CICS Transaction Server for z/OS
Version 5 Release 2

Java Applications in CICS

SC34-7282-00

CICS Transaction Server for z/OS
Version 5 Release 2

Java Applications in CICS

SC34-7282-00

Note

Before using this information and the product it supports, read the information in "Notices" on page 183.

This edition applies to the IBM CICS Transaction Server for z/OS Version 5 Release 2 (product number 5655-Y04) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Preface

This manual documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM® CICS® Transaction Server Version 5 Release 2.

What this information is about

This information tells you how to develop and use Java applications and enterprise beans in CICS.

Who should read this information

This information is intended for:

- Experienced Java application programmers who may have little experience of CICS, and no great need to know more about CICS than is necessary to develop and run Java programs.
- Experienced CICS users and system programmers, who need to know about CICS requirements for Java support.

Location of topics in the information center

The topics in this publication can also be found in the CICS information center. The information center uses content types to structure how the information is displayed.

The information center content types are generally task-oriented, for example; upgrading, configuring, and installing. Other content types include reference, overview and scenario or tutorial-based information. The following mapping shows the relationship between topics in this publication and the information center content types, with links to the external information center:

Table 1. Mapping of PDF topics to information center content types. This table lists the relationship between topics in the PDF and topics in the content types in the information center

Set of topics in this publication	Location in the information center
• Chapter 1, "Getting started with Java," on page 1 • Chapter 2, "Developing Java applications for CICS," on page 23 • Chapter 3, "Setting up Java support," on page 79 • Chapter 4, "Deploying applications to a JVM server," on page 117 • Chapter 5, "Administering Java applications," on page 125 • Chapter 6, "Improving Java performance," on page 137 • Chapter 8, "Troubleshooting Java applications," on page 165	• Getting started • Configuring • Administering • Developing applications • Deploying • Improving performance • Troubleshooting and support

Changes in CICS Transaction Server for z/OS, Version 5 Release 2

For information about changes that have been made in this release, please refer to *What's New* in the information center, or the following publications:

- *CICS Transaction Server for z/OS What's New*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 5.1*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 4.2*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 4.1*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.2*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.1*

Any technical changes that are made to the text after release are indicated by a vertical bar (|) to the left of each new or changed line of information.

Chapter 1. Getting started with 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 an IBM System z® 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" on page 2
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 applications that comply with OSGi" on page 14
CICS includes the Equinox implementation of the OSGi framework to run Java applications that comply with the OSGi specification in a JVM server.

"Web applications and web services in a Liberty JVM server" on page 17
CICS provides a web container that can run lightweight Java servlets and JavaServer Pages. Developers can use the rich features of the Java servlet and

JSP specifications to write modern web applications for CICS. The web container runs in a JVM server and is built on the WebSphere® Application Server Liberty profile technology.

"Java web services" on page 18
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.

Related information:

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.

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 or
| Version 7 Release 1. Either version may be used, but all JVM servers running in a
| CICS region must use the same version. The SDK contains a Java Runtime Environment that supports the full set of Java APIs and a set of development tools. To encourage the adoption of Java on z/OS, a special processor is available in certain System z hardware. This processor is called the IBM System z Application Assist Processor (zAAP) and 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 http://www.ibm.com/servers/eserver/zseries/software/java/.

CICS provides an Eclipse-based tool and the JVM server runtime environment for Java applications:

CICS Explorer® SDK

The CICS Explorer SDK is a freely available download for Eclipse-based Integrated Development Environments (IDEs). The SDK 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 CICS Explorer SDK supports developing Java applications for any supported release of CICS. The SDK includes the Java CICS (JCICS) library of classes to access CICS services and examples to get started with developing applications for CICS. You can also use the tool to convert existing Java applications to OSGi.

The CICS Explorer SDK supports the packaging of Liberty applications in to CICS bundles that can be deployed in CICS.

JVM server

The JVM server is the strategic runtime environment for Java applications in CICS. A JVM server can handle many concurrent requests from 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 specification. JVM server provides the following benefits:

- Eligible Java workloads can run on zAAP 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.
- Web applications can be deployed in to the Liberty JVM server.

"The OSGi Service Platform" on page 4
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.

"JVM server runtime environment" on page 5
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.

"JVM profiles" on page 8
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.

"Structure of a JVM" on page 9
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.

"CICS task and thread management" on page 11
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 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 additional tasks in CICS asynchronously. CICS takes special measure to deal with runaway tasks.

"Shared class cache" on page 13
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.

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 version Java applications, can be deployed into a live production system without having to restart the JVM, and without impacting the other Java applications deployed on 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 on to 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.

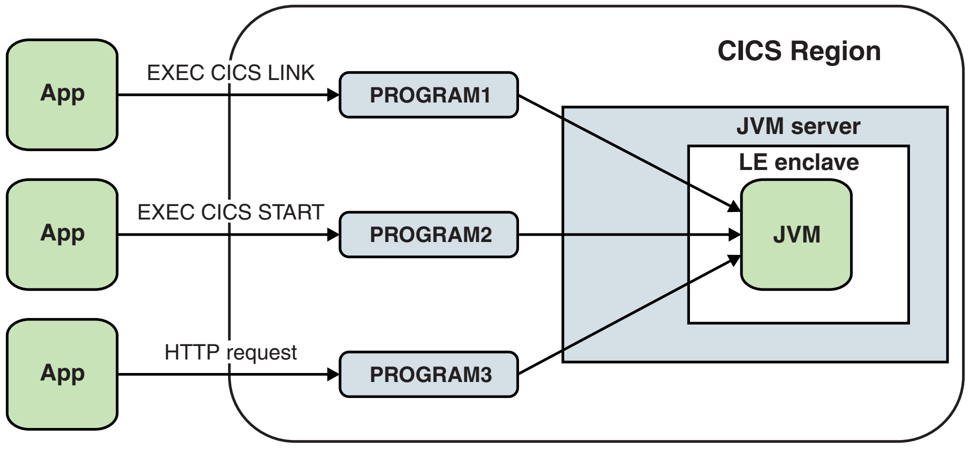
Services are fully dynamic and typically have the same life cycle as the bundle that provides them.

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 WebSphere 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.

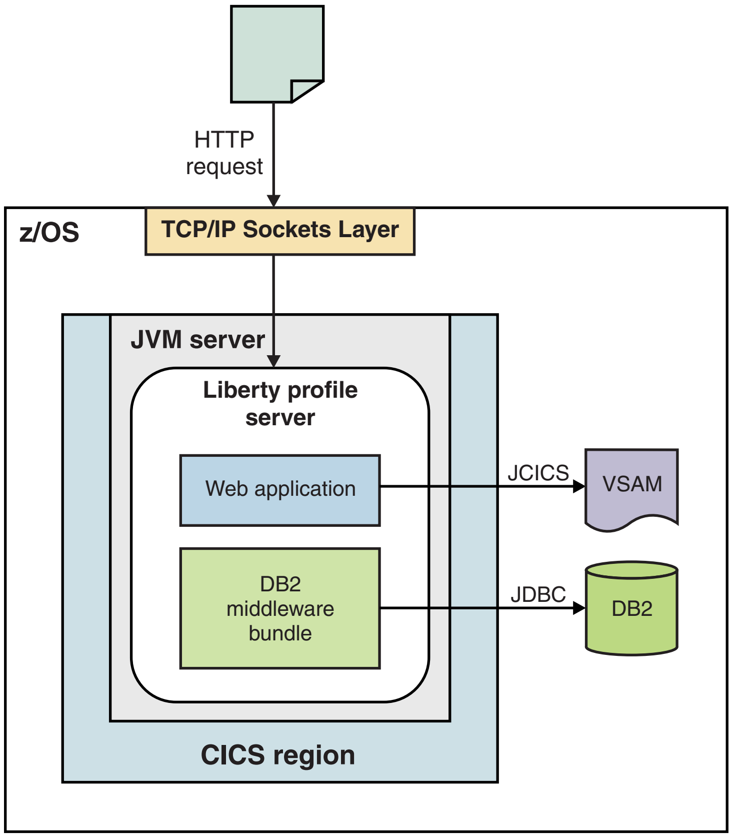
The unit of deployment for a Java application that is packaged using OSGi is a CICS bundle. The BUNDLE resource represents the application to CICS and you can use it to manage the lifecycle of the application. The CICS Explorer SDK provides support for deploying OSGi bundles in a CICS bundle project 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 14.

Java web applications

In addition to running Java applications in an OSGi framework, the JVM server also supports running a WebSphere Application Server Liberty profile. The Liberty profile is a lightweight application server for running web applications. The Liberty profile also runs in an OSGi framework so you can run Java web components and OSGi bundles in the same JVM server. Web applications can use JCICS to access resources and services in CICS, and to access data in DB2. Applications running in the Liberty profile server 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 WAR files directly into the drop-in directory of the Liberty server, 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, CICS Explorer SDK, and IBM WebSphere Application Server Developer Tools for Eclipse version 8.5.5. The latter set of tools is free to use but, apart from CICS Explorer SDK, IBM support is not available for them.

For more information about using a Liberty profile server, see “Web applications and web services in a Liberty JVM server” on page 17.

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 18.

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. See JVMSERVER attributes.

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/cicsts52/JVMProfiles` on z/OS UNIX. 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 2. 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 Version 7 of the IBM 64-bit SDK for z/OS, Java Technology Edition, see User guide for the IBM SDK for z/OS, Java Technology Edition, Version 7.

"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.

"Storage heap in JVMs" on page 10
The runtime storage in JVMs for IBM 64-bit SDK for z/OS, Java Technology Edition Version 7 is managed by a single 64-bit storage heap.

"Where JVMs are constructed" on page 10
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.

"JVMs and the z/OS shared library region" on page 11
The shared library region is a z/OS feature that enables address spaces to share dynamic link library (DLL) files.

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 storage in JVMs for IBM 64-bit SDK for z/OS, Java Technology Edition Version 7 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" on page 145.

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” on page 146.

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” on page 150.

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 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 additional tasks in CICS asynchronously. CICS takes special measure to deal with runaway tasks.

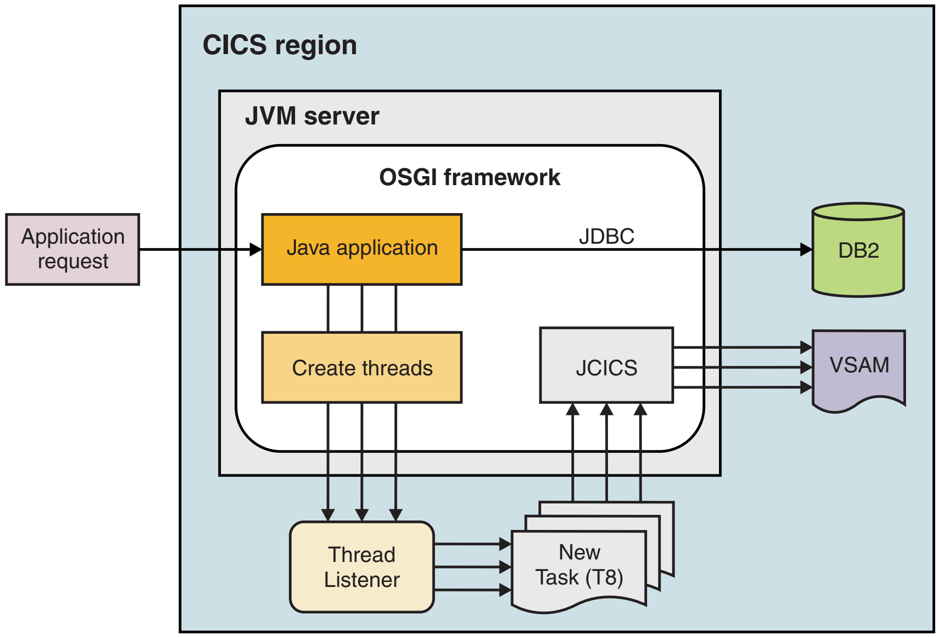
When CICS enables a JVM server, the JVM server runs on a Language Environment process thread. This thread is a child of the TP TCB. Every CICS task is attached to a thread in the JVM by using a T8 TCB. You can control how many T8 TCBs are available to the JVM server by setting the THREADLIMIT attribute 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. If a JVM-based task retains control of the processor for longer than the effective runaway limit, the associated thread in the JVM is terminated with an AICA or similar abend.

The termination of the thread can leave the JVM in an inconsistent state. The Java programming language and JVM are not designed to allow threads to be stopped in this way. For example, locks might have been held by the terminated thread, finalizer blocks might not have run, resources might remain allocated, and shared storage might be left in an inconsistent state. Both application and system state might be affected.

If a thread is terminated due to runaway processing, CICS automatically restarts the JVM server to recover the JVM back to a consistent state. The restart involves a phaseout of the JVM server, followed by an immediate re-enablement. The JVM server cannot process new work during this interval. CICS issues message DFHSJ1009 to report that the restart has happened.

A runaway condition for a task that is running in a JVM server can cause temporary availability problems for all uses of the same JVM server. For this reason, CICS modifies the runaway timeout 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 limit. This higher runaway limit 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 value for that task when it runs in a JVM server is 50000 milliseconds.

