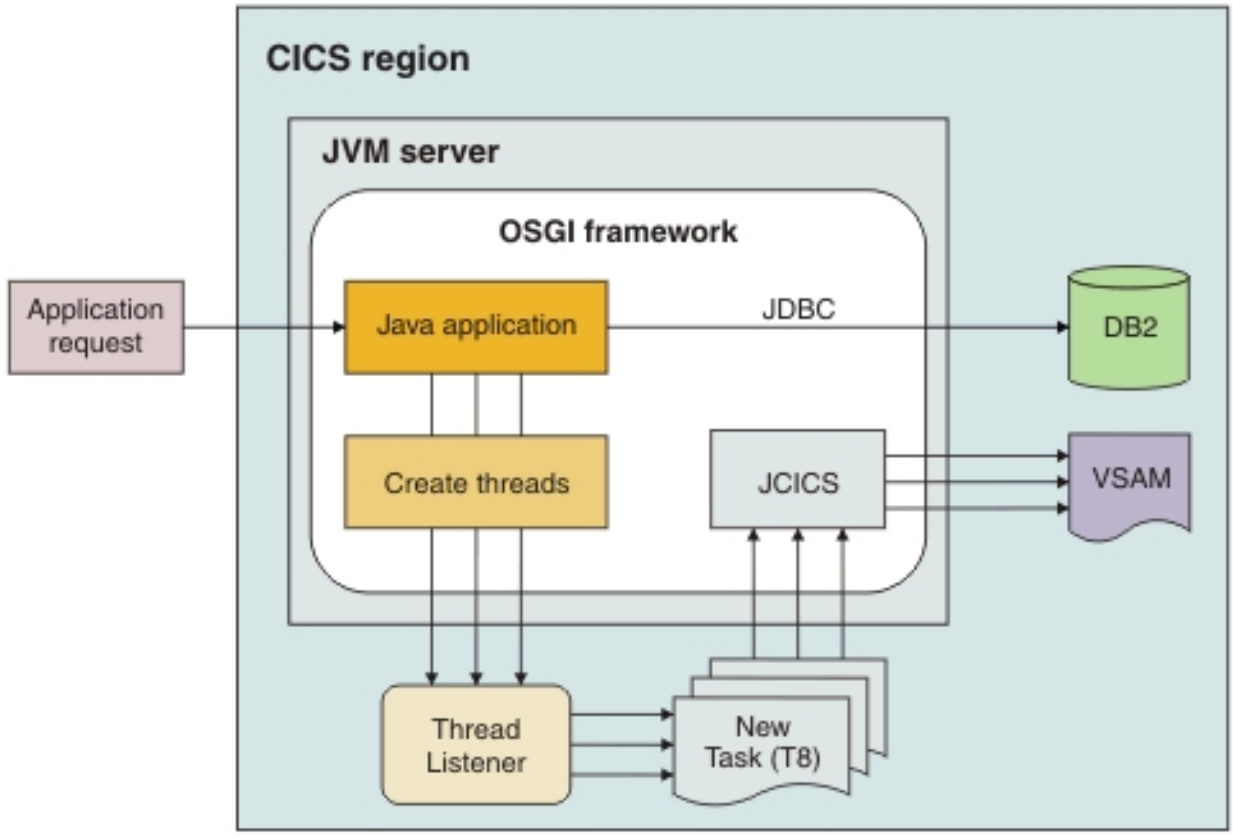
When CICS enables a JVM server, the JVM server runs on a Language Environment process thread. This thread is a child of the TP TCB. Every CICS task is attached to a thread in the JVM by using a T8 TCB. You can control how many T8 TCBs are available to the JVM server by setting the `THREADLIMIT` attribute on the JVMSERVER resource.

The T8 TCBs that are created for the JVM server exist in a virtual pool and cannot be reused by another JVM server that is running in the same CICS region. The maximum number of T8 TCBs that can exist in a CICS region across all JVM servers is 2000 and the maximum for a specific JVM server is 256.

Multithreaded applications

Java applications that are running in an OSGi framework can also start CICS tasks asynchronously by using an `ExecutorService` OSGi service. The JVM server registers the `ExecutorService` as an OSGi service on startup. The `ExecutorService` automatically uses an implementation that is supplied by CICS that creates threads that can use the JCICS API to access CICS services. This approach means the application does not have to use specific JCICS API methods to create threads. However, an application can also use the `CICSExecutorService` to run work on a separate CICS capable thread.

When the JVM server is enabled, it starts the CJSL transaction to create a long-running task that is called the JVM server listener. This listener waits for new thread requests from the application and runs the CJSA transaction to create CICS tasks that are dispatched on a T8 TCB. This process is shown in the following diagram:



In advanced scenarios, an application can use the OSGi service to run many threads asynchronously. These threads all have access to CICS services through JCICS and run under T8 TCBs.

Execution keys for JVM servers

A Java program must use a JVM that is running in the correct execution key. JVM servers run in CICS key. To use a JVM server, the PROGRAM resource for the Java program must have the EXECKEY attribute set to CICS. CICS uses a T8 TCB to run the JVM and obtains MVS storage in CICS key.

Runaway tasks

The CICS JVM server infrastructure supports use of the task runaway detection mechanism. Unlike traditional CICS tasks, a task running Java on a T8 TCB cannot be terminated without consequences to other workload in the same JVM. Language Environment and the JVM server run in a POSIX-compliant environment, which mandates that if a TCB/Thread is terminated, the parent process is also terminated. In turn, all child processes are terminated abruptly - and cause all tasks in the JVM to fail immediately.

A task running in a JVM server that exceeds the modified RUNAWAY interval experiences a more controlled termination process. This differs from the traditional CICS behavior and you should evaluate whether you want runaway intervals to apply to your Java tasks, or what value to set.

JVMSERVER controlled runaway processing

When a task running Java experiences a runaway interval condition, the JVMSERVER intercepts the condition and triggers a DISABLE PHASEOUT. New work is prevented from entering the JVM and existing work is left to drain. Subsequently, should the task complete its processing, the JVMSERVER re-enables and becomes available for new requests. In many cases if a task running Java exceeds the runaway interval value, it is likely to be a bad application, such as a tightly looping application and prevents successful PHASEOUT/RECYCLE of the JVMSERVER. When an application is detected, the runaway timer triggers again after another interval and the JVMSERVER DISABLE PHASEOUT is escalated to a

JVMSERVER DISABLE PURGE. Remaining tasks are subject to PURGE processing and in most cases are terminated. If further runaway intervals are exceeded, the JVMSERVER DISABLE escalates to FORCEPURGE and ultimately KILL - until all running tasks are forcefully terminated. The JVMSERVER recycles back to the ENABLED state ready for new requests. If the JVMSERVER had to escalate as far as a DISABLE KILL request, it is prudent to recycle CICS at the earliest opportunity.

Modified runaway interval value

A runaway condition for a task that is running in a JVM server can cause temporary availability problems for the whole JVM server. For this reason, CICS modifies the runaway interval value that was configured, by multiplying it by a factor of 10 (up to a maximum value of 45 minutes). This new value is the effective runaway interval. This higher runaway interval reduces the possibility of a runaway condition being detected for an inefficient (but otherwise working) application. For example, if the transaction definition specifies RUNAWAY=SYSTEM, and the ICVR system initialization parameter indicates a default limit of 5000 milliseconds, then the effective runaway interval for that task when it runs in a JVM server is 50000 milliseconds.

Setting the runaway interval value

By default the CJSA transaction definition that is used for Liberty JVM servers and for work in an OSGi JVM server started from the CICSExecutorService has runaway detection active and set to the system interval. If you do not want runaway intervals to apply to these tasks, you can run work under your own transaction definitions with the runaway interval set to 0, or another value of your choice. Liberty workload is typically controlled by URIMAPs, while the CICSExecutorService provides the CICSTransactionRunnable and CICSTransactionCallable interfaces to allow customized transaction definitions to be used.

Shared class cache

Shared class cache provides a mechanism for multiple JVMs to share Java application classes stored in a single cache. The IBM SDK for z/OS supports shared class cache. Class cache can be used with OSGi JVM servers, and with non-OSGi JVM servers such as Apache Axis2.

Java 7, not CICS, provides support for using the class cache function with JVM servers. Therefore, you cannot use CICS SPI or CEMT commands to enable or disable a JVM server class cache. To enable or disable a JVM server class cache, you use JVM command line parameters. You also use a JVM command line parameter to set the class cache size.

The following components (files, objects, variables, and compiled classes) are *not* loaded into class cache:

- Native C DLL files specified on the JVM profile library path. These are not loaded because a single copy of each DLL file is used by all JVMs that require access to the file.
- Application objects and variables. These are not loaded because working data is stored in the JVMs.
- Just-in-time (JIT) compiled classes. These are not loaded but are stored in the JVMs because the compilation process for different workloads can vary.

Enabling the cache

By default, class cache is disabled in the JVM. To enable class cache you use a JVM command line parameter, for example:

```
-Xshareclasses:name=cics.&APPLID;
```

Specifying cache size

To specify class cache size, you use a JVM command line parameter. For example, the following parameter sets the size to 20M:

```
-Xscmx20M
```

Important: If your cache was created using a different `compressedRefs` setting than that currently in effect within the JVM, you will receive a message indicating you should use the correct JVM level. In such situations, you should check that the `compressedRefs` settings in effect when you created your cache, are the same as those in effect when your JVM was created.

For more information about Java class data sharing see Class data sharing between JVMs in IBM SDK, Java Technology Edition, Version 7.

Java applications that comply with OSGi

CICS includes the Equinox implementation of the OSGi framework to run Java applications that comply with the OSGi specification in a JVM server.

The OSGi Service Platform specification, as described in "The OSGi Service Platform" on page 3, provides a framework for running and managing modular and dynamic Java applications. The default configuration of a JVM server includes the Equinox implementation of an OSGi framework. Java applications that are deployed on the OSGi framework of a JVM server benefit from the advantages of using OSGi and the qualities of service that are inherent in running applications in CICS.

You might want to use Java applications for any of the following reasons:

- You want to create Java workloads that can run on a zAAP to reduce the cost of transactions.
- You have experience of writing Java applications that use OSGi on other platforms and want to create Java applications in CICS.
- You want to provide Java applications as a set of modular components that can be reused and updated independently, without affecting the availability of applications and the JVM in which they are running.

To effectively develop, deploy, and manage Java applications that comply with OSGi, you must use the IBM CICS SDK for Java and CICS Explorer:

- The IBM CICS SDK for Java enhances an existing Eclipse Integrated Development Environment (IDE) to provide the tools and support to help Java developers create and deploy Java applications in CICS. Use this tool to convert existing Java applications to OSGi bundles.
- CICS Explorer is an Eclipse-based systems management tool that provides system administrators with views for OSGi bundles, OSGi services, and the JVM servers in which they run. Use this tool to enable and disable Java applications, check the status of OSGi bundles and services in the framework, and get some preliminary statistics on the performance of the JVM server.

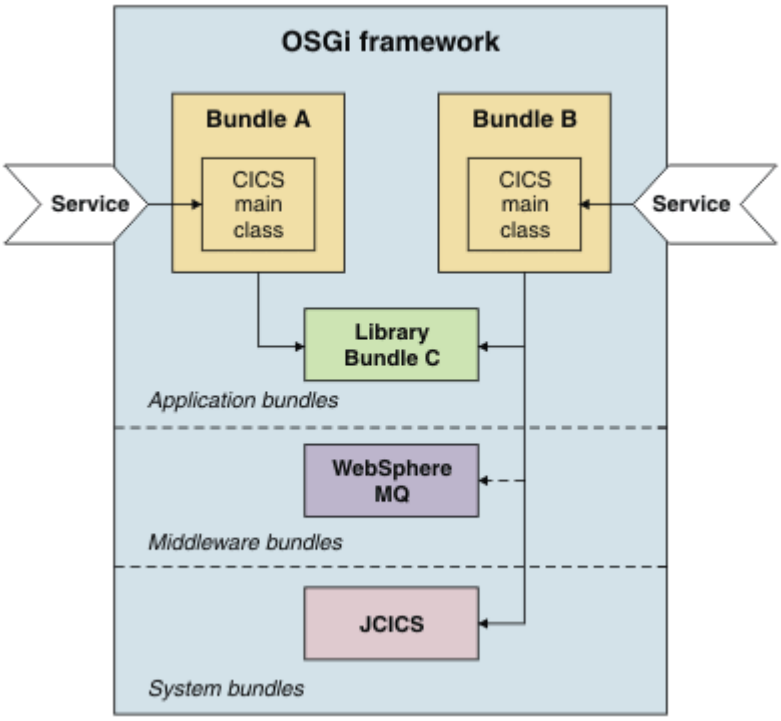
Any Java developer or systems administrator who wants to work with OSGi requires access to these freely available tools.

The following examples describe how you can run Java applications that use OSGi in CICS.

Run multiple Java applications in the same JVM server

The JVM server can handle multiple requests in the same JVM concurrently. Therefore, you can call the same application multiple times concurrently or run more than one application in the same JVM server.

When you decide how to split your applications between JVM servers, you can plan how to use the OSGi model to componentize your applications into a set of OSGi bundles. You must also decide what supporting OSGi bundles are required in the framework to provide services to your applications. The OSGi framework can contain different types of OSGi bundle, as shown in the following diagram:



Application bundles

An application bundle is an OSGi bundle that contains application code. OSGi bundles can be self-contained or have dependencies on other bundles in the framework. These dependencies are managed by the framework, so that an OSGi bundle that has an unresolved dependency cannot run in the framework. In order for the application to be accessible outside the framework in CICS, an OSGi bundle must declare a CICS main class as its OSGi service. If a PROGRAM resource points to the CICS main class, other applications outside the OSGi framework can access the Java application. If you have an OSGi bundle that contains common libraries for one or more applications, a Java developer might decide not to declare a CICS main class. This OSGi bundle is available only to other OSGi bundles in the framework.

The deployment unit for a Java application is a CICS bundle. A CICS bundle can contain any number of OSGi bundles and can be deployed on one or more JVM servers. You can add, update, and remove application bundles independently from managing the JVM server.

Middleware bundles

A middleware bundle is an OSGi bundle that contains classes to implement system services, such as connecting to WebSphere MQ. Another example might be an OSGi bundle that contains native code and must be loaded only once in the OSGi framework. A middleware bundle is managed with the lifecycle of the JVM server, rather than the applications that use its classes. Middleware bundles are specified in the JVM profile of the JVM server and are loaded by CICS when the JVM server starts up.

System bundles

A system bundle is an OSGi bundle that manages the interaction between CICS and the OSGi framework to provide key services to the applications. The primary example is the JCICS OSGi bundles, which provide access to CICS services and resources.

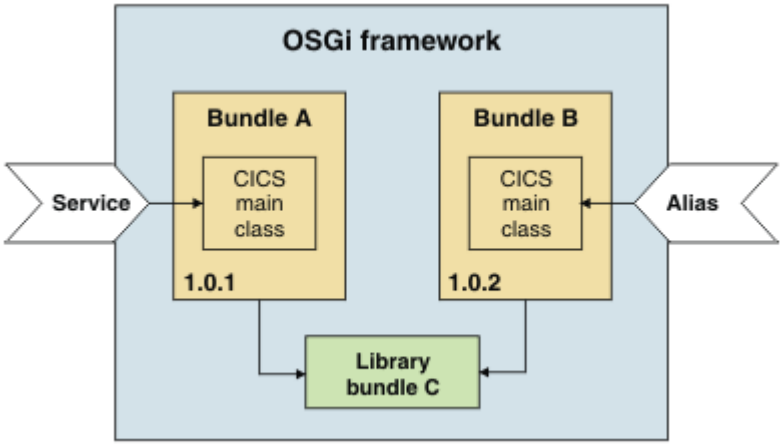
To simplify the management of your Java applications, follow these best practices:

- Deploy tightly coupled OSGi bundles that comprise an application in the same CICS bundle. Tightly coupled bundles export classes directly from each other without using OSGi services. Deploy these OSGi bundles together in a CICS bundle to update and manage them together.
- Avoid creating dependencies between applications. Instead, create a common library in a separate OSGi bundle and manage it in its own CICS bundle. You can update the library separately from the applications.

- Follow OSGi best practices by using versions when you are creating dependencies between bundles. Using a range of versions mean that an application can tolerate compatible updates to bundles that it depends on.
- You should always explicitly declare the packages that your OSGi bundle uses, even if the tooling does not indicate an error. You can do this by adding or updating the `Import-Package` bundle header in your OSGi bundle manifest. Tools such as Eclipse make assumptions about the availability of **`javax.*`** packages that might not be correct for a runtime environment where an explicit Import is necessary.
- Set up a naming convention for the JVM servers and agree the convention between the system programmers and Java developers.
- Avoid the use of singleton OSGi bundles. Discarding a singleton bundle that other bundles depend on can cause the dependent bundles to fail.

Run multiple versions of the same Java application in a JVM server

The OSGi framework supports running multiple versions of an OSGi bundle in a framework, so you can phase in updates to the application without interrupting its availability. While you can install multiple implementations of the same OSGi service into the framework, the service with the highest version property is used when that service is called. In CICS the version property is inferred from the underlying OSGi bundle. So if you want to run multiple versions of the same Java application in a JVM server at the same time and the different versions of the OSGi bundle have the same CICS main class, you must use an alias on one definition of the CICS main class. The alias is specified with the declaration of the CICS main class and registered in the OSGi framework as the OSGi service for a specific version of the application. Specify the alias on another PROGRAM resource to make that version of the application available.



Java applications in a Liberty JVM server

CICS provides a Java EE application server that can run lightweight Java servlets and JavaServer Pages. Developers can use the rich features of the Liberty in CICS specifications to write Java EE applications for CICS. The application server runs in a JVM server and is built on WebSphere Application Server Liberty.

Liberty is a lightweight application server for application development that starts quickly and can run on different platforms. It is optimized for Java developers to quickly develop and test applications, requiring a minimal amount of effort to configure and start the web server. Java developers package the application and web server together for simple deployment by using Eclipse tools that are freely available. Web services support available includes Java API for RESTful Web Services (JAX-RS) and Java API for XML Web Services (JAX-WS). For more information about Liberty, see Liberty overview.

Liberty is installed with CICS to run as a application server in a JVM server. The Liberty JVM server supports a subset of the features that are available in Liberty; you can run OSGi applications, Java servlets, and JSP pages. For more information about what features are supported, see Liberty features.

You might want to use the Liberty JVM server and associated tools for any of the following reasons:

- You want to modernize the presentation interfaces of your CICS application, replacing 3270 screens with web browser and RESTful clients.
- You want to use Java standards-based development tools to package, co-locate, and manage a web client with other existing CICS applications.
- You already use Liberty applications in WebSphere Application Server and want to port them to run in CICS.
- You already use Jetty or similar servlet engines in CICS and want to migrate to an application server that is based on Liberty.
- You want to use data source definitions to access DB2 databases from Java. See Defining the CICS DB2 connection.
- You want to coordinate updates made to CICS recoverable resources with updates made to a remote resource manager via a type 4 JDBC database driver, using the Java Transaction API (JTA).
- You want to develop services that follow REpresentational State Transfer (REST) principles using JAX-RS.
- You want to develop applications through support of a standard, annotation-based model using JAX-WS.
- You want to develop Java EE applications that send and receive secure messages via JMS.

CICS exception handling in Liberty applications

Liberty applications can use several different transactional APIs, including the JCICS API. Most Liberty components (except for EJBs) require the explicit use of the Java Transactions API (JTA) to coordinate transactions across those APIs. For example, if you need JCICS and remote JDBC activity to rollback following an Exception issued in application code, you must start a JTA transaction before interacting with the JDBC connection.

CICS implements an automatic rollback-CICS-transactions-on-Exception policy for simple servlets hosted in Liberty. This policy ensures that CICS transactions roll back if an Exception is thrown from an ordinary servlet. This is sufficient to provide basic transactional integration for simple servlets that use the JCICS API, but the policy does not address some of the more complicated scenarios you might encounter.

For example, the rollback-CICS-transactions-on-Exception policy doesn't integrate with other non-CICS resource adapters such as remote JDBC and JCA connections. If you need to coordinate transactions between CICS and other resource managers, you must use JTA to explicitly coordinate the transactions. This causes Liberty, CICS, and the remote transaction managers, to jointly negotiate whether to commit or rollback the transactions.

The rollback-CICS-transactions-on-Exception policy is available for simple servlets, but isn't available to the entire range of extensibility points available in a Liberty environment. Advanced users who exploit other plugin, callback, and extension points might not experience automated rollback of the CICS transaction when throwing an Exception. If you need predictable transactionality for Exceptions thrown from such components, use JTA to coordinate the transactions; an alternative option is to issue an explicit JCICS Abend to force CICS to rollback the CICS transaction for application detected errors.

CICS tasks for Liberty applications

In order for a Liberty application to use the JCICS API and other CICS resources, such as a JDBC DataSource with type 2 connectivity, requests must run under a CICS task. CICS creates a task for an application request at different times, dependent on the type of request. For HTTP requests, a task is created before the Liberty application is invoked. For other types of requests, for example message-driven beans (MDBs), inbound JCA, and remote EJBs, a task is created as required.

If the application does not interact with CICS, no CICS task is created for non-HTTP requests.

CICS performs the following actions when a CICS task is created:

- The CICS transaction security check occurs.
- CICS monitoring begins for the task.

- CICS trace for the task starts.
- The name of the Java thread is changed to include the CICS task number and transaction ID.

For non-HTTP applications, these actions occur the first time a JCICS API or a JDBC DataSource with type 2 connectivity is used. If the application does not interact with CICS, no CICS monitoring or transaction security occurs.

CICS abend handling for uncaught Java exceptions does not apply unless there is a CICS task. If an application throws an exception before the JCICS API or a JDBC DataSource with type 2 connectivity is used, no AJ05 abend occurs.

Java web services

CICS includes the Axis2 technology to run Java web services. Axis2 is an open source web services engine from the Apache foundation and is provided with CICS to process SOAP messages in a Java environment.

Axis2 is a Java implementation of a web services SOAP engine that supports a number of the web services specifications. It also provides a programming model that describes how to create Java applications that can run in Axis2. Axis2 is provided with CICS to process web services in a Java environment, and therefore supports offloading eligible Java processing to zAAP processors.

The JVM server supports running Axis2 to process inbound and outbound SOAP messages in a Java SOAP pipeline, without changing any of your existing web services. However, you can also create a web service from a Java application and run it in the same JVM server. By deploying the application to the Axis2 repository of the JVM server, both the Java application and SOAP processing are eligible for running on a zEnterprise Application Assist Processor (zAAP).

You might want to use Java web services for one of the following reasons:

- You have experience of Axis2 web services on other platforms and want to create web services in CICS.
- You want to use standard Java APIs to create Java data bindings that integrate with Axis2.
- You have complicated WSDL documents that are difficult to handle with the CICS web services assistants.

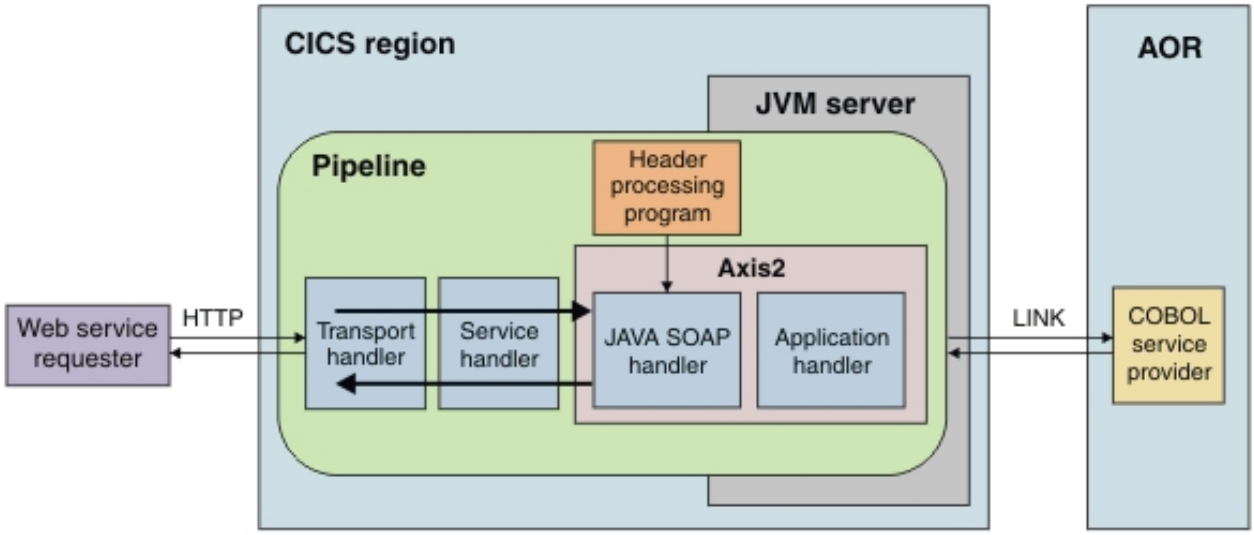
The following examples describe how you can use Java with web services.

Process SOAP messages in a JVM server

Most SOAP processing that occurs in the web services pipeline is performed by the SOAP handler and application handler. You can optionally run this SOAP processing in a JVM server and use zAAPs to run the work. You can continue to use web service applications that are written in COBOL, C, C++, or PL/I.

If you have existing web services, you can update the configuration of your pipelines to use a JVM server. You do not have to change the web services. If the pipeline uses a SOAP header processing program, it is best to rewrite the program in Java by using the Axis2 programming model. The header processing program can share the Java objects with Axis2 without doing any further data conversion. If you have a header processing program in COBOL for example, the data must be converted from Java into COBOL and back again, which can slow down the performance of the SOAP processing.

The scenario shown in the following diagram is an example of a COBOL application that is a web service provider. The request is processed in a pipeline that is configured to support Java. The SOAP handler and application handler are Java programs that are processed by Axis2 and run in a JVM server. The application handler converts the data from XML to COBOL and links to the application.



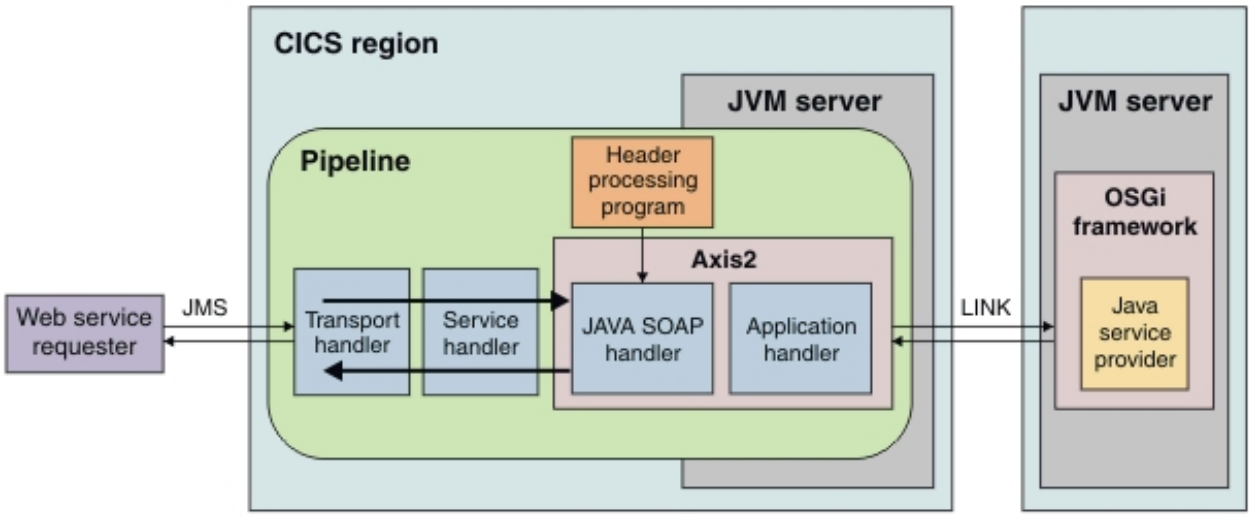
When you are planning your environment, ensure that you use a set of dedicated regions for your JVM servers. In this example, the COBOL application runs in an application-owning region (AOR) that is separate from the CICS region where the JVM server runs. You can use workload management to balance the workloads, for example on the **`EXEC CICS LINK`** from the application handler or on the inbound request from the web service requester.

Write a Java application that uses output from the CICS web services assistant

You can write a Java application that interprets the language structures and uses the data bindings generated by the CICS web services assistant. The web services assistant can produce language structures from WSDL or WSDL from language structures. The assistant also produces a web service binding that describes how to convert the data between XML and the target language during SOAP processing.

If you use the assistant to generate a language structure, you can use IBM Record Generator for Java or the Rational J2C Tools to work with the language structures to generate Java classes. These tools provide a way for Java developers to interact with other CICS applications. In this example, you can use these tools to write a Java application that can handle an inbound SOAP message after CICS has converted the data from XML. For more information, see Interacting with structured data from Java.

The scenario shown in the following diagram is an example of a Java application that is a web service provider. The SOAP processing is handled by Axis2 in a JVM server. The application handler links to the Java application, which is packaged and deployed as one or more OSGi bundles and runs in a JVM server.



The advantage of this approach is that because the data bindings were generated by the web services assistant, the web service is represented in CICS by the WEBSERVICE resource. You can use statistics, resource management, and other facilities in CICS to manage the web service. The disadvantage is that the Java developer must work with language structures for a programming language that might be unfamiliar.

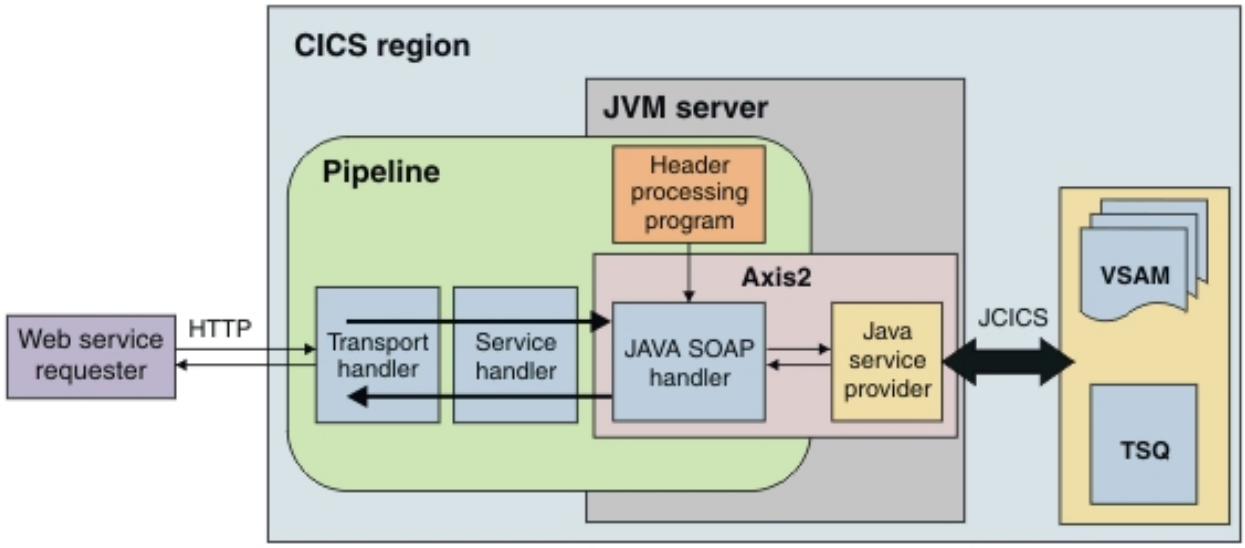
When you are planning your environment for this type of application, use a separate JVM server to run the application:

- You can more effectively manage and tune the JVM servers for the different workloads.
- You can use workload management on the inbound requests and **EXEC CICS LINK** to balance workloads and scale the environment.
- You can take advantage of the OSGi support in CICS to manage the Java application.

Write a Java application that uses Java data bindings

You can write a Java application that generates and parses the XML for SOAP messages. The Java 7 API provides standard Java libraries to work with XML; for example, you can use the Java Architecture for XML Binding (JAXB) to create the Java data bindings, and the Java API for XML Web Services (JAX-WS) libraries to generate and parse the XML. If you use these libraries, the application can run in Axis2 in the same JVM server as the SOAP pipeline processing.

The scenario shown in the following diagram is an example of a Java application that is a web service provider and is processed by the Axis2 SOAP engine in a JVM server.



The Java application uses Java data bindings and interacts with the Java SOAP handler, so there is no application handler. In this example, the web service requester uses HTTP to connect to the CICS region, but you can also use JMS. The Java application uses JCICS to access CICS services, in this example VSAM files and a temporary storage queue.

The advantage of this approach is that the Java developer uses familiar technologies to create the application. Also, the Java developer can work with complex WSDL documents that the web services assistant cannot process to produce a binding. However, this approach has some limitations:

- You cannot use WS-Security for this type of application, so if you want to use security, use SSL to secure the connection.
- No context switch for the user ID occurs in the pipeline processing. To change the user ID on the request, use a URIMAP resource.
- Because you are not using the web service binding from the web services assistant, there is no WEBSERVICE resource.

- If the application is a web service requester, the pipeline processing is bypassed. So you do not get the qualities of service that are available in the pipeline.

If you implement workload management in your CICS regions, you must plan how to route this type of workload. Because the Java application runs in the same JVM server as the SOAP processing, CICS does not provide a routing opportunity. However, you can implement a distributed program link in the JAX-WS application to another program if routing is required.

Spring Boot support in CICS

The CICS Liberty JVM server supports Spring Boot applications by using the Spring application programming model. Spring Boot provides a simpler and faster way of configuring, building, and running Spring applications. Spring was originally designed to simplify Java Enterprise Edition (EE), by using plain old Java objects (POJOs) and dependency injection. It now extends and encompasses many aspects of Java EE development. Spring Boot builds on Spring by adding components to avoid complex configuration, reduce development time, and offer a simpler startup experience. Most Spring Boot applications require little Spring configuration. For more information about Spring and Spring Boot, see Spring Boot overview.

You can build your Spring Boot applications as web application archives (WARs),which allows them be deployed and managed by using CICS bundles in the same way as can other CICS Liberty applications.

Chapter 2. Developing Java applications

You can write Java application programs that use CICS services and run under CICS control. Using the IBM CICS SDK for Java or other Java IDEs, you can develop applications that use the JCICS class library to access CICS resources and interact with programs that are written in other languages. You can also connect to your Java programs by using various protocols and technologies, such as web services or z/OS Connect Enterprise Edition.

CICS provides a JVM server runtime environment for Java application development. You can develop applications using the IBM CICS SDK for Java, Maven modules, or Gradle modules.

The IBM CICS SDK for Java is an Eclipse-based tool that provides support for developing and deploying Java applications to CICS. It contains the JCICS class libraries to develop applications that access CICS resources and services; for example, you can access VSAM files, transient data queues, and temporary storage. You can also use JCICS to link to CICS applications that are written in other languages, such as COBOL, PL/I, and C. The IBM CICS SDK for Java includes a set of samples to help you get started if you are new to developing Java applications for CICS.

The artifacts for Java that are provided by CICS on Maven Central enable you to resolve Java dependencies easily and fast. They provide support for the JCICS library, CICS annotations, and CICS annotation processor. You can declare the dependencies in a Maven or Gradle module using most Java IDEs, and develop the applications in the same way as you do for other platforms.

What you need to know about CICS

CICS is a transaction processing subsystem that provides services for a user to run applications by request. It enables many users to submit requests to run the same applications, using the same files and programs, at the same time. CICS manages the sharing of resources, integrity of data, and prioritization of execution, while maintaining fast response times.

A CICS application is a collection of related programs that together perform a business operation, such as processing a product order or preparing a company payroll. CICS applications run under CICS control, using CICS services and interfaces to access programs and files.

You run CICS applications by submitting a *transaction* request. The term transaction has a special meaning in CICS; See “CICS transactions ” on page 21 for an explanation of the difference between the CICS usage and the more common industry usage. Execution of the transaction consists of running one or more application programs that implement the required function.

To develop Java applications for CICS, you have to understand the relationship between CICS programs, transactions, and tasks. These terms are used throughout CICS documentation and appear in many programming commands. You also have to understand how CICS handles Java applications in the runtime environment.

CICS transactions

A transaction is a piece of processing initiated by a single request.

The request is typically made by a user at a terminal. However, it could be made from a web page, from a remote workstation program, or from an application in another CICS region; or it might be triggered automatically at a predefined time. The CICS web support concepts and structure and the Overview of CICS external interfaces describe different ways of running CICS transactions.

A single transaction consists of one or more *application programs* that, when run, carry out the processing needed.

However, the term *transaction* is used in CICS to mean both a single event and all other transactions of the same type. You describe each transaction type to CICS with a TRANSACTION resource definition. This definition gives the transaction type a name (the transaction identifier, or TRANSID) and tells CICS several

things about the work to be done, such as which program to invoke first, and what kind of authentication is required throughout the execution of the transaction.

You run a transaction by submitting its TRANSID to CICS. CICS uses the information recorded in the TRANSACTION definition to establish the correct execution environment, and starts the first program.

The term *transaction* is now used extensively in the IT industry to describe a *unit of recovery* or what CICS calls a *unit of work*. This is typically a complete logical operation that is recoverable; it can be committed or backed out as an entirety as a result of a programmed command or of a system failure. In many cases, the scope of a CICS transaction is also a single unit of work, but you should be aware of the difference in meaning when reading CICS documentation.

CICS tasks

A task is single instance of the execution of a transaction.

The word *task* has a specific meaning in CICS. When CICS receives a request to run a transaction, it starts a new task that is associated with this *one instance* of the execution of the transaction type. That is, a CICS task is one execution of a transaction, with its own private set of data, usually on behalf of a specific user. You can also consider a task as a *thread*. Tasks are *dispatched* by CICS according to their priority and readiness. When the transaction completes, the task is terminated.

CICS application programs

In Java programs, you can use the Java class library for CICS (JCICS) to access CICS services and link to application programs that are written in other languages.

CICS application programs can be written in COBOL, C, C++, Java, PL/I, or assembler languages. Most of the processing logic is expressed in standard language statements, but to request CICS services, applications use the provided application programming interfaces. COBOL, C, C++, PL/I, or assembler programs can use the **EXEC CICS** application programming interface or the C++ class library. Java programs use the JCICS class library. JCICS is described in "The Java class library for CICS (JCICS)" on page 37.

CICS services

Java programs can access the following CICS services through the JCICS programming interface: Data management, communications, unit-of-work, program, and diagnostic services.

CICS services managers usually have the word control in their title; for example, "terminal control" and "program control". These terms are used extensively in CICS information.

Data management services

CICS provides the following data management services:

- Record-level sharing, with integrity, in accessing Virtual Storage Access Method (VSAM) data sets. CICS logs activity to support data backout (for transaction or system failure) and forward recovery (for media failure). CICS file control manages the VSAM data.

  CICS also implements two proprietary file structures, and provides commands to manipulate them:

  Temporary storage
  Temporary storage (TS) is a means of making data readily available to multiple transactions. Data is kept in queues, which are created as required by programs. Queues can be accessed sequentially or by item number.

  Temporary storage queues can reside in main memory, or can be written to a storage device.

  A temporary storage queue can be thought of as a named scratchpad.

Transient data
Transient data (TD) is also available to multiple transactions, and is kept in queues. However, unlike TS queues, TD queues must be predefined and can be read only sequentially. Each item is removed from the queue when it is read.

Transient data queues are always written to a data set. You can define a transient data queue so that writing a specific number of items to it acts as a trigger to start a specific transaction. For example, the triggered transaction might process the queue.

- Access to data in other databases (including DB2), through interfaces with database products.

Communications services

CICS provides commands that give access to a wide range of terminals (displays, printers, and workstations) by using SNA and TCP/IP protocols. CICS terminal control provides management of SNA and TCP/IP networks.

You can write programs that use Advanced Program-to-Program Communication (APPC) commands to start and communicate with other programs in remote systems, using SNA protocols. CICS APPC implements the peer-to-peer distributed application model.

The following CICS proprietary communications services are provided:

Function shipping
Program requests to access resources (files, queues, and programs) that are defined as existing on remote CICS regions are automatically routed by CICS to the owning region.

Distributed program link (DPL)
Program-link requests for a program defined as existing on a remote CICS region are automatically routed to the owning region. CICS provides commands to maintain the integrity of the distributed application.

Asynchronous processing
CICS provides commands to allow a program to start another transaction in the same, or in a remote, CICS region and optionally pass data to it. The new transaction is scheduled independently, in a new task. This function is similar to the *fork* operation provided by other software products.

Transaction routing
Requests to run transactions that are defined as existing on remote CICS regions are automatically routed to the owning region. Responses to the user are routed back to the region that received the request.

Unit of work services

When CICS creates a new task to run a transaction, a new unit of work (UOW) is started automatically. CICS does not provide a BEGIN command, because one is not required. CICS transactions are always executed in-transaction.

CICS provides a SYNCPOINT command to commit or roll back recoverable work done. When the sync point completes, CICS automatically starts another unit of work. If you terminate your program without issuing a SYNCPOINT command, CICS takes an implicit sync point and attempts to commit the transaction.

The scope of the commit includes all CICS resources that have been defined as recoverable, and any other resource managers that have registered an interest through interfaces provided by CICS.

Program services

CICS provides commands that enable a program to link or transfer control to another program, and return.

Diagnostic services

CICS provides commands that you can use to trace programs and produce dumps.

Java runtime environment in CICS

CICS provides the JVM server environment for running threadsafe Java applications. Applications that are not threadsafe cannot use a JVM server.

The JVM server is a runtime environment that can run tasks in a single JVM. This environment reduces the amount of virtual storage required for each Java task, and allows CICS to run many tasks concurrently.

CICS tasks run in parallel as threads in the same JVM server process. The JVM is shared by all CICS tasks, which might be running multiple applications concurrently. All static data and static classes are also shared. So to use a JVM server in CICS, a Java application must be threadsafe. Each thread runs under a T8 TCB and can access CICS services by using the JCICS API.

Do not use the `System.exit()` method in your applications. This method causes both the JVM server and CICS to shut down, affecting the state and availability of your applications.

Multithreaded applications

You can write application code to start a new thread or call a library that starts a thread. If you want to create threads in your application, the preferred method is to use a generic `ExecutorService` from the OSGi registry. The `ExecutorService` automatically uses `CICSExecutorService` to create CICS threads when the application is running in a JVM server. This approach means the application is easier to port to other environments and you do not have to use specific JCICS API methods.

However, if you are writing an application that is specific to CICS, you can choose to use the `CICSExecutorService` class in the JCICS API to request new threads.

Whichever approach you choose, the newly created threads run as CICS tasks and can access CICS services. When the JVM server is disabled, CICS waits for all CICS tasks running in the JVM to finish. By using the `ExecutorService` or `CICSExecutorService` class, CICS is aware of the tasks that are running and you can ensure that your application work completes before the JVM server shuts down.

You should only use JCICS objects in the task that created them. Any attempt to share the objects between tasks can produce unpredictable results.

For further details on using the CICS ExecutorService refer to “Threads” on page 40.

JVM server startup and shutdown

Because static data is shared by all threads that are running in the JVM server, you can create OSGi bundle activator classes to initialize static data and leave it in the correct state when the JVM shuts down. A JVM server runs until disabled by an administrator, for example to change the configuration of the JVM or to fix a problem. By providing bundle activator classes, you can ensure that the state is correctly set for your applications. CICS has a timeout that specifies how long to wait for these classes to complete before continuing to start or stop the JVM server. You cannot directly use JCICS in startup and termination classes. However, a developer can start a new JCICS-enabled thread from an activator, by using the `CICSExecutorService.runAsCICS()` API. Any JCICS commands will run under the authority of the user id that issued the install command. Therefore it is prudent for an administrator to understand the resources used in bundle activators before they install them.

Developing applications using the IBM CICS SDK for Java

CICS Explorer includes the IBM CICS SDK for Java and optionally the IBM CICS SDK for Java EE and Liberty. This IBM CICS SDK for Java provides an environment for developing and deploying Java applications to CICS, including support for OSGi and web projects.

If you want to develop Java applications without using the SDK, see “Developing applications using Maven or Gradle” on page 31.

You can use the IBM CICS SDK for Java to create new applications, or repackage existing Java applications to comply with the OSGi specification. OSGi provides a mechanism for developing applications by using a component model and deploying those applications to a framework as OSGi

bundles. An *OSGi bundle* is the unit of deployment for an application and contains version information, dependencies, and application code. The main benefit of OSGi is that you can create applications from reusable components that are accessed only through well-defined interfaces called *Java packages*. You can then use OSGi services to access the Java packages. You can also manage the lifecycle and dependencies of Java applications in a granular way. For information about developing applications with OSGi, see OSGi Alliance.

You can use the IBM CICS SDK for Java to develop a Java application to run in any supported release of CICS. Different releases of CICS support different versions of Java, and the JCICS API is also extended in later releases to support more features of CICS. To avoid use of the wrong classes, the IBM CICS SDK for Java provides a feature to set up a target platform or project libraries. You can define which release of CICS you are developing for, and the IBM CICS SDK for Java automatically hides the Java classes that you cannot use.

If you are using the Liberty JVM server, the IBM CICS SDK for Java can help you work with Dynamic Web Projects and OSGi Application Projects. You can create an application that has a modern web layer and business logic that uses JCICS to access CICS services. If your web application needs to access code from another OSGi bundle, it must be deployed as an OSGi Application Project (EBA file). You must either include the other OSGi bundle in the application manifest, or install the other bundle in the Liberty `bundle_repository` as a common library. The EBA file must include a web-enabled OSGi bundle (WAB file) to provide the entry point to the application and to expose it as a URL to a web browser.

Setting up the Target Platform

You must set up and update the target platform of your Eclipse development environment as necessary before developing or deploying OSGi-based Java applications.

About this task

You can use a template target platform as-is or update it with additional support. The CICS Explorer Software Development Kit (SDK) only supplies Java classes necessary for the usage of the CICS or web APIs. To add support for additional interfaces, you must add the OSGi plug-in that contains the third party Java classes to the Eclipse Target Platform. This procedure makes the exported packages available to all applications that use this target platform. If you need to add third party Java classes to your target platform, ensure the JAR file that contains those classes is available as an OSGi plug-in and is copied to the local workstation.

Procedure

1. In Eclipse, click **Window** > **Preferences**.
2. In the **Preferences** page, expand **Plug-in Development** and click **Target Platform**.
3. Create or update a target definition as needed:
   - If you need a new target definition, click **Add** to create a target definition in the wizard.
   a) Select **Template** and select the target platform that matches your CICS version, for example, **CICS TS 5.4**.
   b) Click **Next** in the wizard and then click **Finish**.
   - If your target definition is already in the list, proceed to the following steps.

To update the target definition with additional Java classes:

4. Optional: Select the target definition in the **Target Platform** dialog and click **Edit**, which opens the **Edit Target Definition** dialog.
5. Optional: Under the **Locations** tab, click **Add**. Browse the directory and add the OSGi plug-in that contains the third party bundle JARs.
6. Optional: After the OSGi plug-in content is added, in the **Edit Target Definition** dialog, click **Finish**.
7. After creating or updating the target definition, return to the **Preferences** page and click **Apply and Close**.

Results

You have successfully updated the OSGi environment to include both the third party OSGi bundles and the CICS OSGi bundles that are required for Java application development.

What to do next

Deploy the Java application into a CICS JVM server, and add the third party JARs as an OSGi middleware bundle or to the Liberty shared bundle repository. For further details, see Updating OSGi middleware bundles and Manually tailoring server.xml.

Creating a plug-in project

You create your CICS Java application as an Eclipse plug-in project that complies with the OSGi specification. The OSGi Service Platform provides a mechanism for developing applications by using a component model and deploying those applications into a framework as OSGi bundles. The plug-in project is an OSGi bundle, and contains all the files and artifacts needed for the CICS Java application. The plug-in project is then included in a CICS bundle project before being exported to the host system.

Before you begin

You need to set the Target Platform. For more information, see "Setting up the Target Platform" on page 25.

About this task

This task creates a new plug-in project. You can leave the settings on their default values unless otherwise stated. When the project is created you must edit the manifest and add the JCICS API dependencies.

Procedure

1. On the Eclipse menu bar click **File** > **New** > **Project** to open the New Project wizard.
2. Select **Plug-in Project** from the list provided, then click **Next** to open the New Plug-in Project wizard.
3. In the **Project name** field, enter a name for the project, for example `com.ibm.cics.example.accounting`. In the Target Platform section, select **an OSGi framework** and select **standard** from the menu. Click **Next**.

   The Content pane is displayed.
4. In the **Version** field remove the ".qualifier" from the end of the version number.
5. In the **Execution Environment** field select the Java level that matches the execution environment in your CICS runtime target platform, for example **JavaSE-1.7**.
6. Uncheck the **Generate an activator** check box and click **Finish**.

   The new plug-in project is created in the Package Explorer view.
7. You must now edit the plug-in manifest file and add the `JCICS` and `com.ibm.record` API dependencies. If you do not perform these steps, you will be able to export and install the bundle, but it will not run.
   a) In the Package Explorer view, right-click the project name and click **Plug-in Tools** > **Open Manifest**.

      The manifest file opens in the manifest editor.
   b) Select the **Dependencies** tab and in the Imported Packages section, click **Add**.

      The Package Selection dialog opens.
   c) Select the package `com.ibm.cics.server` and click **OK**.

      The package is displayed in the Imported Packages list.
   d) Repeat the previous step to install the following package, if it is required for your application:

com.ibm.record
The Java API for legacy programs that use IByteBuffer from the Java Record Framework that came with VisualAge. Previously in the `dfjcics.jar` file.

e) Select **File** > **Save** to save the manifest file.

Results

The new plug-in project is created containing the JCICS API dependencies.

What to do next

You can now create your CICS Java application. If you are new to developing Java applications for CICS, you can use the JCICS samples provided with the IBM CICS SDK for Java to help you get started.

Note: After you have developed your application, you must add a CICS-MainClass declaration to the manifest file and declare the classes used in the application. See the related link for more information.

For more information on plug-in development, see the section *Plug-in development environment (PDE) user guide* in the Eclipse Help documentation.

When your Java application is finished, you must deploy it in a CICS bundle to zFS. CICS bundles can contain one or more plug-ins and are the unit of deployment for your application in CICS.

Updating the plug-in project manifest file

When you develop a JCICS application, or package an existing application in a plug-in project, you must update the project manifest file and include a `CICS-MainClass` header.

About this task

The `CICS-MainClass` header is used to declare the classes that can be called by a `LINK`, `START` or `RUN` command, or a transaction initial program. Do not use lazy activation policies for OSGi bundles that declare a CICS main class. CICS activates the OSGi bundles as soon as they are started in the OSGi framework. You must add the declaration manually to the manifest file.

Procedure

1. If the manifest file is not already open in the editor, right-click the project name in the Package Explorer view and click **Plug-in Tools > Open Manifest**.

   The manifest file opens in the manifest editor.
2. Select the **MANIFEST.MF** tab. The content of the file is displayed.
3. Add the following declaration to the manifest file:

   `CICS-MainClass: packagename.classname` where:

   packagename
   Is the fully qualified Java package name.

   classname
   Is the name of the class used in the application. If more than one class is used, repeat the *packagename.classname* element, separated by a comma.

   You can use aliases in the `CICS-MainClass` header; for example, the declaration `CICS-MainClass: examples.hello.HelloCICSWorld; alias=greeting` assigns the alias `greeting` to the `CICS-MainClass` `examples.hello.HelloCICSWorld`. When you define the program to CICS, you use the alias name, `greeting`, instead of the class name. An alias is useful if you have multiple versions of the same program, each with the same class name. By using aliases you can identify the different versions.

The following example shows a manifest file with a `CICS-MainClass` header for the classes `HelloCICSWorld` and `HelloWorld`.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Hello Plug-in
Bundle-SymbolicName: com.ibm.cics.server.examples.hello
Bundle-Version 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Import-Package: com.ibm.cics.core.bundle,
  com.ibm.cics.core.model.builders,
  com.ibm.cics.server;version="[1.300.0,2.0.0)"
CICS-MainClass: examples.hello.HelloCICSWorld,
  examples.hello.HelloWorld
```

4. When you have added all the class declarations, select **File > Save** to save the manifest file.

Results

You can now add the plug-in project to a CICS bundle and deploy it to zFS. CICS bundles can contain one or more plug-ins and are the unit of deployment for your application in CICS.

What to do next

Create a CICS bundle project. See Creating a CICS bundle project in the CICS Explorer product documentation.

Creating a Java EE application

The CICS Explorer and IBM CICS SDK for Java help provides full details on how you can perform each of the following steps to develop and deploy applications.

Procedure

1. Set up a target platform for your Java development.

   .

   The target platform ensures that you use only the Java classes that are appropriate for the target release of CICS in your application development.
2. Create an OSGi Bundle Project or a plug-in project for your Java application development.
   a) The default version of the project is `1.0.0.qualifier`. In the **Version** field either remove the `.qualifier` from the end of the version number, if you do not wish to use one, or set it to something meaningful, for example the date/time stamp.

   Develop your Java application using best practices; for example, to organize the dependencies between OSGi bundles, use Import-Package / Export-Package in preference to Require-Bundle.
3. If you are new to developing Java applications for CICS, you can use the examples that are provided with the IBM CICS SDK for Java to get started.

   To use JCICS in a OSGi Java application, you must import the `com.ibm.cics.server` package.
4. Optional: In Liberty, create a dynamic web application (WAR) or a web-enabled OSGi Bundle Project (WAB) to develop your application presentation layer.

   You can create servlets and JSP pages in a Dynamic Web Project. For a WAR file, you must also add the Liberty libraries to your build path to give you access to the Liberty API bundles. For further details, refer to “Setting up the development environment” on page 67.
5. Package your application for deployment:
   a) If you are deploying a web-enabled OSGi Bundle Project (WAB), create an OSGi Application Project (EBA).
   b) Create one or more CICS bundle projects to reference your EBA, your EAR file, or your web application (WAR file).

CICS bundles are the unit of deployment for your application in CICS. Put the web applications that you want to update and manage together in a CICS bundle project. You must know the name of the JVMSERVER resource in which you want to deploy the application.

You can also add CICS resources to the CICS bundle project, such as PROGRAM, URIMAP, and TRANSACTION resources. These resources are dynamically installed and managed with the Java application.

c) Optional: If you want to deploy the application to a CICS platform, create an application project that references your CICS bundles.

An application provides a single management point for deploying and managing the application across a CICSplex in CICS. For more information, see Packaging CICS applications for deployment in a cloud environment.

d) You should always explicitly declare the packages that your OSGi bundle uses, even if the tooling does not indicate an error. You can do this by adding or updating the **`Import-Package`** bundle header in your OSGi bundle manifest. Tools such as Eclipse make assumptions about the availability of **`javax.*`** packages that might not be correct for a runtime environment where an explicit Import is necessary.

6. Deploy your Java application to zFS by exporting the application project or CICS bundle projects. Alternatively, you can save the projects in a source repository for deployment.

Results

You have successfully developed and exported your application by using the IBM CICS SDK for Java.

What to do next

Install the application in a JVM server. If you do not have authority to create resources in CICS, the system programmer or administrator can create the application for you. You must tell the system programmer or administrator where the exported bundle is located and the name of the target JVM server.

Adding a project to a CICS bundle project

When you create a CICS bundle project a manifest file is created in the `META-INF` directory. You can edit the manifest file to include one or more of the following types of projects; Dynamic Web Project, Enterprise Application Project, OSGi Application Project, or OSGi Bundle Project. The included projects can be source or pre-built. When you export the CICS bundle project, all included projects are contained in the CICS bundle on zFS.

Before you begin

This task describes how to add details of a project to a CICS bundle. If you have not created a CICS bundle project, see Creating a CICS bundle project in the CICS Explorer product documentation.

About this task

You can add details of a project to a CICS bundle by using one of the following wizards; **Dynamic Web Project Include**, **Enterprise Application Project Include**, **OSGi Application Project Include**, or **OSGi Bundle Project Include**. The wizards update the bundle manifest file to include details of the project that is being added, and creates a resource file with a file extension of `.warbundle` , `.earbundle` , `.ebabundle` , or `.osgibundle` that points to the project.

Note: To add OSGi bundles that are not included in an OSGi application project to a CICS bundle project, you must have a `build.properties` file that includes the location of the output folder. For example, the `build.properties` file might have the following content:

```
source.. = src/
output.. = bin/
bin.includes = META-INF/
```

Procedure

1. In the Package Explorer view, right-click the bundle project that you want to update, and click **New** > **Other** to open the New wizard.
2. Expand the CICS Resources folder and click **Dynamic Web Project Include**, **Enterprise Application Project Include**, **OSGi Application Project Include**, or **OSGi Bundle Project Include**. Click **Next**.

   The wizard opens and displays the projects of that type in your workspace. The wizard also displays any built projects (for example, JAR, EAR, EBA, and WAR files) that are contained within the selected bundle project.
3. Click the project to include in the bundle.

   When you click the project, the wizard displays the symbolic name, and the version when applicable. You can hover over the project to identify whether it is a built project or a source project.
4. Optional: For an OSGi project, specify the version or version range to include:
   - Select **Use this version** to include the specific version of the selected OSGi project, as shown in the **Version** field.
   - Select **Use version range** to include the highest version in a defined version range of the selected OSGi project when you export that OSGi project. By default, the version range is from the version of the selected OSGi project to the next highest major version. You can use the fields and the buttons to specify a different range.
5. In the **JVM server** field, enter the name of the JVM server where the application component is going to run.
6. Optional: The name of the resource file that is created is generated from the project name and is displayed in the wizard. You can use the **Back** button to change the file name.
7. Click **Finish**.

Results

A project resource file is added to the bundle project and the manifest file is updated. You can repeat these steps to add more projects to the CICS bundle project.

What to do next

You can add resources to the CICS bundle project for your application. For example, you can create a program to make your Java application available to other applications in CICS.

You can deploy your CICS bundle to a z/OS UNIX file system, as described in Deploying a CICS bundle in the CICS Explorer product documentation. When the CICS bundle project is exported to zFS, all the files and artifacts needed for the application are compiled and exported.

Alternatively, you can package your CICS bundle project in a cloud-style application project for deployment into a CICS platform. By using an application project, you can group together all the CICS bundle projects that comprise your application and deploy and install them in a single step. For more information, see Creating a CICS Application Binding project in the CICS Explorer product documentation.

Updating the project build path

How to update the project build path.

About this task

Using a Dynamic Web Project creates a WAR file archive for deployment. This does not use the OSGi framework in Eclipse so you need to add third party JAR files to the project build path. This example uses IBM MQ JAR files.

Procedure

1. In Eclipse select the web project and right-click **Build Path** > **Configure Build Path**. This will display the Java Build Path window.
2. Add the CICS and Liberty libraries, click **Add Library** > **Liberty JVM server** > **Next** > **Finish**.
3. Click **Add External JARs** and navigate to the directory where the previously downloaded IBM MQ JAR files are located. Select the following JAR files depending on which imports are used in the applications:
   - com.ibm.mq.jar
   - com.ibm.mq.jmql.jar
   - com.ibm.mq.headers.jar

   Note: Step 3 is optional if you are using IBM MQ only.

Results

The build path of the project now has the correct interfaces for development of a web application using both CICS and IBM MQ APIs.

Developing applications using Maven or Gradle

When writing your own build scripts, you can use build toolchains such as Maven or Gradle to resolve Java dependencies. As an alternative to the IBM CICS SDK for Java , they can retrieve libraries from a remote repository or allow-listed local repositories.

CICS provides a set of artifacts on Maven Central, an online repository, for you to resolve Java dependencies. Maven and Gradle are supported by most Java integrated development environments (IDEs) and compatible with popular automation tools such as Jenkins and Travis CI.

What artifacts are available

These artifacts are available on Maven Central:

Table 2. CICS-provided artifacts on Maven Central

Group ID	Artifact ID	Description
`com.ibm.cics`	`com.ibm.cics.ts.bom`	The bill of materials (BOM) that defines the versions of all the artifacts to ensure they are at the same CICS TS level. **Tip:** You're recommended to use the BOM to control version numbers of the other dependencies and omit their version numbers from their own specifications. Learn more ...
	`com.ibm.cics.server`	The CICS Java class library (JCICS), a Java library that provides the **`EXEC CICS`** API support for Java applications in CICS TS. Learn more ...
	`com.ibm.cics.server.invocation.annotations`	CICS annotations, a Java library that provides the `@CICSProgram` annotation to enable CICS programs to invoke Java applications in a Liberty JVM server. Learn more ...
	`com.ibm.cics.server.invocation`	The CICS annotation processor, a Java library that is used during compilation to create metadata that enables CICS programs to invoke Java applications in a Liberty JVM server. Learn more ...

How to declare dependencies

Prerequisites: Before developing Java applications for CICS using Maven or Gradle, you must make sure that:

- You already have Maven or Gradle support installed in your machine or IDE.
- You have created a Maven or Gradle module to include your application, or that you have converted an existing Java project to a Maven or Gradle module. Most Java IDEs support this functionality.

For instructions, see the Maven and Gradle related information in “Setting up the development environment” on page 67.

Note: Not all Eclipse packages support Maven or Gradle by default. For example, if you use a standalone CICS Explorer for Aqua 3.1[1] or earlier, Maven or Gradle support is not included. To use Maven or Gradle,

[1] Aqua stands for z/OS Explorer for Aqua.

install the m2e (Maven integration for Eclipse) plug-in or the Buildship Gradle Integration plug-in through the Eclipse Marketplace.

You declare dependencies in the Maven module's `pom.xml` file or the Gradle module's `build.gradle` file. The following instructions per artifact include snippets showing how to declare dependency on each artifact; you can modify the artifact coordinates according to the information on Maven Central. You can also use other build tools to leverage the Maven Central artifacts, but this topic focuses on Maven and Gradle only.

Note: Snippets on Maven Central are auto-generated and might be missing configurations such as `<scope>import</scope>` and `compileOnly`. Follow the syntax in this topic instead to ensure the dependencies are correctly declared.

com.ibm.cics.ts.bom

All versions are available at com.ibm.cics.ts.bom.

Maven `pom.xml`

```
<dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>com.ibm.cics</groupId>
        <artifactId>com.ibm.cics.ts.bom</artifactId>
        <version>5.4-20200519102234-PH25409</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
</dependencyManagement>
```

In Maven, the BOM controls these configurations of the other dependencies:

Version:

The BOM version controls the version of any other CICS dependency in the same module or its child modules if the version of that dependency is not otherwise specified. Make sure that you specify a BOM version that provides the support for the other dependencies you need in your application, and that your target CICS system is at the same or newer CICS TS release and APAR maintenance level. In the snippet, the `version` tag specifies the BOM version, consisting of:

- the CICS version (`5.4`)
- the time stamp when the BOM is built (`20200519102234`)
- if relevant, the version of the CICS TS APAR (PH25409), which includes server-side updates to the libraries

Scope:

The BOM also specifies that the other dependencies have a `provided` scope, so you don't need to add `<scope>provided</scope>` in those Maven dependencies. When this scope is specified, the dependency will be provided by the eventual runtime and must not be packaged as part of the module. It not only reduces the application size, but also avoids hard-to-diagnose problems caused by inconsistent versions being used or classes being loaded from more than one class loader.

If you do not use a BOM, you must specify `<scope>provided</scope>` when declaring those dependencies in Maven.

Gradle `build.gradle`

```
repositories {
     mavenCentral()
}

dependencies {
    compileOnly enforcedPlatform('com.ibm.cics:com.ibm.cics.ts.bom:5.4-20200519102234-
PH25409')
}
```

In Gradle, the BOM also controls these configurations of the other dependencies:

Version:

The BOM version controls the version of any other CICS dependency in the same module or its child modules if the version of that dependency is not otherwise specified. Make sure that you specify a BOM version that provides the support for the other dependencies you need in your application, and that your target CICS system is at the same or newer CICS TS release and APAR maintenance level. In the snippet, the `version` tag specifies the BOM version, consisting of:

- the CICS version (`5.4`)
- the time stamp when the BOM is built (`20200519102234`)
- if relevant, the version of the CICS TS APAR (PH25409), which includes server-side updates to the libraries

The `enforcedPlatform` keyword ensures that the version specified in the BOM overrides any other versions found in the dependency graph.

Note: `enforcedPlatform` is supported from Gradle 5.0.

Scope:

Similar to the `provided` scope in Maven, Gradle has a `compileOnly` configuration to ensure that the dependency is provided by the CICS TS runtime and not packaged with the module. However, you must specify consistent target configuration for both the BOM and the other dependencies. For example, declare dependency on the JCICS library (`com.ibm.cics.server`) with the BOM that has the `compileOnly` configuration as follows:

```
dependencies {
    compileOnly enforcedPlatform('com.ibm.cics:com.ibm.cics.ts.bom:5.4-20200519102234-
PH25409')
    compileOnly("com.ibm.cics:com.ibm.cics.server")  //dependency on JCICS
}
```

Likewise, if you declare dependency on the annotation processor (`com.ibm.cics.server.invocation`), you should also define a BOM with the `annotationProcessor` configuration. This is because the annotation processor dependency must use the `annotationProcessor` configuration.

Gradle `build.gradle`

```
dependencies {
    annotationProcessor enforcedPlatform('com.ibm.cics:com.ibm.cics.ts.bom:5.4-20200519102234-
PH25409')
    annotationProcessor ("com.ibm.cics:com.ibm.cics.server.invocation")  //dependency on
annotation processor
}
```

Note: You declare dependency on a BOM file for version control. The BOM itself does not import any library; you must also reference other libraries along with the BOM.

com.ibm.cics.server

All versions are available at com.ibm.cics.server on Maven Central.

Maven `pom.xml`

```
<dependencies>
    <dependency>
        <groupId>com.ibm.cics</groupId>
        <artifactId>com.ibm.cics.server</artifactId>
    </dependency>
</dependencies>
```

The version number and the scope configuration are omitted as they are inherited from the BOM. If you don't use a BOM, reference this library as follows, where the version number includes the OSGi `Bundle-Version`, the CICS release, and (if relevant) the APAR number.

```
<dependencies>
    <dependency>
        <groupId>com.ibm.cics</groupId>
```

```
        <artifactId>com.ibm.cics.server</artifactId>
        <version>1.700.1-5.4-PH25409</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

Gradle `build.gradle`

```
repositories {
    mavenCentral()
}

dependencies {
    compileOnly 'com.ibm.cics:com.ibm.cics.server'
}
```

The version number is omitted as they are inherited from the BOM. If you don't use a BOM, reference this library as follows, where the version number includes the OSGi `Bundle-Version`, the CICS release, and (if relevant) the APAR number.

```
repositories {
    mavenCentral()
}

dependencies {
    compileOnly 'com.ibm.cics:com.ibm.cics.server:1.700.1-5.4-PH25409'
}
```

com.ibm.cics.server.invocation.annotations

All versions are available at com.ibm.cics.server.invocation.annotations on Maven Central.

Maven `pom.xml`

```
<dependencies>
    <dependency>
        <groupId>com.ibm.cics</groupId>
        <artifactId>com.ibm.cics.server.invocation.annotations</artifactId>
    </dependency>
</dependencies>
```

The version number and scope configuration are omitted because they are inherited from the BOM. If you don't use a BOM, also specify the version number and `<scope>provided</scope>` for this dependency.

The version number includes the CICS release and (if relevant) the CICS TS APAR number.

Gradle `build.gradle`

```
repositories {
    mavenCentral()
}

dependencies {
    compileOnly 'com.ibm.cics:com.ibm.cics.server.invocation.annotations'
}
```

The version number is omitted because it's inherited from the BOM. If you don't use a BOM, also specify a version number for this dependency.

The version number includes the CICS release and (if relevant) the CICS TS APAR number.

com.ibm.cics.server.invocation

All versions are available at com.ibm.cics.server.invocation on Maven Central.

You're recommended to use a separate processor path for annotation processors, rather than adding them directly to the class path. For this reason, the configuration for `com.ibm.cics.server.invocation` differs from the other artifacts.

Maven `pom.xml`

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <annotationProcessorPaths>
                    <annotationProcessorPath>
                        <groupId>com.ibm.cics</groupId>
                        <artifactId>com.ibm.cics.server.invocation</artifactId>
                        <version>5.4-PH25409</version>
                    </annotationProcessorPath>
                </annotationProcessorPaths>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Note: Maven If you're using Maven, you need to specify the version number of the artifact even if you use a BOM. The version number includes the CICS release and (if relevant) the CICS TS APAR number.

Gradle `build.gradle`

```
repositories {
    mavenCentral()
}

dependencies {
    annotationProcessor 'com.ibm.cics:com.ibm.cics.server.invocation'
}
```

The version number is omitted because it's inherited from the BOM. Make sure that the BOM is defined with the same `annotationProcessor` configuration. If you don't use a BOM, specify a version number for this dependency. The version number includes the CICS release and (if relevant) the CICS TS APAR number.

What's next

You can reference the JCICS Javadoc information when writing the code. After you finish writing the application code, you can build the applications and integrate them into your build toolchain in the same way as you build and deploy other Maven or Gradle modules.

Manually importing Java libraries

When creating your own build scripts, if you want to manually import Java libraries to resolve dependencies instead of using the IBM CICS SDK for Java or Maven Central artifacts, you can copy the `.jar` files out of the CICS installation directories.

Note: Copying `.jar` files manually makes them prone to get out of sync with updates. You can use the artifacts on Maven Central to ensure that you always have the correct version of libraries. Otherwise, you must have a mechanism to refresh the copied `.jar` files. A full refresh is required when a new release of CICS is installed.

The `.jar` files are located within the `/lib` directory of the CICS installation on z/OS.

The application development `.jar` files are:

- `com.ibm.cics.server.invocation.jar`, which provides support for the CICS annotation processor, a Java library that is used to create metadata that enables CICS programs to invoke Java applications in a Liberty JVM server.
- `com.ibm.cics.server.invocation.annotations.jar`, which provides support for CICS annotations, a Java library that provides the `@CICSProgram` annotation to enable CICS programs to invoke Java applications in a Liberty JVM server.
- `com.ibm.cics.server.jar`, which provides support for the JCICS API.

- `com.ibm.record.jar`, which contains the Java API for legacy programs that use IByteBuffer from the Java Record Framework that came with VisualAge.

Considerations for a shared JVM

When you are developing Java applications to run in CICS, be aware that changes to shared resources within the JVM might be seen by all running applications and threads. Ensure that your applications do not leave the JVM in an unexpected state that other applications might rely on.

The following points are important considerations to think about:

- If your application resets the default time zone, other applications that use the same JVM server will use the new default time zone, which might be unexpected.
- Do not use **`System.exit()`** in your applications. Using **`System.exit()`** causes both the JVM server and CICS to shut down.
- Ensure that your applications are Threadsafe. Static variables that are shared between applications need careful review to ensure that there is no cross contamination between applications. A typical pattern to ensure uniqueness, is to use ThreadLocal variables.
- If objects are referenced by static variables, they are not candidates for garbage collection. In a JVM server, static state persists for all applications until the JVM server is disabled by the system programmer.
- It is possible to have multiple connections to DB2 from different applications. Therefore, when a task finishes with DB2, it is best practice to close the connection even if that connection is later deleted when the task completes.
- Sockets created using classes from the `java.net` package are not CICS domain sockets and cannot be managed or monitored by CICS.

Java development using JCICS

You can write Java applications that use the CICS Java class library (JCICS) to access CICS services. JCICS is the Java equivalent of the **`EXEC CICS`** application programming interface (API) that is provided for other CICS supported languages, such as COBOL.

Using JCICS, you can write Java applications that access CICS resources and integrate with programs written in other languages. Most of the functions of the **`EXEC CICS`** API are supported. You can get the JCICS library from any of the following places:

- The `com.ibm.cics:com.ibm.cics.server` artifact on Maven Central
- The `com.ibm.cics.server.jar` file supplied with CICS in the USSHOME directory
- The IBM CICS SDK for Java

Consuming JCICS from OSGi environments

The `com.ibm.cics.server` package (JCICS) will increment in version number when there are API additions, API removals, or bug-fixes during service or development work. Version increments are not guaranteed on a release boundary. Versions, and which CICS release they apply to, are described in Package com.ibm.cics.server.

It is prudent to declare Imports as a compatible range, beginning at your applications minimum supported level, up to (but not including), the next breaking API change. For example: `Import-Package: com.ibm.cics.server;version="[1.600.0,2.0.0)"`

The Java class library for CICS (JCICS)

JCICS supports most of the functions of the **`EXEC CICS`** API commands.

The JCICS classes are fully documented in Javadoc that is generated from the class definitions. The Javadoc is available at JCICS Javadoc information.

JavaBeans

Some of the classes in JCICS can be used as JavaBeans , which means that they can be customized in an application development tool such as Eclipse, serialized, and manipulated using the JavaBeans API.

The following JavaBeans are available in JCICS:

- Program
- ESDS
- KSDS
- RRDS
- TDQ
- TSQ
- AttachInitiator
- EnterRequest

These beans do not define any events; they consist of properties and methods. They can be instantiated at run time in one of three ways:

- By calling the new method for the class itself. This method is preferred.
- By calling `Beans.instantiate()` for the name of the class, with property values set manually.
- By calling `Beans.instantiate()` of a `.ser` file, with property values set at design time.

If either of the first two options are chosen, the property values, including the name of the CICS resource, must be set by invoking the appropriate `set` methods at run time.

Library structure

Each JCICS library component falls into one of four categories: Interfaces, Classes, Exceptions, or Errors.

Interfaces
Some interfaces are provided to define sets of constants. For example, the TerminalSendBits interface provides a set of constants that can be used to construct a `java.util.BitSet`.

Classes
The supplied classes provide most of the JCICS function. The `API` class is an abstract class that provides common initialization for every class that corresponds to a part of the CICS API, except for ABENDs and exceptions. For example, the `Task` class provides a set of methods and variables that correspond to a CICS task.

Errors and Exceptions
The Java language defines both exceptions and errors as subclasses of the class `Throwable` . JCICS defines `CicsError` as a subclass of `Error`. `CicsError` is the superclass for all the other CICS error classes, which are used for severe errors.

JCICS defines `CicsException` as a subclass of `Exception`. `CicsException` is the superclass for all the CICS exception classes (including the `CicsConditionException` classes such as `InvalidQueueIdException`, which represents the CICS QIDERR condition).

See “Error handling and abnormal termination ” on page 47 for further information.

CICS resources

CICS resources, such as programs or temporary storage queues, are represented by instances of the appropriate Java class, identified by the values of various properties such as the name of the resource.

You define CICS resources by using either the CICS Explorer, CEDA transactions, or the CICSPlex® SM WUI. To use implicit remote access, you define a resource locally that points to a remote resource.

For more information on defining CICS resources, see Resource definitions.

Arguments for passing data

You can pass data between programs using channels and containers, or by using a communication area (COMMAREA).

If you use a COMMAREA, you are limited to passing 32 KB at a time. If you use a channel and containers, you can pass more than 32 KB between programs. The COMMAREA or channel, and any other parameters, are passed as arguments to the appropriate methods.

Many of the methods are overloaded; that is, they have different versions that take either a different number of arguments or arguments of a different type. There might be one method that has no arguments, or the minimum mandatory arguments, and another that has all of the arguments. For example, the `Program` class includes the following different `link()` methods:

`link()`
This method does a simple LINK without using a COMMAREA to pass data, nor any other options.

`link(com.ibm.cics.server.CommAreaHolder)`
This method does a simple LINK, using a COMMAREA to pass data but without any other options.

`link(com.ibm.cics.server.CommAreaHolder, int)`
This method does a distributed LINK, using a COMMAREA to pass data and a DATALENGTH value to specify the length of the data within the COMMAREA.

`link(com.ibm.cics.server.Channel)`
This method does a LINK using a channel to pass data in one or more containers.

Serializable classes

A list of the JCICS serializable classes.

- AddressResource
- AttachInitiator
- CommAreaHolder
- EnterRequest
- ESDS
- File
- KeyedFile
- KSDS
- NameResource
- Program
- RemotableResource
- Resource
- RRDS
- StartRequest
- SynchronizationResource
- SyncLevel
- TDQ
- TSQ
- TSQType

Task.out and Task.err

For each Java-related CICS task, CICS automatically creates two Java PrintWriters classes that can be used as standard out and standard error streams. The standard out and standard error streams are public fields in the Task class called out and err.

If a CICS task is being driven from a terminal (the terminal is called a *principal facility* in this case), CICS maps the standard out and standard error streams to the task's terminal.

If the task does not have a terminal as its principal facility, the standard out and standard error streams are sent to System.out and System.err.

Threads

In a JVM server environment, an application that is running in an OSGi framework can use an ExecutorService to create threads that run on CICS tasks asynchronously.

CICS provides an implementation of the Java ExecutorService interface. This implementation creates threads that can use the JCICS API to access CICS services. The JVM server registers the CICS ExecutorService as an OSGi service on startup. Use this service instead of the Java Thread class to create tasks that can use JCICS.

The ExecutorService that is provided by CICS is registered as high priority in the OSGi framework, so that it can be used by applications to create threads. Typically, an application uses the highest priority ExecutorService, unless it filters services to use a specific implementation.

If you want to create threads in your application, the preferred method is to use a generic ExecutorService from the OSGi registry. The OSGi registry automatically uses the CICS ExecutorService to create CICS threads when the application is running in a JVM server. This approach means that the application is decoupled from the implementation, so you do not have to use the JCICS API method to create threads.

However, if you are writing an application that is specific to CICS, you can choose to use a CICSExecutorService class in the JCICS API to request new threads.

CICSExecutorService

This class implements the java.util.concurrent.ExecutorService interface. The CICSExecutorService class provides a static method that is called runAsCICS() that you can use to send a Runnable or Callable Java object for execution on a new JCICS enabled thread. The runAsCICS() method is a utility method which performs the OSGi registry look-up to obtain an instance of a CICSExecutorService for the application.

For work that is spawned from a parent CICS thread, a new CICS task is created, and runs under the task user ID and transaction ID inherited from the parent. If the work is spawned from a non- CICS thread, the default CJSA transaction ID and default CICS user ID are used. If you want to guarantee the new task runs under a transaction ID of your choice, then your Runnable or Callable object should implement the CICSTransactionRunnable or CICSTransactionRunnable interface.

```
CICSExecutorService.runAsCICS(Runnable runnable)
```

```
CICSExecutorService.runAsCICS(Callable callable)
```

Restrictions

For applications that are not running in an OSGi framework, for example an Axis2 Java program, you can access JCICS only on the initial application thread as the ExecutorService is not available. Additionally, you must ensure that all threads other than the initial thread finish before you take any of the following actions:

- link methods in class com.ibm.cics.server.Program

- `setNextTransaction(String)` method in class `com.ibm.cics.server.TerminalPrincipalFacility`
- `setNextCOMMAREA(byte[])` method in class `com.ibm.cics.server.TerminalPrincipalFacility`
- `commit()` method in class `com.ibm.cics.server.Task`
- `rollback()` method in class `com.ibm.cics.server.Task`
- Returning an `AbendException` exception from class `com.ibm.cics.server`

Data encoding

The JVM can use a different code page from CICS for character encoding; CICS must always use an EBCDIC code page, but the JVM can use another encoding such as ASCII. When you are developing an application that uses the JCICS API, you must ensure that you use the correct encoding.

