Redbooks
ibm.com/redbooks

IBM CICS and Liberty: What You Need to Know

Hernan Cunico
Andreas Hümmer
Jonathan Lawrence
Shayla Robinson
Andre Schreiber
Inderpal Singh
Prabhat Srivastava
Phil Wakelin
Dan Zachary

z Systems

IBM

Redbooks

International Technical Support Organization

IBM CICS and Liberty: What You Need to Know

January 2016

SG24-8335-00

Note: Before using this information and the product it supports, read the information in “Notices” on page ix.

First Edition (January 2016)

This edition applies to V5R3M0 of CICS Transaction Server (product number 5655-Y04).

Contents

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property 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 grant you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, MD-NC119, Armonk, NY 10504-1785, US

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided “AS IS”, without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks or registered trademarks of International Business Machines Corporation, and might also be trademarks or registered trademarks in other countries.

CICS®
CICS Explorer®
CICSPlex®
DB2®
IBM®
IBM z Systems™
Language Environment®
MVS™
OMEGAMON®
RACF®
Redbooks®
Redbooks (logo)  ®
System z®
WebSphere®
z Systems™
z/OS®

The following terms are trademarks of other companies:

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Find and read thousands of IBM Redbooks publications

- Search, bookmark, save and organize favorites
- Get personalized notifications of new content
- Link to the latest Redbooks blogs and videos

Get the latest version of the Redbooks Mobile App

iOS **Download Now** Android

Promote your business in an IBM Redbooks publication

Place a Sponsorship Promotion in an IBM® Redbooks® publication, featuring your business or solution with a link to your web site.

Qualified IBM Business Partners may place a full page promotion in the most popular Redbooks publications. Imagine the power of being seen by users who download millions of Redbooks publications each year!

ibm.com/Redbooks
About Redbooks → Business Partner Programs

THIS PAGE INTENTIONALLY LEFT BLANK

Preface

This IBM® Redbooks® publication, intended for architects, application developers, and system programmers, describes how to design and implement Java web-based applications in an IBM CICS® Liberty JVM server. This book is based on IBM CICS Transaction Server V5.3 (CICS TS) using the embedded IBM WebSphere® Application Server Liberty V8.5.5 technology.

Liberty is an asset to your organization, whether you intend to extend existing enterprise services hosted in CICS, or develop new web-based applications supporting new lines of business. Fundamentally, Liberty is a composable, dynamic profile of IBM WebSphere Application Server that enables you to provision Java EE technology on a feature-by-feature basis. Liberty can be provisioned with as little as the HTTP transport and a servlet web container, or with the entire Java EE 6 Web Profile feature set depending on your application requirements.

This publication includes a Technology Essentials section for architects and application developers to help understand the underlying technology, an Up-and-Running section for system programmers implementing the Liberty JVM server for the first time, and a set of real-life application development scenarios.

Authors

This book was produced by a team of specialists from around the world working at the International Technical Support Organization (ITSO), Raleigh Center.

Hernan Cunico is a Senior Information and IT Architect at the ITSO, Raleigh Center. He has over 18 years of experience in the consulting and development fields. His areas of expertise include user experience design, information and application development, middleware, portals, open source, cloud, mobile, and Agile development. He has written extensively on commerce, portals, migrations, HR, and learning solutions.

Andreas Hümmer has been working as a Software Developer for IBM z Systems at DATEV eG in Germany since 2009. During his apprenticeship, he visited the IT-Akademie Augsburg (now European Mainframe Academie) and graduated as a z Systems™ specialist. After that, he joined a team at DATEV, which is responsible for development tools and languages on IBM z/OS®. The main area of his work is the integration of Java in z/OS Batch and CICS environments, including interlanguage communication to legacy code. Besides his job as a software developer, he supports an in-house network for young talents (under 30) working on z Systems.

Jonathan Lawrence is a member of the CICS Level 3 service team at IBM Hursley in the UK. He has over 15 years of experience working with CICS, as a designer, developer, technical consultant, and service engineer. In his service role, he specializes in Java, Open Services Gateway initiative (OSGi), Liberty and Dynamic Scripting, as well as CICS integration technologies such as Web services, IBM MQ, and Link3270 Bridge.

Shayla Robinson is a Software Support Specialist in CPSM Level II Technical Support at IBM in Research Triangle Park, NC. She has worked for IBM for 18 years as a CICS Level 2 Support representative with the last eight years focusing on CPSM Level 2 Support. She holds a Bachelor's degree in Computer Science and a Master's degree in Information Systems. In addition to CICS and IBM CICSPlex® SM, her areas of expertise include Java, OSGi, and Liberty.

Andre Schreiber is a System Programmer at Sparda-Datenverarbeitung eG in Nuremberg, Germany. After graduation as an IT systems specialist, he joined the Mainframe department as a CICS system programmer. He has three years of experience in CICS Transaction Server, CICS Transaction Gateway on z/OS, and IBM OMEGAMON® XE on z/OS, with a special focus on the integration of new workloads (especially Java) inside of CICS. In his role, he specializes in Liberty, OSGi, and Axis2 Java based application server in CICS.

Inderpal Singh joined the IBM CICS Transaction Server development team in 2009. He has worked as a software engineer in many areas, including CICS System Management, CICS cloud enablement, and the CICS Transaction Server Feature Pack for Mobile Extensions. His most recent role is as an evangelist for modernization of clients' IBM z Systems™ assets, where he focuses on helping clients of z Systems achieve the most business value out of the technology. After studying at The University of Sheffield, Inderpal received a Bachelor of Science degree in Computer Science and Artificial Intelligence.

Prabhat Srivastava is a System Programmer at Bankwest, Perth, Western Australia. He has 25 years of experience in mainframe systems programming area. His areas of interest are z/OS, CICS TS, security, mainframe modernization, and web services. Recently, he successfully completed a project to extend Bankwest's core CICS applications to a digital world by using CICS Liberty and is currently working on CICS web services.

Phil Wakelin works for CICS development at IBM UK in Hursley, and is a member of the CICS strategy and design team. He has worked with many CICS technologies for the last 25 years, and is responsible for new functionality in the area of CICS Java support. He is the author of many white papers, SupportPacs, and IBM Redbooks publications in the areas of CICS integration and Java support.

Dan Zachary works for IBM with CICS support in Research Triangle Park, NC. He has 26 years of experience supporting CICS clients throughout the world. His favorite areas of CICS are performance, languages, kernel, dispatcher, storage, monitoring, external resource managers and, now, Liberty JVM server.

Thanks to the following people for their contributions to this project:

- Abigail Bettle
- Alexander Brown
- Adam Coulthard
- Mark Hollands
- David Roberts
- Melita Saville
- Matthew Wilson

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks
- Send your comments in an email to:

  redbooks@us.ibm.com
- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks
- Follow us on Twitter:

  https://twitter.com/ibmredbooks
- Look for us on LinkedIn:

  http://www.linkedin.com/groups?home=&gid=2130806
- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm
- Stay current on recent Redbooks publications with RSS Feeds:

  http://www.redbooks.ibm.com/rss.html

1

Introduction

Our goal with this book is to provide the information necessary for you to understand and take advantage of IBM WebSphere Application Server V8.5.5 Liberty (Liberty) within IBM CICS Transaction Server (CICS TS) V5.3. You will find the use of Liberty advantageous whether you intend to extend existing enterprise services hosted in CICS, or intend to develop new services supporting new lines of business.

The following topics are covered in this chapter:

- What to expect from this book
- What is Liberty?
- Benefits of running web applications in CICS
- What's new in Liberty for CICS TS V5.3

1.1 What to expect from this book

This book provides a technical introduction to many of the concepts surrounding Liberty, including Java Enterprise Edition (Java EE) and the web profile. It then describes the steps necessary to develop, configure, and deploy web applications into the Liberty server within IBM CICS.

1.1.1 Intended audiences

This book provides information for a wide variety of roles from architects to application developers and CICS system programmers, and assumes that you are familiar with the basics concepts of CICS:

- Architects should review this chapter and all of the chapters in the technology essentials section Part 1, “Technology essentials” on page 15.
- Application developers should refer to:
  - Chapter 2, “Application development” on page 17
  - Chapter 5, “Developing and deploying applications” on page 69
  - Part 3, “Scenarios” on page 159
- System programmers should refer to Part 2, “Up and running” on page 67.

1.1.2 Scenarios

With this book, you can find example code to quickly get up and running with Java EE 6 web applications and Liberty. Our main scenarios are based on the CICS general insurance application (GENAPP) Support Pack (CB12), which you can download and try for yourself by following examples in this book. Download it from the following website:

`http://www.ibm.com/support/docview.wss?uid=swg24031760`

1.2 What is Liberty?

Fundamentally, Liberty is a modular implementation (or profile) of WebSphere Application Server technology. It is intended to provide a lighter weight and more dynamic runtime than the classic WebSphere Application Server runtime. It provides support for most of the Java Platform, Enterprise Edition technology previously supported in WebSphere Application Server, but in a composable runtime using configurable components termed features. Liberty is provided with WebSphere Application Server V.8.5 onwards, and runs on various distributed platforms as well as on z/OS. It is also available for developers as a download from the `wasdev.net` website.

A Liberty server can be provisioned with as little as the WebSphere Application Server kernel, web container, and HTTP transport features for a servlet engine. If you require access to a database, Liberty can dynamically initialize the Java Database Connectivity (JDBC) feature, or if you require a RESTful interface, the JAX-RS feature could be initialized. The approach of being able to select the features you require allows Liberty to initialize quickly with a basic web application and have the smallest footprint as possible in the system. The initialization of features and applications is achieved dynamically, meaning you are not usually required to restart your Liberty server to add features. This can be particularly powerful in development environments where developers need code changes to be reflected easily and immediately.

Liberty is built by using Open Services Gateway initiative (OSGi) technology and concepts. The fit-for-purpose nature of the run time relies on the dynamic behavior inherent in the OSGi framework and service registry. As bundles are installed to or uninstalled from the framework, the services that each bundle provides are added or removed from the service registry. The addition and removal of services similarly cascades to other dependent services. The result is a dynamic, composable run time that can be provisioned with only what your application requires and responds dynamically to configuration changes as your application evolves.

The Liberty server in CICS is supplied with and licensed through the installation of CICS TS for z/OS. It supplies the same Liberty technology as supplied with the IBM WebSphere Application Server Liberty products, but with a set of CICS specific user extensions, which provide for the integration with the CICS runtime.

1.2.1 Java EE 6

With Java EE 6 came the delivery of the Java EE 6 Web Profile. Much like the story of WebSphere Application Server and Liberty, until the introduction of the profiles, the Java EE 6 Full Platform was a run time that implemented as many application use cases possible. This approach of providing a large set of APIs meant that Java Platform, Enterprise Edition application servers need to provide the full set of Java Platform, Enterprise Edition APIs, although most applications would use only a small subset.

The introduction of profiles in Java EE 6 alleviates this issue by defining smaller subsets of APIs that are focused on particular styles of application. The first such profile is the Java EE 6 Web Profile. The Java EE 6 Web Profile is a subset of Java EE 6 APIs that are targeted to the creation of web applications.

Liberty inside of CICS TS V5.3 supports the Java EE 6 Web Profile and provides partial support for Java EE 6 full platform. Table 1-1 shows the relationship between the supported features of Liberty inside CICS and the corresponding Java Specification Request (JSR) implemented in the profile. A JSR is a request to develop a new specification or the revision of an existing specification that is delivered as part of Java. A list of all proposed and final JSRs can be found at the Java Community Process website:

https://www.jcp.org/en/home/index

Table 1-1 Liberty-supported features and corresponding JSRs

	Liberty feature	**JSR**	**Description**
Java EE 6 Web Profile	beanValidation-1.0	JSR 303	Bean Validation
	cdi-1.0	JSR 299	Contexts and Dependency Injection
	ejbLite-3.1	JSR 318	EJB Lite subset of the EJB 3.1 specification
	jdbc-4.0	JSR 221	Java Database Connector API
	jpa-2.0	JSR 317	Java Persistence 2.0
	jndi-1.0	JSR 316	Java Naming and Directory Interface
	jsf-2.0	JSR 314	JavaServer Faces 2.0

	Liberty feature	JSR	Description
	jsp-2.2	JSR 245	JavaServer Pages 2.2
	servlet-3.0	JSR 315	Java Servlet 3.0
	webProfile-6.0	JSR 342	Java EE 6 Web Profile
Java EE 6 full platform	jaxb-2.2	JSR 222	Java Architecture for XML Binding 2.0
	jaxrs-1.1	JSR 311	Java API for RESTful Web Services 1.1
	jaxws-2.2	JSR 224	Java API for XML-Based Web Services 2.2
	jca-1.6	JSR 322	Java EE Connector Architecture 1.6
	jms-1.1	JSR 914	Java Message Service API 1.1

1.2.2 Java EE 7

Java EE 7 is the new evolution of the Java Enterprise Edition standard that embraces the latest standards such as HTML5 and web sockets, as well as removing inconsistencies in the existing standard and providing updates for most of the components. It is being rapidly adopted by the Java Platform, Enterprise Edition application server marketplace and at the time of writing is supported by the WebSphere Application Server Liberty FixPack6, and has been announced for support in classic WebSphere Application Server. For more information, see the following announcement letter:

http://www.ibm.com/common/ssi/ShowDoc.wss?docURL=/common/ssi/rep_ca/4/897/ENUS215-084/index.html

Building on this position, IBM announced the following statement covering upcoming Java EE 7 support for CICS TS V5.3: "*IBM intends to deliver support for Java applications that exploit Java EE 7 Full Platform features when running in the WebSphere Liberty profile that is integrated with IBM CICS Transaction Server for z/OS (CICS TS).*"

For more information, see the following announcement letter:

http://www.ibm.com/common/ssi/cgi-bin/ssialias?infotype=an&subtype=ca&supplier=897&letternum=ENUS215-363

1.2.3 Additional resources

More resources are available to support development with Liberty. The following list provides developer-focused community resources:

- WebSphere Application Server Liberty V8.5.5 IBM Knowledge Center

  http://www.ibm.com/support/knowledgecenter/SS7K4U_8.5.5///com.ibm.websphere.wlp.zseries.doc/ae/rwlp_prog_model_support_javaee6.html

- Java Platform, Enterprise Edition 6 (Java EE 6) Specification

  https://jcp.org/en/jsr/detail?id=316

- WASdev community

  http://wasdev.net

  The WASdev community is a hub for information about developing applications for WebSphere Application Server, and using Liberty in particular. Articles, podcasts, videos, and samples are refreshed regularly.