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
```

For more information about Java class data sharing see Class data sharing between JVMs.

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 4, 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 CICS Explorer SDK and the CICS Explorer:

- The CICS Explorer SDK 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.
- The 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.

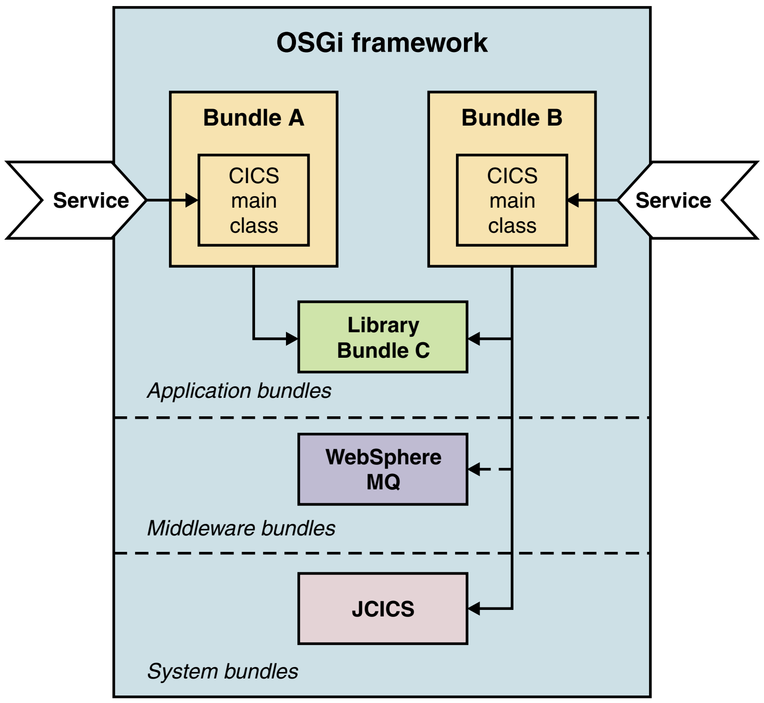
Any Java developer or systems administrator who wants to work with OSGi requires access to these freely available tools.

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 have decided 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. To make the application 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 life cycle 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

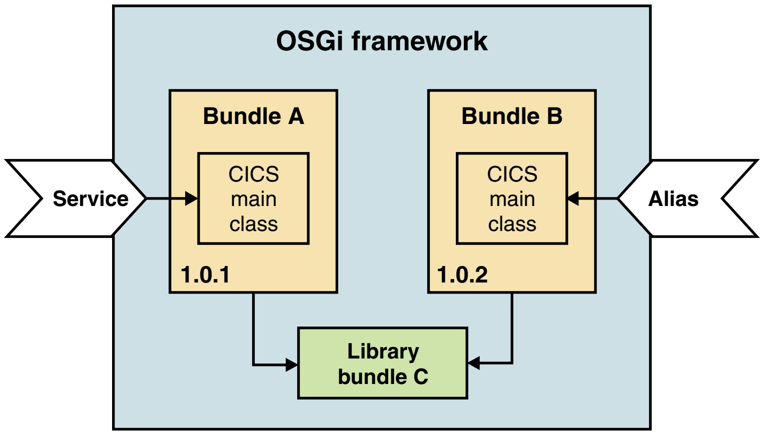
A system bundle is an OSGi bundle that manages the interaction between CICS and the OSGi framework to provide key services to the applications. The primary example is the JCICS OSGi bundles, which provide access to CICS services and resources.

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 creating dependencies between bundles. Using a range of versions mean that an application can tolerate compatible updates to bundles that it depends on.
- 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. However, you cannot have multiple versions of the same OSGi service in the framework. If different versions of the OSGi bundle have the same CICS main class, you can use an alias to override the duplicate service. The alias is specified with the declaration of the CICS main class and registered in the OSGi framework as the OSGi service for the updated version of the bundle. Specify the alias on another PROGRAM resource to make the application available.



Related information:

Configuring a JVM server for an OSGi application
Configure the JVM server to run an OSGi framework if you want to deploy Java applications that are packaged in OSGi bundles.

Getting started with the JCICS examples

Web applications and web services in a Liberty JVM server

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

The Liberty profile is a lightweight web container 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 the Liberty profile, see Liberty profile overview.

The Liberty profile technology is installed with CICS to run as a web container in a JVM server. The Liberty JVM server supports a subset of the features that are available in the Liberty profile; 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 profile 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 a web container that is based on the Liberty profile.
- You want to use DataSource 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.

Related concepts:

Configuring a Liberty JVM server for web applications in Configuring

Developing Java web applications

Related information:

Configuring a JVM server for the Liberty profile
Configure the Liberty JVM server to run a web container if you want to deploy Java web applications that use Java servlets and JSP pages. The web container is based on the Liberty profile technology.

Getting started with the servlet examples

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 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.

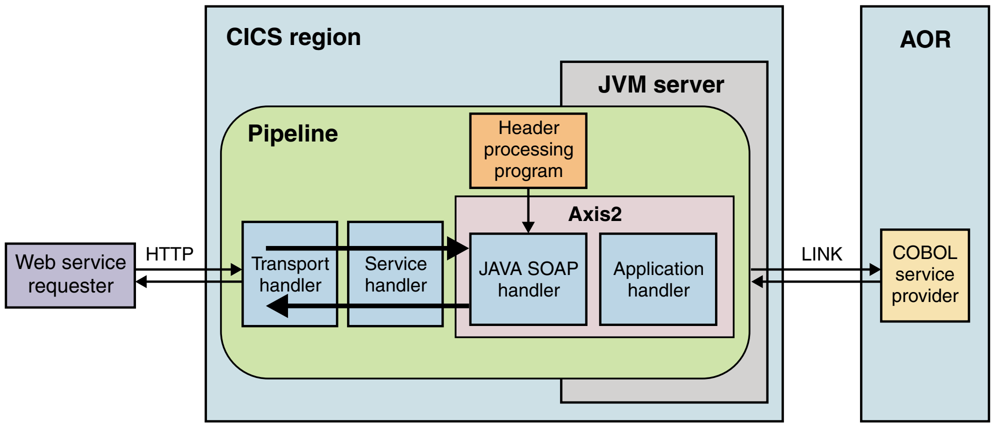
The following examples describe how you can use Java with web services.

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.



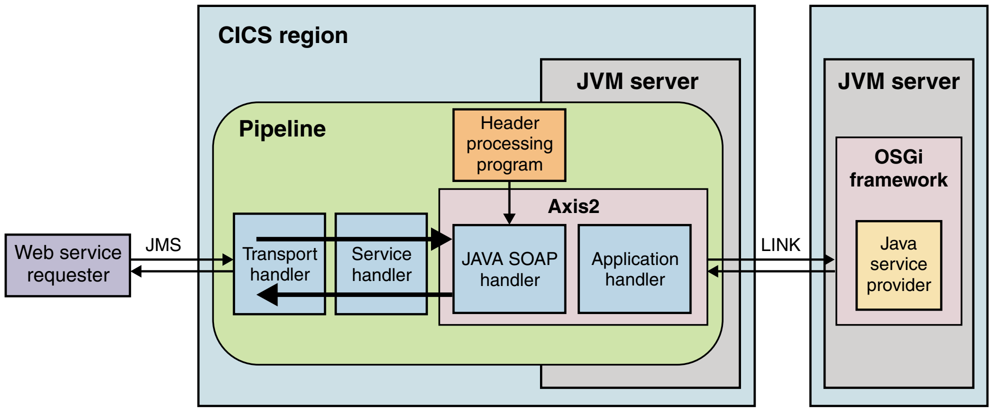
When you are planning your environment, ensure that you use a set of dedicated regions for your JVM servers. In this example, the COBOL application runs in an application-owning region (AOR) that is separate from the CICS region where the JVM server runs. You can use workload management to balance the 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 JZOS or J2C 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" on page 33.

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.

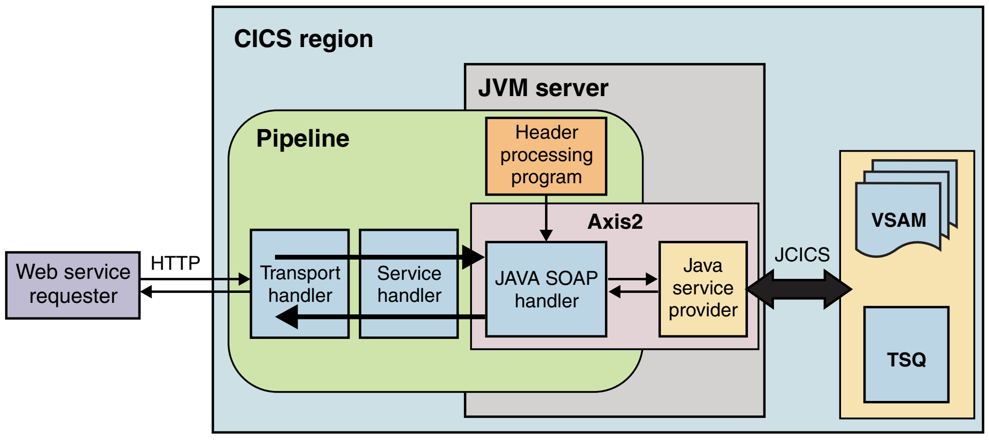
When you are planning your environment for this type of application, use a separate JVM server to run the application:

- 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.

Chapter 2. Developing Java applications for CICS

You can write Java application programs that use CICS services and run under CICS control. Using the CICS Explorer SDK, 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 CICS Transaction Gateway.

CICS provides tools and the runtime environment to support Java applications. The CICS Explorer SDK 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 and C.

The CICS Explorer SDK provides other features:

- You can configure a target environment to ensure that you use only the classes that are supported in a specific release of CICS. The SDK imports the correct libraries for your application development.
- You can package OSGi bundles, OSGi application projects (EBA files), web-enabled OSGi bundle projects (WAB files), and dynamic web projects (WAR files) in one or more CICS bundles for deployment. However, there is little benefit in packaging WAR files and OSGi bundles in the same CICS bundle, as the WAR files do not have access to the OSGi bundles. CICS provides a web container to run applications that include web components, such as servlets or JSP pages. The web container is built on the Liberty technology from WebSphere Application Server.
- You can package CICS bundles into an application project for deployment on a platform.

The SDK includes a set of samples to help you get started if you are new to developing Java applications for CICS.

“What you need to know about CICS” on page 24
CICS is a transaction processing subsystem that provides services for a user to run applications by request, at the same time as many other users are submitting requests to run the same applications, using the same files and programs. CICS manages the sharing of resources, integrity of data, and prioritization of execution, while maintaining fast response times.

“Developing applications using the CICS Explorer SDK” on page 29
The CICS Explorer Software Development Kit (SDK) provides an environment for developing and deploying Java applications to CICS, including support for OSGi and web projects.

“Controlling state in the JVM” on page 31
When you are designing and developing Java applications to run in CICS, ensure that your applications do not leave the JVM in an incorrect state or modify the state in an undesirable way. You use CICS services to help control the state of the JVM.

“Interacting with structured data from Java” on page 33
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 using a COMMAREA defined in a COBOL copybook, or read a record from a VSAM file where the data is defined using a C++ header file. You can use an importer to interact with these forms of structured data.

"Java development using JCICS" on page 34
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.

"Accessing data from Java applications" on page 65
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.

"Connectivity from Java applications in CICS" on page 65
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.

"Using JDBC and SQLJ to access DB2 data from Java programs" on page 66
Java programs that run in CICS can use several methods to access data held in a DB2 database.

"Developing Java web applications to run in Liberty JVM server" on page 72
Configure the Liberty JVM server to run a web container if you want to deploy Java™ web applications that use Java servlets and JSP pages.

What you need to know about CICS

CICS is a transaction processing subsystem that provides services for a user to run applications by request, at the same time as many other users are submitting requests to run the same applications, using the same files and programs. 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 25 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" on page 25
A transaction is a piece of processing initiated by a single request.

"CICS tasks" on page 25
A task is single instance of the execution of a transaction.

"CICS application programs" on page 26
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 services" on page 26
Java programs can access the following CICS services through the JCICS programming interface: Data management, communications, unit-of-work, program, and diagnostic services.

"Java runtime environment in CICS" on page 28
CICS provides the JVM server environment for running threadsafe Java applications. Applications that are not threadsafe cannot use a JVM server.

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 in Product overview and the Overview of CICS external interfaces in Product overview 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.

This 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 35.

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. (Thus 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 the CICS ExecutorService 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 a 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 can use JCICS objects only 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 38.

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 use JCICS in startup and termination classes.

Developing applications using the CICS Explorer SDK

The CICS Explorer Software Development Kit (SDK) provides an environment for developing and deploying Java applications to CICS, including support for OSGi and web projects.

About this task

You can use the SDK to create new applications, or repackage existing Java applications to comply with the OSGi specification. The OSGi Service Platform provides a mechanism for developing applications 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 control 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 the OSGi Alliance website.

You can also use the SDK to work with dynamic web projects and OSGi application projects that include Java servlets and JSP pages. 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.

You can use the SDK 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 has also been extended in later releases to support more features of CICS. To avoid using the wrong classes, the SDK provides a feature to set up a target platform. You can define which release of CICS you are developing for, and the SDK automatically hides the Java classes that you cannot use.

The SDK 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 remove the ".qualifier" from the end of the version number. The qualifier is not currently supported.
3. Develop your Java application using best practices. If you are new to developing Java applications for CICS, you can use the examples that are provided with the CICS Explorer SDK to get started. To use JCICS in a Java application, you must import the com.ibm.cics.server package.

4. Optional: 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 modify the target platform to give you access to the Liberty API bundles. For further details refer to “Setting up the development environment” on page 73.
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 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 a subset of CICS resources to the CICS Bundle project, such as PROGRAM, URIMAP, and TRANSACTION resources. These resources are dynamically installed and managed as part of the 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 applications for deployment in Developing applications.
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 CICS Explorer SDK.

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. For details, see “Deploying OSGi bundles in a JVM server” on page 117. You should also update the Target Platform, see “Updating the target platform.”

“Updating the target platform”
How to manually add 3rd party Java classes to the Target Platform of an Eclipse development environment.

Updating the target platform

How to manually add 3rd party Java classes to the Target Platform of an Eclipse development environment.

Before you begin

Ensure the JAR file containing the 3rd party Java classes is available as an OSGi plug-in and is copied to the local workstation. Ensure the CICS TS V5.2 or **CICS TS V5.2 with Liberty and PHP** Target Platform is already configured.

About this task

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 it is necessary to manually add the OSGi plug-in containing the 3rd party JARs to the Eclipse Target Platform. This will make the exported packages available to all applications that use this target platform.

Procedure

1. In Eclipse click **Windows** > **Preferences** > **Target Platform**.
2. Select the **CICS TS V5.2** item or **CICS TS V5.2 with Liberty and PHP** from the Target definitions.
3. Click **Edit** and then under the **Locations** tab, click **Add**. Browse to the directory where the 3rd party bundle JAR is located.
4. Click **Next** and the OSGi plug-in content will be displayed.
5. Click **Finish** and **Finish** again followed by **OK**.

Results

You have successfully updated the OSGi environment to include both the 3rd party OSGi bundle and the CICS OSGi bundle required for Java application development.

What to do next

Deploy the Java application into a CICS JVM server, and add the 3rd party JAR as an OSGi middleware bundle or to the Liberty shared bundle repository. For further details refer to topics "Updating OSGi middleware bundles" on page 129 and "Manually tailoring `server.xml`" on page 91.

Controlling state in the JVM

When you are designing and developing Java applications to run in CICS, ensure that your applications do not leave the JVM in an incorrect state or modify the state in an undesirable way. You use CICS services to help control the state of the JVM.

Protect the state of a JVM

If your application changes the state of the JVM, ensure that the application also resets to the original state. For example, an application might reset the default time zone, and do calculations based on this time zone. Other applications that use the same JVM use the new default time zone, which might not be appropriate.

Java applications run in the JVM server runtime environment, and are not isolated from other applications that might also be running in the same JVM under different threads. Any changes that an application makes to a JVM affects all other applications running in that JVM.

Do not use System.exit() methods in your applications. Using System.exit() methods causes both the JVM server and CICS to shut down and can affect the state of your applications.

Control static state in a JVM

Do not leave any unwanted state in a JVM. State is shared between all running applications in a JVM server.

An application must reinitialize its own static storage, if it depends on the state of a changeable class field. The values of static variables persist in the JVM for all application and system classes, including classes that might affect the application but are not used explicitly by the application and values used in static initializers.

In most cases, static variables are used to avoid reinitialization of storage, and allowing them to persist can improve performance. If the application requires that the value of these variables is reset, the application must reset the value itself. Try to identify and eliminate any changeable class fields and static initializers that have not been included deliberately as part of the application design.

Define a class field as private and final whenever possible. A native method can write to a final class field, and a non-private method can obtain the object referenced by the class field and can change the state of the object or array.

You can use the ability to pass on state to your advantage in designing your Java applications if you want information to persist from one program invocation to the next. Static state and object instances that are referenced through static state persist in the JVM, so it is permissible for applications to create persistent items that might be of use to future executions of the same application in the same JVM.

For example, an operation reads DB2 information to construct a complex data structure; this might be an expensive operation that you do not want to repeat more times than absolutely necessary. The complex data structure can be stored in application static storage and be accessible to later executions of the application in the same JVM, thus avoiding unnecessary initialization. If objects are anchored in static storage, that is, in the static class fields, they are never be candidates for garbage collection.

In a JVM server, static state persists for all applications until the JVM server is disabled by the system programmer. You can provide OSGi bundle activator classes to maintain the state of objects across restarts of the JVM server. These classes cannot contain JCICS calls.

Close DB2 connections, sockets, and other task lifetime system resources after use

Because Java applications run in the JVM server runtime environment, it is possible to have multiple connections to DB2 from different applications. Therefore, when a task has finished with DB2, it is best practice but not required to close the connection, because the connection is deleted when the task has completed.

If you start threads in an application to manage sockets using the java.net package, the application must manage the connections and close them. Sockets created using the java.net classes use the native sockets capability in the JVM, rather than the CICS sockets domain. CICS is not able to manage or monitor any communications that are performed using these sockets.

The same applies to any other task lifetime system resources used by the application, which must be released after use.

Test applications for possible threadsafe issues

The JVM server runtime environment does not support the use of non-threadsafe Java applications, so you must always write threadsafe Java applications. Ensure that objects maintain a valid state at all times, even when used concurrently by multiple threads. If a Java application works correctly on its first use in a given JVM, but does not behave correctly on subsequent uses, the problem is likely to be due to threadsafe issues.

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 using a COMMAREA defined in a COBOL copybook, or read a record from a VSAM file where the data is defined using a C++ header file. You can use an importer to interact with these forms of structured data.

Importing application data into Java using JZOS and J2C

CICS supports copybook importers so that you can use structured data from other programming languages in Java. Supported importers are provided by JZOS tools and by Rational. Rational tools use the Java EE Connector Architecture (JCA also known as J2C).

The importers map the data types contained in the source program so that your application can access individual fields in data structures. You can use the JZOS or Rational J2C tools to interact with data to produce a Java class, so that you can pass data between Java and other programs in CICS.

CICS supports Java artifacts from the following importers:
- Data binding beans from the J2C tools in Rational Application Developer (RAD) and Rational Developer for System z
- Records from the IBM JZOS Batch Toolkit for z/OS SDK

The IBM Redbooks® publication, Java Application Development for CICS uses an example application called the Heritage Trader application, which manipulates an existing COBOL application. Information is provided on the following topics:
- Instructions for installing JZOS and J2C
- Migrating the COBOL application to JCICS
- Creating a Java data binding class for J2C
- Generating a wrapper class with JZOS
- Example implementations for web, file, and DB2 access using the JCICS API

J2C requirements

You can create Java EE Connector artifacts that you can use to create enterprise applications. The RAD J2C wizard helps you create a class or set of classes that map to COBOL and other application program data structures.

You require RAD on a Windows or Linux workstation to use the Rational J2C importer.

JZOS requirements

The IBM JZOS Batch Toolkit for z/OS SDK is a set of tools that provide Java batch capabilities on z/OS. JZOS includes a launcher for running Java applications directly as batch jobs or started tasks, and a set of Java methods that make access to traditional z/OS data and key system services directly available from Java applications.

JZOS supports automatic generation of record classes from COBOL copybooks and Assembler DSECTs.

The JZOS download includes the *JZOS COBOL Record Generator User's Guide* and the *JZOS Assembler Record Generator User's Guide* in PDF format.

IBM Redbooks

Java Application Development for CICS

Java Connectors for CICS Featuring the Java EE Connector Architecture

Java Stand-alone Applications on z/OS Volume 2

J2C information

RAD: Connecting to enterprise information systems (EIS)

RAD: COBOL Importer overview

CICS Transaction Gateway Programming Guide

JZOS information

JZOS Java Launcher and Toolkit Overview

JZOS Batch Launcher and Toolkit function in IBM SDK for z/OS, Java Technology Edition Installation and User's Guide

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. The library is supplied in the `com.ibm.cics.server.jar` file with CICS and with the CICS Explorer SDK.

"The Java class library for CICS (JCICS)" on page 35
JCICS supports most of the functions of the **EXEC CICS** API commands.

"Data encoding" on page 39
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.

"JCICS API services and examples" on page 40
Many of the options and services available to non-Java programs through the **EXEC CICS** API are available to Java programs through JCICS. This information provides a mapping between the **EXEC CICS** API and JCICS, and provides excerpts of example Java code.

"Using JCICS" on page 63
You use the classes from the JCICS library like normal Java classes. Your applications declare a reference of the required type and a new instance of a class is created using the `new` operator.

"Java restrictions" on page 64
When you are developing Java applications, various restrictions apply otherwise problems might occur when the applications are running in CICS.

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 Class Reference.

"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.

"Library structure" on page 36
Each JCICS library component falls into one of four categories: Interfaces, Classes, Exceptions, or Errors.

"CICS resources" on page 36
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.

"Arguments for passing data" on page 36
You can pass data between programs using channels and containers, or by using a communication area (COMMAREA).

"Serializable classes" on page 37
A list of the JCICS serializable classes.

"Task.out and Task.err" on page 38
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`.

"Threads" on page 38
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.

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 44 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 the *CICS Resource Definition Guide*the *CICSPlex System Manager Concepts and Planning* manual.

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.record.IByteBuffer)`
: This method does a LINK using an object that implements the IByteBuffer interface of the Java Record Framework supplied with VisualAge for Java.

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

Related information:

"Channel and container examples" on page 45
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.

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

Related reference:

"Serialization services" on page 59
JCICS provides support for the CICS serialization services, which let you schedule the use of a resource by a task.

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 called runAsCICS() which you can use to submit a Runnable Java object for running 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.

Because this class is registered as an implementation of the Java ExecutorService interface, any application that requests an ExecutorService is only given the CICSExecutorService when it runs in a JVM server.

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

Restrictions

You must use the execute() method to create threads that can use JCICS. If you use the submit() method, the application gets a Java thread that cannot run JCICS.

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

Related reference:

"Threads and tasks example" on page 59
You can create a thread that starts a task in CICS by using the CICSExecutorService. If you use this service to create a thread, a CICS task is created that can use the JCICS API to access CICS services.

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

Many of the options and services available to non-Java programs through the **EXEC CICS** API are available to Java programs through JCICS. This information provides a mapping between the **EXEC CICS** API and JCICS, and provides excerpts of example Java code.

"CICS exception handling in Java programs" on page 42
CICS ABENDs and exceptions are integrated into the Java exception-handling architecture to handle problems that occur in CICS.

"CICS exception handling in Java Web applications" on page 43
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.

"Error handling and abnormal termination" on page 44
To initiate an ABEND from a Java program, you must invoke one of the Task.abend(), or Task.forceAbend() methods.

"APPC mapped conversations" on page 45
APPC unmapped conversation support is not available from the JCICS API.

"Basic Mapping Support (BMS)" on page 45
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.

"Channel and container examples" on page 45
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.

"Diagnostic services" on page 49
The JCICS application programming interface has support for these CICS trace and dump commands.

"Document services" on page 49
This section describes JCICS support for the commands in the DOCUMENT application programming interface.

"Environment services" on page 50
CICS environment services provide access to CICS data areas, parameters, and resource attributes that are relevant to an application program.

"File services" on page 52
JCICS provides classes and methods that map to the **EXEC CICS** API commands for each type of CICS file and index.

"HTTP and TCP/IP services" on page 55
Getters in classes `HttpHeader`, `NameValueData`, and `FormField` return HTTP headers, name and value pairs, and form field values for the appropriate API commands.

"Java Transaction API (JTA)" on page 56
The Java Transaction API (JTA) can be used to coordinate transactional updates to multiple resource managers.

"Program services" on page 57
JCICS supports the CICS program control commands; LINK, RETURN, and
| INVOKE APPLICATION.

"Scheduling services" on page 58
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.

"Serialization services" on page 59
JCICS provides support for the CICS serialization services, which let you schedule the use of a resource by a task.

"Storage services" on page 59
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.

"Threads and tasks example" on page 59
You can create a thread that starts a task in CICS by using the CICSExecutorService. If you use this service to create a thread, a CICS task is created that can use the JCICS API to access CICS services.

"Temporary storage queue services" on page 60
JCICS supports the CICS temporary storage commands; DELETEQ TS, READQ TS, and WRITEQ TS.

"Terminal services" on page 61
JCICS provides support for these CICS terminal services commands.

"Transient data queue services" on page 61
JCICS supports the CICS transient data commands, DELETEQ TD, READQ TD, and WRITEQ TD. All options are supported except the INTO option.

"Unit of work (UOW) services" on page 62
JCICS provides support for the CICS SYNCPOINT service.

"Web services example" on page 62
JCICS supports all the API commands that are available for working with web services in an application.

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, which extends java.lang.Error and is the base for AbendError and UnknownCicsError.
2. CicsRuntimeException, which 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, which extends java.lang.Exception and has the subclass:

   `CicsConditionException.`
   The base class for all CICS conditions.

   “CICS error-handling commands” on page 43
   CICS condition handling is integrated into the Java exception architecture as described above. The way that the equivalent “**`EXEC CICS`**” command is supported in Java is described below:

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

CICS error-handling commands:

CICS condition handling is integrated into the Java exception architecture as described above. The way that the equivalent "`EXEC CICS`" command is supported in Java is described below:

`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 naming 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 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 56.

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
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 Enhanced inter-program data transfer using channels in Developing applications. For information about tools that allow Java programs to access existing CICS application data, see "Interacting with structured data from Java" on page 33.

Table 3 lists the classes and methods that implement JCICS support for channels and containers.

Table 3. JCICS support for channels and containers

Methods	JCICS class	`EXEC CICS` Commands
containerIterator()	Channel	STARTBROWSE CONTAINER
createContainer()	Channel	
deleteContainer()	Channel	DELETE CONTAINER CHANNEL
getContainer()	Channel	
getName()	Channel	
delete()	Container	DELETE CONTAINER CHANNEL
get(), getLength()	Container	GET CONTAINER CHANNEL [NODATA]
getName()	Container	
put()	Container	PUT CONTAINER CHANNEL
getOwner()	ContainerIterator	
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" on page 47
To create a channel, use the createChannel() method of the Task class.

"Putting data into a container" on page 47
To put data into a Container object, use the Container.put() method.

"Passing a channel to another program or task" on page 47
To pass a channel on a program-link or transfer program control (XCTL) call, use the link() and xctl() methods of the Program class, respectively.

"Receiving the current channel" on page 48
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.

"Getting data from a container" on page 48
Use the Container.get() method to read the data in a container into a byte array.

"Browsing the current channel" on page 48
A JCICS program that is passed a channel can access all of the `Container` objects without receiving the channel explicitly.

"Channel and containers example" on page 48
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

Related concepts:

"Arguments for passing data" on page 36
You can pass data between programs using channels and containers, or by using a communication area (COMMAREA).

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's 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.

To put data into a Container object, use the Container.put() method. Data can be added to a container as a byte array or a string. For example:

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

Or :

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

Passing a channel to another program or task:

To pass a channel on a program-link or transfer program control (XCTL) call, use the link() and xctl() methods of the Program class, respectively.

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

```
programY.xctl(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.put("John Doe");

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

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

     // 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. 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

Methods	JCICS class	**EXEC CICS** Commands
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.