The JCICS API uses the code page that is specified in the CICS region and not the underlying JVM. So if the JVM uses a different file encoding, your application must handle different code pages. To help you determine which code page CICS is using, CICS provides several Java properties:

- The **`com.ibm.cics.jvmserver.supplied.ccsid`** property returns the code page that is specified for the CICS region. By default, the JCICS API uses this code page for its character encoding. However, this value can be overridden in the JVM server configuration.
- The **`com.ibm.cics.jvmserver.override.ccsid`** property returns the value of an override in the JVM profile. The value is a code page that the JCICS API uses for its character encoding, instead of the code page that is used by the CICS region.
- The **`com.ibm.cics.jvmserver.local.ccsid`** property returns the code page that the JCICS API is using for character encoding in the JVM server.

You cannot set any of these properties in your Java application to change the encoding for JCICS. To change the code page, you must ask a system administrator to update the JVM profile to add the JVM system property **`-Dcom.ibm.cics.jvmserver.override.ccsid`**.

Encoding example

Any JCICS methods that accept `java.lang.String` parameters as input are automatically encoded with the correct code page before the data passes to CICS . Similarly, any `java.lang.String` values that are returned from the JCICS API are encoded in the correct code page. The JCICS API provides helper methods in most classes; these helper methods work with strings and data to determine and set the code page on behalf of the application.

If your application uses the `String.getBytes()` or `new String(byte[] bytes)` methods, the application must ensure it uses the correct encoding. If you want to use these methods in your application, you can use the Java property to encode the data correctly:

```
String.getBytes(System.getProperty("com.ibm.cics.jvmserver.local.ccsid"))
String(bytes, System.getProperty("com.ibm.cics.jvmserver.local.ccsid"))
```

The following example shows how to use the JCICS encoding when the application reads a field from a COMMAREA:

```
public static void main(CommAreaHolder ca)
{
    //Convert first 8 bytes of ca into a String using JCICS encoding
    String str=new String(ca.getValue(), 0, 8, System.getProperty("com.ibm.cics.jvmserver.local.ccsid"));
}
```

JCICS API services and examples

CICS supports a range of APIs and services for Java applications. Many of the services available to non-Java programs through the EXEC CICS API are available to Java programs through the JCICS API along with the standard Java SE APIs provided by the Java SDK.

These topics provide details on the JCICS services and the integration with Java exception handling. Other JEE APIs are available in the Liberty JVM server; for further details, see “ Developing Java applications to run in a Liberty JVM server ” on page 67.

CICS exception handling in Java programs

CICS ABENDs and exceptions are integrated into the Java exception-handling architecture to handle problems that occur in CICS.

All regular CICS ABENDs are mapped to a single Java exception, `AbendException`, whereas each CICS condition is mapped to a separate Java exception. This leads to an ABEND-handling model in Java that is similar to the other programming languages; a single handler is given control for every ABEND, and the handler must query the particular ABEND and then decide what to do.

If the exception representing a condition is caught by CICS itself, it is turned into an ABEND.

Java exception-handling is fully integrated with the ABEND and condition-handling in other languages, so that ABENDs can propagate between Java and non-Java programs, in the standard language-independent way. A condition is mapped to an ABEND before it leaves the program that caused or detected the condition.

However, there are several differences to the abend-handling model for other programming languages, resulting from the nature of the Java exception-handling architecture and the implementation of some of the technology underlying the Java API:

- ABENDs that cannot be handled in other programming languages can be caught in Java programs. These ABENDs typically occur during sync point processing. To avoid these ABENDs interrupting Java applications, they are mapped to an extension of an unchecked exception; therefore they do not have to be declared or caught.
- Several internal CICS events, such as program termination, are also mapped to Java exceptions and can therefore be caught by a Java application. Again, to avoid interrupting the normal case, these events are mapped to extensions of an unchecked exception and do not have to be caught or declared.

Three class hierarchies of exceptions relate to CICS :

1. `CicsError` extends `java.lang.Error` and is the base for `AbendError` and `UnknownCicsError`.
2. `CicsRuntimeException` extends `java.lang.RuntimeException` and is in turn extended by:

   `AbendCancelException`
   Represents a CICS ABEND CANCEL.

   `AbendException`
   Represents a normal CICS ABEND.

   `EndOfProgramException`
   Indicates that a linked-to program has terminated normally.
3. `CicsException` extends `java.lang.Exception` and has the subclass:

   `CicsConditionException`
   The base class for all CICS conditions.

CICS error-handling commands

The way that the **EXEC CICS** error-handling commands are supported in Java is described.

CICS condition handling is integrated into the Java exception-handling architecture, as described in “ CICS exception handling in Java programs ” on page 42. The equivalent **EXEC CICS** command is supported in Java in the following ways:

HANDLE ABEND
To handle an ABEND generated by a program in any CICS supported language, use a Java try-catch statement, with `AbendException` appearing in a catch clause.

HANDLE CONDITION
To handle a specific condition, such as PGMIDERR, use a catch clause that names the appropriate exception; in this case `InvalidProgramException`. Alternatively, use a catch clause that names `CicsConditionException` , if all CICS conditions are to be caught.

IGNORE CONDITION
This command is not relevant in Java applications.

POP HANDLE and PUSH HANDLE
These commands are not relevant in Java applications. The Java exceptions that are used to represent CICS ABENDs and conditions are caught by any catch block in scope.

CICS conditions

The condition-handling model in Java is different from other CICS programming languages.

In COBOL, you can define an exception-handling label for each condition. If that condition occurs during the processing of a CICS command, control transfers to the label.

In C and C++, you cannot define an exception-handling label for a condition; to detect a condition, the RESP field in the EIB must be checked after each CICS command.

In Java , any condition returned by a CICS command is mapped into a Java exception. You can include all CICS commands in a try-catch block and do specific processing for each condition, or have a single null catch clause if the particular exception is not relevant. Alternatively, you can let the condition propagate, to be handled by a catch clause at a larger scope.

CICS exception handling in Java Web applications

CICS ABENDs and exceptions are integrated into the Java exception-handling architecture for the Liberty Web container to handle problems that occur in CICS applications. Any Java exception that is not handled by a Web application will be caught by the Web container and drive the servlet exception handling process. As part of this processing any uncommitted CICS units of work will be rolled back by CICS.

All Java Web applications that extend the HttpServlet interface must handle all checked exceptions apart from `IOException` or `ServletException` , as defined on the HttpServlet interface. Checked exceptions included all sub-classes of `com.ibm.cics.server.CicsConditionException` that represents unhandled CICS conditions. Therefore any exception handling code that catches CICS conditions must identify any error conditions which require a unit-of-work to be rolled back and either explicitly call syncpoint rollback using the rollback() method on the Task object, as illustrated in the example, or throw an `AbendException`.

```
try
{
    TSQ tsqQ = new TSQ();
    tsqQ.setName("tsq1");
    tsqQ.writeString("input data");

} catch (IOErrorException e) {
    // Log error
    try
    {
        Task.getTask().rollback();

        } catch (InvalidRequestException e1) {
            throw new RuntimeException(e1);
        }
    }
}
```

Unchecked Java exceptions, which are sub-classes of java.lang.RuntimeException, can be thrown by any Java application including Web applications, and include `com.ibm.cics.server.AbendException` and `com.ibm.cics.server.AbendCancelException`. Therefore any Web application that throws an

AbendException or does not handle a transaction abend will drive the servlet exception handling process and associated unit-of-work rollback processing.

Web applications that commit units-of-work using the Java Transaction API (JTA) will be committed according to the control of the Liberty Transaction Manager. For further details see “ Java Transaction API (JTA) ” on page 80.

JCICS exception mapping

In Java, a condition returned by a CICS command is mapped into a Java exception.

Table 3. Java exception mapping

CICS condition	Java Exception
ALLOCERR	AllocationErrorException
CBIDERR	InvalidControlBlockIdException
CCSIDERR	CCSIDErrorException
CHANNELERR	ChannelErrorException
CONTAINERERR	ContainerErrorException
DISABLED	FileDisabledException ResourceDisabledException
DSIDERR	FileNotFoundException
DSSTAT	DestinationStatusChangeException
DUPKEY	DuplicateKeyException
DUPREC	DuplicateRecordException
END	EndException
ENDDATA	EndOfDataException
ENDFILE	EndOfFileException
ENDINPT	EndOfInputIndicatorException
ENQBUSY	ResourceUnavailableException
ENVDEFERR	InvalidRetrieveOptionException
EOC	EndOfChainIndicatorException
EODS	EndOfDataSetIndicatorException
EOF	EndOfFileIndicatorException
ERROR	ErrorException
EXPIRED	TimeExpiredException
FILENOTFOUND	FileNotFoundException
FUNCERR	FunctionErrorException
IGREQID	InvalidREQIDPrefixException
IGREQCD	InvalidDirectionException
ILLOGIC	LogicException
INBFMH	InboundFMHException

Table 3. Java exception mapping (continued)

CICS condition	Java Exception
INVERRTERM	InvalidErrorTerminalException
INVEXITREQ	InvalidExitRequestException
INVLDC	InvalidLDCException
INVMPSZ	InvalidMapSizeException
INVPARTNSET	InvalidPartitionSetException
INVPARTN	InvalidPartitionException
INVREQ	InvalidRequestException
INVTSREQ	InvalidTSRequestException
IOERR	IOErrorException
ISCINVREQ	ISCInvalidRequestException
ITEMERR	ItemErrorException
JIDERR	InvalidJournalIdException
LENGERR	LengthErrorException
MAPERROR	MapErrorException
MAPFAIL	MapFailureException
NAMEERROR	NameErrorException
NODEIDERR	InvalidNodeIdException
NOJBUFSP	NoJournalBufferSpaceException
NONVAL	NotValidException
NOPASSBKRD	NoPassbookReadException
NOPASSBKWR	NoPassbookWriteException
NOSPACE	NoSpaceException
NOSPOOL	NoSpoolException
NOSTART	StartFailedException
NOSTG	NoStorageException
NOTALLOC	NotAllocatedException
NOTAUTH	NotAuthorisedException
NOTFINISHED	NotFinishedException
NOTFND	RecordNotFoundException NotFoundException
NOTOPEN	NotOpenException
OPENERR	DumpOpenErrorException
OVERFLOW	MapPageOverflowException
PARTNFAIL	PartitionFailureException

Table 3. Java exception mapping (continued)	
CICS condition	**Java Exception**
PGMIDERR	InvalidProgramIdException
QBUSY	QueueBusyException
QIDERR	InvalidQueueIdException
QZERO	QueueZeroException
RDATT	ReadAttentionException
RETPAGE	ReturnedPageException
ROLLEDBACK	RolledBackException
RTEFAIL	RouteFailedException
RTESOME	RoutePartiallyFailedException
SELNERR	DestinationSelectionErrorException
SESSBUSY	SessionBusyException
SESSIONERR	SessionErrorException
SIGNAL	InboundSignalException
SPOLBUSY	SpoolBusyException
SPOLERR	SpoolErrorException
STRELERR	STRELERRException
SUPPRESSED	SuppressedException
SYMBOLERR	SymbolErrorException
SYSBUSY	SystemBusyException
SYSIDERR	InvalidSystemIdException
TASKIDERR	InvalidTaskIdException
TCIDERR	TCIDERRException
TEMPLATERR	TemplateErrorException
TERMERR	TerminalException
TERMIDERR	InvalidTerminalIdException
TOKENERR	TokenErrorException
TRANSIDERR	InvalidTransactionIdException
TSIOERR	TSIOErrorException
UNEXPIN	UnexpectedInformationException
USERIDERR	InvalidUserIdException
WRBRK	WriteBreakException
WRONGSTAT	WrongStatusException

Note: `NonHttpDataException` is thrown by `getContent()` if the CICS command WEB RECEIVE indicates that the data received is a non-HTTP message (by setting TYPE=HTTPNO).

Error handling and abnormal termination

To initiate an ABEND from a Java program, you must invoke one of the `Task.abend()` or `Task.forceAbend()` methods.

Methods	JCICS class	EXEC CICS commands
`abend()` , `forceAbend()`	Task	ABEND

ABEND

To initiate an ABEND from a Java program, invoke one of the `Task.abend()` methods . This causes an abend condition to be set in CICS and an `AbendException` to be thrown. If the `AbendException` is not caught within a higher level of the application object, or handled by an ABEND-handler registered in the calling program (if any), CICS terminates and rolls back the transaction.

The different `abend()` methods are:

- abend (String *abcode*), which causes an ABEND with the ABEND code *abcode*.
- abend (String *abcode* , boolean *dump*), which causes an ABEND with the ABEND code *abcode*. If the **dump** parameter is false, no dump is taken.
- `abend()` , which causes an ABEND with no ABEND code and no dump.

ABEND CANCEL

To initiate an ABEND that cannot be handled, invoke one of the `Task.forceAbend()` methods . As described above, this causes an `AbendCancelException` to be thrown which can be caught in Java programs. If you do so, you must re-throw the exception to complete **ABEND_CANCEL** processing, so that, when control returns to CICS, CICS will terminate and roll back the transaction. Only catch the `AbendCancelException` for notification purposes and then re-throw it.

The different `forceAbend()` methods are:

- `forceAbend` (String *abcode*), which causes an **ABEND CANCEL** with the ABEND code *abcode*.
- `forceAbend` (String *abcode* , boolean *dump*), which causes an **ABEND CANCEL** with the ABEND code *abcode*. If the **dump** parameter is false, no dump is taken.
- `forceAbend()` , which causes an **ABEND CANCEL** with no ABEND code and no dump.

APPC mapped conversations

APPC unmapped conversation support is not available from the JCICS API.

APPC mapped conversations:

Methods	JCICS class	EXEC CICS Commands
`initiate()`	AttachInitiator	ALLOCATE, CONNECT PROCESS
`converse()`	Conversation	CONVERSE
`get*()` methods	Conversation	EXTRACT ATTRIBUTES
`get*()` methods	Conversation	EXTRACT PROCESS
`free()`	Conversation	FREE
`issueAbend()`	Conversation	ISSUE ABEND
`issueConfirmation()`	Conversation	ISSUE CONFIRMATION
`issueError()`	Conversation	ISSUE ERROR
`issuePrepare()`	Conversation	ISSUE PREPARE
`issueSignal()`	Conversation	ISSUE SIGNAL

Methods	JCICS class	EXEC CICS Commands
`receive()`	Conversation	RECEIVE
`send()`	Conversation	SEND
`flush()`	Conversation	WAIT CONVID

Basic Mapping Support (BMS)

Basic mapping support (BMS) is an application programming interface between CICS programs and terminal devices. JCICS provides support for some of the BMS application programming interface.

Methods	JCICS class	EXEC CICS Commands
`sendControl()`	TerminalPrincipalFacility	SEND CONTROL
`sendText()`	TerminalPrincipalFacility	SEND TEXT
	Not supported	SEND MAP, RECEIVE MAP

Channel and container examples

Containers are named blocks of data designed for passing information between programs. Containers are grouped in sets called *channels*. This information explains how you can use channels and containers in your Java application and provides some code examples.

For introductory information about channels and containers, and guidance about using channels in non-Java applications, see Transferring data between programs using channels. For information about tools that allow Java programs to access existing CICS application data, see "Interacting with structured data from Java" on page 131.

Table 4 on page 48 lists the classes and methods that implement JCICS support for channels and containers.

Table 4. JCICS support for channels and containers

Methods	JCICS class	EXEC CICS Commands
`containerIterator()`	Channel	STARTBROWSE CONTAINER
`createContainer()`	Channel	
`delete()`	Channel	DELETE CHANNEL
`deleteContainer()`	Channel	DELETE CONTAINER CHANNEL
`getContainer()`	Channel	
`getContainerCount()`	Channel	QUERY CHANNEL
`getName()`	Channel	
`delete()`	Container	DELETE CONTAINER CHANNEL
`get()`	Container	GET CONTAINER CHANNEL
`getLength()`	Container	GET CONTAINER CHANNEL NODATA
`getDatatype()`	Container	
`getName()`	Container	
`put()`	Container	PUT CONTAINER CHANNEL
`getOwner()`	ContainerIterator	

Table 4. JCICS support for channels and containers (continued)		
Methods	**JCICS class**	**EXEC CICS Commands**
`hasNext()`	ContainerIterator	
`next()`	ContainerIterator	GETNEXT CONTAINER BROWSETOKEN
`remove()`	ContainerIterator	
`link()`	Program	LINK
`setNextChannel()`	TerminalPrincipalFacility	RETURN CHANNEL
`issue()`	StartRequest	START CHANNEL
`createChannel()`	Task	
`getCurrentChannel()`	Task	ASSIGN CHANNEL
`containerIterator()`	Task	STARTBROWSE CONTAINER

The CICS condition CHANNELERR results in a `ChannelErrorException` being thrown; the CONTAINERERR CICS condition results in a `ContainerErrorException` ; the CCSIDERR CICS condition results in a `CCSIDErrorException`.

Creating channels and containers in JCICS

To create a channel, use the `createChannel()` method of the `Task` class.

For example:

```
Task t=Task.getTask();
Channel custData = t.createChannel("Customer_Data");
```

The string supplied to the `createChannel` method is the name by which the `Channel` object is known to CICS. (The name is padded with spaces to 16 characters, to conform to CICS naming conventions.)

To create a new container in the channel, use the `Channel` `createContainer()` method. For example:

```
Container custRec = custData.createContainer("Customer_Record");
```

The string supplied to the `createContainer()` method is the name by which the `Container` object is known to CICS. The name is padded with spaces to 16 characters, if necessary, to conform to CICS naming conventions. If a container of the same name already exists in this channel, a `ContainerErrorException` is thrown.

Putting data into a container

To put data into a `Container` object, use the `Container.put()` method.

Data can be added to a container as a string. For example:

```
String custNo = "00054321";
byte[] custRecIn = custNo.getBytes();
custRec.put(custRecIn);
```

Or :

```
custRec.putString("00054321");
```

Passing a channel to another program or task

To pass a channel on a program-link use the `link()` method of the `Program` class.

```
programX.link(custData);
```

To set the next channel on a program-return call, use the `setNextChannel()` method of the `TerminalPrincipalFacility` class:

```
terminalPF.setNextChannel(custData);
```

To pass a channel on a START request, use the `issue` method of the `StartRequest` class:

```
startrequest.issue(custData);
```

Receiving the current channel

It is not necessary for a program to receive its current channel explicitly. However, a program can get its current channel from the current task.

If a program gets the current channel from the current task, the task can extract containers by name:

```
Task t = Task.getTask();
Channel custData = t.getCurrentChannel();

if (custData != null) {
    Container custRec = custData.getContainer("Customer_Record");
} else {
    System.out.println("There is no Current Channel");
}
```

Getting data from a container

Use the `Container.get()` method to read the data in a container into a byte array.

```
byte[] custInfo = custRec.get();
```

Browsing the current channel

A JCICS program that is passed a channel can access all of the `Container` objects without receiving the channel explicitly.

To do this, it uses a `ContainerIterator` object. The `ContainerIterator` class implements the `java.util.Iterator` interface. When a `Task` object is instantiated from the current task, its `containerIterator()` method returns an `Iterator` for the current channel, or null if there is no current channel. For example:

```
Task t = Task.getTask();
ContainerIterator ci = t.containerIterator();

while (ci.hasNext()) {
    Container custData = ci.next();
    // Process the container...
}
```

Channel and containers example

This example shows an excerpt of a Java class called `Payroll` that calls a COBOL server program named PAYR. The `Payroll` class uses the JCICS `com.ibm.cics.server.Channel` and `com.ibm.cics.server.Container` classes to work with a channel and its containers.

```
import com.ibm.cics.server.*;

public class Payroll
{
    ...
    Task t=Task.getTask();

    // create the payroll_2004 channel
    Channel payroll_2004 = t.createChannel("payroll-2004");

    // create the employee container
    Container employee = payroll_2004.createContainer("employee");

    // put the employee name into the container
    employee.putString("John Doe");

    // create the wage container
    Container wage = payroll_2004.createContainer("wage");

    // put the wage into the container
    wage.putString("2000");

    // Link to the PAYROLL program, passing the payroll_2004 channel
    Program p = new Program();
    p.setName("PAYR");
    p.link(payroll_2004);

    // Get the status container which has been returned
    Container status = payroll_2004.getContainer("status");

    if (status != null)
    {
        // Get the status information
        byte[] payrollStatus = status.get();
    }

    ...
}
```

Figure 1. Java class that uses the JCICS `com.ibm.cics.server.Channel` and `com.ibm.cics.server.Container` classes to pass a channel to a COBOL server program

Diagnostic services

The JCICS application programming interface has support for these CICS trace and dump commands.

Methods	JCICS class	EXEC CICS Commands
	Not supported	DUMP
`enterTrace()`	EnterRequest	ENTER

Document services

This section describes JCICS support for the commands in the DOCUMENT application programming interface.

Class `Document` maps to the **EXEC CICS DOCUMENT** API.

The default no-argument constructor for class `Document` creates a new document in CICS. The constructor `Document(byte[] docToken)` accepts a document token for an existing document that has previously been created. For example, another program can create a document and pass its document token to the Java application in a COMMAREA or container.

Constructors for class `DocumentLocation` map to the AT and TO keywords of the EXEC CICS DOCUMENT API.

Setters and getters for class `SymbolList` map to the SYMBOLLIST, LENGTH, DELIMITER, and UNESCAPE keywords of the **EXEC CICS DOCUMENT** API.

Methods	JCICS class	EXEC CICS Commands
`create*()`	Document	DOCUMENT CREATE
`append*()`	Document	DOCUMENT INSERT
`insert*()`	Document	DOCUMENT INSERT
`addSymbol()`	Document	DOCUMENT SET
`setSymbolList()`	Document	DOCUMENT SET
`retrieve*()`	Document	DOCUMENT RETRIEVE
`get*()`	Document	DOCUMENT

Environment services

CICS environment services provide access to CICS data areas, parameters, and resource attributes that are relevant to an application program.

The **EXEC CICS** commands and options that have equivalent JCICS support are:

- ADDRESS
- ASSIGN
- INQUIRE SYSTEM
- INQUIRE TASK
- INQUIRE TERMINAL/NETNAME

ADDRESS

The following support is provided for the **ADDRESS** API command options.

For complete information about the **EXEC CICS ADDRESS** command, see ADDRESS.

ACEE
The Access Control Environment Element (ACEE) is created by an external security manager when a CICS user signs on. This option not supported in JCICS.

COMMAREA
A COMMAREA contains user data that is passed with a command. The COMMAREA pointer is passed automatically to the linked program by the **CommAreaHolder** argument. See "Arguments for passing data" on page 39 for more information.

CWA
The Common Work Area (CWA) contains global user data, sharable between tasks. A copy of the CWA can be obtained using the `getCWA()` method of the `Region` class.

EIB
The EXEC interface block (EIB) contains information about the CICS command last executed. See EIB fields. Access to EIB values is provided by methods on the appropriate objects. For example,

eibtrnid
is returned by the `getTransactionName()` method of the `Task` class.

eibaid
is returned by `the getAIDbyte()` method of the `TerminalPrincipalFacility` class.

eibcposn
is returned by the `getRow()` and `getColumn()` methods of the `Cursor` class.

TCTUA
The Terminal Control Table User Area (TCTUA) contains user data associated with the terminal that is driving the CICS transaction (the principal facility). This area is used to pass information

between application programs, but only if the same terminal is associated with the application programs involved. The contents of the TCTUA can be obtained using the `getTCTUA()` method of the `TerminalPrincipalFacility` class.

TWA
The Transaction Work Area (TWA) contains user data that is associated with the CICS task. This area is used to pass information between application programs, but only if they are in the same task. A copy of the TWA can be obtained using the `getTWA()` method of the `Task` class.

ASSIGN

The following support is provided for the **ASSIGN** API command options.

For detailed information about this command, see ASSIGN.

Methods	JCICS class
`getABCODE()`	AbendException
`getApplicationContext()`	Task
`getAPPLID()`	Region
`getCurrentChannel()`	Task
`getCWA()`	Region
`getName()`	TerminalPrincipalFacility or ConversationPrincipalFacility
`getFCI()`	Task
`getNetName()`	TerminalPrincipalFacility or ConversationPrincipalFacility
`getPrinSysid()`	TerminalPrincipalFacility or ConversationPrincipalFacility
`getProgramName()`	Task
`getQNAME()`	Task
`getSTARTCODE()`	Task
`getSysid()`	Region
`getTCTUA()`	TerminalPrincipalFacility
`getTERMCODE()`	TerminalPrincipalFacility
`getTWA()`	Task
`getUSERID()` , `Task.getUSERID()`	Task, TerminalPrincipalFacility or ConversationPrincipalFacility

No other ASSIGN options are supported.

INQUIRE SYSTEM

Support is provided for the **INQUIRE SYSTEM** SPI options.

Methods	JCICS class
`getAPPLID()`	Region
`getSYSID()`	Region

No other **INQUIRE SYSTEM** options are supported.

INQUIRE TASK

The following support is provided for the **INQUIRE TASK** API command options.

Methods	JCICS class
`getSTARTCODE()`	Task
`getTransactionName()`	Task
`getUSERID()`	Task

FACILITY
You can find the name of the task's principal facility by calling the `getName()` method on the task's principal facility, which can in turn be found by calling the `getPrincipalFacility()` method on the current `Task` object.

FACILITYTYPE
You can determine the type of facility by using the Java `instanceof` operator to check the class of the returned object reference.

No other INQUIRE TASK options are supported.

INQUIRE TERMINAL and INQUIRE NETNAME

The following support is provided for **INQUIRE TERMINAL** and **INQUIRE NETNAME** SPI options.

Methods	JCICS class
`getUSERID()`	Terminal, ConversationalPrincipalFacility
`Terminal.getUser()`	Terminal, ConversationalPrincipalFacility

You can also find the USERID value by calling the `getUSERID()` method on the current `Task` object, or on the object representing the task's principal facility.

No other **INQUIRE TERMINAL** or **INQUIRE NETNAME** options are supported.

File services

JCICS provides classes and methods that map to the **EXEC CICS** API commands for each type of CICS file and index.

For information about tools that allow Java programs to access existing CICS application data, see Interacting with structured data from Java.

CICS supports the following types of files:

- Key Sequenced Data Sets (KSDS)
- Entry Sequenced Data Sets (ESDS)
- Relative Record Data Sets (RRDS)

KSDS and ESDS files can have alternative (or secondary) indexes. CICS does not support access to an RRDS file through a secondary index. Secondary indexes are treated by CICS as though they were separate KSDS files in their own right, which means they have separate FD entries.

There are a few differences between accessing KSDS, ESDS (primary index), and ESDS (secondary index) files, which means that you cannot always use a common interface.

Records can be read, updated, deleted, and browsed in all types of file, with the exception that records cannot be deleted from an ESDS file.

See VSAM data sets: KSDS, ESDS, RRDS for more information about data sets.

Java commands that read data support only the equivalent of the SET option on **EXEC CICS** commands. The data returned is automatically copied from CICS storage to a Java object.

Categories of Java interfaces relating to File Control

The Java interfaces relating to File Control are in five categories:

File
The superclass for the other file classes; contains methods common to all file classes.

KeyedFile
Contains the interfaces common to a KSDS file accessed using the primary index, a KSDS file accessed using a secondary index, and an ESDS file accessed using a secondary index.

KSDS
Contains the interface specific to KSDS files.

ESDS
Contains the interface specific to ESDS files accessed through Relative Byte Address (RBA, its primary index) or Extended Relative Byte Address (XRBA). To use XRBA instead of RBA, issue the `setXRBA(true)` method.

RRDS
Contains the interface specific to RRDS files accessed through Relative Record Number (RRN, its primary index).

`File` and `FileBrowse` objects

For each file, there are two objects that can be operated on; the `File` object and the `FileBrowse` object.

`File` objects
The File object represents the file itself and can be used with methods to perform the following API operations:

- DELETE
- READ
- REWRITE
- UNLOCK
- WRITE
- STARTBR

A `File` object is created by the user application explicitly starting the required file class. The `FileBrowse` object represents a browse operation on a file. There can be more than one active browse against a specific file at any time, each browse being distinguished by a REQID. Methods can be instantiated for a `FileBrowse` object to perform the following API operations:

- ENDBR
- READNEXT
- READPREV
- RESETBR

`FileBrowse` objects
A `FileBrowse` object is not instantiated explicitly by the user application; it is created and returned to the user class by the methods that perform the STARTBR operation.

Mapping from JCICS classes and methods to CICS API commands

The following tables show how the JCICS classes and methods map to the **`EXEC CICS`** API commands for each type of CICS file and index. In these tables, the JCICS classes and methods are shown in the form `class.method()`. For example, `KeyedFile.read()` references the `read()` method in the `KeyedFile` class.

Classes and methods for keyed files

This table shows the classes and methods for keyed files.

Table 5. Classes and methods for keyed files

KSDS primary or secondary index class and method	ESDS secondary index class and method	CICS File API command
KeyedFile.read()	KeyedFile.read()	**READ**
KeyedFile.readForUpdate()	KeyedFile.readForUpdate()	**READ UPDATE**
KeyedFile.readGeneric()	KeyedFile.readGeneric()	**READ GENERIC**
KeyedFile.rewrite()	KeyedFile.rewrite()	**REWRITE**
KSDS.write()	KSDS.write()	**WRITE**
KSDS.delete()		**DELETE**
KSDS.deleteGeneric()		**DELETE GENERIC**
KeyedFile.unlock()	KeyedFile.unlock()	**UNLOCK**
KeyedFile.startBrowse()	KeyedFile.startBrowse()	**START BROWSE**
KeyedFile.startGenericBrowse()	KeyedFile.startGenericBrowse()	**START BROWSE GENERIC**
KeyedFileBrowse.next()	KeyedFileBrowse.next()	**READNEXT**
KeyedFileBrowse.previous()	KeyedFileBrowse.previous()	**READPREV**
KeyedFileBrowse.reset()	KeyedFileBrowse.reset()	**RESET BROWSE**
FileBrowse.end()	FileBrowse.end()	**END BROWSE**

Classes and methods for non-keyed files

This table shows the classes and methods for non-keyed files. ESDS and RRDS are accessed by their primary indexes.

Table 6. Classes and methods for non-keyed files

ESDS primary index class and method	RRDS primary index class and method	CICS File API command
ESDS.read()	RRDS.read()	**READ**
ESDS.readForUpdate()	RRDS.readForUpdate()	**READ UPDATE**
ESDS.rewrite()	RRDS.rewrite()	**REWRITE**
ESDS.write()	RRDS.write()	**WRITE**
	RRDS.delete()	**DELETE**
KeyedFile.unlock()	RRDS.unlock()	**UNLOCK**
ESDS.startBrowse()	RRDS.startBrowse()	**START BROWSE**
ESDS_Browse.next()	RRDS_Browse.next()	**READNEXT**
ESDS_Browse.previous()	RRDS_Browse.previous()	**READPREV**
ESDS_Browse.reset()	RRDS_Browse.reset()	**RESET BROWSE**
FileBrowse.end()	FileBrowse.end()	**END BROWSE**

Table 6. Classes and methods for non-keyed files (continued)		
ESDS primary index class and method	**RRDS primary index class and method**	**CICS File API command**
`ESDS.setXRBA()`		

Writing and reading data

Data to be written to a file must be in a Java byte array.

Data is read from a file into a `RecordHolder` object; the storage is provided by CICS and is released automatically at the end of the program.

You do not need to specify the **KEYLENGTH** value on any `File` method; the length used is the actual length of the key passed. When a `FileBrowse` object is created, it contains the length of the key specified on the `startBrowse` method, and this length is passed to CICS on subsequent browse requests against that object.

You do not need to provide a **REQID** for a browse operation; each browse object contains a unique REQID which is automatically used for all subsequent browse requests against that browse object.

Samples

In Github, sample CICS Java programs are provided to demonstrate how to use the JCICS API in an OSGi JVM server environment. In particular, use `com.ibm.cicsdev.vsam` for accessing KSDS, ESDS, and RRDS VSAM files.

HTTP and TCP/IP services

Getters in classes `HttpHeader`, `NameValueData`, and `FormField` return HTTP headers, name and value pairs, and form field values for the appropriate API commands.

Methods	JCICS class	EXEC CICS Commands
`get*()`	CertificateInfo	EXTRACT CERTIFICATE / EXTRACT TCPIP
`get*()`	HttpRequest	EXTRACT WEB
`getHeader()`	HttpRequest	WEB READ HTTPHEADER
`getFormField()`	HttpRequest	WEB READ FORMFIELD
`getContent()`	HttpRequest	WEB RECEIVE
`getQueryParm()`	HttpRequest	WEB READ QUERYPARM
`startBrowseHeader()`	HttpRequest	WEB STARTBROWSE HTTPHEADER
`getNextHeader()`	HttpRequest	WEB READNEXT HTTPHEADER
`endBrowseHeader()`	HttpRequest	WEB ENDBROWSE HTTPHEADER
`startBrowseFormField()`	HttpRequest	WEB STARTBROWSE FORMFIELD
`getNextFormField()`	HttpRequest	WEB READNEXT FORMFIELD
`endBrowseFormField()`	HttpRequest	WEB ENDBROWSE FORMFIELD
`startBrowseQueryParm()`	HttpRequest	WEB STARTBROWSE QUERYPARM
`getNextQueryParm()`	HttpRequest	WEB READNEXT QUERYPARM
`endBrowseQueryParm()`	HttpRequest	WEB ENDBROWSE QUERYPARM
`writeHeader()`	HttpResponse	WEB WRITE

Methods	JCICS class	EXEC CICS Commands
`getDocument()`	HttpResponse	WEB RETRIEVE
`getCurrentDocument()`	HttpResponse	WEB RETRIEVE
`sendDocument()`	HttpResponse	WEB SEND

Note: Use the method get `HttpRequestInstance()` to obtain the `HttpRequest` object.

Each incoming HTTP request processed by CICS Web support includes an HTTP header. If the request uses the POST HTTP verb it also includes document data. Each response HTTP request generated by CICS Web support includes an HTTP header and document data.

To process this JCICS provides the following Web and TCP/IP services:

HTTP Header
You can examine the HTTP header using the `HttpRequest` class. With HTTP in GET mode, if a client has filled in an HTTP form and selected the submit button, the query string is submitted.

SSL
CICS Web support provides the `TcpipRequest` class, which is extended by `HttpRequest` to obtain more information about which client submitted the request as well as basic information on the SSL support. If an SSL certificate is provided, you can use the `CertificateInfo` class to examine it in detail.

Documents
If a document is published to the server (HTTP POST), it is provided as a CICS document. You can access it by calling the `getDocument()` method on the `HttpRequest` class. See "Document services" on page 51 for more information about processing existing documents.

To serve the HTTP client web content resulting from a request, the server programmer needs to create a CICS document using the Document Services API and call the `sendDocument()` method.

For more information on CICS Web support see CICS Web support. For more information on the JCICS web classes see JCICS Javadoc information.

Program services

JCICS supports the CICS program control commands; LINK, RETURN, and INVOKE APPLICATION.

For information about tools that allow Java programs to access existing CICS application data, see Interacting with structured data from Java.

Table 7 on page 58 lists the methods and JCICS classes that map to CICS program control commands.

Table 7. Relationship between methods, JCICS classes, and CICS commands

EXEC CICS Commands	JCICS class	JCICS methods
LINK	Program	`link()`
RETURN	TerminalPrincipalFacility	`setNextTransaction()`, `setNextCOMMAREA()`, `setNextChannel()`
INVOKE APPLICATION	Application	`invoke()`

LINK
You can transfer control to another program that is defined to CICS by using the `link()` method. The target program can be in any language that is supported by CICS.

RETURN
Only the pseudoconversational aspects of this command are supported. It is not necessary to make a CICS call to return; the application can terminate as normal. The pseudoconversational functions are supported by methods in the `TerminalPrincipalFacility` class: `setNextTransaction()`

is equivalent to using the TRANSID option of RETURN; `setNextCOMMAREA()` is equivalent to using the COMMAREA option; while `setNextChannel()` is equivalent to using the CHANNEL option . These methods can be invoked at any time during the running of the program, and take effect when the program terminates.

INVOKE
Allows invocation of an application by naming an operation that corresponds to one of its program entry points, without having to know the name of the application entry point program and regardless of whether the program is public or private.

Note: The length of the COMMAREA provided is used as the LENGTH value for CICS. This value should not exceed 24 KB if the COMMAREA is to be passed between any two CICS servers (for any combination of product/version/release). This limit allows for the COMMAREA and space for headers.

Scheduling services

JCICS provides support for the CICS scheduling services, which let you retrieve data stored for a task, cancel interval control requests, and start a task at a specified time.

Methods	JCICS class	EXEC CICS Commands
`cancel()`	StartRequest	CANCEL
`retrieve()`	Task	RETRIEVE
`issue()`	StartRequest	START

To define what is to be retrieved by the `Task.retrieve()` method, use a `java.util.BitSet` object. The `com.ibm.cics.server.RetrieveBits` class defines the bits which can be set in the `BitSet` object; they are:

- RetrieveBits.DATA
- RetrieveBits.RTRANSID
- RetrieveBits.RTERMID
- RetrieveBits.QUEUE

These correspond to the options on the **EXEC CICS** RETRIEVE command.

The `Task.retrieve()` method retrieves up to four different pieces of information in a single invocation, depending on the settings of the `RetrieveBits`. The DATA, RTRANSID, RTERMID and QUEUE data are placed in a `RetrievedData` object, which is held in a `RetrievedDataHolder` object. The following example retrieves the data and transid:

```
BitSet bs = new BitSet();
bs.set(RetrieveBits.DATA, true);
bs.set(RetrieveBits.RTRANSID, true);
RetrievedDataHolder rdh = new RetrievedDataHolder();
t.retrieve(bs, rdh);
byte[] inData = rdh.value.data;
String transid = rdh.value.transId;
```

Serialization services

JCICS provides support for the CICS serialization services, which let you schedule the use of a resource by a task.

Methods	JCICS class	EXEC CICS Commands
`dequeue()`	SynchronizationResource	DEQ
`enqueue()`, `tryEnqueue()`	SynchronizationResource	ENQ

Storage services

No support is provided for explicit storage management using CICS services (such as **EXEC CICS** GETMAIN). You should find that the standard Java storage management facilities are sufficient to meet the needs for task-private storage.

Sharing of data between tasks must be accomplished using CICS resources.

Names are generally represented as Java strings or byte arrays; you must ensure that these are of the necessary length.

Threads and tasks example

If your CICS Java application is running within an OSGi or Liberty environment, you can run work under a separate thread on a separate CICS task/transaction by using the `CICSExecutorService`.

Submit a Java Runnable or Callable object to the Executor service and the submitted application code will run on a separate thread under a new CICS task. Unlike normal threads created from Java, Executor controlled threads have access to the JCICS API and CICS services. In a CICS OSGi or Liberty environment you can use standard OSGi APIs to find the `CICSExecutorService`, or you can use the JCICS API convenience method `CICSExecutorService.runAsCICS()`, which finds the service and submits the Runnable or Callable object on your behalf.

Note: For non-HTTP requests in Liberty, a CICS task is created only when the first JCICS or JDBC DataSource with type 2 connectivity call is made.

The following example shows an excerpt of a Java class that submits a Runnable piece of application code to the `CICSExecutorService`. The application code simply writes to a CICS TSQ.

```
public class ExecutorTest
{
    public static void main(String[] args)
    {
        // Inline the new Runnable class
        class CICSJob implements CICSTransactionRunnable
        {
            public void run()
            {
                // Create a temporary storage queue
                TSQ test_tsq = new TSQ();
                test_tsq.setType(TSQType.MAIN);

                // Set the TSQ name
                test_tsq.setName("TSQWRITE");

                // Write to the temporary storage queue
                // Use the CICS region local CCSID so it is readable
                String test_string = "Hello from a non CICS Thread - "+ threadId;

                try
                {

test_tsq.writeItem(test_string.getBytes(System.getProperty("com.ibm.cics.jvmserver.local.ccsid")));
                }
                catch (Exception e)
                {
                    e.printStackTrace();
                }
            }

            @Override
            public String getTranid()
            {
                // *** This transaction id should be installed and available ***
                return "IJSA";
            }
        }

        // Create and run the new CICSJob Runnable
        Runnable task = new CICSJob();
        CICSExecutorService.runAsCICS(task);
    }
}
```

Temporary storage queue services

JCICS supports the CICS temporary storage commands; DELETEQ TS, READQ TS, and WRITEQ TS.

Interaction between JCICS methods and EXEC CICS commands

For information about tools that allow Java programs to access existing CICS application data, see Interacting with structured data from Java.

Table 8 on page 61 lists the methods and JCICS classes that map to CICS temporary storage commands.

Table 8. Relationship between methods, JCICS classes and CICS commands

Methods	JCICS class	EXEC CICS Commands
`delete()`	TSQ	DELETEQ TS
`readItem()`, `readNextItem()`	TSQ	READQ TS
`writeItem()`, `rewriteItem()`, `writeItemConditional()`, `rewriteItemConditional()`	TSQ	WRITEQ TS

DELETEQ TS
You can delete a temporary storage queue (TSQ) using the `delete()` method in the TSQ class.

READQ TS
The CICS INTO option is not supported in Java programs. You can read a specific item from a TSQ using the `readItem()` and `readNextItem()` methods in the TSQ class. These methods take an `ItemHolder` object as one of their arguments, which will contain the data read in a byte array. The storage for this byte array is created by CICS and is garbage-collected at the end of the program.

WRITEQ TS
You must provide data to be written to a temporary storage queue in a Java byte array. The `writeItem()` and `rewriteItem()` methods suspend if a NOSPACE condition is detected, and wait until space is available to write the data to the queue. The `writeItemConditional()` and `rewriteItemConditional()` methods do not suspend in the case of a NOSPACE condition, but return the condition immediately to the application as a `NoSpaceException`.

Terminal services

JCICS provides support for these CICS terminal services commands.

Methods	JCICS class	EXEC CICS Commands
`converse()`	TerminalPrincipalFacility	CONVERSE
	Not supported	HANDLE AID
`receive()`	TerminalPrincipalFacility	RECEIVE
`send()`	TerminalPrincipalFacility	SEND
	Not supported	WAIT TERMINAL

If a task has a terminal allocated as a principal facility, CICS automatically creates two Java `PrintWriter` components that can be used as standard output and standard error streams. These

components are mapped to the task terminal. The two streams, which have the names `out` and `err`, are public files in the `Task` object and can be used in the same way as System.out and System.err.

Data to be sent to a terminal must be provided in a Java byte array. Data is read from the terminal into a `DataHolder` object. CICS provides the storage for the returned data which is deallocated when the program ends.

Transforming between data and XML

JCICS supports API commands to transform data to XML and vice versa. These commands provide the equivalent functions to the **EXEC CICS TRANSFORM DATATOXML** and **TRANSFORM XMLTODATA** commands.

Methods	JCICS class	EXEC CICS commands
`SetName`	XmlTransform	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
`dataToXML`	Transform	TRANSFORM DATATOXML
`xmltoData`	Transform	TRANSFORM XMLTODATA
`setChannel`	TransformInput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
`setDataContainer`	TransformInput	TRANSFORM DATATOXML TRANSFORM XMLTODATA
`setElementName`	TransformInput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
`setElementNamespace`	TransformInput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
`setNsContainer`	TransformInput	TRANSFORM XMLTODATA
`setTypeName`	TransformInput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
`setTypeNamespace`	TransformInput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
`setXmlContainer`	TransformInput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
`setXmltransform`	TransformInput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
`getElementName`	TransformOutput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
`getElementNamespace`	TransformOutput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
`getTypeName`	TransformOutput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
`getTypeNamespace`	TransformOutput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA

Transient data queue services

JCICS supports the CICS transient data commands, DELETEQ TD, READQ TD, and WRITEQ TD. All options are supported except the INTO option.

Interaction between JCICS methods and EXEC CICS commands

For information about tools that allow Java programs to access existing CICS application data, see Interacting with structured data from Java.

Table 9 on page 63 lists the methods and JCICS classes that map to CICS transient data commands.

Table 9. Relationship between methods, JCICS classes and CICS commands

Methods	JCICS class	EXEC CICS Commands
`delete()`	TDQ	DELETEQ TD
`readData()`, `readDataConditional()`	TDQ	READQ TD
`writeData()`	TDQ	WRITEQ TD

DELETEQ TD
You can delete a transient data queue (TDQ) using the `delete()` method in the `TDQ` class.

READQ TD
The CICS INTO option is not supported in Java programs. You can read from a TDQ using the `readData()` or the `readDataConditional()` method in the `TDQ` class. These methods take as a parameter an instance of a `DataHolder` object that will contain the data read in a byte array. The storage for this byte array is created by CICS and is garbage-collected at the end of the program.

The `readDataConditional()` method drives the CICS NOSUSPEND logic. If a QBUSY condition is detected, it is returned to the application immediately as a `QueueBusyException`.

The `readData()` method suspends if it attempts to access a record in use by another task and there are no more committed records.

WRITEQ TD
You must provide data to be written to a TDQ in a Java byte array.

Unit of work (UOW) services

JCICS provides support for the CICS SYNCPOINT service.

Table 10. Relationship between JCICS and EXEC CICS commands for UOW services

Methods	JCICS class	EXEC CICS Commands
`commit()`, `rollback()`	Task	**SYNCPOINT**

In a Liberty JVM server, UOW syncpointing can be controlled by using the Java Transaction API (JTA). For more information, see “ Java Transaction API (JTA) ” on page 80.

Web services example

JCICS supports all API commands that are available for working with web services in an application.

Methods	JCICS class	EXEC CICS commands
`invoke()`	WebService	INVOKE WEBSERVICE
`create()`	SoapFault	SOAPFAULT CREATE
`addFaultString()`	SoapFault	SOAPFAULT ADD FAULTSTRING

Methods	JCICS class	EXEC CICS commands
addSubCode()	SoapFault	SOAPFAULT ADD SUBCODESTR
delete()	SoapFault	SOAPFAULT DELETE
create()	WSAEpr	WSAEPR CREATE
delete()	WSAContext	WSACONTEXT DELETE
set*()	WSAContext	WSACONTEXT BUILD
get*()	WSAContext	WSACONTEXT GET

The following example shows how you might use JCICS to create a web service request:

```
Channel requesterChannel = Task.getTask().createChannel("TestRequester");
Container appData = requesterChannel.createContainer("DFHWS-DATA");
byte[] exampleData = "ExampleData".getBytes();
appData.put(exampleData);

WebService requester = new WebService();
requester.setName("MyWebservice");
requester.invoke(requesterChannel, "myOperationName");

byte[] response = appData.get();
```

To handle the application data that is sent and received in a web service request, if you are working with structured data, you can use a tool such as IBM Record Generator for Java to generate classes. See “Interacting with structured data from Java” on page 131. You can also use Java to generate and consume XML directly.

Using JCICS

Use the classes from the JCICS library in the same way as Java classes. Applications declare a reference of the required type and a new instance of a class is created using the new operator.

Name CICS resources using the `setName` method to supply the name of the underlying CICS resource. After creating the resource, manipulate objects using standard Java constructs. Call methods of the declared objects in the usual way. Full details of the methods supported for each class are available in the supplied Javadoc.

CICS attempts to pass control to the method with a signature of `main(CommAreaHolder)` in the class specified by the JVMCLASS attribute of the PROGRAM resource. If this method is not found, CICS tries to invoke method `main(String[])`.

For more information, see Java restrictions and JCICS Javadoc information.

This example shows how to create a TSQ object, invoke the `delete` method on the temporary storage queue object you have just created, and catch the thrown exception if the queue is empty.

```
// Define a package name for the program
package unit_test;

// Import the JCICS package
import com.ibm.cics.server.*;

// Declare a class for a CICS application
public class JCICSTSQ
{
    // The main method is called when the application runs
    public static void main(CommAreaHolder cah)
    {
        try
        {
            // Create and name a Temporary Storage queue object
            TSQ tsq = new TSQ();
            tsq.setName("JCICSTSQ");

            // Delete the queue if it exists
            try
```

```
            {
                tsq.delete();
            }
            catch(InvalidQueueIdException e)
            {
                // Absorb QIDERR
                System.out.println("QIDERR ignored!");
            }

            // Write an item to the queue
            String transaction = Task.getTask().getTransactionName();
            String message = "Transaction name is - " + transaction;
            tsq.writeItem(message.getBytes());
        }
        catch(Throwable t)
        {
            System.out.println("Unexpected Throwable: " + t.toString());
        }

        // Return from the application
        return;
    }
}
```

Java restrictions

When developing Java applications for CICS, there are a number of restrictions a programmer should be aware of.

The following restrictions apply for Java applications used in CICS :

- Do not use the `System.exit()` method. Using this method when the application is running in a JVM server will terminate the JVM server, quiesce CICS, and may lead to data inconsistency. Use a Java security policy to prohibit use of `System.exit()`. For related information see Enabling a Java security manager.
- JCICS API calls: these calls cannot be used in the activator classes of OSGi bundles.

  Note: The Java thread that runs the OSGi bundle activator will not be JCICS-enabled. A developer can start a new JCICS-enabled thread from an activator, by using the `CICSExecutorService.runAsCICS()` API. Any JCICS commands will run under the authority of the user id that issued the install command. Therefore it is prudent for an administrator to understand the resources used in OSGi bundle activators before they install them. There is more information on the `runAsCICS()` API at “Threads and tasks example” on page 60.
- Start and stop methods used in OSGi bundle activators: these methods must return in a reasonable amount of time.
- Do not share JCICS objects between threads. You can only call instance methods on JCICS objects from the thread that created them.
- Do not use finalizers in CICS Java programs. For an explanation of why finalizers are not recommended, see IBM SDK for z/OS, Java Technology Edition, Version 7, Troubleshooting and support section.

Guidance for using OSGi

A number of considerations for developing OSGi applications.

Defining dependencies

When an OSGi bundle uses Java packages from another OSGi bundle, the interface between the two bundles must be explicitly expressed. The bundle that uses the package must add the package to the **`Import-Package`** statement in its `manifest.mf`. The bundle that provides the package must add the package to the **`Export-Package`** statement in its `manifest.mf`. When both OSGi bundles are deployed into the environment, the dependency can be resolved.

All packages that are used by an OSGi bundle, including JRE extensions such as **`javax.*`** must be explicitly imported. This is the case even if the run time would otherwise find these packages through

other means such as bootdelegation. Assume that only the core **java.*** packages are available by default.

There are alternative ways of expressing dependencies - in particular the bundle header **Require-Bundle**. However, **Require-Bundle** is more coarse-grained and ties the consumer to a specific bundle. Using **Require-Bundle** also prevents architectural flexibility and restricts the ability to version packages independently.

JRE class visibility, bootdelegation, and system.packages.extra

In OSGi, loading of core JRE packages/classes (`java.*`) is always delegated to the bootstrap classloader. It is assumed that there is only one JRE in the system, and so explicit dependency statements are not required. For that reason, it is never necessary to add a `java.*` dependency to a bundle manifest. However, for other parts of the JRE, application bundles that require these packages must code an Import-Package statement; for example vendor-specific extensions `javax.*` `com.sun.*` and `com.ibm.*` require an import. This is because they are not delegated to the bootstrap classloader and instead treated as part of the OSGi system.

The OSGi framework provides a system bundle that exposes known extension packages to the system automatically. The application bundle registers its dependency by including an Import statement, just as for all other packages provided by OSGi bundles. The advantage of this approach is that extensions can be replaced with newer implementations by installing an OSGi bundle that contains the new code.

An exception to this process is where a particular package is added to the bootdelegation list by using a special OSGi property. Although convenient (as no Import statement is required to access these packages), it restricts the flexibility of OSGi and is not considered best practice. Occasionally there are vendor-specific extensions that aren't automatically added to the system bundle by the OSGi implementation. For these cases, and assuming the package is genuinely available from the JRE, the property `-Dorg.osgi.framework.system.packages.extra` can be used to add the packages to the system bundle and allow application Imports to resolve.

Bundle activators

Bundle activators are classes within an OSGi bundle that implement the **BundleActivator** interface. To use an activator, an OSGi bundle must declare it using the **Bundle-Activator** header in the bundle manifest. The **BundleActivator** interface has `start` and `stop` methods that can be used to perform initialization or termination work. A common pattern is to look up service dependencies for use within the application. However, it is better to employ a component model, such as Declarative Services to activate components and their service dependencies.

Singleton bundles

A singleton bundle is used to prevent any other version of a bundle being loaded in memory, there can be only one resolved version in the run time at any point. The use of a singleton bundle can be desirable where access to a single system resource is required from a set of applications.

OSGi bundle fragments

Fragments are OSGi bundles that are dynamically attached to host bundles by the OSGi framework. They share the class loader for their host bundle, and do not participate in the lifecycle of the bundle - for that reason they do not support bundle activators. Common use-cases for fragments are as bundle patches. A fragment provided ahead of `.` on the **Bundle-ClassPath** allows classes to be preferentially loaded from the fragment instead of the host.

OSGi service registry

The OSGi service registry enables a bundle to publish objects to a shared registry. A service is advertised under a Java interface and made available to other bundles installed in the OSGi environment.

Microservices (µServices)

Microservices are a software architecture style in which complex applications are composed of small, independent components which communicate with each other using language-agnostic APIs. These services are small, highly decoupled, and focus on doing a small task, facilitating a modular approach to system building. The use of µServices between OSGi components provides flexibility and dynamic update capabilities that cannot be achieved by using bundle wiring alone. For this reason, the use of µServices is encouraged over bundle-wiring.

Bundle and package versioning

A favored approach to package versioning in OSGi is the semantic versioning model. Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes
2. MINOR version when you add functionality that is compatible with an earlier version
3. PATCH version when you make bug fixes that are compatible with an earlier version

Execution environment

Execution environments (EEs) are symbolic representations of JREs, for example:

```
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
```

You need to use the lowest version of EE that gives you all the features you require. When creating a new OSGi bundle, the most recent actively maintained Java execution environment is usually adequate - only if a specialized application requires a lower version would you set it at a lower level. When a particular EE is chosen, it must be left alone unless there is a clear advantage to moving up. Increasing the version of your EE can create more work with no real value, such as exposing your code to new warnings, and deprecations.

Developing Java applications to run in a Liberty JVM server

Configure the Liberty JVM server to run a web container if you want to deploy Java EE applications that use WebSphere Application Server Liberty.

Setting up the development environment

You can use the JCICS API to develop Java applications that can access CICS resources. The API is available in the IBM CICS SDK for Java, Maven Central, or in the USSHOME directory of your CICS installation.

About this task

The JCICS API provides you with the Java interface to access CICS services. This API is the Java equivalent of the **EXEC CICS** API that is provided for other CICS supported languages, such as COBOL.

The table shows where the API is provided and what tools you can use to consume it.

Table 11. JCICS API locations

Java code authoring tool	JCICS API
CICS Explorer	Provided in the preinstalled IBM CICS SDK for Java, which resolves dependencies automatically.
Maven and Gradle (to access artifacts from Maven Central)	You can declare the dependency using any Java IDE that supports Maven or Gradle.

Table 11. JCICS API locations (continued)

Java code authoring tool	JCICS API
Any other tool, to import the API `jars` provided in USSHOME	You need to manually import the dependency.

Note: If you want to develop Java EE applications for CICS, you must install the IBM CICS SDK for Java EE and Liberty in CICS Explorer.

CICS Explorer provides the following tools for you to develop, package, and deploy Java applications that are hosted in the CICS JVM server:

- The IBM CICS SDK for Java provides support for the JCICS API.
- Eclipse and the Eclipse Web Tools Platform provide the tools to develop Java EE applications.
- The IBM CICS SDK for Java EE and Liberty provides the Java EE and Liberty APIs in the form of a Java build path library or OSGi target platform.
- CICS Explorer provides the tools to package, deploy, and manage Java applications within CICS bundles.
- Explorer for z/OS provides the tools to work with files, data sets, and jobs on z/OS, including viewing JVM server log files.

The SDKs can resolve dependencies automatically as long as you add the correct library to your build path or select the correct OSGi target platform.

Consuming dependencies from Maven Central offers more flexibility in the choice of IDE and integrates easily into popular build toolchains such as Maven or Gradle. The Maven Central artifacts contain the JCICS, CICS annotation, CICS annotation processor libraries and a bill of material (BOM) for you to declare dependencies and develop applications for CICS in your IDE of choice. The artifacts can be obtained directly from Maven Central, or from locally hosted and allow-listed repositories using tools such as JFrog Artifactory or Sonatype Nexus.

Procedure

- To use the SDKs in CICS Explorer:
  a) The IBM CICS SDK for Java is preinstalled in CICS Explorer. If you want to develop Java EE applications, install the IBM CICS SDK for Java EE and Liberty into your CICS Explorer as a plug-in. For instructions, see Planning CICS Explorer installation and Installing the CICS SDK for Java EE and Liberty.
  b) Restart your development environment.
  c) Add libraries to your build path or select your target platform, for the SDK to revolve dependencies correctly for your projects. See Step 1 in “Creating a Dynamic Web Project” on page 69, “Creating an OSGi Application Project ” on page 70, and “Creating an Enterprise Application Project” on page 72.
- To use Maven or Gradle:
  a) Ensure your environment fulfills either of the following prerequisites:
     – If you want to use Maven or Gradle with the command line, you must install them on your machine. See Downloading and Installing Maven and Installing Gradle.
     – Your IDE must support Maven or Gradle. Such IDEs include Eclipse, IntelliJ IDEA, and Visual Studio Code.
  b) Create your project using Maven or Gradle and import dependencies based on the API you want to use, as described in “Developing applications using Maven or Gradle” on page 31. You might need to add extra Maven or Gradle configuration, depending on your project type. See Step 1 in “Creating a Dynamic Web Project” on page 69, “Creating an OSGi Application Project ” on page 70, and “Creating an Enterprise Application Project” on page 72.
- To use the JAR files in the USSHOME directory, see “Manually importing Java libraries” on page 36.

Results

Your development environment is ready to develop Java applications for CICS.

What to do next

You can start developing Java applications by referring to the JCICS Javadoc information. If you're using the IBM CICS SDK for Java, you can use the examples that are provided with the IBM CICS SDK for Java to get started. For more information, see Java samples: Servlet examples.

Java EE and Liberty applications

To provide modern interfaces to CICS applications, you can develop a presentation layer that uses web application technology. You can use the IBM CICS SDK for Java in CICS Explorer or the CICS-provided artifacts on Maven Central to create, package, and build the applications. The IBM CICS SDK for Java EE and Liberty, which is optionally installed with CICS Explorer, also provides support to deploy the application to run in CICS.

About this task

Three types of web application projects can be deployed on a Liberty server:

- Dynamic Web Project (WAR)
- OSGi Application Project (EBA)
- Enterprise Application Project (EAR)

A WAR can contain dynamic Java EE resources such as Liberty in CICS, filters, and associated metadata, in addition to static resources such as images and HTML files.

An EBA is a Java archive file that can contain WABs and OSGi bundles. WABs are web-enabled OSGi bundles that contain JSP servlets and files, filters, and associated metadata, in addition to static resources such as images and HTML files.

An EAR is a way of organizing WAR and EJB modules into a single container in the same way as an EBA organizes WABs and OSGi bundles.

Creating a Dynamic Web Project

To develop a web presentation layer for your Java application, you can create a Dynamic Web Project.

Before you begin

Ensure that you have set up the development environment.

A restriction added by Liberty and introduced by an APAR applied to CICS TS V5.1, prevents access to OSGi bundles from servlets that are deployed in a WAR file. The restriction includes access to OSGi bundles installed directly in a CICS bundle. To overcome this restriction, you must deploy your application as a WAB, as part of an EBA (OSGi Application Project). An EBA is a container in which web and OSGi components can interact.

About this task

If you are using the IBM CICS SDK for Java or IBM CICS SDK for Java EE and Liberty preinstalled in CICS Explorer (as shown in the following instructions) or IBM Developer for z/OS (IDz), you can refer to the CICS Explorer and IBM CICS SDK for Java help, which provides full details on how you can complete each of the following steps to develop and package web applications.

If you are using a build toolchain such as Maven or Gradle, you can use CICS-provided artifacts on Maven Central to define Java dependencies.

Procedure

1. Create a web project for your application.
   - CICS Explorer If you're using CICS Explorer, create a Dynamic Web Project and update your build path to add the Liberty libraries.
     a. Right-click the Dynamic Web Project and click **Build Path** > **Configure Build Path**. The properties dialog opens for the project.
     b. In the Java Build Path, click the **Libraries** tab.
     c. Click **Add Library** and select **CICS with Java EE and Liberty**.
     d. Click **Next**, select the CICS version, then click **Finish** to complete adding the library.
     e. Click **OK** to save your changes.
   - Gradle For Gradle users, create a Gradle project. In the `build.gradle` file, specify the following and declare dependencies on CICS-provided artifacts.

     ```
     plugins {
         id 'war'
     }
     ```

   - Maven For Maven users, create a Maven project. In the `pom.xml` file, specify `<packaging>war</packaging>` and declare dependencies on CICS-provided artifacts. If you are unfamiliar with Maven, you can start with the maven-archetype-webapp archetype and modify it.
2. Develop your web application. You can use the JCICS API to access CICS services, JDBC to access DB2 and JMS to access IBM MQ. The IBM CICS SDK for Java EE and Liberty includes examples of web components that use JCICS and JDBC.
3. Optional: If you want to secure the application with CICS security, create a `web.xml` file in the Dynamic Web Project to contain a CICS security constraint. The IBM CICS SDK for Java EE and Liberty includes a template for this file that contains the correct information for CICS. See Authenticating users in a Liberty JVM server for further information.
4. Create one or more CICS bundle projects to package your application. Add definitions and imports for CICS resources. Each CICS bundle contains an ID and version so you can manage changes in a granular way.
5. Optional: Add a URIMAP and TRANSACTION resource to a CICS bundle if you want to map inbound web requests from a URI to run under a specific transaction. If you do not define these resources, all work runs under a supplied transaction, which is called CJSA. These resources are installed dynamically and managed as part of the bundle in CICS.

Results

You set up your development environment, created a web application from a Dynamic Web Project, and packaged it for deployment.

What to do next

When you are ready to deploy your application, export the CICS bundle projects to zFS. The referenced projects are built and included in the transfer to zFS. Alternatively, you can follow the Liberty deployment model by exporting the application as a WAR and deploying it to the dropins directory of a running Liberty JVM server.

Creating an OSGi Application Project

An OSGi Application Project (EBA) groups together a set of bundles. The application can consist of different OSGi bundles types.

Before you begin

Ensure that you have set up the development environment.

About this task

If you are using the IBM CICS SDK for Java or IBM CICS SDK for Java EE and Liberty preinstalled in CICS Explorer (as shown in the following instructions) or IBM Developer for z/OS (IDz), you can refer to the CICS Explorer and IBM CICS SDK for Java help, which provides full details on how you can complete each of the following steps to develop and package OSGi applications.

If you are using a build toolchain such as Maven or Gradle, you can use CICS-provided artifacts on Maven Central to define Java dependencies.

Procedure

1. CICS Explorer If you're using CICS Explorer, set up a target platform for your Java development, using the CICS TS 5.4 **with Java EE and Liberty** template. You might get a warning that the target is a newer version than the current Eclipse installation, but you can ignore this warning message.
2. Create an OSGi Bundle Project for your application.
   - CICS Explorer If you're using CICS Explorer, the target platform effectively makes the packages available, so you must include the appropriate Import statements in the bundle manifest. A web-enabled OSGi Bundle Project is the bundle equivalent of a Dynamic Web Project. You can use a web-enabled OSGi Bundle Project to deploy an application within an OSGi Application Project (an Enterprise Bundle Archive, or EBA file). You can mix web-enabled OSGi Bundle Projects (WAB files) and non-web-enabled OSGi Bundle Projects in your OSGi Application Project. A web-enabled OSGi Bundle Project would typically implement the front end of the application, and interact with the non-web OSGi bundles, which contain the business logic.
   - Gradle For Gradle users, create a Gradle-enabled OSGi project using the BND Gradle Plugins, and then declare dependencies on CICS-provided artifacts.
   - Maven For Maven users, create a Maven project. In the `pom.xml` file, specify `<packaging>bundle</packaging>` and the following dependency:

     ```
     <dependency>
       <groupId>net.wasdev.maven.tools.targets</groupId>
       <artifactId>liberty-target</artifactId>
       <version><your_liberty_version></version>
       <type>pom</type>
       <scope>provided</scope>
     </dependency>
     ```

     Instead of specifying the dependency above, you can start with an OSGi bundle archetype, for example, the osgi-web31-liberty artifact, and modify it. Then declare dependencies on CICS-provided artifacts.
3. Develop your web application. You can use the JCICS API to access CICS services and JDBC to connect to DB2. The IBM CICS SDK for Java includes examples of web components and OSGi bundles that use JCICS and DB2. Create OSGi bundles that use JCICS to separate the business from the presentation logic. You can also use semantic versioning in OSGi bundles to manage updates to the business logic of the application. For each WAB or OSGi bundle that uses Db2 through the JDBC DriverManager interface, include an Import-Package header for `com.ibm.db2.jcc` in the bundle manifest. Omitting this import will result in the error message `java.sql.SQLException: No suitable driver found for jdbc:default:connection`. The import is not required when using the JDBC DataSource interface.
4. Optional: If you want to authenticate users of the web application, create a `web.xml` file in the web project to contain a security constraint. The IBM CICS SDK for Java includes a template for this file that contains the correct information for CICS. See Authenticating users in a Liberty JVM server for further information.
5. Create an OSGi Application Project that references your OSGi bundles.
6. Create a CICS bundle project that references the OSGi Application Project. You can also add definitions and imports for CICS resources. Each CICS bundle contains an ID and version so you can manage changes in a granular way.

7. Optional: Add a URIMAP and TRANSACTION resource to a CICS bundle if you want to map inbound web requests from a URI to run under a specific transaction. If you do not define these resources, all work runs under a supplied transaction, which is called CJSA. These resources are installed dynamically and managed as part of the bundle in CICS.

Results

You set up your development environment, created a OSGi web application, and packaged it for deployment.

What to do next

When you are ready to deploy your application, export the CICS bundle projects to zFS. The referenced projects are built and included in the transfer to zFS. Alternatively, you can follow a development deployment model by exporting the application as an EBA file and deploying it to the dropins directory of a running Liberty JVM server. You should be aware that Security and other qualities of service are not configurable using dropins.

Creating an Enterprise Application Project

To develop components such as an Enterprise Java Bean module (EJB module) or to group web projects, EJBs, or both together, you can use an Enterprise Application Project.

Before you begin

Ensure that you have set up the development environment.

About this task

If you are using the IBM CICS SDK for Java or IBM CICS SDK for Java EE and Liberty preinstalled in CICS Explorer (as shown in the following instructions) or IBM Developer for z/OS (IDz), you can refer to the CICS Explorer and IBM CICS SDK for Java help, which provides full details on how you can complete each of the following steps to develop and package Enterprise Applications.

If you are using a build toolchain such as Maven or Gradle, you can use CICS-provided artifacts on Maven Central to define Java dependencies.

Procedure

1. Create a project for you application.
   - CICS Explorer If you're using CICS Explorer, create an Enterprise Application Project.
   - Gradle For Gradle users, create a Gradle project. In the `build.gradle` file, specify the following and declare dependencies on CICS-provided artifacts.

     ```
     plugins {
         id 'ear'
     }
     ```

   - Maven For Maven users, create a Maven project. In the `pom.xml` file, specify `<packaging>ear</packaging>` and declare dependencies on CICS-provided artifacts.
2. Develop the components of your application. These components are typically EJB modules and Dynamic Web Projects. Add the components to your Enterprise Application Project.

   For more information, see Creating an Enterprise JavaBeans (EJB) project.
3. Create one or more CICS bundle projects to package your Enterprise Application. Add definitions and imports for CICS resources. Every CICS bundle contains an ID and version so you can manage changes in a granular way.
4. Optional: Add a URIMAP and TRANSACTION resource to a CICS bundle if you want to map inbound web requests from a URI to run under a specific transaction. If you do not define these resources,

all work runs under a supplied transaction, which is called CJSA. These resources are installed dynamically and managed as part of the bundle in CICS.

Results

You set up your development environment, created an Enterprise Application Project, and packaged it for deployment.

What to do next

When you are ready to deploy your application, export the CICS bundle projects to zFS. The referenced projects are built and included in the transfer to zFS. Alternatively, you can follow the Liberty deployment model by exporting the application as a EAR and deploying it with an `<application>` element, or placing it in the drop-ins directory of a running Liberty JVM server.

Creating a URI map and transaction

You can install applications resources through traditional methods such as CSD or BAS, or you can add application resources to CICS bundles. CICS bundles provide a convenient and co-located technique to group application code and CICS resources together. This is useful if, for example, you deploy a Java EE application in a CICS bundle. You might want to provide a URI map that maps the inbound web requests to run under a specific application transaction.

Before you begin

To create the application resources, you must have a CICS bundle project in your Project Explorer. For more information, see Creating a CICS bundle project in the CICS Explorer product documentation. You use this CICS bundle project to package the application for deployment.

About this task

By default all Java EE application requests use a transaction that is called CJSA that is supplied by CICS. However, you can map the application URI from an inbound request to a different transaction. You might find this feature useful if you want to securely control access to the application because a security administrator can configure CICS to control which transactions are accessed by users.

Procedure

1. Create a definition for the application transaction:
   a) Switch to the Eclipse Resource perspective. Right-click the CICS bundle project and click **New** > **Transaction Definition**.

      The New Transaction Definition wizard opens.
   b) Enter a 4-character name for the transaction.

      Do not start the transaction name with C because this letter is reserved by CICS.
   c) Enter the program name DFHSJTHP.

      You must use this CICS program because it handles the security checking of inbound Java EE requests to the Liberty server.
   d) Click **Finish** to create the definition in the CICS bundle project.

   Do not set attributes to create a remote transaction because the application transaction must always run in the CICS region where the Java EE application is running.
2. Create a definition for the URI map:
   a) Right-click the CICS bundle project and click **New** > **URI Map Definition**.
   b) Enter an 8-character name for the URI map.

      Do not start URI maps names with DFH because this prefix is reserved by CICS.
   c) Enter the host name.

You can either use a * to match any host name, or specify the host name of the machine where your application is going to run.

d) Enter the path for the application URI.

CICS matches the URI in the inbound request to the value in the URI map and runs the application transaction.

e) In the Usage section, select **JVM server** and optionally enter the port number.

f) Click **Finish** to create the URI map.

3. Edit the URI map definition:

a) Edit the Scheme field to enter the scheme for the URI map. HTTP is the default, but you can set HTTPS if you want to use SSL security to encrypt the request.

You can use basic authentication, where a user ID and password are supplied in the HTTP header, on both HTTP and HTTPS requests.

b) Edit the Transaction field to enter the name of the application transaction.

c) Optional: Edit the user ID field to enter a user ID to run the application request.

This value is ignored if basic authentication is enabled. If you do not supply a value and the HTTP request does not include a user ID and password, CICS runs the request under the default user ID of the CICS region.

Results

You created a URI map and a transaction in the CICS bundle project. When the bundle is deployed and installed, these resources are created dynamically in the CICS region.

What to do next

You can create extra resources if you want to run different application operations under different transactions, or if you want to support both HTTP and HTTPS schemes. If your application is ready to deploy, see Deploying a CICS bundle in the CICS Explorer product documentation.

Migrating Java EE applications to run in Liberty JVM server

If you have a Java EE application running in a Liberty instance that accesses CICS over a network, you can run the application in a Liberty JVM server to optimize performance.

About this task

CICS supports a subset of the features that are available in Liberty. For a list of supported features in CICS integrated-mode Liberty, see “Liberty features” on page 117.

If your application uses security, you can continue to use Liberty security features however without further action it is possible that the CICS task will run under transaction CJSA, URIMAP matching in CICS will not be available and any resource access will be performed under the CICS default user ID. To better integrate your security solution with CICS, allowing your CICS tasks to run under the same user ID as determined by Liberty, see Authenticating users in a Liberty JVM server.

Procedure

1. Update the application to use the JCICS API to access CICS services directly, ensuring that the correct JCICS encoding is used when the application passes data to and from CICS.

   For more information about encoding, see “Data encoding” on page 41. This step only applies if you are using CICS integrated-mode Liberty or CICS standard-mode Liberty with the `runAsCICS()` API.
2. If you want to use CICS security for basic authentication, update the security constraint in the `web.xml` file of the Dynamic Web Project to use a CICS role for authentication. This step only applies if you are using CICS integrated-mode Liberty or CICS standard-mode Liberty and submitting work to the `CICSExecutorService` using the `runAsCICS()` method.

```
<auth-constraint>
    <description>All authenticated users of my application</description>
    <role-name>cicsAllAuthenticated</role-name>
</auth-constraint>
```

3. Package the application as a WAR (Dynamic Web Project), an EBA (OSGi Application Project) file or an EAR (Enterprise Application Archive) file, in a CICS bundle.

   CICS bundles are a unit of deployment for an application. All CICS resources in the bundle are dynamically installed and managed together. Create CICS bundle projects for application components that you want to manage together.
4. Deploy the CICS bundle projects to zFS and install the CICS bundles in the Liberty JVM server.

Results

The application is running in a JVM server.

Linking to a Java EE application from a CICS program

You can link to a Java EE application running in a Liberty JVM server either as the initial program of a CICS transaction, or by using the LINK, START, or START CHANNEL commands from any CICS program.

To be linked to by a CICS program, the Java EE application is required to be a plain Java object (POJO) packaged in a Web ARchive (WAR) or Enterprise Application Archive (EAR). You cannot link to an EJB, a CDI bean, or an OSGi application. Dependency injection in the POJO is not supported, including injecting EJBs using `@EJB`. Instead, you can use a JNDI lookup to obtain a reference to a resource such as an EJB. This information applies to CICS integrated-mode Liberty only.

There are three main reasons for linking to a Java EE application from a CICS program:

- Java code is part of an existing web application and you want to link it to a CICS application. You only need to maintain a single piece of logic and your code can access CICS resources by using JCICS APIs.
- You want to write a new piece of function in Java as part of your CICS application. For example, you might want to use third party libraries or APIs that already exist in Java.
- You want to re-implement existing COBOL applications in Java. For example, you might want to reduce the cost of maintenance and make the most of your Java skills, or you might want your applications to be eligible to run on specialty engines rather than general processors.

When you link to a Java EE application from a CICS program, CICS sends a message to a JCA resource adapter running inside Liberty. The JCA resource adapter links to the target Java EE application on the same CICS task as the calling program. The Java EE application runs under the same unit-of-work (UOW) as the calling program, so any updates made to recoverable CICS resources are committed or backed out when the transaction ends. However, when the Java EE application is invoked, there is no JTA transaction context. If the application starts a JTA transaction, a syncpoint is performed to commit the CICS UOW, and create a new one. This also occurs if a JTA transaction is started by the container on behalf of the application, for example if the application calls an EJB with the REQUIRED transaction attribute.

As best-practice, the code that is linked by the CICS program should be part of your application's business logic (rather than presentation logic). For example, it would not make sense to link a servlet from a CICS program because there is no HTTP request involved.

Configuring a Liberty JVM server to link Java EE applications to CICS programs

To configure your Liberty JVM server to support linking Java EE applications, add the `cicsts:link-1.0` feature to `server.xml`. Ensure that you add the feature before deploying the Java EE applications.

Security

When you link a Java EE application to a CICS program, the user ID of the CICS task is passed into the Java EE application. Liberty does not authenticate the user, but trusts the identity tha t is passed in by CICS. However, Liberty does check that the user ID is present in the configured user registry. Where

possible, use the SAF registry in Liberty because it checks the user ID passed in from CICS. If you are using another type of user registry other than the SAF registry, and the same user ID is present in the registry, then that user ID will be passed to the Java EE application. If the user ID is not present in the user registry, the Java EE application is linked with the unauthenticated user ID.

To configure security when you are linking a Java EE application, include the `cicsts:security-1.0` feature in your `server.xml`. If you do not include this feature, the Java EE application will be linked without being authenticated. As a result, any authorization checks in your Java EE applications might not apply. However, access to any CICS resources using the JCICS API will still run under the user ID of the CICS task.

The following web security mechanisms do not apply when you are linking a Java EE application to a CICS program:

- Java EE web security mechanisms. For example: `<auth-constraint>` in `web.xml` or `@HttpConstraint` on a servlet.
- Trust Association Interceptors (TAIs).
- The JVM server profile property `com.ibm.cics.jvmserver.unclassified.userid`.
- URIMAPs.

The Java EE application can perform additional authorization checks by calling an EJB and applying EJB security. For example, authorization can be applied to session bean methods using the **`@RolesAllowed`** annotation. For more information about EJB security, see The Java EE Tutorial.

For more information about security in Liberty see Configuring security for a Liberty JVM server.

Preparing a Java EE application to be called by a CICS program

With CICS TS APAR PH14856, you can use annotations to enable a Java method to be invoked by a CICS application - CICS creates the PROGRAM resource for you. The Java EE application runs in a Liberty JVM server, and can be deployed within a WAR or EAR.

Before you begin

First, identify which Java class and method you want to call. Then, while adhering to site standards and CICS naming rules, determine a suitable CICS program name.

Make sure that the Liberty JVM server is configured to enable linking to Java EE applications. For more information, see Linking to a Java EE application from a CICS program.

Note: To avoid concurrency issues, JCICS fields should be defined within the link-target method, or a subsequent prototype-scoped Bean, and not on the linked-to component class.

Procedure

1. Add the `@CICSProgram` annotation class to the classpath of your Web Project.
   - CICS Explorer If you are using the preinstalled **IBM CICS SDK for Java** in CICS Explorer, the SDK includes the Liberty JVM server libraries, which provide the `@CICSProgram` annotation. Add the **CICS with Java EE and Liberty** library to your Java project by right-clicking the project and configuring the Build Path. This will also provide the CICS annotation processor dependency. For detailed instructions on configuring the Build Path, see Step 1 in “Creating a Dynamic Web Project” on page 69.
   - GradleMaven If you're using your own build toolchain, you need to declare dependence on the `com.ibm.cics.server.invocation.annotations` artifact that's available on Maven Central or use the `com.ibm.cics.server.invocation.annotations.jar` JAR file. For more information, see “Developing applications using Maven or Gradle” on page 31 and “Manually importing Java libraries” on page 36.
2. Create a class to contain the methods that CICS calls.