- WASdev forum

  https://developer.ibm.com/wasdev

  The WASdev forum provides an opportunity for users of Liberty to interact with each other and with the IBM developers working on the product.

- Stack Overflow tags

  http://stackoverflow.com/questions/tagged/websphere-liberty

  Stack Overflow is a Q&A site that is no cost to use. It allows users to ask and answer questions with built-in incentives to reward high-quality answers. Use the websphere-liberty tags to ask or answers questions about Liberty.

1.3 Benefits of running web applications in CICS

So, what are the benefits of running web applications using Liberty in CICS? There are many ways of analyzing this issue but stepping back, there are four key benefits that running web applications in CICS provides: Skills, integration, performance, and cost reduction.

1.3.1 Skills

An important benefit of Liberty in CICS is that it opens to CICS a whole new world of application development. The direction of training and skills growth today is moving towards Java and Java EE web applications. With Liberty in CICS, you can develop Java EE web applications in an integrated development environment (IDE) and then deploy them to CICS. This allows Java web developers to participate in developing, extending, and updating business applications for CICS. For instance, they can use a familiar Java API like JAX-RS to build a new service interface for invoking existing business application programs. They can use JSP technology or servlets to update the presentation interface for CICS transactions, and they can use other features in the Java EE 6 framework to build new business logic applications.

1.3.2 Integration

Integrating Java applications with existing CICS applications has been recognized by many customers as a cost-effective way of modernizing business applications by exploiting the mixed language application serving environment offered by CICS. Java components can run alongside existing COBOL, PL/I, and assembler applications, with CICS providing the runtime integration, without having to provide a dedicated new application server for the Java components.

1.3.3 Performance

An inherent benefit of running your Java EE web applications in a Liberty Java virtual machine (JVM) Server in CICS is the simplicity and speed of accessing CICS resources. A request from a web client enters Liberty and runs on a Java thread just like on any Java EE web server. However, the Liberty server is running within a JVM server in CICS. This results in the Java thread being integrated with CICS in such a way that the request also runs as a CICS transaction.

This means that access to CICS resources, such as a database or file, are highly optimized. There is no need for requests to travel through an adapter and across a network because the transaction has already begun. Put simply, it is generally faster to run your web application as close as possible to your data.

1.3.4 Cost reduction

All Java applications on z/OS can benefit from the price advantages of IBM z Systems specialty processors (IBM System z® Application Assist Processor (zAAP) or IBM System z Integrated Information Processor (zIIP)). Specialty processors provide a way to lower overall z Systems costs both in terms of hardware capital and in terms of software-based charging. Put simply, any Java code executes on a speciality engine. For Liberty web applications, this includes most of the Liberty server runtime, and any servlet or web-based components. However, any calls to CICS either via JCICS, JDBC type 2 drivers, or JCA local ECI will only run on a zIIP during the initial Java phase, as once within CICS workload will not be zIIP eligible.

So Liberty in CICS offers the best of all worlds. It is Liberty, a Java EE application server capable of hosting web applications built using Java EE 6 features and it is CICS, a familiar and robust transaction processing server providing close integration with your data. This combination offers opportunities to modernize presentation logic, leverage growing Java development skills, integrate legacy business processes with new interfaces, simplify and speed access to your data, and gain financially from running Java applications on zAAP processors.

The following subsections introduce three scenarios that illustrate ways that you can benefit from running web applications in a Liberty JVM server in CICS. In later chapters, each scenario is detailed further.

- Porting a web application to Liberty in CICS
- Creating an integration logic interface for existing CICS business logic
- Writing new Java business logic applications

1.3.5 Porting a web application

Web applications written by using the features of the Java EE 6 Web Profile subset can be hosted in CICS TS V5.3. This includes all of the core web container APIs, such as JSP and servlets. It also includes many of the common Java EE API and capabilities such as JAX-RS and JAX-WS and JDBC.

You might consider deploying existing web applications to CICS TS V5.3 for reasons of convenience, consolidation, or flexibility. However, a particularly compelling reason to make that move is for those applications that use the JCA ECI resource adapter to invoke a program on CICS TS. Figure 1-1 on page 7 shows a web application hosted outside of CICS and using JCA with CICS Transaction Gateway (CICS TG) to invoke a program on CICS TS.

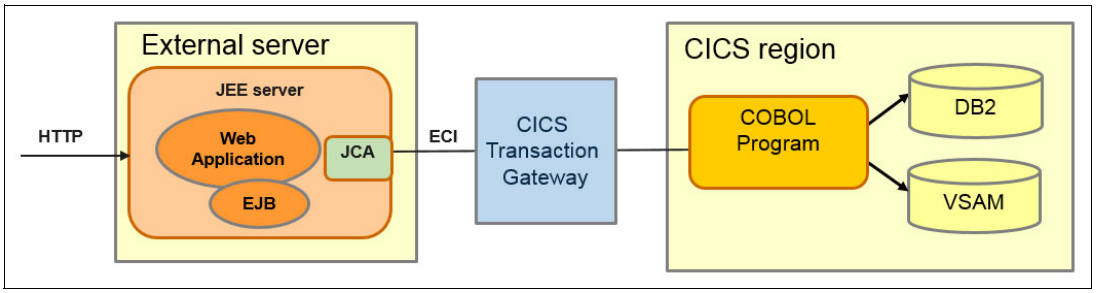


Figure 1-1 JCA-based web application using CICS TG

The JCA ECI call shown in Figure 1-1 involves leaving the Java EE application server and going out through the JCA resource adapter via a network to the CICS Transaction Gateway and then on to a connected CICS region. If there are many such calls to CICS applications within the web application, it might make more sense to move the entire web application to the Liberty JVM server on CICS TS.

Figure 1-2 summarizes how the application might look after the web application is ported to a Liberty JVM server in CICS TS V5.3.

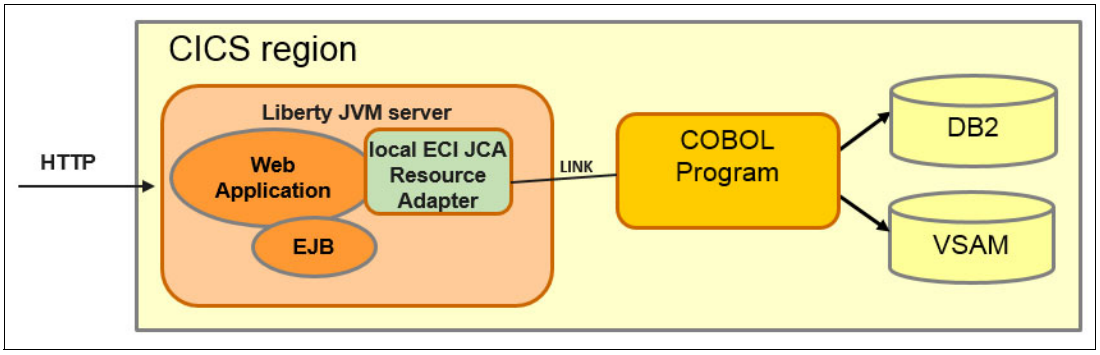


Figure 1-2 JCA-based web application running in CICS

Figure 1-1 and Figure 1-2 show how you can consolidate the two halves of this type of web application. The front-end web application runs in a Liberty JVM server in the same CICS region as the program invoked through JCA. This “lift and shift” consolidation has the potential to improve the performance of the web application by eliminating the network latency. Improved performance is more likely to be realized in cases where the web application makes frequent calls to the CICS COBOL program, or large amounts of data are transferred.

In Figure 1-2, notice that JCA straddles the boundary between Liberty and the JVM server and CICS. This represents that the CICS implementation of the cicsts:JcaLocalEci-1.0 feature treats the JCA ECI call in much the same manner as a local JCICS Program.link(). This technique ensures that the call is highly optimized as well as provides the opportunity for the Liberty JVM server to become a front-end liberty-owning region (LOR). The target programs of the ECI calls could all be defined as dynamic and routed to a series of back-end application-owning regions (AORs) using CICSPlex SM dynamic routing.

In addition to JCA ECI resource adapter calls, there are other ways that a web application can invoke business logic applications in CICS TS. These include web requests via HTTP, web services calls using XML and SOAP, and Java Message Service (JMS) applications. In many cases, web applications that use these methods of communicating with CICS TS can also be ported to a Liberty JVM server in CICS TS with little or no modification.

1.3.6 Creating an integration logic interface

A big benefit of Liberty in CICS is being able to take advantage of the existing and growing pool of Java development skills. For example, those skills can be put to use developing a RESTful (JAX-RS) or web service (JAX-WS) Java application that provides a new service interface to an existing business logic program in CICS TS. Such an application can be deployed directly into a Liberty JVM server collocated with the business logic program in the same CICS region. Refer to Figure 1-3.

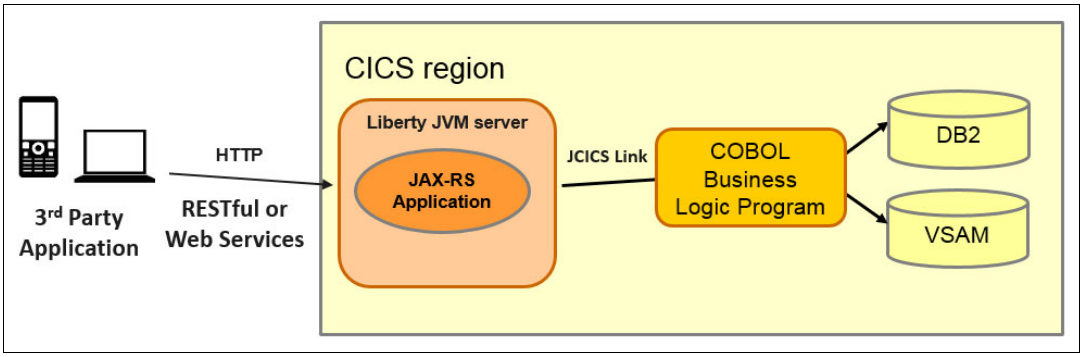


Figure 1-3 Integration with existing CICS business logic

Chapter 10, “Creating an integration logic application” on page 179 gives an example of exposing existing COBOL business logic as a RESTful web service though a Liberty web application.

1.3.7 Creating a Java business logic application

In the previous two sections, the web application was presented as a pass-through to the core business logic contained in the COBOL program. The natural progression to this is to develop new web applications that handle the business logic. With Liberty in CICS, Java EE developers can use Java EE frameworks and APIs such as Enterprise JavaBeans (EJB), Java Persistence API (JPA), Java Transaction API (JTA), and CDI Managed Beans to develop such applications. These new applications can use the JCA local ECI resource adapter to invoke CICS applications, the JCICS classes to access CICS resources such as VSAM files, and JDBC data sources to access relational databases such as IBM DB2® or Derby (Figure 1-4).

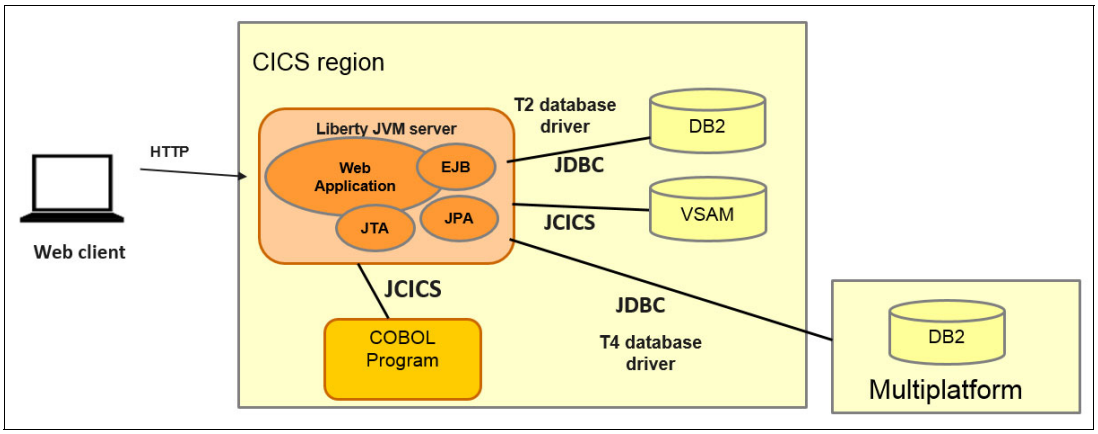


Figure 1-4 Java business logic applications in CICS

Chapter 11, “Creating a business logic application” on page 191 gives an example of creating new Java business logic in a Liberty JVM server to supplement an existing COBOL business application.

1.4 What’s new in Liberty for CICS TS V5.3

As with previous versions, CICS TS V5.3 delivers support for a new set of features in Liberty. This now includes the Java EE 6 Web Profile, as well as a growing subset of features from WebSphere Application Server Liberty V8.5.5.

1.4.1 Supported Liberty features

The following list describes the currently supported set of Liberty features supported in CICS TS V5.3 along with the CICS user features that provide for integration with the CICS runtime. This list is likely to be further extended during the lifetime of CICS TS V5.3. For the latest information, see the CICS TS V5.3 IBM Knowledge Center at this location:

http://www.ibm.com/support/knowledgecenter/SSGMCP_5.3.0/com.ibm.cics.ts.java.doc/topics/liberty_features.html

appSecurity-2.0

The appSecurity-2.0 feature enables support for securing your server and applications. This feature includes a basic user registry and allows the implementation of basic user name and password authentication and Lightweight Third Party Authentication (LTPA) token authentication.

beanValidation-1.0

The beanValidation-1.0 feature provides an annotation-based model for validation JavaBeans. JavaBeans are classes that encapsulate many Java objects into a single object known as a *bean*. The beanValidation-1.0 feature can be used to assert and maintain the integrity of data as it travels through an application.

blueprint-1.0

The blueprint-1.0 feature enables support for deploying OSGi applications that use the OSGi blueprint container application. The Blueprint Container Specification defines a dependency injection framework for OSGi and provides a simple and easy programming model for creating dynamic applications in the OSGi environment without adding complexity to the Java code.

cdi-1.0 and managedBeans-1.0

The cdi-1.0 feature enables the context and dependency injection (CDI) platform for your applications. When activated, it provides these benefits:

- Context management
- Type-safe dependency injection
- Decorators
- Interceptor bindings
- Event model
- Integration into JSF and JSP files using Expression Languages.