“ASSIGN” on page 51
The following support is provided for the **ASSIGN** API command options.

“INQUIRE SYSTEM” on page 52
Support is provided for the **INQUIRE SYSTEM** SPI options.

“INQUIRE TASK” on page 52
The following support is provided for the **INQUIRE TASK** API command options.

“INQUIRE TERMINAL and INQUIRE NETNAME” on page 52
The following support is provided for **INQUIRE TERMINAL** and **INQUIRE NETNAME** SPI options.

ADDRESS:

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

For complete information about the **EXEC CICS ADDRESS** command, see ADDRESS in Reference -> Application development.

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 36 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 contains information about the CICS command last executed. 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 in Reference -> Application development.

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 in Developing applications.

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.

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).

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

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.

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.

The first table shows the classes and methods for keyed files:

Table 4. 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**

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

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**

ESDS primary index class and method	RRDS primary index class and method	CICS File API command
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**
ESDS.setXRBA()		

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.

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
getDocument()	HttpResponse	WEB RETRIEVE

Methods	JCICS class	`EXEC CICS` Commands
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 49 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 Internet, TCP/IP, and HTTP concepts in Product overview. For more information on the JCICS Web classes see the *JCICS Class Reference*.

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.

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();
}
```

Note that, 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 can not 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.

Note that the Liberty default is to wait until the first UserTransaction is created before attempting to recover any indoubt transactions. CICS will always initiate recovery as soon as Liberty initialization is complete, as though `<transaction recoverOnStartup="true"/>` is specified in server.xml. If the JVM server is installed as disabled, recovery will run when it is set to enabled.

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 in Developing applications.

Table 5 lists the methods and JCICS classes that map to CICS program control commands.

Table 5. 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 32,500 bytes if the COMMAREA is to be passed between any two CICS servers (for any combination of product/version/release). This limit allows for the 32,500-byte 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

Related reference:

"Serializable classes" on page 37
A list of the JCICS serializable classes.

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

You can create a thread that starts a task in CICS by using the CICSExecutorService. If you use this service to create a thread, a CICS task is created that can use the JCICS API to access CICS services.

You have two options for creating a thread that starts a CICS task. If you want your code to be portable, you can use the ExecutorService in the OSGi framework. When the application is running in a JVM server, the OSGi framework automatically uses the CICS implementation to start a CICS task when a thread is created. If you are writing an application specifically for CICS, you can use the CICSExecutorService class directly using JCICS.

The following example shows an excerpt of a Java class that starts a thread to create a temporary storage queue and writes some data to the queue.

```
package com.ibm.cics.executor.test;

import com.ibm.cics.server.CICSExecutorService;
import com.ibm.cics.server.TSQ;
import com.ibm.cics.server.TSQType;

public class ExecutorTest
{
    public static void main(String[] args)
    {
        // Inline the new Runnable class
        class CICSJob implements Runnable
        {
            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();
                }
            }
        }

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

Related concepts:

"Threads" on page 38
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.

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 in Developing applications.

Table 6 lists the methods and JCICS classes that map to CICS temporary storage commands.

Table 6. 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.

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 in Developing applications.

Table 7 lists the methods and JCICS classes that map to CICS transient data commands.

Table 7. 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 8. 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 56.

Web services example

JCICS supports all the 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
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, you can use a tool such as JZOS to generate classes for you if you are working with structured data. For more information, see "Interacting with structured data from Java" on page 33. You can also use Java to generate and consume XML directly.

Using JCICS

You use the classes from the JCICS library like normal Java classes. Your applications declare a reference of the required type and a new instance of a class is created using the `new` operator.

About this task

You name CICS resources using the setName method to supply the name of the underlying CICS resource. After you create the resource, you can manipulate objects using standard Java constructs. You can 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.

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.

Do not end CICS Java programs by issuing a System.exit() call. When Java applications run in CICS, the public static void main() method is called through the use of another Java program called the Java wrapper. When you use the wrapper CICS initializes the environment for Java applications and, more importantly, cleans up any processes that are used during the life of the application. Terminating the JVM, even with a clean return code of 0, prevents this cleanup process from running, and might lead to data inconsistency. Using System.exit() when the application is running in a JVM server terminates the JVM server and quiesces CICS immediately.

Procedure

1. Write the main method. 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[]).
2. To create an object using JCICS, follow these steps:
   a. Declare a reference:
      ```
      TSQ tsq;
      ```
   b. Use the new operator to create an object:
      ```
      tsq = new TSQ()
      ```
   c. Use the setName method to give the object a name:
      ```
      tsq.setName("JCICSTSQ");
      ```
3. Use the object to interact with CICS.

Example

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 you are developing Java applications, various restrictions apply otherwise problems might occur when the applications are running in CICS.

The following restrictions apply for Java applications used in CICS:

- System.exit() method: this method cannot be used in Java applications otherwise the application abnormally ends, the JVM server shuts down, and CICS shuts down. Use a security policy to disable support for System.exit(). For related information see “Enabling a Java security manager” on page 162.

- JCICS API calls: these calls cannot be used in the activator classes of OSGi bundles.
- Start and stop methods used in bundle activators: these methods must return in a reasonable amount of time.

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 in Developing applications.
- 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.

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 28.

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.

Using JDBC and SQLJ to access DB2 data from Java programs

Java programs that run in CICS can use several methods to access data held in a DB2 database.

Java programs can:

- Use a JCICS LINK command to link to a CICS program that uses Structured Query Language (SQL) commands to access the data.
- Directly access the data by using the Java Data Base Connectivity (JDBC) or Structured Query Language for Java (SQLJ) application programming interfaces.

“Making JDBC and SQLJ work in the CICS DB2 environment” on page 67
When a Java application for CICS makes JDBC and SQLJ requests, the requests are processed by a JDBC driver supplied by DB2.

“Configuring a JVM server to support DB2” on page 67
A JVM server is the runtime environment for Java applications. You can configure the JVM server to support JDBC and SQLJ-based applications.

“Programming with JDBC and SQLJ in the CICS DB2 environment” on page 68
Java programs for CICS must adhere to the programming rules of the JDBC application programming interfaces (API), which might be more restrictive than the general CICS programming model.

“Acquiring a connection to a database” on page 68
Before executing SQL statements, a JDBC or SQLJ application must acquire a connection or connection context to a database. The application connects to a target data source using one of two Java classes.

“Closing a connection to a database” on page 70
When you close a connection to a database, JDBC and SQLJ resources are automatically released. Typically, a connection to a database is closed when the task ends.

“Committing a unit of work” on page 71
To commit a unit of work, your JDBC and SQLJ applications can issue JDBC

and SQLJ commit and rollback method calls. The DB2 JDBC driver converts these calls into a JCICS commit or a JCICS rollback call, resulting in a CICS syncpoint being taken.

"CICS abends during JDBC or SQLJ requests" on page 72
CICS abends issued during processing of an EXEC SQL request built by the DB2 supplied JDBC driver are not converted into Java Exceptions, and therefore are not catchable by a Java application for CICS. The CICS transaction will abend and rollback to the last syncpoint.

Making JDBC and SQLJ work in the CICS DB2 environment

When a Java application for CICS makes JDBC and SQLJ requests, the requests are processed by a JDBC driver supplied by DB2.

About this task

In a CICS environment, the DB2-supplied JDBC driver is link-edited with the CICS DB2 language interface (stub) DSNCLI. The driver converts the JDBC or SQLJ requests into their EXEC SQL equivalents. The converted requests from the DB2-supplied JDBC driver flow into the CICS DB2 attachment facility in exactly the same way as EXEC SQL requests from any other program (for example, a COBOL program). So there are no operational differences between Java programs for CICS DB2 and other programs for CICS DB2, and all customization and tuning options available using RDO apply to Java programs for CICS DB2.

Use the DB2 Universal JDBC driver that is distributed with DB2. To use a JDBC driver provided by DB2, CICS must be connected to a DB2 subsystem that supports the appropriate level of JDBC driver.

Full details of how to code and build Java applications that use the JDBC and SQLJ application programming interfaces can be found in the *DB2 for z/OS: Programming for Java* that applies to your version of DB2.

Configuring a JVM server to support DB2

A JVM server is the runtime environment for Java applications. You can configure the JVM server to support JDBC and SQLJ-based applications.

Before you begin

To use the JVM server with DB2, you must have the latest version of the IBM Data Server Driver for JDBC and SQLJ. For more information about what APARs are required, see CICS Transaction Server for z/OS V5.2 detailed system requirements. You must also add the DB2 SDSNLOD2 library to the CICS STEPLIB concatenation.

Procedure

Enable your applications to use the JDBC drivers that are supplied with DB2:

1. Set up a JVM server with the correct JDBC driver information. For a Liberty JVM server, see Configuring a Liberty JVM server for web applications in Configuring. For an OSGi JVM server, see Configuring a JVM server for OSGi applications in Configuring.
2. When you run the JDBC driver in a CICS environment, you might want to alter the system properties. You can set system properties that are related to the JDBC driver in the JVM profile for the JVM server. The DB2 environment

variable **DB2SQLJPROPERTIES**, which names a system properties file to be used by the JDBC driver, is not used. Most DB2 JDBC driver system properties are not used in a CICS environment. For a list of DB2 JDBC driver properties, and the properties that are not used or that have a different meaning in a CICS environment, see *DB2 for z/OS: Programming for Java*.

What to do next

If you want to use JDBC or SQLJ from a Java application in an OSGi JVM server with a Java 2 security policy mechanism active, see Enabling a Java security manager in Deploying.

Programming with JDBC and SQLJ in the CICS DB2 environment

Java programs for CICS must adhere to the programming rules of the JDBC application programming interfaces (API), which might be more restrictive than the general CICS programming model.

The IBM Data Server Driver for JDBC and SQLJ provides support for JDBC 4.0 and earlier versions.

| **Note:** SQLJ is not supported in a Liberty JVM server.

You can find more information about the JDBC APIs at the JDBC website. For information about developing Java applications that use the JDBC and SQLJ APIs, see the Information Management Software for z/OS Solutions Information Center.

Acquiring a connection to a database

Before executing SQL statements, a JDBC or SQLJ application must acquire a connection or connection context to a database. The application connects to a target data source using one of two Java classes.

About this task

The two Java classes are:

- DriverManager: This class connects an application to a database which is specified by a database URL.
- DataSource: This interface is preferred over DriverManager because it allows details about the underlying database to be transparent to your application making applications more portable. The DataSource implementation is created externally to the Java program and obtained with a Java Naming and Directory Interface (JNDI) DataSource name lookup. The DataSource interface is only available in a Liberty JVM server.

In a CICS DB2 environment, you do not need to specify a userid and password, the existing CICS DB2 security procedures are used instead. The IBM Data Server Driver for JDBC and SQLJ provides support for JDBC 4 and earlier.

There are JDBC examples for both OSGi and Liberty JVM servers supplied in the CICS Explorer SDK. For more information see, Getting started with the servlet examples in Developing applications.

For more information on using the JDBC DriverManager and DataSource interfaces to acquire a connection, and sample code that you can use in your application, see the DB2 for z/OS: Programming for Java which is appropriate for your version of DB2.

"Acquiring a DriverManager connection to a database"
Using the DriverManager class, the connection to the database is identified by a database Uniform Resource Locator (URL) that is provided to the JDBC driver.

"Acquiring a DataSource connection to a database" on page 70
Using the DataSource interface, the connection to the database is obtained with a Java Naming and Directory Interface (JNDI) DataSource name lookup. JDBC DataSource implementation is provided by the CICS JDBC.

"How many connections can you have?" on page 70
A Java application for CICS can have at most one JDBC connection or SQLJ connection context open at a time.

Acquiring a DriverManager connection to a database

Using the DriverManager class, the connection to the database is identified by a database Uniform Resource Locator (URL) that is provided to the JDBC driver.

About this task

A `DriverManager` connection is configured for an OSGi JVM server by specifying the DB2 JDBC jars and native DLLs in the JVM profile `OSGI_BUNDLES` and `LIBPATH_SUFFIX` JVM profile.

In a Liberty JVM server, the DriverManager configuration is provided by the CICS JDBC Liberty feature.

The JDBC driver recognizes two types of URL:

Default URL
A default URL does not include the location name of a DB2 subsystem. A default URL for DB2 for z/OS can be specified in one of two formats:

```
jdbc:db2os390sqlj:
```
or
```
jdbc:default:connection
```

When a default URL is specified, the application is given a connection to the local DB2 to which CICS is connected. If your installation uses DB2 data sharing, you can access all the data in your sysplex from the local DB2.

Explicit URL
An explicit URL includes the location name of a DB2 subsystem. The basic structure of an explicit URL for DB2 for z/OS is:

```
jdbc:db2os390:<location-name>
```
or
```
jdbc:db2os390sqlj:<location-name>
```

Typically, the location name is the name of the local DB2 to which CICS is connected. However, you can specify the name of a remote DB2 to access. In this case, CICS uses the local DB2 as a pass-through, and uses DB2 Distributed Data Facilities to access the remote DB2.

It is advisable to use a default URL in a CICS environment. The use of an explicit URL might cause specific actions when a connection closes that could be inconvenient when multiple programs are used in the same application suite. Also, when a default URL is used, the behavior of the connection is not affected by the JDBC driver version.

To acquire a connection, your Java application needs to invoke the `getConnection()` method with the URL. For example:

```
Connection connection = DriverManager.getConnection("jdbc:default:connection");
```

Acquiring a DataSource connection to a database

Using the DataSource interface, the connection to the database is obtained with a Java Naming and Directory Interface (JNDI) DataSource name lookup. JDBC DataSource implementation is provided by the CICS JDBC.

About this task

A DataSource connection is configured in a Liberty JVM server with the CICS JDBC Liberty feature.

After the DataSource is configured, the application can use the use the value of the `jndiName` specified in the `cicsts_dataSource` element in the Liberty server configuration to obtain an instance of that `DataSource` class from the JNDI naming service. The `getConnection()` method can then be called on that `DataSource` object to get a connection. For example:

```
Context context = new InitialContext();
DataSource dataSource = (DataSource) context.lookup("jdbc/defaultCICSDataSource");

Connection connection = dataSource.getConnection();
```

How many connections can you have?

A Java application for CICS can have at most one JDBC connection or SQLJ connection context open at a time.

Although JDBC allows an application to have multiple connections at the same time, CICS does not permit this. However, an application can close an existing connection and open a connection to a new DB2 location.

An application that has an open connection should close the connection before linking to another application that wants to use JDBC or SQLJ. For Java programs that are part of an application suite, you need to consider the implications of closing the connection, because if you are using an explicit URL, closing the connection can cause a syncpoint to be taken. If you are using a default URL, a syncpoint does not have to be taken when the connection is closed. See “Committing a unit of work” on page 71 for more information about this.

Closing a connection to a database

When you close a connection to a database, JDBC and SQLJ resources are automatically released. Typically, a connection to a database is closed when the task ends.

For performance reasons, an application might leave the JDBC connection open for a subsequent user of the same JVM to use. If the JDBC connection is left open, the application must ensure that JDBC resources are not leaked over time.

If the application leaves the JDBC or SQLJ connection open, the application must do the following:

- Ensure that JDBC and SQLJ resources are released.

- Recover following a DB2 SIGNON for the underlying DB2 connection.
- Recycle the cached connection if it become invalid; for example, StaleConnection, SQLCODE=4499.

If connections are cached, you are recommended to include logic in your application that recycles the connection after a given number of transactions. Recycling cached connections protects against resource leakage.

Committing a unit of work

To commit a unit of work, your JDBC and SQLJ applications can issue JDBC and SQLJ commit and rollback method calls. The DB2 JDBC driver converts these calls into a JCICS commit or a JCICS rollback call, resulting in a CICS syncpoint being taken.

About this task

A JDBC or SQLJ commit results in the whole CICS unit of work being committed, not just the updates made to DB2. CICS does not support committing work done using a JDBC connection independently of the rest of the CICS unit of work.

A JDBC or SQLJ application can also issue JCICS commit or rollback directly, and this has the same result as issuing a JDBC or SQLJ commit or rollback method call. The whole unit of work is committed or rolled back together, both DB2 updates and updates to CICS controlled resources.

When you are working with JDBC connections, there are some circumstances in which you cannot avoid a syncpoint being taken, and the unit of work being committed, when the connection to the DB2 database is closed. This applies in either of the following circumstances:
- You have used the autocommit property of a JDBC connection.
- You have acquired the DriverManager connection using an explicit URL.

For a stand-alone application, these rules do not cause a problem, as CICS ensures that an end of task syncpoint is taken in addition to any syncpoint that is taken when the connection is closed. However, the JDBC and SQLJ application programming interfaces do not support the concept of multiple application programs for each unit of work. If you have a number of programs that make up an application, one program might access DB2, then call another program that also accesses DB2, in the course of a single unit of work. If you want these programs to be Java programs that use JDBC or SQLJ, you need to ensure that the unit of work is not committed when the connection to DB2 is closed, or else the application will not operate as planned. You should be particularly aware of this requirement if you are replacing programs in an existing application with Java programs that use JDBC or SQLJ, and you want to share the same CICS-DB2 thread between the programs. To address this issue, use a DriverManager with default URL or a DataSource connection.

“Autocommit” on page 72
JDBC applications can use the autocommit property of a JDBC connection. The autocommit property causes a commit after each update to DB2. This commit is a CICS commit, and results in the whole unit of work being committed.

“Syncpoint issues for DriverManager with explicit and default URLs” on page 72
When a Java application for CICS that uses JDBC or SQLJ acquires a connection

using an explicit URL, it operates in an environment similar to that of a DPL server program linked to with the SYNCONRETURN attribute.

Autocommit

JDBC applications can use the autocommit property of a JDBC connection. The autocommit property causes a commit after each update to DB2. This commit is a CICS commit, and results in the whole unit of work being committed.

Using the autocommit property also causes a commit to be taken when a connection is closed, for DriverManager connections that are obtained using an explicit URL or a default URL and type-2 DataSource connections.

The use of autocommit in a CICS environment is not recommended, and for this reason the DB2 JDBC driver sets a default of `autocommit(false)` when it runs in a CICS environment. In non-CICS environments the default is `autocommit(true)`.