   Creating a class is good practice because it keeps the CICS specific code separate from the rest of your application.

3. Create a method for each CICS PROGRAM resource to be created.
4. Annotate each method with the `@CICSProgram` annotation, giving it a parameter of the PROGRAM name, such as `@CICSProgram("PROGNAME")`.

   CICS PROGRAM names:

   - Must be 1 - 8 characters;
   - Must match the pattern `A-Z a-z 0-9 $ @ #`.

   Example of a simple class with a single method, annotated with the `@CICSProgram` annotation:

```
public class CustomerLinkTarget
{
    @CICSProgram("CUSTGET")
    public void getCustomer()
    {
        // do work here
    }
}
```

5. Enable annotation processing for the Web Project.
   - CICS Explorer If you are using CICS Explorer, either:
     - Hover over a `@CICSProgram` annotation with a warning underline and use the quick-fix to enable annotation processing, or:
     - Right-click the Web Project and select **Properties**. Search for the **Annotation Processing** page. Check both **Enable project-specific settings** and **Enable annotation processing**.
   - GradleMaven If you're using a build toolchain such as Gradle or Maven, configure the Java compiler to use `com.ibm.cics.server.invocation` as an annotation processor, as described in Developing applications using other tools.
6. Validate the annotation is correctly specified.
   - CICS Explorer If you are using CICS Explorer, validation happens automatically to ensure that your annotation is correctly positioned and that the method that it annotates and the containing class fulfills the following requirements.
   - Maven If you're using Maven in Eclipse, you can use the m2e-apt plugin to get the annotation processing configured in Eclipse based on the dependencies specified in your `pom.xml` file.

   The annotation:

   - Must be on a method;
   - Must have a value attribute of a PROGRAM name.

   The method:

   - Must be concrete (not abstract);
   - Must be public;
   - Must have no arguments.

   The class:

   - Must have a constructor with no arguments (implicit or explicit), unless all annotated methods are static;
   - Must be top level (not nested or anonymous);
   - Must not have more than one method that is annotated with the same PROGRAM name.
7. Write the content of the annotated method. The content is likely to involve the following stages:
   a) Obtain containers from the channel;
   b) Obtain input data from containers in a channel;
   c) Use data mapping code to convert the input data to Java objects;
   d) Call the application business logic;

e) Use data mapping code to convert the resulting Java objects to output data;

f) Place the output data in containers in a channel.

Example of a class with a single method, annotated with the `@CICSProgram` annotation, and code to take input data from a container and put output data to a container:

```
public class CustomerLinkTarget
{
    @CICSProgram("CUSTGET")
    public void getCustomer()
    {
        Channel currentChannel = Task.getTask().getCurrentChannel();
        Container dataContainer = currentChannel.getContainer("DATA");

        // do work here

        Container resultContainer = currentChannel.createContainer("RESULT");
        byte[] results = null; // change this to be the result of the work
        resultContainer.put(results);
    }
}
```

8. Build the application.
   - CICS Explorer If you are in CICS Explorer , you can right-click the Web Project and select **Export** > **WAR file**, or right-click a containing CICS Bundle Project and select **Export Bundle to z/OS UNIX file system**.
   - If you are using the CICS build toolkit, the annotation processor is invoked automatically.
   - GradleMaven If you are building the Java code by using other tools, ensure that the dependency on the CICS annotation and the annotation processor configuration are correctly specified by using the artifacts on Maven Central. If you've done that in Steps “1” on page 76 and “5” on page 77, they are resolved automatically during build. Otherwise, you must ensure the `com.ibm.cics.server.invocation.annotations.jar` JAR file (which defines the `@CICSProgram` annotation) is on the classpath of the Java compiler. Also, ensure that the `com.ibm.cics.server.invocation.jar` JAR file (containing the annotation processor) is on the classpath of the Java compiler, or is otherwise specified in the **`-processorpath`** option. You can find both JAR files in the *usshome* `/lib` directory on z/OS UNIX , where *usshome* is the value of the **USSHOME** system initialization parameter.
   - If the class is packaged in a library JAR inside the `WEB-INF/lib` directory of a WAR file, export the generated metadata when you are building the JAR. CICS ExplorerIn CICS Explorer , you can do this by adding the library project to the deployment assembly of the Dynamic Web Project. From the properties dialog for the Dynamic Web Project, choose the Deployment Assembly page, click the **Add** button, and select the library project. CICS does not support `@CICSProgram` annotations on classes that are packaged in a utility JAR within an EAR file.

   Note: The CICS annotation processor generates additional classes and XML files within the `com.ibm.cics.server` package to represent your annotated resources, alongside your annotated code.
9. Deploy the application.

Results

If the application is installed by a CICS bundle, PROGRAM resources are created as the CICS bundle becomes ENABLED. If the application is installed directly from `server.xml` or from a file by using an `<application>` element; PROGRAM resources are created as the application is installed.

You can now link to the Java program from another CICS program by using:

```
EXEC CICS LINK PROGRAM("CUSTGET") CHANNEL()
```

Program Lifecycle

When a Java EE application is installed into Liberty the `cicsts:link-1.0` feature searches for methods that are annotated with `@CICSProgram` . For each one, it dynamically installs a PROGRAM resource.

If a Java EE application is installed into Liberty by using a CICS bundle, the PROGRAM resources are created when the bundle is enabled. Otherwise, PROGRAM resources are created when Liberty installs the application.

When an application is removed, CICS deletes any dynamically installed PROGRAM resources that are associated with that application. If the application was installed by using a CICS bundle, CICS deletes the programs when the bundle is disabled. If an application is removed while tasks that invoke the application are still in progress, errors might occur. Therefore, disable any PROGRAM resources that are associated with a Java EE application and allow work to drain before you remove the application. Otherwise, programs are deleted when Liberty uninstalls the application.

In most cases, you do not need to create your own PROGRAM definition. You might want to create your own PROGRAM definition if you do not want CICS to create one for you automatically, or if you want to specify particular attributes. To create a private program as part of a CICS application that is deployed on a platform, you must define it in a CICS bundle that is installed as part of that application. You can create a program definition in the CSD, BAS or in a CICS bundle, and install it yourself. When CICS finds a method that is annotated with `@CICSProgram` and a matching PROGRAM resource is already installed, CICS does not replace it.

When you are creating your program definition, you must specify the same classname as the class that contains the method that is annotated with `@CICSProgram` . You can optionally specify the method name as well. CICS validates this information when the program is invoked. The JVMCLASS attribute should contain the classname and optionally the method name in the format `wlp:classname#methodname` , for example:

```
wlp:com.example.CustomerLinkTarget#getCustomer
```

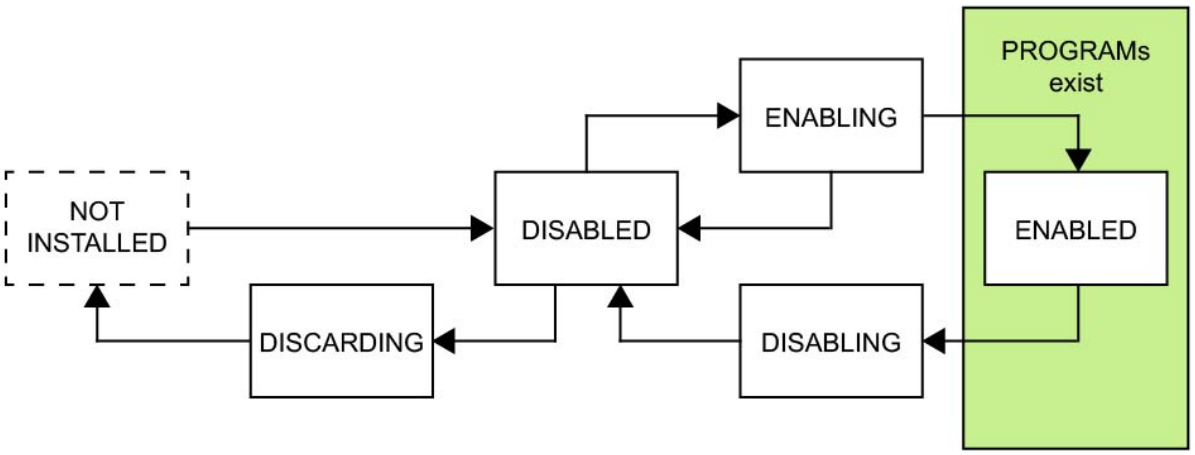


Figure 2. The lifecycle of a CICS bundle, showing when PROGRAM resources for `@CICSProgram` annotations exist

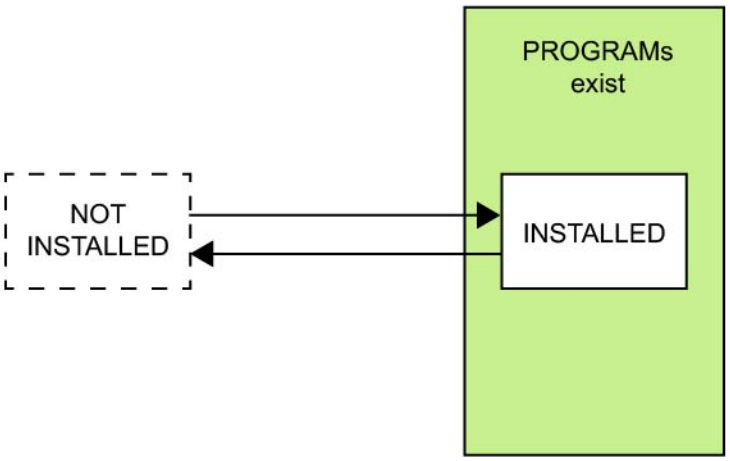


Figure 3. The lifecycle of a stand-alone web application, showing when PROGRAM resources for @CICSProgram annotations exist

Java Transaction API (JTA)

The Java Transaction API (JTA) can be used to coordinate transactional updates to multiple resource managers.

You can use the Java Transaction API (JTA) to coordinate transactional updates to CICS resources and other third party resource managers, such as a type 4 database driver connection within a Liberty JVM server. In this scenario, the Liberty transaction manager is the transaction coordinator and the CICS unit of work is subordinate, as though the transaction had originated outside of the CICS system.

Note: If you have the JVM profile option `com.ibm.cics.jvmserver.wlp.jta.integration=false` and use autoconfigure, or are manually configuring `server.xml` and include the `<cicsts_jta Integration="false"/>` element, then the CICS unit-of-work will not participate in the JTA transaction and is committed or rolled back separately.

A type 2 driver connection to a local DB2 database using a CICS data source is accessed using the CICS DB2 attachment. It is not necessary to use JTA to coordinate with updates to other CICS resources.

In JTA you create a UserTransaction object to encapsulate and coordinate updates to multiple resource managers. The following code fragment shows how to create and use a User Transaction:

```
InitialContext ctx = new InitialContext();
UserTransaction tran = (UserTransaction)ctx.lookup("java:comp/UserTransaction");

DataSource ds = (DataSource)ctx.lookup("jdbc/SomeDB");
Connection con = ds.getConnection();

// Start the User Transaction
tran.begin();

// Perform updates to CICS resources via JCICS API and
// to database resources via JDBC/SQLJ APIs

if (allOk) {
    // Commit updates on both systems
    tran.commit();
} else {
    // Backout updates on both systems
    tran.rollback();
}
```

If you are using an OSGi application, ensure that you include the following entry in the MANIFEST.MF:

```
Import-Package: javax.transaction;version="[1.1,2)"
```

Your development environment might not highlight this dependency by default. It is advisable to explicitly check and to ensure the minimum version of 1.1 is specified. If you allow the runtime environment to resolve the dependency itself, it might resolve to the lower version of the package from the underlying JRE, and conflict with the Liberty runtime.

Unlike a CICS unit of work, a UserTransaction must be explicitly started using the `begin()` method. Invoking `begin()` causes CICS to commit any updates that may have been made prior to starting the UserTransaction. The UserTransaction is terminated by invoking either of the `commit()` or `rollback()` methods, or by the web container when the web application terminates. While the UserTransaction is active, the program cannot invoke the JCICS `Task commit()` or `rollback()` methods.

The JCICS methods `Task.commit()` and `Task.rollback()` will not be valid within a JTA transaction context. If either is attempted, an InvalidRequestException will be thrown.

The Liberty default is to wait until the first UserTransaction is created before attempting to recover any indoubt JTA transactions. However, CICS will initiate transaction recovery as soon as the Liberty JVM server initialization is complete. If the JVM server is installed as disabled, recovery will run when it is set to enabled.

If you are using EJBs, see Using JTA transactions in EJBs .

Java Persistence API (JPA)

The JPA can be used to create object oriented versions of relational database entities for developers to make use of in their applications.

You can use the JPA to provide annotations and XML extensions which you can use to describe tables in their database and their contents, including data types, keys and relationships between tables. Developers can use the API to perform database operations instead of using SQL.

CICS supports jpa-2.0 and jpa-2.1. For information on how these versions differ, see Java Persistence API (JPA) feature overview.

- `Entity objects` are simple Java classes, and can be concrete or abstract. Each represents a row in a database table, and properties and fields are used to maintain states. Each field is mapped to a column in the table, and key information about that particular field is added in; for example, you can specify primary keys, or fields that can't be null.

```
@Entity
@Table(name = "JPA")
public class Employee implements Serializable
{
    @Id
    @Column(name = "EMPNO")
    private Long EMPNO;

    @Column(name = "NAME", length = 8)
    private String NAME;

    private static final long serialVersionUID = 1L;

    public Employee()
    {
        super();
    }

    public Long getEMPNO()
    {
        return this.EMPNO;
    }

    public void setEMPNO(Long EMPNO)
    {
        this.EMPNO = EMPNO;
    }

    public String getNAME()
    {
        return this.NAME;
    }
```

```
    public void setNAME(String NAME)
    {
        this.NAME = NAME;
    }
}
```

- The `EntityManagerFactory` is used to generate an `EntityManager` for the persistence unit. `EntityManager` maintains the active collection of `entity objects` being used by an application. You can use the `EntityManager` class to initialize the classes and create a transaction for managing data integrity. Next, you interact with the data using the `Entity` class `get` and `set` methods, before using the `Entity` transaction to commit the data.

  The following example contains sample code to insert a record:

```
@WebServlet("/Create")
public class Create extends HttpServlet
{
    private static final long serialVersionUID = 1L;

    @PersistenceUnit(unitName = "com.ibm.cics.test.wlp.jpa.annotation.cics.datasource")
    EntityManagerFactory emf;

    InitialContext ctx;

    /**
    * @throws NamingException
    * @see HttpServlet#HttpServlet()
    */
    public Create() throws NamingException
    {
        super();
        ctx = new InitialContext();
    }

    /**
    * @see HttpServlet#doGet(HttpServletRequest request,
    HttpServletResponse response)
    */
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
    {
        // Get the servlet parms
        String id = request.getParameter("id");
        String name = request.getParameter("name");

        // Create a new employee object
        Employee newEmp = new Employee();
        newEmp.setEMPNO(Long.valueOf(id));
        newEmp.setNAME(name);

        // Get the entity manager factory
        EntityManager em = emf.createEntityManager();

        // Get a user transaction
        UserTransaction utx;

        try
        {
            // Start a user transaction and join the entity manager to it
            utx = (UserTransaction) ctx.lookup("java:comp/UserTransaction");
            utx.begin();
            em.joinTransaction();

            // Persist the new employee
            em.persist(newEmp);

            // End the transaction
            utx.commit();
        }
        catch(Exception e)
        {
            throw new ServletException(e);
        }

        response.getOutputStream().println("CREATE operation completed");
    }
}
```

- `@PersistenceUnit` expresses a dependency on an `EntityManagerFactory` and its associated persistence unit. The name of the persistence unit as defined in the `persistence.xml` file. To connect the entities and tables to a database we then create a `persistence.xml` file in our bundle. The `persistence.xml` file describes the database that these entities connect to. The file includes important information such as the name of the provider, the entities themselves, the database connection URL and drivers.

  The following example contains a sample `persistence.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

    <persistence-unit name="com.ibm.cics.test.wlp.jpa.annotation.cics.datasource">
        <jta-data-source>jdbc/jpaDataSource</jta-data-source>

        <class>com.ibm.cics.test.wlp.jpa.annotation.cics.datasource.entities.Employee</class>

        <properties>
            <property name="openjpa.LockTimeout" value="30000" />
            <property name="openjpa.Log" value="none" />
            <property name="openjpa.jdbc.UpdateManager" value="operation-order" />
        </properties>
    </persistence-unit>
</persistence>
```

Enterprise JavaBeans (EJB)

Enterprise JavaBeans (EJB) is a Java API, and a subset of the Java EE specification. EJBs contain the business logic of an application, and are fully supported by CICS Liberty, including the Lite subset.

The Liberty features that provide the support for EJBs are:

Table 12. Liberty features that provide support

Feature	Support	Java EE version
ejbLite-3.1	This feature enables the Lite subset of the EJB technology as defined in the EJB specification. This subset includes support for local session beans that are written to the EJB 3.x APIs.	Java EE 6
mdb-3.1	This feature enables the message-driven bean subset of the EJB technology, which is similar to the support that the ejbLite feature enables for session beans.	Java EE 6
ejbLite-3.2	This feature enables the Lite subset of the EJB technology as defined in the EJB specification. This subset includes support for local session beans that are written to the EJB 3.x APIs, non-persistent EJB timers, and asynchronous local interface methods.	Java EE 7

Table 12. Liberty features that provide support (continued)		
Feature	**Support**	**Java EE version**
mdb-3.2	This feature enables the message-driven bean subset of the EJB technology, which is similar to the support that the ejbLite feature enables for session beans.	Java EE 7
ejbHome-3.2	Enables support of the EJB 2.x APIs, specifically, support for the `javax.ejb.EJBLocalHome` interface. The `javax.ejb.EJBHome` interface is also supported when combined with the ejbRemote feature.	Java EE 7
ejbRemote-3.2	Enables support for remote EJB interfaces	Java EE 7
ejbPersistentTimers-3.2	Enables support for persistent EJB timers.	Java EE 7
ejb-3.2	Enables full EJB 3.2 support. Covers all EJB 3.2 technology, including remote EJB technology.	Java EE 7

Procedure

Enable the feature in the `server.xml` file. For example:

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

For more information, see:

- Developing EJB 3.x applications for information about developing EJB applications by using WebSphere Developer Tools.
- Developing Enterprise bean (EJB) persistent timer applications for information about developing EJB persistent timer applications.
- Using enterprise JavaBeans applications that call local EJB components in another application for information on using enterprise JavaBeans applications that call local EJB components in another application.

Creating an Enterprise JavaBeans (EJB) project

To develop EJBs for your Java application, you can create an EJB project.

Before you begin

Ensure that the web development tools are installed in your Eclipse IDE. For more information, see “Setting up the development environment” on page 67.

About this task

The CICS Explorer and IBM CICS SDK for Java help provides full details on how you can complete each of the following steps to develop and package EJB applications.

Procedure

1. Create an EJB project for your application.
2. Develop your EJB application. You can use the JCICS API to access CICS services, JDBC to access DB2 and JMS to access IBM MQ.
3. Optional: To secure the application, you can use security annotations, or you can specify security constraints in an `ejb-jar.xml` file. For more information, see Enterprise application security.
4. Add your EJB project to an Enterprise Application Project (EAR).

Results

Your development environment is set up, you created an EJB project, and packaged it for deployment.

Using JTA transactions in EJBs

How to use JTA transactions in Enterprise JavaBeans (EJBs) on Liberty.

About this task

EJBs are Java objects that are managed by the Liberty JVM server, allowing a modular architecture of Java applications. The Liberty JVM server supports EJB Lite 3.1 and EJB Lite 3.2, and EJB 3.2. EJBs are deployed to a Liberty server using an Enterprise Application Archive (EAR) file created from an Enterprise Application Project. Enterprise Application Projects can contain both EJB and Web projects.

EJB Lite is enabled by adding the relevant feature `ejbLite-3.1` or `ejbLite-3.2` to the `server.xml` configuration file. EJB is enabled by adding `ejb-3.2` to the `server.xml` configuration file. EJBs are deployed into a container. This container works in the background ensuring that aspects like session management, transactions and security are adhered to.

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 `ejb-jar.xml`. Bean managed transactions are controlled directly using the Java Transaction API (JTA) . In both cases, the CICS unit of work (UOW) remains subordinate to the outcome of the JTA transaction assuming that you have not disabled CICS JTA integration using the `<cicsts_jta Integration="false"/>` `server.xml` element.

There are six different transaction attributes that can be specified for container managed transactions:

- Mandatory
- Required
- RequiresNew
- Supports
- NotSupported
- Never

A JTA transaction is a distributed UOW as defined in the JEE specification. Setting of a method's transaction attribute determines whether or not the CICS task, under which the method executes, runs under its own unit of work or is part of a wider, distributed JTA transaction.

Note: Although it is respected by the Liberty JTA transaction system, the transaction attribute `NotSupported` does not integrate with and is not supported by the CICS UOW. This applies to EJBs in general.

The following table describes the resulting transactional context of an invoked EJB method, depending on the transaction attribute and whether or not the calling application already has a JTA transactional context.

Important: Liberty does not support outbound or inbound transaction propagation. For more information, see Using enterprise JavaBeans with remote interfaces on Liberty.

Table 13. EJB transaction support

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	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

Important: Calling a method marked as `NotSupported` will suspend the JTA transaction but not suspend the CICS UOW. Any modification of CICS resources during this method call will still be recoverable.

Note: If JTA integration is enabled, the transaction attribute `RequiresNew` is supported by a CICS Liberty JVM server, with the restriction that the CICS UOW cannot be nested. Attempting to call a method marked as `RequiresNew` when already in a JTA transaction will cause an exception to be thrown.

If you call an EJB from a servlet or a POJO and do not explicitly configure the EJB transactional attribute, then by default, container-managed transaction management applies, as does a default transaction attribute of `Required`. This means each call to the EJB starts a new JTA transaction with a subordinate CICS UOW and commits the JTA transaction after each call. If you do not require the use of JTA with EJBs consider using the transaction attribute `Never`.

For information about additional Enterprise JavaBeans (EJB) feature restrictions, see Liberty: Runtime environment known issues and restrictions.

Enterprise Java Bean (EJB) methods with remote interfaces

EJB methods with remote interfaces can be remotely accessed or hosted by CICS Liberty by using RMI-IIOP technologies. You can enable remote EJB support with the `ejbRemote-3.2` feature.

When you use remote EJB interfaces, there are considerations that you must be aware of. For more information, see Using enterprise JavaBeans with remote interfaces on Liberty.

Accessing EJB methods with remote interfaces

1. To configure CICS Liberty to run an application which accesses EJB methods with remote interfaces, you must enable the `ejbRemote-3.2` feature by adding the feature into the `server.xml` file, as follows:

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

2. Configure your application binding files, for example `ibm-*-bnd.xml`, for remote EJB references that are defined either in the deployment descriptor `<ejb-ref>`, or with source code annotations, for example `@EJB`. A binding is not required for EJB references that provide a lookup name, either on the annotation or in the deployment descriptor. In the binding file, the EJB reference can be bound by using one of the `java:` names for an EJB or with one of the `corbaname:names`:

   ```
   @EJB(name="TestBean")
       TestRemoteInterface testBean;
   ```

The binding is defined:

```
<ejb-ref name="TestBean" binding-name=
"corbaname:rir:#ejb/global/TestApp/TestModule/TestBean!test.TestRemoteInterface"/>
```

3. Configure your application client to include stub classes.

Hosting EJB methods with remote interfaces

1. To host EJBs in CICS Liberty so they can be called by other JVMs, you must enable the `ejbRemote-3.2` feature by adding the feature to the `server.xml` file, as follows:

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

2. Configure the IIOP server to customize ports and security settings. For more information, see Configuring IIOP-RMI Transport for Remote EJBs.
3. Create an EJB application. For more information, see Creating an Enterprise JavaBeans (EJB) project.
4. Generate stub classes. In Eclipse, right-click the EJB project and select **Java EE Tools** > **Create EJB Client JAR**.
5. Deploy the EJB application to CICS Liberty as part of an EAR. For more information, see Creating an Enterprise Application project.
6. Check the Liberty `messages.log` file to ensure that the EJB is enabled and bound to a namespace. You should see this message:

```
CNTR0167I: The server is binding the ejb.remote.ejb.view.MyBeanRemote
interface of the MyBean enterprise bean in the ejb.remote.ejb.jar module of the
ejb.remote application. The binding location is:
java:global/ejb.remote/ejb.remote.ejb/MyBean!remote.ejb.view.MyBeanRemote
```

Configuring IIOP-RMI transport for remote EJBs

Internet Inter-ORB Protocol Remote Method Invocation (IIOP-RMI) transport is used by CICS Liberty to communicate with EJB methods that have remote interfaces. This communication can be secured by using Common Secure Interoperability Protocol Version 2 (CSIv2).

IIOP-RMI is used by CICS Liberty as the technology for calling EJB methods with remote interfaces. Using the `ejbRemote-3.2` feature supports both inbound and outbound IIOP-RMI calls.

Inbound calls allow CICS Liberty to listen as an object request broker (ORB) on a TCP/IP port for IIOP-RMI requests and call the target EJB method. See “Configuring Inbound IIOP Communication” on page 87 for details.

Outbound calls are where CICS Liberty makes a request to an ORB to start an EJB method. Outbound calls can be made to the same JVM server the call was made for, or any other Java virtual machine (JVM) capable of acting as an ORB. See “Configuring Outbound IIOP Communication” on page 88 for details.

This communication can be secured by using CSIv2, a technology that satisfies the CORBA (Common Object Request Broker Architecture) for authentication, delegation, and privileges. CSIv2 also supports the use of transport layer security (TLS). See Configuring CSIv2 to secure IIOP Communication for details.

For more information, see Common Secure Interoperability version 2 (CSIv2).

Configuring Inbound IIOP Communication

Enable the `ejbRemote-3.2` feature by adding it to the `server.xml` file.

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

Optionally, you can configure an IIOP endpoint in the `server.xml` file.

```
<iiopEndpoint id="defaultIiopEndpoint" host="host.example.com" iiopPort="2809" />
```

Important: By default the IIOP endpoint listens on `localhost:2809`. The default ORB references the IIOP endpoint `defaultIiopEndpoint`. See Configuring CSIv2 to secure IIOP Communication for more information on configuring ORBs for inbound security.

Configuring Outbound IIOP Communication

Enable the `ejbRemote-3.2` feature by adding it to the `server.xml` file.

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

Optionally you can configure an ORB with the name service of the remote server.

```
<orb id="defaultOrb" nameService="corbaname::host.example.com:2809" />
```

Important: By default the ORB references the local IIOP endpoint `defaultIiopEndpoint`. See Configuring CSIv2 to secure IIOP Communication for more information on configuring ORBs for outbound security.

Configuring CSIv2 to secure IIOP communication

The following information covers some of the general cases for configuring both inbound and outbound CSIv2 security for IIOP communication.

Inbound calls allow CICS Liberty to listen as an object request broker (ORB) on a TCP/IP port for IIOP-RMI requests and call the target EJB method.

Outbound calls are where CICS Liberty makes a request to an ORB to start an EJB method. Outbound calls can be made to the same JVM server the call was made for, or any other Java virtual machine (JVM) capable of acting as an ORB.

In the following example, the *client* is the JVM making the outbound request and the *server* is the JVM receiving the inbound request. Either one, or both of these, can be the CICS Liberty JVM server. For more information, see Configuring Common Secure Interoperability version 2 (CSIv2) in Liberty.

Configuring CSIv2 to use TLS

Inbound

- Create a keystore that contains the certificate for the server.

```
<keyStore id="iiopKeyStore" ... />
```

- Create an SSL repertoire (the SSL element) that references the keystore.

```
<ssl id="iiopSSL" keyStoreRef="iiopKeyStore" />
```

- Create an IIOP endpoint with an IIOPS port.

```
<iiopEndpoint id="defaultIiopEndpoint" host="host.example.com" iiopPort="2809">
    <iiopsOptions iiopsPort="9402" sslRef="iiopSSL" />
</iiopEndpoint>
```

Important: By default the IIOPs options `sslRef` references the `defaultSSLConfig` SSL repertoire.

Outbound

- Create a keystore. You can include a key that allows the keystore to trust a root certificate, which trusts all the certificates that are signed by that certificate.

```
<keystore id="iiopTrustStore" ... />
```

- Create an SSL repertoire (the SSL element) that references the keystore.

```
<ssl id="iiopSSL" trustStoreRef="iiopTrustStore" ... />
```

- Create an ORB with the CSIv2 client policy.

```
<orb id="defaultOrb" nameService="corbaname::host.example.com">
    <clientPolicy.csiv2>
        <layers>
            <transportLayer sslRef="iiopSSL" />
        </layers>
    </clientPolicy.csiv2>
</orb>
```

Configuring CSIv2 to allow propagation of the user ID from the client to the server

Inbound

- Create an ORB with the CSIv2 server policy.

```
<orb id="defaultOrb">
    <serverPolicy.csiv2>
        <layers>
            <attributeLayer identityAssertionEnabled="true" />
        </layers>
    </serverPolicy.csiv2>
</orb>
```

- Optionally, you can specify one or more identities to be trusted by the server.

```
<attributeLayer identityAssertionEnabled="true" trustedIdentities="MYUSER" />
```

Outbound

- Create an ORB with the CSIv2 client policy.

```
<orb id="defaultOrb" nameService="corbaname::host.example.com:2809">
    <clientPolicy.csiv2>
        <layers>
            <attributeLayer identityAssertionEnabled="true" />
        </layers>
    </clientPolicy.csiv2>
</orb>
```

- Optionally, you can provide a trusted identity to be authorized by the server.

```
<attributeLayer identityAssertionEnabled="true" trustedIdentity="MYUSER"
     trustedPassword="MYPASSWD" />
```

Important: The trusted user must exist in a user registry on the server. The `trustedPassword` can be encoded by using the Liberty `securityUtility` tool.

Configuring CSIv2 to use TLS Client Authentication

Inbound

- Create a keystore. You can include a key that allows the keystore to trust a root certificate, which trusts all the certificates that are signed by that certificate.

```
<keyStore id="iiopTrustStore" ... />
```

- Create an SSL repertoire (the SSL element) that references the keystore.

```
<ssl id="iiopSSL" trustStoreRef="iiopTrustStore" ... />
```

- Create an IIOP endpoint with an IIOPS endpoint.

```
<iiopEndpoint id="defaultIiopEndpoint" host="host.example.com" port="2809">
    <iiopsOptions iiopsPort="9402" sslRef="iiopSSL" />
</iiopEndpoint>
```

- Create an ORB with the CSIv2 server policy.

```
<orb id="defaultOrb">
    <serverPolicy.csiv2>
        <layers>
            <attributeLayer identityAssertionEnabled="true" ... />
            <transportLayer sslRef="iiopSSL" />
        </layers>
    </serverPolicy.csiv2>
</orb>
```

Outbound

- Create a keystore that contains the clients certificate.

```
<keyStore id="iiopKeyStore" ... />
```

- Create an SSL repertoire (the SSL element) that references the keystore.

```
<ssl id="iiopSSL" keyStoreRef="iiopKeyStore" />
```

- Create an ORB with the CSIv2 client policy.

```
<orb id="defaultOrb">
    <clientPolicy.csiv2>
        <layers>
            <attributeLayer identityAssertionEnabled="true" />
            <transportLayer sslRef="iiopSSL" />
        </layers>
    </clientPolicy.csiv2>
</orb>
```

Java Message Service (JMS)

Java Message Service (JMS) is an API that allows application components based on Java EE to create, send, receive, and read messages. JMS support in Liberty is supplied as a group of related features that support the deployment of JMS resource adapters.

JMS can run in a managed mode in which queues, topics, connections, and other resources are created and managed through server configuration. This includes the configuration of JMS connection factories, queues, topics, and activation specifications. Alternatively it can run in unmanaged mode where all resources are manually configured as part of the application. The Liberty embedded JMS messaging provider is managed, and therefore all resources are set up as part of the server.xml configuration.

JMS specifications

The JMS specification level supported in a Liberty JVM server is JMS 2.0 support. JMS 2.0 support (`jms-2.0`) enables the configuration of resource adapters to access messaging systems using the Java Message Service API at the 2.0 specification level.

JMS clients

Different JMS client providers are supported in the Liberty JVM server through the following Liberty features:

- WebSphere MQ JMS 2.0 client (`wmqJmsClient-2.0`) - the WebSphere MQ JMS client feature that allows JMS 2.0 or 1.1 client applications to send and receive messages from a remote MQ server.
- WebSphere Application Server JMS 2.0 client (`wasJmsClient-2.0`) - WebSphere Application Server client feature that allow JMS 2.0 or 1.1 client applications to send and receive messages from the messaging engine that is enabled through the `wasJmsServer` feature.
- Any other JMS resource adapter that complies with the JCA 1.6 specification can also be used in Liberty by using generic JCA resource adapters links, see Overview of JCA configuration elements.

JMS providers

Liberty in CICS TS supports usage of the:

- Liberty embedded JMS messaging provider.
  - WebSphere messaging server (`wasJmsServer-1.0`) - the JMS server feature enables the embedded JMS messaging provider to be hosted within Liberty by using the server feature so that a separate JMS server does not need to be installed or configured, see Enabling JMS messaging for a single Liberty server . The server can also be hosted in a separate Liberty instance either inside CICS or in a Liberty server hosted in z/OS or on a distributed platform, see Enabling JMS messaging between two Liberty servers . The WebSphere JMS messaging client component can also be configured to talk to JMS via SIBUS running in a WebSphere Application Server, see Enabling interoperability between Liberty and WebSphere Application Server traditional.
  - WebSphere messaging security (`wasJmsSecurity-1.0`) - the JMS security feature provides security support for the embedded JMS messaging provider client and server components. The JMS security feature can be used with the `cicsts:security-1.0` feature to specify which users from the security registry are to be used by a connection factory when authenticating requests against the embedded JMS messaging server. For information on authorization, see Authorizing users to connect to the messaging engine.
- JMS access to IBM MQ in a CICS standard-mode Liberty JVM server when the JMS application connects using either bindings or client mode transport.
- JMS access to IBM MQ in a CICS integrated-mode Liberty JVM server when the JMS application connects using the client mode transport.
- Third-party JMS resource adapters that comply with the JCA 1.6 specification.

Java Management Extensions API (JMX)

The Java Management Extensions API (JMX) is used for resource monitoring and management.

JMX is a Java framework and API that provides a way of exposing application information by using a widely accepted implementation. Various tools, such as JConsole can then be configured to read that information. The information is exposed by using managed beans (MBeans) - non-static Java classes with public constructors. Get and set methods of the bean are exposed as attributes , while all other methods are exposed as operations.

You can connect to JMX in a Liberty JVM server to view the attributes and operations of MBeans , both locally and from a remote machine. A local connection requires adding the `localConnector-1.0` feature to your `server.xml` and allows you to connect from within the same JVM server. Adding the `restConnector-1.0` feature to your `server.xml` allows you to connect by way of a RESTful interface, which provides remote access to JMX.

Using WebSphere MBeans to monitor your applications

1. To begin, you must acquire a reference to your MBeanServer. This example looks for the **`JvmStats`** MBean and uses the **`findMBeanServer`** method to check which server the MBean is registered to. Then, referring to the correct MBeanServer object, you can obtain reference to your MBean and get data back from the attributes that it exposes. This example looks for the **`UpTime`** attribute of the **`JvmStats`** MBean.

```
// Create an ObjectName object for the MBean that we're looking for.
ObjectName beanObjName = null;
beanObjName = new ObjectName("WebSphere:type=JvmStats");

// Obtain the full list of MBeanServers.
java.util.List servers = MBeanServerFactory.findMBeanServer(null);
MBeanServer mbs = null;

// Iterate through our list of MBeanServers and attempt to find the one we want.
for (int i = 0; i < servers.size(); i++)
{
    // Check if the MBean domain matches what we're looking for.
    mbs = (MBeanServer)servers.get(i);

    if ( mbs.isRegistered(beanObjName))
    {
        Object attributeObj = mbs.getAttribute(beanObjName, "UpTime");
```

```
            System.out.println("UpTime of JVM is: " + attributeObj + ".");
        }
    }
```

Remote connectivity to JMX in Liberty

Remote connectivity to JMX in a Liberty JVM server requires use of an SSL connection and Java Platform, Enterprise Edition (JEE) role authorization. The client code then obtains a reference to the remote MBean using a `JMXServiceURL`.

1. All the JMX MBeans accessed through the REST connector are protected by a single JEE role named **`administrator`**. To provide access to this role edit the `server.xml` and add the authenticated user to the administrator role.

   ```
   <administrator-role>
   <user>myuserid</user>
   <group>group1</group>
   </administrator-role>
   ```

   For more information on using JEE roles, see Authorization using SAF role mapping.
2. A remote RESTful JMX client must access the Liberty JVM server by using SSL. To configure SSL support for a Liberty JVM server, refer to topic Configuring SSL (TLS) for a Liberty JVM server using RACF. In addition, the JMX client requires access to the restConnector client-side JAR file and an SSL client keystore containing the server's signing certificate. The `restConnector.jar` comes as part of the CICS WLP installation, which is available at `&USSHOME;/wlp/clients`.
3. In the client-side code, you need to create a `JMXServiceURL` object. This allows you to obtain a reference to the remote MBeanServerConnection object. See example where `<host>` and `<httpsPort>` match those of your server:

```
JMXServiceURL url = new JMXServiceURL("service:jmx:rest://<host>:<httpsPort>/IBMJMXConnectorREST");
JMXConnector jmxConnector = JMXConnectorFactory.connect(url, environment);
MBeanServerConnection mbsc = jmxConnector.getMBeanServerConnection();
```

4. When you successfully obtain the connection, the MBeanServerConnection object provides the same capability and set of methods as a local connection from the MBeanServer object.

For more information about the MBeans that are provided by WebSphere, see Liberty profile: List of provided MBeans.

Java Authorization Contract for Containers (JACC)

Liberty supports authorization that is based on the Java Authorization Contract for Containers (JACC) specification in addition to the default authorization. When security is enabled in Liberty, the default authorization is used unless a JACC provider is specified.

About this task

JACC enables third-party security providers to manage authorization in the application server. The default authorization does not require special setup, and the default authorization engine makes all of the authorization decisions. However, if a JACC provider is configured and set up for Liberty to use, all of the enterprise beans and web authorization decisions are delegated to the JACC provider. JACC defines security contracts between the Application Server and authorization policy modules. These contracts specify how the authorization providers are installed, configured, and used in access decisions. To add the `jacc-1.5` feature to your Liberty server, add a third-party JACC provider which is not a part of Liberty.

You can develop a JACC provider to have custom authorization decisions for Java EE applications by implementing the `com.ibm.wsspi.security.authorization.jacc.ProviderService` interface that is provided in the Liberty server. The JACC specification, JSR 115, defines an interface for authorization providers. In the Liberty server, you must package your JACC provider as a user feature. Your feature must implement the `com.ibm.wsspi.security.authorization.jacc.ProviderService` interface.

Procedure

1. Create an OSGi Bundle Project to develop the Java class.

   Your project might have compile errors. To fix these errors, you need to import two packages, `javax.security.jacc` and `com.ibm.wsspi.security.authorization.jacc`.

   Edit the file `MANIFEST.MF` to import the missing package:

```
Manifest-Version: 1.0
Service-Component: OSGI-INF/myjaccExampleComponent.xml,
Bundle-ManifestVersion: 2
Bundle-Name: com.example.myjaac.osgiBundle
Bundle-SymbolicName: com.example.myjaac.osgiBundle
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Import-Package: com.ibm.wsspi.security.authorization.jacc;version="1.0.0",
javax.security.jacc;version="1.5.0"
```

   An example of the service component XML, `myjaccExampleComponent.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" immediate="true"
 name="TestPolicyServiceProvider">
    <implementation class="com.example.myjaac.osgiBundle.TestPolicyServiceProvider"/>
    <property name="javax.security.jacc.policy.provider" type="String" value=""/>
    <property name="javax.security.jacc.PolicyConfigurationFactory.provider" type="String" value=""/>
    <service>
        <provide interface="com.ibm.wsspi.security.authorization.jacc.ProviderService"/>
    </service>
</scr:component>
```

2. Create a Liberty Feature Project to add the previous OSGi bundle into the user Liberty feature, under `Subsystem-Content` in the feature manifest file.
3. Refine the feature manifest to add the necessary OSGi subsystem content: `com.ibm.ws.javaee.jacc.1.5; version="[1,1.0.200)"; location:="dev/api/spec/"`.

```
Subsystem-ManifestVersion: 1.0
IBM-Feature-Version: 2
IBM-ShortName: jacc15CICSLiberty-1.0
Subsystem-SymbolicName: com.example.myjaac.libertyFeature;visibility:=public
Subsystem-Version: 1.0.0
Subsystem-Type: osgi.subsystem.feature
Subsystem-Content: com.example.myjaac.osgiBundle;version="1.0.0",
 com.ibm.ws.javaee.jacc.1.5;version="[1,1.0.200)";location:="dev/api/spec/"
Manifest-Version: 1.0
```

   If you need to add one more Subsystem-Content, you must add at least one `space` before you type the content. If you do not add a `space`, CICS returns `java.lang.IllegalArgumentException`.
4. Export the Liberty Feature Project as a Liberty Feature (ESA) file.
5. FTP the ESA file to zFS.
6. Use the `installUtility` command to install the ESA file.

```
./wlpenv installUtility install myFeature.esa
```

7. Add the `jacc-1.5` feature and the ESA file containing the JACC provider as a user feature to `server.xml`.

```
<feature>jacc-1.5</feature>
<feature>usr:jacc15CICSLiberty-1.0</feature>
```

Java Authentication Service Provider Interface for Containers (JASPIC)

The Java Authentication Service Provider Interface for Containers (JASPIC) specification defines a service provider interface (SPI). Authentication providers, that implement message authentication mechanisms, can be integrated in client or server message processing containers or runtimes.

About this task

Authentication providers that are integrated through the JASPIC interface, operate on network messages that are provided by their calling container. The providers transform outgoing messages so that the source of the message can be authenticated by the receiving container, and the recipient of the message can be authenticated by the message sender. Incoming messages are authenticated and returned to their calling container, which is the identity that is established as a result of the message authentication.

JSR 196 defines a standard SPI, and standardizes how an authentication module is integrated into Java EE containers. A message processing model and details of a number of interaction points on the client and server are provided. A compatible web container uses the SPI at these points to delegate the corresponding message security processing to a server authentication module (SAM).

Liberty supports the use of third-party authentication providers that are compliant with the servlet container that is specified in `jaspic-1.1`. The servlet container defines interfaces that are used by the security runtime environment in collaboration with the web container. These start authentication modules before and after a web request is processed by an application. Authentication that uses JASPIC modules is used only when JASPIC is enabled in the security configuration.

Procedure

1. Create an OSGi Bundle Project to develop the Java class.

   Your project might have compile errors. To fix these errors, you need to import two packages, `javax.security.auth.message` and `com.ibm.wsspi.security.jaspi`. The Target Platform must be edited to add the missing JARs into the lists `com.ibm.ws.security.jaspic` from `<cics_install>/wlp/lib` directory and `com.ibm.ws.javaee.jaspic.<version_number>` from `<cics_install>/wlp/dev/api/spec` directory. FTP these to your development system and add them to the build path.

   Edit the file `MANIFEST.MF` to import the missing package.

   ```
   Manifest-Version: 1.0
   Bundle-ManifestVersion: 2
   Bundle-Name: com.example.myjaspic.osgiBundle
   Bundle-SymbolicName: com.example.myjaspic.osgiBundle
   Bundle-Version: 1.0.0
   Bundle-RequiredExecutionEnvironment: JavaSE-1.7
   Import-Package: com.ibm.wsspi.security.jaspi;version="1.0.13",
   javax.security.auth.message;version="1.0.0",
   javax.security.auth.message.callback;version="1.0.0",
   javax.security.auth.message.config;version="1.0.0",
   javax.security.auth.message.module;version="1.0.0",
   javax.servlet;version="2.7.0",
   javax.servlet.http;version="2.7.0"
   Service-Component: myjaspicExampleComponent.xml
   ```

   An example of the service component XML, `myjaspicExampleComponent.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="com.example.myjaspic.osgiBundle">
    <implementation class="com.example.myjaspic.osgiBundle.TestJASPICProviderService"/>
    <service>
        <provide interface="com.ibm.wsspi.security.jaspi.ProviderService"/>
    </service>
</scr:component>
```

2. Create a Liberty Feature Project to add the previous OSGi bundle into the user Liberty feature, under `Subsystem-Content` in the feature manifest file.

3. Edit the feature manifest to add the necessary OSGi subsystem content: `com.ibm.websphere.appserver.jaspic-1.1; type="osgi.subsystem.feature"`.

```
Subsystem-ManifestVersion: 1.0
IBM-Feature-Version: 2
IBM-ShortName: jaspic11CICSLiberty-1.0
Subsystem-SymbolicName: com.example.myjaspic.libertyFeature;visibility:=public
Subsystem-Version: 1.0.0.201611081617
Subsystem-Type: osgi.subsystem.feature
Subsystem-Content: com.example.myjaspic.osgiBundle;version="1.0.0",
  com.ibm.websphere.appserver.jaspic-1.1;type="osgi.subsystem.feature",
  com.ibm.websphere.appserver.servlet-3.0;ibm.tolerates:="3.1";type="osgi.subsystem.feature"
Manifest-Version: 1.0
```

   If you need to add one more Subsystem-Content, you must add at least one `space` before you type the content. If you do not add a `space`, CICS returns `java.lang.IllegalArgumentException`.
4. Export the Liberty Feature Project as a Liberty Feature (ESA) file.
5. FTP the ESA file to zFS.
6. Use `installUtility` to install the ESA file.

   ```
   ./wlpenv installUtility install myFeature.esa
   ```
7. Add the `jaspic-1.1` feature and the ESA file containing the JASPIC provider as a user feature to `server.xml`.

   ```
   <feature>jaspic-1.1</feature>
   <feature>usr:jaspic11CICSLiberty-1.0</feature>
   ```

Java EE Connector Architecture (JCA)

JCA connects enterprise information systems such as CICS, to the JEE platform.

JCA supports the qualities of service for security credential management, connection pooling and transaction management, provided by the JEE application server. Using JCA ensures these qualities of service are managed by the JEE application server and not by the application. This means the programmer is free to concentrate on writing business code and need not be concerned with quality of service. For information about the provided qualities of service and configuration guidance see the documentation for your JEE application server. JCA defines a programming interface called the Common Client Interface (CCI). This interface can be used with minor changes to communicate with any enterprise information system.

The programming interface model

Applications that use the CCI have a common structure for all enterprise information systems. JCA connects the enterprise information systems (EIS) such as CICS, to the JEE platform. These connection objects allow a JEE application server to manage the security, transaction context and connection pools for the resource adapter. An application must start by accessing a connection factory from which a connection can be acquired. The properties of the connection can be overridden by a ConnectionSpec object. After a connection has been acquired, an interaction can be created from the connection to make a particular request. The interaction, like the connection, can have custom properties that are set by the InteractionSpec class. To perform the interaction, call the execute() method and use record objects to hold the data. For example:

```
ConnectionFactory cf = <Lookup from JNDI namespace>
Connection c = cf.getConnection(ConnectionSpec);
Interaction i = c.createInteraction();
InteractionSpec is = newInteractionSpec();
i.execute(spec, input, output);
i.close();
c.close();
```

The example shows the following sequence:

1. Use the ConnectionFactory object to create a connection object.

2. Use the Connection object to create an interaction object.
3. Use the Interaction object to run commands on the enterprise information system.
4. Close the interaction and the connection.

If you are using a JEE application server, you create the connection factory by configuring it using the administration interface of the server. In the Liberty server this is defined through the `server.xml` configuration. When you have created a connection factory, enterprise applications can access it by looking it up in the JNDI (Java Naming Directory Interface). This type of environment is called a managed environment, and allows a JEE application server to manage the qualities of service of the connections. For more information about managed environments see your JEE application server documentation.

Record objects

Record objects are used to represent data passing to and from the EIS. It is advised that application development tools are used to generate these Records. Rational Application Developer provides the J2C tooling that allows you to build implementations of the Record interface from specific native language structures such as COBOL copybooks, with in-built support for data marshalling between Java and non-Java data types.

Resource adapter example

You can install a basic example resource adapter and configure instances of the resources it provides, see Configuring and deploying a basic JCA Resource Adapter.

The Common Client Interface

The CCI provides a standard interface that allows developers to communicate with any number of EISs through their respective resource adapters, using a generic programming style. The CCI is closely modeled on the client interface used by Java Database Connectivity (JDBC), and is similar in its idea of Connections and Interactions.

Using the JCA local ECI resource adapter

The JCA local ECI resource adapter is provided with CICS TS and invokes local CICS programs. This is an optimized path to migrate applications using the CICS Transaction Gateway ECI resource adapter into CICS Liberty. This section applies to integrated mode Liberty only.

The JCA local ECI resource adapter is used to connect to CICS programs, passing data in either COMMAREAs or channels and containers. The resource adapter is provided by the CICS Liberty feature.

Note: The JCA local ECI resource adapter and the CICS Transaction Gateway ECI resource adapter cannot be used in the same Liberty JVM server.

Table one shows the JCA objects corresponding to the CICS terms.

Table 14. CICS terms and corresponding JCA objects

CICS term	**JCA object: property**
Abend code	CICSTxnAbendException
COMMAREA	Record
Channel	ECIChannelRecord
Container with a data type of BIT	byte[]
Container with a data type of CHAR	String
Program name	ECIInteractionSpec:FunctionName
Transaction	ECIInteractionSpec:TPNName

For further details see “JCA local ECI support” on page 140.

Configuring the JCA local ECI resource adapter

You can configure the JCA local ECI resource adapter using connection factories as defined in the JCA specification.

To start using the JCA local ECI, add the feature `cicsts:jcaLocalEci-1.0` to the featureManager element of the `server.xml`.

```
<featureManager>
<feature>cicsts:jcaLocalEci-1.0</feature>
</featureManager>
```

The JCA local ECI provides a default connection factory **defaultCICSConnectionFactory** bound to the JNDI name **eis/defaultCICSConnectionFactory** . Optionally if a different JNDI name is required configure additional connection factories using the properties subelement as follows:

```
<connectionFactory id="localEci" jndiName="eis/ECI">
<properties.com.ibm.cics.wlp.jca.local.eci/>
</connectionFactory>
```

Tip: You do not need any attributes on the properties element.

Porting JCA ECI applications into a Liberty JVM server

JCA applications can be easily ported into a Liberty JVM server using the JCA local ECI resource adapter support.

Porting

Porting existing JCA applications that use the CICS Transaction Gateway ECI resource adapter from a stand-alone JEE application server into a CICS Liberty JVM server can be achieved through these steps:

1. Add the `cicsts:jcaLocalEci-1.0` , and `webProfile-6.0` features to the `server.xml` file.

   For example:

   ```
   <featureManager>
   ...
   <feature>cicsts:jcaLocalEci-1.0</feature>
   <feature>webProfile-6.0</feature>
   ...
   </featureManager>
   ```

2. You can either update the source so that the JNDI name of the Connection Factory is **eis/defaultCICSConnectionFactory** , or add a **connectionFactory** and **properties.com.ibm.cics.wlp.jca.local.eci** to `server.xml`.
3. Deploy the application into CICS , see Deploying a Java EE application in a CICS bundle to a Liberty JVM server.

If the application uses any restricted features of the ECI resource adapter, the code for the application will have to be changed to remove these unsupported features. For more information, see Restrictions of the JCA local ECI resource adapter.

Using the local ECI resource adapter to link to a program in CICS

Running a program in CICS using the JCA local ECI resource adapter is done by using the execute() method of the ECIInteraction class.

About this task

This task shows an application developer how to use the JCA local ECI resource adapter to run a CICS program passing in a COMMAREA using a JCA record. For further details on how to extend the Record interface to represent a CICS COMMAREA, see Using the JCA local ECI resource adapter with COMMAREA

and for details on how to link to a CICS program that uses channels and containers, see “Using the JCA local ECI resource adapter with channels and containers” on page 100.

Procedure

1. Use JNDI to look up the ConnectionFactory object named `eis/defaultCICSConnectionFactory`.
2. Get a Connection object from the ConnectionFactory.
3. Get an Interaction object from the Connection.
4. Create a new ECIInteractionSpec object.
5. Use the set methods on ECIInteractionSpec to set the properties of the execution, such as the program name and COMMAREA length.
6. Create a record object to contain the input data (see COMMAREA/Channel topics) and populate the data.
7. Create a record object to contain the output data.
8. Call the execute method on the Interaction, passing the ECIInteractionSpec and two Record objects.
9. Read the data from the output record.

```
package com.ibm.cics.server.examples.wlp;

                        import java.io.ByteArrayInputStream;
                        import java.io.ByteArrayOutputStream;
                        import java.io.IOException;
                        import java.io.InputStream;
                        import java.io.OutputStream;

                        import javax.annotation.Resource;
                        import javax.resource.cci.Connection;
                        import javax.resource.cci.ConnectionFactory;
                        import javax.resource.cci.Interaction;
                        import javax.resource.cci.Record;
                        import javax.resource.cci.Streamable;
                        import javax.servlet.ServletException;
                        import javax.servlet.annotation.WebServlet;
                        import javax.servlet.http.HttpServlet;
                        import javax.servlet.http.HttpServletRequest;
                        import javax.servlet.http.HttpServletResponse;

                        import com.ibm.connector2.cics.ECIInteractionSpec;

                        /**
                        * Servlet implementation class JCAServlet
                        */
                        @WebServlet("/JCAServlet")
                        public class JCAServlet extends HttpServlet
                        {
                        private static final long serialVersionUID = 4283052088313275418L;

                        // 1. Use JNDI to look up the connection factory
                        @Resource(lookup = "eis/defaultCICSConnectionFactory")
                        private ConnectionFactory cf;

                        protected void doGet(HttpServletRequest request, HttpServletResponse
                        response)
                        throws ServletException, IOException
                        {
                        try
                        {
                        // 2. Get the connection object from the connection factory
                        Connection conn = cf.getConnection();

                        // 3. Get an interaction object from the connection
                        Interaction interaction = conn.createInteraction();

                        // 4. Create a new ECIInteractionSpec
                        ECIInteractionSpec is = new ECIInteractionSpec();

                        // 5. Use the set methods on ECIInteractionSpec
                        // to set the properties of execution.
                        // Change these properties to suit the target program
                        is.setCommareaLength(20);
                        is.setFunctionName("PROGNAME");
                        is.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
```

```
// 6. Create a record object to contain the input data and populate
data
// Change the contents to suit the data required by the program
RecordImpl in = new RecordImpl();
byte[] commarea = "COMMAREA contents".getBytes();
ByteArrayInputStream inStream = new ByteArrayInputStream(commarea);
in.read(inStream);

// 7. Create a record object to contain the output data
RecordImpl out = new RecordImpl();

// 8. Call the execute method on the interaction
interaction.execute(is, in, out);

// 9. Read the data from the output record
ByteArrayOutputStream outStream = new ByteArrayOutputStream();
out.write(outStream);
commarea = outStream.toByteArray();
}
catch (Exception e)
{
// Handle any exceptions by wrapping them into an IOException
throw new IOException(e);
}
}

// A simple class which extends Record and Streamable representing a
commarea.
public class RecordImpl implements Streamable, Record
{
private static final long serialVersionUID = -947604396867020977L;

private String contents = new String("");

@Override
public void read(InputStream is)
{
try
{
int total = is.available();
byte[] bytes = null;
if (total > 0)
{
bytes = new byte[total];
is.read(bytes);
}
// Convert the bytes to a string.
contents = new String(bytes);
}
catch (Exception e)
{
// Log the exception
e.printStackTrace();
}
}

@Override
public void write(OutputStream os)
{
try
{
// Output the string as bytes
os.write(contents.getBytes());
}
catch (Exception e)
{
// Log the exception
e.printStackTrace();
}
}

@Override
public String getRecordName()
{
// Required by Record, unused in this sample
return "";
}
@Override
public void setRecordName(String newName)
{
// Required by Record, unused in this sample
```

```
                        }
                        @Override
                        public void setRecordShortDescription(String newDesc)
                        {
                        // Required by Record, unused in this sample
                        }
                        @Override
                        public String getRecordShortDescription()
                        {
                        // Required by Record, unused in this sample
                        return "";
                        }
                        @Override
                        public Object clone() throws CloneNotSupportedException
                        {
                        // Required by Record, unused in this sample
                        return super.clone();
                        }
                        }
                        }
```

Results

You have successfully linked to a program in CICS using the ECI resource adapter.

Using the JCA local ECI resource adapter with channels and containers
To use channels and containers with the JCA local ECI resource adapter, the input and output records must be instances of ECIChannelRecord.

When the ECIChannelRecord is passed to the execute() method of ECIInteraction, the method uses the ECIChannelRecord itself to create a channel and converts the entries inside the ECIChannelRecord into containers before passing them to CICS.

This example shows how to build an input and output record for use by the JCA local resource adapter using the put() and get() methods on the ECI ChannelRecord.

```
ECIChannelRecord in = new
                    ECIChannelRecord("CHANNELNAME");
                    byte[] bitData = "Container with BIT data".getBytes();
                    String charData = "Container with CHAR data";
                    in.put("BITCONTAINER", bitData);
                    in.put("CHARCONTAINER", charData);
                    ECIChannelRecord out = new ECIChannelRecord("CHANNELNAME");

                    interaction.execute(is, in, out);

                    bitData = (byte[]) out.get("BITCONTAINER");
                    charData = (String) out.get("CHARCONTAINER");
```

BIT and CHAR containers are created depending on the type of the entry:

- A BIT container is created when the entry data is of type byte[] or an object that implements the Streamable interface. No code page conversion takes place.
- A CHAR container is created when the entry data is of type String. String data is encoded by Unicode and is converted to the encoding of the container. Data read from this container by **EXEC CICS GET CONTAINER** will be converted according to Using containers for code page conversion.

When creating the ECIChannelRecord, the name must be a valid CICS channel name. Once created the getRecordName() method obtains the name of the channel. When adding containers to the ECIChannelRecord, the container names must be valid CICS container names. Once created the KeySet() method retrieves the names of all the containers.

Using the JCA local ECI resource adapter with COMMAREA
To use COMMAREA with the JCA local ECI resource adapter, the input and output records must be instances of classes that implement `javax.resource.cci.Record` and `javax.resource.cci.Streamable`.

This example shows how to build an input and output record for use by the local ECI resource adapter using the read() and write() methods on the Streamable interface:

```
RecordImpl in = new RecordImpl();
                    byte[] commarea = "COMMAREA contents".getBytes();
                    ByteArrayInputStream inStream = new ByteArrayInputStream(commarea);
                    in.read(inStream);
                    RecordImpl out = new RecordImpl();

                    interaction.execute(is, in, out);

                    ByteArrayOutputStream outStream = new ByteArrayOutputStream();
                    out.write(outStream);
                    commarea = outStream.toByteArray();
```

To retrieve a byte array from the output record, use the write method on the Streamable interface using a **java.io.ByteArrayOutputStream** object. The toByteArray() method on **ByteArrayOutputStream** provides the output data from the COMMAREA in the form of a byte array.

To provide more function for your specific JEE components, you can write implementations of the Record interface that allow you to set the contents of the record using the constructor. In this way you avoid use of the **java.io.ByteArrayInputStream** used in the example.

Rational Application Developer provides the J2C tooling that allows you to build implementations of the Record interface from specific native language structures such as COBOL copybooks, with in-built support for data marshalling between Java and non-Java data types.

Unit of work management with JCA

Transaction Management, when using the CICS local ECI resource adapter, is provided by a CICS Liberty JVM server.

Calls to other CICS programs using the CICS local ECI resource adapter are integrated with CICS unit of work (UOW) management. This allows the UOW to be controlled through either syncpoint commands or a JTA transaction.

Calls to a program in a remote CICS region result in a DPL call using a mirror transaction. This mirror task UOW is coordinated by the calling UOW if a Java transaction context is being used, this means the called program is unable to issue syncpoint calls as it is restricted to the DPL command subset. If the calling program has no JTA transaction context then the mirror task UOW is invoked using the SYNCONRETURN option. In this scenario the called program is able to issue syncpoint commands as its UOW is not coordinated by the calling program.

For more details refer to Programming considerations for distributed program link , “Unit of work (UOW) services” on page 63 and “ Java Transaction API (JTA) ” on page 80.

Enabling trace for the JCA local ECI resource adapter

A detailed trace mechanism is provided for the JCA local ECI resource adapter. Enabling trace can be useful for problem solving in applications using the resource adapter.

- JCA local ECI resource adapter trace is enabled by SJ domain trace level 4 (SJ = 4 or SJ = ALL).
- Trace from the resource adapter will be included in the JVM server trace output in zFS with the component identifier `com.ibm.cics.wlp.jca.local.eci.adapter`.

Restrictions of the JCA local ECI resource adapter

Some API calls that are available on the CICS Transaction Gateway ECI resource adapter are not supported by the CICS TS JCA local ECI resource adapter.

Restricted methods

These API calls are not supported by the JCA local ECI resource adapter

- ECIInteractionSpec class methods setExecuteTimeout(), getExecuteTimeout(), setReplyLength(), getReplyLength(), setTranName(), getTranName()
- CICSConnectionSpec class methods setPassword(), getPassword(), setUserid(), getUserid(), addPropertyChangeListener(), removePropertyChangeListener(), firePropertyChange
- ECIConnection class method getLocalTransaction()

- ECIChannelRecord class method values()
- CICSUserInputException
- The constructor ECIConnectionSpec(String username, String password). ECIConnectionSpec() has been added as an alternative
- The constructor ECIInteractionSpec(int verb, int timeout, String prog, int commLen, int repLen). ECIInteractionSpec(int verb, String prog, int commLen) has been added as an alternative

These calls are not supported when using the IBM CICS SDK for Java to develop web applications. In order to allow portability of existing ECI JCA applications into a Liberty JVM server these methods will continue to function but any settings of transaction, timeout, reply length and transaction name will have no effect. Setting the transaction ID through ECIInteractionSpec.setTPNName() only uses the specified transaction when linking to a remote program (DPL). Linking to a local program will continue to use the current transaction.

Non-managed environments

JCA local ECI only supports managed connection factories (those created via the server.xml configuration). Non-managed connections created using instances of a ManagedConnectionFactory, are not supported.

Exception handling

Exception handling may differ slightly between the CICS Transaction Gateway ECI resource adapter and the CICS TS JCA local ECI resource adapter. Any CICS errors will propagate to the ECI local resource adapter as a CICSException, as they do with the JCICS API. The resource adapter will wrap these exceptions in a ResourceException. To help identify the CICS fault, the CICSException will be set as the cause of the Exception and can be accessed using the getCause() method of java.lang.Throwable.

Asynchronous calls

Asynchronous calls are not fully asynchronous in the JCA local ECI resource adapter. A call using the SYNC_SEND interaction verb will block until the program completes, then the results can be gathered via a subsequent call using the SYNC_RECEIVE interaction verb, using the same ECIInteraction.

Unsupported CICS Transaction Gateway functions

These CICS Transaction Gateway functions are not supported in the CICS TS local ECI resource adapter.

- Remote connections to a CICS Transaction Gateway server
- Identity propagation
- Cross component trace (XCT)
- Request monitoring user exits
- Trace from the resource adapter is controlled by CICS TS, use of the CICSLogTraceLevels is not supported.

CICS remote development feature for Java

The CICS remote development feature for Java provides an ECI resource adapter for use in Liberty running on a developers workstation. The feature enables developers to rapidly test and debug Java applications that use JCA APIs to invoke programs in CICS TS. When ready, the application can be deployed into Liberty running in CICS without any further changes to the application.

The feature connects to a CICS region by using an IP interconnectivity (IPIC) connection that is defined by using the TCPIPSERVICE resource. The trace facility can be used to identify problems with the data sent and received from the program in CICS TS.

Configuring the IPIC connection

Before you can test your Java application with a CICS region, an IPIC connection must be available. Contact your CICS system programmer to request a TCP/IP service that accepts IPIC requests from a Liberty profile by using the following details.

About this task

The following procedure guides the CICS system programmer through the steps to define a TCP/IP service in CICS and install a sample user program for IPIC connections.

Procedure

1. Install IPIC support in CICS by defining a **TCPIPSERVICE** resource with the following attributes:

Table 15. Attributes for TCPIPSERVICE resource

TCPIPSERVICE resource attribute	**Value required**
URM	DFHISAIP
Port number	n
Status	OPEN
Protocol	IPIC
Transaction	CISS
Backlog	0
Socketclose	No

2. Verify that the **TCPIPSERVICE** is in service by issuing the **CEMT INQUIRE TCPIPSERVICE(JCA)** command.
3. Install a sample program to test the IPIC connection.
   a) If you do not already have a copy, download the CICS Transaction Gateway Software Development Kit (SDK) and expand the archive file.
   b) Locate and copy the `cicsprograms/ec01.cpp` member to a COBOL source data set on z/OS.
   c) Compile the EC01 sample program and copy the generated module into a load library that CICS can access.
   d) If the `autoinstall` program is not enabled, define and install a program definition for EC01.
   e) Test the EC01 program by issuing the **CECI LINK PROG(EC01) COMMAREA(' ')**.
   Check that the **RESPONSE** is NORMAL.

Results

IPIC support is now available for use in the CICS region.

Setting up your local Java test environment

Before you can test your Java application with a CICS region, you must check that the required tools are installed and also configure your local work environment.

About this task

To create a local work environment where you can test your Java applications with a CICS region, complete the following steps.

Procedure

1. Download and install the Eclipse IDE for Java EE Developers with WebSphere Developer Tools (WDT). Then, install a local Liberty profile server instance, create the Hello World JavaServer Pages (JSP) and test by deploying the Hello World web application on the server.

   For more information, see Getting started with WebSphere Developer Tools for Eclipse and Liberty.
2. Install the JCA remote ECI resource adapter from the Liberty: Liberty Repository. You can install a feature from the repository by using the **installUtility** command:

   ```
   <liberty_install>/bin/installUtility install
   --acceptLicense jcaRemoteEci-1.0
   ```
3. Add the `usr:jcaRemoteEci-1.0` , `localConnector-1.0` , and `webProfile-6.0` features to the `server.xml` file.
   For example, in Eclipse expand the **WebSphere Application Server Liberty Profile** project and then expand **servers** . Double-click **defaultServer** to edit `server.xml` . Click the **source** tab and add the following features:

   ```
   <featureManager>
   ...
   <feature>usr:jcaRemoteEci-1.0</feature>
   <feature>localConnector-1.0</feature>
   <feature>webProfile-6.0</feature>
   ...
   </featureManager>
   ```
4. Add a **connectionFactory** and **properties.com.ibm.cics.wlp.jca.remote.eci** to `server.xml`.

   The **connectionFactory** jndiName is used by the application to create a connection. The **properties.com.ibm.cics.wlp.jca.remote.eci** is used to configure the JCA remote ECI resource adapter and at a minimum must specify **serverName** with the host name and port number of the IPIC connection defined by the TCPIPSERVICE resource.

   Note: You might need to specify extra parameters. For example, to use Secure Sockets Layer (SSL) and a user ID and password. Table 1 lists the available parameters. Table 2 lists the ECI resource adapter deployment parameters that are not supported by the JCA remote ECI resource adapter. For more information, see ECI resource adapter deployment parameters.

   ```
   <server>
   ...
   <connectionFactory id="com.ibm.cics.wlp.jca.local.eci" jndiName="eis/ECI">
   <properties.com.ibm.cics.wlp.jca.remote.eci serverName="tcp://
   hostname
   :
   port"/>
   </connectionFactory>
   ...
   </server>
   ```

   Table 1 shows the JCA remote ECI resource adapter properties that are supported.

Table 16. Supported JCA remote ECI resource adapter properties

JCA object: property	Notes
applid	
applidQualifier	This property is required.
cipherSuites	
ipicHeartbeatInterva l	
ipicSendSessions	This property default is 5.
keyRingClass	

Table 16. Supported JCA remote ECI resource adapter properties (continued)	
JCA object: property	**Notes**
keyRingPassword	
password	
socketConnectTimeout	
serverName	This property is required.
traceLevel	
traceRequest	
userName	

Table 2 shows CICS Transaction Gateway ECI resource adapter deployment parameters that are not supported by the JCA remote ECI resource adapter.

Table 17. JCA remote ECI resource adapter properties that are not supported
JCA object: property
interceptPlugin
portNumber
tPNName
tranName

Testing the example Java EE JCAServlet application

Add the example Java EE JCAServlet application and then verify that the Java EE application can call a sample program in CICS.

About this task

Complete the following steps to add the Java EE JCAServlet application. Then, verify that the Java EE JCAServlet application can call the EC01 sample program in CICS.

Procedure

1. Create a JCAServlet class in the Hello World web application.

   Expand the **Hello World** project and then expand **Java Resource** . Right-click **New** and select **Servlet**.

   - For **Java package** , enter `com.ibm.ctg.samples.liberty`
   - For **Class name** , enter `JCAServlet`

   Then, click **Finish**.
2. Edit `JCAServlet.java` and replace all of the code with the CICS example `JCAServlet.java` from GitHub.

   For more information, see JCAServlet.java.
3. Expand the **Hello World** project and then expand **Java Resources** > **src** > **com.ibm.ctg.samples.liberty** . Right-click the **JCAServlet.java** application and select **Run As** > **Run on server**.
4. The Liberty server is started and a message is displayed on the Liberty server console that indicates the URL, which you can click to run the Java EE application. The following is an example of a message, which might be displayed.

```
[AUDIT ] CWWKT0016I: Web application available
(default_host):
http://localhost:9080/GenappCustomerSearchWeb/
```

Results

You can now test and debug the Java EE application in your local Liberty server.

Configuring the trace function in your local Liberty profile

Before you can trace your Java web application in your local Liberty profile, you must configure your local work environment.

About this task

Complete the following steps to configure the trace function in your local Liberty profile.

Procedure

Add the `traceRequests="ON"` parameter to the connection factory in your `server.xml` file to enable tracing.

With `traceRequests="ON"` specified, when you send another application request, the Eclipse console shows the request and the response that your application sends and receives from CICS.

The following example shows a request that is sent to CICS and the response received from CICS.

```
Starting DataFlowsMonitor log stream at
Thu Apr 07 14:21:54 BST 2016[00000000001]:
com.ibm.ctg.monitoring.DataFlowsMonitor:eventFired called with
event = RequestEntry
FlowType = EciSynconreturn Fully qualified APPLID = No APPLID
CtgCorrelator =
1Program = EC01Server =
TCP://WINMVS2C.HURSLEY.IBM.COM:27723PayLoad = COMMAREA is 20
bytes00000000 00000000 00000000 00000000 00000000
'????????????????????'
```

```
[00000000001]:
com.ibm.ctg.monitoring.DataFlowsMonitor:eventFired called with
event = ResponseExit
FlowType = EciSynconreturn Fully qualified APPLID = No APPLID
CtgCorrelator =
1OriginData - Transaction Group ID = 1B114040
40404040 40402EF0 F0F0F0F0 F0F0F2D0 8F53F115 220100Program = EC01Server =
TCP://WINMVS2C.HURSLEY.IBM.COM:27723PayLoad = COMMAREA is 20
bytesF0F761F0 F461F1F6 40F1F47A F2F17AF5 F4000000
'??a??a??@??z??z?????'CtgReturnCode = 0CicsReturnCode = 0
```

Results

You can now trace your application to help identify any problems.

Configuring a secure SSL connection

You can secure the IPIC connection from the JCA remote ECI resource adapter to CICS by using SSL.

About this task

Complete the following steps to configure a secure SSL connection.

Completing this setup provides SSL with trusted Certificates exported from both MVS and the local client. An MVS user ID and password are also required for authentication.

Procedure

1. Set up a CICS RACF® environment.

For more information, see Configuring SSL server authentication on the CICS server .

2. Set up the client security.

   For more information, see Configuring SSL server authentication on the client.

3. Configure the client authentication.

   For more information, see Configuring SSL client authentication.

4. Configure the IPIC connection on CICS.

   For more information, see Configuring the IPIC connection on CICS .

5. Modify your `server.xml` to use the local **KeyRingClass** that was created in Step 2 and send your user ID and password.