The managedBeans-1.0 feature enables support for the Managed Beans 1.0 specification. Managed Beans provide a common foundation for different Java EE component types that are managed by a container. Common services provided to Managed Beans include resource injection, lifecycle management, and the use of interceptors.

ejbLite-3.1

Enterprise JavaBeans (EJB) technology is the component architecture for developing and deploying business logic applications in Java. EJB provides a set of features that allow developers to produce transactional, secure, and portable applications. Because most applications will not use the full set of features available with EJB, *Ejblite* was introduced to provide a subset of the most common features of EJB without any of the remote Internet Inter-ORB Protocol (IIOP) connectivity.

jaxb-2.2

The jaxb-2.2 feature enables support for the Java Architecture for XML Binding (JAXB) 2.2 specification. JAXB is a useful technology for developing Java web services and provides two main features, the ability to serialize Java objects into XML, and the inverse, parsing XML into Java objects.

jaxrs-1.1

The jaxrs-1.1 feature enables support for Java API for RESTful Web Services 1.1, and is the key API for developing RESTful-based Java applications. It supports the use of annotations to define web service clients and servers that utilize a Representational State Transfer (REST) architectural style. In 1.3.6, “Creating an integration logic interface” on page 8, we utilize the jaxrs-1.1 feature to expose an application as a JavaScript Object Notation (JSON) RESTful web service.

jaxws-2.2

The jaxws-2.2 feature enables support for the Java API for XML-Based Web Services 2.2, and is used together with JAXB to create Java web services. The feature allows annotation-based handling of XML web services requests into applications.

jca-1.6

The jca-1.6 feature enables the development and use of the Java EE Connector Architecture (JCA) resource adapters. JCA defines an architecture for connecting Java EE applications to enterprise information systems (EISs) applications outside of Java. The jca-1.6 feature is a requirement for the JCA local ECI resource adapter, which is provided by the cicsts:jcaLocalEci0.1-0 feature.

jdbc-4.0

The jdbc-4.0 feature enables the use of Java Database Connectivity (JDBC) technology. The feature enables the configuration of data sources to access databases. This is provided through a Java API that provides querying and manipulating data in a database. This can be used in CICS Liberty to access a remote data source using a type 4 JDBC driver. All access to a local DB2 must use the cicsts:jdbc-1.0 feature and the DB2 type 2 driver.

jms-1.1, wasJmsClient-1.1, wasJmsServer-1.1, wasJmsSecurity-1.0

The Java Message Service (JMS) API is for creating, sending, receiving, and reading messages between clients. The wasJmsClient-1.1 and wasJmsServer-1.1 features are available for Liberty in CICS and provide access to WebSphere Application Server embedded messaging clients running in Liberty or classic WebSphere Application Server. MQLink, which allows IBM MQ and WebSphere Application Server messaging to interact, is not supported within Liberty.

The wasJmsSecurity-1.0 feature enables the WebSphere Embedded Messaging Server to authenticate and authorize access from JMS clients.

jmsMdb-3.1, mdb-3.1

Message-driven beans (MDBs) provide a means for the asynchronous inbound processing of messages when using JMS.

jndi-1.-0

The jndi-1.0 feature enables the use of Java Naming and Directory Inventory (JNDI) to access server-configured resources. JNDI is a Java API for a directory service that allows applications to look up data and objects by using their name.

jpa-2.0, osgi-jpa-1.0

The Java persistence API specification is an API that provides a mechanism for managing persistence and object-relational mapping and functions. Persistence is important to enterprise applications because of the required access to relational databases. Until the EJB 3.0 specification, enterprise applications were required to manage persistence themselves or use third-party solutions to handle database manipulation.

JPA represents a simplification of the persistence programming model. The JPA specification explicitly defines the object-relational mapping, rather than relying on vendor-specific mapping implementations.

json-1.0

The json-1.0 feature provides access to the JavaScript Object Notation (JSON4J) library, and is used together with JAXRS to create RESTful services. The JSON4J library provides a Java model for constructing and manipulating data to be rendered as JSON data. In 1.3.6, “Creating an integration logic interface” on page 8, we utilize the json-1.0 feature to expose an application as a JSON RESTful web service.

ldapRegistry-1.0

The ldapRegistry-1.0 feature enables support for using a Lightweight Directory Access Protocol (LDAP) server as a user registry. LDAP is a lightweight application protocol used to connect, search, and modify information about users, systems, and services. This information can then be shared across all systems that are connected to the LDAP server. LDAP security registries are supported in CICS Liberty with the cicsts:distributedIdentity-1.0 feature.

mongodb-2.0

MongoDB is a popular no-SQL database. MongoDB trades the traditional table-based relational database structure with JSON-like documents. These JSON-like documents use dynamic schema that can make integration of applications easy. The mongodb-2.0 features enable the use of a MongoDB database on a remote system.

monitor-1.0, restConnector-1.0, and localConnector-1.0

The monitor-1.0, restConnector-1.0, and localConnector-1.0 features provide monitoring and analysis capabilities for Java applications and components using Java Management Extensions (JMX) Technology.

The monitor-1.0 features enable the monitoring support for the following user runtime components:

- JVM
- Web applications
- Thread pool
- Session management
- Connection pools

The localConnector-1.0 feature provides the ability for local JMX client applications to access this information using JMX and MBeans. The restConnector-1.0 feature provides remote access to the MBeans using remote RESTful calls into the Liberty secure HTTPS listener.

osgiConsole-1.0

Eclipse Equinox provides an OSGi console that you can use for debugging your OSGi runtime. It can be used to display information about bundles, packages, and services, which can be useful when developing your own features for product extensions.

sessionDatabase-1.0

The database session persistence feature enables HTTP sessions to be stored in a locally configured relational database. This allows servlet session persistence to be shared between multiple Liberty servers and to be recovered across a Liberty server restart. This can be utilized for HTTP connection balancing and high availability configurations. For more information about high availability, see Chapter 3, “Workload management” on page 45.

servlet-3.0, jsp-2.2 and jsf-2.0

The servlet-3.0 feature enables support for HTTP servlets written to the Java Servlet 3.0 specification. The jsp-2.2 feature enables support for JavaServer Pages (JSP) that are written in the JSP 2.2 specification. The jsf-2.0 feature enables support for web applications that use the JavaServer Faces framework. These three technologies make up the foundation of the web application framework for Java applications. For more information about these technologies, see 2.1.1, “Web invocation options” on page 18.

ssl-1.0

The ssl-1.0 feature enables support for Secure Sockets Layer (SSL) connections. A secure HTTPS listener is not started unless the ssl-1.0 feature is enabled.

wab-1.0

The wab-1.0 feature enables the use of web application bundles (WABs) in OSGi applications. WABs are OSGi bundles that are internally structured the same way as a war file and support the same web components. They are deployed in an enterprise bundle archive (EBA) when deployed into a Liberty JVM server.

webcache-1.0 and distributedMap-1.0

The webcache-1.0 feature enables local caching of web application responses to improve response times and throughput, by using the WebSphere DynaCache technology. It achieves this by enabling and using the distributed-Map-1.0 feature, which provides a local cache service.

1.4.2 CICS features

CICS provides a set of user feature extensions, which provide the integration between the CICS runtime and Liberty.

cicsts:core-1.0

This is the core integration feature that provides the runtime integration, the JCICS API, and the transaction support for JTA. It should always be configured.

cicsts:distributedIdentity-1.0

This provides support for distributed identity mapping of credentials from an LDAP registry. This allows an external LDAP registry to be used for authentication, in conjunction with the cicsts:security-1.0 feature and CICS transaction and resource security for authorization. This feature is new in CICS TS V5.3.

cicsts:jcaLocalEci-1.0

This provides support for the locally optimized JCA ECI resource adapter in CICS. This enables JCA-based applications developed using the CICS Transaction Gateway ECI resource adapter to be deployed into CICS without modification.

cicsts:jdbc-1.0

This provides support for JDBC applications that need to use the CICS local DB2 attachment facility via a data source definition and the JDBC type 2 driver.

cicsts:security-1.0

This is the core integration feature that provides the runtime integration, the JCICS API, and the transaction support for JTA.

cicsts:zosConnect-1.0

This provides support for RESTful services deployed into CICS using z/OS Connect.

1.4.3 Unsupported APIs

There are several technologies categorized as optional in Java EE 6 that are not supported in the WebSphere Application Server Liberty implementation, but were previously supported in the classic WebSphere Application Server Java EE 6 runtime. The key technologies that you might encounter are JAX-RPC 1.1, JAXR 1.0, and EJB Entity Beans. Here are some details:

- JAX-RPC: Java API for XML-based RPC (JAX-RPC) specification enables you to develop SOAP-based interoperable and portable web services and web service clients. Its use for web services has been superseded by JAX-WS, which is supported in WebSphere Liberty.

- JAX-R: Java API for XML Registries (JAX-R). The Java Platform, Enterprise Edition (Java EE) 6 platform began the deprecation process for JAX-R because it is based on Universal Description, Discovery, and Integration (UD-DI) 2 technology, which is no longer relevant. If your applications use JAX-R, you might consider using UDDI 3.
- EJB: EJB Entity Beans are declared optional in Java EE 6 specification and are not supported in WebSphere Liberty. They provided a way of abstracting persistent data maintained in a relational database. Their usage has largely been superseded by the Java Persistence API (JPA) or third-party frameworks, such as Hibernate.

In addition, the EJB 1.1 and 2.0 interfaces are only supported by the JEE7 EJB remote feature on Liberty. Therefore, the ejbLite-3.2 feature does not support deployment of EJB 1.1 and 2.0 components.

Migration tools

The WebSphere Application Server Migration Toolkit Version 8.5.5 can be used to help in migrating your existing web applications to Liberty V8.5.5. This can be done by scanning the source code or static binary files of your applications and provide you with information about the preceding Java EE application API compatibility and highlight any changes that you might be required to make when moving from one version of Liberty to another. It also highlights any vendor-specific APIs that are not portable between Java Platform, Enterprise Edition application server. The toolkit can also be used when moving from other Java EE servers to Liberty.

For more information about the WebSphere Application Server Migration Toolkit, see 2.7, “Migrating a Java EE application to Liberty” on page 42.

Part 1

Technology essentials

In this part, we provide essential technology information.

Part 1 contains the following chapters:

2

Application development

In this chapter, we cover the key areas of concern when developing applications to run within Liberty in CICS.

This chapter describes the following topics:

- Application development
- Accessing CICS data
- Database access options
- Optimizing static content
- Application deployment options
- Application redeployment options
- Migrating a Java EE application to Liberty

2.1 Application development

We begin by looking into the application invocation options. Then, we look into data connectivity options, differences between JDBC driver types, and deployment and redeployment options. Finally, we cover the different technologies that are available for optimizing the serving of static data, and then look at the tools available for migrating existing applications to Liberty.

2.1.1 Web invocation options

Many options are available when it comes to the invocation of Liberty web applications when running within CICS. You can run your applications with servlets running presentation logic, or you can expose services using applications that focus on integration logic, using web services technologies.

The following sections explore the technologies that are available when choosing how you invoke your applications.

Servlets

Liberty is, at its heart, a web container that can run lightweight Java servlets, JavaServer Pages (JSP), and JavaServer Faces (JSF). This allows developers to use features of the Java servlet and JSP specifications to write modern interfaces running on Liberty in CICS. Java servlets can be produced by using the Java Servlet API, which enables the development of dynamic web content.

JavaServer Pages (JSP)

JSP extends the Java Servlet API by providing a document-centric or page-centric solution for dynamically generating content for web user interfaces. JSP allows Java code to be written into static web content. This code is then translated into a servlet and executed in the server to generate an output document that is sent to the requesting client.

JavaServer Faces (JSF)

JSF is also an extension of the Java Servlet API. They provide a component-based model-view-controller (MVC) framework. JSF is a Java framework that is designed for enterprise web application development. In particular, JSF removes the need for developers to provide functions in their applications that handle HTTP request parameters, or that convert or validate input.

Web services

Presentation logic applications are not the only type that can be used to interact with CICS. Applications can be written that interact with CICS and Liberty in CICS through integration applications. Two technologies that are available are SOAP web services and Representational State Transfer (RESTful) web services.

SOAP web services

SOAP is a simple XML-based protocol for applications to exchange information over application layer protocols such as HTTP. SOAP can be used to create request-response interactions. SOAP is a protocol that is independent of platform, operating system, and method of transport.

This XML-based protocol consists of the following three parts:

- An envelope that defines what is in the message and how to process it
- A set of encoding rules for expressing instances of application-defined data types
- A convention for representing procedure calls and responses

The most common method of exchanging SOAP messages uses HTTP. However, SOAP can be used with various transport protocols, such as Simple Mail Transfer Protocol (SMTP), File Transfer Protocol (FTP), or Java Message Service (JMS).

For more information about CICS SOAP web services, see *Implementing CICS Web Services*, SG24-7206 at this location:

http://www.redbooks.ibm.com/abstracts/SG247206.html

SOAP web services can also be enabled through Liberty in CICS with the jax-ws-1.0 feature.

RESTful web services

In CICS TS V5.3, you can deploy RESTful web services and SOAP web services. For information about RESTful and JavaScript Object Notation (JSON), see Chapter 9, “Porting a web application” on page 161.