Syncpoint issues for DriverManager with explicit and default URLs

When a Java application for CICS that uses JDBC or SQLJ acquires a connection using an explicit URL, it operates in an environment similar to that of a DPL server program linked to with the SYNCONRETURN attribute.

When an application program that uses JDBC or SQLJ closes the explicit URL connection, if CICS is using the IBM Data Server Driver for JDBC and SQLJ, no implicit syncpoint is taken.

However, the close of an explicit URL connection is only successful when on a unit of work boundary. The application must therefore take a syncpoint, by issuing a JDBC or SQLJ commit method call or a JCICS commit, before closing the connection. (The application could use autocommit(true) to ensure that a syncpoint is taken, but the use of this property is discouraged in the CICS environment.) When the application program closes an explicit URL connection, that is the end of the unit of work.

You can overcome this restriction by acquiring the connection using a **default** URL instead of an explicit URL, or by using a data source that provides a default URL connection (see "Acquiring a connection to a database" on page 68). When a default URL is used, the Java application does not have to close the connection on a unit of work boundary, and no syncpoint is taken when the connection is closed (provided that autocommit(true) has not been specified).

It is recommended that you always use default URL connections in a CICS environment.

CICS abends during JDBC or SQLJ requests

CICS abends issued during processing of an EXEC SQL request built by the DB2 supplied JDBC driver are not converted into Java Exceptions, and therefore are not catchable by a Java application for CICS. The CICS transaction will abend and rollback to the last syncpoint.

Developing Java web applications to run in Liberty JVM server

Configure the Liberty JVM server to run a web container if you want to deploy Java™ web applications that use Java servlets and JSP pages.

To find out how to download and install the CICS Explorer SDK, see Installing the CICS Explorer(r) SDK.

"Setting up the development environment"
To develop web applications to run in the Liberty JVM server, you must install the servlet and JSP feature that is supplied with the CICS Explorer SDK.

"Developing servlet and JSP applications" on page 74
To provide modern interfaces to CICS applications, you can develop a presentation layer that uses Java servlet and JSP technologies. Eclipse-based web development tools provide the development platform to create these applications, and the CICS Explorer SDK provides the support to build, package, and deploy them to run in CICS.

"Migrating Java web applications to run in Liberty JVM server" on page 77
If you have a Java web application running in a Liberty profile instance that accesses CICS over a network, you can run the application in a Liberty JVM server to optimize performance.

Setting up the development environment

To develop web applications to run in the Liberty JVM server, you must install the servlet and JSP feature that is supplied with the CICS Explorer SDK.

About this task

With the developer tools for the Liberty profile, you can develop servlet and JSP applications that can run in an instance of a Liberty profile server. The Liberty profile is an instance of an application server that is supplied with WebSphere Application Server to run simple web applications. It is quick to configure, deploy, and start.

You can also run servlets and JSP pages in a CICS environment, by installing the web application in a Liberty JVM server. The CICS Explorer SDK provides the tools to develop and deploy Java applications for CICS. You can use these tools with the developer tools to deploy Java applications with a web interface to CICS.

Set the target platform to CICS TS V5.x Runtime with Liberty or compilation errors can occur when you create examples in your workplace. See Creating the servlet examples for more information.

Procedure

1. If you already have the CICS Explorer SDK installed but did not install the servlet and JSP feature, you can install the feature from the CICS Explorer SDK download archive.
   a. In the Eclipse IDE, click **Help** > **Install New Software**.

      Note: Make sure that the option **Contact all update sites during install to find required software** is selected. The Liberty profile feature has dependencies on other Eclipse components, so you must ensure the update manager downloads and installs those dependencies.
   b. Select the CICS Explorer SDK download file from the list. Click **Add**. In the "Add site" dialog box, click **Archive**. The
   c. Browse to the CICS Explorer SDK download file and click **Open**.
   d. Expand **IBM CICS Explorer**.
   e. Select the check box next to **IBM CICS SDK for Servlet and JSP support v5.1** and click **Next**.

f. Read and accept the license information, and click **Finish** to install the feature. Any dependencies that are not already installed in your Eclipse IDE are also installed.

2. Optional: If you intend to deploy EBAs, you must install the WebSphere Application Server Developer Tools for Eclipse. You can also use the Developer Tools to test your web applications locally. For more information about downloading and installing the Developer Tools, see Installing WebSphere Application Server Developer Tools for Eclipse.

Results

After you restart your Eclipse IDE, you have the tools to develop, deploy, and test web applications that are suitable for running in a Liberty JVM server.

What to do next

You can start developing web applications to run in the Liberty JVM server. You can use the examples that are provided by the CICS Explorer SDK to get started. For more information about getting started, see topics/gettingstarted_liberty.dita.

Developing servlet and JSP applications

To provide modern interfaces to CICS applications, you can develop a presentation layer that uses Java servlet and JSP technologies. Eclipse-based web development tools provide the development platform to create these applications, and the CICS Explorer SDK provides the support to build, package, and deploy them to run in CICS.

About this task

There are two types of project containing servlets and JSPs that can be deployed on a Liberty server, Dynamic Web Projects and OSGi Bundle Projects.

“Creating a dynamic web project”
To develop a web presentation layer for your Java application, you can create a dynamic web project.

“Creating an OSGi application project” on page 76
An OSGi application project (EBA) groups together a set of bundles. The application can consist of different bundle types such as web enabled bundles and OSGi bundles.

Related tasks:

topics/gettingstarted_liberty.dita
The CICS Explorer SDK contains examples to help you start developing servlets and JSP pages that can run in a Liberty JVM server in CICS.

“Migrating Java web applications to run in Liberty JVM server” on page 77
If you have a Java web application running in a Liberty profile instance that accesses CICS over a network, you can run the application in a Liberty JVM server to optimize performance.

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 the web development tools are installed in your Eclipse IDE. For more information, see "Setting up the development environment" on page 73.

About this task

The CICS Explorer SDK help provides full details on how you can complete each of the following steps to develop and package web applications.

Procedure

1. Create a dynamic web project for your application. You must update your build path to add the Liberty libraries. Although you can create other types of web project in Eclipse, CICS supports only OSGi bundle projects and dynamic web projects.
   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 Liberty JVM server libraries.
   d. Click **Next**, select the CICS version and click **Finish** to complete adding the library.
   e. Click **OK** to save your changes.
2. Develop your web application. You can use the JCICS API to access CICS services and connect to DB2. The CICS Explorer SDK includes examples of web components and OSGi bundles that use JCICS and DB2.
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 CICS Explorer SDK includes a template for this file that contains the correct information for CICS. See "Authenticating users in a Liberty JVM server" on page 157 for further information. CICS security uses basic authentication to check the user ID and password in the application request. You can use Liberty security instead, but you must provide your own security roles and basic user registry. Warning: 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.
4. Create one or more CICS bundle projects to package your application. You can add references to OSGi application projects, dynamic web projects, and OSGi bundle projects, and 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 bundle types such as web enabled bundles and OSGi bundles.

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 73.

About this task

The CICS Explorer SDK help provides full details on how you can complete each of the following steps to develop and package web applications.

Procedure

1. Set up a target platform for your Java development, using the **CICS TS V5.2 with Liberty and PHP** 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. The target platform effectively makes the libraries available, so you must include the appropriate Import statements in the bundle. Although you can create other types of web project in Eclipse, CICS supports only OSGi bundle projects and dynamic web projects. 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 as an OSGi Application Project (an Enterprise Bundle Archive, or EBA file), or to use Java code in other OSGi bundles. 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.
3. Develop your web application. You can use the JCICS API to access CICS services and connect to DB2. The CICS Explorer SDK 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, include an Import-Package header for `com.ibm.db2.jcc` in the bundle manifest.
4. 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 CICS Explorer SDK includes a template for this file that contains the correct information for CICS. See “Authenticating users in a Liberty JVM server” on page 157 for further information. CICS security uses basic authentication to check the user ID and password in the application request. You can use Liberty security instead, but you must provide your own security roles and basic user registry. Warning: 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.
5. You can create an OSGi application project that references your OSGi bundles.
6. Create one or more CICS bundle projects to package your application. You can add references to OSGi application projects, dynamic web projects, and OSGi bundle projects, and 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 the Liberty deployment model by exporting the application an EBA file and deploying it to the dropins directory of a running Liberty JVM server.

Migrating Java web applications to run in Liberty JVM server

If you have a Java web application running in a Liberty profile 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 the Liberty profile. All of the core web container API (such as JSP and servlet) are supported along with many of the common Java EE API and capabilities. For a list of supported features, see topics/liberty_features.dita.

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 userid. To better integrate your security solution with CICS, allowing your CICS tasks to run under the same userid as determined by Liberty, see "Authorizing users to run applications in a Liberty JVM server" on page 159.

If your application uses RequestDispatcher.forward() calls, security applies only to the first servlet that is called by the client. Any requests forwarded to another servlet run under the same task and transaction ID in CICS, and no further security checking applies.

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 39.
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:

   ```
   <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) or EBA (OSGi application project) 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.

Related tasks:

“Developing servlet and JSP applications” on page 74
To provide modern interfaces to CICS applications, you can develop a presentation layer that uses Java servlet and JSP technologies. Eclipse-based web development tools provide the development platform to create these applications, and the CICS Explorer SDK provides the support to build, package, and deploy them to run in CICS.

Chapter 3. 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 used by the CICS region. For more information, see “Setting the location for the JVM profiles” on page 80.
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 81.
3. Give your CICS region permission to access the resources 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 81.
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 84.
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” on page 162.

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 Chapter 4, “Deploying applications to a JVM server,” on page 117.

“Setting the location for the JVM profiles” on page 80
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.

"Setting the memory limits for Java" on page 81
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.

"Giving CICS regions access to z/OS UNIX directories and files" on page 81
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.

"Setting up a JVM server" on page 84
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.

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/cicsts52/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/cicsts52/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." 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."

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" on page 143
   - 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 property	Default directories	Permission	Description
JAVA_HOME	`/usr/lpp/java/J7.0_64/bin`	read and execute	IBM 64-bit SDK for z/OS, Java Technology Edition directories
USSHOME	`/usr/lpp/cicsts/cicsts52`	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.
WLP_USER_DIR	`WORK_DIR/APPLID/JVMSERVER/wlp/usr/`	read and execute	Specifies the directory that contains the configuration files for the Liberty JVM server.
WLP_OUTPUT_DIR	`WLP_USER_DIR/servers/server_name`	read, write, and execute	Specifies the output directory for the Liberty JVM server.

Note: 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.

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## CICSTS52     8192 Mar 15  2008 .
drwx------   4 CICSHT## CICSTS52     8192 Jul  4 16:14 ..
-rw-------   1 CICSHT## CICSTS52     2976 Dec  5  2010 Snap0001.trc
-rw-r--r--   1 CICSHT## CICSTS52     1626 Jul 16 11:15 dfhjvmerr
-rw-r--r--   1 CICSHT## CICSTS52        0 Mar 15  2010 dfhjvmin
-rw-r--r--   1 CICSHT## CICSTS52      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/cicsts52` directory, use the following command:

   ```
   chgrp -R GID /usr/lpp/cicsts/cicsts52
   ```

   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 JSP pages and servlets, configure the JVM server with the DFHWLP.jvmprofile. This profile contains the options to run a web container that is based on the Liberty profile 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 User Guide*.

“Configuring a JVM server for an OSGi application” on page 85
Configure the JVM server to run an OSGi framework if you want to deploy Java applications that are packaged in OSGi bundles.

"Configuring a Liberty JVM server for web applications" on page 87
Configure the Liberty JVM server to run a web container if you want to deploy Java web applications that use Java servlets and JSP pages. The web container is based on the Liberty profile technology.

"Configuring a JVM server for Axis2" on page 97
Configure the JVM server to run Axis2 if you want to run Java web services or process SOAP requests in a pipeline.

"Configuring a JVM server for a CICS Security Token Service" on page 99
Configure the JVM server to run a CICS Security Token Service if you want to validate and process SAML tokens.

"JVM profile validation and properties" on page 100
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.

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. For more information, see Security for Java applications.

Configuring a JVM server for an OSGi application

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 100. Follow the coding rules, including those for the profile name, in "Rules for coding JVM profiles" on page 101.
   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 80.
   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. See Chapter 8, "Troubleshooting Java applications," on page 165 for help.

What to do next

- Install OSGi bundles for the application in the OSGi framework of the JVM server, as described in "Deploying OSGi bundles in a JVM server" on page 117.
- Specify any directories that contain native C dynamic link library (DLL) files, such as DB2 or WebSphere 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 WebSphere 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.

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 for web applications

Configure the Liberty JVM server to run a web container if you want to deploy Java web applications that use Java servlets and JSP pages. The web container is based on the Liberty profile technology.

About this task

The Liberty profile technology is installed with CICS to run as a web container in a JVM server. The Liberty JVM server supports a subset of the features that are available in the Liberty profile.

You have two ways of configuring a Liberty JVM server:

- Autoconfiguring. CICS automatically creates and updates the configuration file for the Liberty profile, `server.xml`, from templates that are supplied in the CICS installation directory. Auto-configuring gets you started quickly with a minimal set of configuration values in the Liberty profile; you might find it useful in a

development environment. To enable auto-configuring, 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 in a production environment.

| You can define the JVM server 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 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, `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 100. Follow the coding rules, including those for the profile name, in "Rules for coding JVM profiles" on page 101.
   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 80.
   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 **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.
   c. Save your changes to the JVM profile. The JVM profile must be saved in EBCDIC file encoding on UNIX System Services and the file suffix must be `.jvmprofile`.
3. Create the Liberty profile server configuration. When using autoconfigure this is performed at JVM server setup. To enable Liberty JVM server autoconfiguration you should set the following JVM system properties in the JVM profile.
   a. Enable autoconfigure by setting **-Dcom.ibm.cics.jvmserver.wlp.autoconfigure=true**.
   b. Set the HTTP server listener port using **-Dcom.ibm.cics.jvmserver.wlp.server.http.port**.

   c. Set the host that the Liberty HTTP server will listen on using **-Dcom.ibm.cics.jvmserver.wlp.server.host**.
   d. Set the Liberty profile server name using **-Dcom.ibm.cics.jvmserver.wlp.server.name**.

   Alternatively you can configure the Liberty profile directory structure and the `server.xml` configuration file manually. This is the default setting, and is appropriate in a production environment when the configuration files need to be carefully controlled. For further details see "Creating a configuration directory structure" on page 90 and Manually tailoring server.xml. You should use autoconfigure if you are defining the JVM server in a CICS bundle, as the `server.xml` configuration file can not be included with the JVM profile in a CICS bundle.
4. 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 and initializes the Liberty server, and the Liberty HTTP listener starts running. 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 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 the Liberty profile successfully started within your JVM server, consult the `message.log` file in the WLP_USER_DIR output directory on zFS.

Note: Enabling the JVM server starts the Liberty profile server and disabling the JVM server stops the Liberty profile server. Do not use the Liberty profile bin/server script to start or stop a Liberty profile server that is running in a JVM server.

What to do next

- Specify any directories that contain native C dynamic link library (DLL) files, such as WebSphere MQ. Middleware and tools that are supplied by IBM or by vendors might require DLL files to be added to the library path.
- Override the environment variables for the Liberty JVM server by resetting the options in the JVM profile that begin with `WLP`.
- If you want to run multiple JVM servers in a CICS region, add the following to the JVM profile of each additional Liberty JVM server: WLP_ZOS_PLATFORM=FALSE. Only one 'security enabled' Liberty JVM server is permitted per CICS region. This is because CICS uses the z/OS platform extensions of Liberty for authentication and authorization. However, these z/OS platform aspects of Liberty are restricted to a single Liberty server within an address space. By default, Liberty JVM servers start the z/OS platform extensions, however only the first Liberty JVM server within a CICS region enables with security successfully.
- Add support for security. See "Configuring security for a Liberty JVM server" on page 154.
- Install the web applications (WAR files or WAB and EBA files), as described in "Deploying a web application in a CICS bundle to a Liberty JVM server" on page 119.

"JVM profile example"
Example JVM profile for Liberty server.

"Creating a configuration directory structure"
Creating a configuration directory structure in zFS for the JVM server.

"Manually tailoring `server.xml`" on page 91
If you are not using automatic configuration, make the following changes to your `server.xml` file to tailor it to your environment.

"Creating a default CICS DB2 JDBC type 2 driver DataSource for Liberty" on page 93
Create a default CICS DB2 JDBC type 2 driver DataSource using the autoconfigure property.

"Manually configuring a CICS DB2 JDBC type 2 driver DataSource for Liberty" on page 94
Liberty profile users in CICS can use JDBC type 2 driver DataSource definitions to access DB2 databases from Java applications.

"Manually configuring a DB2 JDBC type 4 driver DataSource for Liberty" on page 95
Liberty profile users in CICS can use Liberty supplied DataSource definitions to access DB2 databases using JDBC type 4 from Java applications.

Related concepts:

Configuring security for a Liberty JVM server in Securing

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=/usr/lpp/java/J7.0_64
WORK_DIR=.
#***********************************************************************
# JVM server parameters
#
OSGI_FRAMEWORK_TIMEOUT=60
#***********************************************************************
# Liberty JVM server
#
-Dcom.ibm.ws.logging.console.log.level=OFF
-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-8859-1WLP_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
```

Creating a configuration directory structure

Creating a configuration directory structure in zFS for the JVM server.

Procedure

1. Create the configuration directory structure in zFS for the JVM server. The JVM server expects the configuration files to be in the `WLP_USER_DIR/servers/server_name` directory in the working directory, where *WLP_USER_DIR* is the value of the `WLP_USER_DIR` option and *server_name* is the value of the **`-Dcom.ibm.cics.jvmserver.wlp.server.name`** property.
2. Create the Liberty profile 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/config/cics/server.xml` in the installation directory of the Liberty profile. This file must be saved in the file encoding of ISO-8859-1 or UTF-8.
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`." If you want to use security, see Configuring security for a Liberty JVM server.

Manually tailoring `server.xml`

If you are not using automatic configuration, make the following changes to your `server.xml` file to tailor it to your environment.

Configuring 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 is not in use elsewhere, for example by a TCPIPSERVICE in CICS.

HTTPS is available only if SSL is configured (see "Configuring security for a Liberty JVM server" on page 154).

For more information, see the WebSphere Application Server Liberty Profile documentation.

Adding features

Add the following features in the <featureManager> list of features.

- The CICS feature `cicsts:core-1.0` installs the CICS system OSGi bundles into the Liberty framework. This feature is required for the JVM server to start.
- The CICS security feature `cicsts:security-1.0` 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.
- The `jsp-2.2` feature enables support for servlet and JavaServer Pages (JSP) applications. This feature is required by Dynamic Web Projects (WAR files) and OSGi Application Projects containing OSGi Bundle Projects with Web Support that are installed as CICS bundles.
- The `wab-1.0` feature enables support for Web Archive Bundles (WABs) that are inside enterprise bundles (EBAs). This feature is required by OSGi Application Projects containing OSGi Bundle Projects with Web Support that are installed as CICS bundles.

- The `cicsts:jdbc-1.0` 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-2.2</feature>
    <feature>wab-1.0</feature>
    <feature>cicsts:jdbc-1.0</feature>
</featureManager>
```

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 library

Share common JAR files between web applications by placing them in a directory and referring to that directory in a global library definition.

```
<library id="global">
<fileset dir="directory_path" include="*.jar"/>
</library>
```

The global libraries cannot be used by OSGi applications in an EBA, which must use a bundle repository.

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"/>.
```

JTA transaction log

When it uses the Java Transaction API (JTA), 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/"/>
```

Related tasks:

"Creating a configuration directory structure" on page 90
Creating a configuration directory structure in zFS for the JVM server.

Creating a default CICS DB2 JDBC type 2 driver DataSource for Liberty

Create a default CICS DB2 JDBC type 2 driver DataSource using the autoconfigure property.

Before you begin

You must configure your CICS region to connect to DB2. For more information, see Defining the CICS DB2 connection in Configuring.

About this task

You can create a CICS DB2 JDBC type 2 driver DataSource resource in the Liberty server.xml using the JVM profile auto-configuration property. The default configuration can be used with the CICS Explorer SDK CICS Java DataBase Connectivity (JDBC) Servlet and JSP example. The JNDI name is `jdbc/defaultCICSDataSource`.

Procedure

1. Enable autoconfigure by setting **`-Dcom.ibm.cics.jvmserver.wlp.autoconfigure=true`** in the JVM profile.
2. Enable autoconfigure of the default CICS DB2 JDBC type 2 driver DataSource by setting the **`-Dcom.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 default CICS DB2 JDBC type 2 driver DataSource is added to the Liberty server configuration file, `server.xml`.