```
<connectionFactory id="com.ibm.cics.wlp.jca.local.eci"
jndiName="eis/ECI">
<properties.com.ibm.cics.wlp.jca.remote.eci
serverName="ssl://hostname:port"
keyRingClass="C:\Users\IBM_ADMIN\Documents\CICS\JCA\ctgclientkeyring.jks"
keyRingPassword="password"
userName="user_ID"
password="*******"
applid="JCASSL"
applidQualifier="ABCDEFGH"
/>
</connectionFactory>
```

Results

The JCA remote ECI resource adapter secures requests to CICS by using SSL and the key ring, user ID, and password that are specified in `server.xml`.

Developing microservices with MicroProfile

Eclipse MicroProfile defines a programming model for developing microservice applications in an Enterprise Java environment. It is an open source project under The Eclipse Foundation, to bring microservices to the Enterprise Java community. MicroProfile is supported by Liberty.

MicroProfile defines a number of specifications for building microservices that are resilient, secure and easy to monitor.

Table 18. Included in Eclipse MicroProfile 1.2

Specification	Description
JSR 346: Contexts and Dependency Injection for Java EE 1.1	CDI defines a set of services that manage the injection and lifecycle of objects in an Enterprise Java runtime.
JSR 339: JAX-RS 2.0: The Java API for RESTful Web Services	JAX-RS is a Java API for RESTful Web Services.
JSR 353: Java API for JSON Processing	JSON-P is a Java API for processing JSON.
Eclipse MicroProfile Config 1.1	Config is a Java API and SPI for managing application configuration.
Eclipse MicroProfile Fault Tolerance 1.0	Fault Tolerance provides strategies for coping with failures when calling external services.
Eclipse MicroProfile Health Check 1.0	Health Check allows components to report their liveliness to the wider system.
Eclipse MicroProfile Health Metrics 1.0	Health Metrics provide a unified way for applications to expose monitoring data.

Table 18. Included in Eclipse MicroProfile 1.2 (continued)	
Specification	**Description**
Eclipse MicroProfile JWT Propagation 1.0	JWT Propagation allows JSON Web Token (JWT) to be used for authentication and authorization with Java EE role-based access control (RBAC).

Known Restrictions

- CDI is used extensively in the MicroProfile APIs, however Liberty does not support CDI in OSGi web applications that are packaged in enterprise bundle archives (EBAs). Instead, package applications that use MicroProfile in web application archives (WARs) or enterprise application archives (EARs).
- MicroProfile Fault Tolerance 1.0 is designed to manage calls made to other services. It is not designed to manage updates to resources in a transactional context. CICS resources should not be updated in methods annotated `@Bulkhead`, `@CircuitBreaker`, `@Fallback`, `@Timeout` or `@Retry`. CICS cannot guarantee that these updates will be recovered when exceptions occur, even when JTA is used.
- When the feature `mpJwt-1.0` is enabled in the `server.xml` of a Liberty JVM server, all authentication must be done by using JWT bearer tokens. To use any other form of authentication, a separate Liberty JVM server must be used.

Service Architectures in CICS Liberty JVM servers

Monolithic Architecture

Monolithic architecture implements the application in a single unit. Internally the logic can be modular, but externally the application is either entirely available, or not available at all. Monoliths perform well compared to microservices and are less complex when managing security and transaction context. Scaling monoliths involves adding instances of the entire application, individual parts cannot be scaled.

Figure 4. Monolithic architecture

Backing Services

Backing services allow the back end data and programs to be decoupled from the main application. By making the data and programs into separate applications, they are called using a platform agnostic communication method, for example HTTP, socket, message queues, etc. Instead of the main application holding all the logic for communicating with these sources, some responsibility is given to these services.

In CICS, z/OS Connect is used to expose CICS programs as backing services through a REST API. In the example, the application uses JDBC to communicate with a database. SMTP is used to send emails and HTTP is used to call a CICS program through z/OS Connect.

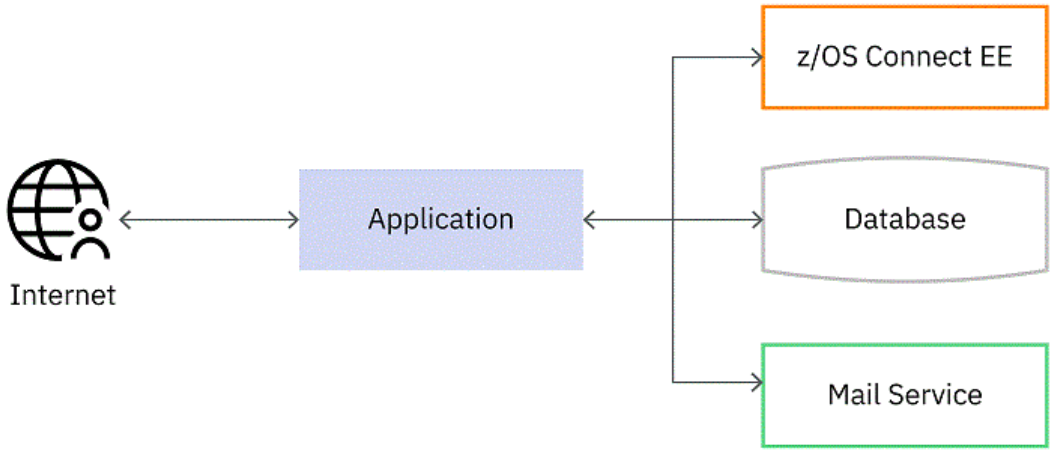


Figure 5. Backing services

Hosted Services

Services are hosted in CICS to further decouple the main application from the various components. Similar to backing services, additional functionality is exposed in CICS through CICS Web Services, or in CICS Liberty with applications using technologies including servlets, JAX-RS and JAX-WS.

JAX-RS is a popular technology for creating RESTful web services, JAX-WS is used to create remote procedure call (RPC) orientated web services.

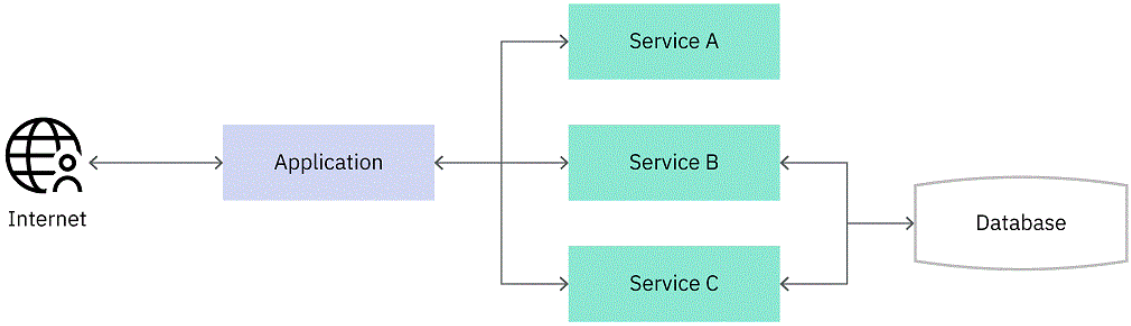


Figure 6. Hosted services

Note: Both REST and RPC are equally valid options for communication in microservices. REST focuses on resource management. RPC focuses on actions. A microservice architecture does not mandate REST, RPC or any other technology.

Microservices

A full microservice architecture is an interconnected web of isolated services with no single central point, though there can be dedicated entry points. Services can communicate with one another as required. Scaling microservices involves adding instances of the parts which require scaling. Microservices are more resilient to failures than monoliths.

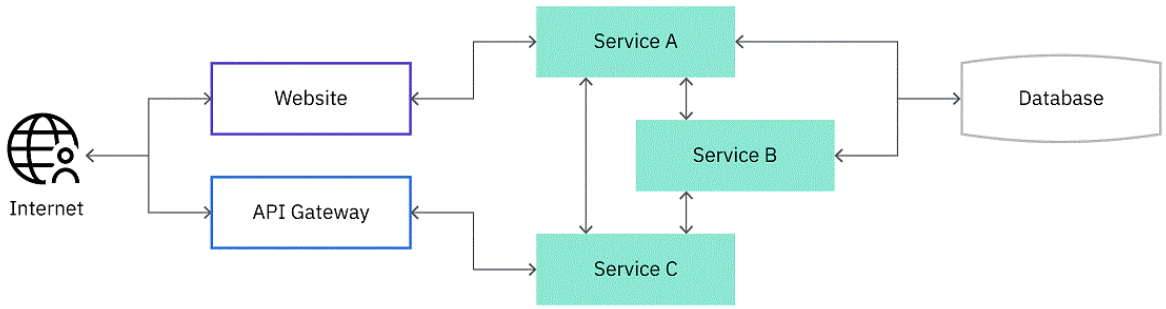


Figure 7. Microservices

Scaling Services in CICS

Services can be scaled in several ways in CICS, depending on region topology and setup. Microservices are typically isolated into a single container. In CICS, a service or set of services could be isolated within a region or JVM server. Scaling can be achieved by running multiple CICS regions hosting the same service or set of services. You can also scale the JVM server by increasing the number of threads.

Securing Microservices

Where possible, microservices should be kept off public networks. API gateways can be used to provide controlled access to microservices. MicroProfile offers a method for using Open ID Connect (OIDC) based JSON Web Tokens (JWT) for role based access control (RBAC) of microservice endpoints. Security tokens offer lightweight and interoperable propagation of user identities across different services.

MicroProfile JWT Authentication 1.0 provides functionality to authenticate and authorize users based on a JWT bearer token. The token can be injected into the service code and used to propagate the identity across the microservice network. Propagation of the JWT can be done manually by including the JWT as a bearer token in the Authorization HTTP header on the outbound request. Alternatively, Liberty can automatically propagate the JWT by configuring a `webTarget` element in `server.xml` with an `authnToken` configured, for example:

```
<webTarget uri="http://microservice.example.ibm.com/protected/*" authnToken="mpjwt" />
```

Important: JWT identities are not automatically mapped to a user registry and will not be propagated into the CICS task user ID. To enable identity mapping, add `mapToUserRegistry=â€�trueâ€�` configuration attribute to the `<mpJwt>` element in `server.xml`.

For more information on configuring MicroProfile JWT Authentication in Liberty, see Configuring the MicroProfile JSON Web Token.

Data Consistency in Microservices

Microservices cannot easily make use of distributed transactions. Instead alternative transaction strategies are used, such as the saga pattern, where events are published after an updated in a service. For example, if service A and service B have updates which should both happen, the following sequence occurs:

1. A updates into a pending state
2. A sends a message to B
3. B updates into a complete state
4. B sends a message to A
5. A updates into a complete state

When to use Microservices

Microservices are best applied where an application can be deconstructed into smaller, isolated, services. A microservice allows for controlled scaling, independent deployment and more autonomous development. The architecture of microservices can create additional complexity, particularly in deployment and data consistency. Communicating over protocols such as HTTP produces a larger performance cost compared to calling in memory. Components can be made more resilient to failure by allowing them to scale individually. Monitoring solutions become more important to aid diagnosis of unhealthy services when managing a microservice architecture.

Spring Boot applications

You can develop Spring Boot applications for use with CICS. You can choose to integrate your Spring Boot application with aspects of Java EE such as Security, Transactions, DataSources and Java Message Service (JMS), or you can use standard Spring configuration and templates with little or no integration with Java EE.

To achieve integration with Java EE and Liberty, build and deploy your Spring Boot application as a WAR file and follow the best practices that are described in topic Building and deploying Spring Boot applications. Each subtopic describes an important aspect of integration and how to ensure Spring Boot integrates with Java EE, Liberty, and CICS capabilities.

JCICS in Spring Boot applications

You can use JCICS in your Spring Boot applications to call CICS services. JCICS is integrated by default for WARs.

Although you can resolve your Spring Boot dependencies against JCICS by using the com.ibm.cics.server artifact on Maven Central, a more consistent approach is to use the JCICS bill of materials (BOM). This ensures you resolve against consistent versions of a range of CICS artifacts as shown in the examples below.

Avoid binding the JCICS library into your application as this is provided by the CICS runtime.

If you are using Maven, you can achieve this by compiling against the JCICS library by using `<scope>provided</scope>`. Or, if you are using the CICS TS BOM, the `<scope>import</scope>` on the `<dependency>` element automatically defers the scope value to the CICS BOM. The CICS BOM applies the 'provided' scope, which ensures JCICS is only included at build time. It is not embedded in your application where it might potentially conflict with the version that is used by the CICS runtime. For example,

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.ibm.cics</groupId>
            <artifactId>com.ibm.cics.ts.bom</artifactId>
            <version>5.5-20190701171918-PH10453</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

If you are using Gradle, you can either use the compileOnly directive, for example: `compileOnly("com.ibm.cics:com.ibm.cics.server:1.700.0-5.5-PH10453")`, or you can use the CICS TS BOM by using the 'enforcedPlatform' directive. Doing so infers the JCICS library version from the BOM in a more consistent and compatible manner. For example,

```
compileOnly enforcedPlatform('com.ibm.cics:com.ibm.cics.ts.bom:5.5-20191121085445-PH14856')
compileOnly("com.ibm.cics:com.ibm.cics.server")
```

Note: Consult Maven Central for the latest version number for appropriate for your release of CICS.

For more information about building applications with Maven and Gradle, see Developing applications using other tools.

JPA in Spring Boot applications

Developers can use the Java Persistence API (JPA) to create object-oriented versions of relational database entities to use in their applications.

To use JPA in your Spring Boot application, first add a JPA artifact to your dependencies in your Spring Boot application. For example:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Java EE's implementation of JPA and Spring data's implementation of JPA both require configuration to define the connection to the database repository that is used by the application.

Just like using JDBC you can use the **`spring.datasource.jndi-name`** property that is defined in `application.properties` to configure the connection to the datasource being used and this will be used dynamically by the JPA EntityManager. Alternatively the data source can also be defined in an @Bean annotated dataSource() method, by performing a JNDI lookup of a data source defined in Liberty.

Security in Spring Boot applications

You have three options when you are using Spring Boot security in CICS.

1. You can use Spring Boot security without integrating with Liberty or CICS security. This option is useful if you are taking an existing Spring Boot application and deploying it unchanged in CICS.
2. You can use Java EE security to authenticate web requests by using any of the Liberty-supported registry types. You can configure it in the standard Java EE method by using a `<security-constraint>`and `<login-config>` in the application's `web.xml`. This option is useful if you want to authenticate users by using any of the supported Liberty registry types, and then control transaction authorization by using CICS security. For more information, see Authenticating users in a Liberty JVM server

   Note: You must ensure that `web.xml` is stored in `src/main/webapp/WEB-INF/`
3. You can integrate Spring Boot security with Java EE security by using Java EE container pre-authentication. It allows you to authenticate users via an external system in order to provide a validated user ID and set of roles to Spring Boot security. To do this, you need to modify the application and create an `@Configuration` annotated class that extends WebSecurityConfigurerAdapter in order to name the roles to be propagated into Spring security. In addition, you then need to configure the standard Java EE security settings in the applications `web.xml` and `<application-bnd>` or EJBROLE profiles if you are using SAF authorization. Use this option if you want to authenticate users by using any of the supported Liberty registry types, and you want to authorize requests by using Java EE role-based access to individual methods

Transactional integration and Spring Boot applications

You can achieve transactional integration when you are developing Spring Boot applications for use with CICS Liberty. The effect of transactional integration between Spring Boot and CICS is to ensure that the CICS Unit of Work (UOW) is coordinated by Liberty's transaction manager. Using the Java Transaction API (JTA) you can coordinate CICS, Liberty, and third-party resource managers, such as a type 4 database driver connection, together as one global transaction. For more information about JTA support in CICS, see Java Transaction API (JTA).

JTA is available for use in a Spring Boot WAR application in various ways:

- Spring Boot's @Transactional annotation: This annotation, which is specified at the class or method level denotes the code segment to be contained within a single global transaction.
- Spring templates: The Spring framework provides two templates for use with programmatic transaction management: the TransactionTemplate and the PlatformTransactionManager interface.

- UserTransaction: It is also possible to use the JTA UserTransaction interface within a Spring Boot application by obtaining the UserTransaction initial context of the hosting Application server (Liberty) through a JNDI lookup. For example, `ctx.lookup("java:comp/UserTransaction");`. The developer can employ a Bean-managed approach to transactions by explicitly coding UserTransaction 'start' and 'end' calls around the resources to be managed.

Threading and Concurrency in Spring Boot applications

The Spring Framework provides abstractions for asynchronous execution of tasks by using the TaskExecutor interface. Executors are the Java SE name for the concept of thread pools. Spring's TaskExecutor interface is identical to the java.util.concurrent.Executor interface. The TaskExecutor was originally created to give other Spring components an abstraction for thread pooling where needed. Spring includes a number of pre-built implementations of TaskExecutor but it is the DefaultManagedTaskExecutor that is most useful for integration with CICS as it looks up the application server's defaultExecutor - which in CICS Liberty is designed to provide CICS enabled threads.

About this task

To run asynchronous tasks in your Spring Boot application by using CICS enabled threads, there are two options. You can either set an Asynchronous Executor for the whole application, or you can choose to specify an AsyncExecutor on a per method basis. If all the tasks you spawn asynchronously require CICS services, then setting the Asynchronous Executor for the whole application is the simplest approach. Otherwise, you need to specify the Asynchronous Executor to use for each and every method where you require asynchronous capability. Here, we demonstrate the whole application approach.

Procedure

1. On your main Spring Boot Application class add the @EnableAsync annotation, implement the interface AsyncConfigurer, and override the getAsyncExecutor() and AsyncUncaughtExceptionHandler methods. Ensure you return an instance of the DefaultManagedTaskExecutor in the getAsyncExecutor() method as this obtains new threads from Liberty's defaultExecutor, which in turn is configured to return CICS enabled threads. For more information about the AsyncConfigurer, see the AsyncConfigurer in the Spring Boot documentation. For usage examples, see EnableAsync in the Spring Boot documentation.

```
@SpringBootApplication
@EnableAsync
public class MyApplication implements AsyncConfigurer
{

    public static void main(String[] args)
    {
        SpringApplication.run(MyApplication.class, args);
    }

    @Override
    @Bean(name = "CICSEnabledTaskExecutor")
    public Executor getAsyncExecutor()
    {
        return new DefaultManagedTaskExecutor();
    }

    @Override
    public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler()
    {
        return new CustomAsyncExceptionHandler();
    }
}

public class CustomAsyncExceptionHandler implements AsyncUncaughtExceptionHandler
{
    @Override
    public void handleUncaughtException(Throwable throwable, Method method, Object... obj)
    {
        System.out.println("Exception Cause - " + throwable.getMessage());
        System.out.println("Method name - " + method.getName());
        for (Object param : obj)
        {
```

```
            System.out.println("Parameter value - " + param);
        }
    }
}
```

2. Add the @Async annotation to either: a class in your application if you wish to run all methods on that class asynchronously, or to individual methods that you wish to run asynchronously. i.e.`@Async("CICSEnabledTaskExecutor")`
3. Add the concurrent-1.0 feature to `server.xml`

JDBC in Spring Boot applications

You can use Spring Data JDBC to implement JDBC based repositories. It allows you to access DB2 and other data sources from your Spring Boot application.

Spring Data JDBC is conceptually simpler than JPA, for more information on how it differs from JPA, see: Reference documentation in the Spring Boot documentation. To use JDBC in your Spring Boot application, add a JDBC artifact to your dependencies in your Spring Boot application to make the necessary Java libraries available. For example, in Maven:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
```

Or in Gradle, (implementation"`org.springframework.boot:spring-boot-starter-data-jdbc`")

To use JDBC in a Spring Boot application, you can define a Liberty dataSource in the `server.xml` just as you would if using JDBC in a Java EE application. This data source can then be located by using a JNDI lookup that references the jndiName attribute on the dataSource element, and then used by the Spring Boot JdbcTemplate object by using one of the following methods:

1. Performing a JNDI lookup of the data source in an @Bean annotated dataSource() method and returning the data source.
2. Naming the data source in the **`spring.datasource.jndi-name`** in the Spring application properties. Spring Boot creates the JdbcTemplate using the data source that is named in `application.properties`.

Note: In option 2, it is also possible to configure all the data source attributes necessary to connect a Spring application to the required data source from within the `application.properties` file. The attributes are all defined in Common application properties in the Spring Boot documentation. However, this ties the application directly the data source and using JNDI is a more flexible approach.

JMS in Spring Boot applications

You can use JMS in Spring Boot applications to send and receive messages by using reliable, asynchronous communication by using messaging providers such as IBM MQ.

To use JMS in your Spring Boot application, add a JMS artifact to your dependencies in your Spring Boot application to make the necessary Java libraries available. For example, in Maven,

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-jms</artifactId>
</dependency>
<dependency>
    <groupId>javax.jms</groupId>
    <artifactId>javax.jms-api</artifactId>
    <scope>provided</scope>
</dependency>
```

or in Gradle,

```
implementation("org.springframework.integration:spring-integration-jms")
compileOnly("javax.jms:javax.jms-api")
```

To send and receive messages by using a JMS messaging provider, you can define a JMS connection factory in the Liberty `server.xml` as you would if you were using JMS in a Java EE application. This connection factory can then be used to reference a remote IBM MQ queue manager by using a JmsTemplate object and either:

1. Performing a JNDI lookup of the connection factory in an @Bean annotated connectionFactory() method and returning the connection factory.
2. Naming the connection factory in the spring.jms.jndi-name in the Spring application properties. Spring Boot then creates the JmsTemplate by using the connection factory that is named in the `application.properties`.

Note: In option 2, it is also possible to configure all the connection factory attributes necessary to connect a Spring application to the required queue manager from within the `application.properties` file. The attributes are all defined in Common application properties. However, this ties the application directly the queue manager and by using JNDI is a more flexible approach.

A message driven POJO (MDP) is used to handle incoming messages in Spring Boot. An @EnableJms annotation is used in the Spring Boot Configuration class to enable discovery of methods annotated @JmsListener. The @JMSListener annotation marks a method to be the target of a JMS message listener that receives incoming messages.

If you want these MDPs to be able to use the JCICS API, then you need to bind the Liberty TaskExecutor to the JmsListenerContainerFactory. This can be achieved as follows:

```
@Bean
public TaskExecutor taskExecutor()
{
    return new DefaultManagedTaskExecutor();
}

@Bean
public JmsListenerContainerFactory<?> myFactory(ConnectionFactory connectionFactory)
{
    DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory();
    factory.setConnectionFactory(connectionFactory);
    factory.setTaskExecutor(taskExecutor());
    return factory;
}
```

Note: This requires the use of the jndi-1.0 and concurrent-1.0 Liberty features.

The Spring Boot @Transactional annotation can also be used on the @JMSListener annotated method to signify that the receiving of the message from the queue and the CICS UOW are to be coordinated by using the same container-managed JTA global transaction.

Building and deploying Spring Boot applications

You can build your Spring Boot applications for use in CICS with Maven or with Gradle.

Building Spring Boot applications as WAR files

You can build Spring Boot applications as a web application archive (WAR), which allows you to integrate Spring Framework transactional management or Spring Boot Security in CICS. See Table 19 on page 115. When built as a WAR, a Spring Boot application can be deployed and managed by using CICS bundles in the same way as other CICS Liberty applications.

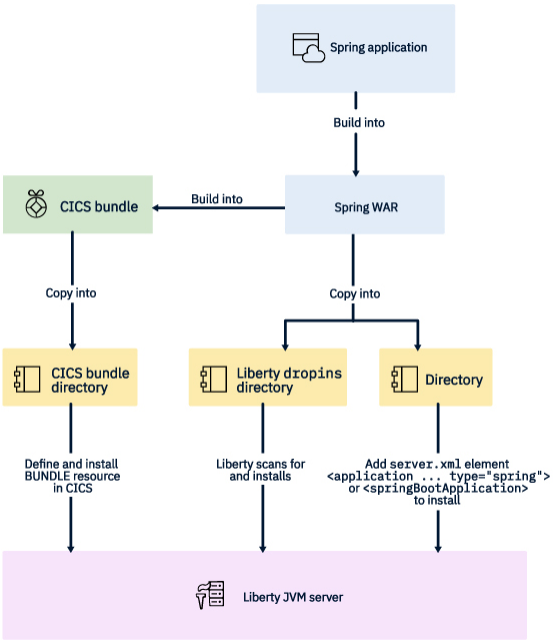
Note: Only Spring Boot applications built as WAR files are supported in CICS TS 5.3 and 5.4. JAR files are supported from CICS TS 5.5.

Table 19. Spring Boot integration

Capability	**Spring Boot applications built into:**
	WAR
CICS JCICS API	Yes

Table 19. Spring Boot integration (continued)	
Capability	**Spring Boot applications built into:**
	WAR
Java Persistance API (JPA)	Yes
Spring security integration with CICS	Yes
Spring transaction integration with CICS	Yes
Java Database Connectivity (JDBC)	Yes
Threading and concurrency	Yes
Java Message Service (JMS)	Yes

The following diagram displays the different options that you can take to run your Spring Boot application on CICS.



Before building, you must set the packaging type of your Spring Boot application as a deployable WAR. Your main application class must extend the SpringBootServletInitializer and override the configure method. You must also declare the Spring Boot embedded web container (typically Tomcat) as a provided dependency in your build script so that it can be replaced with Liberty's web-container at run time.

```
@SpringBootApplication
public class MyApplication extends SpringBootServletInitializer
{
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application)
    {
        return application.sources(MyApplication.class);
    }

    public static void main(String[] args)
    {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

For detailed information about creating a deployable WAR file with Maven or Gradle, see Create a deployable WAR file in the Spring Boot documentation.

For more information about building applications with Maven and Gradle, see Developing applications using other tools

Liberty web server plug-in

The web server plug-in allows the forwarding of HTTP requests from a supported web server, on to one or more Liberty application servers.

There are three main reasons why you would want to use the web server plug-in.

- It provides integration with a web server for the serving of static content.
- It allows termination of the SSL endpoint in the web server when using HTTPS.
- It enables load balancing and failover of HTTP requests across a group of Liberty servers.

The web server plug-in is configured by generating a `plugin-cfg.xml` file on the Liberty server that is copied to the machine hosting the web server. The plug-in takes inbound requests and checks them against the configuration data contained within this file and forwards incoming HTTP requests to the URI and host of the configured Liberty servers.

The procedure for generating `plugin-cfg.xml` with a Liberty profile server uses the `generatePluginConfig` operation, that is exposed by the `com.ibm.ws.jmx.mbeans.generatePluginConfig` MBean provided by Liberty. This JMX MBean can either be invoked remotely using the JConsole utility supplied with the IBM Java SDK in combination with the Liberty server `restConnector-1.0` feature or by developing a custom JMX application to invoke the required operation on the MBean. For further details on using JMX in a CICS Liberty server see “ Java Management Extensions API (JMX) ” on page 91.

Further detailed information on setting up a web server plug-in can be found in the WAS Knowledge Center, see Adding a plug-in configuration to a web server.

Liberty features

CICS supports features from WebSphere Application Server Liberty, which enables Java EE applications to be deployed into a Liberty JVM server.

All features in Tables 2-10 relate to CICS integrated-mode Liberty. The features are also supported in CICS standard-mode Liberty without any of the restrictions, unless noted otherwise. Table 11 provides a set of CICS features to integrate Liberty features with the CICS qualities of service.

The features from Java EE 6 and Java EE 7 must not be used concurrently. For information about editing the `server.xml`, see Server configuration.

Table 20. Liberty features alphabetically

Features A-D	Features E-Jd	Features Jm-MpJ	Features MpM-Z
appClientSupport-1.0	ejb-3.2	jms-1.1	mpMetrics-1.0
appSecurity-1.0	ejbHome-3.2	jmsMdb-3.1	mongodb-2.0
appSecurity-2.0	ejbLite-3.1	jndi-1.0	monitor-1.0
batch-1.0	ejbLite-3.2	jpa-2.0	oauth-2.0
batchManagement-1.0	ejbPersistentTimer-3.2	jpa-2.1	openidConnectClient-1.0
beanValidation-1.0 for JEE7	ejbRemote-3.2	jsf-2.0	openidConnectServer-1.0
beanValidation-1.0 for JEE6	el-3.0	jsf-2.2	osgiConsole-1.0
beanValidation-1.1 for JEE7	j2eeManagement-1.1	json-1.0	osgi.jpa-1.0

Table 20. Liberty features alphabetically (continued)

Features A-D	Features E-Jd	Features Jm-MpJ	Features MpM-Z
beanValidation-1.1 for JEE6	jacc-1.5	jsonp-1.0	restConnector-1.0
blueprint-1.0	jaspic-1.1	jsp-2.2	servlet-3.0
cdi-1.0	javaMail-1.5	jsp-2.3	servlet-3.1
cdi-1.2	javaee-7.0	jta-1.1	sessionDatabase-1.0
cicsts:core-1.0	jaxb-2.2 for JEE7	jta-1.2	ssl-1.0
cicsts:defaultApp-1.0	jaxb-2.2 for JEE6	jwt-1.0	wab-1.0
cicsts:distributedIdentity-1.0	jaxrs-1.1	ldapRegistry-3.0	wasJmsClient-1.1
cicsts:jcaLocalEci-1.0	jaxrs-2.0	localConnector-1.0	wasJmsClient-2.0
cicsts:jdbc-1.0	jaxrsClient-2.0	managedBeans-1.0	wasJmsSecurity-1.0
cicsts:link-1.0	jaxws-2.2 for JEE7	mdb-3.1	wasJmsServer-1.0
cicsts:security-1.0	jaxws-2.2 for JEE6	mdb-3.2	webCache-1.0
cicsts:standard-1.0	jca-1.6	microProfile-1.0	webProfile-6.0
cicsts:zosConnect-1.0	jca-1.7	microProfile-1.2	webProfile-7.0
cicsts:zosConnect-2.0	jcaInboundSecurity-1.0 for JEE7	mpConfig-1.1	websocket-1.0
concurrent-1.0	jcaInboundSecurity-1.0 for JEE6	mpFaultTolerance-1.0	websocket-1.1
distributedMap-1.0	jdbc-4.0	mpHealth-1.0	wmqJmsClient-2.0
	jdbc-4.1	mpJwt-1.0	zosTransaction-1.0

Table 21. Liberty features supported for Java EE 7 Web Profile

Liberty feature	Liberty feature description	Using this feature in CICS
beanValidation-1.0	Provides an annotation-based model for validating JavaBeans.	
beanValidation-1.1	Provides an annotation-based model for validating JavaBeans.	
cdi-1.2	Provides a mechanism to inject components such as EJBs or Managed Beans into other components such as JSPs or EJBs.	
ejbLite-3.2	Enables support for Enterprise JavaBeans written to the EJB Lite subset of the EJB specification.	**Important:** When using EJB-related features, the transaction attribute **`NotSupported`** is respected by the JTA Liberty transaction system but not the CICS unit of work.
el-3.0	Enables support for the Enterprise Language (EL) 3.0 specification.	

Table 21. Liberty features supported for Java EE 7 Web Profile (continued)

Liberty feature	Liberty feature description	Using this feature in CICS
jaxrs-2.0	Provides support for the Java API for RESTful Web Services (JAX-RS) in Liberty.	
jaxrsClient-2.0	Enables support for the Java Client API for JAX-RS 2.0.	The jaxrsClient-2.0 feature is enabled by jaxrs-2.0. Configuring JAX-RS 2.0 client.
jdbc-4.1	Enables the configuration of DataSources to access Databases from applications.	Creating a default CICS DB2 DataSource with type 2 connectivity for Liberty **Note:** The jdbc-4.0 and jdbc-4.1 implementations reside in the same DB2 JCC driver and are mutually exclusive.
jndi-1.0	Provides support for a single Java Naming and Directory Interface (JNDI) entry definition in the server configuration of Liberty.	
jpa-2.1	Enables support for applications that use application-managed and container-managed JPA.	**Note:** If you are using jpa-2.1 and you have applied CICS APAR PI67640, which supplies fixpack 16.0.0.4 for Liberty, you might see the following warning message: `CWWJP9991W Exception Description: Server platform class is not valid: null` This message can be ignored.
jsf-2.2	Provides support for web applications that use the JavaServer Faces (JSF) framework.	
jsonp-1.0	Supports the definition of a Java API to process JavaScript Object Notation. Including the support for the JSON parse, generate, transform, and query function.	
jsp-2.3	Enables support for servlet and JavaServer Pages (JSP) applications.	"Java EE and Liberty applications" on page 69
managedBeans-1.0	Provides a common foundation for different Java EE components types that are managed by a container. Common services provided to Managed Beans include resource injection, lifecycle management and the use of interceptors.	

Table 21. Liberty features supported for Java EE 7 Web Profile (continued)

Liberty feature	Liberty feature description	Using this feature in CICS
servlet-3.1	Provides support for HTTP Servlets written to the Java Servlet specification.	"Java EE and Liberty applications" on page 69 **Restriction:** The use of servlet login and logout API does not propagate the user ID to the CICS task.
webProfile-7.0	Provides a convenient combination of the Liberty features that are required to support the Java EE 7 Web Profile.	
websocket-1.0	Enables a web browser or client application and a web server application to communicate by using one full duplex connection.	
websocket-1.1	Enables a web browser or client application and a web server application to communicate by using one full duplex connection.	

Table 22. Liberty features supported for Java EE 7 Full Platform

Liberty feature	Liberty feature description	Using this feature in CICS
appClientSupport-1.0	Enables the Liberty server to process client modules and support remote client containers.	**Tip:** The Application Client module runs in both the client and the server. The client executes the client specific logic of the application. The other portion of code runs in a client container on the server and communicates data from the business logic running on the server to the client. For more information, see Preparing and running an application client.
batch-1.0	Enables support for the Java Batch 1.0 API defined in JSR-352. This feature does not support Java batch applications that are packaged in an Enterprise Bundle Archive (EBA).	
concurrent-1.0	Enables managed executors to be created, which then permit applications to submit tasks that can run concurrently.	**Restriction:** The transaction property `ManagedTask.SUSPEND` is not supported by a Liberty JVM server. **Restriction:** The user ID that is attached to the transaction of a new thread is always the user ID that is attached to the parent transaction. **Restriction:** Use of a `ManagedThreadFactory` creates standard Java threads, not CICS-enabled Java threads.
	Enables support for Enterprise JavaBeans written to the EJB 3.2 specification.	Enterprise JavaBeans (EJB) **Important:** When using EJB-related features, the transaction attribute **`NotSupported`** is respected by the JTA Liberty transaction system but not the CICS unit of work.
ejbHome-3.2	Provides support for the EJB 2.x APIs.	

Table 22. Liberty features supported for Java EE 7 Full Platform (continued)

Liberty feature	Liberty feature description	Using this feature in CICS
ejbPersistentTimer-3.2	Provides support for persistent EJB timers.	**Restriction:** DB2 JDBC type 2 connectivity is not supported for persisting EJB timers.
ejbRemote-3.2	Provides support for remote EJB interfaces.	
jacc-1.5	Enables support for Java Authorization Contract for Containers (JACC) version 1.5.	Developing a Java Authorization Contract for Containers (JACC) Authorization Provider
jaspic-1.1	Java Authentication SPI for Containers (JASPIC) allows a Java EE Application Server to use custom authentication. JASPIC providers are defined in JSR-196. If a JASPIC provider and a TAI are configured in the same server, then the TAI has no effect. Therefore, JASPIC is a standard Java EE technology and a more portable solution than a TAI for Java EE applications.	
j2eeManagement-1.1	Provides a set of interfaces to manage and monitor applications within the JEE application server.	
javaMail-1.5	Enables applications to use the JavaMail 1.5 API.	
javaee-7.0	Combines the Liberty features that support the Java EE 7.0 Full Platform.	
jaxb-2.2	Provides support to map between Java classes and XML representations.	
jaxws-2.2	Provides support for SOAP web services.	
jca-1.7	Enables the configuration of resource adapters to access Enterprise Information Systems (EIS) from applications.	"Java EE Connector Architecture (JCA)" on page 95 **Restriction:** The use of JCICS API and DB2 JDBC type 2 connectivity capabilities is not supported in threads that are created by the JCA API `javax.resource.spi.BootstrapContext.createTimer()`. For the same effect, use the concurrent APIs (`javax.enterprise.concurrent.ManagedScheduledExecutorService`).
jcaInboundSecurity-1.0	Allows JCA inbound resource adapters to flow security contexts by extending the `javax.resource.spi.work.SecurityContext` abstract class.	

Table 22. Liberty features supported for Java EE 7 Full Platform (continued)		
Liberty feature	**Liberty feature description**	**Using this feature in CICS**
mdb-3.2	Enables the use of Message-Driven Enterprise JavaBeans written to the EJB 3.2 specification. MDBs allow asynchronous processing of messages within a Java EE component.	
wasJmsClient-2.0	Provides applications with access to message queues hosted in Liberty through the JMS API.	"Java Message Service (JMS)" on page 90
wasJmsSecurity-1.0	Enables an embedded messaging server to authenticate and authorize access from clients.	"Java Message Service (JMS)" on page 90
wasJmsServer-1.0	Enables an embedded messaging server in the server. Applications can operate on messages by using the `wasJmsClient` feature.	"Java Message Service (JMS)" on page 90

Table 23. Liberty features supported for Extended Programming Models		
Liberty feature	**Liberty feature description**	**Using this feature in CICS**
json-1.0	Provides access to the JavaScript Object Notation (JSON4J) library that provides a set of JSON handling classes for Java environments.	
jta-1.1	Supports the Java Transaction API (JTA). **Note:** Java Transaction API is a protected Liberty feature.	" Java Transaction API (JTA) " on page 80
jta-1.2	Supports the Java Transaction API (JTA). **Note:** Java Transaction API is a protected Liberty feature.	" Java Transaction API (JTA) " on page 80
mongodb-2.0	Provides support for the MongoDB Java Driver and allows remote database instances to be configured in the server configuration. Applications interact with these databases through the MongoDB APIs.	

Table 24. Liberty features supported for Enterprise OSGi

Liberty feature	Liberty feature description	Using this feature in CICS
blueprint-1.0	Enables support for deploying OSGi applications that use the OSGi blueprint container specification.	**Restriction:** The transaction attribute `NotSupported` is not supported by a Liberty JVM server.
osgi.jpa-1.0	This feature is superseded by the `blueprint-1.0` and `jpa-2.0` features that both include OSGi capability. When those features are both added to the server, this feature is added automatically.	
wab-1.0	Provides support for web application bundles (WAB) that are inside enterprise bundles (EBA).	"Creating an OSGi Application Project " on page 70 **Note:** The `wab-1.0` feature is required by CICS and installed as part of `cicsts:core-1.0` and `cicsts:standard-1.0`.

Table 25. Liberty features supported for MicroProfile

Liberty feature	Liberty feature description	Using this feature in CICS
microProfile-1.0	Combines the Liberty features that support the Micro Profile for enterprise Java.	
microProfile-1.2	Combines the Liberty features that support Micro Profile 1.2 for enterprise Java.	Java 8 is required for this feature.
mpConfig-1.1	Provides a unified mechanism to access configuration, providing a single view of multiple sources.	Java 8 is required for this feature.
mpFaultTolerance-1.0	Provides support for the MicroProfile Fault Tolerance API for enterprise Java.	Java 8 is required for this feature. **Restriction:** MicroProfile Fault Tolerance 1.0 is not designed to work with transactions (UOW, JTA, etc.). Updates to CICS resources should not be made in methods annotated @Bulkhead, @CircuitBreaker, @Fallback, @Retry or @Timeout.
mpHealth-1.0	Provides support for the MicroProfile Health API for enterprise Java.	Java 8 is required for this feature.

Table 25. Liberty features supported for MicroProfile (continued)		
Liberty feature	**Liberty feature description**	**Using this feature in CICS**
mpJwt-1.0	Enables web applications or microservices to use JSON Web Token (JWT) to authenticate users instead of, or in addition to, the configured user registry.	Java 8 is required for this feature. The default value for attribute `ignoreApplicationAuthMethod` is false. This indicates all requests received by Liberty need to have a JWT token in the HTTP header. The default value for attribute `mapToUserRegistry` is false. For integration with CICS security set this value to true.
mpMetrics-1.0	Provides support for the MicroProfile Metrics API for enterprise Java.	Java 8 is required for this feature.

Table 26. Liberty features supported for Operations		
Liberty feature	**Liberty feature description**	**Using this feature in CICS**
batchManagement-1.0	Provides managed batch support for the Java batch container. This includes the Batch REST management interface, job logging support, and a command line utility for external scheduler integration.	
distributedMap-1.0	Provides a local cache service, which can be accessed through the `DistributedMap` API.	
localConnector-1.0	Allows the use of a local JMX connector that is built into the JVM to access JMX resources in the server.	“ Java Management Extensions API (JMX) ” on page 91
monitor-1.0	Enables performance monitoring of Liberty runtime components by using a JMX client.	“ Java Management Extensions API (JMX) ” on page 91
osgiConsole-1.0	Enables an OSGi console to aid with debug of the runtime.	Troubleshooting Java applications
restConnector-1.0	Enables remote access by JMX clients through a REST-based connector and requires SSL and user security configuration.	“ Java Management Extensions API (JMX) ” on page 91
sessionDatabase-1.0	Enables persistence of HTTP sessions to a datasource that uses JDBC.	

Table 26. Liberty features supported for Operations (continued)		
Liberty feature	**Liberty feature description**	**Using this feature in CICS**
webCache-1.0	Enables local caching for web responses. It includes the `distributedMap` feature and performs automatic caching of web application responses to improve response times and throughput.	
wmqJmsClient-2.0	Provides applications with access to message queues hosted on IBM MQ through the `JMS 2.0` API.	**Restriction:** Only supported when the JMS application connects to IBM MQ using the client mode transport. Requires V9.0.1 of the IBM MQ Resource Adapter for Liberty. **Important:** This restriction also applies to CICS standard-mode Liberty.

Table 27. Liberty features supported for Security		
Liberty feature	**Liberty feature description**	**Using this feature in CICS**
appSecurity-1.0	Provides support for securing the server runtime environment and applications. `appSecurity-2.0` supercedes `appSecurity-1.0`.	Configuring security for a Liberty JVM server
appSecurity-2.0	Provides support for securing the server runtime environment and applications. `appSecurity-2.0` supersedes `appSecurity-1.0`.	Configuring security for a Liberty JVM server
jwt-1.0	Allows runtime to create JWT tokens.	
ldapRegistry-3.0	Enables support for using an LDAP server as a user registry. Any server that supports LDAP Version 3.0 can be used. Multiple LDAP registries can be configured, and then federated to achieve a single logical registry view.	Configuring security for a Liberty JVM server by using distributed identity mapping
oauth-2.0	Enables web applications to integrate OAuth 2.0 for authenticating and authorizing users.	Authorization using OAuth 2.0 Configuring persistent OAuth 2.0 services
openidConnectClient-1.0	Enables web applications to integrate OpenID Connect Client 1.0 for authenticating users instead of, or in addition to, the configured user registry.	

Table 27. Liberty features supported for Security (continued)		
Liberty feature	**Liberty feature description**	**Using this feature in CICS**
openidConnectServer-1.0	Enables web applications to integrate OpenID Connect Server 1.0 for authenticating users instead of, or in addition to, the configured user registry.	
ssl-1.0	Provides support for Secure Sockets Layer (SSL) connections and SAF keyrings.	Configuring SSL (TLS) for a Liberty JVM server using RACF Setting up SSL (TLS) client certificate authentication in a Liberty JVM server Configuring SSL (TLS) for a Liberty JVM server using a Java keystore

Table 28. Liberty features supported for z/OS		
Liberty feature	**Liberty feature description**	**Using this feature in CICS**
zosTransaction-1.0	Enables Liberty to synchronize and manage transactional activity between the z/OS Resource Recovery Services (RRS), the transaction manager of the application server, and the resource manager.	**Restriction:** `zosTransaction-1.0` is only supported for JMS applications that connect to IBM MQ using BINDINGS mode transport in CICS standard-mode Liberty.

Table 29. Liberty features supported for Java EE 6 Web Profile		
Liberty feature	**Liberty feature description**	**Using this feature in CICS**
beanValidation-1.0	Provides an annotation-based model for validating JavaBeans.	
beanValidation-1.1	Provides an annotation-based model for validating JavaBeans.	
cdi-1.0	Provides a mechanism to inject components such as EJBs or Managed Beans into other components such as JSPs or EJBs.	
ejbLite-3.1	Enables support for Enterprise JavaBeans written to the EJB Lite subset of the EJB specification.	**Restriction:** The transaction attribute `NotSupported` is not supported by a Liberty JVM server.
jndi-1.0	Provides support for a single Java Naming and Directory Interface (JNDI) entry definition in the server configuration of Liberty.	
jpa-2.0	Enables support for applications that use application-managed and container-managed JPA.	

Table 29. Liberty features supported for Java EE 6 Web Profile (continued)		
Liberty feature	**Liberty feature description**	**Using this feature in CICS**
jsf-2.0	Provides support for web applications that use the JavaServer Faces (JSF) framework.	
jsp-2.2	Enables support for servlet and JavaServer Pages (JSP) applications.	"Java EE and Liberty applications" on page 69
servlet-3.0	Provides support for HTTP Servlets written to the Java Servlet specification.	"Java EE and Liberty applications" on page 69 **Restriction:** The use of servlet login and logout API does not propagate the user ID to the CICS task.
webProfile-6.0	Provides a convenient combination of the Liberty features that are required to support the Java EE 6 Web Profile.	

Table 30. Liberty features supported for Java EE 6 Technologies		
Liberty feature	**Liberty feature description**	**Using this feature in CICS**
jaxb-2.2	Provides support to map between Java classes and XML representations.	
jaxrs-1.1	Provides support for the Java API for RESTful Web Services (JAX-RS) in Liberty.	
jaxws-2.2	Provides support for SOAP web services.	
jca-1.6	Enables the configuration of resource adapters to access Enterprise Information Systems (EIS) from applications.	"Java EE Connector Architecture (JCA)" on page 95 **Restriction:** The use of JCICS API and DB2 JDBC type 2 connectivity capabilities are not supported within threads that are created by the JCA API `javax.resource.spi.BootstrapContext.createTimer()`. Instead, for the same effect, use the concurrent APIs (`javax.enterprise.concurrent.ManagedScheduledExecutorService`).
jcaInboundSecurity-1.0	Allows JCA inbound resource adapters to flow security contexts by extending the `javax.resource.spi.work.SecurityContext` abstract class.	

Table 30. Liberty features supported for Java EE 6 Technologies (continued)		
Liberty feature	**Liberty feature description**	**Using this feature in CICS**
jdbc-4.0	Enables the configuration of DataSources to access Databases from applications.	Creating a default CICS DB2 DataSource with type 2 connectivity for Liberty **Note:** The `jdbc-4.0` and `jdbc-4.1` implementations reside in the same DB2 JCC driver and are mutually exclusive.
jms-1.1	Enables the configuration of resource adapters to access messaging systems using the Java Message Service API.	"Java Message Service (JMS)" on page 90
jmsMdb-3.1	Enables the use of JMS Message-Driven Enterprise JavaBeans. MDBs allow asynchronous processing of messages within a Java EE component.	
mdb-3.1	Enables the use of Message-Driven Enterprise JavaBeans. MDBs allow asynchronous processing of messages within a Java EE component.	
wasJmsClient-1.1	Provides applications with access to message queues hosted in Liberty through the JMS API.	Java Message Service (JMS)
wasJmsSecurity-1.0	Enables an embedded messaging server to authenticate and authorize access from clients.	Java Message Service (JMS)
wasJmsServer-1.0	Enables an embedded messaging server in the server. Applications can operate on messages by using the `wasJmsClient` feature.	Java Message Service (JMS)

For more information about the function in these features, see the documentation for Liberty at Liberty overview. For details of Liberty restrictions, see Runtime environment known restrictions.

The following table provides a set of CICS features to integrate Liberty features with the CICS qualities of service. The Liberty JVM server mode can be set by specifying CICS_WLP_MODE in the JVM profile.

Table 31. CICS Liberty features

CICS Feature	CICS Liberty mode	Description	Using this CICS feature
cicsts:core-1.0	Integrated-mode	Provides core CICS features, Java Transaction API (JTA) 1.0 and `wab-1.0`.	This feature is required when using Integrated-mode CICS Liberty. **Restriction:** The JVM server should be disabled before adding or removing this feature.
cicsts:defaultApp-1.0	Integrated-mode and standard-mode	Verifies that the Liberty server is running and provides information on the server configuration. Browse the JVM Profile, the JVM server logs, the Liberty server.xml, and the messages log by using the FileViewer servlet.	Configuring the CICS Default Web Application
cicsts:distributedIdentity-1.0	Integrated-mode and standard-mode	Provides support for distributed identity mapping.	Configuring security for a Liberty JVM server by using distributed identity mapping
cicsts:jcaLocalEci-1.0	Integrated-mode	Provides a locally optimized JCA ECI resource adapter for calling CICS programs.	"Using the JCA local ECI resource adapter" on page 96 **Restriction:** The JVM server should be disabled before adding or removing this feature.
cicsts:jdbc-1.0	Integrated-mode and standard-mode	Provides support for applications to access a local CICS DB2 database that uses JDBC. This feature has been superseded by jdbc-4.0 and jdbc-4.1, except when used directly with DriverManager.	Acquiring a connection to a database **Restriction:** The JVM server should be disabled before adding or removing this feature.
cicsts:link-1.0	Integrated-mode	Provides support to start a Java EE application that is running in a Liberty JVM server either as the initial program of a CICS transaction or by using the **LINK**, **START**, or **START CHANNEL** commands from any CICS program.	"Linking to a Java EE application from a CICS program" on page 75

Table 31. CICS Liberty features (continued)

CICS Feature	CICS Liberty mode	Description	Using this CICS feature
cicsts:security-1.0	Integrated-mode and standard-mode	Provides integration of Liberty security with CICS security, including propagation of thread identity.	Configuring security for a Liberty JVM server **Restriction:** The JVM server should be disabled before adding or removing this feature.
cicsts:standard-1.0	Standard-mode	Enables users to port and deploy Liberty applications from other platforms to CICS without changing your application. Standard mode is ideal for hosting applications that are written for and rely on the Java EE Full Platform, but do not require full integration with CICS. This feature includes `wab-1.0` which is required by CICS.	CICS standard-mode Liberty: Java EE 7 Full Platform support without full CICS integration
cicsts:zosConnect-1.0	Integrated-mode	Integrates z/OS Connect with CICS Liberty JVM server.	Configuring z/OS Connect EE **Restriction:** The JVM server should be disabled before adding or removing this feature.
cicsts:zosConnect-2.0	Integrated-mode	Integrates z/OS Connect with CICS Liberty JVM server.	Configuring z/OS Connect EE **Restriction:** The JVM server should be disabled before adding or removing this feature.

Accessing data from Java applications

You can write Java applications that can access and update data in DB2 and VSAM. Alternatively, you can link to programs in other languages to access DB2, VSAM, and IMS.

You can use any of the following techniques when writing a Java application to access data in CICS. The CICS recovery manager maintains data integrity.

Accessing relational data

You can write a Java application to access relational data in DB2 using any of the following methods:

- A JCICS **`LINK`** command to link to a program that uses Structured Query Language (SQL) commands to access the data.
- Where a suitable driver is available, use Java Data Base Connectivity (JDBC) or Structured Query Language for Java (SQLJ) calls to access the data directly. Suitable JDBC drivers are available for DB2. For more information about using JDBC and SQLJ application programming interfaces, see Using JDBC and SQLJ to access DB2 data from Java programs .

- JavaBeans that use JDBC or SQLJ as the underlying access mechanism. You can use any suitable Java integrated development environment (IDE) to develop such JavaBeans.

Accessing DL/I data

To access DL/I data in IMS , your Java application must use a JCICS **LINK** command to link to an intermediate program that issues EXEC DLI commands to access the data.

Accessing VSAM data

To access VSAM data, a Java application can use either of the following methods:

- The JCICS file control classes to access VSAM directly.
- A JCICS **LINK** command to link to a program that issues CICS file control commands to access the data.

Interacting with structured data from Java

CICS Java programs often interact with data that was originally designed for use with other programming languages. For example, a Java program might link to a COBOL program by using a COMMAREA defined in a COBOL copybook, or read a record from a VSAM file where the data is defined by using an assembler language DSECT.

Importing structured data into Java

You can use an importer to generate Java classes that facilitate the interaction with structured record data from other languages. The importers map the data types that are contained in the language structure source so that your Java application can easily set and get individual fields in the underlying record structure.

You can use IBM Record Generator for Java or the Rational Java EE Connector (J2C) Tools to interact with data to produce a Java class so that you can pass data between Java and other programs in CICS.

IBM Record Generator for Java V3.0.0

IBM Record Generator for Java is a stand-alone utility that generates Java helper classes based on the associated-data (ADATA) files that are produced from compiling COBOL copybooks or assembler DSECTs. These Java helper classes can then be used in a Java application to marshal data to and from the COBOL-specific or assembler language-specific record structures.

For more information, see IBM Record Generator for Java V3.0.0.

Rational J2C Tools

The Rational J2C Tools, resource adapters, and file importers enable you to create J2C artifacts that you can use to create enterprise applications that connect to enterprise information systems such as CICS. To use the Rational J2C Tools, you require Rational Application Developer for WebSphere Software or IBM Developer for z Systems.

The J2C Tools CICS/IMS Data Binding wizard generates Java classes that map to COBOL, PL/I, or C application program data structures, by using a customizable Eclipse based wizard. These helper classes can then be used in a Java application to marshal data to and from the language-specific record structures.

For more information, see Connecting to enterprise information systems in Rational Application Developer for WebSphere Software product documentation.

Related information

IBM Redbooks: IBM CICS and the JVM server: Developing and Deploying Java Applications

Building Java records from COBOL with the IBM Record Generator for Java

COBOL Importer overview in Rational Application Developer for WebSphere Software product documentation

Generating Java Records from COBOL with Rational J2C Tools

Developing Java applications to use the JZOS Toolkit API in an OSGi JVM server

The IBM JZOS Toolkit consists of classes in package `com.ibm.jzos`, which is distributed with the IBM Java SDKs for z/OS in a single JAR file `ibmjzos.jar`.

These classes give Java applications on z/OS direct access to traditional z/OS data sets and files and access to z/OS system services and converter classes for mapping byte array fields into Java data types.

Before you begin

If the JZOS Toolkit API is not downloaded to your workstation, then transfer the `ibmjzos.jar` file from the relevant version of the IBM Java SDK on z/OS to your workstation.

Procedure

1. Set the Target platform. To prepare to use the JCICS API in your development environment, set the Eclipse Target Platform to ensure it can resolve locally. In an OSGi development environment Target Platform definitions are used to define the plug-ins that applications in the workspace is built against. For CICS Explorer use the Eclipse menu, **Windows** > **Preferences** > **Plug-in Development** > **Target Platform**. Click **Add**, and from the supplied templates select the CICS TS release for your runtime environment. Don't forget to apply the target platform to your workspace.
2. Create an OSGi wrapper bundle for the JZOS Toolkit. If you have the IBM CICS SDK for Java EE and Liberty plug-in, then select **File** > **Import** > **Java Archive into an OSGi bundle** to create a new OSGi Bundle Project. Ensure that the newly created bundle exports all the available JZOS Toolkit packages that are required by the Java application such as `com.ibm.jzos`, `com.ibm.jzos.fields`, or `com.ibm.jzos.wlm`. This ensures that these packages are available to be imported by other OSGi projects in the Eclipse workspace.

   For example

   ```
   Export-Package: com.ibm.jzos,
     com.ibm.jzos.fields
   ```

3. Create a CICS Java application.
   a) Create an OSGi Bundle Project in Eclipse by using the wizard **File** > **New** > **Other Plug-in Project**.
   b) Create a Java package `com.ibm.cicsdev.jzos.sample` and add a class ZFilePrint.
   c) Copy in the following code example, which opens an MVS data set pointed to by the `//INPUT DD` and writes the output to a CICS temporary storage queue.

   ```
   package com.ibm.cicsdev.jzos.sample;

   import com.ibm.jzos.ZFile;
   import com.ibm.jzos.ZUtil;
   import com.ibm.cics.server.TSQ;

   public class ZFilePrint
   {
       public static void main(String[] args) throws Exception
       {
           ZFile zFile = new ZFile("//DD:INPUT", "rb,type=record,noseek");
           TSQ tsqQ = new TSQ();
           tsqQ.setName("JZOSTSQ");

           try
           {
               byte[] recBuf = new byte[zFile.getLrecl()];
               int nRead;
               String encoding = ZUtil.getDefaultPlatformEncoding();
   ```

```
            while ((nRead = zFile.read(recBuf)) >= 0)
            {
                String line = new String(recBuf, 0, nRead, encoding);
                tsqQ.writeString(line);
            }
        }
        finally
        {
            zFile.close();
        }
    }
}
```

4. Add the following Import-Package statements to the bundle manifest for the JCICS and JZOS packages. The JCICS import should follow best practice to specify a range of versions the application operates with. Typically this range will go up to, but not include, the next API breaking change. For JCICS that would be version 2.0.0, so the range `com.ibm.cics.server;version="[1.401.0,2.0.0)"` is used in the example as this is the minimum level that is required to support the JCICS `TSQ.writeString()` method. The JZOS package is not taken from a versioned bundle. It is displayed to the runtime from the underlying JAR file with no version, and so `com.ibm.jzos` can be listed without a referenced version, which allows any available version (including 0.0.0) to be chosen.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: com.ibm.cicsdev.jzos.sample
Bundle-SymbolicName: com.ibm.cicsdev.jzos.sample
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Import-Package: com.ibm.cics.server;version="[1.401.0,2.0.0)",
 com.ibm.jzos
```

5. Add a CICS-MainClass: definition to the bundle manifest to register a **MainClass** service for your `com.ibm.cicsdev.jzos.sample.ZFilePrint` class.

   This allows the Java class to be linked to using a CICS program definition. Your manifest now looks similar to the following example:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: com.ibm.cicsdev.jzos.sample
Bundle-SymbolicName: com.ibm.cicsdev.jzos.sample
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Import-Package: com.ibm.cics.server;version="[1.401.0,2.0.0)",
 com.ibm.jzos
CICS-MainClass: com.ibm.cicsdev.jzos.sample.ZFilePrint
```

Results

The application is now ready to be tested, and can be deployed into a CICS OSGi JVM server by using a CICS Bundle Project as follows:

1. Create a CICS Bundle Project in Eclipse and add the OSGi Bundle Project by using the menu **New OSGi Bundle Project Include**.
2. Deploy to zFS by using the menu **Export Bundle Project to z/OS UNIX file system**.
3. Create a CICS BUNDLE definition that references this zFS location and install it.
4. Create a CICS PROGRAM definition that names the CICS-MainClass: `com.ibm.cicsdev.jzos.sample.ZFilePrint` in the `JVMClass` attribute and install it.
5. Before you run the application, you need to define an MVS DD in the CICS JCL referencing a valid MVS data set and then restart your CICS region. For instance

```
//INPUT    DD DISP=SHR,DSN=CICS.USER.INPUT
```

6. If you need to run the application from a 3270 console, create a TRANSACTION definition that references the PROGRAM defined in step 4.

When started, your Java class `ZFilePrint` reads the defined MVS data set by using the JZOS Toolkit API and then write the contents to a CICS temporary storage queue using the JCICS API.

Accessing IBM MQ from Java programs

Java programs that run in CICS can use either the IBM MQ classes for Java, or the IBM MQ classes for JMS, to access IBM MQ.

IBM MQ classes for Java encapsulate the Message Queue Interface (MQI), the native IBM MQ API. If you are familiar with the use of the MQI in procedural languages, you can transfer this knowledge to the Java environment. The classes use a similar object model to the C++ and .NET interfaces to IBM MQ. In addition, you can exploit the full range of features of IBM MQ , beyond those available through JMS.

IBM MQ classes for JMS implements the JMS interfaces for IBM MQ as the messaging system. If your organization is new to IBM MQ , but already has JMS application development skills, you might find it easier to use the familiar JMS API to access IBM MQ resources, rather than one of the other APIs provided with IBM MQ.

The IBM MQ classes for Java can only be used in an OSGi JVM server. The IBM MQ classes for JMS can be used in:

- An OSGI JVM server.
- A CICS standard-mode Liberty JVM server when the JMS application connects to a queue manager using either bindings or client mode transport.
- A CICS integrated-mode Liberty JVM server when the JMS application connects to a queue manager using client mode transport.

Using IBM MQ classes for Java in an OSGi JVM server

Java programs running in an OSGi JVM server can use the IBM MQ classes for Java to access IBM MQ.

For an introduction to the IBM MQ classes for Java, see Using WebSphere MQ classes for Java.

Using IBM MQ classes for Java work in the CICS OSGi IBM MQ environment

The IBM MQ classes for Java provided by the IBM MQ product provide a Java variant of the Message Queue Interface (MQI) for use by CICS Java applications.

In a CICS environment, the classes supplied by IBM MQ allow only connections in bindings mode. Any attempt to use connections to a remote queue manager in client mode results in an exception.

In bindings mode, the call request is transformed into an IBM MQ MQI call, and is processed as normal by the existing CICS-MQ adapter. The converted requests flow into the CICS-MQ adapter in exactly the same way as MQI requests from any other program (for example, a COBOL program). So there are no operational differences between Java programs and other programs accessing IBM MQ.

For information about the connection options, see Connection options for WebSphere MQ classes for Java.

Configuring an OSGi JVM server to support IBM MQ classes for Java

A JVM server is the runtime environment for Java applications. You can configure an OSGi JVM server to support applications that use IBM MQ classes for Java.

About this task

To enable the OSGi JVM server to support applications that use IBM MQ classes for Java, IBM MQ for Java bundles need to be added to the set of middleware bundles that run in the OSGi framework within the JVM server. The framework must also have access to the associated set of native libraries.

Procedure

1. Add the IBM MQ classes for Java to the JVM server as an OSGi middleware bundle.

   To add the classes, from IBM MQ Version 8.0, include the following lines in the JVM profile for the OSGi JVM server:

   ```
   OSGI_BUNDLES=<MQ_ROOT>/OSGi/com.ibm.mq.osgi.allclientprereqs_<VERSION>.jar,\
   <MQ_ROOT>/OSGi/com.ibm.mq.osgi.allclient_<VERSION>.jar
   ```

   For WebSphere MQ for z/OS Version 7.1, include the following line:

   ```
   OSGI_BUNDLES=<MQ_ROOT>/OSGi/com.ibm.mq.osgi.java_<VERSION>.jar
   ```

   where:

   - *MQ_ROOT* is the `java/lib/` directory of the IBM MQ for z/OS Unix System Services installation, for example, `/usr/lpp/V8R0M0/java/lib`.
   - *VERSION* is the version of the IBM MQ classes for Java that you are using, for example, `8.0.0.0`.
2. Add the directory containing the IBM MQ classes for Java native libraries to the LIBPATH_SUFFIX option in the JVM profile for the OSGi JVM server.

   For example:

   ```
   LIBPATH_SUFFIX=<MQ_ROOT>
   ```

   where *MQ_ROOT* is the `java/lib/` directory of the IBM MQ for z/OS Unix System Services installation, for example, `/usr/lpp/V8R0M0/java/lib`.

Programming with IBM MQ classes for Java in the CICS IBM MQ environment

IBM MQ classes for Java encapsulates the Message Queue Interface (MQI), the native IBM MQ API, and uses the same object model as other object-oriented interfaces. You can exploit the full range of features of IBM MQ, not all of which are available in IBM MQ classes for JMS.

Support for use of IBM MQ classes for Java in CICS applications is provided from WebSphere MQ for z/OS 7.1.

For more information about the IBM MQ classes for Java, see Using WebSphere MQ classes for Java.

Committing a unit of work involving WebSphere MQ requests

Messages sent and received by the IBM MQ classes for Java in a CICS JVM server environment are always associated with the CICS unit of work (UOW).

The UOW can only be completed by calling the commit or rollback methods on the `com.ibm.cics.server.Task` object, or by the CICS task ending normally, in which case the UOW is implicitly committed. Use of the transaction control methods on MQQueueManager is not supported.

For a mixed language application, an **`EXEC CICS SYNCPOINT`** command issued from a non- Java program will commit the whole unit of work, including the updates made to IBM MQ by a Java program.

CICS abends during the processing of IBM MQ requests

The use of IBM MQ classes for Java results in the issuing of a IBM MQ MQI command. CICS abends issued during processing of the MQI command are not converted into Java exceptions, and therefore are not catchable by a CICS Java application.

In this situation, the CICS transaction will abend and roll back to the last syncpoint.

Using IBM MQ classes for JMS in a Liberty JVM server

Java programs running in a Liberty JVM server can use the IBM MQ classes for JMS to access IBM MQ.

For an introduction to the IBM MQ classes for Java, see Using IBM MQ classes for JMS.

Using IBM MQ classes for JMS work in the CICS Liberty IBM MQ environment

When a CICS Java application makes JMS requests, the requests are processed by the MQ messaging provider.

In a CICS environment, the IBM MQ classes for JMS allow connections to be made as follows:

- In a CICS standard-mode Liberty JVM server JMS applications can connect to a queue manager using either bindings or client mode transports.
- In a CICS integrated-mode Liberty JVM server JMS applications can only connect to a queue manager using client mode MQ transport. The use of bindings mode is not supported.

When using a bindings mode transport:

- Any work submitted to the `CICSExecutorService` using the `runAsCICS()` method that work must not include any JMS requests.
- The queue manager used must be different to the queue manager specified on any CICS MQCONN resource installed in the CICS region hosting the Liberty JVM server.

To use a specific level of JMS, CICS must be connected to an IBM MQ queue manager that supports the appropriate level of JMS. For more information, see WebSphere Application Server Liberty and the IBM MQ resource adapter.

Configuring a Liberty JVM server to support IBM MQ classes for JMS

A JVM server is the runtime environment for Java applications. You can configure a CICS Liberty JVM server to support applications that use IBM MQ classes for JMS.

Procedure

1. Add the `wmqJmsClient-2.0` feature to the `server.xml` file.

   If you want to perform a JNDI lookup, then you must also add the jndi-1.0 feature.

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

   Adding the `wmqJmsClient-2.0` feature enables the Liberty server to load the necessary IBM MQ bundles that lets you define the IBM MQ JMS resources. For example, the connection factory and activation specification properties provide client libraries to connect to the IBM MQ network.
2. If you are configuring JMS applications to connect in the BINDINGS mode, then add the `zosTransaction-1.0` feature:

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

3. Specify the location of the IBM MQ Resource Adapter by adding the following entry to the `server.xml` file:

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

   The value attribute specifies the absolute path to the IBM MQ Resource Adapter file, `wmq.jmsra.rar` . Obtain the `wmq.jmsra.rar` file and install it from Fix Central.
4. Add the connection factory definitions to the `server.xml` file.
   - For a CLIENT mode connection, add the following elements:

     ```
     <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="2"/>
```

- For a BINDINGS mode connection, add the following elements:

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

<connectionManager id="ConMgr6" maxPoolSize="2"/>
```

5. Add the queue definitions to the `server.xml`:

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

6. If you are configuring JMS applications to connect in the BINDINGS mode, use the `wmqJmsClient` element in the `server.xml` file to specify the location of the IBM MQ native libraries.

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

   To allow the JMS applications to connect by using BINDINGS mode to IBM MQ , you must have both Liberty and IBM MQ deployed on the same server.

7. If you are using message-driven beans add the **`mdb-3.2`** feature to `server.xml`:

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

- For a CLIENT mode connection, add a `jmsActivationSpec` element as follows:

```
<jmsActivationSpec id="MQ.JMS.mdb.app/MQ.JMS.mdbEJB/MessageDrivenBean">
    <properties.wmqJms transportType="CLIENT"
        destinationRef="jms/queue1" destinationType="javax.jms.Queue"
        hostName="localhost" port="1414"
        channel="SYSTEM.DEF.SVRCONN" queueManager="qm1"/>
</jmsActivationSpec>
```

- For a BINDINGS mode connection, add a `jmsActivationSpec` element as follows:

```
<jmsActivationSpec id="MQ.JMS.mdb.app/MQ.JMS.mdbEJB/MessageDrivenBean">
    <properties.wmqJms transportType="BINDINGS"
        destinationRef="jms/queue1" destinationType="javax.jms.Queue"
        queueManager="qm1"/>
</jmsActivationSpec>
```

Results

You have configured a CICS Liberty JVM server to support applications that use IBM MQ classes for JMS.

Programming with IBM MQ classes for JMS in the CICS IBM MQ environment

CICS Java programs may invoke various functions provided by the IBM MQ classes for JMS. The IBM MQ classes for JMS form part of the JMS-compliant IBM MQ offering.

From the perspective of the JMS specification, the IBM MQ classes for JMS treat a CICS Liberty JVM server as a Java EE compliant application server, with no JTA transaction in progress. In an integrated-mode Liberty server, the Java Transaction API (JTA) UserTransaction interface should be used to ensure messages sent and received by the IBM MQ classes for JMS are associated with the CICS UOW.

Support is provided for using the classic (JMS 1.1) and simplified (JMS 2.0) interfaces, provided that CICS is connected to a level of IBM MQ queue manager that supports the appropriate level of JMS and is using a suitable version of the IBM MQ classes for JMS.

Note: WebSphere MQ for z/OS version 7.1 only supports JMS 1.1 whereas IBM MQ Version 8.0 and above support both JMS 1.1 and JMS 2.0.

Committing a unit of work in a JMS environment

Messages sent and received by the IBM MQ classes for JMS in a Liberty JVM server can be associated with a CICS unit of work (UOW). How this is done differs based on the mode used to connect to the queue manager.

When using JMS in a CICS standard-mode Liberty JVM server there is no implicit CICS task or UOW. To ensure IBM MQ updates are transactional, the application should use the Java Transaction API (JTA).

When connected in client mode in a CICS integrated-mode Liberty JVM server, messages sent and received by the IBM MQ classes for JMS in a Liberty JVM server can be associated with a CICS UOW. This association is not made by default and the application must call the `UserTransaction begin()` method to establish this association before updating any IBM MQ or CICS resources.

To complete a UOW, the `UserTransaction commit()` or `rollback()` methods should be used. The use of the `commit()` and `rollback()` methods on the following objects to commit or roll back the UOW is not supported:

- `Session` (JMS 1.1)
- `JmsContext` (JMS 2.0)
- `com.ibm.cics.server.Task`

For more information on `UserTransaction`, see Java Transaction API (JTA).

CICS abends during the processing of JMS requests

The use of IBM MQ classes for JMS and the bindings mode transport results in the issuing of IBM MQ MQI commands. In a CICS integrated-mode Liberty JVM server, CICS abends issued during processing of the MQI command are not converted into Java exceptions, and therefore are not catchable by a CICS Java application.

In this situation, the CICS transaction will abend and roll back to the last syncpoint.

Using IBM MQ classes for JMS in an OSGi JVM server

Java programs running in an OSGi JVM server can use the IBM MQ classes for JMS to access IBM MQ.

For an introduction to the IBM MQ classes for JMS, see Using WebSphere MQ classes for JMS.

Setting up an OSGi JVM server to use the IBM MQ classes for JMS

When a CICS Java application makes JMS requests, the requests are processed by the MQ classes supplied by IBM MQ.

In a CICS environment, the classes supplied by IBM MQ allow only connections in bindings mode. Any attempt to use connections to a remote queue manager in client mode results in an exception.

JMS requests are transformed into IBM MQ MQI calls, and processed as normal by the existing CICS-MQ adapter. The converted requests flow into the CICS-MQ adapter in exactly the same way as MQI requests from any other program (for example, a COBOL program).

To use a specific level of JMS, CICS must be connected to a IBM MQ queue manager that supports the appropriate level of JMS. For more information, see Using WebSphere MQ classes for JMS in a CICS OSGi JVM server.

Support from IBM MQ using the IBM MQ classes for JMS is provided in WebSphere MQ for z/OS version 7.1 and IBM MQ Version 8.0.

- Version 7.1 apply MQ fixpack 7.1.0.7 or any later fixpack level
- Version 8.0 apply base APAR PI28482 and fixpack 8.0.0.4 or any later fixpack level

Configuring an OSGi JVM server to support JMS

A JVM server is the runtime environment for Java applications. You can configure an OSGi JVM server to support applications that use JMS.

About this task

To enable the OSGi JVM server to support applications that use IBM MQ classes for JMS, the IBM MQ for JMS bundles need to be added to the set of middleware bundles that run in the OSGi framework within the JVM server. The framework must also have access to the associated set of native libraries.

For instructions, see Setting up the JVM server environment.

Programming with IBM MQ classes for JMS in the CICS IBM MQ environment

CICS Java programs may invoke various functions provided by the IBM MQ classes for JMS. The IBM MQ classes for JMS form part of the JMS-compliant IBM MQ offering.

From the perspective of the JMS specification, the IBM MQ classes for JMS treat a CICS JVM server as a Java EE compliant application server, that always has a JTA transaction in progress. In particular, the IBM MQ classes for JMS assume that they are running in an EJB container. This results in restrictions on the use of the JMS API in a CICS environment. For more details, see JMS API restrictions.

Support is provided for using the classic (JMS 1.1) and simplified (JMS 2.0) interfaces, provided that CICS is connected to a level of IBM MQ queue manager that supports the appropriate level of JMS and is using a suitable version of the IBM MQ classes for JMS.

Note: WebSphere MQ for z/OS version 7.1 only supports JMS 1.1 whereas IBM MQ Version 8.0 supports both JMS 1.1 and JMS 2.0.

Creating and configuring connection factories and destinations for a JMS application

CICS Java programs can create and configure connection factories and destinations.

The following approaches can be used to create and configure the IBM MQ implementations of connection factories and destinations:

- Using JNDI to retrieve administered objects
- Using the IBM JMS extensions
- Using the IBM MQ JMS extensions

Most users of JMS use a JNDI repository to locate a pre-configured set of connection factories and destinations. CICS does not provide a JNDI implementation, and the use of LDAP is not possible in an OSGi environment.

For details of the available options, and an example of how to register an initial context factory and the IBM MQ object factories with OSGi using the start method of a bundle adaptor, see Creating and configuring connection factories and destinations.

The connection between CICS and a IBM MQ QMGR is policed using the user ID of the CICS address space. Resource access to a queue is authorized by the transaction user ID. Specifying a user ID and password with a connection factory is therefore not supported.

Committing a unit of work in a JMS environment

When using a bindings mode connection, messages sent and received by the IBM MQ classes for JMS in a CICS JVM server environment are always associated with the CICS unit of work (UOW).

The UOW can only be completed by calling the commit or rollback methods on the `com.ibm.cics.server.Task` object, or by the CICS task ending normally, in which case the UOW is implicitly committed. The use of JMS API requests to commit or roll back the unit of work is not supported.

One exception to the above rule concerns support for non-persistent messages. These are also supported by the native MQI. For more details, see Transactional behavior.

For a mixed language application, an **EXEC CICS SYNCPOINT** command issued from a non- Java program will commit the whole unit of work, including the updates made to IBM MQ by a Java program.

CICS abends during the processing of JMS requests

The use of IBM MQ classes for JMS and the bindings mode transport results in the issuing of IBM MQ MQI commands. CICS abends issued during processing of the MQI command are not converted into Java exceptions, and therefore are not catchable by a CICS Java application.

In this situation, the CICS transaction abends and rolls back to the last syncpoint.

Connectivity from Java applications in CICS

Java programs in the CICS environment can open TCP/IP sockets and communicate with external processes. You can use Java programs as a gateway to connect to other enterprise applications that might not be available to CICS programs in other languages. For example, you can write a Java program to communicate with a remote servlet or database.

In some cases, this connectivity is integrated with CICS to provide enterprise qualities of service, such as distributed transactions and identity propagation. In other cases, you can use connectivity without distributed transactions and other services provided by CICS. Depending on the type of connectivity you require, third party vendor products might be available which enable connectivity with enterprise applications that are not natively supported by CICS.

Generally, JVMs in the CICS environment are similar in capability to batch mode JVMs. A batch mode JVM runs as a stand-alone process outside the CICS environment, and is typically started from a UNIX System Services command line or with a JCL job. Most applications that can work in a batch mode JVM can also run in a JVM in CICS to the same extent. For example, if you write a batch mode Java application to communicate with a non-IBM database using a third-party JDBC driver, then the same application is likely to work in a JVM in CICS. If you want to use vendor supplied code such as non-IBM JDBC drivers in a JVM in CICS , consult with your vendor to determine whether they support their code running in a JVM in CICS.

For more information about Java application behavior in CICS , see “ Java runtime environment in CICS ” on page 24.

Batch mode applications that run in a JVM in the CICS environment do not usually exploit the capabilities of CICS. For example, if a Java program in CICS updates records in a non-IBM database using a third-party JDBC driver, CICS is not aware of this activity, and does not attempt to include the updates in the current CICS transaction.

JCA local ECI support

You can deploy JCA ECI applications into a Liberty JVM server that is configured to use the JCA local ECI resource adapter. This topic applies to CICS integrated-mode Liberty only.

For information on developing applications refer to “Java EE Connector Architecture (JCA)” on page 95. To find out more about porting existing CICS Transaction Gateway applications, refer to “Porting JCA ECI applications into a Liberty JVM server” on page 97. For information on configuring JCA, see “Configuring the JCA local ECI resource adapter” on page 97.

The JCA ECI programming interfaces provided by the CICS TS JCA local ECI resource adapter are documented in Javadoc that is generated from the class definitions. The Javadoc is available at JCA local ECI Javadoc information.

The libraries and OSGi bundle required for application development are provided by the IBM CICS SDK for Java.

Packaging existing applications to run in a JVM server

If you are running Java applications in pooled JVMs, you can move them to run in a JVM server. Because a JVM server can handle multiple requests for Java applications in the same JVM, you can reduce the number of JVMs that are required to run the same workload. You must package the Java application as one or more OSGi bundles. You can use one of three methods to package the application:

Converting an existing Java project to a plug-in project

If you have an existing Java project, you can convert it to an OSGi plug-in project. The OSGi bundle can run in a pooled JVM environment and a JVM server.

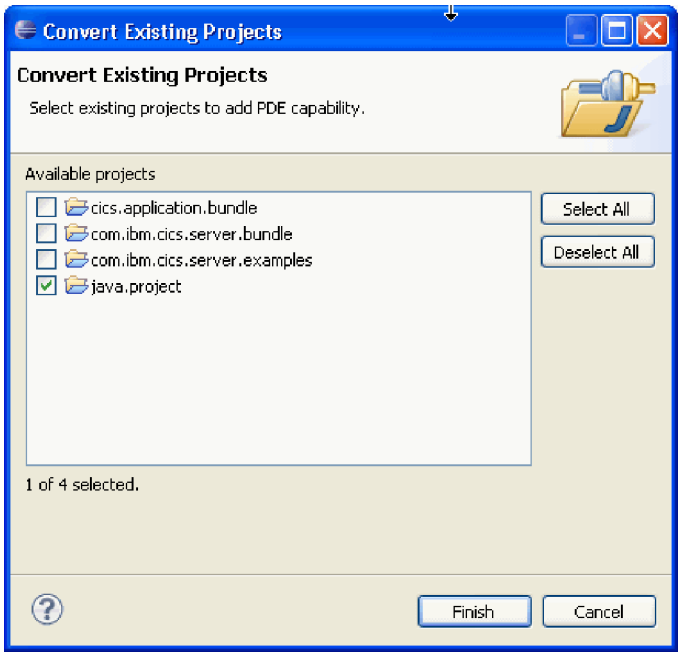
About this task

This task assumes that you have an existing Java project in your workspace, and you want to convert it to an OSGi plug-in project.

Procedure

1. In the Package Explorer view, right-click the Java project that you want to convert to a plug-in project, and click **Configure** > **Convert to Plug-in Projects**.

   The Convert Existing Projects dialog is displayed.



   The dialog contains a list of all the Java projects in your workspace. The one you chose to convert is selected. You can change your selection, or select more than one Java project to convert to a plug-in project.

2. Click **Finish**.

   The Java project is converted to a plug-in project. The project name does not change, but the project now includes a manifest file and a build properties file.

3. Required: You must now edit the plug-in manifest file and add the JCICS API dependencies. If you do not perform these steps, you will be able to export and install the bundle, but it will not run.

   Note: In CICS versions before CICS TS version 4.2 you had to add the Java class library, `dfjcics.jar`, to the Java build path. With CICS TS version 4.2, OSGi manages the build path for you. Before you perform the following steps you must edit the current build path and remove any references to `dfhjcics.jar`. If you do not remove all references to `dfhjcics.jar`, a `NoSuchMethodException` error occurs at run time.

a) In the Package Explorer view, right-click the project name and click **Plug-in Tools** > **Open Manifest**.

The manifest file opens in the manifest editor.

b) **Important**: In CICS versions before CICS TS version 4.2, the Java class library, known as JCICS, is supplied in the `dfjcics.jar` JAR file. In CICS TS version 4.2 the library is supplied in the `com.ibm.cics.server.jar` file. If your project manifest contains the declaration: **`Import-Package: dfhjcics.jar;`** you must remove the declaration before continuing with the remaining steps.

c) Select the **Dependencies** tab and in the Imported Packages section, click **ADD**.

The Package Selection dialog opens.

d) Select the package `com.ibm.cics.server` and click **OK**.

The package is displayed in the Imported Packages list.

e) Optional: Repeat the previous step to install the following package, if it is required for your application:

`com.ibm.record`
The Java API for legacy programs that use IByteBuffer from the Java Record Framework that came with VisualAge. Previously in the `dfjcics.jar` file.

f) Select **File** > **Save** to save the manifest file.

Results

You have successfully converted your existing Java project to a plug-in project.

What to do next

You must now update the manifest file to add a CICS-MainClass declaration. For more information, see the related link.

Importing the contents of a JAR file into an OSGi plug-in project

You can create a plug-in project from an existing JAR file. This method is useful when the application is already threadsafe and no refactoring or recompiling is required. The OSGi bundle can run in a pooled JVM environment and a JVM server.

About this task

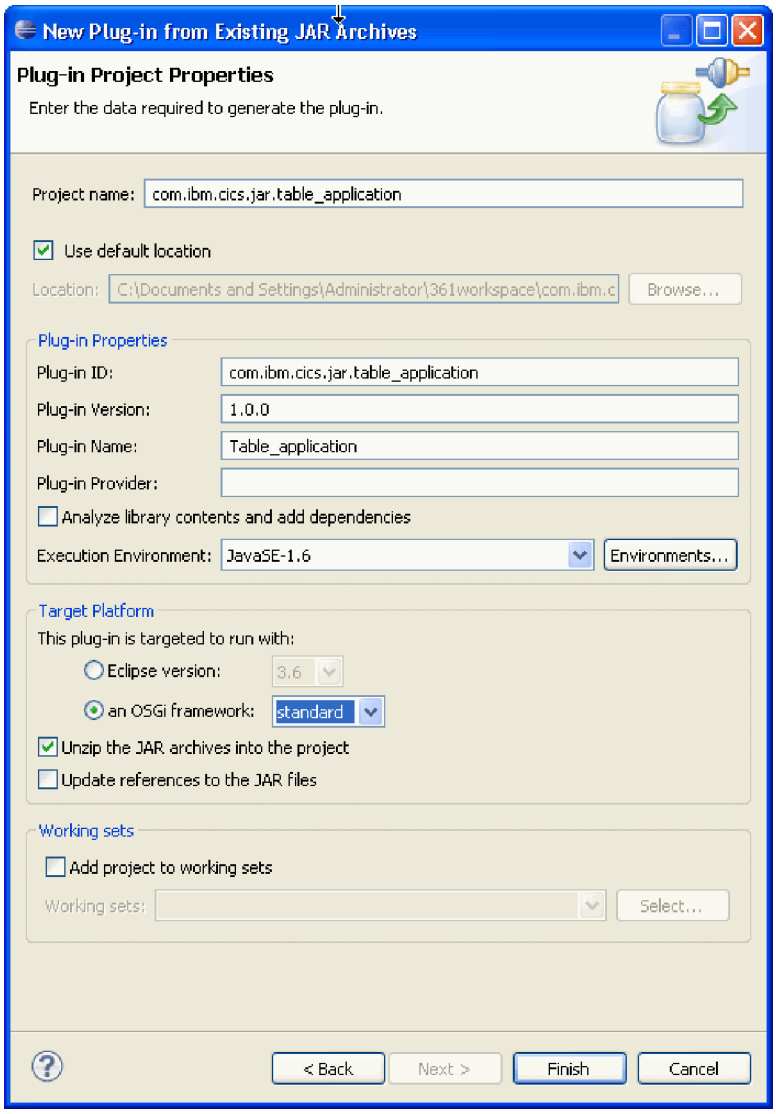
This task creates a new OSGi plug-in project from an existing JAR file. The JAR file must be on your local file system.

Procedure

1. On the Eclipse menu bar, click **File** > **New** > **Project** to open the New wizard.
2. Expand the **Plug-in Development** folder and click **Plug-in from Existing JAR Archives**. Click **Next**.

   The JAR selection dialog opens.
3. Locate the JAR file to convert. If the file is in your Eclipse workspace, click **Add**. If the file is in a folder on your computer, click **Add External** and browse to the JAR file. Select the required file and click **Open** to add it in the Jar selection dialog. Click **Next**.

   The Plug-in Project Properties dialog opens.



4. In the **Project name** field, enter the name of the project that you want to create. A project name is mandatory.
5. Complete the following fields in the Plug-in Properties section as required:

   Plug-in ID
   The plug-in ID is automatically generated from the project name; however, you can change the ID if you want to.