For information about the use of RESTful and JSON in CICS or Liberty in CICS, see *Implementing IBM CICS JSON Web Services for Mobile Applications*, SG24-8161:

http://www.redbooks.ibm.com/abstracts/sg248161.html?Open

2.2 Accessing CICS data

There are three key ways for web applications to access the data from CICS applications:

- JCICS
- JCA
- JDBC

In addition, Liberty provides access to non-relational databases, such as the non-SQL database MongoDB.

2.2.1 CICS Java class library (JCICS) API

CICS includes the JCICS, the Java equivalent of the EXEC CICS API. By 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 the IBM CICS Explorer® software development kit (SDK). JCICS provides the ability to link to any other CICS applications, and to directly access VSAM files or CICS managed temporary storage or transient data queues.

Java EE Connector Architecture (JCA)

The JCA APIs are used to connect enterprise systems, such as CICS, to the Java EE platform. CICS TS V5.3 provides a local implementation of the JCA ECI resource adapter that was previously only provided by the CICS Transaction Gateway.

The support of JCA inside of CICS allows you to migrate Java Platform, Enterprise Edition applications from other application servers into CICS without modification. An example of this would be an enterprise who hosts their presentation logic on a distributed application server, which uses JCA ECI calls into CICS to access business logic. In most cases this application can be moved from the distributed application server to run inside of Liberty in CICS. You might choose to do this if you require resources through CICS access and need to reduce the number of tiers in your architecture for ease of maintenance or to gain potential operational savings.

2.3 Database access options

There are a number of ways available to access data depending on your needs. Namely, local IBM DB2, remote relational DB, or document-oriented databases. For example:

- Traditional local DB2 via the DB2 attachment facility.
- Remote database using a JDBC type 4 driver. This can be either a remote DB2 or another relational database with a type 4 driver, such as the Apace Derby database.
- MongoDB using the MongoDB API.

2.3.1 JDBC access options

You can define data sources using driver type 2 or type 4. The main difference between these two drivers is that, with type 2, you use a database-specific API to make the calls and then access the data. With type 4, you simply use Java libraries to connect directly to the database.

Figure 2-1 on page 21 illustrates in more detail some scenarios with two different driver types within a Liberty environment. Some additional information, such as EXEC SQL or the distributed data facility (DDF), is included in the graphic as a general sample, although not expanded on in this book.

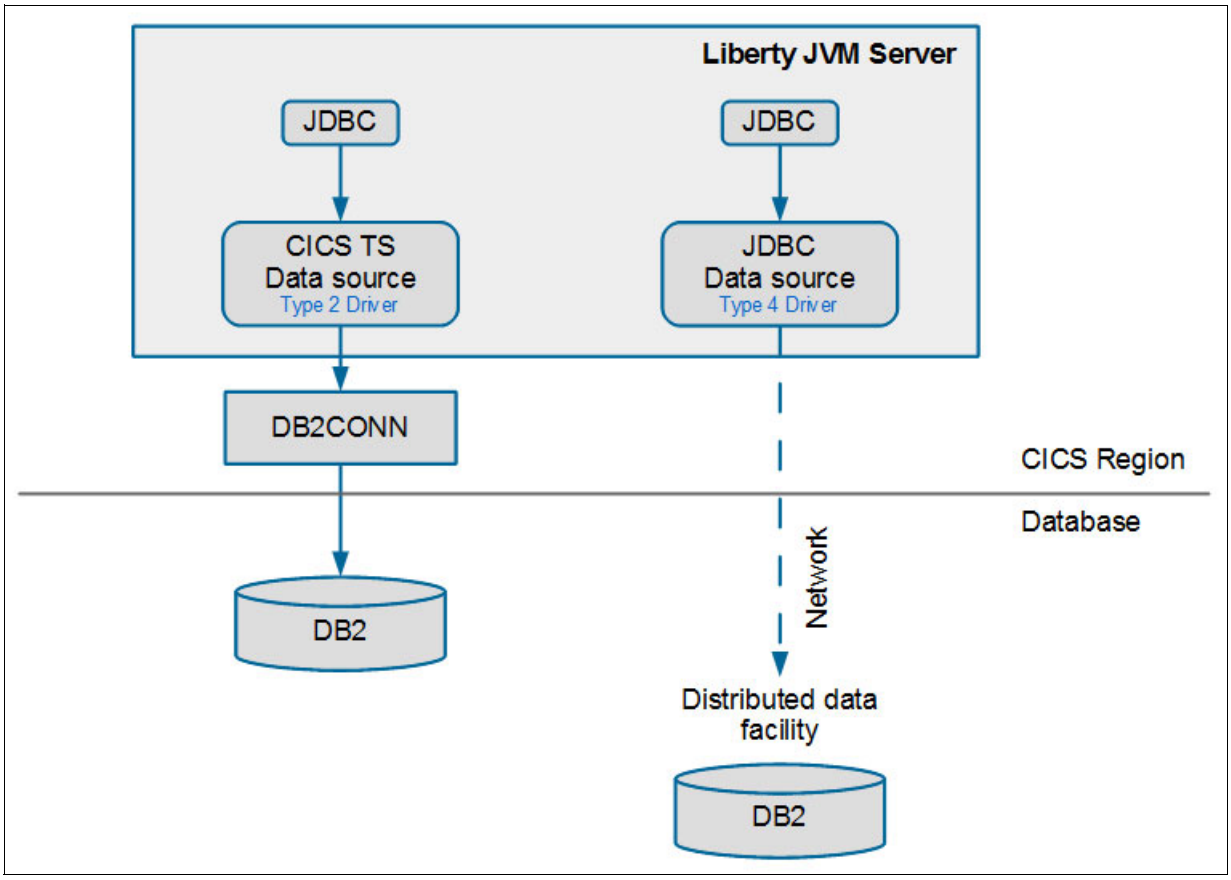


Figure 2-1 JDBC type 2 and type 4 drivers

When it comes to creating the actual data source for Liberty, you would do all of this configuration in the `server.xml` file. The following subsections describe in more detail the implementation differences between these driver types.

JDBC type 2

As mentioned earlier, type 2 driver connections require a database-specific driver to connect to a particular local database. The driver provides the APIs and is compiled for a specific platform. Being database-specific, the connection parameters might differ between data sources. From here on, we focus on DB2.

Before moving forward with the creation of a CICS DB2 JDBC type 2 driver data source, you first need to configure your CICS region to connect to DB2 in the same manner as though you were using a traditional CICS COBOL program issuing SQL commands. When defining a CICS DB2 connection, set the global attributes for that connection, the relationship between CICS transactions and DB2 resources, among other items.

For more information about how to define the CICS DB2 connection, refer to the *Defining the CICS DB2 connection* portion of the IBM Knowledge Center at this location:

`http://www.ibm.com/support/knowledgecenter/SSGMCP_5.3.0/com.ibm.cics.ts.doc/dfhtk/topics/dfhtk2c.html`

After you have the CICS DB2 connection defined, creating a type 2 driver data source requires that you edit the `server.xml` file. Some of the elements that need to be updated to define a type 2 driver data source include:

1. Add the `cicsts:jdbc-1.0` feature to the featureManager element. This enables use of the `cicsts_jdbcDriver` and `cicsts_dataSource` elements that are used later in the `server.xml` file.

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

2. Add a `cicsts_jdbcDriver` element. This enables support for access to JDBC type 2 driver data sources using either `java.sql.DriverManager` or `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, the preceding steps are sufficient. To access DB2 connections using data source definitions, a `cicsts_dataSource` element is required for each data source for which access is to be defined. A `cicsts_dataSource` specifies a `jndiName` attribute. This defines the Java Naming and Directory Interface (JNDI) name that is a reference used by your application program when establishing a connection to that data source. A definition might look like this:

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

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

4. Optionally, you can set properties for the `cicsts_dataSource` element using a `properties.db2.jcc` element. The following example shows how to do this:

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

   Not all properties permitted in the `properties.db2.jcc` element are appropriate for JDBC type 2 driver data sources. The inappropriate properties are ignored.

Not relevant properties: Unrecognized properties are ignored. The following properties are not relevant for the CICS JDBC type 2 driver support:

- driverType
- serverName
- portNumber
- user
- password
- databaseName

If specified, they are also ignored and a warning message issued.

The Liberty server, when started, is configured to allow access to DB2 databases using a JDBC type 2 driver connection.

Note: Dynamic updates of the CICS data source and its components are not supported. Updating the configuration while the Liberty server is running can result in DB2 application failures. Be sure to recycle the server to activate any changes.

JDBC type 4

Unlike the type 2 driver that we just analyzed, the DB2 JDBC type 4 driver data source does not use the CICS DB2 connection resource.

DB2 JDBC type 4 driver data source can also be manually enabled via the `server.xml` configuration file. This allows the establishment of connections to remote DB2 databases. Because 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).

Here are some of the elements that you need to update to define a Type 4 data source:

1. Add the `jdbc-4.0` feature to the `featureManager` element. This enables use of the `dataSource` and `jdbcDriver` elements that are 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" />
</library>
```

The `dataSource` element specifies a `jndiName` attribute. This defines the JNDI name that is a reference used by your application program when establishing a connection to that data source. The required properties are set in the `properties.db2.jcc` element as follows:

- driverType
  - Description: Database driver type, must be set to 4 to use the pure Java driver
  - Default value: 4
  - Required: false
  - Data type: int
- serverName
  - Description: The host name of the server where the database is running. This is the SQL DOMAIN value of the DB2 DISPLAY DDF command
  - Default value: localhost
  - Required: false
  - Data type: string
- portNumber
  - Description: Port on which to obtain database connections. This is the TCPPORT value of the DB2 DISPLAY DDF command
  - Default value: 50000
  - Required: false
  - Data type: int
- databaseName
  - Description: Specifies the name for the data source. This is the LOCATION value of the DB2 DISPLAY DDF command
  - Required: true
  - Data type: string
- user
  - Description: The user ID to connect to the database
  - Required: true
  - Data type: string
- password
  - Description: The password of the user ID to connect to the database. The value can be stored in clear text or encoded form. It is recommended that you encode the password by using the securityUtility tool with the encode option
  - Required: true
  - Data type: string

  For more information about the Liberty profile **`securityUtility`** command, see the IBM Knowledge Center at this location:

  `http://www.ibm.com/support/knowledgecenter/SS7K4U_8.5.5/com.ibm.websphere.wlp.zseries.doc/ae/rwlp_command_securityutil.html`

The Liberty server, when started, is configured to allow access to DB2 databases using a JDBC type 4 driver connection.

2.3.2 Java Transaction API

The Java Transaction API (JTA) is used to coordinate transactional updates to Java based resource managers in the Liberty JVM server environment. This includes XA-based data sources, such as type 4 driver data sources, a remote transactional JCA resource adapter, or third-party resource managers.

When using a type 4 JDBC driver, 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.

When using a type 2 JDBC driver, all requests flow through the CICS DB2 attachment facility. Therefore, it is not necessary to use JTA to coordinate with updates to other CICS resources, as the CICS recovery manager is the transaction coordinator.

In JTA, you create a UserTransaction object to encapsulate the global transaction. This provides two key methods `begin()` and `commit()`, which demarcate the transactional scope of the work performed.

The following code fragment illustrates how to control 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();
}
```

Unlike a CICS unit-of-work (UOW), a UserTransaction must be explicitly started by using the `begin()` method. Invoking `begin()` causes CICS to commit any updates that might have been made before 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, it is not possible to issue a CICS **`synncpoint`** command, from either the JCICS API or linked to programs that issue EXEC CICS commands. The effect of this means that the Java program cannot invoke the JCICS Task `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 if a Liberty JVM server fails during two-phase commit processing of a Java transaction, and the Java transaction is left in doubt, CICS initiates transaction recovery as

soon as the Liberty JVM server initialization is complete. If the JVM server is installed as disabled, recovery runs when it is set to enabled.

2.3.3 Access to non-SQL databases

Document-oriented databases, such as MongoDB, are increasing in popularity. You query these databases by using custom C++ syntax as opposed to SQL queries. In these non-SQL databases, the data is stored in BSON files, which are a binary representation of JSON files. A typical BSON entry looks like this:

```
{
  ID: 6535435
  name: CICS
  area: TP
}
```

At the time of this writing, MongoDB is available for distributed platforms and Linux on z Systems. The mongodb-2.0 driver is available to allow for remote database instances to be configured in the server configuration. Applications would interact with these databases through the MongoDB APIs.

You must download the MongoDB driver from GitHub at this location:

https://github.com/mongodb/mongo-java-driver/releases

After the driver is downloaded, place the driver in a directory where the CICS Liberty JVM server can access it.

Similarly, as with the JDBC drivers, MongoDB connections are defined in the `server.xml` file. The following are some of the elements that you need to update to enable MongoDB:

1. Add in a shared library reference to the MongoDB Java driver that was just downloaded:

   ```
   <library id="MongoLib">
      <file name="${server.config.dir}/lib/mongo-java-2.11.1.jar" />
   </library>
   ```

2. Configure a library reference for each application that uses MongoDB:

   ```
   <application name="MyApp1" location="MyApp1.war">
      <classloader commonLibraryRef="MongoLib"/>
   </application>
   ```

3. Enable the MongoDB feature and, if required, the JNDI feature:

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

   JNDI is only required if you are planning to use the JNDI lookup method of accessing MongoDB.

4. Configure a reference to the MongoDB database:

   ```
   <mongo id="mongo" libraryRef="MongoLib" hostNames="winmvsXX.myserver.com"
   ports="27017"/>
   ```

   The values in hostNames and ports must point at the user's instance of MongoDB.

5. Configure a reference to MongoDB for use within the application itself:

   ```
   <MongoDB jndiName="mongo/testdb" mongoRef="mongo" databaseName="db-test" />
   ```

The `jndiName` is used within the application to gain reference to MongoDB. The `mongoRef` attributes reference the ID of the mongo tag in `server.xml`.

Accessing MongoDB from your web application

Update your application `web.xml` file to match the MongoDB references just defined in the `server.xml` file:

```
<resource-env-ref>
   <resource-env-ref-name>mongo/testdb</resource-env-ref-name>
   <resource-env-ref-type>com.mongodb.DB</resource-env-ref-type>
</resource-env-ref>
```

The `resource-env-ref-name` matches the value of the `jndiName` that you defined in the `server.xml` file.