```
<cicsts_dataSource id="defaultCICSDataSource" jndiName="jdbc/defaultCICSDataSource"/>
<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>
```

Related concepts:

Configuring security for a Liberty JVM server in Securing

Related tasks:

"Manually configuring a DB2 JDBC type 4 driver DataSource for Liberty" on page 95
Liberty profile users in CICS can use Liberty supplied DataSource definitions to access DB2 databases using JDBC type 4 from Java applications.

Manually configuring a CICS DB2 JDBC type 2 driver DataSource for Liberty

Liberty profile users in CICS can use JDBC type 2 driver DataSource definitions to access DB2 databases from Java applications.

Before you begin

You must configure your CICS region to connect to DB2. For more information, see Defining the CICS DB2 connection in Configuring.

About this task

This task explains how to define the elements that are required in the server.xml configuration file to enable JDBC type 2 driver connections. This allows the establishment of connections to local DB2 databases.

Note:

When **`-Dcom.ibm.cics.jvmserver.wlp.autoconfigure`** is set to true and **`-Dcom.ibm.cics.jvmserver.wlp.jdbc.driver.location`** is set in the JVM profile, then when the JVM server is enabled, the existing example JDBC configuration in the `server.xml` file is replaced with the default configuration and any user updates are lost.

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 will enable support for access to JDBC type 2 driver datasources either using **`java.sql.DriverManager`** or via **`javax.sql.DataSource`**. 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 in the server.xml file is used and the others are ignored.

3. If just **`java.sql.DriverManager`** support is required, then the preceding steps are sufficient. In order to access DB2 connections via DataSource definitions then a **`cicsts_dataSource`** element is required for each DataSource for which access is to be defined. A **`cicsts_dataSource`** specifies a jndiName attribute, this defines the JNDI name that is a reference used by your application program when establishing a connection to that DataSource. A definition might look like this:

   ```
   <cicsts_dataSource id="defaultCICSDataSource" jndiName="jdbc/defaultCICSDataSource"/>
   ```

   Note: The DataSource class used is **`com.ibm.db2.jcc.DB2SimpleDataSource`** that implements **`javax.sql.DataSource`**.

4. Optional: You can set properties for the **cicsts_dataSource** using a **properties.db2.jcc** element. The example below shows how to do this:

   ```
   <cicsts_dataSource id="defaultCICSDataSource" jndiName="jdbc/defaultCICSDataSource">
       <properties.db2.jcc currentSchema="DB2USER" fullyMaterializeLobData="true" />
   </cicsts_dataSource>
   ```

   Not all properties permitted on the **properties.db2.jcc** element are appropriate for JDBC type 2 driver DataSources. The inappropriate properties are ignored.

   Note: Unrecognized properties are ignored. The following properties are not relevant for the CICS JDBC type 2 driver support. If specified they will also be ignored and a warning message issued: "driverType","serverName","portNumber","user","password","databaseName".

Results

The Liberty server, when started, will be configured to allow access to DB2 databases via a JDBC type 2 driver connection.

| **Note:** Dynamic updates of the CICS DataSource and its components are not
| supported. Updating the configuration whilst the Liberty server is running can
| result in DB2 application failures. You should recycle the server to activate any
| changes.

Related concepts:

Configuring security for a Liberty JVM server in Securing

Related tasks:

"Manually configuring a DB2 JDBC type 4 driver DataSource for Liberty"
Liberty profile users in CICS can use Liberty supplied DataSource definitions to access DB2 databases using JDBC type 4 from Java applications.

Manually configuring a DB2 JDBC type 4 driver DataSource for Liberty

Liberty profile users in CICS can use Liberty supplied DataSource definitions to access DB2 databases using JDBC type 4 from Java applications.

Before you begin

The Liberty DB2 JDBC type 2 DataSource does not use the CICS DB2connection resource, however, 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 connections. This allows the establishment of connections to remote DB2 databases. As updates made to a DB2 database using a JDBC type 4 driver do not use the CICS DB2 connection resource, they are not part of the CICS unit of work unless they are made within a JTA user transaction, see "Java Transaction API (JTA)" on page 56.

Procedure

1. Add the **jdbc-4.0** feature to the featureManager element. This enables use of the **dataSource** and **jdbcDriver** elements used later in the `server.xml`file.

```
<featureManager>
    <feature>jdbc-4.0</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" />
    <fileset dir="/usr/lpp/db2v11/jdbc/lib" />
</library>
```

The **dataSource** specifies a**jndiName** attribute, this defines the JNDI name that is a reference used by your application program when establishing a connection to that DataSource. 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 datasource. 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 profile: securityUtility command.

Required: true

Data type: string

Results

The Liberty server, when started, will be configured to allow access to DB2 databases via a JDBC type 4 driver connection. For more information see, Liberty profile: Configuration elements in the server.xml file.

Related concepts:

Configuring security for a Liberty JVM server in Securing

Related tasks:

"Creating a default CICS DB2 JDBC type 2 driver DataSource for Liberty" on page 93
Create a default CICS DB2 JDBC type 2 driver DataSource using the autoconfigure property.

"Manually configuring a CICS DB2 JDBC type 2 driver DataSource for Liberty" on page 94
Liberty profile users in CICS can use JDBC type 2 driver DataSource definitions to access DB2 databases from Java applications.

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 100. Follow the coding rules, including those for the profile name, in "Rules for coding JVM profiles" on page 101.
   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 80.
   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. See Chapter 8, "Troubleshooting Java applications," on page 165 for help.

What to do next

- Specify any directories that contain native C dynamic link library (DLL) files, such as DB2 or WebSphere 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 in Developing applications.

"JVM profile example"
Example JVM profile configured to start Axis2.

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 the section Working with bundles in the *CICS*
| *Explorer User Guide*.

Procedure

Create a JVMSERVER resource for the JVM server.

1. 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.

2. 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.
3. 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." Follow the coding rules in "Rules for coding JVM profiles" on page 101.

   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.
4. 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. See Chapter 8, "Troubleshooting Java applications," on page 165.

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 WebSphere 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 104. 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/cicsts52/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 User guide for the IBM SDK for z/OS, 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.

“Validation of JVM profile options” on page 103
CICS carries out a number of checks on key options specified in your JVM profiles whenever you start JVMs. These checks enable the early detection of problems in your JVM setup.

“Options for JVMs in a CICS environment” on page 104
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.

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 104 and "JVM system properties" on page 112.

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 9. 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 10 on page 103

Table 10. 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 10. 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
```

Validation of JVM profile options

CICS carries out a number of checks on key options specified in your JVM profiles whenever you start JVMs. These checks enable the early detection of problems in your JVM setup.

CICS carries out checks relating to the following JVM profile options:

CLASSPATH_PREFIX, CLASSPATH_SUFFIX

If either of these options are found in the JVM profile, the JVM server starts without the OSGi framework.

JAVA_HOME

CICS checks the following points for this directory:

- The directory exists in z/OS UNIX.
- CICS has at least read permission to access the directory.

- The JDK_INSTALL_OK file is present in the directory, indicating a completed installation of the IBM 64-bit SDK for z/OS, Java Technology Edition 7 files in this location.
- The Java release number in the JDK_INSTALL_OK file is a version supported by CICS.

If any problems are found, CICS issues an error message and does not start the JVM.

OSGI_BUNDLES
: For JVM server profiles, CICS checks that the specified JAR files are OSGi bundles. CICS also checks that the middleware bundles are correctly delimited and have the right separators.

Deprecated class path options: LIBPATH, CLASSPATH
: A warning message is issued at JVM startup if you have one or more of these options in a JVM profile. Do not use these options in JVM profiles. The message advises on the correct option to use instead.

Deprecated class path options: CICS_HOME, TMPREFIX, TMSUFFIX
: These options are ignored.

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 101.

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=YES`, or they begin with a hyphen, for example `-Xmx16M`.
- Keyword value pairs are either CICS variables such as `JAVA_PIPELINE=YES`, 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 without being parsed by CICS. For more information, see "JVM system properties" on page 112.

"Symbols used with JVM options" on page 105
You can use various substitution symbols in JVM server options. 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.

"JVM server options" on page 105
JVM server options, with descriptions and applicability to different uses of JVM server.

"JVM command-line options" on page 111
JVM command-line options, with descriptions.

"JVM system properties" on page 112
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.

Symbols used with JVM options:

You can use various substitution symbols in JVM server options. 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 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, with descriptions and applicability to different uses of JVM server.

Applicability of options to different uses of JVM server

Different options are applicable, depending on how the JVM server is being used. The following table indicates whether an option is mandatory, supported but optional, or not supported for a particular use of a JVM server.

Table 11. Options by JVM server use

Option	OSGi	Liberty	Axis2	STS
CLASSPATH_PREFIX	Not supported	Not supported	Supported	Supported
CLASSPATH_SUFFIX	Not supported	Not supported	Supported	Supported
JAVA_DUMP_TDUMP_PATTERN	Supported	Supported	Supported	Supported
JAVA_HOME	Mandatory	Mandatory	Mandatory	Mandatory
JAVA_PIPELINE	Not supported	Not supported	Mandatory	Not supported
JNDI_REGISTRATION	Supported	Not supported	Supported	Supported
JVMTRACE	Supported	Supported	Supported	Supported
LIBPATH_PREFIX	Supported - use only under the guidance of IBM service personnel	Supported - use only under the guidance of IBM service personnel	Supported - use only under the guidance of IBM service personnel	Supported - use only under the guidance of IBM service personnel
LIBPATH_SUFFIX	Supported	Supported	Supported	Supported
OSGI_BUNDLES	Supported	Not supported	Not supported	Not supported
OSGI_FRAMEWORK_TIMEOUT	Supported	Supported	Not supported	Not supported
PRINT_JVM_OPTIONS	Supported	Supported	Supported	Supported
SECURITY_TOKEN_SERVICE	Not supported	Not supported	Not supported	Mandatory
STDERR	Supported	Supported	Supported	Supported
STDIN	Supported	Supported	Supported	Supported
STDOUT	Supported	Supported	Supported	Supported
USEROUTPUTCLASS	Supported	Not supported	Supported	Supported
WLP_INSTALL_DIR	Not supported	Mandatory	Not supported	Not supported
WLP_OUTPUT_DIR	Not supported	Supported	Not supported	Not supported
WLP_USER_DIR	Not supported	Supported	Not supported	Not supported
WLP_ZOS_PLATFORM	Not supported	Supported	Not supported	Not supported
WORK_DIR	Supported	Supported	Supported	Supported
WSDL_VALIDATOR	Supported	Not supported	Supported	Supported

Descriptions of options

Default values where applicable are the values that CICS uses when the option is not specified. Some or all of the sample JVM profiles might specify a value that is different from the default value.

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=YES 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.

JAVA_DUMP_TDUMP_PATTERN=
A z/OS UNIX System Services environment variable that specifies the file name 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={YES,NO}
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 NO. 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=YES and
| SECURITY_TOKEN_SERVICE=YES are not compatible.

| **JNDI_REGISTRATION={YES|NO}**
| 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=NO. 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*}
Specifies the name of the z/OS UNIX file to which Java tracing is written during the startup and termination of a JVM server. If you do not set a value for this option, CICS automatically creates unique trace files for each JVM server. CICS uses the &APPLID; and &JVMSERVER; symbols, and the date and time stamp when the JVM server started. This file is created in the directory that is specified by the `WORK_DIR` option.

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 additional 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.

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 WebSphere MQ or DB2. If you specify more than one OSGi bundle, use commas to separate them.

OSGI_FRAMEWORK_TIMEOUT={60|*number***}**
Specifies the number of seconds that CICS waits for the OSGi framework to initialize or shut down before timing 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={YES|NO}
If this option is set to YES, whenever a JVM starts, the options that are passed to the JVM at startup 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.

| **SECURITY_TOKEN_SERVICE={YES|NO}**
| If this option is set to YES, the JVM server can use security tokens. If this
| option is set to NO, Security Token Service support is disabled for the JVM
| server.

| **Note:** The options SECURITY_TOKEN_SERVICE=YES and
| JAVA_PIPELINE=YES are not compatible.

STDERR={&APPLID;.&JVMSERVER;.Dyyyymmdd.Thhmmss.dfhjvmerr|*file_name*}

Specifies the name of the z/OS UNIX file to be used for `stderr`. If the file does not exist, it is created in the directory that is specified by the `WORK_DIR` option. If the file exists, output is appended to the end of the file. If you do not set a value for this option, CICS automatically creates unique output files for each JVM server. CICS uses the &APPLID; and &JVMSERVER; symbols, and the date and time stamp when the JVM server started. 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 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. This is the case 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 to be used for `stdin`. CICS does not create this file unless you specify a value for this option.

STDOUT={&APPLID;.&JVMSERVER;.Dyyyymmdd.Thhmmss.dfhjvmout|*file_name*}

Specifies the name of the z/OS UNIX file that is to be used for output to the `stdout` file. If the file does not exist, it is created in the directory that is specified by the `WORK_DIR` option. If the file exists, output is appended to the end of the file. If you do not set a value for this option, CICS automatically creates unique output files for each JVM server. CICS uses the &APPLID; and &JVMSERVER; symbols, and the date and time stamp when the JVM server started.

If you specify the `USEROUTPUTCLASS` option in 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 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. 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" on page 170.

WLP_INSTALL_DIR={$USSHOME;/wlp|*directory_path*/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/cicsts52/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 112.

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/`*server_name*, 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=FALSE
Disables the z/OS platform extensions in a Liberty JVM server. This prevents the use the use of the cicsts:security-1.0 feature, and allows more than one Liberty JVM server to be started in the same region.

WORK_DIR={.|*directory_name*}
Specifies the working directory on z/OS UNIX that the CICS region uses for activities that are related to Java. The CICS JVM interface uses this directory when it creates the `stdin`, `stdout`, and `stderr` files. 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. You specify a directory name that uses the &APPLID; symbol, for which CICS substitutes the actual CICS region APPLID. So 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 execute 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 directory on z/OS UNIX, which is the home directory for CICS files as defined by the **USSHOME** system initialization parameter.

You can also use the option `USEROUTPUTCLASS` to name a Java class that intercepts, redirects, and formats the `stderr` and `stdout` output from a JVM. The supplied sample classes for output redirection use the directory that is specified by `WORK_DIR` in some circumstances.

| **WSDL_VALIDATOR={YES|NO}**
| 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=NO. Opting out can prevent potential
| clashes with newer JAR filess, wasted storage, and slower startup.

JVM command-line options:

JVM command-line options, with descriptions.

List of command-line options

-agentlib
Specifies whether debugging support is enabled in the JVM.

For more information, see "Debugging a Java application" on page 176. For more information about the Java Platform Debugger Architecture (JPDA), see Oracle Technology Network Java website.

-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.

-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

-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.

For information about the **-Xmso** JVM option and default value, see JVM command-line options.

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

-Dcom.ibm.cics indicates that the property is specific to the IBM JVM in a CICS environment.

-Dcom.ibm indicates a general JVM property that is used more widely.

-Djava.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 100.

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 101.

Applicability of properties to different uses of JVM server

Different properties are applicable, depending on how the JVM server is being used. The following table indicates whether a property is mandatory, optional, or inapplicable for a particular use of a JVM server.

Table 12. Options by JVM server use

Option	OSGi	Liberty	Axis2	STS
-Dcom.ibm.cics.jvmserver.applid	Supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.configroot	Supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.controller.timeout	Supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.local.ccsid	Supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.name	Supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.override.ccsid	Supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.supplied.ccsid	Supported	Supported	Not supported	Not supported

Table 12. Options by JVM server use (continued)

Option	OSGi	Liberty	Axis2	STS
-Dcom.ibm.cics.jvmserver.threadjoin.timeout	Supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.trace.filename	Supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.trace.format	Supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.trigger.timeout	Supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.wlp.args	Not supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.wlp.autoconfigure	Not supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.wlp.jdbc.driver.location	Not supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.wlp.optimize.static.resources	Not supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.wlp.server.host	Not supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.wlp.server.http.port	Not supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.wlp.server.https.port	Not supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.jvmserver.wlp.server.name	Not supported	Supported	Not supported	Not supported
-Dcom.ibm.cics.sts.config	Not supported	Not supported	Not supported	Supported
-Dcom.ibm.ws.logging.console.log.level	Not supported	Supported	Not supported	Not supported
-Dconsole.encoding	Supported	Supported	Supported	Supported
-Dfile.encoding	Supported	Supported	Supported	Supported
-Djava.security.manager	Supported	Supported	Supported	Supported
-Djava.security.policy	Supported	Supported	Supported	Supported

Descriptions of properties

-Dcom.ibm.cics.jvmserver.applid=
The CICS region application identifier (APPLID). This is a read-only property. You can view its value for use in an application but you cannot change it.

-Dcom.ibm.cics.jvmserver.configroot=*zFS_directory*
The zFS location where configuration files, such as the JVM profile of a JVM server, can be found. This is a read-only property. You can view its value for use in an application but you cannot change it.

-Dcom.ibm.cics.jvmserver.controller.timeout={*time***|**<u>**60000ms**</u>**}**
Use this property only under IBM service guidance. It is subject to change at any time.

-Dcom.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 view its value for use in an application but you cannot change it.

-Dcom.ibm.cics.jvmserver.name=
The name of the JVM server. This is a read-only property. You can view its value for use in an application but you cannot change it.

-Dcom.ibm.cics.jvmserver.override.ccsid=
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 in Reference -> System definition.

-Dcom.ibm.cics.jvmserver.supplied.ccsid=
The default CCSID for the local region. This is a read-only property. You can view its value for use in an application but you cannot change it.

-Dcom.ibm.cics.jvmserver.threadjoin.timeout={*time*|30000ms}
Use this property only under IBM service guidance. It is subject to change at any time.

-Dcom.ibm.cics.jvmserver.trace.filename=
The name of the JVM server trace file. This is a read-only property. You can view its value for use in an application but you cannot change it.

-Dcom.ibm.cics.jvmserver.trace.format={FULL|SHORT|ABBREV}
Controls the format of the trace. You may vary the trace format for your own purposes but you must set it to `FULL` when you send diagnostic information to IBM service.

-Dcom.ibm.cics.jvmserver.trigger.timeout={*time*|500ms}
Use this property only under IBM service guidance. It is subject to change at any time.

-Dcom.ibm.cics.jvmserver.wlp.args=
Use this property only under IBM service guidance.

-Dcom.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 directory structure and configuration files in zFS. CICS also updates the `server.xml` file if you provide values for other Java properties, such as an HTTP port number.

-Dcom.ibm.cics.jvmserver.wlp.jdbc.driver.location=
Specifies the location of the directory in zFS that contains the DB2 JDBC drivers. The location must contain the DB2 JDBC driver `classes` and `lib` directories. If the autoconfigure property **-Dcom.ibm.cics.jvmserver.wlp.autoconfigure** is set to 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.

-Dcom.ibm.cics.jvmserver.wlp.optimize.static.resources={true|false}
Enables CICS to use fewer transactions to satisfy a request, by using static resource optimization. The following types of file are recognized as static: `.css`, `.gif`, `.ico`, `.jpg`, `.jpeg`, and `.png`.

-Dcom.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 unlikely to be appropriate for running a web application in CICS, so either use this property to provide a different value or update the `server.xml` file. This property is optional and is used only when the **-Dcom.ibm.cics.jvmserver.wlp.autoconfigure** property is set to true.