   Plug-in Name
   The plug-in name is automatically generated from the project name; however, you can change the name if you want to.

   Execution Environment
   This field specifies the minimum level of JRE required for the plug-in to run. Select the Java level that matches the execution environment in your CICS runtime target platform.
6. In the Target Platform section, select **an OSGI framework** and select **standard** from the menu.
7. Ensure that **Unzip the JAR archives into the project** is selected and click **Finish**.

   Eclipse creates the plug-in project in the workspace.
8. Required: You must now edit the plug-in manifest file and add the JCICS API dependencies. If you do not perform these steps, you will be able to export and install the bundle, but it will not run.

   a) In the Package Explorer view, right-click the project name and click **Plug-in Tools** > **Open Manifest**.

   The manifest file opens in the manifest editor.

b) Select the **Dependencies** tab and in the Imported Packages section, click **ADD**.

The Package Selection dialog opens.

c) Select the package `com.ibm.cics.server` and click **OK**.

The package is displayed in the Imported Packages list.

d) Optional: Repeat the previous step to install the following package, if it is required for your application:

`com.ibm.record`
The Java API for legacy programs that use IByteBuffer from the Java Record Framework that came with VisualAge. Previously in the `dfjcics.jar` file.

e) Select **File** > **Save** to save the manifest file.

Results

You have created an OSGi plug-in project from an existing JAR file.

What to do next

You must now update the manifest file to add a CICS-MainClass declaration. For more information, see the related link.

Importing a binary JAR file into an OSGi plug-in project

You can create a plug-in project from an existing binary JAR file. This method is useful in situations where there are licensing restrictions or where the binary file cannot be extracted. However, an OSGi bundle that contains a JAR file is not supported in a pooled JVM environment.

About this task

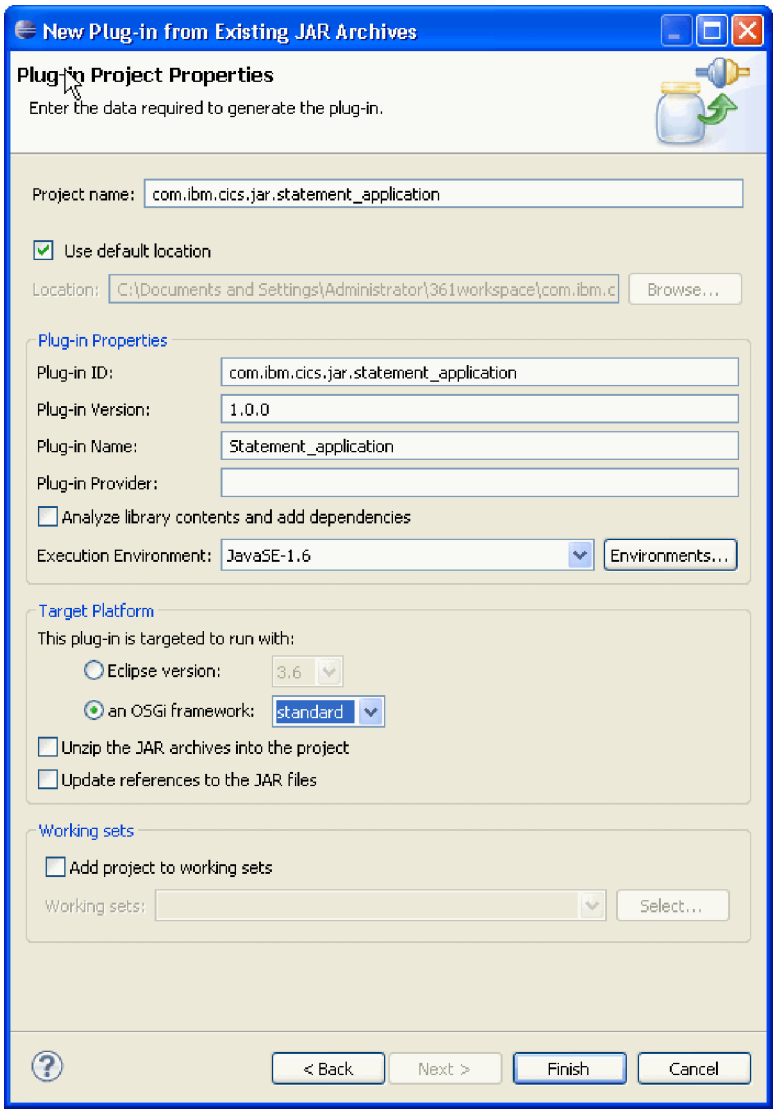
This task creates a new OSGi plug-in project from an existing binary JAR file. The JAR file must be on your local file system.

Procedure

1. On the Eclipse menu bar click **File** > **New** > **Project** to open the New wizard.
2. Expand the **Plug-in Development** folder and click **Plug-in from Existing JAR Archives**. Click **Next**.

   The JAR selection dialog opens.
3. Locate the JAR file to convert. If the file is in your Eclipse workspace, click **Add**. If the file is in a folder on your computer, click **Add External** and browse to the JAR file. Select the required file and click **Open** to add it in the Jar selection dialog. Click **Next**.

   The Plug-in Project Properties dialog opens.



4. In the **Project name** field, enter the name of the project that you want to create. A project name is mandatory.
5. Complete the following fields in the Plug-in Properties section as required:

   Plug-in ID
   The plug-in ID is automatically generated from the project name; however, you can change the ID if you want to.

   Plug-in Name
   The plug-in name is automatically generated from the project name; however, you can change the name if you want to.

   Execution Environment
   This field specifies the minimum level of JRE required for the plug-in to run. Select the Java level that matches the execution environment in your CICS runtime target platform.
6. In the Target Platform section, select **an OSGI framework** and select **standard** from the menu.
7. Ensure that **Unzip the JAR archives into the project** is not selected and click **Finish**.

   Eclipse creates the plug-in project in the workspace. The project contains the binary JAR file but the project is not supported in a pooled JVM environment.
8. Required: You must now edit the plug-in manifest file and add the JCICS API dependencies. If you do not perform these steps, you will be able to export and install the bundle, but it will not run.

   a) In the Package Explorer view, right-click the project name and click **Plug-in Tools** > **Open Manifest**.

The manifest file opens in the manifest editor.

b) Select the **Dependencies** tab and in the Imported Packages section, click **ADD**.

The Package Selection dialog opens.

c) Select the package `com.ibm.cics.server` and click **OK**.

The package is displayed in the Imported Packages list.

d) Optional: Repeat the previous step to install the following package, if it is required for your application:

`com.ibm.record`
The Java API for legacy programs that use IByteBuffer from the Java Record Framework that came with VisualAge. Previously in the `dfjcics.jar` file.

e) Select **File** > **Save** to save the manifest file.

Results

You have successfully created the plug-in project in the workspace.

What to do next

You must now update the manifest file to add a CICS-MainClass declaration. For more information, see the related link.

Chapter 3. Deploying applications to a JVM server

To deploy a Java application to a JVM server, the application must be packaged appropriately to install and run successfully. You can use the IBM CICS SDK for Java to package and deploy the application.

You have a number of options for deploying Java applications:

- Deploy one or more CICS bundles that include the OSGi bundles for the application into a JVM server that is running an OSGi framework.
- Deploy one or more CICS bundles that include one or more WAR files into a Liberty JVM server.
- Deploy one or more CICS bundles that include Enterprise Bundle Archive (EBA) files into a Liberty JVM server.
- Deploy one or more CICS bundles that include EAR files into a Liberty JVM server.
- Deploy an application bundle that comprises the CICS bundles and OSGi bundles into a platform.

Deploying OSGi bundles in a JVM server

To deploy a Java application in a JVM server, you must install the OSGi bundles for the application in the OSGi framework of the target JVM server.

Before you begin

The CICS bundle that contains the OSGi bundles for the application must be deployed to zFS. The target JVM server must be enabled in the CICS region.

About this task

A CICS bundle can contain one or more OSGi bundles. Because the CICS bundle is the unit of deployment, all the OSGi bundles are managed together as part of the BUNDLE resource. The OSGi framework also manages the lifecycle of the OSGi bundles, including the management of dependencies and versioning.

Ensure that all OSGi bundles that comprise a Java application component are deployed in the same CICS bundle. If there are dependencies between OSGi bundles, deploy them in the same CICS bundle. When you install the CICS BUNDLE resource, CICS ensures that all the dependencies between the OSGi bundles are resolved.

If you have dependencies on an OSGi bundle that contains a library of common code, create a separate CICS bundle for the library. In this case, it is important to install the CICS BUNDLE resource that contains the library first. If you install the Java application before the CICS bundles that it depends on, the OSGi framework is unable to resolve the dependencies of the Java application.

Do not attempt to install a CICS bundle that contains an OSGi bundle into a Liberty JVM server, as this configuration is not supported. Instead, you can either package the OSGi bundle together with your web application in an enterprise bundle archive (EBA), or you can use the WebSphere Liberty Profile bundle repository to make the OSGi bundle available to all web applications in the Liberty JVM server.

Procedure

1. Create a BUNDLE resource that specifies the directory of the bundle in zFS:
   a) In the CICS SM perspective, click **Definitions** > **Bundle Definitions** in the CICS Explorer menu bar to open the Bundles Definitions view.
   b) Right-click anywhere in the view and click **New** to open the New Bundle Definition wizard.
      Enter the details for the BUNDLE resource in the wizard fields.
   c) Install the BUNDLE resource.
      You can either install the resource in an enabled or disabled state:

- If you install the resource in a DISABLED state, CICS installs the OSGi bundles in the framework and resolves the dependencies, but does not attempt to start the bundles.
- If you install the resource in an ENABLED state, CICS installs the OSGi bundles, resolves the dependencies, and starts the OSGi bundles. If the OSGi bundle contains a lazy bundle activator, the OSGi framework does not attempt to start the bundle until it is first called by another OSGi bundle.

2. Optional: Enable the BUNDLE resource to start the OSGi bundles in the framework if the resource is not already in an ENABLED state.
3. Click **Operations** > **Bundles** in the CICS Explorer menu bar to open the Bundles view. Check the state of the BUNDLE resource.
   - If the BUNDLE resource is in an ENABLED state, CICS was able to install all the resources in the bundle successfully.
   - If the BUNDLE resource is in a DISABLED state, CICS was unable to install one or more resources in the bundle.

   If the BUNDLE resource failed to install in the enabled state, check the bundle parts for the BUNDLE resource. If any of the bundle parts are in the UNUSABLE state, CICS was unable to create the OSGi bundles. Typically, this state indicates that there is a problem with the CICS bundle in zFS. You must discard the BUNDLE resource, fix the problem, and then install the BUNDLE resource again.
4. Click **Operations** > **Java** > **OSGi Bundles** in the CICS Explorer menu bar to open the OSGi Bundles view. Check the state of the installed OSGi bundles and services in the OSGi framework.
   - If the OSGi bundle is in the STARTING state, the bundle activator has been called but not yet returned. If the OSGi bundle has a lazy activation policy, the bundle remains in this state until it is called in the OSGi framework.
   - If the OSGi bundles and OSGi services are active, the Java application is ready.
   - If the OSGi service is inactive it is possible that CICS detected an OSGi service with that name already exists in the OSGi framework.
   - If you disable the BUNDLE resource, the OSGi bundle moves to the RESOLVED state.
   - If the OSGi bundle is in the INSTALLED state, either it has not started or it failed to start because the dependencies in the OSGi bundle could not be resolved.
5. “ Invoking a Java application in a JVM server ” on page 151

Results
The BUNDLE is enabled, the OSGi bundles are successfully installed in the OSGi framework, and any OSGi services are active. The OSGi bundles are available to other bundles in the framework.

What to do next
You can make the Java application available to other CICS applications outside the OSGi framework, as described in “ Invoking a Java application in a JVM server ” on page 151.

Deploying a Java EE application in a CICS bundle to a Liberty JVM server

You can deploy a Java EE application that is packaged as a CICS bundle in a Liberty JVM server.

Before you begin
The Java EE application, either in the form of WAR files, EAR files or an EBA file, must be deployed as a CICS bundle in zFS. The target JVM server must be enabled in the CICS region.

For general information about creating and repackaging Java applications, see “Developing applications using the IBM CICS SDK for Java ” on page 24.

If you have dependencies on an OSGi bundle that contains a library of common code, install the bundle into the Liberty bundle repository, see "Deploying OSGi bundles in a JVM server" on page 147.

About this task

The CICS application model is to package Java application components in CICS bundles and deploy them to zFS. By installing the CICS bundles, you can manage the lifecycle of the application components.

A Java EE application can contain:

- One or more WAR files that provide the presentation layer and business logic of the application
- An OSGi Application Project, exported to an EBA file, which contains a web-enabled OSGi Bundle Project to provide the presentation layer and a set of further OSGi bundles that provide the business logic
- An Enterprise Application Archive (EAR) file containing one or more WAR files that provide the presentation layer and business logic

Procedure

1. Create a BUNDLE resource that specifies the directory of the bundle in zFS:
   a) In the CICS SM perspective in CICS Explorer, click **Definitions** > **Bundle Definitions** in the CICS Explorer menu bar to open the Bundles Definitions view.
   b) Right-click anywhere in the view and click **New** to open the New Bundle Definition wizard.
   Enter the details for the BUNDLE resource in the wizard fields.
   c) Install the BUNDLE resource.
   You can install the resource in an enabled or disabled state:
   - If you install the resource in a DISABLED state, CICS does not attempt to install the Java EE applications into the Liberty server.
   - If you install the resource in an ENABLED state, CICS installs the Java EE applications (WAR, EAR, EBA files) in the `${server.output.dir}/installedApps` directory and adds an `<application>` entry into `${server.output.dir}/installedApps.xml`.
2. Optional: Enable the BUNDLE resource to start the Java EE applications in the Liberty server, if the resource is not already in an ENABLED state.
3. Click **Operations** > **Bundles** in the CICS Explorer menu bar to open the Bundles view. Check the state of the BUNDLE resource.
   - If the BUNDLE resource is in an ENABLED state, CICS installed all the resources in the bundle successfully.
   - If the BUNDLE resource is in a DISABLED state, CICS was unable to install one or more resources in the bundle.

   If the BUNDLE resource failed to install in the enabled state, check the bundle parts for the BUNDLE resource. If any of the bundle parts are in the UNUSABLE state, a message is issued to explain the cause of the problem. For example, this state can indicate that there is a problem with the CICS bundle in zFS, or the associated JVMSERVER resource is not available. You must discard the BUNDLE resource, resolve the reported issue, and then install the BUNDLE resource again.
4. Optional: To run Java EE application requests on an application transaction, you can create URIMAP and TRANSACTION resources.
   Defining a URI map is useful if you want to control security to the application, because you can map the URI to a specific transaction and use transaction security. Typically, these resources are created as part of the CICS bundle and are managed with the application. However, you can choose to define these resources separately if preferred.
   a) Create a TRANSACTION resource for the application that sets the PROGRAM attribute to DFHSJTHP.

This CICS program handles the security checking of inbound Java EE requests to the Liberty JVM server. If you set any remote attributes, they are ignored by CICS because the transaction must always attach in the local CICS region.

b) Create a URIMAP resource that has a USAGE type of JVMSERVER. Set the TRANSACTION attribute to the name of the application transaction and set the SCHEME attribute to HTTP or HTTPS.

You can also use the USERID attribute to set a user ID. This value is ignored if the application security authentication mechanisms are used. If no authentication occurs and no user ID is set on the URI map, the work runs under CICS default user ID.

Results

The CICS resources are enabled, and the Java EE applications are successfully installed into the Liberty JVM server.

What to do next

You can test that the Java application is available through a web client. To update or remove the application, see Administering Java applications.

Deploying Java EE applications directly to a Liberty JVM server

You can deploy Java EE applications by defining the `application` element in `server.xml`, or by copying the application into a previously defined `dropins` directory.

Before you begin

The JVM server must be configured to use Liberty technology.

About this task

Java EE applications can be packaged as a Web Archive (WAR), a Enterprise Bundle Archive (EBA), or a Enterprise Application Archive (EAR).

Liberty provides two methods to install Java EE applications:

- You can add an `application` element in `server.xml`.
- Alternatively you can copy the application into the `dropins` directory of the Liberty JVM server. If you use `dropins`, CICS is will always run under the transaction CJSA and will not benefit from extra qualities of service such as CICS security.

Note:

- Do not use both techniques to deploy the same application into the same JVM server.
- If you accept the defaults that are provided by CICS autoconfigure, the `dropins` directory is not automatically created.

Procedure

- **To deploy an application by adding it to the server configuration file:**

  You must configure the following attributes for the application element in the `server.xml`:

  – `id` - Must be unique and is used internally by the server.
  – `name` - Must be unique.
  – `type` - Specifies the type of application. The supported types are WAR, EBA, and EAR.
  – `location` - Specifies the location of the application. The location can be an absolute path or a URL.

  For example:

```
<application
    id="com.ibm.cics.server.examples.wlp.tsq.app"
```

```
    name="com.ibm.cics.server.examples.wlp.tsq.app"
    type="eba"
    location="${server.output.dir}/path_to_app"/>
```

- **To create the dropins directory and deploy applications to it:**

  a) To enable `dropins`, you need to add configuration that is similar to the following example to your `server.xml`:

  ```
  <applicationMonitor dropins="dropins" dropinsEnabled="true" pollingRate="5s"
  updateTrigger="disabled"/>
  ```

  For more information, see Controlling dynamic updates.

  b) Use FTP to transfer the exported file in binary mode to the `dropins` directory. The directory path is `WLP_USER_DIR/servers/server_name/dropins`, where `server_name` is the value of the `com.ibm.cics.jvmserver.wlp.server.name` property. If the property is not set, the property is `defaultServer`.

Results

The Liberty JVM server installs the application.

What to do next

Access the Java EE application from a web browser to ensure that it is running correctly. To remove the application file, delete the WAR, EBA or EAR file from the `dropins` directory. If it was deployed with an application element, remove that element from `server.xml`.

Deploying common libraries to a Liberty JVM server

Deploy the common libraries according to whether it is supplied as DLL files, JAR files or OSGi bundles.

Procedure

- For common libraries supplied as DLL files, copy the files to a directory that is referred to by the `LIBPATH_SUFFIX` option of the JVM profile.

  For more information about LIBPATH_PREFIX and LIBPATH_SUFFIX, see JVM server options.
- For common libraries supplied as OSGi bundle JAR files, copy the JAR files to a directory that is referred to in a bundleRepository definition in the `server.xml` file.

  For more information, see *Bundle repository* in Manually tailoring server.xml.
- For common libraries supplied as JAR files but not OSGi bundles, copy the JAR files to a directory that is referred to in a global library definition in the `server.xml` file.

  For more information, see *Global/shared library* in Manually tailoring server.xml.

Invoking a Java application in a JVM server

There are many ways to call a Java application that is running in a JVM server. The method used will depend upon the characteristics of the JVM server.

About this task

You can invoke a web application running in a Liberty JVM server by using a HTTP request with a specific URL. Web applications cannot be driven directly from **EXEC CICS LINK** or **EXEC CICS START**. If you have Java EE applications that are implemented as Plain Old Java Objects (POJOs) and packaged in a WAR or an EAR, you can invoke their business logic components by using **EXEC CICS LINK** or **EXEC CICS START**.

To invoke a Java application that is running in an OSGi JVM server, you can either **EXEC CICS LINK** to a PROGRAM defined by Java, or **EXEC CICS START** a TRANSACTION that has a target PROGRAM defined

by Java. The PROGRAM definition specifies a JVMSERVER, and the name of a CICS generated OSGi service you want to invoke. Such linkable OSGi services are created by CICS when you install an OSGi bundle that includes a `CICS-MainClass` header in its manifest. The `CICS-MainClass` header identifies the main method of the Java class in the OSGi bundle that you want to act as an entry-point to the application.

An OSGi service is a well-defined interface that is registered in the OSGi framework. OSGi bundles and remote applications use the OSGi service to call application code that is packaged in an OSGi bundle. An OSGi bundle can export more than one OSGi service. For more information, see Updating OSGi bundles in an OSGi JVM server.

Invoking Java function in a classpath based JVM server is usually performed as part of a specific capability of a JVM server, such as Batch, Axis2 and SAML. For these capabilities the DFHSJJI vendor interface is provided.

Procedure

- For a web application developed as a web archive (WAR) file, as an enterprise application archive (EAR) file, or as an enterprise bundle archive (EBA) file containing web application bundle (WAB) files and running in a Liberty JVM server, invoke the application from the client browser by using a URL.

  For more information about invoking business logic components of Java EE applications, see " Preparing a Java EE application to be called by a CICS program " on page 76.
- For OSGi bundles that are deployed in an OSGi JVM server, follow these steps:
  a) Determine the symbolic name of the active OSGi service that you want to use in the OSGi framework.

     Click **Operations** > **Java** > **OSGi Services** in CICS Explorer to list the OSGi services that are active.
  b) Create a PROGRAM resource to represent the OSGi service to other CICS applications:
     - In the JVM attribute, specify YES to indicate that the program is a Java program.
     - In the JVMCLASS attribute, specify the symbolic name of the OSGi service. This value is case sensitive.
     - In the JVMSERVER attribute, specify the name of the JVMSERVER resource in which the OSGi service is running.
  c) You can call the Java application in either of two ways:
     - Use a 3270 or **EXEC CICS START** request that specifies a transaction identifier. Create a TRANSACTION resource that defines the PROGRAM resource for the OSGi service.
     - Use an **EXEC CICS LINK** request, an ECI call, or an EXCI call. Name the PROGRAM resource for the OSGi service when coding the request.
- For Axis2, Batch, or SAML function, see Configuring a JVM server for Axis2, Modern Batch overview, and Configuring CICS for SAML.

Results

You have created the definition to make your Java application available to other components. When CICS receives the request in the target JVM server, it invokes the specified Java class or Web application on a new CICS Java thread. If the associated OSGi service or Web application is not registered or is inactive, an error is returned to the calling program.

Deploying a CICS non-OSGi Java application

The Java applications are included in a CICS bundle and can be deployed directly to a z/OS UNIX System Services (z/OS UNIX) file system from CICS Explorer. The exported bundle includes the application JAR files that are used by CICS.

About this task

This task outlines the steps to deploy a non-OSGi Java application. The process is the same as for an OSGi application; the only difference is that CICS uses the application JAR file instead of the bundle. If you are not authorized to deploy the bundle directly to a z/OS file system, you can export the bundle as a compressed file. For more information, see Exporting a CICS bundle project to your local file system in the CICS Explorer product documentation.

Procedure

1. Convert the Java application to a plug-in project.
   Follow the instructions in “Converting an existing Java project to a plug-in project” on page 141.
2. Add the plug-in project to a CICS bundle.
   Follow the instructions in “Adding a project to a CICS bundle project” on page 29.
3. Deploy the bundle project to a z/OS Unix file system.
   Follow the instructions in Deploying a CICS bundle in the CICS Explorer product documentation.

Results

The Java application is exported to z/OS UNIX. The exported bundle includes the application JAR files.

Chapter 4. Setting up Java support

Perform the basic setup tasks to support Java in your CICS region and configure a JVM server to run Java applications.

Before you begin

The Java components that are required for CICS are set up during the installation of the product. You must ensure that the Java components are installed correctly by using the information in Verifying your Java components installation in Installing.

About this task

CICS uses files in z/OS UNIX to start the JVM. You must ensure that your CICS region is configured to use the correct zFS directories, and that those directories have the correct permissions. After you configure CICS and set up zFS, you can configure a JVM server to run Java applications.

Procedure

1. Set the JVMPROFILEDIR system initialization parameter to a suitable directory in z/OS UNIX where you want to store the JVM profiles that are used by the CICS region.

   For more information, see “Setting the location for the JVM profiles” on page 155.
2. Ensure that your CICS region has enough memory to run Java applications.

   For more information, see “Setting the memory limits for Java” on page 156.
3. Give your CICS region permission to access the resources that are held in z/OS UNIX, including your JVM profiles, directories, and files that are required to create JVMs.

   For more information, see “Giving CICS regions access to z/OS UNIX directories and files” on page 157.
4. Set up a JVM server.

   You can configure a JVM server to run different workloads. For more information, see “Setting up a JVM server” on page 159.
5. Optional: Enable a Java security manager to protect a Java application from performing potentially unsafe actions.

   For more information, see Enabling a Java security manager.
6. Set the JAVA_DUMP_TDUMP_PATTERN unformatted storage dump parameter.

   The dump is written to a sequential MVS data set, which can be changed by specifying a value for the environment variable JAVA_DUMP_TDUMP_PATTERN. Ensure that the CICS region user ID has UPDATE access to data sets matching this pattern, otherwise diagnostic data is lost. For more information, see Using dump agents on z/OS.

Results

You set up your CICS region to support Java and created a JVM server to run Java applications.

What to do next

If you are upgrading existing Java applications, follow the guidance in Upgrading. To start running Java applications in a JVM server, see Deploying applications to a JVM server.

Setting the location for the JVM profiles

CICS loads the JVM profiles from the z/OS UNIX directory that is specified by the **JVMPROFILEDIR** system initialization parameter. You must change the value of the **JVMPROFILEDIR** parameter to a new

location and copy the supplied sample JVM profiles into this directory so that you can use them to verify your installation.

Before you begin

The USSHOME system initialization parameter must specify the root directory for CICS files on z/OS UNIX.

About this task

The CICS-supplied sample JVM profiles are customized for your system during the CICS installation process, so you can use them immediately to verify your installation. You can customize copies of these files for your own Java applications.

The settings that are suitable for use in JVM profiles can change from one CICS release to another, so for ease of problem determination, use the CICS-supplied samples as the basis for all profiles. Check the upgrading information to find out what options are new or changed in the JVM profiles.

Procedure

1. Set the JVMPROFILEDIR system initialization parameter to the location on z/OS UNIX where you want to store the JVM profiles used by the CICS region.

   The value that you specify can be up to 240 characters long.

   The supplied setting for the **JVMPROFILEDIR** system initialization parameter is `/usr/lpp/cicsts/cicsts54/JVMProfiles`, which is the installation location for the sample JVM profiles. This directory is not a safe place to store your customized JVM profiles, because you risk losing your changes if the sample JVM profiles are overwritten when program maintenance is applied. So you must always change **JVMPROFILEDIR** to specify a different z/OS UNIX directory where you can store your JVM profiles. Choose a directory where you can give appropriate permissions to the users who must customize the JVM profiles.
2. Copy the supplied sample JVM profiles from their installation location to the z/OS UNIX directory.

   When you install CICS, the sample JVM profiles are placed in a zFS directory. This directory is specified by the **USSDIR** parameter in the DFHISTAR installation job. The default installation directory is `/usr/lpp/cicsts/cicsts54/JVMProfiles`.

Results

You have copied the sample JVM profiles to a zFS directory and configured CICS to use that directory. The sample JVM profiles contain default values so that you can use them immediately to set up a JVM server.

What to do next

Ensure that CICS and Java have enough memory to run Java applications, as described in “Setting the memory limits for Java” on page 156. You must also ensure that the CICS region has access to the z/OS UNIX directories where Java is installed and the Java applications are deployed. For more information, see “Giving CICS regions access to z/OS UNIX directories and files” on page 157.

Setting the memory limits for Java

Java applications require more memory than programs written in other languages. You must ensure that CICS and Java have enough storage and memory available to run Java applications.

About this task

Java uses storage below the 16 MB line, 31-bit storage, and 64-bit storage. The storage required for the JVM heap comes from the CICS region storage in MVS, and not the CICS DSAs.

Procedure

1. Ensure that the z/OS **MEMLIMIT** parameter is set to a suitable value.

This parameter limits the amount of 64-bit storage that the CICS address space can use. CICS uses the 64-bit version of Java and you must ensure that **MEMLIMIT** is set to a large enough value for both this and other use of 64-bit storage in the CICS region.

See the following topics:

- Calculating storage requirements for JVM servers
- Estimating, checking, and setting MEMLIMIT in Improving performance

2. Ensure that the **REGION** parameter on the startup job stream is large enough for Java to run.

   Each JVM require some storage below the 16 MB line to run applications, including just-in-time compiled code, and working storage to pass parameters to CICS.

Giving CICS regions access to z/OS UNIX directories and files

CICS requires access to directories and files in z/OS UNIX. During installation, each of your CICS regions is assigned a z/OS UNIX user identifier (UID). The regions are connected to a RACF group that is assigned a z/OS UNIX group identifier (GID). Use the UID and GID to grant permission for the CICS region to access the directories and files in z/OS UNIX.

Before you begin

Ensure that you are either a superuser on z/OS UNIX, or the owner of the directories and files. The owner of directories and files is initially set as the UID of the system programmer who installs the product. The owner of the directories and files must be connected to the RACF group that was assigned a GID during installation. The owner can have that RACF group as their default group (DFLTGRP) or can be connected to it as one of their supplementary groups.

About this task

z/OS UNIX System Services treats each CICS region as a UNIX user. You can grant user permissions to access z/OS UNIX directories and files in different ways. For example, you can give the appropriate group permissions for the directory or file to the RACF group to which your CICS regions connect. This option might be best for a production environment and is explained in the following steps.

Procedure

1. Identify the directories and files in z/OS UNIX to which your CICS regions require access.

JVM server options	Default directories	Permission	Description
JAVA_HOME	`/usr/lpp/java/J7.0_64`	read and execute	IBM 64-bit SDK for z/OS, Java Technology Edition directories
USSHOME	`/usr/lpp/cicsts/cicsts54`	read and execute	The installation directory for CICS files on z/OS UNIX. Files in this directory include sample profiles and CICS-supplied JAR files.
WORK_DIR	`/u/`*`CICS region userid`*	read, write, and execute	The working directory for the CICS region. This directory contains input, output, and messages from the JVMs.
JVMPROFILEDIR	*`USSHOME`*`/JVMProfiles/`	read and execute	Directory that contains the JVM profiles for the CICS region, as specified in the **JVMPROFILEDIR** system initialization parameter.

JVM server options	Default directories	Permission	Description
WLP_USER_DIR	WORK_DIR/APPLID/ JVMSERVER/wlp/usr/	read, write, and execute	Specifies the directory that contains the configuration files for the Liberty JVM server. WLP_USER_DIR needs additional x permissions (read, write, execute) if Liberty JVM server autoconfigure is used as CICS must be able to write to server.xml.
WLP_OUTPUT_DIR	WLP_USER_DIR/servers	read, write, and execute	Specifies the output directory for the Liberty JVM server.

2. List the directories and files to show the permissions.

   Go to the directory where you want to start, and issue the following UNIX command:

   ```
   ls -la
   ```

   If this command is issued in the z/OS UNIX System Services shell environment when the current directory is the home directory of CICSHT##, you might see a list such as the following example:

   ```
   /u/cicsht##:>ls -la
   total 256
   drwxr-xr-x   2 CICSHT## CICSTS54     8192 Mar 15  2008 .
   drwx------   4 CICSHT## CICSTS54     8192 Jul  4 16:14 ..
   -rw-------   1 CICSHT## CICSTS54     2976 Dec  5  2010 Snap0001.trc
   -rw-r--r--   1 CICSHT## CICSTS54     1626 Jul 16 11:15 dfhjvmerr
   -rw-r--r--   1 CICSHT## CICSTS54        0 Mar 15  2010 dfhjvmin
   -rw-r--r--   1 CICSHT## CICSTS54      458 Oct  9 14:28 dfhjvmout
   /u/cicsht##:>
   ```

3. If you are using the group permissions to give access, check that the group permissions for each of the directories and files give the level of access that CICS requires for the resource.

   Permissions are indicated, in three sets, by the characters `r, w, x` and `-`. These characters represent read, write, execute, and none, and are shown in the left column of the command line, starting with the second character. The first set are the owner permissions, the second set are the group permissions, and the third set are other permissions.

   In the previous example, the owner has read and write permissions to `dfhjvmerr`, `dfhjvmin`, and `dfhjvmout`, but the group and all others have only read permissions.

4. If you want to change the group permissions for a resource, use the UNIX command `chmod`.
   The following example sets the group permissions for the named directory and its subdirectories and files to read, write, and execute. `-R` applies permissions recursively to all subdirectories and files:

   ```
   chmod -R g=rwx directory
   ```

   The following example sets the group permissions for the named file to read and execute:

   ```
   chmod g+rx filename
   ```

   The following example turns off the write permission for the group on two named files:

   ```
   chmod g-w filename filename
   ```

   In all these examples, g designates group permissions. If you want to correct other permissions, u designates user (owner) permissions, and o designates other permissions.

5. Assign the group permissions for each resource to the RACF group that you chose for your CICS regions to access z/OS UNIX. You must assign group permissions for each directory and its subdirectories, and for the files in them.

   Enter the following UNIX command:

   ```
   chgrp -R GID directory
   ```

GID is the numeric GID of the RACF group and *directory* is the full path of a directory to which you want to assign the CICS regions permissions.

For example, to assign the group permissions for the `/usr/lpp/cicsts/cicsts54` directory, use the following command:

```
chgrp -R GID /usr/lpp/cicsts/cicsts54
```

Because your CICS region user IDs are connected to the RACF group, the CICS regions have the appropriate permissions for all these directories and files.

Results

You have ensured that CICS has the appropriate permissions to access the directories and files in z/OS UNIX to run Java applications.

When you change the CICS facility that you are setting up, such as moving files or creating new files, remember to repeat this procedure to ensure that your CICS regions have permission to access the new or moved files.

What to do next

Verify that your Java support is set up correctly using the sample programs and profiles.

Setting up a JVM server

To run Java applications, web applications, Axis2, or a CICS Security Token Service in a JVM server, you must set up the CICS resources and create a JVM profile that passes options to the JVM.

About this task

A JVM server can handle multiple concurrent requests for different Java applications in a single JVM. The JVMSERVER resource represents the JVM server in CICS. The resource defines the JVM profile that specifies configuration options for the JVM, the program that provides values to the Language Environment enclave, and the thread limit. A JVM server can run different types of workload. A JVM profile is supplied for each different use of the JVM server:

- To run applications that are packaged as OSGi bundles, configure the JVM server with the `DFHOSGI.jvmprofile`. This profile contains the options to run an OSGi framework in the JVM server.
- To run applications that include Liberty in CICS, configure the JVM server with the `DFHWLP.jvmprofile`. This profile contains the options to run a web container that is based on Liberty technology. The web container also includes an OSGi framework and can therefore run applications that are packaged as OSGi bundles.
- To run SOAP processing for web services with the Axis2 SOAP engine, configure the JVM server with the `DFHJVMAX.jvmprofile`. This profile contains the options to run Axis2 in the JVM server.
- To run a CICS Security Token Service (STS), configure the JVM server with the `DFHJVMST.jvmprofile`. This profile contains the options to run an STS.

Any changes that you make to the profiles apply to all JVM servers that use it. When you customize each profile, make sure that the changes are suitable for all the Java applications that use the JVM server.

You can either configure JVM servers and JVM profiles with CICS online resource definition, or you can use the CICS Explorer to define and package JVMSERVER resources and JVM profiles in CICS bundles. For more information, see Working with bundles in the CICS Explorer product documentation.

Results

The JVM server is configured and ready to run a Java workload.

What to do next

Configure the security for your Java environment. Give appropriate access to application developers to deploy and install Java applications, and authorize application users to run Java programs and transactions in CICS.

Configuring an OSGi JVM server

Configure the JVM server to run an OSGi framework if you want to deploy Java applications that are packaged in OSGi bundles.

About this task

The JVM server contains an OSGi framework that handles the class loading automatically, so you cannot add standard class path options to the JVM profile. The supplied sample, DFHOSGI.jvmprofile, is suitable for an OSGi JVM server. This task shows you how to define a JVM server for an OSGi application from this sample profile.

You can define the JVM server either with CICS online resource definition or in a CICS bundle in CICS Explorer .

Procedure

1. Create a JVMSERVER resource for the JVM server.
   a) Specify a name for the JVM profile for the JVM server.

      On the JVMPROFILE attribute of JVMSERVER, specify a 1 - 8 character name. This name is used for the prefix of the JVM profile, which is the file that holds the configuration options for the JVM server. You do not need to specify the suffix, .jvmprofile, here.

   b) Specify the thread limit for the JVM server.

      On the THREADLIMIT attribute of JVMSERVER, specify the maximum number of threads that are allowed in the Language Environment enclave for the JVM server. The number of threads depends on the workload that you want to run in the JVM server. To start with, you can accept the default value and tune the environment later. You can set up to 256 threads in a JVM server.
2. Create the JVM profile to define the configuration options for the JVM server.

   You can use the sample profile, DFHOSGI.jvmprofile, as a basis. This profile contains a subset of options that are suitable for starting the JVM server. All options and values for the JVM profile are described in “JVM profile validation and properties” on page 177. Follow the coding rules, including those for the profile name, in “Rules for coding JVM profiles” on page 177.
   a) Set the location for the JVM profile.

      The JVM profile must be in the directory that you specify on the system initialization parameter, JVMPROFILEDIR. For more information, see “Setting the location for the JVM profiles” on page 155.

   b) Make the following changes to the sample profile:

      - Set JAVA_HOME to the location of your installed IBM Java SDK.
      - Set WORK_DIR to your choice of destination directory for messages, trace, and output from the JVM server.
      - Set TZ to specify the timezone for timestamps on messages from the JVM server. An example for the United Kingdom is TZ=GMT0BST,M3.5.0,M10.4.0.

   c) Save your changes to the JVM profile.

      The JVM profile must be saved as EBCDIC on the z/OS UNIX System Services file system.
3. Install and enable the JVMSERVER resource.

Results

CICS creates a Language Environment enclave and passes the options from the JVM profile to the JVM server. The JVM server starts up and the OSGi framework resolves any OSGi middleware bundles. When the JVM server completes startup successfully, the JVMSERVER resource installs in the ENABLED state.

If an error occurs, for example CICS is unable to find or read the JVM profile, the JVM server fails to start. The JVMSERVER resource installs in the DISABLED state, and CICS issues error messages to the system log.

What to do next

- Configure the location for JVM logs as described in Controlling the location for JVM stdout, stderr, JVMTRACE, and dump output.
- Install OSGi bundles for the application in the OSGi framework of the JVM server, as described in Deploying OSGi bundles in a JVM server.
- Specify any directories that contain native C dynamic link library (DLL) files, such as DB2 or IBM MQ. You specify these directories on the LIBPATH_SUFFIX option in the JVM profile.
- Specify middleware bundles that you want to run in the OSGi framework. Middleware bundles are a type of OSGi bundle that contains Java classes to implement shared services, such as connecting to IBM MQ and DB2. You specify these bundles on the OSGI_BUNDLES option in the JVM profile.

JVM profile example

Example JVM profile for an OSGi application.

The following excerpt shows an example JVM profile that is configured to start an OSGi framework that uses Db2 Version 11 and the JDBC 4.0 OSGi middleware bundle:

```
#*********************************************************************
#
#                          Required parameters
#                          -------------------
#
# When using a JVM server, the set of CICS options that are supported
JAVA_HOME=/usr/lpp/java/J7.0_64
WORK_DIR=.
LIBPATH_SUFFIX=/usr/lpp/db2v11/jdbc/lib
...
#*********************************************************************
#
#                     JVM server specific parameters
#                     ------------------------------
#
OSGI_BUNDLES=/usr/lpp/db2v11/jdbc/classes/db2jcc4.jar,\
             /usr/lpp/db2v11/jdbc/classes/db2jcc_license_cisuz.jar
OSGI_FRAMEWORK_TIMEOUT=60
#
#*********************************************************************
#
#                               JVM options
#                               -----------
# The following option sets the Garbage collection Policy.
#
-Xgcpolicy:gencon
#
#*********************************************************************
#
#                  Setting user JVM system properties
#                  ----------------------------------
#
# -Dcom.ibm.cics.some.property=some_value
#
#*********************************************************************
#
#              Unix System Services Environment Variables
#              ------------------------------------------
#
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,SYSDUMP),ONINTERRUPT(NONE)"
```

```
#
#
```

Configuring a Liberty JVM server

Configure the Liberty JVM server if you want to deploy Java EE applications such as EJBs, JSP, JSF and servlets.

About this task

You have two ways of configuring a Liberty JVM server:

Autoconfigure
CICS automatically creates and updates the configuration file for Liberty, `server.xml`, from templates that are supplied in the CICS installation directory. Autoconfigure gets you started quickly with a minimal set of configuration values in Liberty. To enable autoconfigure, set the JVM system property, **`-Dcom.ibm.cics.jvmserver.wlp.autoconfigure`** property to `true`. If you are defining the JVM server in a CICS bundle, set this option.

Manually configuring
This is the default setting. You supply the configuration files and all values. Manually configuring is appropriate where you want to remain in full control of the Liberty server configuration.

To define the JVM server, see Ways of defining CICS resources.

Procedure

1. Create a JVMSERVER resource. If you want to create a JVMSERVER resource within a CICS bundle, see Artifacts that can be deployed in bundles.
   a) Specify a name for the JVM profile for the JVM server.
      On the JVMPROFILE attribute of JVMSERVER, specify a 1 - 8 character name. This name is used for the file name of the JVM profile, which is the file that holds the configuration options for the JVM server. You do not need to specify the file type, `.jvmprofile`, here.
   b) Specify the thread limit for the JVM server.
      On the THREADLIMIT attribute of the JVMSERVER, specify the maximum number of threads you want to allocate. The actual number of threads that are used depends on the workload that you run in the JVM server. To start with, you can accept the default value and tune the environment later. You can set up to 256 threads in a JVM server.
   c) Set the location for the JVM profile.
      The JVM profile must be in the directory that you specify on the system initialization parameter, **`JVMPROFILEDIR`**. For more information, see “Setting the location for the JVM profiles” on page 155.
2. Create the JVM profile to define the configuration options for the JVM server.
   You can use the sample profile, `DFHWLP.jvmprofile`, as a basis. This profile contains a subset of options that are suitable for starting the JVM server. All options and values for the JVM profile are described in “JVM profile validation and properties” on page 177. Follow the coding rules, including those for the profile name, in “Rules for coding JVM profiles” on page 177.
   a) Make the following changes to the sample profile:
      - Set **`JAVA_HOME`** to the location of your installed IBM Java SDK.
      - Set **`WORK_DIR`** to your choice of destination directory for messages, trace, and output from the JVM server.
      - Set **`WLP_INSTALL_DIR`** to `&USSHOME;/wlp`
      - Set **`TZ`** to specify the timezone for time stamps on messages from the JVM server. An example for the United Kingdom is `TZ=GMT0BST,M3.5.0,M10.4.0`
      - Set **`-Dfile.encoding`** to ISO-8859-1, for example `-Dfile.encoding=ISO-8859-1`.

- (Optional) Set **CICS_WLP_MODE** to choose the level of integration between CICS and Liberty.

  See JVM server options for more information about JVM server options.

b) Save your changes to the JVM profile.

   The JVM profile must be saved in EBCDIC file encoding on UNIX System Services and the file type must be `.jvmprofile`.

3. Create the Liberty server configuration.

   Manually creating JVM servers is appropriate when the configuration files need to be carefully controlled. For more information, see "Manually creating a Liberty server" on page 165 and Manually tailoring server.xml.

   Important: You should use autoconfigure if you are defining the JVM server in a CICS bundle, as the `server.xml` configuration file cannot be included with the JVM profile in a CICS bundle.

4. Install and enable the JVMSERVER resource.

Results

The JVMSERVER reads the JVM profile and initializes itself based on the provided settings. If autoconfigure is enabled and a Liberty server configuration does not exist, it will be created. If autoconfigure is not enabled and there is no configuration, or the configuration is incorrect, the JVMSERVER will become DISABLED and report an appropriate failure. On subsequent start up, the JVMSERVER will use the existing configuration and launch the Liberty server instance. When the JVMSERVER completes startup successfully, the JVMSERVER resource installs in the ENABLED state.

If an error occurs, for example, CICS is unable to find or read the JVM profile, the JVM server fails to initialize. The JVM server is installed in the DISABLED state and CICS issues error messages to the system log. See Troubleshooting Liberty JVM servers and Java web applications for help. To confirm that Liberty successfully started within your JVM server, consult the `messages.log` file in the WLP_USER_DIR output directory on zFS.

CAUTION: Do not use the Liberty bin/server script to start or stop a Liberty server that is running in a JVM server.

Note: In CICS integrated-mode Liberty, the current number of threads indicated by the JVM server will return a positive value and can fluctuate even when no workload is running. This is because threads are pooled within Liberty for efficiency.

What to do next

- Run the CICS Liberty default web application to verify the Liberty JVM server is running by using the following URL: `http://server:port/com.ibm.cics.wlp.defaultapp/`. For more information, see Configuring the CICS Default Web Application.
- Specify any directories that contain native C dynamic link library (DLL) files, such as IBM MQ. Middleware and tools that are supplied by IBM or by vendors might require DLL files to be added to the library path.
- Add support for security. See Configuring security for a Liberty JVM server.
- Install the Java EE applications (EAR files, WAR files, and EBA files), as described in Deploying a Java EE application in a CICS bundle to a Liberty JVM server.
- Liberty bootstrap properties can be placed in the JVM profile to achieve the same effect as using a Liberty bootstrap.properties file.
- By default, Liberty and OSGi caches are not cleared on start-up of the JVM server. Should you encounter caching issues, or receive guidance from the IBM Service team to clean your server, this can be achieved by using one of two approaches:
  - Add `-Dcom.ibm.cics.jvmserver.wlp.args=--clean` to your JVM profile.
  - Add `-Dorg.osgi.framework.storage.clean=onFirstInit` to your JVM profile.

In both cases, remove the option once the server has started to ensure subsequent restarts are not performance impacted.

- Be aware that by default, when Liberty is configured, two defaulted settings are applied but are not visible in `server.xml`. See CICS Liberty defaulted settings for more information.
- For more information on general Liberty set up see this overview on Liberty, Liberty overview.

CICS standard-mode Liberty: Java EE 7 Full Platform support without full CICS integration

Use the CICS embedded Liberty JVM server in standard mode to port and deploy Liberty applications from other platforms to CICS without changing your application. Standard mode is ideal for hosting applications that are written for and rely on the Java Enterprise Edition (Java EE) Full Platform, but do not require full integration with CICS. Applications running on CICS standard-mode Liberty can take advantage of Liberty services, management, and security, and benefit from the performance and capabilities of Java on z/OS, the z Systems platform, and close proximity to data in DB2 and IBM MQ.

CICS standard-mode Liberty is based on the Java EE 7 certified IBM WebSphere Application Server Liberty. Java EE extends the core Java SE by providing the APIs and environment for running multi-tiered, scalable, and secure network applications. Java EE 7 Full Platform includes the Web Profile and several other features, such as Enterprise JavaBeans and Batch Applications for the Java Platform.

Manage the creation, life-cycle and configuration of CICS standard-mode Liberty using CICS JVM server technology. Applications running on CICS standard-mode Liberty do not have access to CICS resources by default, but can submit work to the `CICSExecutorService` using the `runAsCICS()` method. Work submitted to the `CICSExecutorService` has full access to the JCICS API, runs in a CICS unit-of-work under a CICS task, and is committed on completion of the thread. Work submitted to the `CICSExecutorService` does not have access to the Java EE APIs.

JVM profile example

Example JVM profile for Liberty server.

The following excerpt shows an example JVM profile that is configured to automatically create the required configuration files and directory structure. It uses DB2 Version 11:

```
#*********************************************************************
# JVM profile: DFHWLP
#
JAVA_HOME=/java/java71_64/J7.1_64
WORK_DIR=.
#*********************************************************************
# JVM server parameters
#
OSGI_FRAMEWORK_TIMEOUT=60
#*********************************************************************
# Liberty JVM server
#
-Dcom.ibm.ws.logging.console.log.level=INFO
-Dcom.ibm.cics.jvmserver.wlp.autoconfigure=true
-Dcom.ibm.cics.jvmserver.wlp.server.http.port=12345
-Dcom.ibm.cics.jvmserver.wlp.server.host=*
-Dcom.ibm.cics.jvmserver.wlp.jdbc.driver.location=/usr/lpp/db2v11/jdbc
-Dfile.encoding=ISO-8859-1
WLP_INSTALL_DIR=&USSHOME;/wlp
WLP_USER_DIR=./&APPLID;/&JVMSERVER;
#*********************************************************************
# JVM options
-Xgcpolicy:gencon
-Xms128M
-Xmx256M
-Xmso128K
#*********************************************************************
# Unix System Services Environment Variables
TZ=CET-1CEST,M3.5.0,M10.5.0
```

Manually creating a Liberty server

Manually creating a Liberty server in zFS for the JVM server.

Procedure

1. Create the Liberty server directory structure in zFS for the JVM server.

   The JVM server expects configuration files to be in the *WLP_USER_DIR*/servers/*server_name* directory, where *WLP_USER_DIR* is the value of the WLP_USER_DIR option and *server_name* is the value of the **com.ibm.cics.jvmserver.wlp.server.name** property. The *server_name* property is always prefixed with -D. For more information on these server options, see JVM server options.
2. Create the Liberty server configuration in the *server_name* directory.

   As a minimum, you must create the server.xml file. You can base it on the template that is supplied as wlp/templates/servers/defaultServer/server.xml in the installation directory of Liberty. This file must be saved in the ASCII file encoding of ISO-8859-1 and tagged with this encoding using the UNIX command **chtag -c ISO8859-1 -t <file>**.
3. Edit the server.xml file for your installation.

   Update the <httpEndpoint> with the host name and port number. For information about configuring server.xml in a JVM server, see "Manually tailoring server.xml" on page 166. If you want to use security, see Configuring security for a Liberty JVM server.

   Attention: The server.xml file configures a single Liberty JVM server. Do not attempt to share a server.xml file among two or more Liberty JVM servers.

Configuring the CICS Default Web Application

The CICS Liberty Default Web Application can be used to verify that the Liberty server is running and view the server configuration. You can use it to view the JVM profile and server logs, and the Liberty server.xml and messages.log files.

Before you begin

Without application security enabled, full access to the Default Web Application is available to all users. If you have autoconfigure enabled and run with CICS security (sec=yes), or you have manually configured your server.xml by adding the cicsts:security-1.0 feature, your user ID requires additional permissions to run the application. For access to the Default servlet and basic information, you need to be in the User role. For access to the FileViewer servlet, you need to be in the Administrator role.

Procedure

1. If you are using SAF authorization, and your server.xml contains the <safAuthorization .../> element, you need to create these profiles:

   a) To access the Default servlet, use the following example:

```
RDEFINE EJBROLE BBGZDFLT.com.ibm.cics.wlp.defaultapp.User UACC(NONE)
PERMIT BBGZDFLT.com.ibm.cics.wlp.defaultapp.User CLASS(EJBROLE) ID(WLPSVRS) ACCESS(READ)
SETROPTS RACLIST(EJBROLE) REFRESH
```

   b) To access the FileViewer servlet, use the following example:

```
RDEFINE EJBROLE BBGZDFLT.com.ibm.cics.wlp.defaultapp.Administrator UACC(NONE)
PERMIT BBGZDFLT.com.ibm.cics.wlp.defaultapp.Administrator CLASS(EJBROLE) ID(WLPSVRS) ACCESS(READ)
SETROPTS RACLIST(EJBROLE) REFRESH
```

2. Alternatively, if you do not have SAF authorization configured, then the default JEE role-based access is used.

   - CICS provides a default authorization definition as shown in the following configuration. Access to the Default servlet is granted through the User role to the special subject

ALL_AUTHENTICATED_USERS, which means all users are authenticated. By default CICS does not provide access to the FileViewer servlet.

```
<authorization-roles id="com.ibm.cics.wlp.defaultapp">
    <security-role name="User">
         <special-subject type="ALL_AUTHENTICATED_USERS"/>
    </security-role>
</authorization-roles>
```

- However, the default JEE role information can be customized in `server.xml` by adding an authorization element in the example that follows. This example extends the default configuration by adding `user2` into the `Administrator` role and giving access to the FileViewer servlet.

```
<authorization-roles id="com.ibm.cics.wlp.defaultapp">
    <security-role name="User">
        <user name="user1"/>
        <user name="user2"/>
   </security-role>
    <security-role name="Administrator">
        <user name="user2"/>
    </security-role>
</authorization-roles>
```

  In this case, `user1` can access the Default servlet but not the FileViewer servlet and `user2` can access the Default servlet and the FileViewer servlet.

Results

You have successfully configured the CICS Default Web Application.

Manually tailoring server.xml

If you want to make manual changes to your `server.xml`, there are some basic configurations you can apply. Your CICS region user ID needs to have both read and write access to the `server.xml` file.

Configuring the HTTP endpoint

If you want web access to your application, update the `httpEndpoint` attribute with the host name and port numbers you require. For example:

```
<httpEndpoint host="winmvs2c.example.com" httpPort="28216" httpsPort="28217"
    id="defaultHttpEndpoint"/>
```

Use a port number that can be opened by the CICS region, either exclusively or as a shared port.

HTTPS is available only if SSL is configured as described in Configuring SSL (TLS) for a Liberty JVM server using a Java keystore.

For more information, see Liberty overview.

Adding features

Add the following features in the `<featureManager>` list of features.

- CICS feature `cicsts:core-1.0`. This feature installs the CICS system OSGi bundles into the Liberty framework. This feature is required to start a CICS integrated-mode Liberty JVM server. You must also define a SAF registry.
- CICS feature `cicsts:standard-1.0`. This feature is required to start a CICS standard-mode Liberty JVM server. The `cicsts:standard-1.0` feature does not have access to CICS resources by default. For more information see “CICS standard-mode Liberty: Java EE 7 Full Platform support without full CICS integration” on page 164.

  Note: Specify either the `cicsts:core-1.0` or `cicsts:standard-1.0` feature. You cannot specify both features in `server.xml`.

- CICS security feature `cicsts:security-1.0`. This feature installs the CICS system OSGi bundles that are required for CICS Liberty security into the Liberty framework. This feature is required when CICS external security is enabled (**SEC**=YES in the SIT) and you want security in the Liberty server. To use the `cicsts:security-1.0` feature, you must also configure a user registry. For more information, see "User registry" on page 168.
- `jsp-2.3`. This feature enables support for servlet and JavaServer Pages (JSP) applications. This feature is required by Dynamic Web Projects (WAR files) and OSGi Application Projects that contain OSGi Bundle Projects with Web Support that are installed as CICS bundles.
- `cicsts:jdbc-1.0`. This feature enables applications to access DB2 through the JDBC DriverManager or DataSource interfaces.

Example:

```
<featureManager>
    <feature>cicsts:core-1.0</feature>
    <feature>cicsts:security-1.0</feature>
    <feature>jsp-2.3</feature>
    <feature>cicsts:jdbc-1.0</feature>
</featureManager>
```

For more information, see Liberty features.

CICS bundle deployed applications

If you want to deploy Liberty applications that use CICS bundles, the `server.xml` file must include the entry:

```
<include location="${server.output.dir}/installedApps.xml"/>
```

The included file is used to define CICS bundle deployed applications.

Bundle repository

Share common OSGi bundles by placing them in a directory and referring to that directory in a `bundleRepository` element. For example:

```
<bundleRepository>
    <fileset dir="directory_path" include="*.jar"/>
</bundleRepository>
```

Global/shared library

Share common JAR files between web applications by placing them in a directory and referring to that directory in a global/shared library definition.

```
<library id="global">
    <fileset dir="directory_path" include="*.jar"/>
</library>
```

The global/shared libraries cannot be used by OSGi applications in an EBA, which must use a bundle repository. For more information, see Providing global libraries for all Java EE applications or Liberty: Shared libraries.

Liberty server application and configuration update monitoring

The Liberty JVM server scans the `server.xml` file for updates. By default, it scans every 500 milliseconds. To vary this value, add an entry such as:

```
<config monitorInterval="5s" updateTrigger="polled"/>
```

It also scans the `dropins` directory to detect the addition, update, or removal of applications. If you install your web applications in CICS bundles, disable the `dropins` directory as follows:

```
<applicationMonitor updateTrigger="disabled" dropins="dropins"
dropinsEnabled="false" pollingRate="5s"/>
```

For more information, see Controlling dynamic updates.

JTA transaction log

When the Java Transaction API (JTA) is used, the Liberty transaction manager stores its recoverable log files in the zFS filing system. The default location for the transaction logs is `${WLP_USER_DIR}/tranlog/`. This location can be overridden by adding a transaction element to `server.xml` such as

```
<transaction transactionLogDirectory="/u/cics/CICSPRD/DFHWLP/tranlog/"/>
```

CICS default web application

The CICS default web application **`CICSDefaultApp`** is a configuration service that validates the Liberty JVM server has started. To make the application available, add the JVM profile option `com.ibm.cics.jvmserver.wlp.defaultapp=true` to your JVM profile, or if you are not using autoconfigure, add the `cicsts:defaultApp-1.0` feature to `server.xml`. Run the application by using the URL `http://<server>:<port>/com.ibm.cics.wlp.defaultapp/`.

```
<featureManager>
    <feature>cicsts:defaultApp-1.0</feature>
</featureManager>
```

User registry

Unless you are using distributed identity mapping, you must define a SAF registry to use the CICS security feature:

```
<safRegistry id="saf"/>
```

For more information, see Configuring security for a Liberty JVM server by using distributed identity mapping or Java Database Connectivity 4.1.

CICS JTA integration

If an XA transaction is used by Liberty, the CICS unit-of-work becomes subordinate to the XA transaction by default. You can opt out of this automatic integration of CICS with JTA by setting the JVM profile option `com.ibm.cics.jvmserver.wlp.jta.integration=false`. Alternatively you can manually set the **`cicsts_jta`** element directly in your `server.xml`.

```
<cicsts_jta integration="false"/>
```

Modifying Lightweight Third-Party Authentication (LTPA) support

LTPA is configured by default when security is enabled for Liberty servers. LTPA enables web users to re-use their logged-in credentials across different applications or servers, using tokens signed by keys owned by the Liberty server. In secure deployment scenarios, you should modify the default password for the LTPA keys file to protect server security. You can also modify the expiry interval of the tokens, and change the default file location, which is required if sharing keys between multiple Liberty servers.

Auto-configuring a Db2 DataSource with type 2 connectivity through CICS using the `jdbc-4.0` or `jdbc-4.1` feature

Create a CICS-aware Db2® DataSource with type 2 connectivity by using the auto-configure property. This uses the `jdbc-4.0` feature if you use Java EE 6, or `jdbc-4.1` if you use Java EE 7.

Before you begin

Configure your CICS region to connect to Db2. For more information, see Defining the CICS DB2 connection.

About this task

You can create a Db2 DataSource with type 2 connectivity in the Liberty `server.xml`, which operates through the CICS DB2CONN, by using the JVM profile auto-configure property. The JNDI name is `jdbc/defaultCICSDataSource`.

If there is already a Db2 DataSource with type 2 connectivity with `id="defaultCICSDataSource"`, you are not able to use auto-configure to create a new one. For more information on creating one manually, see Manually configuring a Db2 DataSource with type 2 connectivity through CICS using the jdbc-4.0 or jdbc-4.1 feature.

Procedure

1. Enable auto-configure by setting **`-Dcom.ibm.cics.jvmserver.wlp.autoconfigure=true`** in the JVM profile.
2. Set the **`com.ibm.cics.jvmserver.wlp.jdbc.driver.location`** in the JVM profile to the location of the Db2 JDBC library.
   For example: `-Dcom.ibm.cics.jvmserver.wlp.jdbc.driver.location=/usr/lpp/db2v11/jdbc`
3. Install and enable the JVMSERVER resource.

Results

A Db2 DataSource with type 2 connectivity is added to the Liberty server configuration file, `server.xml`. This example uses the `jdbc-4.1` feature.

```
<featureManager>
    ...
    <feature>jdbc-4.1</feature>
</featureManager>
```

```
<dataSource id="defaultCICSDataSource" jndiName="jdbc/defaultCICSDataSource" transactional="false">
    <jdbcDriver libraryRef="defaultCICSDb2Library"/>
    <properties.db2.jcc driverType="2"/>
    <connectionManager agedTimeout="0"/>
</dataSource>

<library id="defaultCICSDb2Library">
    <fileset dir="/usr/lpp/db2v11/jdbc/classes" includes="db2jcc4.jar db2jcc_license_cisuz.jar"/>
    <fileset dir="/usr/lpp/db2v11/jdbc/lib" includes="libdb2jcct2zos4_64.so"/>
</library>
```

Manually configuring a Db2 DataSource with type 2 connectivity through CICS using the `jdbc-4.0` or `jdbc-4.1` feature

A CICS Liberty JVM server can be configured to use a JDBC DataSource with type 2 connectivity through CICS to access Db2 databases from Java applications.

Before you begin

You should configure your CICS region to connect to Db2. For more information, see Defining the CICS DB2 connection.

About this task

This task explains how to define the elements that are required in `server.xml` to enable JDBC type 2 driver connectivity to a local Db2 database.

Procedure

1. Add the `jdbc-4.1` or `jdbc-4.0` feature to the `server.xml` file.

   Use the `jdbc-4.0` feature if you use Java EE 6, or `jdbc-4.1` if you use Java EE 7.

   ```
   <featureManager>
       <feature>jdbc-4.1</feature>
   </featureManager>
   ```

2. Add a library element to the `server.xml` file to specify the location on zFS of the JDBC driver.

```
<library id="defaultCICSDb2Library">
    <fileset dir="/usr/lpp/db2v11/jdbc/classes" includes="db2jcc4.jar db2jcc_license_cisuz.jar"/>
    <fileset dir="/usr/lpp/db2v11/jdbc/lib" includes="libdb2jcct2zos4_64.so"/>
</library>
```

3. To access Db2 through a DataSource definition, a **`dataSource`** element is required. The **`jndiName`** attribute is required to define the JNDI name that is referenced by your application.

   You can set attributes for the **`dataSource`** by using a **`properties.db2.jcc`** element. The following example shows how to do this:

```
<dataSource id="defaultCICSDataSource" jndiName="jdbc/defaultCICSDataSource" transactional="false">
    <jdbcDriver libraryRef="defaultCICSDb2Library"/>
    <properties.db2.jcc driverType="2"/>
    <connectionManager agedTimeout="0"/>
</dataSource>
```

   `transactional="false"` is required to allow CICS to manage the transactions.

   `driverType="2"` is required to use the type 2 connectivity to Db2.

   `agedTimeout="0"` is required to disable Liberty connection pooling. Liberty connection pooling is not required as the DB2CONN resource provides Db2 connection pooling.

Results

The Liberty JVM server is configured to allow access to Db2 databases by using JDBC type 2 connectivity through a CICS DB2CONN resource.

Manually configuring a Db2 DataSource with type 4 connectivity through Liberty using the `jdbc-4.0` or `jdbc-4.1` feature

A CICS Liberty JVM server can be configured to use a JDBC DataSource with type 4 connectivity to access Db2 databases from Java applications.

Before you begin

The Liberty Db2 DataSource with type 4 connectivity does not use the CICS Db2 connection resource. However, if you do not have APARs PI18798 and PI18799 applied, you need to add the DB2 SDSNLOAD and SDSNLOD2 libraries to the CICS STEPLIB concatenation.

About this task

This task explains how to manually define the elements that are required in the `server.xml` configuration file to enable JDBC type 4 driver connectivity to a local or remote Db2 database. Updates that are made to a Db2 database that uses type 4 connectivity do not use the CICS Db2 connection resource. They are not part of a two-phase commit transaction unless the DataSource connection is of type `javax.sql.XADataSource`, and they are made within a JTA user transaction, see Java Transaction API (JTA).

Procedure

1. Add the `jdbc-4.0` or `jdbc-4.1` feature to the featureManager element. This enables use of the **`dataSource`** and **`jdbcDriver`** elements that are used later in the `server.xml`file.

   Use the `jdbc-4.0` feature if you use Java EE 6, or `jdbc-4.1` if you use Java EE 7.

   ```
   <featureManager>
       <feature>jdbc-4.1</feature>
   </featureManager>
   ```

2. Add **`dataSource`** and **`jdbcDriver`** elements. The **`dataSource`** element must refer to a library definition that specifies the library from which the JDBC driver components (the Db2 JDBC jar and native DLL files) are to be loaded. Typical definitions might look like this:

   ```
   <dataSource jndiName="jdbc/defaultCICSDataSource">
       <jdbcDriver libraryRef="db2Lib"/>
       <properties.db2.jcc driverType="4"
           serverName="winmvs2c.hursley.ibm.com"
           portNumber="41100"
           databaseName="DSNV11P2"
           user="DBUSER"
           password="{xor}Lz4sLCgwLTs="/>
   </dataSource>

   <library id="db2Lib">
       <fileset dir="/usr/lpp/db2v11/jdbc/classes" includes="db2jcc4.jar
           db2jcc_license_cisuz.jar" />
   </library>
   ```

   If you do not have APARs PI18798 and PI18799 applied, you need to add a fileset entry for the Db2 native library to the library configuration, for example:

   ```
   <fileset dir="/usr/lpp/db2v11/jdbc/lib" />
   ```

   The **`dataSource`** specifies the **`jndiName`** attribute that is referenced by your application program when you are establishing a connection to that data source. The required properties are set in the **`properties.db2.jcc`** element as follows:

   driverType
   Description: Database driver type, must be set to 4 to use the pure Java driver.

   Default value: 4

   Required: false

Data type: int

serverName
Description: The host name of server where the database is running. This is the SQL DOMAIN value of the Db2 DISPLAY DDF command.

Default value: localhost

Required: false

Data type: string

portNumber
Description: Port on which to obtain database connections. This is the TCPPORT value of the Db2 DISPLAY DDF command.

Default value: 50000

Required: false

Data type: int

databaseName
Description: specifies the name for the data source. This is the LOCATION value of the Db2 DISPLAY DDF command.

Required: true

Data type: string

user
Description: The user ID used to connect to the database.

Required: true

Data type: string

password
Description: The password of the user ID used to connect to the database. The value can be stored in clear text or encoded form. It is recommended that you encode the password. To do so, use the securityUtility tool with the encode option, see Liberty: securityUtility command.

Required: true

Data type: string

Results

The Liberty server, when started, is configured to allow access to Db2 databases through a JDBC type 4 connectivity. For more information, see Java Database Connectivity 4.1.

Manually configuring a Db2 DataSource or the DriverManager interface with type 2 connectivity through CICS using the `cicsts:jdbc-1.0` feature

A CICS Liberty JVM server can be configured to use JDBC type 2 connectivity through CICS, providing Java applications with either a **`javax.sql.DataSource`** or a **`java.sql.DriverManager`** interface to access Db2 databases.

Before you begin

Configure your CICS region to connect to Db2. For more information, see Defining the CICS DB2 connection.

About this task

Although you can configure CICS Liberty to access Db2 with type 2 connectivity using the **`cicsts:jdbc-1.0`** feature, the preferred method is to use the Liberty `jdbc-4.x` features. For more information, see Manually configuring a Db2 DataSource with type 2 connectivity through CICS using the

jdbc-4.0 or jdbc-4.1 feature. However, if you want to use the DriverManager interface, you must use the **cicsts:jdbc-1.0** feature as described in the following procedure.

Procedure

1. Add the `cicsts:jdbc-1.0` feature to the featureManager element.

   This enables use of the **cicsts_jdbcDriver** and **cicsts_dataSource** elements, used later in the `server.xml` file.

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

2. Add a **cicsts_jdbcDriver** element. This enables JDBC type 2 connectivity with the **java.sql.DriverManager** or **javax.sql.DataSource** interface.

   The **cicsts_jdbcDriver** element must refer to a library definition that specifies the library from which the JDBC driver components (the Db2 JDBC jar and native dll files) are to be loaded. Typical definitions might look like this:

```
<cicsts_jdbcDriver libraryRef="defaultCICSDb2Library"/>
<library id="defaultCICSDb2Library">
    <fileset dir="/usr/lpp/db2v11/jdbc/classes" includes="db2jcc4.jar db2jcc_license_cisuz.jar"/>
    <fileset dir="/usr/lpp/db2v11/jdbc/lib" includes="libdb2jcct2zos4_64.so"/>
</library>
```

   Note: Only one **cicsts_jdbcDriver** element is required. If more than one is specified, only the last **cicsts_jdbcDriver** element in the `server.xml` file is used and the others are ignored.

   If you require only **java.sql.DriverManager** support, the preceding steps are sufficient.

3. To access Db2 using the DataSource interface, a **cicsts_dataSource** element is required, but the preferred method for DataSource access is to use the Liberty `jdbc-4.x` features, as described in Manually configuring a Db2 DataSource with type 2 connectivity through CICS using the jdbc-4.0 or jdbc-4.1 feature. A **cicsts_dataSource** requires a `jndiName` attribute to define the JNDI name that is referenced by your application. A definition might look like this:

```
<cicsts_dataSource id="defaultCICSDataSource" jndiName="jdbc/defaultCICSDataSource"/>
```

   Tip: The DataSource class that is used is **com.ibm.db2.jcc.DB2SimpleDataSource**, which implements **javax.sql.DataSource**.

4. Optional: You can set attributes for the **cicsts_dataSource** by using a **properties.db2.jcc** element. The following example shows how to do this:

```
<cicsts_dataSource id="defaultCICSDataSource" jndiName="jdbc/defaultCICSDataSource">
    <properties.db2.jcc currentSchema="DB2USER" fullyMaterializeLobData="true" />
</cicsts_dataSource>
```

   Some of the attributes, which can be specified on the **properties.db2.jcc** element are not valid for DataSources with type 2 connectivity. If these invalid attributes are specified, they are ignored and a warning message is issued. The following attributes are not valid:

   - *driverType*
   - *serverName*
   - *portNumber*
   - *user*
   - *password*
   - *databaseName*

Results

The Liberty JVM server can connect to Db2 databases with JDBC type 2 connectivity through a CICS DB2CONN resource.

Note: Dynamic updates of the `cicsts_dataSource` and its components are not supported. Updating the configuration while the Liberty server is running can result in Db2 application failures. You should recycle the server to activate any changes.

CICS Liberty defaulted settings

When Liberty is configured, by default, two configuration settings are applied but are not visible in `server.xml`.

Defaulted configuration settings

If you wish to use different settings than the two non-visible defaulted settings, you can specify them in `server.xml`. The two settings are:

- **`<applicationManager autoExpand="true"/>`** This setting causes application file archives to be automatically expanded into the `${server.config.dir}/apps` directory on first use. This avoids expansion of file archives into the Liberty work area on server startup, reducing zFS file I/O and making more efficient use of the Java shared class cache. If you wish to override this setting and switch it off, then you should place the XML element: `<applicationManager autoExpand="false"/>` in your `server.xml` file.
- **`<transaction recoverOnStartup="true" waitForRecovery="true"/>`** These settings specify that following a server failure, transaction recovery should occur at server startup, and the server should wait for recovery to complete before permitting further transactional work. See Configuring the transaction service startup for more information.

Configuring a JVM server for Axis2

Configure the JVM server to run Axis2 if you want to run Java web services or process SOAP requests in a pipeline.

About this task

Axis2 is a Java SOAP engine that can process web service requests in provider and requester pipelines. When you configure a JVM server to run Axis2, CICS automatically adds the required JAR files to the class path.

You can define the JVM server either with CICS online resource definition or in a CICS bundle.

Procedure

1. Create a JVMSERVER resource for the JVM server.
   a) Specify a name for the JVM profile for the JVM server.

      On the JVMPROFILE attribute of JVMSERVER, specify a 1 - 8 character name. This name is used for the prefix of the JVM profile, which is the file that holds the configuration options for the JVM server. You do not need to specify the suffix, .jvmprofile, here.

   b) Specify the thread limit for the JVM server.

      On the THREADLIMIT attribute of JVMSERVER, specify the maximum number of threads that are allowed in the Language Environment enclave for the JVM server. The number of threads that are required depend on the workload that you want to run in the JVM server. To start with, you can accept the default value and then tune the environment. You can set up to 256 threads in a JVM server.
2. Create the JVM profile to define the configuration options for the JVM server.

You can use the sample profile, DFHJVMAX.jvmprofile, as a basis. This profile contains a subset of options that are suitable for starting the JVM server. All options and values for the JVM profile are described in "JVM profile validation and properties" on page 177. Follow the coding rules, including those for the profile name, in "Rules for coding JVM profiles" on page 177.

a) Set the location for the JVM profile.

   The JVM profile must be in the directory that you specify on the system initialization parameter, JVMPROFILEDIR. For more information, see "Setting the location for the JVM profiles" on page 155.

b) Make the following changes to the sample profile:

   - Set JAVA_HOME to the location of your installed IBM Java SDK.
   - Set JAVA_PIPELINE to run Axis2.
   - Set CLASSPATH_SUFFIX to specify classes for Axis2 applications and SOAP handlers that are written in Java.
   - Set WORK_DIR to your choice of destination directory for messages, trace, and output from the JVM server.
   - Set TZ to specify the timezone for timestamps on messages from the JVM server.An example for the United Kingdom is TZ=GMT0BST,M3.5.0,M10.4.0.

c) Save your changes to the JVM profile.

   The JVM profile must be saved as EBCDIC on the z/OS UNIX System Services file system.

3. Install and enable the JVMSERVER resource.

Results

CICS creates a Language Environment enclave and passes the options from the JVM profile to the JVM server. The JVM server starts up and loads the Axis2 JAR files. When the JVM server completes startup successfully, the JVMSERVER resource installs in the ENABLED state.

If an error occurs, for example CICS is unable to find or read the JVM profile, the JVM server fails to start. The JVMSERVER resource installs in the DISABLED state and CICS issues error messages to the system log.

What to do next

- Specify any directories that contain native C dynamic link library (DLL) files, such as DB2 or IBM MQ. You specify these directories on the LIBPATH_SUFFIX option in the JVM profile.
- Configure CICS to run web service requests in the JVM server, as described in Using Java with web services.

JVM profile example

Example JVM profile configured to start Axis2.

The following excerpt shows an example JVM profile that is configured to start Axis2:

```
#*********************************************************************
#
#                          Required parameters
#                          -------------------
#
# When using a JVM server, the set of CICS options that are supported
JAVA_HOME=/usr/lpp/java/J7.0_64
WORK_DIR=.
LIBPATH_SUFFIX=/usr/lpp/db2910/lib
...
#*********************************************************************
#
#                     JVM server specific parameters
#                     ------------------------------
#
JAVA_PIPELINE=YES
#
```

```
#*********************************************************************
#
#                              JVM options
#                              -----------
# The following option sets the Garbage collection Policy.
#
-Xgcpolicy:gencon
#
#*********************************************************************
#
#                 Setting user JVM system properties
#                 ----------------------------------
#
# -Dcom.ibm.cics.some.property=some_value
#
#*********************************************************************
#
#              Unix System Services Environment Variables
#              ------------------------------------------
#
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,SYSDUMP),ONINTERRUPT(NONE)"
#
#
```

Configuring a JVM server for a CICS Security Token Service

Configure the JVM server to run a CICS Security Token Service if you want to validate and process SAML tokens.

About this task

The supplied sample DFHJVMST.jvmprofile is suitable for a JVM server that runs a CICS Security Token Service.

You can define the JVM server either with CICS online resource definition or in a CICS bundle. For more help with using the CICS Explorer to create and edit resources in CICS bundles, see Working with bundles in the CICS Explorer product documentation.

Procedure

Create a JVMSERVER resource for the JVM server.

a) Specify a name for the JVM profile for the JVM server.

   On the JVMPROFILE attribute, specify a 1 - 8 character name. This name is used for the prefix of the JVM profile, which is the file that holds the configuration options for the JVM server. You do not need to specify the suffix .jvmprofile.

b) Specify the thread limit for the JVM server.

   The number of threads depends on the workload that you want to run in the JVM server. To start with, you can accept the default value and then tune the environment later. You can set up to 256 threads in a JVM server.

c) Create the JVM profile to define the configuration options for the JVM server.

   The JVM profile must be in the directory that you specify on the system initialization parameter, JVMPROFILEDIR. You can use the sample profile, DFHJVMST.jvmprofile, as a basis. This profile contains a subset of options that are suitable for starting the JVM server. You can either copy DFHJVMST.jvmprofile from the installation directory into the directory that you specify on JVMPROFILEDIR, or select it in CICS Explorer and save to the target directory.

   All options and values for the JVM profile are described in “JVM profile validation and properties” on page 177. Follow the coding rules in “Rules for coding JVM profiles” on page 177.

   Make the following changes to the sample profile:

   - Set JAVA_HOME to the location of your installed IBM Java SDK.
   - Set WORK_DIR to your choice of destination directory for messages, trace, and output from the JVM server.

- Set SECURITY_TOKEN_SERVICE to YES.
- Set TZ to specify the timezone for timestamps on messages from the JVM server. An example for the United Kingdom is TZ=GMT0BST,M3.5.0,M10.4.0.

d) Save your changes to the JVM profile

The JVM profile must be saved as EBCDIC on the USS file system.

Results

When you install and enable the JVMSERVER resource, CICS creates a Language Environment enclave and passes the options from the JVM profile to the JVM server. The JVM starts up and the OSGi framework resolves any OSGi middleware bundles. When the JVM server completes startup successfully, the JVMSERVER resource installs in the ENABLED state.

If an error occurs, for example CICS is unable to find or read the JVM profile, the JVM server fails to initialize. The JVMSERVER resource installs in the DISABLED state and CICS issues error message.

What to do next

You can further customize the JVM server, for example:

- Specify any directories that contain native C dynamic link library (DLL) files, such as DB2 or IBM MQ. You specify these directories on the LIBPATH_SUFFIX option.
- For more information see Configuring the CICS Security Token Service.

JVM profile validation and properties

JVM profiles contain a set of options and system properties that are passed to the JVM when it starts. Some JVM profile options are specific to the CICS environment and are not used for JVMs in other environments. CICS validates that the JVM profile is coded correctly when you start the JVM server.

The JVM options are described in “Options for JVMs in a CICS environment” on page 179. CICS provides sample profiles for each JVM server configuration that is supported by CICS. These sample profiles have default values for the most common JVM options. The sample profiles are stored in zFS in `/usr/lpp/cicsts/cicsts54/JVMProfiles/`.

You can also specify z/OS UNIX System Services environment variables in a JVM profile. Name and value pairs that are not valid JVM options are treated as z/OS UNIX System Services environment variables, and are exported. z/OS UNIX System Services environment variables specified in a JVM profile apply only to JVMs created with that profile.

Examples of environment variables include the `WLP_INSTALL_DIR` variable for the Liberty profile, and the TZ variable for changing the time zone of the JVM.

The Java class libraries include other system properties that you can set in a JVM profile. For example, applications might also have their own system properties. The IBM Java documentation is the primary source of Java information. For more information about the JVM system properties, see z/OS User Guide for IBM SDK, Java Technology Edition, Version 7.

Rules for coding JVM profiles

You can edit JVM profiles using any standard text editor. Follow these rules when coding your JVM profiles.