You can now make use of MongoDB in your web application. There are two main supported ways of doing this:

- Resource injection works by adding the following to the top of your Java class:

```
@Resource(name = "mongo/testdb")
protected DB testdb;
```

- A JNDI lookup can be used by adding the following to a method of your Java class:

```
DB lookup = (DB) new InitialContext().lookup("java:comp/env/mongo/testdb");
```

Limitations

For the MongoDB feature, the following restrictions apply for WebSphere Liberty Profile:

- There is no integration with Java transaction or CICS recovery, as there is no transactional access to the database.
- MongoDB is not supported on IBM z/OS. Therefore, connections must be made to a remote database, such as Linux on z Systems or a distributed platform.
- Only versions 2.10.0 and later are supported for the Java driver.

For more information about the current level of support, see the MongoDB website:

https://www.mongodb.com/press/mongodb-announces-support-for-ibm-z-systems-mainframe-solutions

2.4 Optimizing static content

Caching the static content is a key technique for improving application performance. It can be used to host internal or external websites. Caching static content at the edge, like images and scripts, is only the beginning as far as offloading requests. For a typical web application, caching can take place at several different layers:

- At the web client or browser
- In a caching proxy set-up in front of the Liberty profile in CICS
- In an HTTP web server
- In the application server in DynaCache
- In the back-end database tier

2.4.1 Static content separation

The content of a web application can consist of two types: Dynamic or static. The dynamic runtime components are those that require some type of processing to be generated (such as servlets, JSP, and PHP scripts). The static content are the pieces that will never, or only occasionally, change (such as HTML files, graphic images, audio files, or style sheets).

When developing Java Platform, Enterprise Edition web applications, the easiest approach is to deploy the entire application onto the application server. But, with this approach, both the static and runtime components of the application are deployed to the application server, the Liberty Profile running in CICS. Some of these static elements can be large and frequently requested, so for performance reasons, especially in a high workload environment, the preference is for static content to be served from a dedicated web server rather than a fully functional application server.

Optimizing static data in CICS

In the CICS Liberty JVM server, by default, all HTTP requests run as CICS transactions on a T8 task control block (TCB). However, requests to serve static data, such as .gif or .jpeg files, do not need the security or transaction environment this provides. Therefore, as an optimization, you can use the following JVM system properties to ensure that requests for static data run as Liberty threads only, and do not run as CICS transactions. This will both improve the efficiency of the requests and reduce queuing for CICS tasks:

- `com.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, .js, and .png.

- `com.ibm.cics.jvmserver.wlp.optimize.static.resources.extra=`

If `com.ibm.cics.jvmserver.wlp.optimize.static.resources` =true, you can provide a custom list of extra static resources for optimization. Items must be comma-separated, and begin with a period, for example: .css, .gif, .ico.

Using a dedicated web server

In the Servlet 2.2 specification, Sun introduced the web archive (WAR) file that can contain all of the compiled Java code that makes up a web application. It can contain servlets, JSP pages, utility classes, static documents, client-side applets, beans and classes, and metadata that ties everything together. However, if the static content within an application is to be served separately from the dynamic content, a simple way is to ensure the images are not deployed within the WAR using a local reference (such as src="`images/images1.gif`") but instead referenced using absolute paths to remote URLs on a dedicated web server from where the static data can be cached.

Figure 2-2 shows a topology where the static content is serviced within the web server and the dynamic content is serviced on the server side in the Liberty profile. This configuration is a way to separate the static and dynamic content, but it is not necessarily the best way. For instance, you need to use a separate host name and port number for each server.

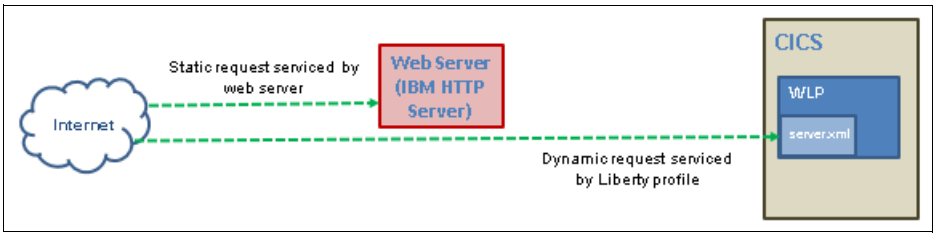


Figure 2-2 Serving static content in web server and dynamic content in back-end

Using the web server plug-in

Another option is to use the WebSphere Application Server to provide a way to let the web server instance know about all the applications that are running within the Liberty Profile in CICS and how to access them. The configuration in Figure 2-3 shows that all requests are received by the web server, which allows the web server plug-in to forward any requests to the Liberty Profile in CICS, depending on how it is configured. This allows the web server to serve the static content without having to send the request to the back end for servicing. In addition, the dynamic content can still be served by the back end. The user in this case will only be aware of the host name and port number of the web server. This configuration also provides easier scalability if wanted.

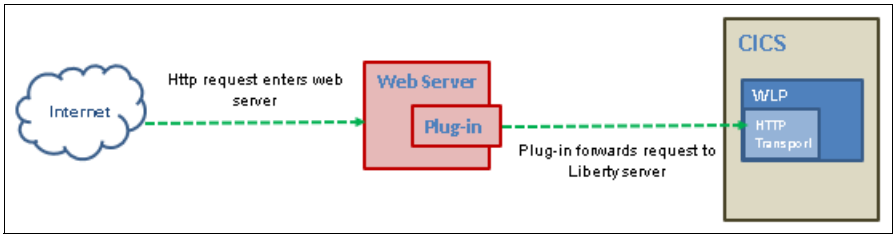


Figure 2-3 Using a web server plug-in to forward request to Liberty in CICS

Many web browsers and intermediate servers cache web content to help speed up its delivery and display. This is useful for servicing static data, the unchanging content such as HTML, graphics, and JavaScript files. But what about dynamically generated web content?

2.4.2 Dynamic caching

Static caching is a valuable benefit. However, dynamic caching is much more useful, as it provides caching of HTML pages based on its request path, query strings, cookies, and request headers. Dynamic requests tend to use far more enterprise resources in building a response, so dynamic caching can dramatically enhance performance.

Caching the output of servlets, commands, and JavaServer Pages (JSP) improves application performance by serving requests from an in-memory cache. Cache entries contain servlet output, the results of a servlet after it runs, and metadata. The Liberty Profile uses a service called *dynamic caching* to consolidate these caching activities. Dynamic caching (or DynaCache) is an in-memory cache with the ability to offload the content on disks. Dynamic caching is more complex than static caching and requires detailed knowledge of the application. You must consider the candidates for dynamic caching carefully, because dynamically generated content can be different based on the state of the application. Therefore, it is important to consider under what conditions dynamically generated content can be cached to return the correct response. This requires knowledge of the application, its possible states, and other data, such as parameters that ensure the dynamic data is generated in a deterministic manner. A good candidate for dynamic caching is any data that is dynamic and, at the same time, is stable long enough for it to be reused.

The concept behind dynamic caching is to store results of a first execution of a dynamically generated web page. Any subsequent requests made to the same page go to cache. This helps to avoid the overhead incurred by executing an application that renders output that does not change. The dynamic cache service of the Liberty Profile allows you to cache the output of servlets and JSP files.

The caching service works within the application server process. It is invoked before a service() method call of a servlet. The service can either execute the service() method and save the output in the cache or directly serve cached content and avoid a call to the service() method. Because Java Platform, Enterprise Edition applications have high read/write ratios and can tolerate small degrees of latency in the currency of their data, the dynamic cache can create an opportunity to improve server response time, throughput, and scalability.

To cache the servlets and JSP, when the application server starts and a targeted servlet or JSP is called for the first time, no cached entry is found and so the service() method of the servlet is called. The DynaCache web container intercepts the response from the service() invocation and caches the results as shown in Figure 2-4.

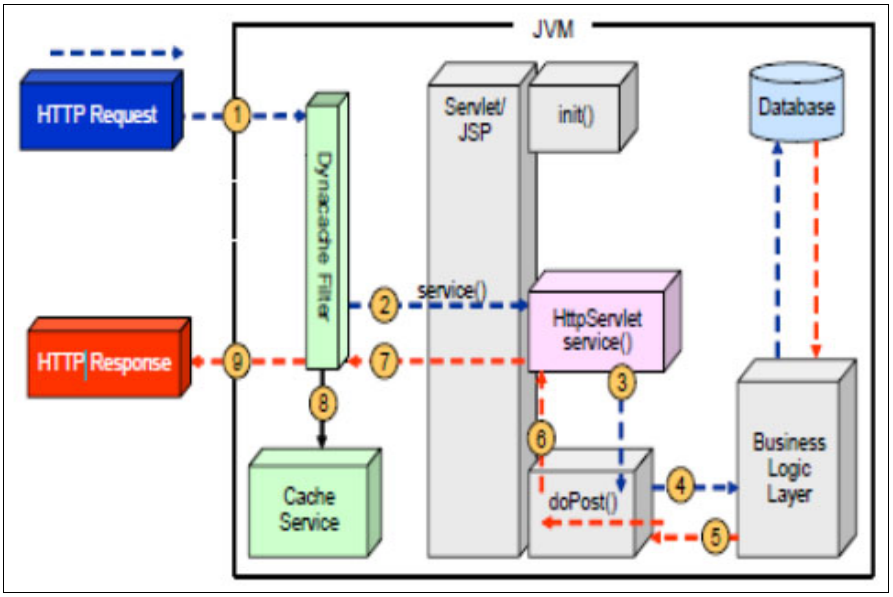


Figure 2-4 Before caching, a request must traverse all layers

The next time that the servlet is called and a match is found in the caching rules engine, the cached results are returned and the service() method is not called. This bypasses all of the processing that would have been done by the servlet, resulting in a substantial performance boost. However, if there is no cache entry for this ID, the service() method is run as normal, and the results are caught and placed in the cache before they are returned to the requester.

Figure 2-5 on page 31 illustrates how DynaCache increases performance by bypassing the service() method call entirely whenever a cache hit occurs.

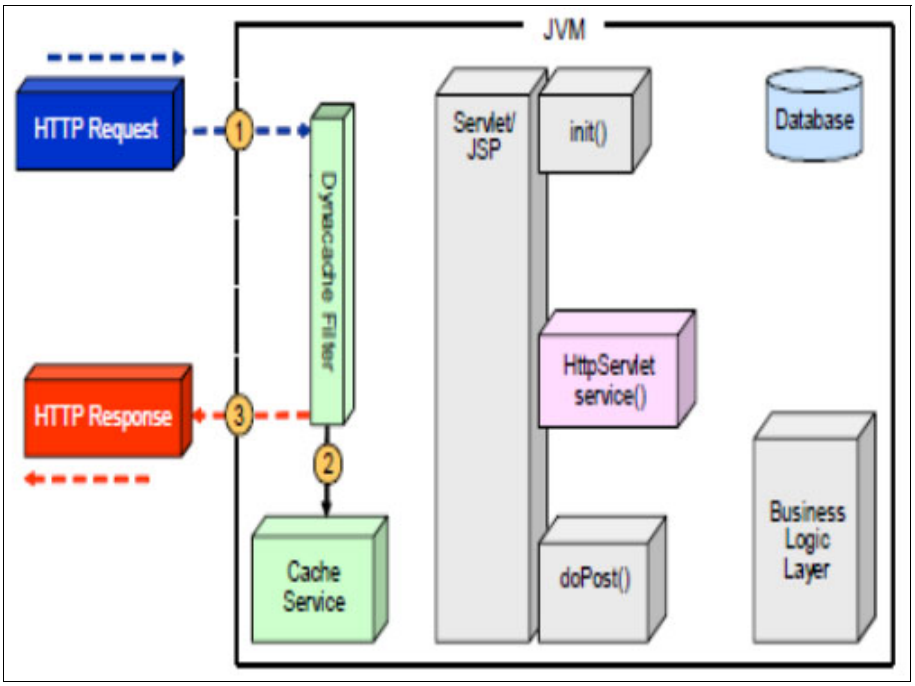


Figure 2-5 A cache hit means the servlet and business logic are not called

Servlets and JSP are configured for caching with entries in the `cachespec.xml` file. In Liberty, this file has to be packaged within the application, either in WEB-INF or META-INF. The file defines the structure and elements for `cachespec.xml`. This is in `dev/api/ibm/schema`.

DynaCache stores objects in a JVM heap or on disk. It later retrieves and serves them from its cache based on data matching rules. It creates a unique user-defined key in which to store and retrieve cache items.

With the Liberty profile, caches are defined in the `server.xml` file. To enable caching, add the webCache-1.0 feature to the list of enabled features in the `server.xml` file. In Liberty, there is no distinction between servlet cache instances and object cache instances. All cache instances are a distributed map. Applications can include a `cache-spec.xml` file to customize the response caching.