-Dcom.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 the Liberty profile. 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. This property is optional and is used only when the **-Dcom.ibm.cics.jvmserver.wlp.autoconfigure** property is set to true.

-Dcom.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 the Liberty profile. 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. This property is optional and is used only when the **-Dcom.ibm.cics.jvmserver.wlp.autoconfigure** property is set to true.

-Dcom.ibm.cics.jvmserver.wlp.server.name={defaultServer|*server_name*}
Specifies the name of the Liberty profile server. The default for this property is `defaultServer`. This property is optional and you should not need to specify it as it affects the location of the Liberty server configuration and output files and directories on the zFS system.

-Dcom.ibm.cics.sts.config=*path*
The location and name of the STS configuration file.

| **-Dcom.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.

-Dconsole.encoding=
Specifies the encoding for JVM server output files.

-Dfile.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 for file encoding.
- In a profile that is configured for Axis2, you must specify an EBCDIC code page.

-Djava.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:

- `-Djava.security.manager=default`
- `-Djava.security.manager=""`
- `-Djava.security.manager=`

All these statements enable the default security manager. If you do not include the **-Djava.security.manager** system property in your JVM profile, the JVM runs without Java security enabled. To disable Java security for a JVM, comment out this system property.

-Djava.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 **-Djava.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" on page 162.

Chapter 4. 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 CICS Explorer SDK to package and deploy the application.

All Java applications that run in a JVM server must be threadsafe and must comply with the OSGi specification, so ensure that your application meets this requirement before deploying it. 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 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.

“Deploying a web application in a CICS bundle to a Liberty JVM server” on page 119
You can deploy a Java web application that is packaged as a CICS bundle in a Liberty JVM server.

“Deploying a web application as a dropin directly to a Liberty JVM server” on page 121
You can follow the Liberty profile deployment model and export and deploy a WAR or EBA file to a Liberty JVM server.

“Invoking a Java application in a JVM server” on page 122
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. For example, HTTP requests from a client browser are used to call Web applications in a Liberty JVM server. EXEC CICS LINK or EXEC CICS START are used to drive OSGi bundle applications in an OSGi JVM server. While the vendor interface DFHSJJI, can be used to link to classpath Java function in a non-OSGi JVM server.

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.

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 122. To update or remove an application, see Chapter 5, “Administering Java applications,” on page 125.

Deploying a web application in a CICS bundle to a Liberty JVM server

You can deploy a Java web application that is packaged as a CICS bundle in a Liberty JVM server.

Before you begin

The Java web application, either in the form of WAR 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 CICS Explorer SDK” on page 29.

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 117.

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 web application either contains one or more WAR files that provide the presentation layer and business logic of the application, or 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.

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 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 web applications into the Liberty server.
      - If you install the resource in an ENABLED state, CICS installs the web applications (WAR, 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 web 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, CICS was unable to create the resource for that bundle part. 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. Optional: To run web 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 web 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 application transaction and set the scheme to HTTP or HTTPS. You can also use the **USERID** attribute to set a user ID. This value is ignored if the application uses basic authentication. If you do not use basic authentication or set a user ID on the URI map, the work runs under the CICS default user ID.

Results

The CICS resources are enabled, and the web 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 a web application as a dropin directly to a Liberty JVM server

You can follow the Liberty profile deployment model and export and deploy a WAR or EBA file to a Liberty JVM server.

Before you begin

The JVM server must be configured to use the Liberty profile technology and must be enabled in the CICS region.

About this task

The Liberty profile deployment model provides a quick way to deploy and run web components in an application server. You can follow this model by deploying web applications as web archive (WAR) or enterprise bundle archive (EBA) files into the `dropins` directory of the Liberty JVM server. If you use this model, CICS is not aware of the web application that is running in the JVM server. Therefore, do not deploy the same application by using both models into the same JVM server because errors might result. Applications that are deployed by this method do not benefit from additional qualities of service such as security and always run under transaction CJSA.

| **Note:** If you accept the defaults that are provided by CICS autoconfigure, the
| `dropins` directory is not automatically created.

Procedure

1. Export the Eclipse web project as a WAR or EBA file to your local workstation.
2. Add this line to the `server.xml` in your CICS Liberty configuration.

```
<applicationMonitor dropins="dropins" dropinsEnabled="true" pollingRate="5s"
    updateTrigger="disabled"/>
```

3. Use FTP to transfer the exported file in binary mode to the `dropins` directory of the Liberty profile server. 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.

Results

The Liberty JVM server automatically detects the deployed WAR or EBA file and installs it.

What to do next

Access the web application from a web browser to ensure that it is running correctly. To remove the application file, delete the WAR or EBA file from the `dropins` directory.

| You can use the CICS Explorer SDK to package and deploy applications. For more
| information see Chapter 4, "Deploying applications to a JVM server," on page 117.

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. For example, HTTP requests from a client browser are used to call Web applications in a Liberty JVM server. EXEC CICS LINK or EXEC CICS START are used to drive OSGi bundle applications in an OSGi JVM server. While the vendor interface DFHSJJI, can be used to link to classpath Java function in a non-OSGi 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. To invoke a Java application that is running in an OSGi JVM server, you can either EXEC CICS LINK to a PROGRAM defined Java, or EXEC CICS START a TRANSACTION that has a target PROGRAM defined as Java. The PROGRAM definition specifies a JVMSERVER, and the name of a CICS generated OSGi service you wish 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 wish 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" on page 127.

Note: In a Liberty JVM server, you can also create OSGi services within an Enterprise Bundle Archive (EBA), or within the OSGi application project that defines an EBA (Web application). However these OSGi services cannot be the target of a CICS PROGRAM definition, nor are they visible to components packaged outside of that same EBA.

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 WAR file or as an EBA file containing WAB files and running in a Liberty JVM server, invoke the application from the client browser by using a URL.
- For OSGi bundles that are deployed in an OSGi JVM server, follow these steps:
  1. 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.
  2. 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.
  3. 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 Axis, Batch or SAML function see, "Configuring a JVM server for Axis2" on page 97, and .

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.

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.

“Removing OSGi bundles from a JVM server” on page 129
If you want to remove OSGi bundles from the JVM server, use the CICS Explorer to disable and discard the BUNDLE resource.

“Moving applications to a JVM server” on page 130
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.

“Managing the thread limit of JVM servers” on page 131
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.

“OSGi bundle recovery on a CICS restart” on page 132
When you restart a CICS region that contains OSGi bundles, CICS recovers the BUNDLE resources and installs the OSGi bundles into the framework of the JVM server.

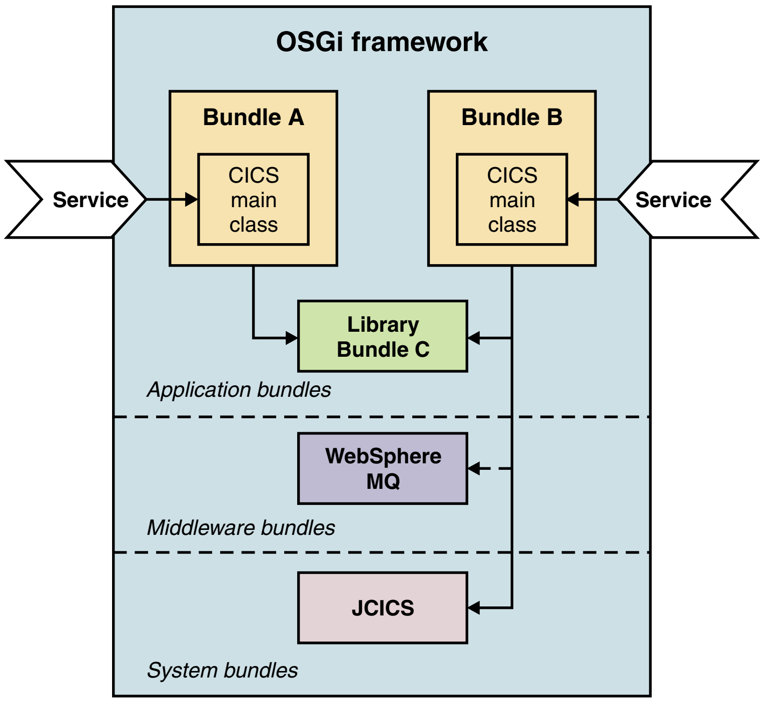
“Writing Java classes to redirect JVM stdout and stderr output” on page 133
Use the `USEROUTPUTCLASS` option in a JVM profile to name a Java class that intercepts the stdout and stderr output from the JVM. You can update this class to specify your choice of time stamps and record headers, and to redirect the output.

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 a WebSphere 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 WebSphere 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" on page 127
If a Java developer provides an updated version of a CICS bundle, you can either completely replace the CICS bundle or you can phase in a new version and then remove the old version.

"Updating bundles that contain common libraries" on page 128
OSGi bundles that contain common libraries for use by other OSGi bundles must be updated in a specific order.

"Updating OSGi middleware bundles" on page 129
If you want to update the middleware bundles that are running in an OSGi framework, you must stop and restart the JVM server.

Updating OSGi bundles

If a Java developer provides an updated version of a CICS bundle, you can either completely replace the CICS bundle or you can phase in a new version and then remove the old version.

Before you begin

An updated CICS bundle that contains the new version of an OSGi bundle must be present in zFS.

Procedure

- To replace an existing OSGi bundle in an OSGi JVM server:
  1. Disable and discard the BUNDLE resource for the CICS bundle that you want to update. The OSGi bundles and services that are part of that CICS bundle are removed from the OSGi framework.
  2. Optional: Edit the BUNDLE resource definition if the updated CICS bundle is deployed in a different directory.
  3. 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 bundles and services in the **Operations** > **Java** views in CICS Explorer.
- To phase in a new version in an OSGi server and have both bundles running in the framework at the same time, the OSGi service must have an alias specified. If no alias is specified, the service is listed as inactive in the framework because it is considered a duplicate of the service that is already running.
  1. Create a BUNDLE resource to pick up the changed CICS bundle. The OSGi bundles and services in the CICS bundle are installed in the OSGi framework. The OSGi service is in the inactive state, unless an alias is specified in the bundle manifest.
  2. Check the status of the OSGi bundles and services in the **Operations** > **Java** views in CICS Explorer. Two versions of the OSGi bundle are listed in the OSGi bundles view. The OSGi service for the bundle is in the inactive state, unless an alias is specified. If an alias is specified, both OSGi services are active.
  3. Disable the BUNDLE resource that points to the old version of the OSGi bundle. CICS removes the OSGi service that is associated with the bundle and sets the OSGi bundle to the resolved state. As a result, the OSGi service for the changed OSGi bundle moves from inactive to active state.
  4. If there is an alias for the OSGi service, you can specify the alias in a PROGRAM resource to call the updated application from outside the JVM server.

Results

If you are using an OSGi JVM server the symbolic version of the OSGi bundle increases, indicating that the Java code is updated. The updated OSGi bundle is available in the OSGi framework and can be called from outside the JVM server.

If you are using an EBA in a Liberty JVM server the OSGi packages are refreshed when the bundle is enabled, and are available to the web application within the EBA.

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 pick up the latest 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 running on an older version.

When you update an OSGi bundle that contains common libraries, you can completely replace the CICS bundle. However, if classes have not been loaded in the library, the dependent bundles might receive errors. You can phase in 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

1. Create a BUNDLE resource that points to the new version of the OSGi bundle. CICS creates 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 OSGi Bundles view in the CICS Explorer. The list shows two entries for the same OSGi bundle symbolic name with different versions running in the framework.
3. To pick up the new version of the library in a dependent Java application:
   a. Disable and discard the BUNDLE resource for the Java application. Alternatively, ask the Java developer to update the version information for the OSGi bundle and deploy a new version of the CICS bundle to maintain the availability of the application.
   b. Install the BUNDLE resource. When the OSGi bundle is loaded in the framework, it picks up the latest version of the common libraries.

4. Check the status of the BUNDLE resource in the Bundles view of the CICS Explorer.

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

If you want 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 Explorer SDK to repackage a Java application using OSGi. For more information on how to migrate applications that use third party JARs, see .

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 "Setting up a JVM server" on page 84. 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 the 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 the CICS Explorer and CICS statistics to monitor the JVM server. If the performance is not optimal, adjust the thread limit.

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” on page 142.

OSGi bundle recovery on a CICS restart

When you restart a CICS region that contains OSGi bundles, CICS recovers the BUNDLE resources and installs the OSGi bundles into the framework of the JVM server.

OSGi bundles that are packaged in CICS bundles are not stored in the CSD. The BUNDLE resource itself is stored in the catalog, so that on a restart of the CICS region, the OSGi bundles are dynamically re-created when the BUNDLE resource is restored.

On a cold, warm, or emergency restart of CICS, the JVM server is started asynchronously to BUNDLE resource recovery. The JVM server must be fully available to successfully restore an OSGi bundle on a CICS restart. Therefore, although the BUNDLE resources are recovered during the last phase of CICS startup, the OSGi bundles are installed only when the JVM server has completed its startup.

BUNDLE resources and the OSGi bundles that they contain are installed in the correct order to ensure that the dependencies between both CICS bundles and OSGi bundles are resolved in the framework. If CICS fails to install an OSGi bundle, the BUNDLE resource installs in a disabled state. You can use the IBM CICS Explorer to view the state of BUNDLE resources, OSGi bundles, and OSGi services.

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 and stderr output 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/cicsts52/samples/com.ibm.cics.samples`. The `/usr/lpp/cicsts/cicsts52` directory is the install directory for CICS files on z/OS UNIX. This directory is specified by the **`USSDIR`** parameter in the DFHISTAR install job. The sample classes are also shipped as a class file, `com.ibm.cics.samples.jar`, which is in the directory `/usr/lpp/cicsts/cicsts52/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" on page 170 has information about:

- The types of output from JVMs that are and are not intercepted by the class 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 containing APPLID, date, time, transaction ID, task number, and program name. The log files are created 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.

“Possible destinations for output” on page 135
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.

“Handling output redirection errors and internal errors” on page 135
If your classes use CICS facilities to redirect output, they should include appropriate exception handling to deal with errors in using these facilities.

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. 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

Even if CICS is well-tuned, applications that are written inefficiently perform badly when compared with applications that are written well. One way of improving Java performance is to modify your applications so that less garbage is generated; this can produce significant savings on garbage collection costs because if less garbage is produced, less time is spent handling garbage collection. To improve performance, always ensure that your Java applications are written efficiently, and ensure that the Java environment is well-tuned.

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” on page 138
Tuning CICS JVMs to achieve the best overall performance for a given application workload involves several different factors. You must decide what the desired performance characteristics of your Java workload are. When you establish these characteristics, you can determine what parameters to change and how to change them.

“Analyzing Java applications using IBM Health Center” on page 139
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.

“Garbage collection and heap expansion” on page 140
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.

“Improving JVM server performance” on page 142
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.

"Language Environment enclave storage for JVMs" on page 146
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. You can modify the runtime options for the enclave to tune the storage that is allocated by MVS.

"Tuning the z/OS shared library region" on page 150
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.

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 desired 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. JVMs use 64-bit storage so it is possible to run many pooled JVMs and JVM servers in a CICS region. If pooled JVMs use a smaller storage heap, it might be possible to run more of them in the CICS region. However, choosing this goal might increase processor costs. Tuning JVMs to minimize the storage heap size results in a greater frequency of 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. 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 User guide for the IBM SDK for z/OS, Java Technology Edition, Version 7.

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.

You can change the garbage collection policy by updating the JVM profile. For details of all the garbage collection options, see Specifying garbage collection policy in the User guide for Java Version 7 on z/OS.

Example: a multithreaded application producing large amounts of garbage

Figure 2 shows the storage heap in a JVM server at various stages for the `gencon` garbage collection policy. A JVM server can run many concurrent requests for the same application; an application can therefore produce larger amounts of garbage. Garbage collection in a JVM server is handled automatically by the JVM.

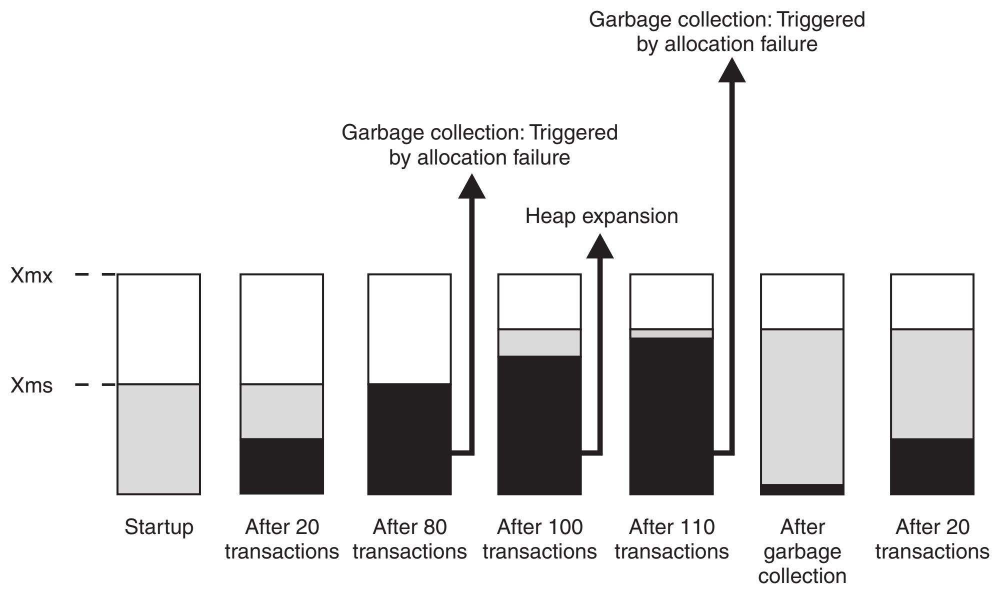


Figure 2. Storage heap in a JVM server with large amounts of garbage

During the first 20 transactions, the active part of the storage heap starts to fill. After 80 transactions, the heap becomes full and an allocation failure occurs, which triggers a minor garbage collection in the JVM. The garbage collection cleans up the short lived objects. However, because application requests are still running, some of the objects are still referenced, so they are not eligible for garbage collection.

After 100 transactions, the Garbage Collector cannot find enough space for all the currently needed objects and it expands the storage heap . Some storage from the maximum amount of storage reserved for the storage heap (the amount specified by the `-Xmx` option) is added to the active part of the heap. The application continues to produce objects, but the heap expansion has now created enough space so that the current transaction can complete.

After 110 transactions, the storage heap is largely occupied. Another allocation failure occurs that triggers a major garbage collection. The JVM cleans up many of the longer lived objects used by previous transactions. After another 20 transactions, the heap starts to fill up again.

When you use the `gencon` policy, many minor garbage collections can occur to manage the heap size before a major garbage collection occurs. You can find out how many garbage collections have occurred, the heap occupancy, and other information by using JVM server statistics.

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 you have enough storage available to adjust the heap sizes 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.

“Examining processor usage by JVM servers” on page 143
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.

“Calculating storage requirements for JVM servers” on page 143
To increase the number of JVM servers in a CICS region, you must ensure that enough storage is available to CICS.

“Tuning JVM server heap and garbage collection” on page 145
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.

“Tuning the JVM server startup environment” on page 146
If you are running multiple JVM servers, you can improve performance by tuning the JVM startup environment.

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. Switch 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.

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 master 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" on page 168.
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 increase the number of JVM servers in a CICS region, you must ensure that enough storage is available to CICS.

About this task

JVMs use storage below the 16 MB line, 31-bit storage, and 64-bit storage. To run a JVM server incurs a one-off storage cost, no matter how many JVMs run in the CICS region. Each JVM server and its Language Environment enclave also require a certain amount of 31-bit and 64-bit storage. The JVM heap sizes are managed by the JVM and CICS uses the default values. You can adjust the heap size if required as part of tuning the environment.