Name of a JVM profile

- The name can be up to eight characters in length.
- The name can be any name that is valid for a file in z/OS UNIX System Services. Do not use a name beginning with DFH, because these characters are reserved for use by CICS.
- Because JVM profiles are UNIX files, case is important. When you specify the name in CICS, you must enter it using the same combination of uppercase and lowercase characters that is present in the z/OS UNIX file name.

- JVM profiles on the file system must have the file extension `.jvmprofile`. The file extension is set to lowercase and must not be changed.

Directory

Do not use quotation marks when specifying values for directories in a JVM profile.

CEDA

The CEDA panels accept mixed case input for the JVMPROFILE field irrespective of your terminal UCTRAN setting. However, you must enter the name of a JVM profile in mixed case when you use CEDA from the command line or when you use another CICS transaction. Ensure that your terminal is correctly configured with uppercase translation suppressed. You can use the supplied CEOT transaction to alter the uppercase translation status (UCTRAN) for your own terminal, for the current session only.

Case sensitivity

All parameter keywords and operands are case-sensitive, and must be specified exactly as shown in "Options for JVMs in a CICS environment" on page 179 and "JVM system properties" on page 189.

Class path separator character

Use the `:` (colon) character to separate the directory paths that you specify on a class path option, such as `CLASSPATH_SUFFIX`.

Continuation

For JVM options the value is delimited by the end of the line in the text file. If a value that you are entering or editing is too long for an editor window, you can break the line to avoid scrolling. To continue on the next line, terminate the current line with the backslash character and a blank continuation character, as in this example:

```
CLASSPATH_SUFFIX=/u/example/pathToJarOrZipFile/jarfile.jar:\
/u/example/pathToRootDirectoryForClasses
```

Do not put more than one JVM option on the same line.

Comments

To add comments or to comment out an option instead of deleting it, begin each line of the comment with a # symbol. Comment lines are ignored when the file is read by the JVM launcher.

Blank lines are also ignored. You can use blank lines as a separator between options or groups of options.

The profile parsing code removes inline comments based on UNIX-like shell processing, so the documentation process of the sample JVM profile is improved. An inline comment is defined as follows:

- The comment starts with a # symbol
- It is preceded with one or more spaces (or tabs)
- It is not contained in quoted text

Table 32. Inline comment examples

Code	**Result**
`MYVAR=myValue # Comment`	MYVAR=myValue
`MYVAR=#myValue # Comment`	MYVAR=#myValue
`MYVAR=myValue "# Quoted comment" # Comment`	MYVAR=myValue "# Quoted comment"

Character escape sequences

You can code the escape sequences shown in Table 33 on page 179

Table 33. Escape sequences

Escape sequence	Character value
\b	Backspace
\t	Horizontal tab
\n	Newline
\r	Carriage return
\"	Double quotation mark
\'	Single quotation mark
\\	Backslash
xxx	The character corresponding to the octal value *xxx*, where *xxx* is between values 000 - 377
\uxxxx	The Unicode character with encoding *xxxx*, where *xxxx* is 1 - 4 hexadecimal digits. (See note for more information.)

Note: Unicode \u escapes are distinct from the other escape types. The Unicode escape sequences are processed before the other escape sequences described in Table 33 on page 179. A Unicode escape is an alternative way to represent a character that might not be displayable on non-Unicode systems. The character escapes, however, can represent special characters in a way that prevents the usual interpretation of those characters.

Multiple instances of options

If more than one instance of the same option is included in a JVM profile, the value for the last option found is used, and previous values are ignored.

Storage sizes

When specifying storage-related options in a JVM profile, specify storage sizes in multiples of 1024 bytes. Use the letter K to indicate KB, the letter M to indicate MB, and the letter G to indicate GB. For example, to specify 6 291 456 bytes as the initial size of the heap, code **-Xms** in one of the following ways:

```
-Xms6144K
-Xms6M
```

Options for JVMs in a CICS environment

The options in a JVM profile are used by CICS to start JVM servers. Some options are specific to CICS, but you can also specify environment variables and Java system properties.

Coding rules

When you specify JVM options, make sure that you follow the coding rules. For more information, see “Rules for coding JVM profiles” on page 177.

Format

The format of options can vary:

- Options in a JVM profile either take the form of a keyword and value, separated by an equal sign (=), for example `JAVA_PIPELINE=TRUE`, or they begin with a hyphen, for example `-Xmx16M`.

- Keyword value pairs are either CICS variables such as `JAVA_PIPELINE=TRUE`, or if not recognized as CICS options, they are treated as z/OS UNIX System Services environment variables, and are exported.
- Options that begin with `-D` in a JVM profile are JVM system properties. Options that begin with `-X` are treated as JVM command-line options. Any option that begins with `-` is passed to the JVM after any substitution symbols have been expanded. For more information, see "JVM system properties" on page 189.

Symbols used with JVM options

You can use substitution symbols in any variable or JVM server property specified in the JVM profile. The values of these symbols are determined at JVM server startup, so you can use a common profile for many JVM servers and CICS regions.

The following symbols are supported:

&APPLID;
When you use this symbol, the APPLID of the CICS region is substituted at run time. In this way, you can use the same profile for all regions, and still have region-specific working directories or output destinations. The APPLID is always in uppercase.

&CONFIGROOT;
When you use this symbol, the absolute path of the directory where the JVM profile is located is substituted at run time. For JVM servers that are defined in CICS bundles, the JVM profiles are by default located in the root directory for the bundle. For JVM servers that are defined by other methods, the JVM profiles are in the directory that is specified by the JVMPROFILEDIR system initialization parameter.

&DATE;
When you use this symbol, the symbol is replaced with the current date in the format *Dyymmdd* at run time.

&JVMSERVER;
When you use this symbol, the name of the JVMSERVER resource is substituted at run time. Use this symbol to create unique output or dump files for each JVM server.

&TIME;
When you use this symbol, the symbol is replaced with the JVM start time in the format *Thhmmss* at run time.

&USSHOME;
When you use this symbol, the symbol is replaced with the value of the USSHOME system initialization parameter. You can specify this symbol to automatically pick up the home directory for z/OS UNIX where CICS supplies its libraries for Java and the Liberty profile.

JVM server options

JVM server options, how they apply to different uses of a JVM server, and their descriptions are listed.

How options apply to different uses of a JVM server

The following table indicates whether an option is required, optional, or not supported for a particular use of a JVM server.

Table 34. JVM server options and how they apply to different uses of a JVM server

Option	OSGi	Liberty	Axis2	STS
_DFH_UMASK	Optional	Optional	Optional	Optional
CICS_WLP_MODE	Not supported	Optional	Not supported	Not supported
CLASSPATH_PREFIX	Not supported	Not supported	Optional	Optional
CLASSPATH_SUFFIX	Not supported	Not supported	Optional	Optional
DISPLAY_JAVA_VERSION	Optional	Optional	Optional	Optional

Table 34. JVM server options and how they apply to different uses of a JVM server (continued)				
Option	**OSGi**	**Liberty**	**Axis2**	**STS**
IDENTITY_PREFIX	Optional	Optional	Optional	Optional
JAVA_DUMP_TDUMP_PATTERN	Optional	Optional	Optional	Optional
JAVA_HOME	Required	Required	Required	Required
JAVA_PIPELINE	Not supported	Not supported	Required	Not supported
JNDI_REGISTRATION	Optional	Not supported	Optional	Optional
JVMTRACE	Optional	Optional	Optional	Optional
LIBPATH_PREFIX	Optional - use only under the guidance of IBM service personnel	Optional - use only under the guidance of IBM service personnel	Optional - use only under the guidance of IBM service personnel	Optional - use only under the guidance of IBM service personnel
LIBPATH_SUFFIX	Optional	Optional	Optional	Optional
LOG_FILES_MAX	Optional	Optional	Optional	Optional
LOG_PATH_COMPATIBILITY	Optional	Optional	Optional	Optional
OSGI_BUNDLES	Optional	Not supported	Not supported	Not supported
OSGI_CONSOLE	Optional	Not supported	Not supported	Not supported
OSGI_FRAMEWORK_TIMEOUT	Optional	Optional	Not supported	Not supported
PRINT_JVM_OPTIONS	Optional	Optional	Optional	Optional
PURGE_ESCALATION_TIMEOUT	Optional	Optional	Optional	Optional
SECURITY_TOKEN_SERVICE	Not supported	Not supported	Not supported	Required
STDERR	Optional	Optional	Optional	Optional
STDIN	Optional	Optional	Optional	Optional
STDOUT	Optional	Optional	Optional	Optional
USEROUTPUTCLASS	Optional	Not supported	Optional	Optional
WLP_INSTALL_DIR	Not supported	Required	Not supported	Not supported
WLP_LINK_TIMEOUT	Not supported	Optional	Not supported	Not supported
WLP_OUTPUT_DIR	Not supported	Optional	Not supported	Not supported
WLP_USER_DIR	Not supported	Optional	Not supported	Not supported
WLP_ZOS_PLATFORM	Not supported	Optional	Not supported	Not supported
WORK_DIR	Optional	Optional	Optional	Optional
WSDL_VALIDATOR	Optional	Not supported	Optional	Optional
ZCEE_INSTALL_DIR	Not supported	Optional	Not supported	Not supported

JVM server options and descriptions

Default values, where applicable, are the values that CICS uses when the option is not specified. The sample JVM profiles might specify a value that is different from the default value.

Note: You can still use options that are previously documented as YES|NO, but TRUE|FALSE is the preferred syntax.

_DFH_UMASK={007|nnn}

Sets the UNIX System Services process UMASK that applies when JVMSERVER files are created. This value is a three digit octal. For example, the default value of 007 allows the intended `read/write/execute` permissions of owner and group to be respected, while preventing `read/write/execute` being given to `other` when a file is created. The supplied value must fall within the range of `000` (least restrictive) to 777 (most restrictive). UMASK applies for the lifetime of the JVM.

CICS_WLP_MODE={INTEGRATED|STANDARD}

For a Liberty JVM server, choose the level of integration between CICS and Liberty.

Specify the INTEGRATED mode to use CICS integrated-mode Liberty. The Liberty JVM server runs with CICS enabled threads, respects CICS security, integrates with a CICS unit of work, and makes the Java class library for CICS (JCICS) API available for your Java web applications. If this option is omitted or not valid, the default of INTEGRATED is used.

Specify the STANDARD mode to use CICS standard-mode Liberty. The Liberty JVM server runs in a mode that is more standard to all Liberty supported platforms. This mode allows you to port and deploy your Liberty applications from other platforms to CICS without change. The JVM server retains control of the Liberty server and manages the server creation, lifecycle, and configuration. However, threads are not CICS enabled by default and do not run within a CICS transaction context. CICS unit of work integration, CICS security integration, and the JCICS API are not directly available from your Java application.

CLASSPATH_PREFIX, CLASSPATH_SUFFIX=*class_pathnames*

Use these options to specify directory paths, Java archive files, and compressed files to be searched by a JVM that is not OSGi enabled. For example, it is used for Java web services. Do not set a class path if you want to use an OSGi framework because the OSGi framework handles the class loading for you. If you use these options to specify the standard class path for Axis2, you must also specify JAVA_PIPELINE=TRUE to start the Axis2 engine.

CLASSPATH_PREFIX adds class path entries to the beginning of the standard class path, and CLASSPATH_SUFFIX adds them to the end of the standard class path. You can specify entries on separate lines by using a \ (backslash) at the end of each line that is to be continued.

Use the CLASSPATH_PREFIX option with care. Classes in CLASSPATH_PREFIX take precedence over classes of the same name that are supplied by CICS and the Java run time and the wrong classes might be loaded.

CICS builds a base class path for a JVM by using the `/lib` subdirectories of the directories that are specified by the **USSHOME** system initialization parameter and the JAVA_HOME option in the JVM profile. This base class path contains the Java archive files that are supplied by CICS and by the JVM. It is not visible in the JVM profile. You do not specify these files again in the class paths in the JVM profile.

Use a colon, not a comma, to separate multiple items that you specify by using the CLASSPATH_PREFIX or CLASSPATH_SUFFIX options.

DISPLAY_JAVA_VERSION={TRUE|FALSE}

If this option is set to TRUE, whenever a JVM is started by an application CICS writes message DFHSJ0901 to the MSGUSER log, showing the version and build of the IBM Software Developer Kit for z/OS, Java Technology Edition that is in use.

IDENTITY_PREFIX={TRUE|FALSE}

In order to establish the origin of JVM server output, all STDOUT, and STDERR entries that are routed to JES are written with a prefix string of the JVM server name, which is useful if multiple JVM servers are sharing a JES destination. This behavior can be disabled by setting IDENTITY_PREFIX=FALSE, this disables usage of the prefix string.

JAVA_DUMP_TDUMP_PATTERN=

A z/OS UNIX System Services environment variable that specifies the file name pattern to be used for transaction dumps (TDUMPs) from the JVM. Java TDUMPs are written to a data set destination in the event of a JVM abend.

JAVA_HOME=/usr/lpp/java/*javadir*/

Specifies the installation location for IBM 64-bit SDK for z/OS, Java Technology Edition in z/OS UNIX. This location contains subdirectories and Java archive files that are required for Java support.

The supplied sample JVM profiles contain a path that was generated by the **JAVADIR** parameter in the DFHISTAR CICS installation job. The default for the **JAVADIR** parameter is `java/J7.0_64/`, which is the default installation location for the IBM 64-bit SDK for z/OS, Java Technology Edition. This value produces a `JAVA_HOME` setting in the JVM profiles of `/usr/lpp/java/J7.0_64/`.

JAVA_PIPELINE={TRUE|FALSE}

Adds the required Java archive files to the class path so that a JVM server can support web services processing in Java standard SOAP pipelines. The default value is FALSE. If you set this value, the JVM server is configured to support Axis2 instead of OSGi. You can add more JAR files to the class path by using the `CLASSPATH` options.

Note: The options JAVA_PIPELINE=TRUE and SECURITY_TOKEN_SERVICE=TRUE are not compatible.

JNDI_REGISTRATION={TRUE|FALSE}

Specifies that the JNDI registration JAR files are automatically added to the JVM runtime environment to support the usage of the JNDI by Java applications. This option is ignored for Liberty JVM servers. It is possible to opt out of the automatic addition of these files by setting JNDI_REGISTRATION=FALSE. If this function is not required, opting out can prevent potential clashes with newer JAR files, can keep the JVM footprint smaller, and avoids unnecessary class loading.

JVMTRACE={{&APPLID;.&JVMSERVER;.}Dyyyymmdd.Thhmmss.dfhjvmtrc|*file_name*|JOBLOG|//DD:data_definition}

Specifies the name of the z/OS UNIX file or JES DD to which Java tracing is written during operation of a JVM server. If you do not set a value for this option, CICS automatically creates unique trace files for each JVM server.

If `JVMTRACE` is left to default or is a relative filename then the output location depends on the `LOG_PATH_COMPATIBILITY` option. If `LOG_PATH_COMPATIBILITY=FALSE`, the files are placed in the `WORK_DIR/applid/jvmserver` directory. If `LOG_PATH_COMPATIBILITY=TRUE`, the files are placed in the `WORK_DIR` directory.

If an absolute filename is specified for `JVMTRACE` then CICS will create any directories within the path that do not exist.

If the file exists, output is appended to the end of the file. To create unique output files for each JVM server, use the `JVMSERVER` and `APPLID` symbols in your file name, as demonstrated in the sample JVM profiles. If `JVMTRACE` is left to default then CICS uses the `APPLID` and `JVMSERVER` symbols, and the date and time stamp when the JVM server started to create unique output files.

To route to a JES DD, specify the data definition name from JES by using the syntax `//DD:data_definition`.

If this option is set to `JOBLOG`, `JVMTRACE` is routed to the current `stdout` location.

LIBPATH_PREFIX, LIBPATH_SUFFIX=*pathnames*

Specifies directory paths to be searched for native C dynamic link library (DLL) files that are used by the JVM, and that have the extension `.so` in z/OS UNIX. This includes files that are required to run the JVM and extra native libraries that are loaded by application code or services.

The base library path for the JVM is built automatically by using the directories that are specified by the **USSHOME** system initialization parameter and the `JAVA_HOME` option in the JVM profile. The base

library path is not visible in the JVM profile. It includes all the DLL files that are required to run the JVM and the native libraries that are used by CICS.

You can extend the library path by using the `LIBPATH_SUFFIX` option. This option adds directories to the end of the library path after the base library path. Use this option to specify directories that contain any additional native libraries that are used by your applications. Also, use this option to specify directories that are used by any services that are not included in the standard JVM setup for CICS. For example, the additional native libraries might include the DLL files that are required to use the DB2 JDBC drivers.

The `LIBPATH_PREFIX` option adds directories to the beginning of the library path before the base library path. Use this option with care. If DLL files in the specified directories have the same name as DLL files on the base library path, they are loaded instead of the supplied files.

Use a colon, not a comma, to separate multiple items that you specify by using the `LIBPATH_PREFIX` or `LIBPATH_SUFFIX` option.

DLL files that are on the library path for use by your applications must be compiled and linked with the `XPLink` option. Compiling and linking with the `XPLink` option provides optimum performance. The DLL files that are supplied on the base library path and the DLL files that are used by services such as the DB2 JDBC drivers are built with the `XPLink` option.

LOG_FILES_MAX={0|*number*}

Specifies the number of old log files that are kept on the system. A default setting of 0 ensures that all old versions of the log file are retained. You can change this value to specify how many old log files you want to remain on the file system.

If **LOG_PATH_COMPATIBILITY=TRUE**, **LOG_FILES_MAX** is ignored.

If **STDOUT**, **STDERR**, and **JVMTRACE** use the default scheme, or if customized, include the &DATE;.&TIME; pattern, then only the newest nn of each log type is kept on the system. If your customization does not include any variables, which make the output unique, then the files are appended to, and there is no requirement for deletion. The clean-up does not apply if the output variables are customized to route output to DD:// or JOBLOG. Special value 0 means do not delete.

LOG_PATH_COMPATIBILITY={TRUE|FALSE}

The default value for this behavior is LOG_PATH_COMPATIBILITY=FALSE, which provides a consolidated log output behavior. The new behavior places the **JVMSERVER** log files in the same output directory structure as used by existing subcomponents of the **JVMSERVER**, for example: the OSGi framework, and the Liberty server. To revert to behavior from previous releases, set the parameter to LOG_PATH_COMPATIBILITY=TRUE, and the **JVMSERVER** log directories are created in the original location.

OSGI_BUNDLES=*pathnames*

Specifies the directory path for middleware bundles that are enabled in the OSGi framework of an OSGi JVM server. These OSGi bundles contain classes to implement system functions in the framework, such as connecting to IBM MQ or DB2. If you specify more than one OSGi bundle, use commas to separate them.

OSGI_CONSOLE={TRUE|FALSE}

Adds the required OSGi bundles to the OSGi framework to enable the OSGi console. You must also set the following properties in the JVM profile: `-Dosgi.console=host:port` and `-Dosgi.file.encoding={ISO-8859-1|US-ASCII|ASCII}`. The default value is FALSE. If you want to look at the state of OSGi bundles and services, see Troubleshooting Java applications.

OSGI_FRAMEWORK_TIMEOUT={60|*number*}

Specifies the number of seconds that CICS waits for the OSGi framework to initialize or shut down before it times out. You can set a value in the range 1 - 60000 seconds. The default value is 60 seconds. If the OSGi framework takes longer to start than the specified number of seconds, the JVM server fails to initialize and a DFHSJ0215 message is issued by CICS. Error messages are also written

to the JVM server log files in zFS. If the OSGi framework takes longer to shut down than the specified number of seconds, the JVM server fails to shut down normally.

PRINT_JVM_OPTIONS={TRUE|FALSE}

If this option is set to TRUE, whenever a JVM starts, the options that are passed to the JVM at start are also printed to SYSPRINT. The output is produced every time a JVM starts with this option in its profile. You can use this option to check the contents of the class paths for a particular JVM profile, including the base library path and the base class path that are built by CICS, which are not visible in the JVM profile.

PURGE_ESCALATION_TIMEOUT={15|*time*}

Specifies the interval in seconds between the disable actions that CICS performs when a JVM server encounters a TCB failure or a runaway task. After each timeout, CICS escalates to the next disable action (for example, from phaseout to purge), until the JVM server has been recycled.

CICS performs the following steps in sequence:

1. CICS disables the JVMSERVER resource with the PHASEOUT option to allow existing work in the JVM to complete where possible and prevent new work from using the JVM.
2. If the PHASEOUT operation fails to disable the JVMSERVER within the interval specified by the PURGE_ESCALATION_TIMEOUT JVM server option, CICS escalates to the next disable action PURGE until the JVMSERVER is disabled.

   For a Liberty JVM server, there is a minimum of 60-second timeout from phaseout to purge.
3. If the PURGE operation fails to disable the JVMSERVER within the interval, CICS escalates to the next disable action FORCEPURGE.
4. If the FORCEPURGE operation fails to disable the JVMSERVER within the interval, CICS escalates to KILL.
5. After the JVMSERVER is successfully disabled, message DFHSJ1008 is issued.
6. CICS attempts to re-enable the resource to create a new JVM.

SECURITY_TOKEN_SERVICE={TRUE|FALSE}

If this option is set to TRUE, the JVM server can use security tokens. If this option is set to FALSE, Security Token Service support is disabled for the JVM server.

Note: The options SECURITY_TOKEN_SERVICE=TRUE and JAVA_PIPELINE=TRUE are not compatible.

STDERR={{&APPLID;.&JVMSERVER;.}Dyyyymmdd.Thhmmss.dfhjvmerr|*file_name*|JOBLOG|//DD:data_definition}

Specifies the name of the z/OS UNIX file or JES DD to which the `stderr` stream is redirected. If you do not set a value for this option, CICS automatically creates unique trace files for each JVM server.

If STDERR is left to default or is a relative filename then the output location depends on the LOG_PATH_COMPATIBILITY option. If LOG_PATH_COMPATIBILITY=FALSE, the files are placed in the WORK_DIR/applid/jvmserver directory. If LOG_PATH_COMPATIBILITY=TRUE, the files are placed in the WORK_DIR directory.

If an absolute filename is specified for STDERR then CICS will create any directories within the path that do not exist.

If the file exists, output is appended to the end of the file. To create unique output files for each JVM server, use the JVMSERVER and APPLID symbols in your file name, as demonstrated in the sample JVM profiles. If STDERR is left to default then CICS uses the APPLID and JVMSERVER symbols, and the date and time stamp when the JVM server started to create unique output files.

To route to a JES DD, specify the data definition name from JES by using the syntax //DD:data_definition.

If this option is set to JOBLOG, STDERR is routed to SYSOUT if defined, or to a dynamic SYSnnn if not.

If you specify the USEROUTPUTCLASS option on a JVM profile, the Java class that is named on that option handles the System.err requests instead. The z/OS UNIX file that is named by the STDERR option might still be used if the class named by the USEROUTPUTCLASS option cannot write data to its intended destination; for example when you use the supplied sample class com.ibm.cics.samples.SJMergedStream. You can also use the file if output is directed to it for any other reason by a class that is named by the USEROUTPUTCLASS option.

STDIN=*file_name*

Specifies the name of the z/OS UNIX file from which the stdin stream is read. CICS does not create this file unless you specify a value for this option.

STDOUT={{&APPLID;.&JVMSERVER;.}Dyyyymmdd.Thhmmss.dfhjvmout|*file_name*|JOBLOG|//DD:data_definition}

Specifies the name of the z/OS UNIX file or JES DD to which the stdout stream is redirected. If you do not set a value for this option, CICS automatically creates unique trace files for each JVM server.

If STDOUT is left to default or is a relative filename then the output location depends on the LOG_PATH_COMPATIBILITY option. If LOG_PATH_COMPATIBILITY=FALSE, the files are placed in the WORK_DIR/applid/jvmserver directory. If LOG_PATH_COMPATIBILITY=TRUE, the files are placed in the WORK_DIR directory.

If an absolute filename is specified for STDOUT then CICS will create any directories within the path that do not exist.

If the file exists, output is appended to the end of the file. To create unique output files for each JVM server, use the JVMSERVER and APPLID symbols in your file name, as demonstrated in the sample JVM profiles. If STDOUT is left to default then CICS uses the APPLID and JVMSERVER symbols, and the date and time stamp when the JVM server started to create unique output files.

To route to a JES DD, specify the data definition name from JES by using the syntax //DD:data_definition.

If this option is set to JOBLOG, STDOUT is routed to SYSPRINT if defined, or to a dynamic SYSnnn if not.

If you specify the USEROUTPUTCLASS option on a JVM profile, the Java class that is named on that option handles the System.out requests instead. The z/OS UNIX file that is named by the STDOUT option might still be used if the class named by the USEROUTPUTCLASS option cannot write data to its intended destination; for example when you use the supplied sample class com.ibm.cics.samples.SJMergedStream. You can also use the file if output is directed to it for any other reason by a class that is named by the USEROUTPUTCLASS option.

USEROUTPUTCLASS=*classname*

Specifies the fully qualified name of a Java class that intercepts the output from the JVM and messages from JVM internals. You can use this Java class to redirect the output and messages from your JVMs, and you can add time stamps and headers to the output records. This is not supported for Liberty. If the Java class cannot write data to its intended destination, the files that are named in the STDOUT and STDERR options might still be used.

Specifying the USEROUTPUTCLASS option has a negative effect on the performance of JVMs. For best performance in a production environment, do not use this option. However, this option can be useful to application developers who are using the same CICS region because the JVM output can be directed to an identifiable destination.

For more information about this class and the supplied samples, see Controlling the location for JVM stdout, stderr, JVMTRACE, and dump output.

WLP_INSTALL_DIR={&USSHOME;/wlp}

Specifies the installation directory of the Liberty profile technology. The Liberty profile is installed in the z/OS UNIX home for CICS in a subdirectory called wlp. The default installation directory is /usr/lpp/cicsts/cicsts54/wlp. Always use the &USSHOME; symbol to set the correct file path and append the wlp directory.

This environment variable is required if you want to start a Liberty JVM server. If you set this environment variable, you can also supply other environment variables and system properties to configure the Liberty JVM server. The environment variables are prefixed with WLP, and the system properties are described in "JVM system properties" on page 189.

WLP_LINK_TIMEOUT={30000|*number*}

Specifies the number of milliseconds that CICS waits to dispatch a request to invoke an application in the Liberty JVM server before it times out. If you specify 0, CICS will wait indefinitely. The default value is 30000 milliseconds. If the task has no been dispatched to the Liberty JVM server after the specified number of milliseconds, the EXEC CICS LINK command will fail and a DFHSJ1006 message is issued by CICS.

WLP_OUTPUT_DIR=$WLP_USER_DIR/servers

Specifies the directory that contains output files for the Liberty profile. By default, the Liberty profile stores logs, the work area, configuration files, and applications, for the server in a directory that is named after the server.

This environment variable is optional. If you do not specify it, CICS defaults to *$WORK_DIR/&APPLID;/&JVMSERVER;*/wlp/usr/servers, replacing the symbols with runtime values.

If this environment variable is set, the output logs and workarea are stored in $WLP_OUTPUT_DIR/*server_name*.

WLP_USER_DIR={&APPLID;/&JVMSERVER;/wlp/usr/|*directory_path*}

Specifies the directory that contains the configuration files for the Liberty JVM server.
This environment variable is optional. If you do not specify it, CICS uses *&APPLID;/&JVMSERVER;*/wlp/usr/ in the working directory, replacing the symbols with runtime values. Configuration files are written to servers/*server_name*.

WLP_ZOS_PLATFORM={TRUE|FALSE}

If this option is set to FALSE, the z/OS platform extensions in a Liberty JVM server are disabled. Use of the cicsts:security-1.0 and cicsts:distributedIdentity-1.0 features is not permitted in this mode.

WORK_DIR={.|/tmp|*directory_name*}

Specifies the working directory on z/OS UNIX that the CICS region uses for activities that are related to JVMSERVER. The CICS JVMSERVER uses this directory as the route of configuration and output. A period (.) is defined in the supplied JVM profiles, indicating that the home directory of the CICS region user ID is to be used as the working directory. This directory can be created during CICS installation. If the directory does not exist or if WORK_DIR is omitted, /tmp is used as the z/OS UNIX directory name.

You can specify an absolute path or relative path to the working directory. A relative working directory is relative to the home directory of the CICS region user ID. If you do not want to use the home directory as the working directory for activities that are related to Java, or if your CICS regions are sharing the z/OS user identifier (UID) and so have the same home directory, you can create a different working directory for each CICS region.

If you specify a directory name that uses the &APPLID; symbol (whereby CICS substitutes the actual CICS region APPLID), you can have a unique working directory for each region, even if all the CICS regions share the set of JVM profiles. For example, if you specify:

```
WORK_DIR=/u/&APPLID;/javaoutput
```

each CICS region that uses that JVM profile has its own working directory. Ensure that the relevant directories are created on z/OS UNIX, and that the CICS regions are given read, write, and run access to them.

You can also specify a fixed name for the working directory. In this situation, you must also ensure that the relevant directory is created on z/OS UNIX, and access permissions are given to the correct CICS regions. If you use a fixed name for the working directory, the output files from all the JVM

servers in the CICS regions that share the JVM profile are created in that directory. If you use fixed file names for your output files, the output from all the JVM servers in those CICS regions is appended to the same z/OS UNIX files. To avoid appending to the same files, use the JVMSERVER symbol and the APPLID symbols to produce unique output and dump files for each JVM server.

Do not define your working directories in the CICS installation directory on z/OS UNIX, which is the home directory for CICS files as defined by the **USSHOME** system initialization parameter.

WSDL_VALIDATOR={TRUE|FALSE}

Enables validation for SOAP requests and responses against their definition and schema. This option is ignored for Liberty JVM servers. For more information, see Validating SOAP messages. It is possible to turn off this option by setting WSDL_VALIDATOR=FALSE. Opting out can prevent potential clashes with newer JAR files, wasted storage, and slower start.

ZCEE_INSTALL_DIR={*<installation_directory>*}

Provides the location of the z/OS Connect Enterprise Edition feature installation. For z/OS Connect Enterprise Edition V2.0, the default is */usr/lpp/IBM/zosconnect/v2r0/runtime* . For z/OS Connect Enterprise Edition V3.0, the default is */usr/lpp/IBM/zosconnect/v3r0/runtime* .

JVM command-line options

JVM command-line options, with descriptions.

List of command-line options

Note: This list is not exhaustive. It is a list of useful IBM® JVM options. Options that include -X are specific to the IBM JVM.

-agentlib

Specifies whether debugging support is enabled in the JVM.

For more information, see Debugging a Java application. For more information about the Java Platform Debugger Architecture (JPDA), see Oracle Technology Network Java website.

-Xcompressedrefs

Java 1.7.1 sets compressed references by default. This setting instructs the virtual machine (VM) to store all references to objects, classes, threads, and monitors as 32-bit values, rather than 64-bit values. The use of compressed references improves the performance of many applications because objects are smaller, resulting in less frequent garbage collection, and improved memory cache usage. However, this is at the expense of a large initial allocation of 31-bit storage.

Before Java 1.7.1, the use of compressed references was optional. To balance the use of `-Xcompressedrefs` in a JVM server, and to offset the large initial 31-bit storage allocation, a JVM server automatically sets the `-XXnosuballoc32bitmem` option. The effect of this option is to avoid a large initial allocation in favor of incremental allocations as required. For many applications, this behavior is an adequate balance between performance and storage use. For applications that use many references, reducing the available 31-bit storage (or if operating within a 31-bit storage constrained environment) then the use of `-Xnocompressedrefs` might be preferable - consider using this option if you are constrained on 31-bit storage.

-Xnocompressedrefs

The use of `-Xnocompressedrefs` might be preferable for applications that use many references that reduce the available 31-bit storage (or if operating within a 31-bit storage constrained environment).

-Xms

Specifies the initial size of the heap. Specify storage sizes in multiples of 1024 bytes. Use the letter K to indicate KB, the letter M to indicate MB, and the letter G to indicate GB. For example, to specify 6,291,456 bytes as the initial size of the heap, code **-Xms** in one of the following ways:

```
-Xms6144K
-Xms6M
```

Specify *size* as a number of KB or MB. For information, see JVM command-line options in IBM SDK, Java Technology Edition 7.0.0.

-Xmso

Sets the initial stack size for operating system threads.

For more information about the **-Xmso** JVM option and the default value, see JVM command-line options in IBM SDK, Java Technology Edition 7.0.0.

-Xmx

Specifies the maximum size of the heap. This fixed amount of storage is allocated by the JVM during JVM initialization.

Specify *size* as a number of KB or MB.

-Xscmx

Specifies the size of the shared class cache. The minimum size is 4 KB: the maximum and default sizes are platform-dependent.

Specify *size* as a number of KB or MB. For information, see JVM command-line options in IBM SDK, Java Technology Edition 7.0.0.

-Xshareclasses

Specify this option to enable class data sharing in a shared class cache. The JVM connects to an existing cache or creates a cache if one does not exist. You can have multiple caches and you can specify the correct cache by adding a suboption to the **-Xshareclasses** option. For more information, see Class data sharing between JVMs in IBM SDK, Java Technology Edition 7.0.0.

-XX:[+|-]EnableCPUMonitor

Note: This option will only take effect for Java 8.

This defaults to **-XX:-EnableCPUMonitor** when running in a JVM server, however, if you want to use the enhanced JMX CPU-monitoring capabilities, it should be set to **-XX:+EnableCPUMonitor**. Enabling this option will incur an increased CPU usage.

JVM system properties

JVM system properties provide configuration information specific to the JVM and its runtime environment. You provide JVM system properties by adding them to the JVM profile. At run time, CICS reads the properties from the JVM profile, and passes them to the JVM.

Property prefix

System properties must be set by using a -D prefix, for example the correct syntax for **com.ibm.cics** is **-Dcom.ibm.cics**.

com.ibm.cics indicates that the property is specific to the IBM JVM in a CICS environment.
com.ibm indicates a general JVM property that is used more widely.
java.ibm also indicates a general JVM property that is used more widely.

For information about general properties, see “JVM profile validation and properties” on page 177.

Property coding rules

Properties must be specified according to a set of coding rules. For more information about the rules, see “Rules for coding JVM profiles” on page 177.

Applicability of properties to different uses of JVM server

The three types of JVM server are OSGi, Liberty, and Classpath. Classpath JVM servers can be further refined to Axis2 capable, Security Token Server (STS) capable, Batch capable, and Mobile capable. The following table shows the options that apply to each specific capability. The table also indicates whether or not a property is supported for a particular use of a JVM server. Be aware that some properties are

read-only. Changing a read-only property might result in runtime environment failure. For details about these properties, see "Read-only properties" on page 192.

Table 35. Options by JVM server use

System property	OSGi	Liberty	Classpath
com.ibm.cics.json.enableAxis2Handlers	Not supported	Not supported	Supported
com.ibm.cics.jvmserver.applid	Supported	Supported	Supported
com.ibm.cics.jvmserver.cics.product.name	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.cics.product.version	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.configroot	Supported	Supported	Supported
com.ibm.cics.jvmserver.controller.timeout	Supported	Supported	Not supported
com.ibm.cics.jvmserver.local.ccsid	Supported	Supported	Supported
com.ibm.cics.jvmserver.name	Supported	Supported	Supported
com.ibm.cics.jvmserver.override.ccsid	Supported	Supported	Supported
com.ibm.cics.jvmserver.supplied.ccsid	Supported	Supported	Supported
com.ibm.cics.jvmserver.threadjoin.timeout	Supported	Supported	Not supported
com.ibm.cics.jvmserver.trace.filename	Supported	Supported	Supported
com.ibm.cics.jvmserver.trace.format	Supported	Supported	Supported
com.ibm.cics.jvmserver.trace.specification	Supported	Supported	Supported
com.ibm.cics.jvmserver.trigger.timeout	Supported	Supported	Not supported
com.ibm.cics.jvmserver.unclassified.tranid	Supported	Supported	Not supported
com.ibm.cics.jvmserver.unclassified.userid	Supported	Supported	Not supported
com.ibm.cics.jvmserver.wlp.args	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.wlp.autoconfigure	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.wlp.defaultapp	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.wlp.install.dir	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.wlp.jdbc.driver.location	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.wlp.jta.integration	Not supported	Supported	Not supported

Table 35. Options by JVM server use (continued)

System property	OSGi	Liberty	Classpath
com.ibm.cics.jvmserver.wlp.latebinding	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.wlp.optimize.static.resources	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.wlp.optimize.static.resources.extra	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.wlp.reserve.thread.percentage	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.wlp.server.config.dir	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.wlp.server.host	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.wlp.server.http.port	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.wlp.server.https.port	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.wlp.server.name	Not supported	Supported	Not supported
com.ibm.cics.jvmserver.wlp.server.output.dir	Not supported	Supported	Not supported
com.ibm.cics.sts.config	Not supported	Not supported	Supported (STS only)
com.ibm.ws.logging.console.log.level	Not supported	Supported	Not supported
com.ibm.ws.zos.core.angelName	Not supported	Supported	Not supported
com.ibm.ws.zos.core.angelRequired	Not supported	Supported	Not supported
console.encoding	Supported	Supported	Supported
file.encoding	Supported	Supported	Supported
java.security.manager	Supported	Not supported	Supported
java.security.policy	Supported	Not supported	Supported
org.osgi.framework.storage.clean	Supported	Supported	Not supported
org.osgi.framework.system.packages.extra	Supported	Supported	Not supported
osgi.compatibility.bootdelegation	Supported	Supported	Not supported

Read-only properties

com.ibm.cics.json.enableAxis2Handlers
Indicates that a JVM requires the ability to run Axis2 handler programs when processing JSON data. This property is only relevant to a JVM that has JAVA_PIPELINE=YES specified and is configured to support JSON pipelines. This option is not relevant to z/OS Connect in CICS, and should only be enabled if the capability is required. Enabling this option will ensure that Axis2 Handler programs can run during a JSON workload, but there is likely to be a performance penalty in enabling this option, and some of the capabilities of mapping level 4.2 and later WSBind files will not be available for use.

com.ibm.cics.jvmserver.applid
Specifies the CICS region application identifier (APPLID). This is a read-only property. You can use this property in an application but you should not change it.

com.ibm.cics.jvmserver.cics.product.name
Specifies the name of the CICS product under which Liberty is running. This is a read-only property. You can use this property in an application but you should not change it.

com.ibm.cics.jvmserver.cics.product.version
Specifies the version of the CICS product under which Liberty is running. This is a read-only property. You can use this property in an application but you should not change it.

com.ibm.cics.jvmserver.configroot
Specifies the location where configuration files, such as the JVM profile of a JVM server, can be found. This is a read-only property. You can use this property in an application but you should not change it.

com.ibm.cics.jvmserver.local.ccsid
Specifies the code page for file encoding when the JCICS API is used. This is a read-only property. You can use this property in an application but you should not change it.

com.ibm.cics.jvmserver.name
Specifies the name of the JVM server. This is a read-only property. You can use this property in an application but you should not change it.

com.ibm.cics.jvmserver.supplied.ccsid
Specifies the default CCSID for the local CICS region. This is a read-only property. You can use this property in an application but you should not change it.

com.ibm.cics.jvmserver.trace.filename
Specifies the name of the JVM server trace file. This is a read-only property. You can use this property in an application but you should not change it.

com.ibm.cics.jvmserver.wlp.install.dir
Specifies the location of the Liberty installation. This is a read-only property. You can use this property in an application but you should not change it.

com.ibm.cics.jvmserver.wlp.server.config.dir
Specifies the location of the Liberty configuration directory. This is a read-only property. You can use this property in an application but you should not change it.

com.ibm.cics.jvmserver.wlp.server.output.dir
Specifies the location of the Liberty output directory where you can find Liberty logs. This is a read-only property. You can use this property in an application but you should not change it.

Properties that can be changed

com.ibm.cics.jvmserver.controller.timeout={*time*|90000ms}
Only use numerical characters when changing this value.

Warning: This property is subject to change at any time.

com.ibm.cics.jvmserver.override.ccsid=

Warning: This property is intended for advanced users.

It overrides the code page for file encoding when the JCICS API is used. By default, JCICS uses the value of the **LOCALCCSID** system initialization parameter as the file encoding. If you choose to override this value, set the code page in this property. Use an EBCDIC code page. You must ensure that your applications are consistent with the new code page, or errors might occur. For more information about valid CCSIDs, see LOCALCCSID system initialization parameter.

com.ibm.cics.jvmserver.threadjoin.timeout={*time*|30000ms}
Controls the timeout value when requests that are waiting for threads are queuing for service.Only use numerical characters when changing this value.

com.ibm.cics.jvmserver.trace.format={FULL|SHORT|ABBREV}
Controls the format of the trace which can be varied for your own purposes. You must set it to SHORT when you send diagnostic information to IBM service.

com.ibm.cics.jvmserver.trace.specification={filter_text}

Warning: Use this property only under IBM service guidance. This property is subject to change at any time.

Specifies a JVM server trace filter string allowing finer grained control over package and class trace from the JVM server. {filter_text} is a colon separated string of clauses that sets the trace level of one or more specified components. If not specified, the default value is equivalent to **com.ibm.cics.*=ALL**.

The SJ domain trace flag remains the main switch, but this trace specification allows for additional filtering of specific components.

For any given class or package, the most specific filter clause applies. Each filter clause can be set to one of the following levels:

{ALL, DEBUG, ENTRYEXIT, EVENT, INFO, WARNING, ERROR, NONE}

Example 1:

```
com.ibm.cics.jvmserver.trace.specification=com.ibm.cics.*=NONE
```

A single filter clause that suppresses all output.

Example 2:

```
com.ibm.cics.jvmserver.trace.specification=com.ibm.cics.*=NONE:com.ibm.cics.wlp.*=ALL
```

This example has two filter clauses. The first filter clause suppresses all trace. The second filter clause is more specific for all packages under the **com.ibm.cics.wlp** component and ensures all of their trace output is written.

Example 3:

```
com.ibm.cics.jvmserver.trace.specification=com.ibm.cics.wlp.impl.CICSTaskWrapper=NONE:com.ib
m.cics.wlp.impl.CICSTaskInterceptor=NONE
```

This example has two filter clauses. All trace is written, except trace that is produced from the specific CICSTaskWrapper and CICSTaskInterceptor classes of the **com.ibm.cics.wlp.impl** package.

com.ibm.cics.jvmserver.trigger.timeout={*time*|500ms}
Only use numerical characters when changing this value.

Warning: Use this property only under IBM service guidance. This property is subject to change at any time.

com.ibm.cics.jvmserver.unclassified.tranid={transaction id}
Specifies the default transaction that is used for unclassified work that is run in a JVM server.

- In a Liberty JVM server, unclassified work runs under transaction CJSU, unless you specify the `com.ibm.cics.jvmserver.unclassified.tranid` property.

- In an OSGI JVM server, unclassified work runs under transaction CJSA, unless you specify the `com.ibm.cics.jvmserver.unclassified.tranid` property.

The user must ensure that the transaction ID specified is defined to CICS, by duplicating the CJSA or CJSU transaction.

com.ibm.cics.jvmserver.unclassified.userid={user id}
Allows users to change the default user ID under which unclassified work is run as a CICS task in a JVM server. If not specified, the CICS default user ID is used. The user ID specified must be defined to RACF® and have the necessary permissions to run the work.

Unclassified work is any request not identified by the HTTP classification component of Liberty, for example JMS, inbound JCA, EJB requests and so on.

com.ibm.cics.jvmserver.wlp.args=

Provides a way to set Liberty server options during start-up. For a list of server options, see the 'options' section in Server command options.

Warning: This property is typically used under IBM service guidance.

com.ibm.cics.jvmserver.wlp.autoconfigure={false|true}
Specifies whether CICS automatically creates and updates the `server.xml` file for the Liberty JVM server. If you set this property to true, CICS creates the necessary Liberty directory structure and configuration files. CICS also updates the `server.xml` file if you provide values for other Java properties, such as an HTTP port number.

com.ibm.cics.jvmserver.wlp.defaultapp={false|true}

Instructs CICS to add the `defaultApp-1.0` feature to `server.xml`, which installs the default CICS web application that can be used to verify that the Liberty server has installed and started correctly.

Tip: This property is used only when **com.ibm.cics.jvmserver.wlp.autoconfigure=true**.

com.ibm.cics.jvmserver.wlp.jdbc.driver.location=
Specifies the location of the directory that contains the DB2 JDBC drivers. The location must contain the DB2 JDBC driver `classes` and `lib` directories. If the autoconfigure property **com.ibm.cics.jvmserver.wlp.autoconfigure=true**, when the JVM server is enabled, the existing example configuration in `server.xml` is replaced with the default configuration and any user updates are lost.

com.ibm.cics.jvmserver.wlp.jta.integration={false|true}
Enables CICS integration with the Java Transaction API (JTA). When transactions that are created through the JTA interface are in effect, the CICS unit of work is subordinate to the Java Transaction Manager.

Tip: This property is used only when **com.ibm.cics.jvmserver.wlp.autoconfigure=true**.

com.ibm.cics.jvmserver.wlp.latebinding={NONHTTP|COMPATIBILITY}

Warning: Use this property only under IBM service guidance. This property is subject to change at any time.

com.ibm.cics.jvmserver.wlp.optimize.static.resources={true|false}

Enables requests for static content to be processed on a non-CICS thread. The following types of file are recognized as static: `.css` `.gif` `.ico` `.jpg` `.jpeg` `.js` and `.png`.

com.ibm.cics.jvmserver.wlp.reserve.thread.percentage={percent|10}

Reserves a percentage of the threadlimit of the Liberty JVM server for use by OSGi Applications. The value can be between 1% and 50%.

com.ibm.cics.jvmserver.wlp.optimize.static.resources.extra=

Specifies a custom list of extra static resources for optimization. Items must be comma-separated, and begin with a period, for example: `.css`, `.gif`, `.ico`.

Tip: This value is only respected when **com.ibm.cics.jvmserver.wlp.optimize.static.resources=true**.

com.ibm.cics.jvmserver.wlp.server.host={*|*hostname*|*IP_address*}
Specifies the name or IP address in IPv4 or IPv6 format of the host for HTTP requests to access the web application. The Liberty JVM server uses * as the default value. This value is not appropriate for running a web application in CICS, so either use this property to provide a different value or update the `server.xml` file.

Tip: This property is used only when **com.ibm.cics.jvmserver.wlp.autoconfigure=true**.

com.ibm.cics.jvmserver.wlp.server.http.port={9080|*port_number*}
Specifies a port to accept HTTP requests for a Java web application. CICS uses the default value that is supplied by Liberty. The Liberty JVM server does not use a TCPIPSERVICE resource. Ensure that the port number is free or shared on the z/OS system.

Tip: This property is used only when **com.ibm.cics.jvmserver.wlp.autoconfigure=true**.

com.ibm.cics.jvmserver.wlp.server.https.port={9443|*port_number*}
Specifies a port to accept HTTPS requests for a Java web application. CICS uses the default value that is supplied by Liberty. The Liberty JVM server does not use a TCPIPSERVICE resource, so ensure that the port number is free or shared on the z/OS system.

Tip: This property is used only when **com.ibm.cics.jvmserver.wlp.autoconfigure=true**.

com.ibm.cics.jvmserver.wlp.server.name={defaultServer|*server_name*}
Specifies the name of the Liberty server. You should not need to specify this property as it affects the location of the Liberty server configuration and output files and directories.

com.ibm.cics.sts.config=*path*
Specifies the location and name of the STS configuration file.

com.ibm.ws.logging.console.log.level={INFO|AUDIT|WARNING|ERROR|OFF}
Controls which messages Liberty writes to the JVM server stdout file. Liberty console messages are also written to the Liberty `messages.log` file independent of the setting of this property.

com.ibm.ws.zos.core.angelName=*named_angel*
Specifies a named angel process for the Liberty JVM server to connect to. If you do not specify **com.ibm.ws.zos.core.angelName**, when required, the default angel process is used for Liberty JVM server startup.

com.ibm.ws.zos.core.angelRequired={false|true}
Indicates whether an angel process is required for Liberty JVM server startup.

console.encoding=
Specifies the encoding for JVM server output files.

file.encoding=
Specifies the code page for reading and writing characters by the JVM. By default, a JVM on z/OS uses the EBCDIC code page IBM1047 (or cp1047).

- In a profile that is configured for OSGi, you can specify any code page that is supported by the JVM. CICS tolerates any code page because JCICS uses the local CCSID of the CICS region for its character encoding.
- In a profile that is configured for the Liberty JVM server, the supplied default value is ISO-8859-1. You can also use UTF-8. Any other code page is not supported.
- In a profile that is configured for Axis2, you must specify an EBCDIC code page.

java.security.manager={default| "" | |*other_security_manager*}
Specifies the Java security manager to be enabled for the JVM. To enable the default Java security manager, include this system property in one of the following formats:

- `java.security.manager=default`
- `java.security.manager=""`
- `java.security.manager=`

All these statements enable the default security manager. If you do not include the **java.security.manager** system property in your JVM profile, the JVM runs with Java security disabled.

java.security.policy=
Describes the location of extra policy files that you want the security manager to use to determine the security policy for the JVM. A default policy file is provided with the JVM in `/usr/lpp/java/J7.0_64/lib/security/java.policy`, where the `java/J7.0_64` subdirectory names are the default values when you install the IBM 64-bit SDK for z/OS, Java Technology Edition. The default security manager always uses this default policy file to determine the security policy for the JVM, and you can use the **java.security.policy** system property to specify any policy files that you want the security manager to take into account, in addition to the default policy file.

To enable CICS Java applications to run successfully when Java security is active, specify, as a minimum, an extra policy file that gives CICS the permissions it requires to run the application.

For information about enabling Java security, see Enabling a Java security manager.

org.osgi.framework.storage.clean={onFirstInit}
This option is specific to OSGi-enabled JVM servers, including Liberty. It specifies if and when the storage area for the OSGi framework should be cleaned. If no value is specified, the framework storage area will not be cleaned. **onFirstInit** flushes the bundle cache when the framework instance is first initialized. i.e when the JVM server is enabled. Framework storage cleaning is not necessary under normal operations.

org.osgi.framework.system.packages.extra=
This option is specific to OSGi-enabled JVM servers, including Liberty, which allows extensions of the JRE and custom Java packages to be exposed through the OSGi framework for subsequent bundle import resolution. JVM vendors might provide different extensions in the JRE. In an IBM JVM server, the option is augmented to include the set of packages which CICS chooses to expose from the IBM JRE. You can set this property to define additional packages, if required. For further information, see OSGi Alliance Specifications.

osgi.compatibility.bootdelegation={true|false}
This option is specific to the Equinox implementation of OSGi. It applies to OSGi-enabled JVM servers, including Liberty. When set to *true*, the OSGi framework employs a last resort bootdelegation strategy for packages that are not found through the normal OSGi bundle dependency resolution mechanism. This option allows the OSGi run time to be more tolerant if explicit dependencies were overlooked at development time. As a last resort algorithm, a small amount of overhead is incurred compared to direct resolution (where the package is explicitly listed in the Import-Package bundle header).

For strict OSGi compliance, increased portability, and optimum performance, set this option to *false* and ensure all the packages that are used in your OSGi bundles are explicitly declared in the bundle `MANFEST.MF`.

Setting the time zone

The TZ environment variable specifies the 'local' time of a system. You can set this for a JVM server by adding it to the JVM profile. If you do not set the TZ variable, the system defaults to UTC. Once the TZ variable is set, a JVM automatically transitions to and from daylight savings time as required, without a restart or further intervention.

When setting the time zone for a JVM server, you should be aware of the following issues:

- The TZ variable in your JVM profile should match your local MVS system offset from GMT.
- Customized time zones are not supported and will result in failover to UTC or a mixed time zone output in the JVMTRACE file.
- If you see LOCALTIME as the time zone string, there is an inconsistency in your configuration. This can be between your local MVS time and the TZ you are setting, or between your local MVS time and your default setting in the JVM profile. The output will be in mixed time zones although each entry will be correct.

Using the POSIX time zone format

The POSIX time zone format has a short form and a long form. You can use either to set the TZ environment variable, but using the short form reduces the chance of input errors.

Long form examples with daylight saving (Greenwich Mean Time, Central European Time, Eastern Standard Time):

```
TZ=GMT0BST,M3.5.0,M10.4.0
TZ=CET-1CEST,M3.5.0,M10.5.0
TZ=EST5EDT,M3.2.0,M11.1.0
```

Short form examples with daylight saving (Greenwich Mean Time, Central European Time, Eastern Standard Time):

```
TZ=GMT0BST
TZ=CET-1CEST
TZ=EST5EDT
```

Examples with no daylight saving (Malaysian Time, China Standard Time, Singapore Time):

```
TZ=MYT-8
TZ=CST-8
TZ=SGT-8
```

To find out what time zone your system is running on, log on to USS and enter `echo $TZ`. The result is the long form of the value your TZ environment variable should be set to.

```
/u/user:>echo $TZ
GMT0BST,M3.5.0,M10.4.0
```

For a more detailed breakdown of the POSIX time zone format, see POSIX and Olson time zone formats on the IBM developerWorks site.

Chapter 5. Administering Java applications

After you have enabled your Java applications, you can monitor the CICS region to understand how the applications are performing. You can tune the environment to optimize the performance of the application.

About this task

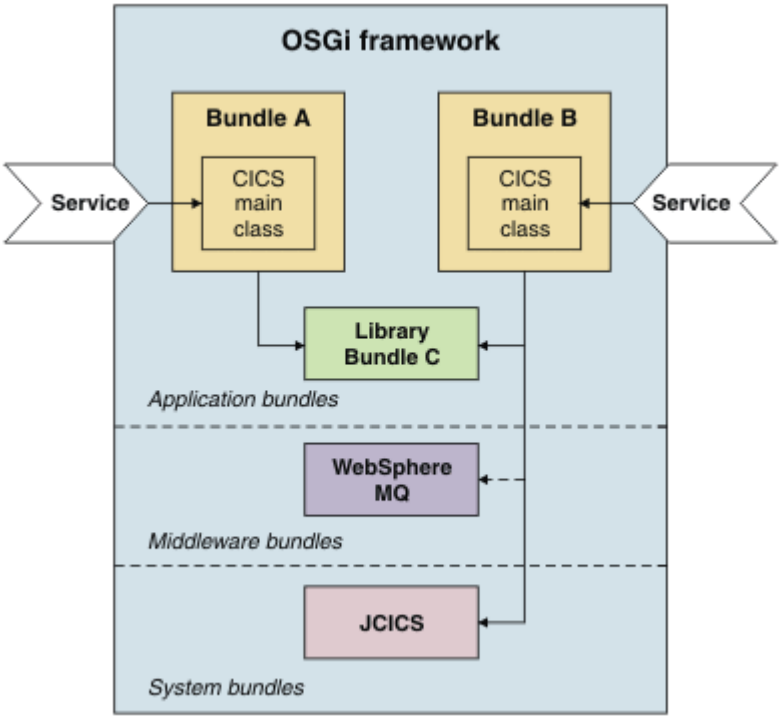
You can use statistics and monitoring to gather information about how the Java applications are performing in the CICS region. In particular, you can check how the JVMs are performing. After you gather the information, you can make changes to a JVM or a Language Environment enclave to improve performance.

Updating OSGi bundles in a JVM server

The process for updating OSGi bundles in the OSGi framework depends on the type of bundle and its dependencies. You can update OSGi bundles for applications without restarting the JVM server. However, updating a middleware bundle requires a restart of the JVM server.

About this task

In a typical JVM server, the OSGi framework contains a mixture of OSGi bundles as shown in the following diagram.



Bundle A and Bundle B are separate Java applications that are packaged as OSGi bundles in separate CICS bundles. Both applications have a dependency on a common library that is packaged in Bundle C. Bundle C is separately managed and updated. In addition, Bundle B has a dependency on an IBM MQ middleware bundle and the JCICS system bundle.

Bundle A and B can both be independently updated without affecting any of the other bundles in the framework. However, updating Bundle C can affect both the bundles that depend on it. Any exported packages in Bundle C remain in memory in the OSGi framework, so to pick up changes in Bundle C, Bundles A and B also have to be updated in the framework.

Middleware bundles contain framework services and are managed with the life cycle of the JVM server. For example, you might have native code that you want to load once in the framework or you might want to add a driver to access another product such as IBM MQ.

System bundles are provided by CICS to manage the interaction with the OSGi framework. These bundles are serviced by IBM as part of the product. An example of a system bundle is the `com.ibm.cics.server.jar` file, which provides most of the JCICS API to access CICS services.

Updating OSGi bundles in an OSGi JVM server

If a Java developer provides an updated version of an OSGi bundle, you can either replace the CICS bundle that refers to it completely, or phase in a new version of the OSGi bundle.

About this task

The update method to use depends on the following factors:

- Whether service outages can be tolerated during the update.
- Whether CICS resource changes can be tolerated during the update.

Using CICS bundle PHASEIN to dynamically update an OSGi bundle without updating CICS resources

Use this update method to phase in a new version of an OSGi bundle when service outages and CICS resource changes cannot be tolerated during the update.

Before you begin

The new version of the JAR file for the OSGi bundle must be present in the same zFS directory as the old version of the OSGi bundle, that is, the same directory as the associated `osgibundle bundlepart` file. By default, this directory is the directory that is named in the `BUNDLE` resource definition. This new version of the OSGi bundle must have a higher version than the one that is currently installed in the OSGi framework, and the version must be in the version range that was defined when the OSGi bundle reference was added to the CICS bundle project.

Procedure

To phase in a new version of an OSGi bundle in an OSGi JVM server, use the following steps.

1. In the **Bundles** view in CICS Explorer, right-click the CICS bundle that contains the OSGi bundle, click **Phase In**, then click **OK**. The new version of the OSGi bundle is phased in, new versions of any services that are implemented by the new version of the OSGi bundle are installed into the OSGi framework, and any old versions of the services are removed from the OSGi framework.
2. In the **OSGi Services** view in CICS Explorer check that all OSGi services for the new version of the OSGi bundle are all in the active state.
3. In the **OSGi Bundles** view in CICS Explorer check that the new version of the OSGi bundle is listed and is in the active state.

Results

The new version of the OSGi bundle is used for all new service requests. Existing requests continue to use the old version. The symbolic version of the OSGi bundle increases, indicating that the Java code is updated.

What to do next

If you are satisfied that the new version of the OSGi service is working well, there is no more to do. Optionally, you can delete the old version of the OSGi JAR file from zFS, but it is not compulsory. It might

also be useful to retain the OSGi JAR file so that you can restore that version if problems arise with the new version.

If you are not satisfied with the new version of the OSGi service and want to restore the old version, use the following steps.

1. Delete the new version of the OSGi bundle JAR from zFS.
2. In the **Bundles** view of CICS Explorer, right-click the CICS bundle that contains the OSGi bundle, click **Phase In**, then click **OK**. Because the old version of the OSGi bundle is now the one with the highest version on zFS, it is reinstalled into the OSGi framework and the defective new version removed.
3. In the **OSGi Services** view in CICS Explorer, check that only the OSGi services for the old version of the OSGi bundle are listed, and are all in the active state.
4. In the **OSGi Bundles** view in CICS Explorer, check that only the old version of the OSGi bundle is listed and is in the active state.

If there is a CICS cold, warm, or emergency restart, the new version of the OSGi bundle is automatically restored. You do not need to change any CICS resource definitions to ensure that this happens.

Phasing in an OSGi bundle with CICS resource changes

Use this update method to phase in a new version of an OSGi bundle when service outages cannot be tolerated during the update process. New CICS resources are created during the update.

Before you begin

A CICS bundle that contains the new version of an OSGi bundle is already defined and exported to zFS. The new version of the OSGi bundle must have a higher version specified in the OSGi bundle manifest than the version that is currently installed in the OSGi framework. You can have both OSGi bundles running in the framework at the same time.

Procedure

To phase in a new version of an OSGi bundle in an OSGi JVM server, use the following steps.

1. In the **Bundle Definitions** view in CICS Explorer, right-click anywhere and click **New** to create a BUNDLE resource to pick up the new CICS bundle project on zFS.
2. In the **Bundle Definitions** view in CICS Explorer, right-click the BUNDLE resource that you created in the previous step and click **Install**. Select the install target, then click **OK** to install the OSGi bundles and services in the CICS bundle into the OSGi framework.
3. Check the status of the OSGi bundles in the **OSGi Bundles** view in CICS Explorer. Two versions of the OSGi bundle are listed with a state of active.
4. In the **OSGi Services** view in CICS Explorer, check the OSGi services that are implemented by both versions of the OSGi bundle are all in the active state. The OSGi services that reference the OSGi bundle with the highest semantic version are used for any new service invocations.

Results

The updated OSGi bundle is available in the OSGi framework along with the old version of the OSGi bundle.

The new version of the OSGi bundle is used for all new service requests. Existing requests continue to use the old version.

What to do next

When you are satisfied that the new version of the OSGi service is working well, use the following steps to remove the old version from the OSGi framework:

1. Disable the BUNDLE resource that points to the old version of the OSGi bundle. In the **Bundles** view in CICS Explorer, right-click the old BUNDLE, click **Disable**, then click **OK**. The old version of any services that are implemented by the OSGi bundle are removed from the OSGi framework; only the new versions of the services are now listed in the **OSGi Services** view. In the **OSGi Bundles** view, the old OSGi bundle state is now in resolved.
2. Discard the BUNDLE resource that points to the old version of the OSGi bundle. In the **Bundles** view in CICS Explorer, right-click the old BUNDLE, click **Discard**, then click **OK**. In the **OSGi Bundles** view, the OSGi bundle from the OSGi framework is removed, and only the new version of the OSGi bundle is listed.

To ensure that the new version of the OSGi bundle is installed if there is a cold start of a CICS region, make sure that you update any CICS group lists (GRPLIST system initialization parameter) that reference the CSD groups that contain the BUNDLE definition for the old version, to reference the CSD groups that contain the new BUNDLE definition that you created in Step 1 of the procedure.

If you want to restore the old version of the OSGi bundle, use the previous two steps to disable and discard the BUNDLE resource that points at the new version of the OSGi bundle. The old version of the service is listed in the **OSGi Services** view in CICS Explorer, and it is used for any new service invocations.

Replacing OSGi bundles in an OSGi JVM server

Use this update method when service outages can be tolerated during the update process. No new CICS resources are created, but you might need to update the existing BUNDLE resource definition.

Before you begin

To replace the CICS bundle completely, an updated CICS bundle that contains the new version of the OSGi bundle must be present in zFS.

Procedure

To replace an existing OSGi bundle in an OSGi JVM server with a new version of the OSGi bundle, use the following steps.

1. In the **Bundles** view in CICS Explorer, disable and discard the BUNDLE resource for the CICS bundle that you want to update. The OSGi services that are part of that CICS bundle are removed from the OSGi framework and are not listed in the **OSGi Services** view of CICS Explorer.

   Note: No services that are implemented by the OSGi bundle are available in the OSGi framework from this point until the completion of step 3, so any users of these services suffer a service outage.
2. Optional: Edit the BUNDLE resource definition if the updated CICS bundle is deployed in a different directory in zFS.
3. In the **Bundles** view in CICS Explorer, install the BUNDLE resource definition to pick up the changed OSGi bundle. The OSGi bundles and services in the CICS bundle are installed in the OSGi framework.
4. Check the status of the OSGi bundle in the **OSGi Bundles** view in CICS Explorer. The new version of the OSGi bundle is listed with a state of active.
5. In the **OSGi Services** view in CICS Explorer, check that the new version of all the OSGi services that are implemented by the new version of the OSGi bundle are in the active state.

Results

The new version of the OSGi bundle is used for all new service requests. Existing requests continue to use the old version. The symbolic version of the OSGi bundle increases, indicating that the Java code is updated.

Updating bundles that contain common libraries

OSGi bundles that contain common libraries for use by other OSGi bundles must be updated in a specific order.

Before you begin

An updated CICS bundle that contains the new version of the OSGi bundle must be present in zFS. If you manage common libraries in a separate CICS bundle, you can manage the lifecycle of these libraries separately from the applications that depend on them.

About this task

Typically, an OSGi bundle specifies a range of supported versions in a dependency on another OSGi bundle. Using a range provides more flexibility to make compatible changes in the framework. When you are updating bundles that contain common libraries, the version number of the OSGi bundle increases. However, the running applications are already using a version of the bundle that satisfies the dependencies. To obtain the most recent version of the library, you must refresh the OSGi bundles for the applications. It is therefore possible to update specific applications to use different versions of the library, and leave other applications to run on an older version.

When you update an OSGi bundle that contains common libraries, you can completely replace the CICS BUNDLE resource. However, if classes are not loaded in the library, the dependent bundles might receive errors. Alternatively, you can install a new version of the library and run it in the framework alongside the original version. If the OSGi bundles have different version numbers, the OSGi framework can run both bundles concurrently.

Procedure

To replace an existing OSGi bundle in an OSGi JVM server:

1. Define and install a CICS BUNDLE resource that points to the new version of the CICS bundle, which contains the OSGi bundle that defines the common libraries. CICS defines the new version of the OSGi bundle in the OSGi framework. The existing OSGi bundles continue to use the previous version of the library.
2. Check the status of the OSGi bundles in the **OSGi Bundles** view in CICS Explorer (**Operations** > **Java** > **OSGi Bundles**). The list shows two entries for the same OSGi bundle symbolic name with different versions that are running in the framework.
3. To obtain the new version of the library in a dependent Java application, use one of the following methods:
   - Replace the CICS bundle for the Java application.
     a. Disable and discard the CICS BUNDLE resource for the Java application.
     b. Reinstall the CICS BUNDLE resource for the Java application.
   - Phase in a new version of the Java application.
     a. Ask the Java developer to update the version information for the OSGi bundle. The new version of the OSGi bundle must have a higher version specified in the OSGi bundle manifest and be within the version range specified when the OSGi Bundle Project was added to the CICS bundle. Optionally, the new version of the OSGi bundle could also have its dependencies modified to specifically require the new version of the OSGi bundle that defines the common libraries.
     b. Copy the JAR for the new version of OSGi bundle to the root directory of the CICS BUNDLE resource.
     c. In the **Operations** > **Bundles** view in CICS Explorer, right-click the CICS bundle that contains the OSGi bundle, click **Phase In**, then click **OK** to phase in the new version of the OSGi bundle.

   When the OSGi bundle is loaded in the framework, it obtains the latest version of the common libraries.

4. Check the status of the CICS BUNDLE resource in the **Bundles** view in CICS Explorer (**Operations** > **Bundles**).

Results

You have updated an OSGi bundle that contains common libraries and updated a Java application to use the latest version of the libraries.

Updating OSGi middleware bundles

To update the middleware bundles that are running in an OSGi framework, you must stop and restart the JVM server.

About this task

OSGi middleware bundles are installed in the OSGi framework during the initialization of the JVM server. If you want to update a middleware bundle, for example to apply a patch or use a new version, you must stop and restart the JVM server to pick up the changed bundle.

You can manage the lifecycle of the JVM server and edit the JVM profile by using CICS Explorer.

Procedure

1. Ensure that the new version of the middleware bundle is in a directory on zFS to which CICS has read and execute access. CICS also requires read access to the files.
2. If the zFS directory or file name is different from the values that are specified in the JVM profile, edit the `OSGI_BUNDLES` option in the JVM profile for the JVM server.
   a) Open the **JVM Servers** view in CICS Explorer to find out the name and location of the JVM profile in zFS.

      You must be connected with a region or CICSplex selected to see the JVMSERVER resources.

   b) Open the **z/OS UNIX Files** view and browse to the directory that contains the JVM profile.

   c) Edit the JVM profile to update the `OSGI_BUNDLES` option.
3. Disable the JVMSERVER resource to shut down the JVM server.

   Disabling the JVMSERVER also disables any BUNDLE resources that contain OSGi bundles that are installed in that JVM server.
4. Enable the JVMSERVER resource to start the JVM server with the updated JVM profile.

   The JVM server starts up and installs the new version of the middleware bundle in the OSGi framework. CICS also enables the BUNDLE resources that were disabled and installs the OSGi bundles and services in the updated framework.

Results

The OSGi framework contains the updated middleware bundles and the OSGi bundles and services for Java applications that were installed before you shut down the JVM server.

Removing OSGi bundles from a JVM server

If you want to remove OSGi bundles from the JVM server, use the CICS Explorer to disable and discard the BUNDLE resource.

About this task

The BUNDLE resource provides life-cycle management for the collection of OSGi bundles and OSGi services that are defined in the CICS bundle. Removing OSGi bundles from the OSGi framework does not automatically affect the state of other installed OSGi bundles and services. If you remove a bundle that is a prerequisite for another bundle, the state of the dependent bundle does not generally change until you explicitly refresh that bundle. An exception is in the use of singleton bundles. If you uninstall a singleton

bundle that other bundles depend on, the dependent bundles cannot use the services of the uninstalled bundle. The reported status of the CICS BUNDLE resource might not accurately reflect the status of the OSGi bundle.

Procedure

1. Click **Operations** > **Java** > **OSGi Bundles** to find out which BUNDLE resource contains the OSGi bundle.
2. Click **Operations** > **Bundles** to disable the BUNDLE resource.

   CICS disables each resource that is defined in the CICS bundle. For OSGi bundles and services, CICS sends a request to the OSGi framework in the JVM server to unregister any OSGi services and moves the OSGi bundles into a resolved state. Any in-flight transactions complete, but any new links to the OSGi service from CICS applications return with an error.
3. Discard the BUNDLE resource.

   CICS sends a request to the OSGi framework to remove the OSGi bundles from the JVM server.

Results

You have removed the OSGi bundles and services from the OSGi framework.

What to do next

If you have PROGRAM resources pointing to OSGi services that are no longer in the OSGi framework, you might want to disable and discard the PROGRAM resources.

Moving applications to a JVM server

If you are running Java applications in pooled JVMs, you can move them to run in a JVM server. Because a JVM server can handle multiple requests for Java applications in the same JVM, you can reduce the number of JVMs that are required to run the same workload.

Before you begin

Ensure that the application is threadsafe and is packaged as one or more OSGi bundles. The OSGi bundles must be deployed in a CICS bundle to zFS and specify the correct target JVMSERVER resource.

The Java developer can use the CICS SDK for Java that is included with CICS Explorer to repackage a Java application using OSGi. For more information on how to migrate applications that use third party JARs, see Upgrading the Java environment.

About this task

You can either use an existing JVM server or create a JVM server for your application. Do not move an application to a JVM server where the thread limit and usage are already high, because you might introduce locking contentions in the JVM server.

Procedure

1. Create or update a JVM server:
   - If you decide to create a JVM server, see Configuring a Liberty JVM server. Many of the settings in a JVM profile for a pooled JVM do not apply to JVM servers. The only option that you might want to copy from the pooled JVM profile to the DFHOSGI profile is the `LIBPATH_SUFFIX` option.
   - If you use an existing JVM server, you might have to increase the THREADLIMIT attribute on the JVMSERVER resource to handle the additional application or update the options in the JVM server profile. If you change the JVM profile, restart the JVM server to pick up the changes.
2. Create a BUNDLE resource that points to the deployed bundle in zFS.

When you install the BUNDLE resource, CICS loads the OSGi bundles into the OSGi framework in the JVM server. The OSGi framework resolves the OSGi bundles and registers the OSGi services.

Use CICS Explorer to check that the BUNDLE resource is enabled. You can also use the OSGi Bundles and OSGi Services views to check the state of the OSGi bundles and services.

3. Update the PROGRAM resource for the application:
   a) Ensure that the EXECKEY attribute is set to CICS.

      All JVM server work runs in CICS key.

   b) Remove the JVM profile name and enter the name of the JVMSERVER resource.

   c) Ensure that the JVMCLASS attribute matches the OSGi service of the Java application.

   d) Reinstall the PROGRAM resource for the application.

   The PROGRAM resource uses the OSGi service to make an OSGi bundle available to other CICS applications outside the JVM server.

Results

When the Java application is called, it runs in the JVM server.

What to do next

You can use the JVM server view in CICS Explorer and CICS statistics to monitor the JVM server. If the performance is not optimal, adjust the thread limit.

Updating Java EE applications in a Liberty JVM server

There are three methods to update Java EE applications in a Liberty JVM server: refresh the CICS bundles, update the applications in the drop-ins folder, and use `<application>` elements.

About this task

The process to update Java EE applications in a Liberty server depends on how the applications are deployed:

- Applications deployed in CICS bundles

  In this scenario, the application must be added as a bundle part to a CICS bundle project using CICS Explorer and then exported to z/OS File System (zFS). It is then installed into CICS using a BUNDLE definition that refers to the exported project.

- Applications deployed directly to the Liberty `drop-ins` folder

  In this scenario, the Java archive is copied directly to a previously defined `drop-ins` directory.

- Applications deployed in an `<application>` element into `server.xml`

  In this scenario, a reference to the application is added into `server.xml`, together with further application attributes and descriptive elements.

Procedure

Applications deployed in CICS bundles

- To refresh the CICS bundle, a bundle that contains the Java EE application must already be installed and enabled in the CICS region. For more information, see Deploying a Java EE application in a CICS bundle to a Liberty JVM server.
  a) In the **Bundles** view in CICS Explorer, disable the BUNDLE resource for the CICS bundle that you want to update.

     Note: The applications that are part of that CICS bundle are removed from the Liberty server run time and are not available from this point until the last step completes. Any users of these services suffer a service outage.

b) Export the new version of the CICS bundle that contains the Java EE application to the same zFS location as the old version.

c) In the **Bundles** view in CICS Explorer, enable the BUNDLE resource definition to pick up the Java EE application. The applications are reinstalled into the Liberty server.

d) Check the status of the CICS bundle in CICS Explorer. The CICS bundle is listed with a state of active.

When the new version of the Java EE application becomes active, it is used for all new requests.

Applications deployed directly to the Liberty `drop-ins` folder

- To use the `drop-ins` directory with a Liberty server, the `server.xml` configuration must be updated to enable this function. For more information, see Deploying Java EE applications directly to a Liberty JVM server.

  a) Export the new version of the archive (WAR, EAR, or EBA) from your Eclipse environment.

  Note: The applications that are part of that CICS bundle are removed from the Liberty server run time. Any users of these services suffer a service outage.

  b) Copy this new archive into the `drop-ins` directory, replacing the original version.

  The Liberty server scans the directory, uninstalls the previous version, and installs the new version.

  When the new version of the Java EE application becomes active, it is used for all new requests.

Applications deployed in an `<application>` element into `server.xml`

- To allow applications to be dynamically updated, the `updateTrigger` attribute of the `<applicationMonitor>` element must be set to `polled`. For more information, see Controlling dynamic updates.

  a) Export the new version of the archive (WAR, EAR, or EBA) from your Eclipse environment.

  Note: The applications that are part of that CICS bundle are removed from the Liberty server run time. Any users of these services suffer a service outage.

  b) Copy this new archive into the location specified in your `<application>` element.

  The Liberty server scans the file for modification and if a change is detected, it uninstalls the previous version and installs the new version.

  When the new version of the Java EE application becomes active, it is used for all new requests.

Managing the thread limit of JVM servers

JVM servers are limited in the number of threads that they can use to run Java applications. The CICS region also has a limit on the number of threads, because each thread uses a T8 TCB. You can adjust the thread limit using CICS statistics to balance the number of JVM servers in the region against the performance of the applications running in each JVM server.

About this task

Each JVM server can have a maximum of 256 threads to run Java applications. In a CICS region you can have a maximum of 2000 threads. If you have many JVM servers running in the CICS region (for example, more than seven), you cannot set the maximum value for every JVM server. You can adjust the thread limit of each JVM server to balance the number of JVM servers in the CICS region against the performance of the Java applications.

The thread limit is set on the JVMSERVER resource, so set an initial value and use CICS statistics to adjust the number of threads when you test your Java workloads.

Procedure

1. Enable the JVMSERVER resources and run your Java application workload.
2. Collect JVMSERVER resource statistics using an appropriate statistics interval.

You can use the **Operations** > **Java** > **JVM Servers** view in CICS Explorer, or you can use the DFH0STAT statistics program.

3. Check how many times and how long a task waited for a thread.

   The "JVMSERVER thread limit waits" and "JVMSERVER thread limit wait time" fields contain this information.

   - If the values in these fields are high and many tasks are suspended with the JVMTHRD wait, the JVM server does not have enough threads available. Increasing the number of threads can increase the processor usage, so check you have enough MVS resource available.
   - If the values in these fields are low and the peak number of tasks is below the maximum number of threads available, you can free up threads for other JVM servers by reducing the thread limit.
4. To check the availability of MVS resource, use the dispatcher TCB pool and TCB mode statistics to assess the T8 TCB usage across the CICS region.

   Each thread in a JVM server uses a T8 TCB and you are limited to 2000 in a region. T8 TCBs cannot be shared between JVM servers, although all TCBs are in a THRD TCB pool. If the number of waiting TCBs and processor usage is low, it indicates that there is enough MVS resource available.
5. To adjust the number of threads that can run in the JVM server, change the THREADLIMIT value on the JVMSERVER resource.
6. Run the Java application workload again and use the statistics to check that the number of waiting tasks has reduced.

What to do next

To tune the performance of your JVM servers, see Improving JVM server performance.

Writing Java classes to redirect JVM stdout and stderr output

Use the USEROUTPUTCLASS option in a JVM profile to name a Java class that intercepts the `stdout` stream and `stderr` stream from the JVM. You can update this class to specify your choice of time stamps and record headers, and to redirect the output.

CICS supplies sample Java classes, `com.ibm.cics.samples.SJMergedStream`, and `com.ibm.cics.samples.SJTaskStream`, that you can use for this purpose. Sample source is provided for both these classes, in the directory `/usr/lpp/cicsts/cicsts54/samples/com.ibm.cics.samples`. The `/usr/lpp/cicsts/cicsts54` directory is the installation directory for CICS files on z/OS UNIX. This directory is specified by the **USSDIR** parameter in the DFHISTAR installation job. The sample classes are also shipped as a class file, `com.ibm.cics.samples.jar`, which is in the directory `/usr/lpp/cicsts/cicsts54/lib`. You can modify these classes, or write your own classes based on the samples.

Controlling the location for JVM stdout, stderr, JVMTRACE, and dump output has information about:

- The types of output from JVMs that are and are not intercepted by the class that is named by the USEROUTPUTCLASS option. The class that you use must be able to deal with all the types of output that it might intercept.
- The behavior of the supplied sample classes. The `com.ibm.cics.samples.SJMergedStream` class creates two merged log files for JVM output and for error messages, with a header on each record that contains APPLID, date, time, transaction ID, task number, and program name. The log files are created by using transient data queues, if they are available; or z/OS UNIX files, if the transient data queues are not available, or cannot be used by the Java application. The `com.ibm.cics.samples.SJTaskStream` class directs the output from a single task to z/OS UNIX files, adding time stamps and headers, to provide output streams that are specific to a single task.

For a JVM server to use an output redirection class, you must create an OSGi bundle that contains your output redirection class. You must ensure that the bundle activator registers an instance of your class as a service in the framework and sets the property `com.ibm.cics.server.outputredirectionplugin.name=`*`class_name`*. You can use the constant `com.ibm.cics.server.Constants.CICS_USER_OUTPUT_CLASSNAME_PROPERTY` to get

the property name. The following code excerpt shows how you might register your service in the bundle activator:

```
Properties serviceProperties = new Properties();
    serviceProperties.put(Constants.CICS_USER_OUTPUT_CLASSNAME_PROPERTY,
MyOwnStreamPlugin.class.getName());
    context.registerService(OutputRedirectionPlugin.class.getName(), new MyOwnStreamPlugin(),
serviceProperties);
```

You can either add the OSGi bundle to the OSGI_BUNDLES option in the JVM profile or ensure that the bundle is installed in the framework when the first task is run. Whichever method you use, you must still specify the class in the USEROUTPUTCLASS option.

If you decide to write your own classes, you need to know about:

- The `OutputRedirectionPlugin` interface
- Possible destinations for output
- Handling output redirection errors and internal errors

The output redirection interface

CICS supplies an interface called `com.ibm.cics.server.OutputRedirectionPlugin` in `com.ibm.cics.server.jar`, which can be implemented by classes that intercept the stdout and stderr output from the JVM. The supplied samples implement this interface.

The following sample classes are provided:

- A superclass `com.ibm.cics.samples.SJStream` that implements this interface
- The subclasses `com.ibm.cics.samples.SJMergedStream` and `com.ibm.cics.samples.SJTaskStream`, which are the classes named in the JVM profile

Like the sample classes, ensure that your class implements the interface `OutputRedirectionPlugin` directly, or extends a class that implements the interface. You can either inherit from the superclass `com.ibm.cics.samples.SJStream`, or implement a class structure with the same interface. Using either method, your class must extend `java.io.OutputStream`.

The `initRedirect()` method receives a set of parameters that are used by the output redirection class or classes. The following code shows the interface:

```
package com.ibm.cics.server;

import java.io.*;

public interface OutputRedirectionPlugin {

  public boolean initRedirect( String inDest,
                               PrintStream inPS,
                               String inApplid,
                               String inProgramName,
                               Integer inTaskNumber,
                               String inTransid
                              );
  }
```

The superclass `com.ibm.cics.samples.SJStream` contains the common components of `com.ibm.cics.samples.SJMergedStream` and `com.ibm.cics.samples.SJTaskStream`. It contains an `initRedirect()` method that returns `false`, which effectively disables output redirection unless this method is overridden by another method in a subclass. It does not implement a `writeRecord()` method, and such a method must be provided by any subclass to control the output redirection process. You can use this method in your own class structure. The initialization of output redirection can also be performed using a constructor, rather than the `initRedirect()` method.

The **inPS** parameter contains either the original `System.out` print stream or the original `System.err` print stream of the JVM. You can write logging to either of these underlying logging destinations. You must not call the `close()` method on either of these print streams because they remain closed permanently and are not available for further use.

Possible destinations for output

The CICS-supplied sample classes direct output from JVMs to a directory that is specific to a CICS region; the directory name is created using the applid associated with the CICS region. When you write your own classes, if you prefer, you can send output from several CICS regions to the same z/OS UNIX directory or file.

For example, you might want to create a single file containing the output associated with a particular application that runs in several different CICS regions.

Threads that are started programmatically using Thread.start() are not able to make CICS requests. For these applications, the output from the JVM is intercepted by the class you have specified for USEROUTPUTCLASS, but it cannot be redirected using CICS facilities (such as transient data queues). You can direct output from these applications to z/OS UNIX files, as the supplied sample classes do.

Handling output redirection errors and internal errors

If your classes use CICS facilities to redirect output, they should include appropriate exception handling to deal with errors in using these facilities.

For example, if you are writing to the transient data queues CSJO and CSJE, and using the CICS-supplied definitions for these queues, the following exceptions might be thrown by TDQ.writeData:

- IOErrorException
- LengthErrorException
- NoSpaceException
- NotOpenException

If your classes direct output to z/OS UNIX files, they should include appropriate exception handling to deal with errors that occur when writing to z/OS UNIX. The most common cause of these errors is a security exception.

The Java programs that will run in JVMs that name your classes on the USEROUTPUTCLASS options should include appropriate exception handling to deal with any exceptions that might be thrown by your classes. The CICS-supplied sample classes handle exceptions internally, by using a Try/Catch block to catch all throwable exceptions, and then writing one or more error messages to report the problem. When an error is detected while redirecting an output message, these error messages are written to `System.err`, making them available for redirection. However, if an error is found while redirecting an error message, then the messages which report this problem are written to the file indicated by the STDERR option in the JVM profile used by the JVM that is servicing the request. Because the sample classes trap all errors in this way, this means that the calling programs do not need to handle any exceptions thrown by the output redirection class. You can use this method to avoid making changes to your calling programs. Be careful that you do not send the output redirection class into a loop by attempting to redirect the error message issued by the class to the destination which has failed.

Chapter 6. Security for Java applications

You can secure Java applications to ensure that only authorized users can deploy and install applications, and access those applications from the web or through CICS. You can also use a Java security manager to protect the Java application from performing potentially unsafe actions.

You can add security at different points in the Java application lifecycle:

- Implement security checking for defining and installing Java application resources. Java applications are packaged in CICS bundles, so you must ensure that users who are allowed to install applications in the JVM server can install this type of resource.
- Implement security checking for application users to ensure that only authorized users can access an application.
- Implement security checking for CICS Java tasks that are started using the CICSExecutorService. All such CICS tasks run under the CJSA transaction and the default user ID.
- Implement security restrictions on the Java API by using a Java security manager.

Java applications can run in an OSGi framework or a Liberty server. Liberty is designed to host web applications and includes an OSGi framework. The security configuration for a Liberty server is different, because Liberty has its own security model.

To configure security for OSGi applications, use CICS resource security to authorize which users can manage the lifecycle of the JVMSERVER and the Java applications. Use CICS transaction security to determine who can access the application.

Configuring security for OSGi applications

Use CICS resource security to authorize which users can manage the lifecycle of the JVMSERVER and the Java applications. Use CICS transaction security to determine who can access the application.

Procedure

- Authorize application developers and system administrators to create, view, update, and remove JVMSERVER and BUNDLE resources as appropriate. The JVMSERVER resource controls the availability of the JVM server. The BUNDLE resource is a unit of deployment for the Java application and controls the availability of the application.
- Authorize users to run the application by ensuring the relevant user ID is allowed to attach the transaction under which the application will run.

Results

You have successfully configured security for Java applications that run in an OSGi framework.

Configuring security for a Liberty JVM server

You can use the CICS Liberty security feature to authenticate users and authorize access to web applications through Java Platform, Enterprise Edition roles (Java EE roles), providing integration with CICS transaction and resource security. You can also use CICS resource security to authorize the appropriate users to manage the lifecycle of both the JVMSERVER resource and Java web applications that are deployed in a CICS BUNDLE resource. In this topic, authentication verifies the identity of a given user, typically by requiring the user to enter a username and password. Authorization grants access control permissions based on the identity of the authenticated user.

Before you begin

1. Ensure that the CICS region is configured to use SAF security and is defined with SEC=YES as a system initialization parameter.
2. Authorize application developers and system administrators to create, view, update, and remove JVMSERVER and BUNDLE resources to deploy web applications into a Liberty JVM server.

The JVMSERVER resource controls the availability of the JVM server, and the BUNDLE resource is a unit of deployment for the Java applications and controls the availability of the applications. The default behavior of the CICS TS security feature, `cicsts:security-1.0`, is to use the SAF registry. If you use an LDAP registry, a SAF registry is not created, for more information, see Configuring security for a Liberty JVM server by using distributed identity mapping. The basic user registry (which is also used by `quickStartSecurity`) is only suitable for simple security testing. Be aware that if you configure and run with basic user registry and you need to switch to `cicsts:security-1.0`, you need to delete the session tokens.

About this task

This task explains how to configure security for a Liberty JVM server and integrate Liberty security with CICS security. For information about how to configure security for Link to Liberty, see Linking to a Java EE application from a CICS program.

The default transaction ID for running web requests is CJSA. However, you can configure CICS to run web requests under a different transaction ID by using a URIMAP of type JVMSERVER. Typically, you might specify a URIMAP to match the generic context root (URI) of a web application to scope the transaction ID to the set of servlets that make up the application. Or you might choose to run each individual servlet under a different transaction with a more precise URI.

The default user ID for running web requests is the CICS default user ID. If a URIMAP is available and contains a static user ID, it is used in preference to the default user ID. If the web request contains a user ID in its security header, it takes precedence over all other mechanisms.

Tasks that emanate from Liberty that are not classified as web requests run under the CJSU transaction by default. Although there is no URIMAP style mechanism for these types of tasks, you can override the default transaction ID by using the JVM profile property of `com.ibm.cics.jvmserver.unclassified.tranid` and the default user ID by using the JVM profile property `com.ibm.cics.jvmserver.unclassified.userid`.

Note: The user ID requires permission to attach the specified transaction. For more information, see Transaction security.

Procedure

1. Configure the Liberty angel process to provide authentication and authorization services to the Liberty JVM server, see The Liberty server angel process.

   Tip: If you have a named angel process, you need to configure your Liberty JVM server to connect to it by adding the following line to your JVM profile.

   ```
   -Dcom.ibm.ws.zos.core.angelName=<named_angel>
   ```

2. Optional: Enforce the requirement to connect to the Liberty angel process when the Liberty JVM server is being enabled by adding the following line to your JVM profile:

   ```
   -Dcom.ibm.ws.zos.core.angelRequired=true
   ```

   This option prevents the Liberty JVM server from starting if the angel process is unavailable.

   With CICS TS V5.4 with APAR PI92676 or later, this option instructs CICS to call the Liberty angel check API to verify whether an angel process is available for Liberty JVM server startup.

   If the angel process is unavailable, CICS reacts as follows:

- If the Liberty JVM server is being enabled through the CEMT transaction, a message is issued, and the Liberty JVM server is disabled.
- If the Liberty JVM server is being enabled by the **SET JVMSERVER** SPI command or by using the CMCI through the CICS Explorer, a message is issued, and the Liberty JVM server is disabled.
- If the Liberty JVM server is being enabled by the CICS CREATE SPI, by BAS, or from GRPLIST, a message is issued, and CICS will wait 30 seconds before retrying the Liberty angel check API call. If the angel process is unavailable on the fifth attempt, a WTOR message is issued, giving the operator the option to continue waiting or to disable the JVMSERVER resource.

3. Add the `cicsts:security-1.0` feature to the **`featureManager`** list in the `server.xml`,

```
<featureManager>
   ...
   <feature>cicsts:security-1.0</feature>
</featureManager>
...
```

4. Add the System Authorization Facility (SAF) registry to `server.xml` by using the following example:

```
<safRegistry id="saf" enableFailover="false"/>
```

5. Save the changes to `server.xml`.
6. Optional: Alternatively, if you are autoconfiguring the Liberty JVM server and the **SEC** system initialization parameter is set to YES in the CICS region, the Liberty JVM server is dynamically configured to support Liberty JVM security when the JVM server is restarted. For more information, see Configuring a Liberty JVM server.

   If the **SEC** system initialization parameter is set to NO, you can still use Liberty security for authentication or SSL support. If CICS security is turned off, and you want to use a Liberty security, you must configure the `server.xml` file manually:

   a. Add the appSecurity`-2.0` feature to the **`featureManager`** list.
   b. Add a user registry to authenticate users. Liberty security supports SAF, LDAP, and basic user registries. For more information, see Configuring a user registry in Liberty.
   c. Add security-role definitions to authorize access to application resources, see “Authorizing users to run applications in a Liberty JVM server” on page 218.

Results

The web container is automatically configured to use the z/OS® Security feature of Liberty. A SAF registry is used for authentication, and Java EE roles are respected for authorization. Authorization constraints and security roles govern who can access the application. These are usually defined in the deployment descriptor (`web.xml`) of the application, but might also be defined as security annotations in the source-code. Typically, users and groups are mapped to roles by the applications *<application-bnd>* element in `server.xml`. Alternatively, if the*<safAuthorization>* element is configured in `server.xml`, the mappings are held in SAF (as EJBROLES in RACF).

What to do next

Note: You can also delegate authentication to another identity by configuring the RunAs specification for Liberty, see Configuring RunAs authentication in Liberty.

- Configure Liberty application security authentication rules, see “Authenticating users in a Liberty JVM server” on page 216.
- Define authorization rules for web applications, see “Authorizing users to run applications in a Liberty JVM server” on page 218 and “Authorization using SAF role mapping” on page 222.
- Modify the Liberty authentication cache.

For more information about using Secure Sockets Layer (SSL), see “Configuring SSL (TLS) for a Liberty JVM server using a Java keystore” on page 227.

The Liberty angel process

When you include the `cicsts:security-1.0` feature, the CICS Liberty JVM server uses the angel process to call z/OS authorized services such as System Authorization Facility (SAF).

Optionally, you can name an angel process. If an angel process is not given a name, it becomes the default angel process. You can only have one default angel process. If you try to create another, it will fail to start. All the Liberty servers that are running on a z/OS image can share a single angel process. This is regardless of the level of code that the servers are running or whether they are running in a CICS JVM server. For further information on named angel processes, see Named angel.

Important: Install the latest version of the angel process, regardless of which product it is bundled with. The latest version might be bundled with other IBM software, and might supersede the version that is bundled with CICS.

Running the angel process started task

1. Locate the JCL procedure for the **started** task in the USSHOME directory, for example: `/usr/lpp/cicsts54/wlp/templates/zos/procs/bbgzangl.jcl`
2. Modify and copy the JCL procedure to a JES procedure library. You can set **ROOT** to the value of USSHOME/wlp, for example: ROOT=/usr/lpp/cicsts54/wlp
3. Start the angel process. In the following examples, `[.identifier]` indicates an optional identifier that can be up to 8 characters.
   a. To start the angel process without naming it, use the following command:

   ```
   START BBGZANGL[.identifier]
   ```

   b. To start the angel process as a named angel process, code the NAME parameter on the operator START command. For example:

   ```
   START BBGZANGL[.identifier],NAME=<named_angel>
   ```

   The angel process name is 1 - 54 characters inclusive, and must use only the following characters:
   `A-Z 0-9 ! # $ + - / : < > = ? @ [ ] ^ _ ` { } | ~`

   Note: A Liberty server can use its own named angel process. One benefit of this isolation is that the angel process can be serviced without affecting any other Liberty server instances on the LPAR. The angel process must be running before the Liberty JVM server starts.
4. Start the Liberty JVM server. By default, the server connects to the unnamed angel process if one is available. To connect to a specific angel process, set the `com.ibm.ws.zos.core.angelName` property in the JVM server profile, for example:

   ```
   -Dcom.ibm.ws.zos.core.angelName=named_angel
   ```

5. You can specify that CICS checks for the presence of a running angel process before enabling, by setting the `com.ibm.ws.zos.core.angelRequired` property in the JVM server profile to true. For example:

   ```
   -Dcom.ibm.ws.zos.core.angelRequired=true
   ```

   The server fails if the angel process is not available during startup. Use of this property allows a quicker and cleaner failure.

Interacting with the angel process started task

In the following examples, `[.identifier]` indicates an optional identifier that can be up to eight characters.

- Display the Liberty JVM servers that are connected to the angel process:

```
MODIFY BBGZANGL[.identifier],DISPLAY,SERVERS,PID
```

A list of job names and process identifiers (PID) are displayed:

```
15.48.45 STC82204  CWWKB0067I ANGEL DISPLAY OF ACTIVE SERVERS
15.48.45 STC82204  CWWKB0080I ACTIVE SERVER ASID 5c JOBNAME IYK3ZNA1 PID 83953428
15.48.45 STC82204  CWWKB0080I ACTIVE SERVER ASID 5c JOBNAME IYK3ZNA1 PID 33621002
```

Each Liberty JVM server runs under a unique PID, and is returned by the CICS command **INQUIRE JVMSERVER**.

- Stop the angel process.

```
STOP BBGZANGL[.identifier]
```

Note: The Liberty JVM server must be stopped before restarting or applying maintenance to the angel process.

SAF profiles used by the angel process

This section describes the SAF profiles to which access is required for CICS processing. For information on the full set of SAF profiles defined by Liberty, refer to Enabling z/OS authorized services on Liberty for z/OS.

- The Liberty JVM server runs under the authority of the CICS region user ID. This user ID must be able to connect to the angel process to use authorized services. The user ID that the angel process runs under needs access to the SAF STARTED profile, for example:

```
RDEFINE STARTED BBGZANGL.* UACC(NONE) STDATA(USER(WLPUSER))
SETROPTS RACLIST(STARTED) REFRESH
```

- For the Liberty JVM server to connect to an angel process, create a profile for the angel (**BBG.ANGEL**, or **BBG.ANGEL.<namedAngelName>** if you are using a named angel process) in the **SERVER** class. Give the CICS region user ID (*cics_region_user*) authority to access it, for example, in RACF:

```
RDEFINE SERVER BBG.ANGEL UACC(NONE)
PERMIT BBG.ANGEL CLASS(SERVER) ACCESS(READ) ID(cics_region_user)
```

- For a Liberty server to use the z/OS authorized services, create a **SERVER** profile for the authorized module **BBGZSAFM** and give the CICS region user ID (*cics_region_user*) to the profile:

```
RDEFINE SERVER BBG.AUTHMOD.BBGZSAFM UACC(NONE)
PERMIT BBG.AUTHMOD.BBGZSAFM CLASS(SERVER) ACCESS(READ) ID(cics_region_user)
```

- Give the Liberty JVM server, under the authority of the CICS region user ID (*cics_region_user*), access to the SAF user registry and SAF authorization services (**SAFCRED**) in the **SERVER** class. For example, in RACF:

```
RDEFINE SERVER BBG.AUTHMOD.BBGZSAFM.SAFCRED UACC(NONE)
PERMIT BBG.AUTHMOD.BBGZSAFM.SAFCRED CLASS(SERVER) ACCESS(READ) ID(cics_region_user)
```

- Create a **SERVER** profile for the **IFAUSAGE** services (**PRODMGR**) and allow the CICS region user ID access to it. This allows the Liberty JVM server to register and unregister from **IFAUSAGE** when the CICS JVM server is enabled and disabled:

```
RDEFINE SERVER BBG.AUTHMOD.BBGZSAFM.PRODMGR UACC(NONE)
PERMIT BBG.AUTHMOD.BBGZSAFM.PRODMGR CLASS(SERVER) ACCESS(READ) ID(cics_region_user)
```

- Refresh the **SERVER** resource:

```
SETROPTS RACLIST(SERVER) REFRESH
```

The following table summarizes the SAF security profiles used by a Liberty server running in a CICS JVM server.

Table 36. SAF profile table for CICS Liberty security				
Class	**Profile**	**CICS region user ID** 1	**Unauthenticated user ID** 2	**Authenticated user ID** 3
Required for angel process registration at Liberty server startup				
SERVER	BBG.ANGEL	READ		
SERVER	BBG.ANGEL.<namedAngelName>	READ		
SERVER	BBG.AUTHMOD.BBGZSAFM	READ		
SERVER	BBG.AUTHMOD.BBGZSAFM.SAFCRED	READ		
SERVER	BBG.AUTHMOD.BBGZSAFM.PRODMGR	READ		
Required for authentication or authorization				
SERVER	BBG.SECPFX.BBGZDFLT 4	READ		
APPL	BBGZDFLT 4		READ	READ
EJBROLE	BBGZDFLT.<resource>.<role> 5			READ

1. User ID that is associated with the CICS job or started task.
2. User ID used for unauthenticated requests in Liberty. The value is controlled using the `unauthenticatedUser` attribute of the `<safCredentials>` element. This defaults to `WSGUEST`.
3. User ID authenticated by the Liberty server.
4. BBGZDFLT is the default value for the security profile prefix which is set using the `profilePrefix` attribute of the `<safCredentials>` element, for example: `<safCredentials profilePrefix="BBGZDFLT"/>`.
5. EJBROLE profiles are required if the `<safAuthorization>` element is configured. The default pattern for the profile is controlled by the SAF role mapper element which defaults to `<safRoleMapper profilePattern="%profilePrefix%.%resource%.%role%"/>`.

For more information, see Liberty: Process types on z/OS.

Authenticating users in a Liberty JVM server

Although you can configure CICS security for all web applications that run in a Liberty JVM server, the web application will only authenticate users if it includes a security constraint. The security constraint is defined by an application developer in the deployment descriptor (web.xml) of the Dynamic Web Project or OSGi Application Project. The security constraint defines what is to be protected (URL) and by which roles.

A `<login-config>` element defines the way a user gains access to web container and the method used for authentication. The supported methods are either HTTP basic authentication, form based authentication or SSL client authentication. For further details on how to define application security for CICS see SSL security for Explorer connections in the CICS Explorer product documentation. Here is an example of those elements in web.xml:

```
<!-- Secure the application -->
<security-constraint>
    <display-name>com.ibm.cics.server.examples.wlp.tsq.web_SecurityConstraint</display-name>
        <web-resource-name>com.ibm.cics.server.examples.wlp.tsq.web</web-resource-name>
        <description>Protection area for com.ibm.cics.server.examples.wlp.tsq.web</description>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
```

```
        <description>Only SuperUser can access this application</description>
        <role-name>SuperUser</role-name>
    </auth-constraint>
    <user-data-constraint>
        <!-- Force the use of SSL -->
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>

<!-- Declare the roles referenced in this deployment descriptor -->
<security-role>
    <description>The SuperUser role</description>
    <role-name>SuperUser</role-name>
</security-role>

<!--Determine the authentication method -->
<login-config>
        <auth-method>BASIC</auth-method>
</login-config>
```

Note: If you use RequestDispatcher.forward() methods to forward requests from one servlet to another, the security check occurs only on the first servlet that is requested from the client.

Tasks that are authenticated in CICS using Liberty security can use the user ID derived from any of the Liberty application security mechanisms to authorize transaction and resource security checks in CICS. The CICS user ID is determined according to the following criteria:

1. Liberty application security authentication.

   Integration with the SAF user registry is part of the CICS Liberty security feature, unless distributed identity mapping is used. Any of the application security mechanisms supported by Liberty are supported in CICS, this includes HTTP basic authentication, form login, SSL client certificate authentication, identity assertion using a custom login module, JACC, JASPIC, or a Trust Association Interceptor (TAI). All SAF user IDs authenticated by Liberty must be granted read access to the Liberty JVM server APPL class profile. The name of this is determined by the profilePrefix setting in the safCredentials element in the Liberty server configuration file server.xml.

   ```
   <safCredentials profilePrefix="BBGZDFLT"/>
   ```

   The APPL class is also used by CICS terminal users to control access to specific CICS regions and your Liberty JVM server can use the same profile as the CICS APPLID depending upon your security requirements. If you do not specify this element, then the default profilePrefix of BBGZDFLT is used.

   You must define the APPLID and users must have access to the it. To configure and activate the BBGZDFLT profile in the APPL class:

   ```
   RDEFINE APPL BBGZDFLT UACC(NONE)
   SETROPTS CLASSACT(APPL)
   ```

   The users must be given read access to the BBGZDFLT profile in the APPL class in order to authenticate. To allow user AUSER to authenticate against the BBGZDFLT APPL class profile:

   ```
   PERMIT BBGZDFLT CLASS(APPL) ACCESS(READ) ID(AUSER)
   ```

   The Liberty SAF unauthenticated user id must be given read access to the APPL class profile. The SAF unauthenticated user id can be specified in the safCredentials element in the Liberty server configuration file server.xml.

   ```
   <safCredentials unauthenticatedUser="WSGUEST"/>
   ```

   If you do not specify the element, then the default unauthenticatedUser is WSGUEST. To allow the SAF unauthenticated user id WSGUEST read access to the BBGZDFLT profile in the APPL class:

   ```
   PERMIT BBGZDFLT CLASS(APPL) ACCESS(READ) ID(WSGUEST)
   ```

   If you use WSGUEST, then you should follow the steps to configure the SAF user registry as described in Setting up the System Authorization Facility (SAF) unauthenticated user.

The WLP z/OS System Security Access Domain (WZSSAD) refers to the permissions granted to the Liberty server. These permissions control which System Authorization Facility (SAF) application domains and resource profiles the server is permitted to query when authenticating and authorizing users. The CICS region user ID must be granted permission within the WZSSAD domain to make authentication calls. To grant permission to authenticate, the CICS region ID must be granted READ access to the BBG.SECPFX.<APPL> profile in the SERVER class:

```
RDEFINE SERVER BBG.SECPFX.BBGZDFLT UACC(NONE)
PERMIT BBG.SECPFX.BBGZDFLT CLASS(SERVER) ACCESS(READ) ID(cics_region_user)
```

For more details refer to Liberty: Accessing z/OS security resources using WZSSAD.

2. If an unauthenticated subject is supplied from Liberty, then the USERID defined in the URIMAP will be used.
3. If no USERID is defined in the URIMAP the request will run under the CICS default user ID.

Note:

Due to the way that security processing for Liberty transactions is deferred during CICS transaction attach processing, the user ID used in the CICS Monitoring Facility (CMF) records, the z/OS Workload Manager (WLM) classification, and the task association data and the UEPUSID global user exits field for the XAPADMGR exit, will be determined as follows; the user ID in the HTTP security header, or if there isn't one, the user ID taken from matching URIMAP. If neither exist, the CICS default user ID will be used.

Be aware that Liberty caches authenticated user IDs and, unlike CICS, does not check for an expired user ID within the cache period. You can configure the cache timeout by using the standard Liberty configuration process. Please see Configuring the authentication cache in Liberty.

Authorizing users to run applications in a Liberty JVM server

You can use Java EE application security roles to authorize access to Java EE applications. Additionally, in a Liberty JVM server you can further restrict access to transactions (run as part of the application) by using CICS transaction and resource security.

About this task

Your application is secured by providing an authorization constraint, the `<auth_constraint>` element, in the deployment descriptor (`web.xml`). If present, this ensures that access to your application is achieved only by a user that is a member of an authorized role. User or group membership to a Java EE role is determined in one of two ways:

- Use an `<application-bnd>` element in the `<application>` element of your `server.xml` to describe the user/group to role mappings directly in XML.
- Use `<safAuthorization>` in your `server.xml` to allow users/groups role membership to be mapped by SAF (typically using EJBROLES).

For more information, see Authorization using SAF role mapping.

Using CICS security allows you to re-use existing security procedures but requires that individual web applications are accessed from different URIMAPs. Using role-based security allows you to use existing standard Java EE security definitions from another Java EE application server. For more information, see “Authenticating users in a Liberty JVM server” on page 216.

If you want to use CICS transaction and resource authorization exclusively, or prefer to use finer-grained annotation-based role checking in code, you can defer the authorization decision to those components by using the special subject ALL_AUTHENTICATED_USERS role, as shown in the following example. If you deploy a Liberty application in a CICS bundle, CICS automatically configures this for you.

Note: Access checks are performed for the declarative security annotations and CICS transaction and resource security only after the configured constraints (`web.xml`) are verified

```
<application id="com.ibm.cics.server.examples.wlp.tsq.app"
    name="com.ibm.cics.server.examples.wlp.tsq.app" type="eba"
```

```
    location="${server.output.dir}/installedApps/com.ibm.cics.server.examples.wlp.tsq.app.eba">
    <application-bnd>
        <security-role name="cicsAllAuthenticated">
            <special-subject type="ALL_AUTHENTICATED_USERS"/>
        </security-role>
    </application-bnd>
</application>
```

Using this special subject, and giving the cicsAllAuthenticated role access to all URLs in your web applications deployment descriptor (`web.xml`), allows access to the web application using any authenticated user ID and authorization to the transaction must be controlled using CICS transaction security. If you deploy your application directly to the dropins directory, it is not configured to use CICS security as dropins does not support security.

If you are using safAuthorization then the `<application-bnd>` no longer acts as the source of user ID to role mapping. Instead, EJBROLEs in SAF determine which SAF users are in which roles (EJBROLEs). With safAuthorization the `<application-bnd>` is ignored. To achieve the same effect and allow all authenticated users to be authorized to run your application, the `<auth-constraint>` in `web.xml` must use the special role `**`, for example:

```
<auth-constraint>
        <description>special role for all authenticated users</description>
        <role-name>**</role-name>
</auth-constraint>
```

- The special role name `**` is a shorthand for any authenticated user independent of role.
- The special role name `*` is a shorthand for all role names defined in the deployment descriptor.

When the special role name `**` appears in an authorization constraint, it indicates that any authenticated user, independent of role, is authorized to perform the constrained requests. Special roles do not need an additional `<security-role>` declaration in `web.xml`.

To use CICS transaction or resource security you should follow the following steps:

Procedure

1. Define a URIMAP of type JVMSERVER for each web application. Typically, you might specify a URIMAP to match the generic context root (URI) of a web application to scope the transaction ID to the set of servlets that make up the application. Or you may choose to run each individual servlet under a different transaction with a more precise URI.
2. Authorize all users of the web application to use the transaction specified in the URIMAP using CICS transaction or resource security profiles.

Authorization using OAuth 2.0

OAuth 2.0 is an open standard for delegated authorization. With the OAuth authorization framework, a user can grant a third-party application access to information that is stored with another HTTP service without sharing their access permissions or the full extent of their data.

WebSphere Liberty supports OAuth 2.0, and can be used as an OAuth service provider endpoint and an OAuth protected resource enforcement endpoint. Liberty supports persistent OAuth 2.0 services. See Configuring persistent OAuth 2.0 services for more information. Clients can be defined locally with the localStore and client elements. This guide will use local clients to enable OAuth 2.0 authorization.

Before you begin

SAF security is a common use-case in CICS, and this procedure uses SAF in the examples.

Ensure that the CICS region is configured to use SAF security and is defined with SEC=YES as a system initialization parameter.

Optionally, you can grant an administrator user access to the SAF EJBROLE `BBGZDFLT.com.ibm.ws.security.oauth20.clientManager`. The security role `clientManager`

controls access to the management interfaces, allowing local clients to be queried, and persistent local clients to be created. The administrator user controls OAuth 2.0 local clients.

Configure the Liberty angel process to provide authentication and authorization services to the Liberty JVM server; for more information see The Liberty server angel process.

For background information on OAuth, see oauth-2.0.

About this task

The following procedure covers how to:

- Create an OAuth 2.0 service provided in a Liberty JVM server
- Create a locally configured client
- Use this local client to grant an OAuth 2.0 token to a relying party application, also known as a third-party web application
- Use this token to access protected resources in an application

Restriction: Db2 JDBC type 2 connectivity is not supported for persistent OAuth 2.0 services.

Procedure

1. Configure an OAuth 2.0 service provider.
   a) Add the `oauth-2.0` and the `cicsts:security-1.0` features to the `featureManager` element in `server.xml`.

```
<featureManager>
    ...
    <feature>oauth-2.0</feature>
    <feature>cicsts:security-1.0</feature>
</featureManager>
...
```

   b) Configure an OAuth 2.0 provider in `server.xml`.

```
<oauthProvider id="myProvider">
</oauthProvider>
```

2. Configure a local client for the relying party application. Local clients define the details of the relying party application, including the name, secret password and redirect URI of the application.
   a) Define a meaningful local client name and create a secret password which is used by the server for authorization. The local client application listens on a URI, and the server supplies authorization codes.
   b) Configure an OAuth 2.0 local client in the `oauthProvider` element of `server.xml`, supplying the local client ID, secret password and the redirect URI.

```
<oauthProvider id="myProvider">
    <localStore>
        <client id="myClient" redirect="https://client.example.ibm.com/webApp/redirect"
secret="mySecret" />
    </localStore>
</oauthProvider>
```

   Important:

   Although it is not shown in this example, it is important to encode passwords and limit access to `server.xml` configuration. Passwords can be encoded using the Liberty `securityUtility`, found in `USS_HOME/wlp/bin/securityUtility`. For more information, see Liberty: securityUtility command.

   Note: More than one local client can be configured in the `localStore` element.
3. When the relying party application requires access to protected resources on the server, the user must authorize access to these resources first.

a) The relying party application requires the user to authenticate with the server, and select the type of access for the relying party application by linking or redirecting the user to the authorization endpoint:

```
https://hostname:port/oauth2/endpoint/provider_name/authorize
```

or

```
https://hostname:port/oauth2/declarativeEndpoint/provider_name/authorize
```

Additional parameters are required in the query parameters of the URL. For the local client that was configured in step two, the following GET request is required:

```
https://zos.example.ibm.com/oauth2/endpoint/myProvider/authorize?
response_type=code&client_id=myClient&client_secret=mySecret&redirect_uri=https://
client.example.ibm.com/webApp/redirect
```

When the user has selected the access for the relying party application, they are redirected back to the relying party application using the redirect URI:

```
https://client.example.ibm.com/webApp/redirect?code=access_code
```

The relying party application must store this access code to request an OAuth token.

Note: For local clients, the user must exist in a user register in the Liberty JVM server. See Authenticating users in a Liberty JVM server for more information on authenticating users in Liberty JVM servers.

b) The relying party application requests an OAuth 2.0 token by sending a POST request to the server:

```
https://hostname:port/oauth2/endpoint/provider_name/token
```

The relying party application sends the authorization code received from the authorization endpoint, the local client ID, and secret password in the POST data:

```
POST https://zos.example.ibm.com/oauth2/endpoint/myProvider/token HTTP/1.1
Content-Type: application/www-form-urlencoded

grant_type=authorization_code&code=code&client_id=myClient&client_secret=mySecret&redirect
_url=https://client.example.ibm.com/webApp/redirect
```

This returns a JSON document containing the token.

4. Use the token to access protected resources.

a) Add the token to the `Authorization` header on the HTTP request.

```
Authorization: Bearer <token>
```

Results

Users are able to authorize third-party applications to access their protected resources in a Liberty JVM server through OAuth 2.0 authorizations flows. The Liberty JVM server can configure the provider of these tokens and create locally configured clients.

There are several methods of granting tokens, for more information, see OAuth 2.0 service invocation.

Configuring persistent OAuth 2.0 services

WebSphere Liberty supports persisting OAuth 2.0 local clients and tokens to a database. With persistent OAuth 2.0, an authorized local client can continue to access OAuth 2.0 services after a restart.

Before you begin

SAF security is a common use-case in CICS, and this procedure uses SAF in the examples.

- Gain the necessary access to create tables and read/write to these tables in a database and configure it in the Liberty `server.xml`.
- Grant access to the SAF EJBROLE `BBGZDFLT.com.ibm.ws.security.oauth20.clientManager` to an administrator user to control OAuth 2.0 local clients.
- Create an OAuth 2.0 provider in the Liberty `server.xml`. For more information, see Authorization using OAuth 2.0.

About this task

The following steps create a persistent OAuth 2.0 local client. This local client is used to grant OAuth 2.0 tokens.

Restriction: Db2 JDBC type 2 connectivity is not supported for persistent OAuth 2.0 services.

Procedure

1. Create the necessary tables using IBM Db2 for persistent OAuth services as a guide.
2. Create a persistent local client by sending a POST request to the URL:

   ```
   https://hostname:port/oauth2/endpoint/provider_name/registration
   ```

   Use the JSON document which is described in the first table in Configuring an OpenID Connect Provider to accept client registration requests; for example:

   ```
   {
       "client_id": "client_id",
       "client_secret": "client_secret",
       "grant_types": [ "authorization_code", "refresh_token" ],
       "redirect_uris": [ "https://client.example.ibm.com/webApp/redirect" ]
   }
   ```

Results

A persistent OAuth 2.0 local client is created. When this local client is used to produce tokens, the tokens are persisted to the database. If the server restarts, the persistent local client and tokens remain valid.

Authorization using SAF role mapping

Mapping Java EE roles to users and groups can be achieved in different ways. In distributed systems, a basic registry or LDAP registry would typically be used in conjunction with an application specific `<application-bnd>` element, to map users from those registries into *roles*. The deployment descriptor of the application determines which roles can access which parts of the application.

About this task

On z/OS, there is an additional registry type, the System Authorization Facility (SAF) registry. A Liberty JVM server implicitly uses this type for authentication when the `cicsts:security-1.0` feature is installed unless configured to use LDAP. You can choose to make use of SAF authorization. When using SAF authorization, user to role mappings are used to map roles to EJBROLE resource profiles using the SAF role mapper. The server queries SAF to determine if the user has the required READ access to the EJBROLE resource profile.

In a Liberty JVM server, if you want to use Java EE roles without SAF authorization, you cannot use CICS bundles to install your applications. This is because a CICS bundle installed application automatically creates an `<application-bnd>` element and uses the `ALL_AUTHENTICATED_USERS` special-subject, which prevents you from defining the element yourself. Instead, you must create an `<application>` element in `server.xml` directly and configure the `<application-bnd>` with the roles and users you require.

If, however, you choose to use Java EE roles and SAF authorization, you can continue to use CICS bundles to lifecycle your web applications. The `<application-bnd>` is ignored by Liberty in favor of using

the role mappings determined by the SAF registry. Role mappings are determined by virtue of a user belonging to an EJB role.

Tip: Special subjects ALL_AUTHENTICATED_USERS and EVERYONE can not be used when SAF authorization is enabled.

Tip: It is advisable to create or update your EJB roles before starting the CICS region. Liberty issues a RACROUTE REQUEST=LIST with GOBAL=NO in order to support a minimum version of z/OS. The address space will not see updates until it is restarted (or started).

Procedure

1. Add the <safAuthorization id="saf"/> element to your server.xml. If you are using the cicsts:distributedIdentity-1.0 feature, this is defined for you.
2. Create the EJB roles you require, with reference to the prefix scheme described.
3. Add users to those EJB roles.

   By default, if SAF authorization is used the application will use the pattern <profile_prefix>.<resource>.<role> to determine if a user is in a role. The profile_prefix defaults to BBGZDFLT but can be modified using the <safCredential> element. For more information, see Liberty: Accessing z/OS security resources using WZSSAD.

   The role mapping preferences can be modified using the <safRoleMapper> element in the server.xml that defaults to

   ```
   <safRoleMapper profilePattern="myprofile.%resource%.%role%" toUpperCase="true"/>
   ```

   Users can then be authorized to a particular EJB role using the following RACF commands, where WEBUSER is the authenticated user ID.

   ```
   RDEFINE EJBROLE BBGZDFLT.MYAPP.ROLE UACC(NONE)
   PERMIT BBGZDFLT.MYAPP.ROLE CLASS(EJBROLE) ACCESS(READ) ID(WEBUSER)
   ```

4. Optional: If you are deploying the CICS servlet examples and want to use the Java EE role security with SAF authorization, create a SAF EJBROLE for each servlet that you have deployed. For example, if you use the default APPL class of BBGZDFLT, define the following EJBROLE security definitions:

```
RDEFINE EJBROLE BBGZDFLT.com.ibm.cics.server.examples.wlp.hello.war.cicsAllAuthenticated UACC(NONE)
RDEFINE EJBROLE BBGZDFLT.com.ibm.cics.server.examples.wlp.tsq.app.cicsAllAuthenticated UACC(NONE)
RDEFINE EJBROLE BBGZDFLT.com.ibm.cics.server.examples.wlp.jdbc.app.cicsAllAuthenticated UACC(NONE)
SETROPTS RACLIST(EJBROLE) REFRESH
```

   Give read access to the defined roles for each web user ID that requires authorization:

```
PERMIT BBGZDFLT.com.ibm.cics.server.examples.wlp.hello.war.cicsAllAuthenticated
       CLASS(EJBROLE) ID(user) ACCESS(READ)
PERMIT BBGZDFLT.com.ibm.cics.server.examples.wlp.tsq.app.cicsAllAuthenticated
       CLASS(EJBROLE) ID(user) ACCESS(READ)
PERMIT BBGZDFLT.com.ibm.cics.server.examples.wlp.jdbc.app.cicsAllAuthenticated
       CLASS(EJBROLE) ID(user) ACCESS(READ)
SETROPTS RACLIST(EJBROLE) REFRESH
```

Results

You can authorize access to web applications using CICS Security, Java EE role security, or both by defining the roles and the users in the roles.

Configuring security for a Liberty JVM server by using an LDAP registry

Liberty uses a user registry to authenticate a user and retrieve information about users and groups to perform security-related operations, including authentication and authorization. Default CICS Liberty security uses the SAF registry. However, many transactions that run on CICS are initiated by users

who authenticate their identities on distributed application servers, so CICS also supports the use of a Lightweight Directory Access Protocol (LDAP) registry in Liberty. To use LDAP, it is necessary to manually configure the `server.xml`.

Before you begin

- Ensure that the CICS region is configured to use SAF security and is defined with SEC=YES as a system initialization parameter.
- Authorize application developers and system administrators to create, view, update, and remove JVMSERVER and BUNDLE resources to deploy web applications into a Liberty JVM server. The JVMSERVER resource controls the availability of the JVM server, and the BUNDLE resource is a unit of deployment for the Java applications and controls the availability of the applications.

About this task

This task explains how to configure LDAP security for a Liberty JVM server, and integrate Liberty security with CICS security. Distributed identity mapping can be used to associate a SAF user ID with a distributed identity. You can use the CICS distributed identity mapping feature to set up distributed identity mapping. A user can then log on to a CICS web application with their distributed identity, as authenticated by an LDAP server. Filters that are defined in the z/OS security product (RACMAP) determine the mapping of this identity to a SAF user ID. This SAF user ID can then be used to authorize access to web applications through JEE application role security, providing integration with CICS transaction and resource security. You can map a SAF user ID to one or more distributed identities.

The default transaction ID for running any web request is CJSA. You can configure CICS to run web requests under a different transaction ID by using a URIMAP of type JVMSERVER. You can specify a URIMAP to match the generic context root (URI) of a web application to scope the transaction ID to the set of servlets that make up the application. Or you can choose to run each individual servlet under a different transaction with a more precise URI.

There are three scenarios for this task:

- Scenario 1 – Distributed identity mapping with SAF authorization
- Scenario 2 – Distributed identity mapping without SAF authorization
- Scenario 3 – LDAP for authentication and authorization

Procedure

1. **Distributed identity mapping with SAF authorization**

   You can use the CICS distributed identity mapping feature, `cicsts:distributedIdentity-1.0` to enable LDAP distributed identities to be mapped to SAF user IDs. When used with the CICS security feature `cicsts:security-1.0`, Liberty LDAP security is used for authentication and JEE application role security from EJB role mappings are respected for authorization. CICS transactions run under the mapped SAF user ID providing integration with CICS transaction and resource security.

   a. Configure the WebSphere Liberty angel process to provide authentication and authorization services to the Liberty JVM server, for more information see The Liberty server angel process.

   b. Add the `cicsts:security-1.0` and the `cicsts:distributedIdentity-1.0` feature to the featureManager list in the `server.xml`.

```
<featureManager>
   ...
   <feature>cicsts:security-1.0</feature>
   <feature>cicsts:distributedIdentity-1.0</feature>
</featureManager>
...
```

c. Configure Liberty to use LDAP authentication by defining the LDAP server in the `server.xml`, for example:

```
<ldapRegistry id="ldap"
              host="host.domain.com" port="389"
              ldapType="IBM Tivoli Directory Server"
              baseDN="ou=users,dc=domain,dc=com"
              ignoreCase="true">
</ldapRegistry>
```

Full details on configuring LDAP user registries with Liberty are available in Configuring LDAP user registries in Liberty.

d. Remove the safRegistry element, if present. Save the changes to the `server.xml`.

e. Make the necessary RACF definitions, including setting up the RACMAPs to map distributed identities to SAF user IDs as which are described in `Configuring LDAP user registries in Liberty` and providing access for these user IDs to the appropriate EJBROLES as described in "Authorization using SAF role mapping" on page 222. CICS configures SAF authorization and the mapDistributedIdentities attributes in the safCredentials configuration element for you.

When the `cicsts:distributedIdentity-1.0` feature is used with the `cicsts:security-1.0` feature, Liberty LDAP security is used for authentication, and JEE application role security from EJB role mappings are respected for authorization. CICS transactions run under the RACMAP mapped user ID providing integration with CICS transaction and resource security.

What to do next

Back to top

2. **Distributed identity mapping without SAF authorization**

It is possible to allow CICS transactions to run under a RACMAP mapped user ID while respecting the roles configured in the application's `<application-bnd>` element. This might be useful when migrating work from distributed Liberty to CICS Liberty. Be aware that CICS bundles cannot be used to install applications as SAF authorization is not being used. See "Authorization using SAF role mapping" on page 222 for more details.

a. Configure the WebSphere Liberty angel process to provide authentication and authorization services to the Liberty JVM server, for more information, see The Liberty server angel process.

b. Add the `cicsts:security-1.0` and the `ldapRegistry-3.0` feature to the featureManager list in the `server.xml`.

```
<featureManager>
   ...
   <feature>cicsts:security-1.0</feature>
   <feature>ldapRegistry-3.0</feature>
</featureManager>
...
```

c. Configure Liberty to use LDAP authentication by defining the LDAP server in the `server.xml`, for example:

```
<ldapRegistry id="ldap"
              host="host.domain.com" port="389"
              ldapType="IBM Tivoli Directory Server"
              baseDN="ou=users,dc=domain,dc=com"
              ignoreCase="true">
</ldapRegistry>
```

Full details on configuring LDAP user registries with the Liberty are available in Configuring LDAP user registries in Liberty.

d. Configure Liberty to use distributed identity filters to map the distributed identities to SAF user IDs by setting the `mapDistributedIdentities` attribute in the `safCredentials` configuration element to `true` in the `server.xml`.

e. Remove the safRegistry element, if present. Save the changes to the `server.xml`.

f. Make the necessary RACF definitions, including setting up the RACMAPs to map distributed identities to SAF user IDs as which are described in Configuring LDAP user registries in Liberty.

g. If JEE application role security is required for authorization then refer to the topic "Authorization using SAF role mapping" on page 222. Be aware that CICS bundles cannot be used to install applications when SAF is not used for JEE role authorization.

Applications use Liberty LDAP security for authentication, and JEE application role security in an <application-bnd> element are respected for authorization of the distributed identity. In CICS, transactions run under the RACMAP mapped user ID, providing integration with CICS transaction and resource security.

What to do next

Back to top

3. **LDAP for authentication and authorization**

LDAP security can be used in a CICS Liberty JVM server for both authentication and authorization using JEE application role security. URIMAP definitions can then be used to set the user ID under which transaction run. This scenario might be useful if migrating a distributed application into a CICS Liberty JVM server, without requiring any significant security resource changes.

a. Add the `cicsts:security-1.0` and the `ldapRegistry-3.0` feature to the featureManager list in the `server.xml`.

```
<featureManager>
   ...
   <feature>cicsts:security-1.0</feature>
   <feature>ldapRegistry-3.0</feature>
</featureManager>
...
```

b. Configure Liberty to use LDAP authentication by defining the LDAP server in the `server.xml`, for example:

```
  <ldapRegistry id="ldap"
                host="host.domain.com" port="389"
                ldapType="IBM Tivoli Directory Server"
                baseDN="ou=users,dc=domain,dc=com"
                ignoreCase="true">
  </ldapRegistry>
```

Full details on configuring LDAP user registries with Liberty are available in Configuring LDAP user registries in Liberty.

c. Remove the safRegistry element, if present. Save the changes to the `server.xml`.

d. To configure JEE application role security for authorization refer to the topic "Authorization using SAF role mapping" on page 222. Be aware that CICS bundles cannot be used to install applications when SAF is not used for JEE role authorization.

Applications use Liberty LDAP security for authentication, and JEE application role security in an <application-bnd>element are respected for authorization. In CICS transactions run under the URIMAP or CICS DFLTUSER user ID as appropriate.

What to do next

Back to top

What to do next

This applies to all three scenarios:

- Modify the Liberty authentication cache.
- Set up URIMAP definitions to map web application URIs to transaction IDs.

This applies to scenarios 1 and 2:

- Set up CICS transaction security definitions to authorize access to URIs based on the mapped user ID.

Back to top

Configuring SSL (TLS) for a Liberty JVM server using a Java keystore

You can configure a Liberty JVM server to use SSL for data encryption, and optionally authenticate with the server by using a client certificate. Certificates can be stored in a Java keystore or in a SAF key ring such as RACF.

About this task

Enabling SSL in a Liberty JVM server requires adding the **ssl-1.0** Liberty feature, a keystore, and an HTTPS port. CICS automatically creates and updates the `server.xml` file. Autoconfiguring always results in the creation of a Java keystore.

It is important to understand that any web request to a Liberty JVM server uses the JVM support for TCP/IP sockets and SSL processing, not CICS sockets domain.

Procedure

- To use autoconfigure to configure SSL, complete the following steps:
  a) Ensure autoconfigure is enabled in the JVM profile by using the JVM system property **-Dcom.ibm.cics.jvmserver.wlp.autoconfigure=true**.
  b) Set the SSL port by setting the JVM system property **-Dcom.ibm.cics.jvmserver.wlp.server.https.port** in the JVM profile.
  c) Restart the JVM server to add the necessary configuration elements to `server.xml`.

Results

SSL for a Liberty JVM server is successfully configured.

Configuring SSL (TLS) for a Liberty JVM server using RACF

You can configure a Liberty JVM server to use SSL for data encryption, and optionally authenticate with the server by using a client certificate. Certificates can be stored in a Java keystore or in a SAF key ring such as RACF.

About this task

Enabling SSL in a Liberty JVM server requires adding the **ssl-1.0** Liberty feature, a keystore, and an HTTPS port. Configuring `server.xml` manually. You edit the `server.xml` file to add the required elements and values. You must follow the manual procedure if you want to use a RACF key ring.

It is important to understand that any web request to a Liberty JVM server uses the JVM support for TCP/IP sockets and SSL processing, not CICS sockets domain.

Procedure

- To manually configure SSL, you need to create a signing certificate. Use this signing certificate to create a server certificate. Then, export the signing certificate to the client web browser where it is used to authenticate the server certificate.
  a) Create a certificate authority (CA) certificate (signing certificate). An example, using RACF commands, follows:

```
RACDCERT GENCERT
  CERTAUTH
  SUBJECTSDN(CN('CICS Sample Certification Authority')
```

```
   O('IBM')
   OU('CICS'))
   SIZE(2048)
  WITHLABEL('CICS-Sample-Certification')
```

The SIZE of the certificate should be a minimum of 2048 bits. For more information, see the RACF RACDCERT GENCERT (Generate certificate) command.

b) Create a server certificate that uses the signing certificate from step 2, where <userid> is the CICS region user ID. The hostname is the host name of the server that the Liberty server HTTPS port is configured to use.

```
RACDCERT ID(<userid>)
GENCERT
  SUBJECTSDN(CN('<hostname>')
   O('IBM')
   OU('CICS'))
   SIZE(2048)
   SIGNWITH (CERTAUTH LABEL('CICS-Sample-Certification'))
  WITHLABEL('<userid>-Liberty-Server')
```

The SIZE of the certificate should be a minimum of 2048 bits. For more information, see the RACF RACDCERT GENCERT (Generate certificate) command.

c) Connect the signing certificate and server certificate to a RACF key ring.

You can use RACF with the following command, and replace the value of <keyring> with the name of the key ring you want to use. Replace the value of <userid> with the CICS region user ID.

```
RACDCERT ID(<userid>) CONNECT(RING(<keyring>)
       LABEL('CICS-Sample-Certification')
       CERTAUTH)

RACDCERT ID(<userid>) CONNECT(RING(<keyring>)
       LABEL('<userid>-Liberty-Server'))
```

Export the signing certificate to a CER file:

```
RACDCERT CERTAUTH EXPORT(LABEL('CICS-Sample-Certification'))
       DSN('<userid>.CERT.LIBCERT')
       FORMAT(CERTDER)
       PASSWORD('password')
```

FTP the exported certificate in binary to your workstation, and import it into your browser as a certificate authority certificate.

d) Edit the server.xml file and add the SSL feature, and the keystore. Set the HTTPS port (value is 9443 in the following example) and restart your CICS region. The SAF key ring must be specified in the URL form safkeyring://<userid>/<keyring>. The <userid> value must be set to the CICS region user ID and the <keyring> value must be set to the name of the key ring. The password field is not used for accessing the SAF key ring and must be set to password.

```
<featureManager>
  ...
       <feature>ssl-1.0</feature>
</featureManager>
...
<httpEndpoint host="*" httpPort="9080" httpsPort="9443"
    id="defaultHttpEndpoint"/>
...
.
<keyStore filebased="false" id="racfKeyStore"
        location="safkeyring://<userid>/<keyring>"
        password="password"
        readOnly="true"
        type="JCERACFKS"/>
<ssl id="defaultSSLConfig" keyStoreRef="racfKeyStore"
      sslProtocol="SSL_TLS"
      serverKeyAlias="<userid>-Liberty-Server" />
```

Results

SSL for a Liberty JVM server is successfully configured.

Setting up SSL (TLS) client certificate authentication in a Liberty JVM server

SSL client certificate authentication allows the client and server to provide certificates to the opposite party for mutual verification. It is often used in situations where an extra level of authentication is required because of security concerns.

Before you begin

You must complete the task Configuring SSL (TLS) for a Liberty JVM server using RACF. If you do not already have your CICS Liberty security set up, you must complete Configuring security for a Liberty JVM server before proceeding.

About this task

The following setup information assumes that you are using RACF keystores to store your certificates for SSL client certificate authentication.

Procedure

1. Create a personal certificate using a signing certificate and associate the personal certificate with a RACF user ID.

   Then, export the personal certificate to a data set in CER format and then FTP in binary to your work station. Import the personal certificate to the web browser as a personal certificate. When the certificate is imported into the web browser, it can supply an SSL client certificate and connect to the HTTPS port in the Liberty server. Use the following RACF command, where `<clientuserid>` is the RACF user ID and `<hostname>` is the host name of the client computer.

   ```
   RACDCERT ID(<clientuserid>)
   GENCERT
     SUBJECTSDN(CN('<hostname>')
      O('IBM')
      OU('CICS'))
      SIZE(2048)
      SIGNWITH (CERTAUTH LABEL('CICS-Sample-Certification'))
     WITHLABEL('<clientuserid>-certificate')
   ```

   Export the personal certificate as you have done earlier in this step.

   ```
   RACDCERT ID(<clientuserid>)
     EXPORT(LABEL('<clientuserid>-certificate'))
      DSN('USERID.CERT.CLICERT')
      FORMAT(PKCS12DER)
      PASSWORD('password')
   ```

   Update the `server.xml` SSL element to support SSL client certificate authentication:

   ```
     <ssl id="defaultSSLConfig" keyStoreRef="racfKeyStore"
        sslProtocol="SSL_TLS"
        serverKeyAlias="<userid>-Liberty-Server"
        clientAuthenticationSupported="true"/>
   ```

   Additionally, if you want to ensure all clients must supply a valid SSL client certificate, add the **`clientAuthentication`** attribute to the SSL element as follows:

   ```
     <ssl id="defaultSSLConfig" keyStoreRef="racfKeyStore"
        sslProtocol="SSL_TLS"
        serverKeyAlias="<userid>-Liberty-Server"
        clientAuthenticationSupported="true"
        clientAuthentication="true"/>
   ```

2. You can authenticate a web request in CICS under the identity of the client user ID in step 2. Then, deploy the web application with a `login-config` element for CLIENT-CERT in the `web.xml`. The `web.xml` file can be found inside the source files for the web application that you are deploying.

```
<login-config>
          <auth-method>CLIENT-CERT</auth-method>
</login-config
```

   Instead, if you want to allow failover to HTTP basic authentication if SSL client certificate authentication is not configured, add the webAppSecurity element to `server.xml`.

```
<webAppSecurity allowFailOverToBasicAuth="true" />
```

3. Finally, set up CICS transaction security to authorize access to the CICS transaction based on the authenticated client user ID.

   For further information, see "Authorizing users to run applications in a Liberty JVM server" on page 218.

Using the syncToOSThread function

You can use the syncToOSThread function of Liberty in a CICS Liberty JVM server. SyncToOSThread enables a Java subject, authenticated by Liberty, to be synchronized with the operating system (OS) thread identity. Without syncToOSThread, the operating system thread identity defaults to be the CICS region user ID, this is the identity used to authorize access to resources outside of CICS control such as zFS files. With syncToOSThread in effect, the user's subject is used to access these operating system resources.

About this task

Enabling syncToOSThread requires the Liberty appSecurity-1.0 and zosSecurity-1.0 features. These features are included with the cicsts:security-1.0 feature. You must also define the syncToOSThread configuration element in the Liberty `server.xml` and add a special `<env-entry/>` to the application's deployment descriptor (`web.xml`). In addition, the SAF registry must be used for authentication, the angel process must be up and running, and the server must be connected to the angel process. For more information about the angel process, see Liberty: Process types on z/OS.

Procedure

1. Configure the syncToOSThread configuration element in the Liberty `server.xml` and add the required `<env-entry/>` to each web application's deployment descriptor by following steps 1 and 2 in Enabling syncToOSThread for applications
2. Grant the Liberty server permission to perform syncToOSThread operations by configuring SAF with either of the following profiles:
   - Grant the CICS region user ID CONTROL access to the BBG.SYNC.*<profilePrefix>* profile in the FACILITY class, where *<profilePrefix>* is specified on the `<safCredentials />` element. This allows the Liberty server to synchronize any Java subject with the OS thread identity:

     ```
     PERMIT BBG.SYNC.<profilePrefix> ID(<serverUserId>) ACCESS(CONTROL) CLASS(FACILITY)
     ```

   - Grant the CICS region user ID READ access to the BBG.SYNC.*<profilePrefix>* profile in the FACILITY class. Additionally, grant the CICS region user ID READ access to one or more BBG.SYNC.*<AuthUserid/>* profiles in the SURROGATE class, one for each authenticated user ID to be synchronized with the OS identity:

     ```
     PERMIT BBG.SYNC.<profilePrefix> ID(<serverUserId>) ACCESS(READ) CLASS(FACILITY)
     PERMIT BBG.SYNC.<AuthUserid> ID(<serverUserId>) ACCESS(READ) CLASS(SURROGAT)
     ```

   Restriction: A servlet configured as the welcome page in `web.xml`, does not support the syncToOSThread function.

Enabling a Java security manager

By default, Java applications have no security restrictions placed on activities requested of the Java API. To use Java security to protect a Java application from performing potentially unsafe actions, you can enable a security manager for the JVM in which the application runs.

About this task

The security manager enforces a security policy, which is a set of permissions (system access privileges) that are assigned to code sources. A default policy file is supplied with the Java platform. However, to enable Java applications to run successfully in CICS when Java security is active, you must specify an additional policy file that gives CICS the permissions it requires to run the application.

You must specify this additional policy file for each kind of JVM that has a security manager enabled. CICS provides some examples that you can use to create your own policies.

Notes: Enabling a Java security manager is not supported in a Liberty JVM server.

- The OSGi security agent example creates an OSGi middleware bundle called `com.ibm.cics.server.examples.security` in your project that contains a security profile. This profile applies to all OSGi bundles in the framework in which it is installed.
- The `example.permissions` file contains permissions that are specific to running applications in a JVM server, including a check to ensure that applications do not use the `System.exit()` method.
- CICS must have read and execute access to the directory in zFS where you deploy the OSGi bundle.

For applications that run in the OSGi framework of a JVM server:

Procedure

1. Create a plug-in project in the IBM CICS SDK for Java and select the supplied OSGi security agent example.
2. In the project, select the `example.permissions` file to edit the permissions for your security policy.
   a) Validate that the CICS zFS and Db2 installation directories are correctly specified.
   b) Add other permissions as necessary.
3. Deploy the OSGi bundle to a suitable directory in zFS such as `/u/bundles`.
4. Edit the JVM profile for the JVM server to add the OSGi bundle to the `OSGI_BUNDLES` option before any other bundles:

   ```
   OSGI_BUNDLES=/u/bundles/com.ibm.cics.server.examples.security_1.0.0.jar
   ```

5. Add the following Java property to the JVM profile to enable security.

   ```
   -Djava.security.policy=all.policy
   ```

6. Add the following Java environment variable to the JVM profile to enable security in the OSGi framework:

   ```
   org.osgi.framework.security=osgi
   ```

7. To allow the OSGi framework to start with Java 2 security, add the following policy:

   ```
   grant { permission java.security.AllPermission; };
   ```

8. Save your changes and enable the JVMSERVER resource to install the middleware bundle in the JVM server.
9. Optional: Activate Java 2 security.
   a) To activate a Java 2 security policy mechanism, add it to the appropriate JVM profile. You must also edit your Java 2 security policy to grant appropriate permissions.
   b) To use JDBC or SQLJ from a Java application with a Java 2 security policy mechanism active, use the IBM Data Server Driver for JDBC and SQLJ.
   c) To activate a Java 2 security policy mechanism, edit the JVM profile.

d) Edit the Java 2 security policy to grant permissions to the JDBC driver, by adding the lines that are shown in Example 1. In place of db2*xxx*, specify a directory below which all your Db2 libraries are located. The permissions are applied to all the directories and files below this level. This enables you to use JDBC and SQLJ.

e) Edit the Java 2 security policy to grant read permissions, by adding the lines that are shown in Example 2. If you do not add read permission, running a Java program produces AccessControlExceptions and unpredictable results. You can use JDBC and SQLJ with a Java 2 security policy.

Example 1:

```
grant codeBase "file:/usr/lpp/db2xxx/-" {
  permission java.security.AllPermission;
};
```

Example 2:

```
grant {

// allows anyone to read properties
permission java.util.PropertyPermission "*", "read";

};
```

Results

When the Java application is called, the JVM determines the code source for the class and consults the security policy before granting the class the appropriate permissions.

Chapter 7. Improving Java performance

You can take various actions to improve the performance of Java applications and the JVMs in which they run.

About this task

In addition to fine-tuning CICS itself, you can further improve the performance of Java applications in the following ways:

- Ensuring that the Java applications are well written
- Tuning the Java Runtime Environment (JVM)
- Tuning the language in which the JVM runs

Procedure

1. Determine the performance goals for your Java workload.

   Some of the most common goals include minimizing processor usage or application response times. After you decide on the goal, you can tune the Java environment.
2. Analyze your Java applications to ensure that they are running efficiently and do not generate too much garbage.

   IBM has tools that can help you to analyze Java applications to improve the efficiency and performance of particular methods and the application as a whole.
3. Tune the JVM server.

   You can use statistics and IBM tools to analyze the storage settings, garbage collection, task waits, and other information to tune the performance of the JVM.
4. Tune the Language Environment enclave in which a JVM runs.

   JVMs use MVS storage, obtained by calls to MVS Language Environment services. You can modify the runtime options for Language Environment to tune the storage that is allocated by MVS.
5. Optional: If you use the z/OS shared library region to share DLLs between JVMs in different CICS regions, you can tune the storage settings.

Determining performance goals for your Java workload

Tuning CICS JVMs to achieve the best overall performance for a given application workload involves several different factors. You must decide what the preferred performance characteristics of your Java workload are. When you establish these characteristics, you can determine what parameters to change and how to change them.

The following performance goals for Java workloads are most common:

Minimum overall processor usage
: This goal prioritizes the most efficient use of the available processor resource. If a workload is tuned to achieve this goal, the total use of the processor across the entire workload is minimized, but individual tasks might experience high processor consumption. Tuning for the minimum overall processor usage involves specifying large storage heap sizes for your JVMs to minimize the number of garbage collections.

Minimum application response times
: This goal prioritizes ensuring that an application task returns to the caller as rapidly as possible. This goal might be especially relevant if there are Service Level Agreements to be achieved. If a workload is tuned to achieve this goal, applications respond consistently and quickly, though a higher processor usage might occur for garbage collections. Tuning for minimum application response times involves keeping the heap size small and possibly using the gencon garbage collection policy.

Minimum JVM storage heap size

This goal prioritizes reducing the amount of storage used by JVMs. You can reduce the amount of storage that is used in the JVM, by reducing the JVM heap size.

Note: Reducing the JVM heap size might result in more frequent garbage collection events.

Other factors can affect the response times of your applications. The most significant of these is the Just In Time (JIT) compiler. The JIT compiler optimizes your application code dynamically at run time and provides many benefits, but it requires a certain amount of processor resource to do this.

Analyzing Java applications using IBM Health Center

To improve the performance of a Java application, you can use IBM Health Center to analyze the application. This tool provides recommendations to help you improve the performance and efficiency of your application.

About this task

IBM Health Center is available in the IBM Support Assistant Workbench. These free tools are available to download from IBM as described in the Getting Started guide for IBM Health Center. Try to run the application in a JVM on its own. If you are running a mixed workload in a JVM server, it might be more difficult to analyze a particular application.

Procedure

1. Add the required connection options to the JVM profile of the JVM server.

   The IBM Health Center documentation describes what options you must add to connect to the JVM from the tool.
2. Start IBM Health Center and connect it to your running JVM.

   IBM Health Center reports JVM activity in real time so wait a few moments for it to monitor the JVM.
3. Select the **Profiling** link to profile the application.

   You can check the time spent in different methods. Check the methods with the highest usage to look for any potential problems.

   Tip: The **Analysis and Recommendations** tab can identify particular methods that might be good candidates for optimization.
4. Select the **Locking** link to check for locking contentions in the application.

   If the Java workload is unable to use all the available processor, locking might be the cause. Locking in the application can reduce the amount of parallel threads that can run.
5. Select the **Garbage Collection** link to check the heap usage and garbage collection.

   The **Garbage Collection** tab can tell you how much heap is being used and how often the JVM pauses to perform garbage collection.

   a) Check the proportion of time spent in garbage collection.

      This information is presented in the Summary section. If the time spent in garbage collection is more than 2%, you might need to adjust your garbage collection.

   b) Check the pause time for garbage collection.

      If the pause time is more than 10 milliseconds, the garbage collection might be having an effect on application response times.

   c) Divide the rate of garbage collection by the number of transactions to find out approximately how much garbage is produced by each transaction.

      If the amount of garbage seems high for the application, you might have to investigate the application further.

What to do next

After you have analyzed the application, you can tune the Java environment for your Java workloads.

Garbage collection and heap expansion

Garbage collection and heap expansion are an essential part of the operation of a JVM. The frequency of garbage collection in a JVM is affected by the amount of garbage, or objects, created by the applications that run in the JVM.

Allocation failures

When a JVM runs out of space in the storage heap and is unable to allocate any more objects (an allocation failure), a garbage collection is triggered. The Garbage Collector cleans up objects in the storage heap that are no longer being referenced by applications and frees some of the space. Garbage collection stops all other processes from running in the JVM for the duration of the garbage collection cycle, so time spent on garbage collection is time that is not being used to run applications. For a detailed explanation of the JVM garbage collection process, see Generational Concurrent Garbage Collector and z/OS User Guide for IBM SDK, Java Technology Edition, Version 8.

When a garbage collection is triggered by an allocation failure, but the garbage collection does not free enough space, the Garbage Collector expands the storage heap. During heap expansion, the Garbage Collector takes storage from the maximum amount of storage reserved for the heap (the amount specified by the `-Xmx` option), and adds it to the active part of the heap (which began as the size specified by the `-Xms` option). Heap expansion does not increase the amount of storage required for the JVM, because the maximum amount of storage specified by the `-Xmx` option has already been allocated to the JVM at startup. If the value of the `-Xms` option provides sufficient storage in the active part of the heap for your applications, the Garbage Collector does not have to carry out heap expansion at all.

At some point during the lifetime of the JVM, the Garbage Collector stops expanding the storage heap, because the heap has reached a state where the Garbage Collector is satisfied with the frequency of garbage collection and the amount of space freed by the process. The Garbage Collector does not aim to eliminate allocation failures, so some garbage collection can still be triggered by allocation failures after the Garbage Collector has stopped expanding the storage heap. Depending on your performance goals, you might consider this frequency of garbage collection to be excessive.

Garbage collection options

You can use different policies for garbage collection that make trade-offs between throughput of the application and the overall system, and the pause times that are caused by garbage collection. Garbage collection is controlled by the `-Xgcpolicy` option:

`-Xgcpolicy:optthruput`
This policy delivers high throughput to applications but at the cost of occasional pauses, when garbage collection occurs.

`-Xgcpolicy:gencon`
This policy helps to minimize the time that is spent in any garbage collection pause. Use this garbage collection policy with JVM servers. You can check which policy is being used by the JVM server by inquiring on the JVMSERVER resource. The JVM server statistics have fields that tell you how many major and minor garbage collection events occur and what processor time is spent on garbage collection.

When you use this policy, it is also worth considering the `-Xgc:concurrentScavenge` setting - which is not a default setting - if your system has a large heap and is response-time sensitive. In these situations it can help to reduce garbage collection pause times. For more information, see -Xgc:concurrentScavenge.

`-XX:+HeapManagementMXBeanCompatibility`
This policy is set by default if you have APAR PI87181 applied and are using Java 8 SR5 and above. The policy ensures consistent garbage collection statistics with previous levels of Java. For more information, see -XX:[+|-]HeapManagementMXBeanCompatibility.

-XX:-HeapManagementMXBeanCompatibility
You can choose to opt in to using this policy. The policy enables the default heap changes in Java 8 SR5 and above; however in some cases, the garbage collection statistics might indicate the heap usage to be greater than the maximum heap size. For more information, see -XX:[+|-]HeapManagementMXBeanCompatibility.

You can change the garbage collection policy by updating the JVM profile. For details of all the garbage collection options, see .

Improving JVM server performance

To improve the performance of applications that run in a JVM server, you can tune different parts of the environment, including the garbage collection and the size of the heap.

About this task

CICS provides statistics reports on the JVM server, which include details of how long tasks wait for threads, heap sizes, frequency of garbage collection, and processor usage. You can also use additional IBM tools that monitor and analyze the JVM directly to tune JVM servers and help with problem diagnosis. You can use the statistics to check that the JVM is performing efficiently, particularly that the heap sizes are appropriate and garbage collection is optimized.

Procedure

1. Check the amount of processor time that is used by the JVM server.

   Dispatcher statistics can tell you how much processor time the T8 TCBs are using. JVM server statistics tell you how long the JVM is spending in garbage collection and how many garbage collections occurred. Application response times and processor usage can be adversely affected by the JVM garbage collection.
2. Ensure that there is enough available storage capacity in the CICS address space. The CICS address space contains the Language Environment heap size that is required by the JVM server.
3. Tune the garbage collection and heap in the JVM.

   A small heap can lead to very frequent garbage collections, but too large a heap can lead to inefficient use of MVS storage. You can use IBM Health Center to visualize and tune garbage collection and adjust the heap accordingly.

What to do next

For more detailed analysis of memory usage and heap sizes, you can use the Memory Analyzer tool in IBM Support Assistant to analyze Java heap memory using system dump or heap dump snapshots of a Java process.

To start one or more JVM servers in a CICS region, you must ensure that enough storage capacity is available for the JVM to use, excluding any storage capacity that is allocated to CICS.

Examining processor usage by JVM servers

You can use the CICS monitoring facility to monitor the processor time that is used by transactions running in a JVM server. All threads in a JVM server run on T8 TCBs.

About this task

You can use the DFH$MOLS utility to print the SMF records or use a tool such as CICS Performance Analyzer to analyze the SMF records.

Procedure

1. Turn on monitoring in the CICS region to collect the performance class of monitoring data.
2. Check the performance data group DFHTASK.

In particular, you can look at the following fields:

Field ID	Field name	Description
283	MAXTTDLY	The elapsed time for which the user task waited to obtain a T8 TCB, because the CICS region reached the limit of available threads. The thread limit is 2000 for each CICS region and each JVM server can have up to 256 threads.
400®	T8CPUT	The processor time during which the user task was dispatched by the CICS dispatcher domain on a CICS T8 mode TCB. When a thread is allocated a T8 TCB, that same TCB remains associated with the thread until the processing completes.
401	JVMTHDWT	The elapsed time that the user task waited to obtain a JVM server thread because the CICS system had reached the thread limit for a JVM server in the CICS region. This does not apply to Liberty JVM servers.

3. To improve processor usage, reduce or eliminate the use of tracing where possible.
   a) In a production environment, consider running your CICS region with the CICS main system trace flag set off.

      Having this flag on significantly increases the processor cost of running a Java program. You can set the flag off by initializing CICS with SYSTR=OFF, or by using the CETR transaction.

   b) Ensure that you activate JVM trace only for special transactions.

      JVM tracing can produce large amounts of output in a very short time, and increases the processor cost. For more information about controlling JVM tracing, see Diagnostics for Java.

4. Do not use the USEROUTPUTCLASS option in JVM profiles in a production environment.

   Specifying this option has a negative effect on the performance of JVMs. The USEROUTPUTCLASS option enables developers using the same CICS region to separate JVM output, and direct it to a suitable destination, but it involves the building and invocation of additional class instances.

Calculating storage requirements for JVM servers

To start one or more JVM servers in a CICS region, you must ensure that there is enough free storage available for each JVM to use. CICS and other products that are running in the same region might require a considerable amount of z/OS storage. CICS allocations such as DSALIM and EDSALIM affect storage availability and might be over-allocated compared to the peak requirements.

About this task

The storage that is required for a JVM server does not come from CICS DSA storage. Some is managed by the Language Environment handling requests such as **`malloc()`** issued by C code, and some are managed directly by the JVM that uses z/OS storage management requests such as **`IARV64`**. The Language Environment uses z/OS storage services. The release of the JVM in use has a bearing on whether Language Environment or z/OS manage certain types of storage. For example, the storage for the Java Heap might be managed by Language Environment in one JVM release but managed by z/OS in another.

There are a few user configuration options that directly specify the amount of storage for a JVM. For example, the Java command line option **`-Xmx3G`** requests 3 GB of 64-bit storage for the Java Heap.

The amount of storage that is used outside of the Java Heap is a function of what the JVM server is asked to handle in terms of the workload and Java classes that are run. This might be a considerable amount. A JVM server might be configured to handle anything from a relatively simple to a complex workload. An example of a complex workload is when a CICS Liberty JVM server is started. Even a simple workload might require many internal JVM processes that can require a considerable amount of storage. The dynamic nature of this type of storage makes an accurate estimate impossible.

JVM server dynamic storage requirements

The first JVM that starts causes the amount of storage to be defined for the **USS SHRLIBRGNSIZE** parameter to be allocated within 31-bit ELSQA storage. This reduces the amount of available 31-bit user region storage. The **D OMVS,L** command output shows the value of **SHRLIBRGNSIZE**.

4K of 24-bit storage is required for each JVM thread, and a single JVM server might legitimately start many more than 100 threads even before you consider the number of CICS managed JVM server threads defined by THREADLIMIT. In addition, UNIX System Services require 256 K of contiguous 24-bit storage during the process of creating a new thread.

A single JVM server might use several 100 MB of 31-bit user region storage, for example, the JIT compiler alone might use up to 128 MB in z/OS subpool 1 or 2. On top of the JIT usage, you must add other dynamic requirements of the JVM and all products in the region that interact with the JVM. The Java command line option **-Xcompressedrefs** might be responsible for much of the 31-bit storage usage. From Java 7.1, **-Xcompressedrefs** is the default, so **-Xnocompressedrefs** might need to be specified to reduce the 31-bit storage usage to avoid 878-10 abends.

The amount of 64-bit z/OS storage that can be used by a JVM server in addition to the Java Heap might range between 100 MB - 1 GB or higher. In addition to this, up to 20 MB per thread is allocated. The amount is calculated as the DFHAXRO STACK64 maximum stack size + 4 MB. For example, if STACK64(1M,1M,16M) is used, the value is 16 MB + 4 MB, hence 20 MB is allocated. All of this counts towards the CICS view of **MEMLIMIT** when GDSA expansion is required. But for the z/OS MEMLIMIT check, a value between 3 MB and the allocated size is counted depending on the peak requirement for the Usable area of the stack.

The biggest challenge to running a JVM server is the amount of 31-bit, and possibly 24-bit, storage that is required as that is always limited to some extent. 64-bit usage is limited by **MEMLIMIT**, and can be set to an artificially high value initially and reduced if required.

It might be possible to significantly reduce the amount of storage that is used by tuning Language Environment and Java options.

It is possible to see Language Environment owned 31-bit z/OS storage subpool 1 usage growing over time, and while it might look like a Storage Leak, that is not necessarily the case. In most instances, both the growth and the total amount of subpool 1 storage can be corrected by tuning the Language Environment **HEAP64** runtime option 31-bit parameters, or by setting **-Xnocompressedrefs**, or both. Similarly, growth over time in 64-bit storage might have the same underlying cause and can be corrected by tuning the Language Environment **HEAP64** 64-bit parameters.

The following procedure aims to provide a simple estimate for the initial sizing, and room for expansion of both 31-bit and 64-bit storage must be possible to mitigate against an estimate that is too small. DFH0STAT reports show the actual storage usage when a JVM server is active.

Note: In terms of the number of threads, assume a background value of 100 when a CICS Liberty JVM server is used, or 50 otherwise, and to that add the THREADLIMIT of the JVM server.

Procedure

1. Run the sample statistics program DFH0STAT before starting a JVM server.
   a) The 24-bit requirement is 256 K plus the number of threads * 4 K.
   b) Start with the 31-bit requirement for **SHRLIBRGNSIZE** shown by the output of `D OMVS,L`.
   c) The remainder of the requirements cannot be accurately estimated. You can add a best guess of 50 - 300 MB per JVM server to the size of the 31-bit requirement depending on whether it is expected to run a simple or complex workload.

      The sums can be compared to the `Private Area storage available below 16 Mb` and `Private Area storage available above 16 Mb` from DFH0STAT to ensure that the allocations fit.

- If you want to optimize the 31-bit storage settings to match the allocated storage more closely to the actual usage, you can adjust the **SHRLIBRGNSIZE** parameter and the Language Environment options. See "Tuning the z/OS shared library region" on page 246 and "Modifying the enclave of a JVM server with DFHAXRO" on page 245.

2. Calculate how much 64-bit storage is needed for each additional JVM server by adding up the following storage requirements:
   - The **-Xmx** value.
   - The value from **-Xcmsx**, if used.
   - The estimated total number of threads * 20 MB. (This is to account for the way that CICS GDSA expansion views **MEMLIMIT**.)
   - The remainder of the 64-bit requirements cannot be estimated. You can add a best guess of 256 MB - 1 GB.
   - Round up to the next GB multiple.
3. Check the **MEMLIMIT** value to determine whether you have enough 64-bit storage available to run the JVM server. The z/OS **MEMLIMIT** parameter limits the amount of 64-bit storage for the CICS region.

   If you want to optimize the 64-bit storage usage, see "Modifying the enclave of a JVM server with DFHAXRO" on page 245.

Tuning JVM server heap and garbage collection

Garbage collection in a JVM server is handled by the JVM automatically. You can tune the garbage collection process and heap size to ensure that application response times and processor usage are optimal.