The storage required for the JVM heap comes from the CICS region storage (from MVS storage, not EDSA storage). Larger JVM heaps reduce the number of JVMs that can be present in a CICS region, and increase the region size to support them. However, if the heap size is set too small, excessive garbage collection takes place, which affects performance. You can tune the JVM storage options to achieve the best performance for your Java workloads. The JVM storage options help to determine the processor usage, storage usage, and task response times for Java applications.

Procedure

1. Determine the amount of free storage available below the bar by using the sample statistics program DFH0STAT. The storage reports include the amount of user storage allocated in 31-bit storage and below the 16 MB line.
   - If you have no JVM servers running, subtract the storage that is reserved for the z/OS shared library region from the total amount of free storage in the CICS address space. The storage is controlled by the **SHRLIBRGNSIZE** parameter in MVS and is allocated once when the first JVM is started in the region.
   - If you have JVM servers running, subtract the value of the **SHRLIBRGNSIZE** parameter from the total amount of free storage. Each JVM that is running uses 12 KB of storage below the 16 MB line. The Language Environment enclave for each JVM uses 31-bit storage for the heap and library heap. The amount of allocated 31-bit storage is set by the HEAP64 and LIBHEAP64 options in DFHOSGI. You must also subtract these values from the total amount of free storage to work out how much storage is currently available.

   If you want to change the 31-bit storage settings, you can adjust the **SHRLIBRGNSIZE** parameter and the Language Environment options. See "Tuning the z/OS shared library region" on page 150 and "Modifying the enclave of a JVM server with DFHAXRO" on page 149.
2. Calculate how much 64-bit storage is required for each additional JVM server. You can calculate the 64-bit storage requirements for a JVM server by adding up the following storage requirements:
   - The **-Xmx** value. The default value for this parameter is set by the JVM, so check the documentation in the Java information center.
   - The value of the 64-bit storage that is allocated by the HEAP64 option in DFHAXRO.
   - The value of the 64-bit storage that is allocated by the LIBHEAP64 option in DFHAXRO.
   - The value of the 64-bit storage that is allocated by the STACK64 option in DFHAXRO. Multiply this value by the number of threads that are allowed in the JVM server. To calculate the number of allowed threads, add the THREADLIMIT attribute value on the JVMSERVER resource to the value of the **-Xgcthreads** parameter. This Java option controls the number of garbage collection helper threads in the JVM.

3. Check the **MEMLIMIT** value to determine whether you have enough 64-bit storage available to run additional JVM servers. You must allow for the other CICS facilities that use 64-bit storage.

   The z/OS **MEMLIMIT** parameter limits the amount of 64-bit (above-the-bar) storage for the CICS region. For information about the CICS facilities that use 64-bit storage, and how to check and adjust this parameter, see Estimating, checking, and setting MEMLIMIT in Improving performance.

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/cicsts52/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.
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 runs as a z/OS UNIX System Services process in a Language Environment enclave that is created using the Language Environment preinitialization module, CELQPIPI. You can modify the runtime options for the enclave to tune the storage that is allocated by MVS.

JVMs use MVS Language Environment services rather than CICS Language Environment services. As a result, all storage obtained by the JVM is MVS storage, obtained by calls to MVS Language Environment services. This storage resides within the CICS address space but is not included in the CICS dynamic storage areas (DSAs). All JVMs that run in CICS use 64-bit storage.

The Language Environment enclave for each JVM must contain not only the JVM storage heap, but also a basic amount of storage for each JVM. This basic storage cost represents the amount of storage in the Language Environment enclave that is used for the structure of the JVM. When you calculate the total size of the JVM, the basic storage cost must be added to the storage that is used for the storage heap.

The Language Environment runtime options are set by DFHAXRO. The default values provided by these programs for a JVM enclave are shown in Table 13:

Table 13. Language Environment runtime options used by CICS for the JVM enclave

Language Environment runtime options	JVM server values
Heap storage	`HEAP64(100M,4M,KEEP,4M,512K, KEEP,1K,1K,KEEP)`
Library heap storage	`LIBHEAP64(3M,3M)`
Library routine stack frames that can reside anywhere in storage	`STACK64(1M,1M,32M)`
Optional user heap storage management for multithreaded applications	`HEAPPOOLS64(ALIGN)`
Optional heap storage management for multithreaded applications	`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)`

For information about Language Environment runtime options, see *z/OS Language Environment Customization*.

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 149. This program is set on the JVMSERVER resource, so you can use different options the Language Environment enclave for individual JVM servers if required. The default Language Environment storage settings that control the initial size of, and incremental additions to, the Language Environment enclave heap storage can make inefficient use of MVS storage. The storage settings that CICS supplies are more efficient. You can also modify these settings to match more closely with the storage use of your JVMs. Ensure that the heap sizes are set to avoid many segment allocations and frees.

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" on page 148
You can identify a suitable value for the initial allocation of Language Environment enclave heap storage in a JVM server by generating storage reports. Generating storage reports increase processor costs, so run them at an appropriate time in a production environment.

"Modifying the enclave of a JVM server with DFHAXRO" on page 149
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. For CICS, the storage settings that are supplied in DFHAXRO are more appropriate than the default Language Environment storage settings.

Identifying Language Environment storage needs for JVM servers

You can identify a suitable value for the initial allocation of Language Environment enclave heap storage in a JVM server by generating storage reports. Generating storage reports increase processor costs, so run them at an appropriate time in a production environment.

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 and 31-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. 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 initial Language Environment enclave heap storage in DFHAXRO. Language Environment can make additions to the heap storage, but it cannot remove unwanted storage that is given in the initial allocation. Allocate enough storage to ensure the number of segments allocated and freed are minimal.

You can also use this technique to set the initial size and increment values for the `LIBHEAP64` and `STACK64` runtime options.

Example

The following example is a storage report from Language Environment:

```
64bit User HEAP statistics:
      Initial size:                                              50M
      Increment size:                                             4M
      Total heap storage used:                              91977408
      Suggested initial size:                                    88M
      Successful Get Heap requests:                             2439
      Successful Free Heap requests:                            1619
      Number of segments allocated:                                1
      Number of segments freed:                                    0
31bit User HEAP statistics:
      Initial size:                                           524288
      Increment size:                                         524288
      Total heap storage used (sugg. initial size):          8440784
      Successful Get Heap requests:                             1965
      Successful Free Heap requests:                            1904
      Number of segments allocated:                                2
      Number of segments freed:                                    0
```

Based on the values for Language Environment enclave heap storage in the example, you can set these values for heap storage in DFHAXRO:

```
HEAP64(88M,4M,KEEP,10M,512K,KEEP,1K,1K,KEEP)
```

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. For CICS, the storage settings that are supplied in DFHAXRO are more appropriate than the default Language Environment storage settings.

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 CICSTS52.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.

- To control the permissions on files that are created by the JVM Server, add the following line:
  ```
  DC  C'ENVAR("_EDC_UMASK_DFLT=nnn")'
  ```
  Where *nnn* is the mask value you require.
- 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 bring up 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.

You can find more information about shared libraries in z/OS UNIX on the z/OS UNIX System Services Web site at http://www.ibm.com/servers/eserver/zseries/zos/unix/perform/sharelib.html.

Chapter 7. 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 profile server. The Liberty profile is designed to host web applications and includes an OSGi framework. The security configuration for a Liberty profile server is different, because the Liberty profile has its own security model.

"Configuring security for OSGi applications" on page 154
Use CICS security to authorize the appropriate users to manage the lifecycle of the JVM server and Java applications, and to access the application through CICS.

"Configuring security for a Liberty JVM server" on page 154
You can use the CICS Liberty security feature to authenticate users and authorize access to web applications through Java Platform, Enterprise Edition 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.

"Configuring SSL for a Liberty JVM server" on page 161
Communications are secured with the Secure Sockets Layer (SSL) protocol. The SSL protocol provides transport layer security including authenticity, data signing, and data encryption to ensure a secure connection between a client and server. You can configure a Liberty JVM server to use SSL for data encryption, and optionally authenticate with the server using a client certificate. Client certificates can be stored in a Java key store or in a SAF keyring.

"Enabling a Java security manager" on page 162
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.

Configuring security for OSGi applications

Use CICS security to authorize the appropriate users to manage the lifecycle of the JVM server and Java applications, and to access the application through CICS.

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.

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, 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.

Before you begin

Ensure that the CICS region is configured to use SAF security and is defined with SIT=YES as a system initialization parameter. Then authorize application developers and system administrators to create, view, update, and remove JVMSERVER and BUNDLE resources in order 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 security for a Liberty JVM server and integrate Liberty security with CICS security.

The default transaction ID for running any web request 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.

Procedure

1. Configure the WebSphere Liberty profile angel process to provide authentication and authorization services to the Liberty JVM server, see The Liberty server angel process.
2. Add the `cicsts:security-1.0` feature to the featuremanager list in the `server.xml`,

```
<featureManager>
   ...
   <feature>cicsts:security-1.0</feature>
</featureManager>
...
```

3. Save the changes to `server.xml`.

   Note: 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 for web applications" on page 87.

   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 switched off, and you want to use a Liberty security, you must configure the `server.xml` file manually:

   a. Add the appSecurity 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 for the Liberty profile.
   c. Add security-role definitions to authorize access to application resources, see "Authorizing users to run applications in a Liberty JVM server" on page 159.

Results

When the `cicsts:security-1.0` feature is used, the web container is automatically configured to use the z/OS Security feature of Liberty. Additionally a SAF Registry is used for authentication, and Java Platform, Enterprise Edition roles in an <application-bnd> element are respected for authorization.

What to do next

- Configure Liberty application security authentication rules, see "Authenticating users in a Liberty JVM server" on page 157.
- Define authorization rules for web applications, see "Authorizing users to run applications in a Liberty JVM server" on page 159 and "JEE application role security" on page 160.
- Modify the Liberty authentication cache.

For more information about using Secure Sockets Layer (SSL), see "Configuring SSL for a Liberty JVM server" on page 161.

"The Liberty server angel process" on page 156
The angel process provides authorized services to WebSphere Application Server Liberty Profile servers.

"Authenticating users in a Liberty JVM server" on page 157
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.

"Authorizing users to run applications in a Liberty JVM server" on page 159
To authorize user IDs to run transactions in a Liberty JVM server, you can

either use CICS transaction and resource security or you can use JEE application security roles to authorize access to JEE applications.

"JEE application role security" on page 160
JEE application role security can be configured in different ways depending upon the authorization type you wish to use. 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.

The Liberty server angel process

The angel process provides authorized services to WebSphere Application Server Liberty Profile servers.

The angel process is started from the MVS console. All the WebSphere Application Server Liberty Profile servers that are running on a z/OS image can share a single angel, regardless of the level of code that the servers are running or whether they are running in a CICS JVM server.

If you are not using the CICS Liberty Security feature, you do not need to run an angel process or implement the associated security rules.

The angel process started task

The angel process started task JCL procedure is shipped with CICS in the USSHOME directory, for example:

`/usr/lpp/cicsts52/wlp/templates/zos/procs/bbgzangl.jcl`.

The JCL must be copied to a JES procedure library and modified. ROOT should be set to the value `USSHOME/wlp`, for example:

`/usr/lpp/cicsts52/wlp`.

The angel process must be running before the Liberty JVM server starts. To start or stop the angel process, issue the following operator commands:

```
START BBGZANGL
```

```
STOP BBGZANGL
```

To display the Liberty JVM servers that are connected to the angel process, issue the following operator command:

```
MODIFY BBGZANGL,DISPLAY,SERVERS
```

The returned message will list the CICS regions that have an active Liberty JVM server and other non-CICS Liberty JVM servers that are connected to the angel process.

The angel process started task SAF rules

The user ID that the angel process runs under needs the SAF STARTED profile, for example:

```
RDEF STARTED BBGZANGL.* UACC(NONE) STDATA(USER(WLPUSER))
SETROPTS RACLIST(STARTED) REFRESH
```

The user ID that the CICS Liberty JVM server runs under is the CICS region user ID. This user ID needs to be able to connect to the angel process to use authorized services. The only authorized service supported when running in a CICS JVM server are the z/OS user registry services and SAF authorization services (SAFCRED) that are implemented by the CICS Liberty security feature. If you are not using this feature, you do not need to run the angel process.

Create a SERVER profile for the angel process and permit access from the CICS region user ID. This allows the Liberty JVM server to connect to the angel process. To allow a CICS region where the region user ID is REGION1 to connect to the angel:

```
RDEF SERVER BBG.ANGEL UACC(NONE)
PERMIT BBG.ANGEL CLASS(SERVER) ACCESS(READ) ID(REGION1)
```

Create a SERVER profile for the SAF authorized user registry services and SAF authorization services (SAFCRED) and permit the CICS region user ID. This allows the Liberty JVM server to access the authorized services necessary for the CICS Liberty security feature. To allow a CICS region where the region user ID is REGION1 to use the CICS Liberty security feature:

```
RDEF SERVER BBG.AUTHMOD.BBGZSAFM.SAFCRED UACC(NONE)
PERMIT BBG.AUTHMOD.BBGZSAFM.SAFCRED CLASS(SERVER) ACCESS(READ) ID(REGION1)
```

Refresh the SERVER resource:

```
SETROPTS RACLIST(SERVER) REFRESH
```

Create a SERVER profile for the authorized module BBGZSAFM and permit the CICS region user ID (REGION1) to the profile. This action allows a Liberty server to use the z/OS Authorized services. To allow a CICS region where the region user ID is REGION1 to access the authorized module:

```
RDEF SERVER BBG.AUTHMOD.BBGZSAFM UACC(NONE)
PERMIT BBG.AUTHMOD.BBGZSAFM CLASS(SERVER) ACCESS(READ) ID(REGION1)
```

Create a SERVER profile for the IFAUSAGE services (PRODMGR) and permit the CICS region user ID. This allows the Liberty JVM server to register and unregister from IFAUSAGE when the CICS JVM server is enabled and disabled. To allow a CICS region where the region user ID is REGION1 to register and unregister from IFAUSAGE:

```
RDEF SERVER BBG.AUTHMOD.BBGZSAFM.PRODMGR UACC(NONE)
PERMIT BBG.AUTHMOD.BBGZSAFM.PRODMGR CLASS(SERVER) ACCESS(READ) ID(REGION1)
```

For more information see Liberty profile: 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. Further details on how to define application security for CICS is described in the CICS Explorer SDK. 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>
```

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, and any of the application security mechanisms supported by WebSphere Application Server Liberty profile are supported in CICS. This includes HTTP basic authentication, form login, SSL client certificate authentication or identity assertion using a custom login module or a Trust Association Interceptor (TAI). All SAF user IDs authenticated by Liberty must be granted read access to the Liberty JVM server APPLID in the APPL class. 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 APPLID as BBGZDFLT:

   ```
   RDEFINE APPL BBGZDFLT UACC(NONE)
   SETROPTS CLASSACT(APPL)
   ```

   The users must be given read access to the APPLID in order to authenticate. To allow user AUSER to authenticate against the BBGZDFLT APPLID:

   ```
   PERMIT BBGZDFLT CLASS(APPL) ACCESS(READ) ID(AUSER)
   ```

   The Liberty SAF unauthenticated user id must be given read access to the APPLID. 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 APPLID:

```
PERMIT BBGZDFLT CLASS(APPL) ACCESS(READ) ID(WSGUEST)
```

For more details refer to Liberty profile: 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, the task association data and the UEPUSID global user exits field for the XAPADMGR exit, will always be the region user ID.

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 on the Liberty profile.

Authorizing users to run applications in a Liberty JVM server

To authorize user IDs to run transactions in a Liberty JVM server, you can either use CICS transaction and resource security or you can use JEE application security roles to authorize access to JEE applications.

About this task

Using CICS security allows you to use existing security procedures but requires that individual web applications are accessed from different URIMAPs whereas using role-based security allows you to reuse existing standard JEE security definitions from another JEE application server. See “Authenticating users in a Liberty JVM server” on page 157. If you wish to use CICS authorization exclusively, you can choose to avoid further Liberty role-based checking using the special subject ALL_AUTHENTICATED_USERS in the definition of your application in server.xml. If you deploy a Liberty application in a CICS bundle, CICS automatically configures this for you.

```
<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), will allow 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.

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.

JEE application role security

JEE application role security can be configured in different ways depending upon the authorization type you wish to use. 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 SAF registry. A Liberty JVM server implicitly uses this type for authentication when the `cicsts:security-1.0` feature is installed. You can optionally choose to use it for authorization. The SAF registry type includes support for "user to role" mappings (EJB roles). As a result, the mapping information can be taken directly from the SAF registry itself.

In a Liberty JVM server, if you wish to use JEE roles without SAF authorization, then 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, thus preventing 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 JEE roles and SAF authorization, you can continue to use CICS bundles to lifecycle your web applications. The `<application-bnd>` is ignored by Liberty in favour of using the role mappings determined by the SAF registry. Role mappings are determined by virtue of a 'user' belonging to an EJB role.

Procedure

1. Add the `<safAuthorization id="saf"/>` element to your `server.xml`.
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 profile: 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)
```

Results

You can use authorize access to web applications using CICS Security and/or JEE role security by defining the roles and the users within the roles.

Configuring SSL for a Liberty JVM server

Communications are secured with the Secure Sockets Layer (SSL) protocol. The SSL protocol provides transport layer security including authenticity, data signing, and data encryption to ensure a secure connection between a client and server. You can configure a Liberty JVM server to use SSL for data encryption, and optionally authenticate with the server using a client certificate. Client certificates can be stored in a Java key store or in a SAF keyring.

About this task

Enabling SSL in a Liberty JVM server requires the ssl-1.0 Liberty feature, a key store and an HTTPS port. You have two alternative ways of enabling SSL:

- Autoconfiguring the configuration file for the Liberty profile, server.xml. CICS automatically creates and updates the `server.xml` file. Autoconfiguring will always result in the creation of a Java key store.
- 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 SAF keyring.

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

1. To use autoconfigure to configure SSL, perform the following steps:
   a. Ensure autoconfigure is enabled in the JVM profile using the JVM system property **`-Dcom.ibm.cics.jvmserver.wlp.autoconfigure`**.
   b. Set the SSL port using the **`-Dcom.ibm.cics.jvmserver.wlp.server.https.port`** in the JVM profile.
   c. Restart the JVM server.
2. To manually configure SSL:
   a. Edit the server.xml file and add the SSL feature, the key store, and set the HTTPS port. You can encrypt the Java key store password using the securityUtility tool that is provided with the Liberty profile. For more information about adding support for SSL, see Enabling SSL communication for the Liberty profile.
   b. To manually add a SAF keyring, first create the SAF keyring and import the required certificates. Edit the server.xml file to add the SSL feature and the key store and set the HTTPS port. The password field is not used for accessing the SAF keyring and must be set to"`password`". The SAF keyring must be specified in the URL `safkeyring://USERID/KeyringName` where userid is "`optional`". An example SAF keyring configuration is shown here:

```
<featureManager>
  ...
  <feature>ssl-1.0ssl-1.0</feature>
</featureManager>
...
<httpEndpoint hosthost="*" httpPort="9080" httpsPort="9443"
    id="defaultHttpEndpoint"/>
...
<keyStore fileBased="false" id="defaultKeyStore"
    location="safkeyring:///CICSKeyring"
    password="password" readOnly="true" type="JCERACFKS"/>
```

Results

You have successfully configured SSL for a Liberty JVM server.

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 CICS Explorer SDK 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. Enabling a Java security manager in Deploying describes how to set up a Java 2 security policy.
   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 `db2xxx`, 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 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 JConsole and IBM Health Center. For full details, see Using diagnostics tools in the User guide for Java Version 7 on z/OS.

For troubleshooting web applications that are running in a Liberty JVM server, see Chapter 9, "Troubleshooting Liberty JVM servers and Java web applications," on page 179.

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 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 execute 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. Add `PRINT_JVM_OPTIONS=YES` to the JVM profile. When you specify this option, 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` 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 Java applications in a JVM server in Upgrading.
   - 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. 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 176.
4. If you are getting out-of-memory errors, there might be insufficient 64-bit 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" on page 139.

   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" on page 146.
5. If you are getting encoding errors when you install or run a Java application, you might have 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 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 message DFHSJ0904 is issued and reports a problem similar to the following example, it is likely that you have hit the JVM server's thread limit:

```
Exception 'java.lang.Exception: CICSThreadExecutor: no work found for Task 45921.
The work this Task was started to perform has already timed-out.'
```

   Try to address the problem as follows:
   - 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 in Improving performance.

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" on page 168
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.

"Controlling the location for JVM stdout, stderr, JVMTRACE, and dump output" on page 170
Output from Java applications that are running in a JVM is normally written to the z/OS UNIX files that are named by the STDOUT and STDERR options in the JVM profile for the JVM. JAVADUMP files are written to the JVM's working directory on z/OS UNIX, and the more detailed Java TDUMPs are written to the file named by the JAVA_DUMP_TDUMP_PATTERN option. Most of these file names can be customized at run time to uniquely identify the JVMs that produced them. JVMTRACE specifies the name of the z/OS UNIX file to which CICS Java tracing is written during the startup and termination of a JVM server, and the running of Java applications.During application development, you can also redirect the output from the JVM and messages from JVM internals by using a Java class.

"Managing the OSGi log files of JVM servers" on page 174
The OSGi framework writes errors to a set of log files in the working directory of the JVM server. You can manage the number and size of the log files for each JVM server if the defaults are not appropriate for your environment.

"CICS component tracing for JVM servers" on page 174
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.

"Activating and managing tracing for JVM servers" on page 175
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.

"Debugging a Java application" on page 176
The JVM in CICS supports the Java Platform Debugger Architecture (JPDA), which is the standard debugging mechanism provided in the Java 2 Platform. This architecture provides a set of APIs that allow the attachment of a remote debugger to a JVM.

"The CICS JVM plugin mechanism" on page 177
In addition to the standard JPDA debug interfaces in the JVM, CICS provides a set of interception points (plugins) in the CICS Java middleware, which can be useful for debugging applications. You can use these plugins to insert additional Java programs immediately before and after the application Java code is run.

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 in Reference -> Diagnostics 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 switch on tracing to produce additional diagnostic information. The trace points for the JVM domain are listed in JVM domain trace points in Reference -> Diagnostics.

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 stdout, stderr, JVMTRACE, and dump output" on page 170 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 175 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 176 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 "plugins") 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 plugin mechanism" on page 177.

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. The *Diagnostics Guide* has information about the system properties that you can use to control the internal trace facility and to output JVM trace information to various destinations. 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.

Controlling the location for JVM stdout, stderr, JVMTRACE, and dump output

Output from Java applications that are running in a JVM is normally written to the z/OS UNIX files that are named by the STDOUT and STDERR options in the JVM profile for the JVM. JAVADUMP files are written to the JVM's working directory on z/OS UNIX, and the more detailed Java TDUMPs are written to the file named by the JAVA_DUMP_TDUMP_PATTERN option. Most of these file names can be customized at run time to uniquely identify the JVMs that produced them.
| JVMTRACE specifies the name of the z/OS UNIX file to which CICS Java tracing
| is written during the startup and termination of a JVM server, and the running of
| Java applications.During application development, you can also redirect the output from the JVM and messages from JVM internals by using a Java class.

In the standard setup for a CICS JVM, the file that is named by the STDOUT option in the JVM profile is used for System.out requests, and the file that is named by the STDERR option is used for System.err requests. The output files are z/OS UNIX files in the working directory that is named by the WORK_DIR option in the JVM profile.

You can specify a fixed file name for the `stdout` and `stderr` files. 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, and the output from different JVMs is interleaved with no record headers. This situation is not helpful for problem determination.

A better choice is to specify a variable file name for the `stdout` and `stderr` files. The files can then be made unique to each individual JVM during the lifetime of the CICS region. You can also include additional identifying information.
- You can include the CICS region applid in the file name by using the &APPLID; symbol.

Other identifying information in file names includes the &DATE; and &TIME; symbols.

&DATE; is replaced by the current date in the form Dyymmdd

&TIME; is replaced by the current time in the format Thhmmss.

If you want to create unique output or dump files for each JVM server, use the &JVMSERVER; symbol. When you use this symbol, the name of the JVMSERVER resource is substituted at run time.

The 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, and you cannot customize the names for these files.

| If you do not set a value for the JVMTRACE option, CICS automatically creates
| unique trace files for each JVM server. CICS uses the &APPLID; and
| &JVMSERVER; symbols, and the date and time stamp when the JVM server
| started. This file is created in the directory that is specified by the WORK_DIR
| option.

TDUMPs output from the JVM, which contain more detailed memory dump output, including the JVM address space, are written to a data set destination. The name of the destination is specified by the JAVA_DUMP_TDUMP_PATTERN option in the JVM profile. You can use the &APPLID;, &DATE;, and &TIME; 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 its `stderr` file 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.

During application development, you can use the `USEROUTPUTCLASS` option in a JVM profile to name a Java class that intercepts and redirects the output from the JVM and messages from JVM internals. 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. Specifying this option has a negative effect on the performance of JVMs, so it should not be used in a production environment.

"Redirecting JVM stdout and stderr output"
During application development, the `USEROUTPUTCLASS` option can be used by developers to separate out their own JVM stdout and stderr output in a CICS region, and direct it 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.

"The sample classes com.ibm.cics.samples.SJMergedStream and com.ibm.cics.samples.SJTaskStream" on page 172
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 to create a log that correlates JVM activity with CICS activity.

"Control of Java dump options" on page 173
The -Xdump option can be used in a JVM profile to specify dump options to the JVM.

Redirecting JVM stdout and stderr output

During application development, the `USEROUTPUTCLASS` option can be used by developers to separate out their own JVM stdout and stderr output in a CICS region, and direct it 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 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 named by the `STDOUT` and `STDERR` options in the JVM profile are still used for some messages 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.

The source for the supplied user output classes is provided as samples, so you can modify the classes as you want, or write your own classes based on the samples.

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" on page 133.

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 to create a log 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 like this.

The com.ibm.cics.samples.SJMergedStream class directs output from the JVM to the transient data queues CSJO (for stdout output), and CSJE (for stderr output and internal messages). These transient data queues are supplied in group DFHDCTG, and they are indirected to CSSL, but you can redefine them if required.

As well as redirecting the output, the class adds a header to each record containing 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/cicsts52/lib`, where `/usr/lpp/cicsts/cicsts52` 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. You must take special steps to enable these samples in an OSGi environment. For more information, see "Writing Java classes to redirect JVM stdout and stderr output" on page 133.

For Java applications executing on threads other than those attached by CICS, which are not able to make CICS requests, the output from the JVM cannot be redirected 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 described above, or if these files are unavailable, 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, and add time stamps and headers, to provide output streams that are specific to a single task. The CICS-supplied sample class, com.ibm.cics.samples.SJTaskStream is set up to do this. The class directs the output for each task to two z/OS UNIX files, one for stdout output and one for stderr output, that are uniquely named using a task number (in the format `YYYYMMDD.task.tasknumber`). The z/OS UNIX files are stored in the `stdout` directory for stdout output, or `stderr` directory for stderr output. The process is the same for both Java applications executing on threads attached by CICS, and Java applications that are executing on other threads.

Error handling

The length of messages issued by the JVM can vary, and the maximum record length for the CSSL queue (133 bytes) might not be sufficient to contain some of the messages you receive. If this happens, the sample output redirection class issues an error message, and 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 in order to view the output. For information about how to define transient data queues, see TDQUEUE resources in Reference -> System definition. 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 specified on the `WORK_DIR` option in the JVM profile, and *applid* is the APPLID identifier 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 issued. If the error occurred while processing an output message, then the error messages are directed to System.err, and are eligible for redirection. However, if the error occurred while processing an error message, then the new error messages are sent to the file named by the `STDERR` option in the JVM profile. This avoids 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.

Managing the OSGi log files of JVM servers

The OSGi framework writes errors to a set of log files in the working directory of the JVM server. You can manage the number and size of the log files for each JVM server if the defaults are not appropriate for your environment.

About this task

The OSGi framework writes errors to a log file in the `$WORK_DIR/applid/jvmserver/configuration` directory on zFS, where *$WORK_DIR* is the working directory of the JVM server, *applid* is the CICS APPLID, and *jvmserver* is the name of the JVMSERVER resource. If you do not set a value for *$WORK_DIR* in the JVM profile, the logs are output to the z/OS UNIX home directory for the CICS region.

The OSGi framework continues to write to the log file until it reaches 1000 KB in size. After this, the OSGi framework creates another log file to write out further error messages. You can have up to ten log files in the directory. After the tenth log file is full, the OSGi framework writes over the oldest log file. Each JVM server can therefore have up to 10,000 KB of storage allocated to log files in zFS.

You can add options to the JVM profile to change the number and size of log files that are used by the OSGi framework to reduce or increase the number of files and the storage usage.

Procedure

- To change the maximum number of log files, add the **`eclipse.log.backup.max`** parameter to the JVM profile.
- To change the maximum size of each log file, add the **`eclipse.log.size.max`** parameter to the JVM profile.

Example

The following example shows a JVM profile with the two parameters specified. In this example, the OSGi framework can use up to five log files and each log file has a maximum size of 500 KB.

```
#Parameters to control the number and size of OSGi logs
#
-Declipse.log.backup.max=5
-Declipse.log.size.max=500
#
#
```

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 in Reference -> Diagnostics.

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 and 4 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 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."

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 *`applid.jvmserver.`*`dfhjvmtrc` and is created by CICS in the working directory of the JVM 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 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. View the tracing results in the *`applid.jvmserver.`*`dfhjvmtrc` file. 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. To manage the size of the file, you can delete old entries. If you disable the JVMSERVER resource, you can delete the file or rename the file if you want to

retain the information separately. When you enable the JVMSERVER resource, CICS appends trace entries to the trace file if it already exists and creates a file in zFS if a trace file does not exist.

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 2 Platform. This architecture provides a set of APIs that allow the attachment of a remote debugger to a JVM.

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.

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` however this can cause the JVM server to lock when the debugging agent attempts to connect. Specifying a suspend value of `n` should prevent the JVM server from locking.
2. 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.
3. 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. The JVM suspends execution; the Java application has not started.
4. Set a breakpoint at a suitable point in the Java application by specifying the full Java class name and source code line number. Because the application class is generally not yet loaded, the debugger indicates that activation of this breakpoint is deferred until the class is loaded. Let the JVM run through the CICS middleware code to the application breakpoint, at which point it suspends execution again.
5. Examine the loaded classes and variables and set further breakpoints to step through the code as required.
6. 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 plugin mechanism

In addition to the standard JPDA debug interfaces in the JVM, CICS provides a set of interception points (plugins) in the CICS Java middleware, which can be useful for debugging applications. You can use these plugins 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 plugins. The plugins 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 plugins 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 plugin that provides methods that are called immediately before and after a Java program is invoked.

To deploy a plugin, you package the plugin as an OSGi bundle. For more information see "Deploying OSGi bundles in a JVM server" on page 117.

Two Java programming interfaces are provided.

Both interfaces are supplied in `com.ibm.cics.server.jar`, and are documented in the Javadoc. For more information see "The Java class library for CICS (JCICS)" on page 35.

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 plugin 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 plugin 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 plugin initialiser
    public void init()
    {
        // This method is called when the CICS Java middleware loads and
        // registers the plugin. 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)
    {
    }
}
```

Chapter 9. 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 the Liberty profile 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 profile technology that is used to run web applications also produces diagnostics that are available in zFS. For general setup errors and application problems, see Chapter 8, "Troubleshooting Java applications," on page 165.

Avoiding problems

CICS uses the values of the region APPLID and the JVMSERVER resource name to create unique file and directory names on the zFS file system. 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. 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:

```
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. Use the CICS messages and any errors in 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 the Liberty profile server. If you set any other value, the JVM server fails to start.

Web application is not available after it is deployed to the dropins directory

1. 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`. This error message indicates that the port number or host name that are specified are incorrect. The host name might be invalid or the port number might be in use.

CICS CPU is increased after a Liberty JVM server is enabled

1. The Liberty JVM server is scanning the `dropins` directory too frequently and causing too much I/O and CPU usage. The frequency that the Liberty JVM server scans the `dropins` directory for applications is configurable. The default interval that is supplied in the configuration templates is 5 seconds, but you can increase this value or disable the application scan.

   To fix this problem, edit the following XML in the `server.xml` file:

```
<config monitorInterval="5s" updateTrigger="polled"/>
<applicationMonitor updateTrigger="disabled" pollingRate="5s" dropins="dropins" dropinsEnabled="false"/>
```

If you install your web applications in CICS bundles, do not disable polling in the <config> element, but disable the application scan. For information on how to edit this setting, see Controlling dynamic updates.

Application not available

1. 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 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 configuration information. Any configuration errors are reported in `dfhjvmerr` and might provide some useful information. If you are using CICS security, check that the CICS security feature is specified. 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. If you receive an HTTP 403 authorization failure, 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 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 is using the correct transaction.
3. If the URIMAP is using the correct transaction, 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, you can check the status of the BUNDLE resource.
3. The request might have been purged before it completed. The errors messages in the log describe why the request was purged.

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 the `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

If the Liberty server 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.

Using the productInfo script to verify integrity of the Liberty profile

You can verify the integrity of the Liberty profile 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 profile 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 profile 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* as specified in the JVM profile and is called *`APPLID.JVMSERVER`*`.wlpenv`, where *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 profile command as an argument, for example:

```
./CICSPRD.DFHWLP.wlpenv productInfo version
./CICSPRD.DFHWLP.wlpenv server dump --archive=package_file_name.dump.pax  --include=heap
```

Note that 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 the Liberty commands, see Liberty profile: productInfo command and Generating a Liberty profile server dump from the command prompt

Notices

This information was developed for products and services offered in the U.S.A. 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 right 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 give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (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

The following paragraph does not apply in the United Kingdom or any other country where such provisions are inconsistent with local law:

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 states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This publication 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.

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 United Kingdom Laboratories, MP151, Hursley Park, Winchester, Hampshire, England, SO21 2JN.

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 Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Privacy Policy Considerations

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 is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

CICSPlex SM Web User Interface :

For the WUI 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 personally identifiable information for purposes of session management, authentication, enhanced user usability, or other usage tracking or functional purposes. These cookies cannot be disabled.

For the WUI Data Interface: Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user's user name and other personally identifiable information for purposes of session management, authentication, or other usage tracking or functional purposes. These cookies cannot be disabled.

For the WUI Hello World page: Depending upon the configurations deployed, this Software Offering may use session cookies that collect no personally identifiable information. 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 personally identifiable information 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's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www-01.ibm.com/software/info/product-privacy/.

“Trademarks”

Trademarks

Bibliography

CICS books for CICS Transaction Server for z/OS

General

CICS Transaction Server for z/OS Program Directory, GI13-3326
CICS Transaction Server for z/OS What's New, GC34-7302
CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.1, GC34-7296
CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.2, GC34-7297
CICS Transaction Server for z/OS Upgrading from CICS TS Version 4.1, GC34-7298
CICS Transaction Server for z/OS Upgrading from CICS TS Version 4.2, GC34-7299
CICS Transaction Server for z/OS Upgrading from CICS TS Version 5.1, GC34-7300
CICS Transaction Server for z/OS Installation Guide, GC34-7279

Access to CICS

CICS Internet Guide, SC34-7281

CICS Web Services Guide, SC34-7301

Administration

CICS System Definition Guide, SC34-7293
CICS Customization Guide, SC34-7269
CICS Resource Definition Guide, SC34-7290
CICS Operations and Utilities Guide, SC34-7285
CICS RACF Security Guide, SC34-7288
CICS Supplied Transactions, SC34-7292

Programming

CICS Application Programming Guide, SC34-7266
CICS Application Programming Reference, SC34-7267
CICS System Programming Reference, SC34-7294
CICS Front End Programming Interface User's Guide, SC34-7277
CICS C++ OO Class Libraries, SC34-7270
CICS Distributed Transaction Programming Guide, SC34-7275
CICS Business Transaction Services, SC34-7268
Java Applications in CICS, SC34-7282

Diagnosis

CICS Problem Determination Guide, GC34-7287
CICS Performance Guide, SC34-7286
CICS Messages and Codes Vol 1, GC34-7283
CICS Messages and Codes Vol 2, GC34-7284
CICS Diagnosis Reference, GC34-7274
CICS Recovery and Restart Guide, SC34-7289
CICS Data Areas, GC34-7271
CICS Trace Entries, SC34-7295
CICS Debugging Tools Interfaces Reference, GC34-7273

Communication

CICS Intercommunication Guide, SC34-7280
CICS External Interfaces Guide, SC34-7276

Databases

CICS DB2 Guide, SC34-7272

CICS IMS Database Control Guide, SC34-7278

CICS Shared Data Tables Guide, SC34-7291

CICSPlex SM books for CICS Transaction Server for z/OS

General

CICSPlex SM Concepts and Planning, SC34-7306
CICSPlex SM Web User Interface Guide, SC34-7316

Administration and Management

CICSPlex SM Administration, SC34-7303
CICSPlex SM Operations Views Reference, SC34-7312
CICSPlex SM Monitor Views Reference, SC34-7311
CICSPlex SM Managing Workloads, SC34-7309
CICSPlex SM Managing Resource Usage, SC34-7308
CICSPlex SM Managing Business Applications, SC34-7307

Programming

CICSPlex SM Application Programming Guide, SC34-7304
CICSPlex SM Application Programming Reference, SC34-7305

Diagnosis

CICSPlex SM Resource Tables Reference Vol 1, SC34-7314
CICSPlex SM Resource Tables Reference Vol 2, SC34-7315
CICSPlex SM Messages and Codes, GC34-7310
CICSPlex SM Problem Determination, GC34-7313

Other CICS publications

The following publications contain further information about CICS, but are not provided as part of CICS Transaction Server for z/OS, Version 5 Release 2.

Designing and Programming CICS Applications, SR23-9692

CICS Application Migration Aid Guide, SC33-0768

CICS Family: API Structure, SC33-1007

CICS Family: Client/Server Programming, SC33-1435

CICS Family: Interproduct Communication, SC34-6853

CICS Family: Communicating from CICS on System/390, SC34-6854

CICS Transaction Gateway for z/OS Administration, SC34-5528

CICS Family: General Information, GC33-0155

CICS 4.1 Sample Applications Guide, SC33-1173

CICS/ESA 3.3 XRF Guide , SC33-0661

Other IBM publications

The following publications contain information about related IBM products.
IBM Developer Kit and Runtime Environment, Java 2 Technology Edition Diagnostics Guide, SC34-6358
Persistent Reusable Java Virtual Machine User's Guide, SC34-6201

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully.

You can perform most tasks required to set up, run, and maintain your CICS system in one of these ways:
- using a 3270 emulator logged on to CICS
- using a 3270 emulator logged on to TSO
- using a 3270 emulator as an MVS system console

IBM Personal Communications provides 3270 emulation with accessibility features for people with disabilities. You can use this product to provide the accessibility features you need in your CICS system.

Index

Readers' Comments — We'd Like to Hear from You

CICS Transaction Server for z/OS
Version 5 Release 2
Java Applications in CICS

Publication No. SC34-7282-00

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:
- Send your comments to the address on the reverse side of this form.
- Send a fax to the following number: +44 1962 816151
- Send your comments via email to: idrcf@uk.ibm.com

If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	Email address

Readers' Comments — We'd Like to Hear from You
SC34-7282-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM United Kingdom Limited
User Technologies Department (MP189)
Hursley Park
Winchester
Hampshire
United Kingdom
SO21 2JN

Fold and Tape **Please do not staple** Fold and Tape

SC34-7282-00

Cut or Fold
Along Line

SC34-7282-00