About this task

The garbage collection process affects application response times and processor usage. Garbage collection temporarily stops all work in the JVM and can therefore affect application response times. If you set a small heap size, you can save on memory, but it can lead to more frequent garbage collections and more processor time spent in garbage collection. If you set a heap size that is too large, the JVM makes inefficient use of MVS storage and this can potentially lead to data cache misses and even paging. CICS provides statistics that you can use to analyze the JVM server. You can also use IBM Health Center, which provides the advantage of analyzing the data for you and recommending tuning options.

Procedure

1. Collect JVM server and dispatcher statistics over an appropriate interval. The JVM server statistics can tell you how many major and minor garbage collections take place and the amount of time that elapsed performing garbage collection. The dispatcher statistics can tell you about processor usage for T8 TCBs across the CICS region.
2. Use the dispatcher TCB mode statistics for T8 TCBs to find out how much processor time is spent on JVM server threads.

   The "Accum CPU Time / TCB" field shows the accumulated processor time taken for all the TCBs that are, or have been, attached in this TCB mode. The "TCB attaches" field shows the number of T8 TCBs that have been used in the statistics interval. Use these numbers to work out approximately how much processor time each T8 TCB has used.
3. Use the JVM server statistics to find the percentage of time that is spent in garbage collection.

   Divide the time of the statistics interval by how much elapsed time is spent in garbage collection. Aim for less than 2% of processor usage in garbage collection. If the percentage is higher, you can increase the size of the heap so that garbage collection occurs less frequently.
4. Divide the heap freed value by the number of transactions that have run in the interval to find out how much garbage per transaction is being collected.

   You can find out how many transactions have run by looking at the dispatcher statistics for T8 TCBs. Each thread in a JVM server uses a T8 TCB.
5. Optional: Write the verbosegc log data to a file, which can be done with the parameter **-Xverbosegclog:***path_to_file*. This data can be analyzed by another ISA tool - Garbage Collection and Memory Visualizer.

The JVM writes garbage collection messages in XML to the file that is specified in the STDERR option in the JVM profile. For examples and explanations of the messages, see IBM SDK for z/OS, Java Technology Edition, Version 7, Troubleshooting and support section.

Tip: You can use the file in the Memory Analyzer tool to perform more detailed analysis.

Results

The outcome of your tuning can vary depending on your Java workload, the maintenance level of CICS and of the IBM SDK for z/OS, and other factors. For more detailed information about the storage and garbage collection settings and the tuning possibilities for JVMs, see IBM SDK for z/OS, Java Technology Edition, Version 7, Troubleshooting and support section.

IBM Health Center and Memory Analyzer are two IBM monitoring and diagnostic tools for Java that are supplied by the IBM Support Assistant workbench. You can download these tools free of charge from the site.

Tuning the JVM server startup environment

If you are running multiple JVM servers, you can improve performance by tuning the JVM startup environment.

About this task

When a JVM server starts, the server has to load a set of libraries in the `/usr/lpp/cicsts/cicsts54/lib` directory. If you start a large number of JVM servers at the same time, the time taken to load the required libraries might cause some JVM servers to time out, or some JVM servers might take an excessively long time to start. To reduce JVM server startup time, you should tune the JVM startup environment.

Procedure

1. Create a shared class cache for the JVM servers to load the libraries a single time.

   To use a shared class cache, add the **`-Xshareclasses`** option to the JVM profile of each JVM server. For more information see Class data sharing between JVMs in IBM SDK, Java Technology Edition 7.0.0.
2. Increase the timeout value for the OSGi framework.

   The DFHOSGI.jvmprofile contains the `OSGI_FRAMEWORK_TIMEOUT` option that specifies how long CICS waits for the JVM server to start and shut down. If the value is exceeded, the JVM server fails to initialize or shut down correctly. The default value is 60 seconds, so you should increase this value for your own environment.

Language Environment enclave storage for JVMs

A JVM server has both static and dynamic storage requirements, primarily in 64-bit storage. It may use a significant amount of 31-bit storage.

Note: The amount of 31-bit storage used will depend on several factors:

- The configuration parameters
- The design and use of other products
- The design of the JVM
- The Java workload.

For example, the use of **`-Xcompressedrefs`** might improve performance, but requires 31-bit storage and should always be used with **`-XXnosuballoc32bitmem`** to ensure that the JVM dynamically allocates 31-bit storage for compressed references based on demand. For more information about of these options, see Default settings for the JVM in IBM SDK, Java Technology Edition 7.0.0. *Just-in-time compilation* (JIT) also requires 31-bit storage for the compiled class code.

A JVM runs as a z/OS UNIX System Services process in a Language Environment enclave that is created using the Language Environment preinitialization module, **CELQPIPI**.

JVM storage requests are handled by Language Environment, which in turn allocates z/OS storage based on the defined runtime options.

The Language Environment runtime options are set by DFHAXRO. The default values provided by these programs for a JVM enclave are shown in Table 37 on page 241:

Table 37. Language Environment runtime options used by CICS for the JVM enclave

Language Environment runtime options	Example JVM server values
Heap storage	`HEAP64(256M,4M,KEEP,4M,1M,FREE,1K,1K,KEEP)`
Library heap storage	`LIBHEAP64(5M,3M)`
Library routine stack frames that can reside anywhere in storage	`STACK64(1M,1M,16M)`
Optional heap storage management for multithreaded applications (64 bit)	`HEAPPOOLS64(ALIGN)`
Optional heap storage management for multithreaded applications (31 bit)	`HEAPPOOLS(ALIGN)`
Amount of storage reserved for the out-of-storage condition and the initial content of storage when allocated and freed	`STORAGE(NONE,NONE,NONE)`

Note: For current JVM server values, refer to the DFHAXRO member in Library SDFHSAMP.

Language Environment runtime options, such as HEAP64, work on the principle of an initial value for that type of storage: for example, 256 MB 64-bit. When HEAP64 cannot contain a new request, an increment is allocated of the specified size (4 MB above) or of the request size plus control information, whichever is larger. Extra increments are allocated as required to meet demand. When an increment is empty, Language Environment will either KEEP or FREE the z/OS storage based on the runtime value.

For full information about Language Environment runtime options, see z/OS Language Environment Customization.

Where possible, the 31-bit and 64-bit initial size should cover the total 31-bit and 64-bit storage requirements, although a few increments is acceptable. This reduces both overall z/OS storage requirements and CPU time, compared to when there are many increments.

The HEAP64 31-bit increment size should not be set to less than 1M and the FREE option should be used. In the previous example, the 31-bit parameters were set to 4M, 1M, and FREE.

Language Environment 31-bit and 64-bit HEAP usage can be seen by activating the RPTO(ON) and RPTS(ON) options in DFHAXRO. An Language Environment storage report is produced when the JVM server is stopped.

You can override the Language Environment runtime options by modifying and recompiling the sample program DFHAXRO, which is described in “Modifying the enclave of a JVM server with DFHAXRO” on page 245. This program is set on the JVMSERVER resource, so you can use different names, which is why there are different options for individual JVM servers, if required.

The amounts of storage required for a JVM in a Language Environment enclave might require changes to installation exits, IEALIMIT or IEFUSI, which you use to limit the **REGION** and **MEMLIMIT** sizes. A possible approach is to have a Java owning region (JOR), to which all Java program requests are routed. Such a region runs only Java workloads, minimizing the amount of CICS DSA storage required and allowing the maximum amount of MVS storage to be allocated to JVMs.

Identifying Language Environment storage needs for JVM servers

After identifying the actual storage needs, it is possible to determine whether the supplied **DFHAXRO** options need to be modified or not. This allows values to be chosen that either avoid the need for incremental storage allocations, or reduce the number to an acceptable level.

About this task

The HEAP64 runtime option in DFHAXRO controls the heap size of the Language Environment enclave for a JVM server. This option includes settings for 64-bit, 31-bit, and 24-bit storage. You can use your own program instead of DFHAXRO if preferred. The program must be specified on the JVMSERVER resource.

Procedure

1. Set the `RPTO(ON)` and `RPTS(ON)` options in DFHAXRO.

   These options are in comments in the supplied source of DFHAXRO. Specifying these options causes Language Environment to report on the storage options and to write a storage report showing the actual storage used.
2. Disable the JVMSERVER resource.

   The JVM server shuts down and the Language Environment enclave is removed.
3. Enable the JVMSERVER resource.

   CICS uses the Language Environment runtime options in DFHAXRO to create the enclave for the JVM server. The JVM also starts up.
4. Run your Java workloads in the JVM server to collect data about the storage that is used by the Language Environment enclave.
5. Remove the `RPTO(ON)` and `RPTS(ON)` options from DFHAXRO.
6. Disable the JVMSERVER resource to generate the storage reports.

   The storage reports include a suggestion for the initial Language Environment enclave heap storage. The entry “Suggested initial size” in the 64-bit user heap statistics contains the suggested value and is equal to the total amount of Language Environment enclave heap storage that was used by the JVM server.

Results

The storage reports are saved in an `stderr` file in z/OS UNIX, or can also go to your CICS JES output if you are using the **JOBLOG** or **DD://** routing syntax. The directory depends on whether you have redirected output for the JVM in the JVM profile. If no redirection exists, the file is saved in the working directory for the JVM. If no value is set for WORK_DIR in the profile, the file is saved in the `/tmp` directory.

Use the information in the storage reports to select a suitable value for the Language Environment enclave heap storage in the **DFHAXRO HEAP64** option. Storage requirements might change from one CICS execution to the next, and are typically not the same for different CICS systems that share the one DFHAXRO, thus requiring a compromise.

The normal aim is to set the HEAP64 initial allocations to the suggested sizes to avoid or reduce the number of increments. The more increments that are used, the more likely that the ratio of z/OS storage compared to actively used Language Environment storage increases. Many increments can also cause an increase in the amount of CPU time that is used by Language Environment to manage the HEAP64 storage requests. Java 7.0 allocates the Java Heap in Language Environment 64-bit HEAP storage and this might result in a large initial allocation being suggested. In this case, you should subtract the **-Xmx** value from the suggested size and use the remainder, which should be less than **-Xmx**, as the initial allocation. This forces Language Environment to allocate the Java Heap in its own Memory Object as a Language Environment HEAP increment. Later Java releases allocate the JVM Heap as a Memory Object via IARV64 and not through a Language Environment storage request. If a Java migration is performed with an initial allocation that includes **-Xmx**, it normally doubles the storage that is used for the Java Heap, and might result in MEMLIMIT being too small.

Allocating many increments might produce the effect of a Storage Leak, which manifests as a continual increase in z/OS storage over time. In practice, this is more likely to be Storage Creep, which is characterized by an increase in both z/OS allocated storage and Language Environment free storage. A Storage Leak shows a continual increase in both z/OS and Language Environment used storage. 31-bit HEAP64 storage is allocated in z/OS subpool 1. In Java 7.0, JIT storage is allocated in z/OS subpool 1 and will naturally grow in size over time in increments of slightly over 2MB in size. In later Java releases, JIT storage is allocated in z/OS subpool 2 in 2MB increments.

The effect of the revised options should be evaluated at least one time and adjusted as required. Tuning should also be repeated at suitable intervals to assess the effect of any changes to storage usage due to application changes and other changes. Tuning should also be repeated whenever the CICS or Java release changes as storage usage patterns might change.

Note: If you increase the 31 bit **HEAP64** initial size, you must also change **HEAPP** to avoid over-allocating **HEAPPOOLS** 31-bit storage. In the example below, the HEAPPOOLS percentage values should be reduced from 10% to 1%.

HEAPPOOLS and HEAPPOOLS64 are active in the default DFHAXRO and can be effective when configured, but the correct values are dependent on the workload and hence precise tuning might be difficult.

STACK64 should be checked to ensure that the maximum storage used is not close to the defined limit, which is typically 16 MB. Exceeding the limit will results in runtime errors.

What is not obvious from LE RPTSTG output is that, while using STACK64(1M,1M,16M) provides a safe value for JVM thread stack expansion, it can result in a large MEMLIMIT being required to avoid CICS SOS Above the Bar during GDSA expansion. With the 16M maximum, 20 MB is allocated per JVM thread in three Memory Objects - one of 16+1 MB, one of 2 MB and one of 1 MB. Only 3 MB is normally usable (leaving 17 MB as hidden storage) and this is counted towards the z/OS IARV64 MEMLIMIT check. However, CICS counts all 20 MB to decide whether it can expand the GDSA by a multiple of GB without exceeding MEMLIMIT. A single JVM server can legitimately use more than 200 threads, and 200 threads equates to approximately 4 GB towards the CICS MEMLIMIT check. This is not a bug. Therefore, reducing the STACK64 maximum to a lower value that still permits some expansion can help towards reducing the MEMLIMIT size and the possibility of SOS Above the Bar. It is recommended that a smaller STACK64 maximum be validated in a suitable load test environment before being used in production.

Example

The following example is **RPTOPTS** output based on these DFHAXRO options:

```
HEAPPOOLS(ALIGN,8,10,32,10,128,10,256,10,1024,10,2048,10,0,10,0,10,0,10,0,10,0,10,0,10)
HEAPPOOLS64(ALIGN,8,4000,32,2000,128,700,256,350,1024,100,2048,50,3072,50,4096,50,8192,
            25,16384,10,32768,5,65536,5)
HEAP64(256M,4M,KEEP,4194304,1048576,KEEP,1024,1024,KEEP)
LIBHEAP64(3M,3M,FREE,16384,8192,FREE,8192,4096,FREE)
STACK64(1M,1M,16M)
THREADSTACK64(OFF,1M,1M,128M)
```

The following example is partial **RPTSTG** output:

```
STACK64 statistics:
Initial size:                                 1M
Increment size:                               1M
Maximum used by all concurrent threads:       1M
Largest used by any thread:                   1M - no change required
Number of increments allocated:               0
THREADSTACK64 statistics:
Initial size:                                 1M
Increment size:                               1M
Maximum used by all concurrent threads:       0M
Largest used by any thread:                   0M - not used
Number of increments allocated:               0
64bit User HEAP statistics:
Initial size:                                 256M
Increment size:                               4M
Total heap storage used:                      730857472
Suggested initial size:                       697M - use this
Successful Get Heap requests:                 783546
Successful Free Heap requests:                780785
Number of segments allocated:                 135 - too many increments
```

```
Number of segments freed:                        0
31bit User HEAP statistics:
Initial size:                                    4194304
Increment size:                                  1048576
Totalheap storage used (suggested initial size):  137165672 - use this
Successful Get Heap requests:                    1345332
Successful Free Heap requests:                   1345260
Number of segments allocated:                    125 - too many increments
Number of segments freed:                        0
64bit Library HEAP statistics:
Initial size:                                    3M
Increment size:                                  3M
Total heap storage used:                         4640032
Suggested initial size:                          5M
Successful Get Heap requests:                    113381
Successful Free Heap requests:                   112860
Number of segments allocated:                    1 - low, so no change required
Number of segments freed:                        0
31bit Library HEAP statistics:
Initial size:                                    16384
Increment size:                                  8192
Total heap storage used (suggested initial size): 520
Successful Get Heap requests:                    33725
Successful Free Heap requests:                   33725
Number of segments allocated:                    1 - low, so no change required
Number of segments freed:                        0

Suggested Percentages for current CellSizes:
HEAPP(ALIGN,8,1,32,1,128,1,256,1,1024,1,2048,1,0)
```

When reviewing **RPTSTG** output, remember that the **HEAP64** increment sizes are for the minimum amount of storage that Language Environment allocates, and any increment could be substantially bigger than that value. Hence it is not possible to accurately determine how much z/OS storage was used when 1 or more increments have been allocated. The actual number of increments is reported for 64bit HEAP (that is, 135), for 31bit **HEAP** the actual number of increments is one less than is shown (that is, 124 not 125).

Because of the way that Language Environment's storage management works when increments are used, the amount of 31-bit and 64-bit z/OS storage allocated may be significantly higher than shown in **RPTSTG** "maximum used".

The suggested **DFHAXRO** changes are:

```
* Heap storage
DC     C'HEAP64(700M,'         Initial 64bit heap - change (Note 1)
DC     C'4M,'                   64bit Heap increment
DC     C'KEEP,'                 64bit Increments kept
DC     C'128M,'                 Initial 31bit heap - change (Note 2)
DC     C'2M,'                   31bit Heap increment - change (Note 3)
DC     C'FREE,'                 31bit Increments freed - change (Note 4)
DC     C'1K,'                   Initial 24bit heap
DC     C'1K,'                   24bit Heap increment
DC     C'KEEP) '                24bit Increments kept

* Heap  pools
DC     C'HP64(ALIGN) '
DC     C'HEAPP(ALIGN,8,1,32,1,128,1,256,1,1024,1,2048,1,0) ' - change (Note 5)

* Library Heap storage
DC     C'LIBHEAP64(3M,3M) '     Initial 64bit heap - do not change (Note 6)

* 64bit stack storage
DC     C'STACK64(1M,1M,16M) ' - consider a change (Note 7)
```

Note:

1. As shown by **RPTSTG** output 64bit "Suggested initial size" plus a small increase.
2. As shown by **RPTSTG** output 31bit "Suggested initial size" but with a small reduction as we are using FREE.
3. The 31-bit **HEAP** increment may be better as a value of 2M instead of 1M.
4. Optionally, using 31-bit **HEAP FREE** may result in less z/OS storage being allocated to map the "Total heap storage used" than with KEEP.
5. As recommended by **RPTSTG** output after the **HEAPPOOLS** statistics, but may benefit from further optimization. The default of 10% of the 31-bit Heap initial size of 128MB is likely to result in an

excessive amount of storage being allocated. A minimum of 6 pools each of 10% of the initial heap size of 128MB causes 77MB to be allocated. This will be included in the "Total heap storage used" value (because the **HEAPPOOLS** storage extents are allocated there), irrespective of what percentage of the pool s is productively used. Using **HEAPPOOLS** cell sizes greater than 256 bytes might result in inefficient use of Language Environment HEAP storage.

6. Only one increment was required, which is not a problem.
7. The largest used was 1MB. Reducing the maximum of 16M to a value such as 8 MB or even lower would significantly reduce the amount of STACK64 storage that CICS counts towards **MEMLIMIT** when checking to see whether it can allocate a new GDSA extent. STACK64 changes should be tested thoroughly before migrating them into a production environment.

This is an example of using 31-bit **HEAP FREE** on another run of the same JVM server. The "Number of segments" shows the number of **GETMAIN**s and **FREEMAIN**s performed, which was low for the time that the JVM server was active. The difference of 2 shows that the enclave terminated with only the initial allocation plus one increment, which is likely to be less than the "Total heap storage" and shows the effectiveness of FREE. "Total heap storage used" was higher, but any total often changes from one run of a JVM server to another, hence basing changes on only one set of **RPTSTG** may not provide the best possible settings.

```
31bit User HEAP statistics:
Initial size: 134217728
Increment size: 2097152
Total heap storage used (suggested initial size): 154056664
Successful Get Heap requests: 3253239
Successful Free Heap requests: 3253176
Number of segments allocated: 149
Number of segments freed: 147
```

It is important to read the Language Environment Debugging Guide in order to correctly interpret **RPTSTG** output.

Modifying the enclave of a JVM server with DFHAXRO

DFHAXRO is a sample program that provides a default set of runtime options for the Language Environment enclave in which a JVM server runs. For example, it defines storage allocation parameters for the JVM heap and stack. It is not possible to provide default runtime options that are optimized for all workloads. Consider identifying actual storage usage, and override the defaults as required, to optimize the ratio of used storage to allocated storage.

About this task

You can update the sample program to tune the Language Environment enclave or you can base your own program on the sample. The program is defined on the JVMSERVER resource and is called during the CELQPIPI preinitialization phase of the Language Environment enclave that is created for a JVM server.

You must write the program in assembly language and it must not be translated with the CICS translator. The options are specified as character strings, comprising a 2-byte string length followed by the runtime option. The maximum length for all Language Environment runtime options is 255 bytes, so use the abbreviated version of each option and restrict your changes to a total of under 200 bytes.

Procedure

1. Copy the DFHAXRO program to a new location to edit the runtime options.

   If maintenance is applied to your CICS region, you might want to reflect the changes in your program. The source for DFHAXRO is in the CICSTS54.CICS.SDFHSAMP library.
2. Edit the runtime options, using the abbreviation for each option.

   The z/OS Language Environment Programming Guide has complete information about Language Environment runtime options.

   - Keep the size of the list of options to a minimum for quick processing and because CICS adds some options to this list.

- Use the HEAP64 option to specify the initial heap allocation.
- The ALL31 option, the POSIX option, and the XPLINK option are forced on by CICS. The ABTERMENC option is set to (ABEND) and the TRAP option is set to (ON,NOSPIE) by CICS.
- The output that is produced by the RPTO and RPTS options is written to the CESE transient data queue.
- Any options that produce output do so at each JVM termination. Consider the volume of output that might be produced and directed to CESE.

3. Use the DFHASMVS procedure to compile the program.

Results

When you enable the JVMSERVER resource, CICS creates the Language Environment enclave by using the runtime options that you specified in the DFHAXRO program. CICS checks the length of the runtime options before it passes them to Language Environment. If the length is greater than 255 bytes, CICS does not attempt to start the JVM server and writes error messages to CSMT. The values that you specify are not checked by CICS before they are passed to Language Environment.

Tuning the z/OS shared library region

The shared library region is a z/OS feature that enables address spaces to share dynamic link library (DLL) files. This feature enables your CICS regions to share the DLLs that are needed for JVMs, rather than each region having to load them individually. This can greatly reduce the amount of real storage used by MVS, and the time it takes for the regions to load the files.

The storage that is reserved for the shared library region is allocated in each CICS region when the first JVM is started in the region. The amount of storage that is allocated is controlled by the **SHRLIBRGNSIZE** parameter in z/OS, which is in the BPXPRMxx member of SYS1.PARMLIB. The minimum is 16 MB, and the z/OS default is 64 MB. You can tune the amount of storage that is allocated for the shared library region by investigating how much space you need, bearing in mind that other applications besides CICS might be using the shared library region, and adjusting the **SHRLIBRGNSIZE** parameter accordingly.

If you want to reduce the amount of storage that is allocated for the shared library region, first check that you do not have wasted space in your shared library region. Bring up your normal workload on the z/OS system, then issue the command **D OMVS,L** to display the library statistics. If there is unused space in the shared library region, you can reduce the setting for **SHRLIBRGNSIZE** to remove this space. If CICS is the only user of the shared library region, you can reduce the **SHRLIBRGNSIZE** to the minimum of 16 MB, because the DLLs needed for the JVM only use around 10 MB of the region.

If you find that all the space in the shared library region is being used, but you still want to reduce this storage allocation in your CICS regions, there are three possible courses of action that you can consider:

1. It is possible to set the shared library region size smaller than the amount of storage that you need for the files. When the shared library region is full, files are loaded into private storage instead, and do not benefit from the sharing facility. If you choose this course of action, you should make sure that you restart your more important applications first, to ensure that they are able to make use of the shared library region. This course of action is most appropriate if most of the space in the shared library region is being used by non-critical applications.
2. The DLLs that are placed in the shared library region are those marked with the extended attribute +l. You can remove this attribute from some of your files to prevent them going into the shared library region, and so reduce the amount of storage that you need for the shared library region. If you choose this course of action, select files that are less frequently shared, and also try not to select files that have the extension .so. Files with the extension .so, if they are not placed in the shared library region, are shared by means of user shared libraries, and this sharing facility is less efficient than using the shared library region. This course of action is most appropriate if large files that do not have the extension .so are using most of the space in the shared library region.
3. If you remove the extended attribute +l from all the files relating to the CICS JVM, then your CICS regions do not use the shared library region at all, and no storage is allocated for it within the CICS regions. If you choose this course of action, you do not benefit from the shared library region's sharing

facility. This course of action is most appropriate if other applications on the z/OS system require a large shared library region, and you do not want to allocate this amount of storage in your CICS regions.

If you choose to remove the extended attribute +l from any of your files, when you replace those files with new versions (for example, during a software upgrade), remember to check that the new versions of the files do not have this attribute.

Chapter 8. Troubleshooting Java applications

If you have a problem with a Java application, you can use the diagnostics that are provided by CICS and the JVM to determine the cause of the problem.

About this task

CICS provides some statistics, messages, and tracing to help you diagnose problems that are related to Java. The diagnostic tools and interfaces that are provided with Java can give you more detailed information about what is happening in the JVM than CICS because CICS is unaware of many of the activities in a JVM.

You can use freely available tools that perform real-time and offline analysis of a JVM, for example IBM Health Center. For full details, see IBM Monitoring and Diagnostic Tools for Java - Health Center.

For troubleshooting web applications that are running in a Liberty JVM server, see "Troubleshooting Liberty JVM servers and Java web applications" on page 253. For information about where to find log files, see "Controlling the location for JVM output, logs, dumps and trace" on page 259.

Procedure

1. If you are unable to start a JVM server, check that the setup of your Java installation is correct.

   Use the CICS messages and any errors in the `stderr` file for the JVM to determine what might be causing the problem.

   a) Check that the correct version of the Java SDK is installed and that CICS has access to it in z/OS UNIX.

      For a list of supported SDKs, see High-level language and compiler support .

   b) Check that the **USSHOME** system initialization parameter is set in the CICS region.

      This parameter specifies the home for files on z/OS UNIX.

   c) Check that the **JVMPROFILEDIR** system initialization parameter is set correctly in the CICS region.

      This parameter specifies the location of the JVM profiles on z/OS UNIX.

   d) Check that the CICS region has read and run access to the z/OS UNIX directories that contain the JVM profiles.

   e) Check that the CICS region has write access to the working directory of the JVM.

      This directory is specified in the `WORK_DIR` option in the JVM profile.

   f) Check that the `JAVA_HOME` option in the JVM profiles points to the directory that contains the Java SDK.

   g) Check that SDFJAUTH is in the STEPLIB concatenation of the CICS startup JCL.

   h) If you are using WebSphere MQ or DB2 DLL files, check that the 64-bit versions of these files are available to CICS.

   i) If you modify DFHAXRO to configure the Language Environment enclave, ensure that the runtime options do not exceed 200 bytes and that the options are valid.

      CICS does not validate the options that you specify before it passes them to Language Environment. Check SYSOUT for any error messages from Language Environment.

2. If your setup is correct, gather diagnostic information to determine what is happening to the application and the JVM.

   a) To obtain the diagnostics, you must use `PRINT_JVM_OPTIONS=TRUE`. The default for this option is `PRINT_JVM_OPTIONS=FALSE`, so if it is left to default no options for diagnostics are presented. When you specify `PRINT_JVM_OPTIONS=TRUE`, all the options that are passed to the JVM at startup, including the contents of the class paths, are printed to SYSPRINT. The information is produced every time a JVM is started with this option in its profile.

b) Check the `dfhjvmout` and `dfhjvmerr` files for information and error messages from the JVM.

These files are in the directory that is specified by the `WORK_DIR/applid/jvmserver` option in the JVM profile. The files might have different names if the STDOUT and STDERR options were changed in the JVM profile.

3. If the application is failing or performing poorly, debug the application.

- If you receive `java.lang.ClassNotFoundException` errors and the transaction abends with the AJ05 code, the application might not be able to access IBM or vendor classes in the OSGi framework. For more information about how to fix this problem, see Upgrading the Java environment.
- Use the CEDX transaction to debug the application transaction. For a Liberty JVM server, if you are using a URI map to match the inbound application request to an application transaction, debug that transaction. If you use the default transaction CJSA, you must set the MAXACTIVE attribute to 1 on the DFHEDFTC transaction class (or DFHEDFTO transaction class if you use CEDY). This setting is required because a number of CJSA tasks might be running and you might debug the wrong transaction. Do not use CEDX on the CJSA transaction in a production environment.
- To use a debugger with the JVM server, you must set some options in the JVM profile. For more information, see "Debugging a Java application" on page 264.
- If you want to determine the status of OSGi bundles and services, use the OSGi console. Set the following properties in the JVM profile: `-Dosgi.file.encoding=ISO-8859-1`, and `-Dosgi.console=host:port` where `host` is the host name of the system the JVM server is running on, and `port` is a free port on the same system. While the `osgi.console.encoding` property was designed to allow the OSGi console to use a preferred encoding without putting the whole JVM into that encoding, an outstanding bug in the Equinox OSGi framework prevents its use, instead you must set the `file.encoding` value to an ASCII based encoding. If you are using an OSGi JVM server, add **`OSGI_CONSOLE=TRUE`** to the JVM profile. If you are using a Liberty JVM server, add the osgiConsole-1.0 feature to the `server.xml`. Connect to the OSGi console by using a Telnet session with the host and port properties you specified in the JVM profile.

  Note: If you type the `exit` command into the OSGi console, it will issue a system.exit(0) call to the environment that the JVMSERVER runs in. The command to disconnect your terminal from the OSGi console is `disconnect`. system.exit(0) is an abrupt stop of all threads and workload, and if left to continue processing, can leave the JVM and CICS in an indeterminate state. CICS is designed to perform an immediate shutdown in this eventuality to avoid subsequent complications. For this reason, it is important to control write access to both the JVM profile, and `server.xml`. A Liberty JVM server offers further protection by requiring inclusion of the osgiConsole-1.0 feature before the OSGi console is able to run. The OSGi console is primarily a development and debug aid, and is not expected to run in a production environment.

4. If you are getting out-of-memory errors, it might indicate that the JVM or CICS address space was not allocated enough storage, the application might have a memory leak, or the heap size might be insufficient.

a) Use CICS statistics or a tool such as IBM Health Center to monitor the JVM. If the application has a memory leak, the amount of live data that remains after garbage collection gradually increases over time until the heap is exhausted.

The JVM server statistics report the size of the heap after the last garbage collection and the maximum and peak size of the heap. For more information, see Analyzing Java applications using IBM Health Center.

b) Run the storage reports for Language Environment to find out whether the amount of storage is sufficient.

For more information, see Language Environment enclave storage for JVMs.

5. If you are getting encoding errors when you install or run a Java application, maybe you set up conflicting or an unsupported combination of code pages.

JVMs on z/OS typically use an EBCDIC code page for file encoding; the default for non-Liberty JVM servers is IBM1047 (or cp1047), but the JVM can use other code pages for file encoding if required. CICS requires an EBCDIC code page to handle character data and all JCICS calls must use an EBCDIC

code page. The code page is set in the **LOCALCCSID** system initialization parameter for the CICS region.

a) Check the JVM server logs to see whether any warning messages were issued relating to the value of **LOCALCCSID**.

   If this parameter is set to a non-EBCDIC code page, a code page that is not supported by the JVM, or an EBCDIC code page that is not supported (such as 930), the JVM server uses cp1047.

b) JCICS calls use the code page that is specified in the **LOCALCCSID** system initialization parameter.

   If your application expects a different code page, you get encoding errors. To use a different code page for JCICS, set the **-Dcom.cics.jvmserver.override.ccsid=** parameter in the JVM profile.

c) If you are using the **-Dcom.cics.jvmserver.override.ccsid=** parameter in the JVM profile, ensure that the CCSID is an EBCDIC code page.

   The application must use EBCDIC when it uses JCICS calls.

d) If you are running SOAP processing in an Axis2 JVM server, ensure that the **-Dfile.encoding** JVM property specifies an EBCDIC property.

   If you specify a non-EBCDIC code page, such a UTF-8, the web service request fails and the response contains corrupted data.

6. If you experience startup timeouts or timeouts under workload, there are various parameters that you can tune to help resolve the issue. The following give an indication of values you can tune:

   - Modify your **-Dcom.ibm.cics.jvmserver.threadjoin.timeout** setting to control how long an HTTP request waits to obtain a JVM server thread.
   - Increase the **THREADLIMIT** value on the JVMSERVER resource.
   - If **THREADLIMIT** is already set to the maximum permitted value, then you might be attempting to run more work than a single JVM server can handle. Consider balancing the workload between multiple JVM servers or multiple regions.

     Alternatively, your CICS system might be unresponsive because of other constraints. Follow the standard procedures to diagnose performance problems. See Improving the performance of a CICS system.

What to do next

If you cannot fix the cause of the problem, contact IBM support. Make sure that you provide the required information, as listed in the MustGather for reporting Java problems.

Diagnostics for Java

Many of the usual sources of CICS diagnostic information contain information that applies to Java applications. In addition to the information supplied by CICS, there are a number of interfaces specific to the JVM that you can use for problem determination.

CICS diagnostic tools for Java

CICS has statistics and monitoring data that you can collect on running Java applications. When errors occur, transactions abend and messages are written to the appropriate log. See CICS messages for a list of the abends and messages that apply to the JVM (SJ) domain. Messages related to Java are in the format DFHSJxxxx.

You can also turn on tracing to produce additional diagnostic information. The trace points for the JVM domain are listed in JVM domain trace points.

When the first JVM is started in a CICS region after initialization, CICS issues message DFHSJ0207, showing the version of Java that is being used.

The Java SDK provides diagnostic tools and interfaces that give you more detailed information about what is happening in the JVM. Messages and diagnostic information from the JVM are written to the `stderr`

log file for the JVM. If you encounter a Java problem, always consult this file. For example, if CICS issues a message to indicate that the JVM has abended, the `stderr` log file is the primary source of diagnostic information. "Controlling the location for JVM output, logs, dumps and trace" on page 259 tells you how to control the location of output from the JVM, and how to redirect messages from JVM internals and output from Java applications running in a JVM.

When you develop Java applications for CICS, it is important to consider the requirements for thread safety and transaction isolation in CICS. If a Java application works correctly on its first use, but does not behave correctly on subsequent uses, then the problem is likely to be due to isolation issues.

OSGi diagnostic files

The OSGi framework produces diagnostic files in zFS that you can use to help troubleshoot problems with OSGi bundles and services in a JVM server:

OSGi cache
The OSGi cache is in the *$WORK_DIR*/*applid*/*jvmserver*/configuration/org.eclipse.osgi directory of the JVM server. *$WORK_DIR* is the working directory of the JVM server, *applid* is the CICS APPLID, and *jvmserver* is the name of the JVMSERVER resource. The OSGi cache contains framework metadata and other information that is required to run the framework. The cache is replaced when the JVM server starts up.

OSGi logs
If an error occurs in the OSGi framework, an OSGi log is created in the *$WORK_DIR*/*applid*/*jvmserver*/configuration/ directory of the JVM server. The file extension is `.log`.

JVM diagnostic tools

The CICS documentation provides information about some of the Java diagnostic tools and interfaces:

- "Activating and managing tracing for JVM servers" on page 263 describes how you can use the component tracing provided by the CETR transaction to trace the life cycle of the JVM server and the tasks running inside it. JVM servers do not use auxiliary or GTF tracing. Instead, the tracing is written to a file on zFS that is uniquely named for each JVM server.
- "Debugging a Java application" on page 264 describes how you can use a remote debugger to step through the application code for a Java application that is running in a JVM. CICS also provides a set of interception points (or plug-in) in the CICS Java middleware, which allows additional Java programs to be inserted immediately before and after the application Java code is run, for debugging, logging, or other purposes. For more information, see "The CICS JVM plug-in mechanism" on page 265.

Many more diagnostic tools and interfaces are available for the JVM. See IBM SDK for z/OS, Java Technology Edition, Version 7, Troubleshooting and support section for information about further facilities that can be used for problem determination for JVMs. The following facilities provide useful diagnostic information:

- The internal trace facility of the JVM can be used directly, without going through the interfaces provided by CICS. For information about the system properties that you can use to control the internal trace facility and to output JVM trace information to various destinations, see Using CICS trace. You can use these system properties to output trace from any method or class within the JVM, and to find the value of any parameters and return types on the method call.
- If you experience memory leaks in the JVM, you can request a heap dump from the JVM. A heap dump generates a dump of all the live objects (objects still in use) that are in the heap of the JVM. You can also analyze memory leaks using the IBM Health Center and Memory Analyzer tools, which are both available with IBM Support Assistant. For more information about Java tools, see IBM Monitoring and Diagnostic Tools for Java - Health Center.
- The HPROF profiler, that is shipped with the IBM 64-bit SDK for z/OS, Java Technology Edition, provides performance information for applications that run in the JVM, so you can see which parts of a program are using the most memory or processor time.

- The JVM provides interfaces for monitoring, profiling, and RAS (Reliability, Availability, and Serviceability).

With all interfaces, options, or system properties available for the IBM JVM that are not specific to the CICS environment, use the IBM JVM documentation as the primary source of information.

Troubleshooting Liberty JVM servers and Java web applications

If you have a problem with a Java web application, you can use the diagnostics that are provided by CICS and Liberty to determine the cause of the problem.

CICS provides statistics, messages, and tracing to help you diagnose problems that are related to running Java web applications in a Liberty JVM server. The Liberty technology that is used to run web applications also produces diagnostics that are available in zFS. For general setup errors and application problems, see Troubleshooting and support.

Avoiding problems

CICS uses the values of the region APPLID and the JVMSERVER resource name to create unique zFS file and directory names. Some of the acceptable characters have special meanings in the UNIX System Services shell. For example, the dollar sign ($) means the start of an environment variable name. Some of these characters can cause an `Exception` in the Equinox OSGi framework and prevent the JVM server from starting. Avoid using non-alphanumeric characters in the region APPLID and JVM server name. If you do use these characters, you might need to use the backslash (\) as an escape character in the UNIX System Services shell. For example, if you called your JVM server MY$JVMS and wanted to read the JVM system out file:

```
cat CICSPRD.MY\$JVMS.D20140319.T124122.dfhjvmout
```

Unable to start Liberty JVM server

1. If you are unable to start a Liberty JVM server, check that your setup is correct; see Configuring a Liberty JVM server for more information. Use the messages in the CICS system log and the Liberty `messages.log` file that is located below `WLP_OUTPUT_DIR` to determine what might be causing the problem.
2. Check that the **`-Dfile.encoding`** JVM property in the JVM profile specifies either ISO-8859-1 or UTF-8. These are the two code pages that are supported by Liberty. If you set any other value, the JVM server fails to start.

Unable to authenticate a user when trying to access a protected web application in a CICS Liberty JVM server

CICS **`JESMSGLG`** log contains the message:

```
ICH420I PROGRAM DFHSIP FROM LIBRARY hlq.SDFHAUTH CAUSED THE
ENVIRONMENT TO BECOME UNCONTROLLED
BPXP014I ENVIRONMENT MUST BE CONTROLLED FOR DAEMON (BPX.DAEMON) PROCESSING.
```

The Liberty `messages.log` contains the message:

```
CWWKS1100A: Authentication did not succeed for user ID user.
An invalid user ID or password was specified.
```

The CICS Liberty JVM server security implementation uses the Liberty angel process to perform authorized security checks. If Liberty is unable to connect to the angel process, it will fail over to using USS security which requires all members in the STEPLIB and DFHRPL concatenations to be program controlled.

Attention: Be aware that the Liberty server only connects to the angel process at server startup. The JVM server needs to be restarted to complete authentication.

Unable to authenticate a user with user ID and password, cannot access APPL-ID when trying to access a protected web application in a CICS Liberty JVM server

Liberty `messages.log` contains the message:

```
com.ibm.ws.security.saf.SAFServiceResult E CWWKS2909E:
A SAF authentication or authorization attempt was rejected because the server
is not authorized to access the following SAF resource:
APPL-ID APPL-ID. Internal error code 0x03008108.
```

The CICS Liberty JVM server security requires access to SAF security profiles in classes APPL and SERVER. If access is not granted then Liberty will not be able to authenticate the user ID and password. Details of how to configure this can be found here Authenticating users in a Liberty JVM server

Web application is not available after it is deployed to the `dropins` directory

If you receive a CWWK0221E error message in `dfhjvmerr`, check that you set the right values for the host name and port number in the JVM profile and `server.xml`. The port might be in use by another process and port sharing disabled. The host name might not be resolvable by the client.

CICS CPU use increased after a Liberty JVM server is enabled

Liberty can be configured to regularly check for updates to both configuration and installed applications using the *<config>* and *<applicationMonitor>* elements in `server.xml`. If the configuration polling rate or application monitor interval is set too frequently it can cause excessive use of CPU and I/O.

For *<config>* you can reduce the frequency using the *monitorInterval* attribute. Do not set the *updateTrigger* attribute to disabled because CICS requires Liberty to pick up configuration changes within a few seconds.

For *<applicationMonitor>* you can reduce the frequency using the *pollingRate* attribute, or change *updateTrigger* attribute to "mbean", or disable it.

For more information see Controlling dynamic updates.

Application not available

You copy a WAR file into the `dropins` directory but your application is not available. Check the Liberty `messages.log` file for error messages. If you receive the CWWKZ0013E error message, you already have a web application running in the Liberty JVM server with the same name. To fix this problem, change the name of the web application and deploy to the `dropins` directory.

Web application returns `Context Root Not Found`

You enabled your Liberty JVM server and deployed your Web application, the JVM server reports it is enabled, but when you are accessing your application, you receive `Context Root Not Found`. Accessing the Web application a short time later results in success. This is a known timing window in which the server reports it is enabled while applications are still starting in the background. You are more likely to experience this condition in a multi-region environment that uses Sysplex Distributor or port sharing. You are also likely to experience this condition if you use automation to access the application triggered from the `enabled` status. If you are using Sysplex Distributor or port sharing, TCP/IP automation can be used to silence a port and then resume the port once the web application is available. Workarounds might involve the addition of a pause in automation scripts, or the application writing a flag to a known location when it is available.

Web application is not requesting authentication

You configured security, but the web application is not requesting authentication.

1. Although you can configure CICS security for web applications, the web application uses security only if it includes a security restraint in the WAR file. Check that a security restraint was defined by the application developer in the `web.xml` file in the Dynamic Web Project.

2. Check that the `server.xml` file contains the correct security information. Any configuration errors are reported in `dfhjvmerr` and might provide some useful information. If you are using CICS security, check that the feature **`cicsts:security-1.0`** is specified in `server.xml`. If CICS security is switched off, check that you specified a basic user registry to authenticate application users.
3. Check that `server.xml` is configured either for `<safAuthorization>` to take advantage of EJBRoles, or for a local role mapping in an `<application-bnd>` element. The `<application-bnd>` element is found with in the `<application>` element in `server.xml` or `installedApps.xml`. The default security-role added by CICS for a local role mapping is **`cicsAllAuthenticated`**.

Web application is returning an HTTP 403 error code

The web application is returning an HTTP 403 error code in the web browser because either your user ID is revoked or you are not authorized to run the application transaction.

1. Check the CICS message log for the error message ICH408I to see what type of authorization failure occurred. To fix the problem, make sure that the user ID has a valid password and is authorized to run the transaction.
2. If no ICH408I message is found check the `messages.log` file.
   - For the following message:

     ```
     CWWKS3005E: A configuration exception has occurred.
     No UserRegistry implementation service is available.
     Ensure that you have a user registry configured.
     ```

     You must ensure that you have configured a SAF registry in `server.xml`. For more information, see Manually tailoring server.xml.
   - For the following message, when distributed identity is in use:

     ```
     CWWKS9104A: Authorization failed for user alidist:defaultRealm
     while invoking LdapTests on /basic.
     The user is not granted access to any of the required roles: [testing].
     ```

     If `server.xml` is configured for **`<safAuthorization>`** or includes the `cicsts:distributedIdentity-1.0` feature, then ensure the appropriate EJBRoles for the RACMAPped user ID have been defined. For more information, see Authorization using SAF role mapping. If `server.xml` is not configured for **`<safAuthorization>`** and does not include the `cicsts:distributedIdentity-1.0` feature, then ensure that the appropriate distributed user ID is defined to have access to the appropriate role in an **`<application-bnd>`** element. For more information, see Authorizing users to run applications in a Liberty JVM server.
3. If the application is retuning an exception for the class `com.ibm.ws.webcontainer.util.Base64Decode`, check `dfhjvmerr` for error messages. If you see configuration error messages, for example CWWKS4106E or CWWKS4000E, the server is trying to access configuration files that were created in a different encoding. This type of configuration error can occur when you change the **`file.encoding`** value and restart the JVM server. To fix the problem, you can either revert to the previous encoding and restart the JVM server, or delete the configuration files. The JVM server re-creates the files in the correct file encoding when it starts.

Web application is returning an HTTP 500 error code

The web application is returning an HTTP 500 error in the web browser. If you receive an HTTP 500 error, a configuration error occurred.

1. Check the CICS message log for DFHSJ messages, which might give you more information about the specific cause of the error.
2. If you are using a URIMAP to run application requests on a specific transaction, make sure that the URIMAP specifies the correct transaction ID.
3. Make sure that the SCHEME and USAGE attributes are set correctly. The SCHEME must match the application request, either HTTP or HTTPS. The USAGE attribute must be set to JVMSERVER.

Web application is returning an HTTP 503 error code

The web application is returning an HTTP 503 error in the web browser. If you receive an HTTP 503 error, the application is not available.

1. Check the CICS message log for DFHSJ messages for additional information.
2. Make sure that the TRANSACTION and URIMAP resources for the application are enabled. If these resources are packaged as part of the application in a CICS bundle, check the status of the BUNDLE resource.
3. The request might have been purged before it completed. The error messages in the log describe why the request was purged.

Unable to access your web application by using distributed identity mapping

If you are using distributed identity mapping and see the following message in the `messages.log` file:

```
FFDC1015I: An FFDC Incident has been created: "com.ibm.ws.security.saf.SAFException:

CWWKS2905E: SAF service IRRSIA00_CREATE did not succeed because
user null was not found in the SAF registry.
SAF return code 0x00000008. RACF return code 0x00000008. RACF reason code 0x00000010.

FFDC1015I: An FFDC Incident has been created:
"javax.security.auth.login.CredentialException: could not create SAF credential
for <distid> DistId
```

Check the CICS message log for the error message ICH408I to see what type of authorization failure occurred. If it is `ICH408I USER(<userid>) GROUP(TSOUSER ) NAME(<name>) DISTRIBUTED IDENTITY IS NOT DEFINED: 776 cn= <distid> DistId,ou=users,dc=domain,dc=com LdapRegistry` you need to create the appropriate RACMAP for the distributed identity being used to access the application. The **RACMAP QUERY** command is useful for debugging. For example:

```
RACMAP QUERY USERDIDFILTER(NAME('ou=users,dc=domain,dc=com')) REGISTRY(NAME('LdapRegistry'))
```

The web application is returning exceptions

The web application is returning exceptions in the web browser; for example, the application is retuning an exception for the class `com.ibm.ws.webcontainer.util.Base64Decode`.

1. Check `dfhjvmerr` for error messages.
2. If you see configuration error messages, for example CWWKS4106E or CWWKS4000E, the server is trying to access configuration files that were created in a different encoding. This type of configuration error can occur when you change the **`file.encoding`** value and restart the JVM server. To fix the problem, you can either revert to the previous encoding and restart the JVM server, or delete the configuration files. The JVM server re-creates the files in the correct file encoding when it starts.

Error message CWWKB0109E in `messages.log`

If Liberty fails to shut down cleanly, the next Liberty JVM server with WLP_ZOS_PLATFORM=TRUE will write the error message CWWKB0109E to the `messages.log` file. You do not need to fix this error and it can be ignored.

Otherwise, the CICS region user ID is not allowed to register and unregister from **IFAUSAGE**, see The Liberty server angel process.

Error message WTRN0078E

An attempt by the transaction manager to call start on a transactional resource has resulted in an error. The error code was XAER_PROTO. If you experience this error, the most likely scenario is that you have the default JTA integration in operation on your Liberty server, and your application uses a bean method declared as REQUIRES_NEW. For example: @Transactional(value = TxType.REQUIRES_NEW) void yourMethod{} The use of REQUIRES_NEW inside an XA transaction is not supported by CICS. You must alter the application before it will run.

Error message DFHSJ1004 in MSGUSER, but no corresponding STDERR exception

A symptom of running out of zFS file system space could be a DFHSJ1004 with no corresponding STDERR exception. The message is sent because of the lack of space, but there is no exception in STDERR because there is no space to write a message to the files.

You can plan and monitor the size of your file system using the techniques detailed in Managing file systems.

Using the productInfo script to verify integrity of Liberty

You can verify the integrity of the Liberty installation after you install CICS or applying service, by using the productInfo script.

1. Change directory to the `CICS USSHOME` directory.
2. As productInfo uses Java, you must ensure that Java is included in your PATH. Alternatively, set the **JAVA_HOME** environment variable to the value of **JAVA_HOME** in your JVM profile, for example:

   ```
   export JAVA_HOME=/usr/lpp/java/J7.0_64
   ```

3. Run the productInfo script, supplying the validate option `wlp/bin/productInfo validate`. No errors should be reported. For more information about the Liberty **productInfo** script, see Verifying the integrity of Liberty profile installation.

Using the `wlpenv` script to run Liberty commands

You might be asked by IBM service to run one or more of the Liberty supplied commands, such as **productInfo** or **server dump**. To run these commands, you can use the wlpenv script as a wrapper to set the required environment. The script is created and updated every time that you enable a Liberty JVM server after the JVM profile has been successfully parsed. Because the script is unique for each JVM server in each CICS region, it is created in the *WORK_DIR/APPLID/JVMSERVER* as specified by default in the JVM profile and is called `wlpenv`. *APPLID* is the value of the CICS region APPLID and *JVMSERVER* is the name of the JVMSERVER resource.

To run the **wlpenv** script in the UNIX System Services shell, change directory to the *WORK_DIR* as specified in the JVM profile and run the script with the Liberty command as an argument, for example:

```
./wlpenv productInfo version
```

```
./wlpenv server dump --archive=package_file_name.dump.pax --include=heap
```

For the **server dump** command, you do not supply the server name because it is set by the wlpenv script to the value set the last time the JVM server was enabled.

For more information about Liberty commands, see Liberty: productInfo command and Generating a Liberty server dump from the command line.

Troubleshooting invoking a Java EE application

EXEC CICS LINK command fails with RESP = PGMIDERR, RESP2 = 1

1. Check the application to determine whether the correct artifacts have been generated.
   a. Check that annotation processing is enabled on the source project.

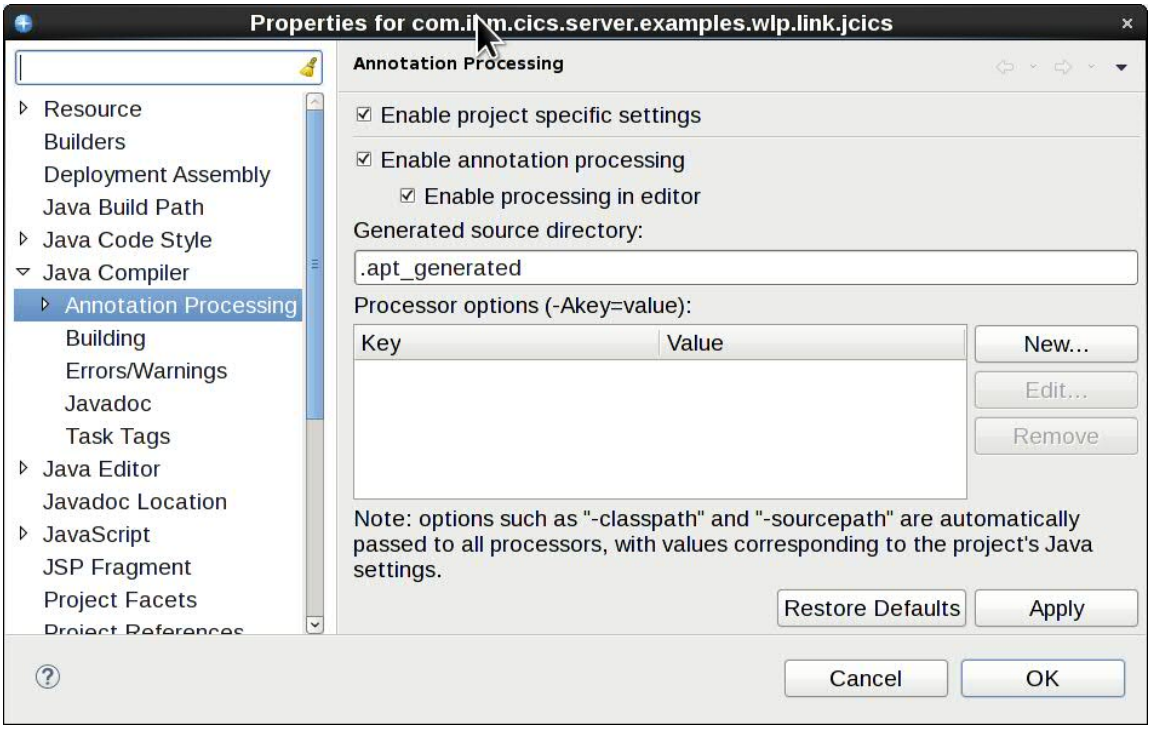


Figure 8. Check annotation processing is enabled

b. Check if an @CICSProgram has been added to a Java method and that it compiles correctly.

c. Export the application and check for generated code in the `com.ibm.cics.server.invocation.proxy` package. For example, on a workstation, open the WAR or EAR file using an archive manager, or on z/OS use the `jar -tf` command, to examine the contents of the WAR or EAR file. If code has not been generated, check you have the latest version of the CICS Explorer, CICS build toolkit, or the annotation processor.

2. Review the CICS message log for messages similar to:

- DFHSJ1204: A linkable service has been registered for class examples.TSQ.ClassOne method anotherMethod with program name LINKJCIN in JVMSERVER LINKJVM
- DFHPG0101: Resource definition for LINKJCIN has been added.

If these messages don't appear then:

a. Ensure you have a Liberty JVM server in the enabled state.

b. Ensure you have the `cicsts:link-1.0` feature configured in your `server.xml`. If it is configured you will see message J2CA7001I: Resource adapter com.ibm.cics.wlp.program.link.connectorinstalled in messages.log.

c. If you are deploying your application using a CICS bundle, ensure the bundle is installed and enabled.

d. Ensure the application is installed in Liberty, if it is, in the `messages.log` you will get a message including the name of the user's application. For example: CWWKZ0001I: Application com.ibm.cics.test.javalink started.

EXEC CICS LINK command fails with RESP = PGMIDERR, RESP = 27

This indicates that CICS tried to invoke a Java EE application in Liberty but a timeout occurred before the application was successful. The most common cause for this issue is that there was no thread available in the JVM server. To resolve this, increase the JVM server thread limit or increase the value of WLP_LINK_TIMEOUT to allow the tasks to wait longer to acquire a thread. For more information see WLP_LINK_TIMEOUT in JVM server options and Managing the thread limit of JVM servers.

JCICS API call throws a CICSRuntimeException

```
com.ibm.cics.server.CicsRuntimeException:
DTCTSQ_READNEXT: No JCICS context is associated with the current thread.
```

The most likely cause of this exception is that you created a JCICS object on one thread and tried to call its instance methods from a different thread. Change your application to construct the JCICS object on the same thread that calls its methods.

Patterns that lead to inadvertently using an object on a different thread include:

- Constructing a JCICS object in constructor of a `java.lang.Runnable` or `java.util.concurrent.Callable`. Construct the object in the `run()` method instead.
- Assigned JCICS objects to static variables. Use instance variables instead.
- Passing a JCICS object as a parameter to a method that is executed by another thread. The thread should construct JCICS object itself.

Transaction abends AJ05 when using invoking a Java EE application

The following exceptions will be logged to the `dfhvjmerr` file:

```
com.ibm.cics.server.InvalidRequestException: CICS INVREQ Condition(RESP=INVREQ, RESP2=200)
java.lang.RuntimeException:
javax.transaction.RollbackException:
XAResource start association error:XAER_PROTO
```

Using JTA with Link to Liberty is only supported wth CICS JTA integration disabled. Configure this by using `<cicsts_jta integration="false"/>` in `server.xml`.

Error message CWWKC2262E The server is unable to process the 4.0 version and the http://xmlns.jcp.org/xml/ns/javaee namespace

If the server is unable to process the 4.0 version and the `http://xmlns.jcp.org/xml/ns/javaee` namespace, this typically means that an application server, such as Tomcat has not been excluded from the build script. In Gradle, ensure that you have specified `providedRuntime("org.springframework.boot:spring-boot-starter-tomcat")`, while in Maven you have used the scope 'provided', for example:

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-tomcat</artifactId>
<scope>provided</scope>
</dependency>
```

Controlling the location for JVM output, logs, dumps and trace

Output from Java applications that are running in a JVM server can be written to the z/OS UNIX files. The z/OS UNIX files are named by the STDOUT, STDERR, and JVMTRACE options in the JVM server or routed to the JES log. In a Liberty JVM server, Liberty server output can be found in `messages.log` relative to the configured log directory.

By default, the output from Java applications that are running in a JVM server is written to the z/OS UNIX file system. The z/OS UNIX file system follows the file name convention `DATE.TIME.<dfhjvmxxx>` within the directory structure of `$WORK_DIR/APPLID/JVMSERVER`. These overrides might also be used to route the output to the JES log. For more information, see Using a DD statement to route JVM server output to JES.

If you want to override the defaults, you can specify a zFS file name for the STDOUT, STDERR, and JVMTRACE options. However, if you use a fixed file name, the output from all the JVMs that were created with that JVM profile is appended to the same file. The output from different JVMs is interleaved with no record headers. This situation is not helpful for problem determination.

If you customize these values, a better choice is to specify a variable file name for the STDOUT, STDERR, and JVMTRACE options. The files can then be made unique to each individual JVM during the lifetime of the CICS region.

You can include the CICS region APPLID in the file name by using the *APPLID* symbol.

You can include extra identifying information in file names. Such identifying information includes the *DATE* and *TIME* symbols.

DATE is replaced by the date the profile parses on JVM server start, in the form Dyymmdd.
TIME is replaced by the time the profile parses on JVM server start, in the format Thhmmss.
JVMSERVER is replaced by the name of the JVMSERVER resource.

Further customization can be achieved at the programmatic level that uses the USEROUTPUTCLASS option, which does not work with Liberty. The USEROUTPUTCLASS option, which is specified in the JVM profile, names a Java class. A Java class intercepts and redirects the output from the JVM to a custom location such as a CICS transient data queue. You can add time stamps and headers to the output records, and identify the output from individual transactions that are running in the JVM. CICS supplies sample classes that perform these tasks.

The default location for JAVADUMP files output from the JVM is the working directory on z/OS UNIX named by the WORK_DIR option in the JVM profile. JAVADUMP files are uniquely identified by a time stamp in their names. To override the default, you can use **-Xdump:directory=<path>** to specify a location for all dump types to be written to. For details about -Xdump, see -Xdump.

The more detailed Java TDUMPs are written to the file named by the JAVA_DUMP_TDUMP_PATTERN option. You can use the *APPLID*, *DATE*, and *TIME*, and *JVMSERVER* symbols in this value to make the name unique to the individual JVM, as shown in the sample JVM profiles included with CICS.

The JVM writes information to the stderr stream when it generates a JAVADUMP or a TDUMP. For more information about the contents of JAVADUMP and TDUMP files, see IBM SDK for z/OS, Java Technology Edition, Version 7, Troubleshooting and support section.

Using a DD statement to route JVM server output to JES

You can update the JVM server to redirect output to a specific location.

JVM server STDOUT, STDERR, JVMTRACE, and messages.log output can be routed to the JES log. This allows JVM server log file output to be managed together with other CICS logs such as the MSGUSR.

Using the JOBLOG parameter results in STDOUT and JVMTRACE being routed to SYSPRINT if defined or to a dynamic SYSnnn if not. If only JVMTRACE=JOBLOG is specified, JVMTRACE is routed to the current stdout location. STDERR is routed to SYSOUT if defined or to a dynamic SYSnnn if not, for example:

```
STDOUT=JOBLOG
STDERR=JOBLOG
JVMTRACE=JOBLOG
```

Output can also be routed to any MVS data definition (DD) defined to JES, for example if the CICS region JCL specifies the DD statements JVMOUT, JVMERR, and MSGLOG.

```
//JVMOUT   DD SYSOUT=*
//JVMERR   DD SYSOUT=*
//MSGLOG   DD SYSOUT=*    <--- redirects Liberty messages.log
```

If the DD statements configured in Liberty are not defined in CICS runtime JCL, these logs are automatically redirected to the specified DD output and are listed in CICS job output when Liberty is started.

The following JVM profile options can then be used in the JVM profile to route stdout and stderr streams to the JVMOUT and JVMERR destinations. If omitted, the JVM server will automatically create

those destinations. A MSGLOG statement automatically redirects `messages.log` to JES without the need for any JVM profile configuration.

```
STDOUT=//DD:JVMOUT
STDERR=//DD:JVMERR
```

To establish the origin of the JVM server output, all `stdout`, and `stderr` entries that are routed to JES are written with a prefix string of the JVM server name, which is useful if multiple JVM servers are sharing a destination. This behavior can be disabled by using the JVM profile option `IDENTITY_PREFIX`, which if set to FALSE disables use of the prefix string.

If you choose not to specify a destination, the output will redirect to the zFS default file, however you can set it to send to specific zFS files. See "Controlling the location for JVM output, logs, dumps and trace" on page 259.

Redirecting the JVM stdout and stderr streams

During application development, the USEROUTPUTCLASS option can be used by developers to separate out their own `stdout` and `stderr` entries in a CICS region, and direct them to an identifiable destination of their choice. You can use a Java class to redirect the output, and you can add time stamps and headers to the output records. Dump output cannot be intercepted by this method.

Specifying the USEROUTPUTCLASS option has a negative effect on the performance of JVMs. For best performance in a production environment, do not use this option.

Output that is written to `System.out()` or `System.err()`, either by an application or by system code, can be redirected by the output redirection class. The z/OS UNIX files that are named by the STDOUT and STDERR options in the JVM profile are still used for some messages that are issued by the JVM, or if the class named by the USEROUTPUTCLASS option is unable to write data to its intended destination. You must therefore still specify appropriate file names for these files.

To use the USEROUTPUTCLASS option, specify `USEROUTPUTCLASS=[java class]` in a JVM profile, naming the Java class of your choice. The class extends `java.io.OutputStream`. The supplied sample JVM profiles contain the commented-out option `USEROUTPUTCLASS=com.ibm.cics.samples.SJMergedStream`, which names the supplied sample class. Uncomment this option to use the `com.ibm.cics.samples.SJMergedStream` class to handle output from JVMs with that profile. CICS also supplies an alternative sample Java class, `com.ibm.cics.samples.SJTaskStream`.

For JVM servers, you package your output redirection class as an OSGi bundle to run the class in the OSGi framework. For more information, see Writing Java classes to redirect JVM stdout and stderr output.

Note: Output redirection samples function in OSGi and classpath JVM servers and not in a Liberty JVM server.

The sample classes com.ibm.cics.samples.SJMergedStream and com.ibm.cics.samples.SJTaskStream

For Java application threads that can make CICS requests, you can intercept the output from the JVM and write it to a transient data queue. A log is created that correlates JVM activity with CICS activity.

You can add time stamps, task and transaction identifiers, and program names when the output is intercepted. You can therefore create a merged log file that contains the output from multiple JVMs. You can use this log file to correlate JVM activity with CICS activity. The sample class, `com.ibm.cics.samples.SJMergedStream`, is set up to create merged log files.

The `com.ibm.cics.samples.SJMergedStream` class directs output from the JVM to the transient data queues CSJO (for the `stdout` stream), and CSJE (for the `stderr` stream and internal messages). These transient data queues are supplied in group DFHDCTG, and they are redirected to CSSL, but you can redefine them if required.

By redirecting the output, the class adds a header to each record that contains the date, time, APPLID, TRANSID, task number, and program name. The result is two merged log files for JVM output and for error messages, in which the source of the output and messages can easily be identified.

The classes are shipped in the file `com.ibm.cics.samples.jar`, which is in the directory `/usr/lpp/cicsts/cicsts54/lib`, where `/usr/lpp/cicsts/cicsts54` is the installation directory for CICS files on z/OS UNIX. The source for the classes is also provided as samples, so you can modify the classes as you want, or write your own classes based on the samples. The classes are packaged as an OSGi bundle JAR. These classes can either be deployed into a CLASSPATH JVM server or as a middleware bundle that uses the OSGI_BUNDLES JVM server option in an OSGi JVM server. For more information, see Writing Java classes to redirect JVM stdout and stderr output.

Java applications that run on threads other than the ones that are attached by CICS are not able to make CICS requests. The output from the JVM cannot be redirected by using CICS facilities. The `com.ibm.cics.samples.SJMergedStream` class still intercepts the output and adds a header to each record. The output is written to the z/OS UNIX files */work_dir/applid/*`stdout/CSJO` and */work_dir/applid/*`stderr/CSJE` as referred to previously. If these files are unavailable, the output is written to the z/OS UNIX files named by the STDOUT and STDERR options in the JVM profile.

As an alternative to creating merged log files for your JVM output, you can direct the output from a single task to z/OS UNIX files. You can also add time stamps and headers, to provide output streams that are specific to a single task. The sample class that is supplied with CICS, `com.ibm.cics.samples.SJTaskStream` is set up for this purpose. The class directs the output for each task to two z/OS UNIX files. One is for the `stdout` stream and one is for the `stderr` stream. The output entries within the streams are uniquely named by using a task number (in the format YYYYMMDD.task.*tasknumber*). The z/OS UNIX files are stored in directories called STDOUT and STDERR respectively. The process is the same for Java applications which run on threads that are attached by CICS, and Java applications that are running on other threads.

Error handling

The length of messages that are given by the JVM can vary. The maximum record length for the CSSL queue (133 bytes) might not be sufficient to contain some of the messages you receive. If you receive more messages than the maximum record length for the queue, the sample output redirection class issues an error message. The text of the message might be affected.

If you find that you are receiving messages longer than 133 bytes from the JVM, redefine CSJO and CSJE as separate transient data queues. Make them extrapartition destinations, and increase the record length for the queue. You can allocate the queue to a physical data set or to a system output data set. You might find a system output data set more convenient in this case, because you do not then need to close the queue to view the output. For information about how to define transient data queues, see TDQUEUE resources. If you redefine CSJO and CSJE, ensure that they are installed as soon as possible during a cold start, in the same way as for transient data queues that are defined in group DFHDCTG.

If the transient data queues CSJO and CSJE cannot be accessed, output is written to the z/OS UNIX files */work_dir/applid/*`stdout/CSJO` and */work_dir/applid/*`stderr/CSJE`, where *work_dir* is the directory that is specified on the `WORK_DIR` option in the JVM profile, and *applid* is the APPLID identifier that is associated with the CICS region. If these files are unavailable, the output is written to the z/OS UNIX files named by the STDOUT and STDERR options in the JVM profile.

When an error is encountered by the sample output redirection classes, one or more error messages are given. If the error occurred while you processed an output message, then the error messages are directed to `System.err`, and are eligible for redirection. However, if the error occurred while you processed an error message, then the new error messages are sent to the file named by the STDERR option in the JVM profile, avoiding a recursive loop in the Java class. The classes do not return exceptions to the calling Java program.

Control of Java dump options

The -Xdump option can be used in a JVM profile to specify dump options to the JVM.

Information about Java dump options can be found in IBM SDK for z/OS, Java Technology Edition, Version 7, Troubleshooting and support section.

CICS component tracing for JVM servers

In addition to the logging produced by Java, CICS provides some standard trace points in the SJ (JVM) and AP domains for 0, 1, and 2 trace levels. These trace points trace the actions that CICS takes in setting up and managing JVM servers.

You can activate the SJ and AP domain trace points at levels 0, 1, and 2 using the CETR transaction. For details of all the standard trace points in the SJ domain, see JVM domain trace points.

SJ and AP component tracing

The SJ component traces exceptions and processing in SJ domain to the internal trace table. The AP component traces the installation of OSGi bundles in the OSGi framework. SJ level 3, 4, and 5 tracing produce Java logging that is written to a trace file in zFS. The name and location of the trace file is determined by the JVMTRACE option in the JVM profile.

SJ level 4 and 5 tracing produces verbose logging information in the trace file. If you want to use this trace level, you must ensure that there is enough space in zFS for the file. For more information about activating and managing trace, see “Activating and managing tracing for JVM servers” on page 263.

Activating and managing tracing for JVM servers

You can activate JVM server tracing by turning on SJ and AP component tracing. Small amounts of trace are written to the internal trace table, but Java also writes out logging information to a unique file in zFS for each JVM server. This file does not wrap so you must manage its size in zFS.

About this task

JVM server tracing does not use auxiliary or GTF tracing. CICS writes some information to the internal trace table. However, most diagnostic information is logged by Java and written to a file in zFS. This file is uniquely named for each JVM server. The default file name has the format *&DATE;.&TIME;*.dfhjvmtrc and is created by CICS in the $WORK_DIR/&APPLID;/&JVMSERVER; directory when you enable the JVMSERVER resource. You can change the name and location of the trace file in the JVM profile. If you delete or rename the trace file when the JVM server is running, CICS does not re-create the file and the logging information is not written to another file.

Procedure

1. Use the CETR transaction to activate tracing for the JVM server.

   You can use two components to produce tracing and logging information for a JVM server:

   - Select the SJ component to trace the actions taken by CICS to start and stop the JVM server. The JVM logs diagnostic information in the zFS file.
   - Select the AP component to trace the installation of OSGi bundles.
2. Set the tracing level for the SJ and AP components:
   - SJ level 0 produces tracing for exceptions only, such as errors during the initialization of the JVM server or problems in the OSGi framework. SJ level 1 and level 2 produces more CICS tracing from the SJ domain. This tracing is written to the internal trace table.
   - SJ level 3 produces additional logging from the JVM, such as warning and information messages in the OSGi framework. This information is written to the trace file in zFS.

- SJ level 4, 5 and AP level 2 produce debug information from CICS and the JVM, which provides much more detailed information about the JVM server processing. This information is written to the trace file in zFS.

3. Each trace entry has a date and time stamp. You can change the name and the location of this trace file by using the JVMTRACE profile option.
4. If you are using the default JVMTRACE settings, when you enable the JVMSERVER resource CICS creates a new unique trace file for the life of the JVM.

   If you disable the JVMSERVER resource, you can delete the trace file or rename the file if you want to retain the information separately.
5. To manage the number of files you can set the LOG_FILES_MAX option to control the number of old trace files that are retained on the JVM server startup.

Debugging a Java application

The JVM in CICS supports the Java Platform Debugger Architecture (JPDA), which is the standard debugging mechanism provided in the Java Platform.

About this task

You can use any tool that supports JDPA to debug a Java application running in CICS. For example, you can use the Java Debugger (JDB) that is included with the Java SDK on z/OS. To attach a JPDA remote debugger, you must set some options in the JVM profile.

IBM provides monitoring and diagnostic tools for Java, including Health Center. IBM Health Center is available in the IBM Support Assistant Workbench. These free tools are available to download from IBM as described in the Getting Started guide for IBM Health Center.

Procedure

1. Add the debugging option to the JVM profile to start the JVM in debug mode:

   ```
   -agentlib:jdwp=transport=dt_socket,server=y,address=port,suspend=n
   ```

   Select a free port to connect to the debugger remotely.

   If the JVM profile is shared by more than one JVM server, you can use a different JVM profile for debugging.

   Note: The default value for suspend is y. This value suspends the JVM and waits for the remote client debugger to attach before processing continues. Specifying a value of n will prevent the JVM server from suspending.
2. Add these properties to the JVM profile when debugging a Liberty JVM server to avoid hot-swap complications with Liberty trace. This will also indicate to Liberty that it should operate in a debug cognizant mode:

   ```
   -Dwas.debug.mode=true
   -Dcom.ibm.websphere.ras.inject.at.transform=true
   ```
3. Attach the debugger to the JVM.

   If an error occurs during the connection, for example the port value is incorrect, messages are written to the JVM standard output and standard error streams.
4. Using the debugger, check the initial state of the JVM. For example, check the identity of threads that are started and system classes that are loaded.
5. Set a breakpoint at a suitable point in the Java application by specifying the full Java class name and source code line number. If the debugger indicates that activation of this breakpoint is deferred, it is because the class might not yet have loaded.

   Let the JVM run through the CICS middleware code to the application breakpoint, at which point it suspends execution again.

6. Examine the source code of the loaded classes and variables and set further breakpoints to step through the code as required.
7. End the debug session. You can let the application run to completion, at which point the connection between the debugger and the CICS JVM closes. Some debuggers support forced termination of the JVM, which results in an abend and error messages on the CICS system console.

The CICS JVM plug-in mechanism

In addition to the standard JPDA debug interfaces in the JVM, CICS provides a set of interception points (plug-ins) in the CICS Java middleware, which can be useful for debugging applications. You can use these plug-ins to insert additional Java programs immediately before and after the application Java code is run.

Information about the application, for example, class name and method name, is made available to the plug-ins. The plug-ins can also use the JCICS API to obtain information about the application, and can also be used in conjunction with the standard JPDA interfaces to provide additional debug facilities specifically for CICS. The plug-ins can also be used for purposes other than debugging, in a similar way to CICS user exits.

The Java exit is a CICS Java wrapper plug-in that provides methods that are called immediately before and after a Java program is invoked.

To deploy a plug-in, you package the plug-in as an OSGi bundle. For more information see Deploying OSGi bundles in a JVM server.

Two Java programming interfaces are provided.

Both interfaces are supplied in `com.ibm.cics.server.jar`, and are documented in the Javadoc.

The Java programming interfaces are:

- DebugControl: `com.ibm.cics.server.debug.DebugControl`. This programming interface defines the method calls that can be made to an implementation supplied by the user.
- Plugin: `com.ibm.cics.server.debug.Plugin`. This is a general purpose programming interface that you use for registering the plug-in implementation.

Here is an example of the DebugControl interface:

```
public interface DebugControl
{
    // called before an application object method or program main is invoked
    public void startDebug(java.lang.String className,java.lang.String methodName);

    // called after an application object method or program main is invoked
    public void stopDebug(java.lang.String className,java.lang.String methodName);

    // called before an application object is deleted
    public void exitDebug();

}
public interface Plugin
{
    // initaliser, called when plug-in is registered
    public void init();
}
```

Here is an example implementation of the DebugControl and Plugin interfaces:

```
import com.ibm.cics.server.debug.*;

public class SampleCICSDebugPlugin
    implements Plugin, DebugControl
{
    // Implementation of the plug-in initialiser
    public void init()
    {
        // This method is called when the CICS Java middleware loads and
        // registers the plug-in. It can be used to perform any initialization
        // required for the debug control implementation.
```

```
    }

    // Implementations of the debug control methods
    public void startDebug(java.lang.String className,java.lang.String methodName)
    {
        // This method is called immediately before the application method is
        // invoked. It can be used to start operation of a debugging tool. JCICS
        // calls such as Task.getTask can be used here to obtain further
        // information about the application.
    }

    public void stopDebug(java.lang.String className,java.lang.String methodName)
    {
        // This method is called immediately after the application method is
        // invoked. It can be used to suspend operation of a debugging tool.
    }

    public void exitDebug()
    {
        // This method is called immediately before an application object is
        // deleted. It can be used to terminate operation of a debugging tool.
    }

    public static void main(com.ibm.cics.server.CommAreaHolder ca)
    {
    }
}
```

Notices

This information was developed for products and services offered in the United States of America. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property rights may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who want to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119 Armonk,
NY 10504-1785
United States of America

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Client Relationship Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

The performance data discussed herein is presented as derived under specific operating conditions. Actual results may vary.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Programming interface information

IBM CICS supplies some documentation that can be considered to be Programming Interfaces, and some documentation that cannot be considered to be a Programming Interface.

Programming Interfaces that allow the customer to write programs to obtain the services of CICS Transaction Server for z/OS, Version 5 Release 4 (CICS TS 5.4) are included in the following sections of the online product documentation:

- Developing applications
- Developing system programs
- Securing overview
- Developing for external interfaces
- Application development reference
- Reference: system programming
- Reference: connectivity

Information that is NOT intended to be used as a Programming Interface of CICS TS 5.4, but that might be misconstrued as Programming Interfaces, is included in the following sections of the online product documentation:

- Troubleshooting and support
- Reference: diagnostics

If you access the CICS documentation in manuals in PDF format, Programming Interfaces that allow the customer to write programs to obtain the services of CICS TS 5.4 are included in the following manuals:

- Application Programming Guide and Application Programming Reference
- Business Transaction Services

- Customization Guide
- C++ OO Class Libraries
- Debugging Tools Interfaces Reference
- Distributed Transaction Programming Guide
- External Interfaces Guide
- Front End Programming Interface Guide
- IMS Database Control Guide
- Installation Guide
- Security Guide
- CICS Transactions
- CICSPlex System Manager (CICSPlex SM) Managing Workloads
- CICSPlex SM Managing Resource Usage
- CICSPlex SM Application Programming Guide and Application Programming Reference
- Java Applications in CICS

If you access the CICS documentation in manuals in PDF format, information that is NOT intended to be used as a Programming Interface of CICS TS 5.4, but that might be misconstrued as Programming Interfaces, is included in the following manuals:

- Data Areas
- Diagnosis Reference
- Problem Determination Guide
- CICSPlex SM Problem Determination Guide

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Apache, Apache Axis2, Apache Maven, Apache Ivy, the Apache Software Foundation (ASF) logo, and the ASF feather logo are trademarks of Apache Software Foundation.

Gradle and the Gradlephant logo are registered trademark of Gradle, Inc. and its subsidiaries in the United States and/or other countries.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Red Hat®, and Hibernate® are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

Spring Boot is a trademark of Pivotal Software, Inc. in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Zowe™, the Zowe logo and the Open Mainframe Project™ are trademarks of The Linux Foundation.

The Stack Exchange name and logos are trademarks of Stack Exchange Inc.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability
These terms and conditions are in addition to any terms of use for the IBM website.

Personal use
You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use
You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights
Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM online privacy statement

IBM Software products, including software as a service solutions, (*Software Offerings*) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information (PII) is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect PII. If this Software Offering uses cookies to collect PII, specific information about this offering's use of cookies is set forth below:

For the CICSPlex SM Web User Interface (main interface):
Depending upon the configurations deployed, this Software Offering may use session and persistent cookies that collect each user's user name and other PII for purposes of session management, authentication, enhanced user usability, or other usage tracking or functional purposes. These cookies cannot be disabled.

For the CICSPlex SM Web User Interface (data interface):
Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user's user name and other PII for purposes of session management, authentication, or other usage tracking or functional purposes. These cookies cannot be disabled.

For the CICSPlex SM Web User Interface ("hello world" page):
Depending upon the configurations deployed, this Software Offering may use session cookies that do not collect PII. These cookies cannot be disabled.

For CICS Explorer:

Depending upon the configurations deployed, this Software Offering may use session and persistent preferences that collect each user’s user name and password, for purposes of session management, authentication, and single sign-on configuration. These preferences cannot be disabled, although storing a user's password on disk in encrypted form can only be enabled by the user's explicit action to check a check box during sign-on.

If the configurations deployed for this Software Offering provide you, as customer, the ability to collect PII from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM Privacy Policy and IBM Online Privacy Statement, the section entitled *Cookies, Web Beacons and Other Technologies* and the IBM Software Products and Software-as-a-Service Privacy Statement.

Index

Special Characters

A

B

C

D

E

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Z