To define the cache, use the distributedMap element:

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

<distributeMap id="baseCache" jndiName="services/cache/baseCache">
  <diskCache/>
</distributeMap>
```

When you enable the webCache-1.0 feature, the following caches are configured and created:

- basecache with JNDI name services/cache/basecache
- default with JNDI name services/cache/distributedmap

The `messages.log` file shows the following messages for the cache initialization:

```
DYNA0053I: Offload to disk is enabled for cache named baseCache in directory
/u/reds10/cicsts53/workdir/CREDS10A/DFHWLP/wlp/usr/servers/defaultServer/workarea/
_dynacache/baseCache.
DYNA0061I: Flush to disk on stop is disabled for cache name baseCache.
DYNA0059I: The disk cache configuration for cache name baseCache. The
configuration is:   DiskCacheSize=100000  DiskCacheSizeInGB=3
DiskCachePerformanceLevel=3  DelayOffload=true  DiskCacheEvictionPolicy=1
DiskCacheHighThreshold=80  DiskCacheLowThreshold=70  DataHashtableSize=477551
DepIdHashtableSize=47743  TemplateHashtableSize=1031
DYNA1001I: WebSphere Dynamic Cache instance named baseCache initialized
successfully.
DYNA1071I: The cache provider default is being used.
DYNA1056I: Dynamic Cache (object cache) initialized successfully.
DYNA1055I: Dynamic Cache (servlet cache) initialized successfully.
```

By default, this cache is in the server `workarea` directory, as shown in Figure 2-6.

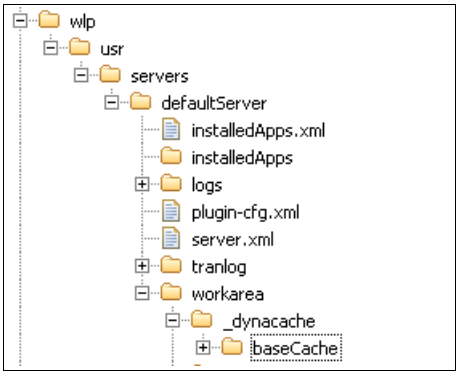


Figure 2-6 Directory structure containing the cache

2.5 Application deployment options

In Java EE applications, modules are packaged based on their functionality. There are three key archive types supported for deployment into Liberty in CICS:

- Web archive (WAR)

  This is used to package a dynamic web project containing servlets, JSP files, supporting files, gif, and HTML pages. This is the simplest form of archive, and is best used when either simple web applications with limited external dependencies or when migrating a web application from another server.

- Enterprise archive (EAR):

  This is used to package enterprise applications containing EJB beans and one or more web applications. This is the best archive to use when deploying EJB applications because Liberty does not support EJB in Open Services Gateway initiative (OSGi).

- Enterprise bundle archive (EBA):

  This is used to deploy OSGi application projects containing OSGi bundles and web enabled OSGi bundles (WABs). This is the archive type to use for modular OSGi application development, allowing you to share OSGi bundles between components in the same runtime.

When dealing with Liberty applications running in CICS, several ways are available to deploy applications as an archive file (WAR, EAR, or EBA). The first set of options uses Liberty deployment methods. You can deploy an application either by dropping it into a previously defined `dropins` directory or by adding an application entry to the server configuration. In addition to this mechanism, you can use the CICS deployment method, which enables you to package Liberty applications inside CICS bundles. In addition, you can share common utility bundles between multiple applications when needed. This section provides an overview of these options.

Figure 2-7 demonstrates the various ways a web application can be deployed to the Liberty Profile in CICS.

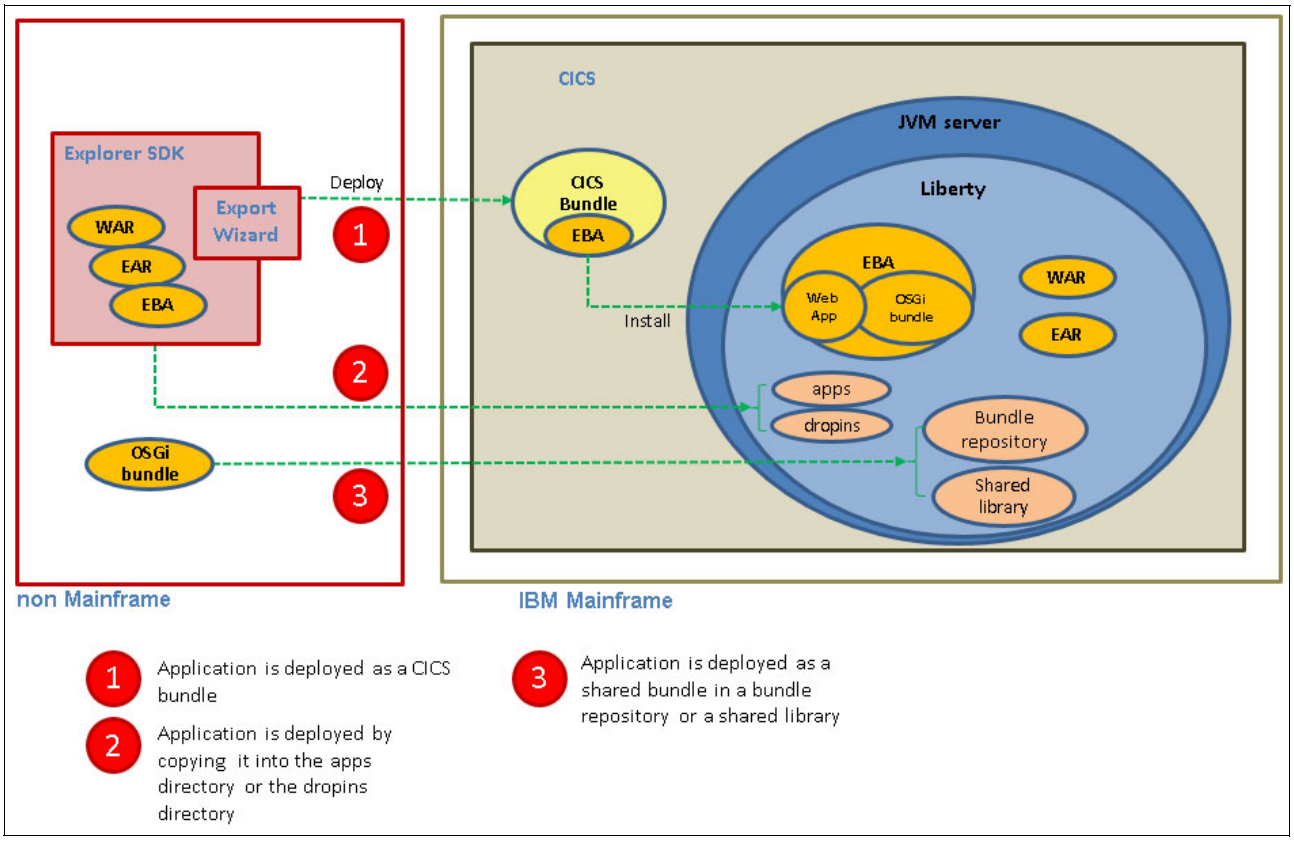


Figure 2-7 Various ways to deploy a web application

2.5.1 Liberty deployment methods

The Liberty Profile supports two models for deploying applications. By default, Liberty looks for applications in two directories: The apps and `dropins` directories in the server configuration directory. By using the Liberty runtime, web applications can be copied into the server `dropins` directory to be deployed, or they can be specified directly as applications within the `server.xml` configuration file.

Deploy to the dropins directory for a quick start

In a development environment, running your application in an expanded format is common. The quickest way to do this is to use the dropins deployment method because this automatically deploys your application. This method uses a predefined `dropins` directory that is periodically scanned by Liberty. If any new applications are in this directory, the Liberty configuration is updated with the changes so that the new application is deployed. Your application can be packaged as an archive file, a directory, or as a loose application where files are in multiple locations. When you use this model, CICS is not aware of the web application that is running in the Liberty JVM server.

With this approach, there is no application specific entry in the `server.xml` file, so you do not have as much control over certain aspects. Applications that are deployed by this method do not benefit from additional qualities of service, such as security, and always run under the CJSA transaction.

Note: By default, in CICS, this deployment option is turned off. If you accept the defaults provided by CICS, which specifies `-Dcom.ibm.cics.jvmserver.wlp.autoconfigure=true` in the JVM profile, the `dropins` directory is not automatically created or monitored by the Liberty JVM server.

To automatically deploy applications using the dropins method, all you need to do is to enable the feature in the `applicationMonitor` element. Then, just copy the application files in binary mode to the `${server.config.dir}/dropins` folder:

1. Verify application monitoring is defined in the `server.xml`, and update as needed. Here is an example of the `applicationMonitor` element.

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

   The `dropins` property specifies the name of the directory used as the `dropins` directory.

   The `dropinsEnabled` property determines whether the applications in the `dropins` directory are deployed. In CICS by default, this is set to `false`.

   The `pollingRate` property determines the interval amount and the unit of time (such as ms for milliseconds, s for seconds, and so on).

   The `updateTrigger` property specifies whether the Liberty server should monitor for application updates. It has three possible values:

   polled The runtime environment scans the `server.xml` file for changes using the timing interval specified by the monitorInterval property on the <config../> element.

   Mbean The runtime environment only looks for updates when prompted to do so through a call to an MBean. This is the mode that is used by the developer tools to update the `server.xml` file, unless you override it.

   disabled The updates are not dynamically applied.

2. Copy the file in binary mode to `${server.config.dir}/dropins`.

There are several options for placing applications in the `dropins` directory. Each provides a way for the server to determine the application type.

The following list describes the placement options:

- Place the archive file with its identifying suffix (ear, war, eba, and so on) directly in the `dropins` directory: `${server.config.dir}/dropins/myApp.war`
- Extract the archive file into a directory that is named with the application name and the identifying suffix: `${server.config.dir}/dropins/myApp.war/WEB-INF/...`
- Place the archive file or the extracted archive into a subdirectory that is named with the identifying suffix: `${server.config.dir}/dropins/war/myApp/WEB-INF/...`

If you put your applications in the `dropins` directory, you must not include an entry for the application in the server configuration file or deploy the application in a CICS bundle resource. Otherwise, the server tries to load the application more than once and an error might occur.

Deploy to the apps directory for more control

In certain situations, the preference might be to deploy an application to Liberty with more control. As an alternative to using the dropins mechanism, you can manually define your application in the server configuration and use the apps directory. This method consists of a set of statements that locate the application and define its properties. This allows you to specify additional properties, such as the context root, alter application start behavior, and so on. You can use the `<application…/>` element, which supports the following attributes:

- ID: Must be unique and is used internally by the server.
- Name: Must be unique and dependent on the application.
- Type: Specifies the type of application; supported types are WAR, EBA, and EAR.
- Location: Specifies the path to the application; this can be an absolute path or URL.

The location attribute is the only required attribute for a declared application. The location can be on the file system or at a URL.

The steps provided here show how to configure an application in the server configuration:

1. Define the application to the server configuration (`server.xml`)

   ```
   <application id=myApp type="war" name="myApp" location="myApp.war"/>
   ```

2. Copy the file in binary mode to `${server.config.dir}/apps`

Type-specific application elements can be used to simplify the declaration of applications:

- `<republication.../>`
- `<osgiApplication.../>`
- `<enterpriseApplication.../>`

These elements are used in a manner identical to the generic <application.../> element, but eliminate the type attribute because the application type is built into the shorthand of the element.

2.5.2 CICS deployment methods

CICS TS allows you to package web applications inside a CICS bundle project where you have to reference the Liberty applications (WARs, EARs, or EBAs) as components. The BUNDLE resource represents the application so you can also manage its availability by enabling and disabling it in CICS. It is a collection of CICS resources, artifacts, references, and a manifest that you can deploy into a CICS region to represent all or part of an application.

Deploy as a CICS bundle to manage application lifecycle

When you have created a CICS bundle project in CICS Explorer, you must export it to a CICS region. You can either export the project directly to zFS (or any server with an FTP daemon), or you can use the CICS build toolkit on a build machine to automate this function. You must also define and enable a BUNDLE resource for the CICS bundle project. CICS then creates the Bundle project resources dynamically for you. The CICS bundle deployment process from CICS Explorer automates the creation of the Liberty application reference using the `installedApps.xml` configuration file, which is included from server.xml. This approach is best suited to a production environment.

Note: You can export a Bundle directly to a zFS system only if your network is using Internet Protocol version 4 (IPv4). The procedure does not work for IPv6 networks. The port can be secured to accept Secure Sockets Layer (SSL) traffic. If this is done, CICS Explorer detects this and switches to SSL mode so that no data flows decrypted. If an SSL connection cannot be established, basic authentication is used.

The steps presented here show how to deploy the application in a CICS bundle resource:

1. Verify that the following line is in server.xml. This file is used to define CICS Bundle deployed applications.

   ```
   <include location="${server.output.dir}/installedApps.xml"/>
   ```

2. Create a CICS bundle definition to control the lifecycle of the application.
3. Add the application reference to the CICS bundle.
4. Export the CICS bundle project to a zFS location.
5. Create and install the CICS BUNDLE resource definition.

The `installedApps` directory is used to define CICS bundle deployed applications. This include is added automatically if `-Dcom.ibm.cics.jvmserver.wlp.autoconfigure=true` is specified in the JVM profile.

2.5.3 Deploying shared (middleware) bundle

OSGi applications can share many common bundles. To simplify maintenance, an application does not have to include its own copy of each bundle. Instead, bundles can be shared from a bundle repository or a shared library. You can deploy the shared bundles in various ways depending on how it is supplied, as DLL files, JAR files, or OSGi bundles.

Libpath

For middleware supplied as .dll files, copy the files to a directory that is referred to by the LIBPATH_SUFFIX option of the JVM profile.

Bundle repository

For middleware supplied as OSGi bundle JAR files, you can share these common OSGi bundles both locally or remotely so that your OSGi applications can access them. This can be accomplished by placing them in a directory and configuring the bundleRepository element in the server.xml.

Here is a local repository configuration:

```
<bundleRepository>
    <fileset dir="directory_path" include="*.jar"/>
<bundleRepository>
```

The directory_path used to define a local OSGi bundle repository is the path to the directory containing the common OSGi bundles.

Here is a remote repository configuration type:

```
<bundleRepository location="URL" />
```

The URL used to define a remote OSGi bundle repository defines the location of an OSGi Bundle Repository XML file, and it supports HTTP, HTTPS, and file protocol types. You can define a remote and local repository at the same time by using both a location attribute and nested fileset tags within the same bundleRepository entry.

Shared libraries

Shared libraries provide a collection of classes that can be used by multiple applications or data sources. You can use shared libraries and global libraries to reduce the number of duplicate library files on your system. Libraries have three elements: <folder>, <file>, and <fileset>. Copy the JAR files to a directory that is referred to in a global library definition in the server.xml file:

```
<library>
  <folder dir="..." />
  <file name="..." />
  <fileset dir="..." includes="*.jar" scanInterval="5s" />
</library>
```

Where:

- <folder>: All resources under each configured folder will be loadable.
- <file>: Each configured file should be either a native library or a container for resources (such as a JAR or a ZIP file). All resources within a container are loadable, and any other file type that is specified will have no effect.
- <fileset>: Each configured fileset is effectively a collection of files. Each file in the fileset should be a native library or a container for resources (such as a JAR or a ZIP file). All resources within a container are loadable, and any other file type that is specified will have no effect.

A library on the Liberty profile server is composed of a collection of file sets that reference the JAR files to be included in the library.

There are three types of shared libraries in Liberty:

- Global shared libraries that allow all applications to share a common instance of the library
- Configured common libraries that allow specific applications to share a common instance of the library
- Configured private libraries that allow specific applications to use private instances of the library

A global shared library can be used by any application. The library is visible to all applications deployed in the server. All applications access the same instances of those classes. The JAR files are placed in a directory and then are specified in the class loader configuration for each application. You can place global libraries in one of the following locations:

- `${shared.config.dir}/lib/global`
- `${server.config.dir}/lib/global`

Using a global library can be useful in some situations, especially for CICS deployed WAR or EAR components, but it does have drawbacks:

- All applications have visibility to the same static variables in the library classes (they are not isolated)
- It is not possible to provide different versions of the same library for use by different applications

When the application is started, if files are present in the global location, and that application does not have a <classloader> element configured, the application uses the global libraries. If a class loader configuration is present for the application, the global libraries are not used unless explicitly referenced.

The following code shows a global library entry added to the server configuration (server.xml) file:

```
<fileset dir="/u/myApp/shared" id="myappshare" includes="*.jar"/>
  <library filesetRef=" myappshare" id="global"/>
```

A configured library can be used in an application by creating a classloader definition for the application in the server configuration. The application classloader can share the in-memory copy of the library classes with other applications, or it can have its own private copy of the classes that are loaded from the same location. A shared library can be used only by applications that are explicitly configured in the `server.xml` file. Applications without configuration, such as those placed in the `dropins` directory or those deployed using CICS, cannot use a global shared library.

The following code shows a library entry added to the server configuration (`server.xml`) file and applications using the classloader child element:

```
<library id="myLibrary">
   <fileset dir="${shared.resource.dir}/lib" includes="*.jar" />
  <file name="${shared.resource.dir}/App/app.jar"/
</library>

<application location="Application1.war">
   <classloader commonLibraryRef="myLibrary"/>
</application>

<application location="Application2.war">
   <classloader privateLibraryRef="myLibrary"/>
</application>
```

In this example, Application1 shares its copy of the myLibrary classes with any other application that specifies it as a commonLibraryRef. Application2 has its own private copy of the myLibrary classes. Common libraries are useful where different versions of a library should be made visible to different applications, but the same instance of each version can be safely shared by the consuming applications. This limits the memory use of the library classes. Private libraries are useful where a shared library instance needs to be isolated for each application, or if the library needs to be able to access the application classes.

2.6 Application redeployment options

When configuring the Liberty Profile to run in CICS, important considerations are the methods for updating applications when you want to make changes. In the previous section, we described the various methods for deploying applications to run in Liberty Profile. In this section, we look at the ways to redeploy those applications so your changes are picked up.

2.6.1 Redeploy by using Liberty deployment methods

Within Liberty, three types of dynamic updates can be controlled through the configuration:

- Changing the server configuration
- Adding and removing applications
- Updating installed applications

For all deployed applications, you can configure whether application monitoring is enabled and how often it should check for updates to applications. When deployed applications are monitored for updates, the updates are dynamically applied to the running application. This applies both to applications that are deployed from the `dropins` directory or applications that are deployed through configuration entries in server.xml. In addition, for the `dropins` directory, you can also configure the name and location of the directory and choose whether to deploy the applications that are in that directory.

Within a CICS Liberty JVM server, application monitoring is turned off by default. So if you are using the Liberty deployment methods to deploy your applications, you must enable application monitoring by specifying the applicationMonitor element within server.xml. This element defines how the Liberty server responds to application additions, updates, and deletions. For example, the following line causes Liberty to scan the `dropins` folder every 5 seconds for any application updates, and for any applications that have been added or removed. The pollingRate is applicable only when the updateTrigger is set to polled:

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

For applications that are deployed using the app directory, the config element can be used to control how often the `server.xml` file is scanned for updates:

```
<config monitorInterval="5s" updateTrigger="polled"/>
```

For applications in the `dropins` directory, the file name and file extension are used by the application monitor to determine the type of application, and to generate the application ID and application name. Therefore, specifying the version number for the application by using the file name or file extension is not possible. In a production environment, we suggest that you not use the `dropins` directory.

2.6.2 Redeploy using CICS bundle deployment

Within a CICS environment, the preference is to deploy web applications using a CICS bundle. This allows CICS to manage the lifecycle of the application so that updating and managing are easier.

There are three main methods for redeploying Liberty web applications when using a CICS bundle project.

- Refresh the bundle in the Liberty JVM
- Create a version of the bundle with a new context root URI
- Use cloned JVM servers and set up load balancing

In this section, we look at each of these options. We also discuss the shared bundles and what to consider for deployment.

Refresh the bundle

The first method that you can use to redeploy a CICS bundle is to replace the deployed CICS bundle project completely with a new version of the project. This method can be used when a server outage can be tolerated during the update process. No new CICS resources are created, but you might need to update the existing CICS BUNDLE resource definition. To replace an existing Bundle with a new version of the application, you can do the following steps:

1. Create a new version of the web application.
2. Export to the same zFS location as the old version.
3. Disable the installed BUNDLE resource that represents the previous version. This removes and stops the application.
4. Enable the installed BUNDLE resource. This reinstalls the application.

Note: At Step 2, you can optionally export to a different zFS directory, which would enable you to roll back to the previous version more easily. However, if you do this, you need to edit the BUNDLE resource to modify the BUNDLEDIR attribute to reference the new zFS directory, and then discard and reinstall the BUNDLE resource.

The new version of the application is now deployed to the Liberty server. This method is a quick way of redeploying changes to an application, but has the disadvantage that it will cause an outage in a production environment.

Version the bundle and context root

The next two methods allow you to maintain multiple versions of the bundle as one. Within CICS, managing changes to CICS bundles is accomplished by using version control if possible. Each CICS bundle project has an ID and version information that uniquely identifies it. This means that the CICS bundle ID does not need to be unique because it is qualified by the version. This allows you to have more than one version of the CICS bundle installed at the same time.

The second approach to using version control is to create a new version of the bundle with a new URL for the context root. Using this approach allows for two versions of the application to be active at the same time. The application can then be switched between different versions by updating the reference to the URL. Use the following steps to perform this process:

1. Create a new version of the web application with a new URI context root
2. Add the web application to a new version of your CICS bundle project (using the same ID but a new version)
3. Export to a new location in zFS
4. Create a new BUNDLE resource referring to the new zFS location
5. Install new CICS BUNDLE, and the web application is enabled because the URL is unique

The updated version of the bundle is deployed alongside the old version. This versioning method results in no outage to the users, but since Liberty does not allow web applications with duplicate URLs to be installed, it does require a different URL for the context root of the new version. Therefore, you need some forwarding proxy to handle this externally to CICS.

When you are satisfied that the new version of the application is working, you can disable and discard the BUNDLE resource for the old version of the application. In addition, to ensure that the new version of the bundle is installed if a cold or initial start occurs, ensure that you update any CICS group lists that reference the CSD groups that contain the BUNDLE definition for the old version to reference the CSD groups that contain the new bundle definition.

Table 2-1 shows a versioning scheme that can be used with a web application deployed in an EBA in a CICS bundles project.

Table 2-1 Versioning scheme for CICS bundles with EBAs

Item	**Configuration attribute**	**V100**	**V101**
CICS bundle resource definition	BUNDLE resource in CSD	MYBUN100	MYBUN101
CICS bundle zFS location	BUNDLEDIR attribute on BUNDLE resource	mybundle.eba.cicsbundle_1.0.0	/mybundle.eba.cicsbundle_1.0.1
CICS bundle ID	cics.xml id=	""mybundle.eba.cicsbundle	mybundle.eba.cicsbundle
CICS bundle version	cics.xml bundleMajorVer="1" bundleMinorVer="0" bundleMicroVer="n"	1.0.0	1.0.1
EBA symbolic name	OSGi application manifest: Application-SymbolicName:	mybundle.eba	mybundle.eba
EBA version	OSGi application manifest: Application-Version:	1.0.0	1.0.1
WAB bundle ID	WAB MANIFEST.MF Bundle-SymbolicName:	mybundle.wab	mybundle.wab
WAB bundle version	WAB MANIFEST.MF Bundle-Version:	1.0.0	1.0.1
Context root URI	ibm-web-ext.xml <context-root uri=" " />	app100	app101

Cloned JVM servers

The third method for redeploying your web applications is to use cloned JVM servers and use a load balancing mechanism to distribute the web requests between the servers. An instance of a JVM server can be easily created or cloned. The cloned JVM servers will appear identical to the user. As users request a CICS web application, they will not distinguish what JVM clone is servicing their requests.

Start the newly cloned JVM server, and install the new version of your application. Meanwhile, continue to run the old version in the old JVM server. When you are happy with the newly cloned server and updated application, phase out the old JVM server.

This solution can handle an outage, although, during initialization of the Liberty JVM server, there will be a period during which the HTTP listener is active and the web applications might not yet have finished installing. In this case, Liberty returns the error Context Root Not Found until the application is successfully started.

2.7 Migrating a Java EE application to Liberty

To migrate a Java EE application from a third-party Java Platform, Enterprise Edition server to Liberty in CICS, the Java EE environment that is required by the application must be provided by Liberty. This environment consists of the Liberty features installed in the server, instance configuration for specific resources (for example, data sources, connection factories) and other data, such as system properties. The primary mechanism for configuring the Liberty server environment, the `server.xml` configuration file, is different from that used by other Java Platform, Enterprise Edition application servers so the required environment must be migrated to server.xml definitions.

This section gives an overview of the main considerations when migrating a Java Platform, Enterprise Edition application to Liberty in CICS and the tools available to help with this. The main considerations are:

- Application validation
- Java EE server environment migration
- External dependency migration

These are considered in turn, although they would not necessarily always be performed in this exact order.

2.7.1 Application validation

The first step in migrating a Java Platform, Enterprise Edition application to Liberty in CICS is to check that the application is eligible for migration, that is, it uses only Java Platform, Enterprise Edition features that are supported in Liberty in CICS. This can be done manually by checking the application design documentation, source code, and so on, or tools can be used to assist.

WebSphere Liberty provides a suite of tools to assist in migrating web applications from WebSphere Application Server full profile or other Java Platform, Enterprise Edition servers to the Liberty profile. The rest of this section gives an overview of these tools and how they can be used to help configure Liberty in CICS. In Chapter 8, “Implementing security options” on page 143, we guide you through the process of using the tools to help migrate a web application.

Migration toolkit for application binaries files (technology preview)

This command-line tool is a Java application that can analyze the packaged binary of an existing Java Platform, Enterprise Edition application and produce a report in the form of an HTML file:

- An overview Technology Report that identifies the Java Platform, Enterprise Edition technologies, roughly equivalent to Liberty features, which are used by the application, and whether those technologies are supported in the target server environment.

- A more detailed Analysis Report for migrating the application from a specified source server to a specified target server. This tool incorporates a set of source-target specific rules, which are used to check the application binary and produce a report of the rules that match.

The command line Analysis Report tool is a limited version of the Eclipse-based Migration Toolkit below, capable of analyzing migration only from WebSphere V7.0+ and Java SE V6+ to Liberty.

WebSphere Application Server Migration Toolkit

This Eclipse-based tool performs the same task as the second of the command-line tools listed, although capable of analyzing rules for migration from earlier versions of WebSphere, and third-party servers. Because it is integrated into Eclipse, it can also pinpoint source code locations where changes need to be made.

For more information about the WebSphere Application Migration Toolkit and its use, see:

https://developer.ibm.com/wasdev/docs/move-applications-liberty-using-migration-toolkit

Note: Because the WebSphere Application Server Migration Toolkit is not tailored for CICS, it does not report features that are unsupported in Liberty in CICS, nor any restrictions that apply to the available features. For details about the available features and restrictions that need to be checked against the application dependencies, see the latest CICS IBM Knowledge Center page at this site:

http://www.ibm.com/support/knowledgecenter/SSGMCP_5.3.0/com.ibm.cics.ts.java.doc/topics/liberty_features.html

What to do if your application depends on unsupported features

In this case, the application has to be remediated to remove dependencies on the unsupported features before it can be migrated. This is a manual development task, which can be assisted by the tool in the WebSphere Application Server Migration Toolkit. The tool can help by suggesting source code changes as quick fixes.

2.7.2 Server environment migration

The next stage of migration is to extract details of the server configuration from the original Java Platform, Enterprise Edition server, and replicate these in server.xml. Only those elements of the configuration that are required by the application need to be migrated. The required subset can be determined referring to the results of the application validation stage above.

There are a number of ways to determine which features and resources need to be configured in Liberty:

- Manual analysis of the application, with information from the application migration exercise

  The results of the validation step above will show which features and resources are required by the application. This informs you which of the configuration properties from the original Java Platform, Enterprise Edition server will need to be migrated.

- From the existing Java Platform, Enterprise Edition server configuration

  The existing Java Platform, Enterprise Edition server configuration can be checked from its configuration files or administrative console. Alternatively, in the case of the WebSphere Application Server full profile, the configuration can be extracted from the running server as a set of properties, using wsadmin AdminTask.extractConfigProperties.

Equivalent configuration declarations can then be added to server.xml.

- Using the WebSphere Configuration Migration tool

  WebSphere Liberty provides an Eclipse-based tool, which can analyze the configuration of another Java Platform, Enterprise Edition server and generate configuration fragments suitable for inclusion in a WebSphere Liberty configuration.

  An overview of the migration tool is available at this site:

  https://developer.ibm.com/wasdev/downloads/#asset/tools-WebSphere_Configuration_Migration_Tool

In Chapter 8, “Implementing security options” on page 143, we give an example of updating a `server.xml` Liberty configuration file to support a web application using servlets, JSP, and a JCA connection.

2.7.3 External dependency migration

So far in this section, we discussed the issue of migrating the application view of a Java Platform, Enterprise Edition server, comprising the Java Platform, Enterprise Edition features and resources required by the application. In some cases, an application has external dependencies, for example on a database, web services, or JCA connection. When the application is moved to Liberty in CICS, it might make sense for some of these dependencies to be migrated at the same time.

Therefore, the last stage in the migration is to consider whether any of the application’s external dependencies should be migrated along with the application, for example:

- A JDBC connection to a local database might be migrated to DB2 on z/OS accessed using a Type-2 JDBC connection
- Web services calls to CICS programs can be replaced with direct JCICS calls
- A JCA connection using CTG can be changed to use a local CICS ECI connector

The migration of these external dependencies needs to be considered on a case-by-case basis. The resources that are potential candidates for migration are those that have earlier been configured in server.xml, plus other external dependencies, such as web services endpoints, which might not explicitly appear in server.xml.

3

Workload management

Workload management is the process of defining performance goals for the items of work in a system (such as in a CICSplex or across the network). Using a workload manager technology, the workload can be allocated to meet those performance goals. Workload management consists of two main features:

- Load balancing
- Affinity management

Load balancing (workload balancing) refers to the distribution of requests across multiple, available target regions. This allows you to increase the overall throughput of the system. There are different methods of workload balancing:

- The first approach is to use a predefined round-robin or randomized technique.
- The second approach is to use an advisor that checks the availability of the target servers before selecting the server to route the request to.
- The third approach is to have performance monitors in each server providing statistics that can be used to select a server based on its overall level.

Affinity management provides a way to manage your requests to a specific server instance, such as a CICS region or Liberty server. Using affinities means that you have the ability to route subsequent requests in a logical conversation to the same server that serviced the first request. This is a characteristic of a CICS application that constrains transactions to run in a specific region or server due to the presence of shared state, such as temporary storage. For web applications, affinities can be an issue when load balancing HTTP requests because of the use of HTTP session data. For more information about this, see 3.2.3, “HTTP session management” on page 52.

Within CICS, the ability to distribute the workload across multiple resources is important because it can prevent a CICS region from becoming constrained, and allow multiple CICS regions to be better used. There are different methods that can be used to provide high system availability and workload management for CICS web applications running in a Liberty JVM. In this section, we discuss workload balancing as it relates to the following technologies:

- IP load balancing
- Web server plug-in
- CICSPlex SM workload management

3.1 IP load balancing

To avoid being dependent on a single CICS router region, consider using more than one Liberty owning region to share the incoming HTTP connection from the network. You can use several techniques to balance the connections between your Liberty owning regions. The z/OS Communications Server integrates TCP/IP and IBM Workload Manager (WLM) on z/OS to provide two main technologies that can be used:

- Port sharing
- Sysplex Distributor

3.1.1 Port sharing

TCP/IP port sharing is a simple way to balance HTTP connections across a group of cloned Liberty JVM servers, providing failover and workload balancing across multiple Liberty owning regions within a single logical partition (LPAR). This load balancing is based primarily on the number of established sockets to each port.

Each Liberty server is configured to listen on the same TCP/IP port number, with either the `SHAREPORT` or `SHAREPORTWLM` option specified in the TCP/IP stack profile. This enables a group of cloned regions to listen on the same port. As the incoming client connections arrive for this port, TCP/IP distributes them across the available CICS regions listening on this shared port using a weighted round-robin approach. TCP/IP selects the CICS region with the least number of connections at the time of the incoming client connection request. `SHAREPORT` selects a CICS region based on a round-robin distribution. Figure 3-2 on page 48 illustrates configuring port sharing across cloned CICS regions. This configuration allows multiple Liberty servers to listen on the same port from the same IP stack. In the example that is shown in Figure 3-2 on page 48, CICSA and CICSB have Liberty servers that are listening on port 1234. All requests addressing this IP stack's port 1234 are distributed between these CICS regions, in a round-robin fashion.

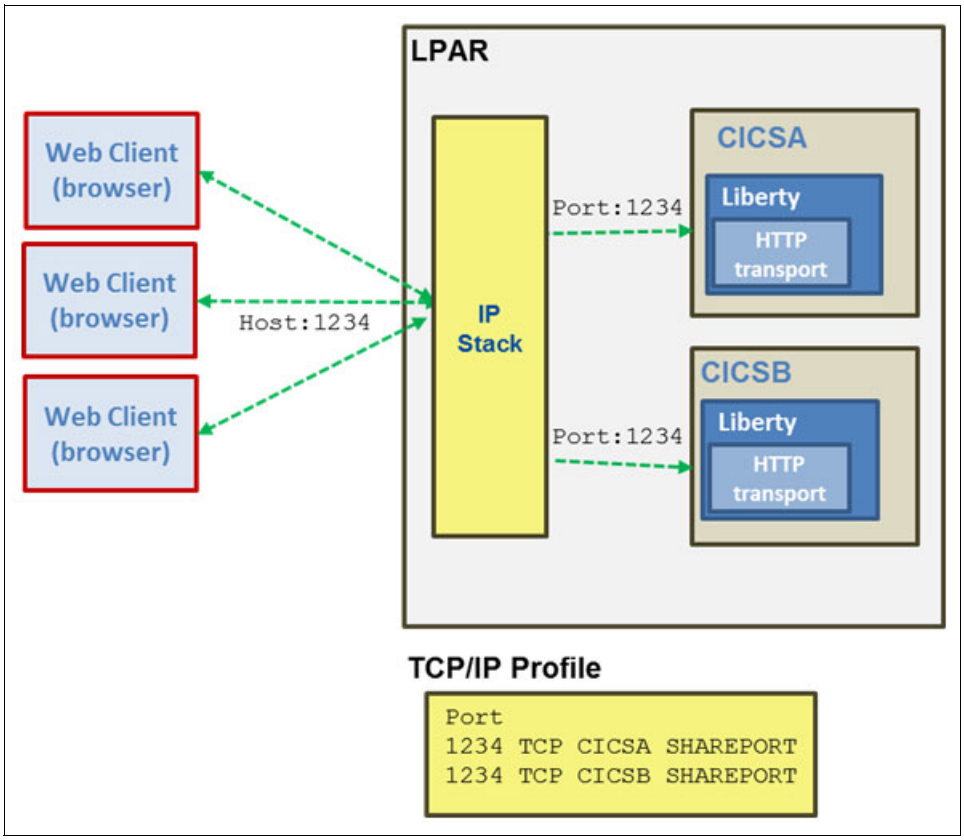


Figure 3-1 TCP/IP SHAREPORT configuration

Port sharing is ideal for failover scenarios because it can rapidly detect failing regions. It also works in tandem with the second option provided by TCP/IP, Sysplex Distributor, to provide a combined high-availability approach across CICS regions in the same LPAR and across different LPARs in a sysplex.

3.1.2 Sysplex Distributor

The second option is to set up an Internet Protocol network with Sysplex Distributor, which is designed to address the requirement of one single network-visible IP address for the whole of the sysplex.

Sysplex Distributor uses dynamic virtual IP addresses (DVIPAs) to increase availability and help distribute the workload by allowing connections to be distributed across listeners on multiple IP stacks within a sysplex. This option must be used if the CICS regions reside on separate LPARs or connect to separate IP stacks within the same LPAR.

With this function, you can implement a DVIPA as a single network-visible IP address that is used for a set of servers belonging to the same sysplex. A client on the IP network sees the sysplex as one IP address, regardless of the number of servers in the backend. With Sysplex Distributor, clients receive the benefits of workload distribution and a failover mechanism that is positioned at the entry to the sysplex.

There are several advantages to using Sysplex Distributor over other load balancing implementations:

- Cross-system coupling facility (XRF) links can be used between the distributing stack and target servers.
- Sysplex Distributor provides a total z/OS solution for TCP/IP workload distribution.
- Sysplex Distributor provides real-time load balancing for TCP/IP applications, even if clients cache the IP address of the server.
- Sysplex Distributor provides for takeover of the virtual IP address (VIPA) by a backup system if the distributing stack fails.
- Sysplex Distributor enables nondisruptive take back of the VIPA original owner to get the workload to where it belongs. The distributing function can be backed up and taken over.

HTTP listeners for multiple Liberty servers can be configured to listen on the same distributed DVIPA and port, and each VIPA is defined with the `VIPADISTRIBUTE` option. The TCP/IP stack then balances connection requests across the listeners.

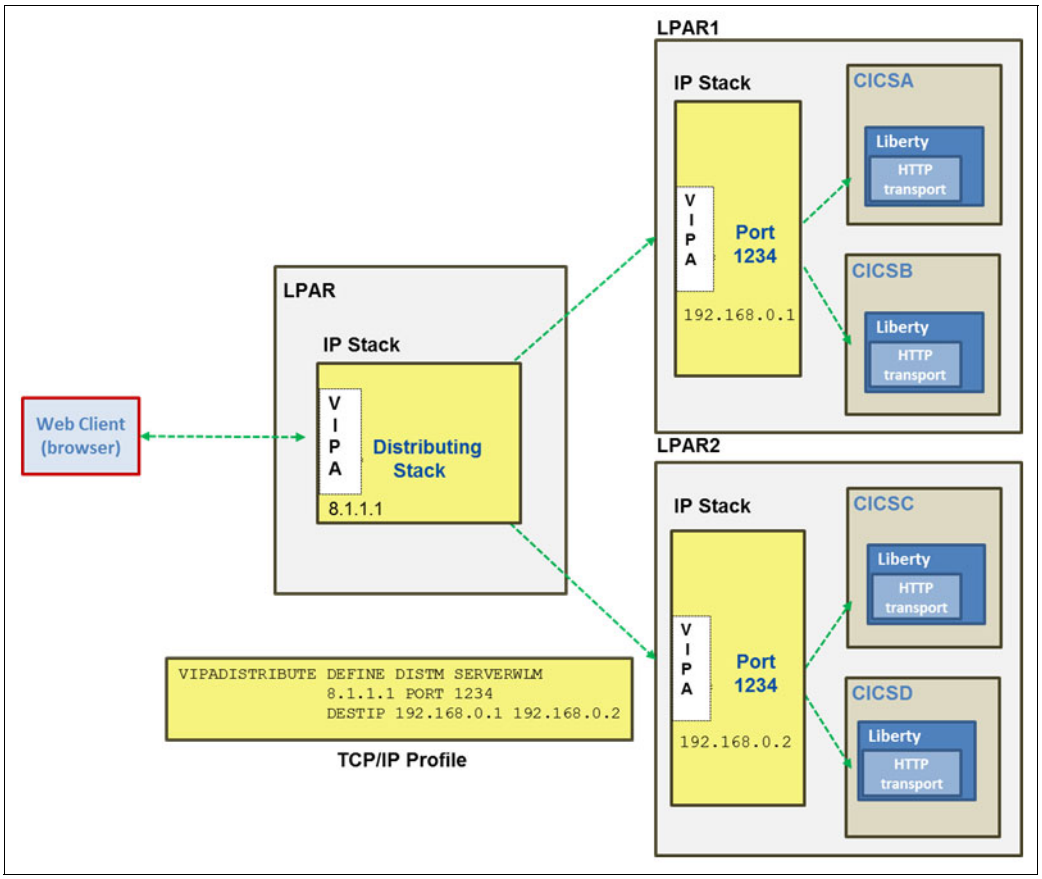


Figure 3-2 Sysplex Distributor with TCP port sharing

3.2 Web server plug-in

The web server plug-in is provided with WebSphere Application Server or WebSphere Application Server Liberty Profile and is licensed for usage with CICS TS. The plug-in acts as an HTTP client that forwards the HTTP requests from the web server to the application server, which in this case is the Liberty server in a CICS region. It is implemented as a library

that needs to be installed into the web server and is loaded at startup, and uses the HTTP protocol to redirect HTTP requests from the web server to the Liberty server in CICS.

The web server can either run on the same z/OS system as the application server Figure 3-4, or on a different system than the application server, as shown in Figure 3-3.

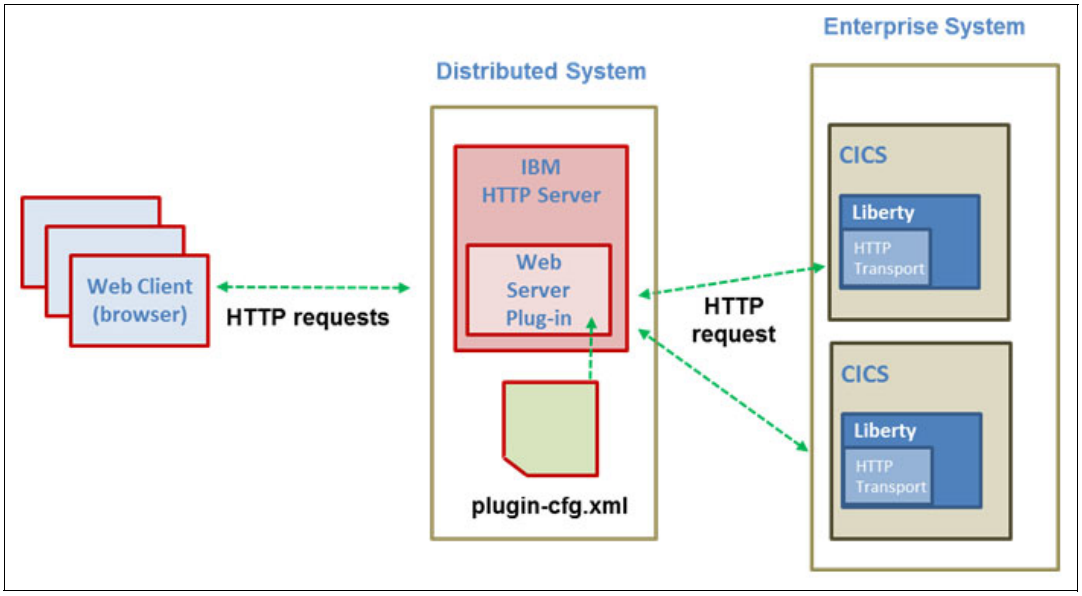


Figure 3-3 Remote web server plug-in scenario

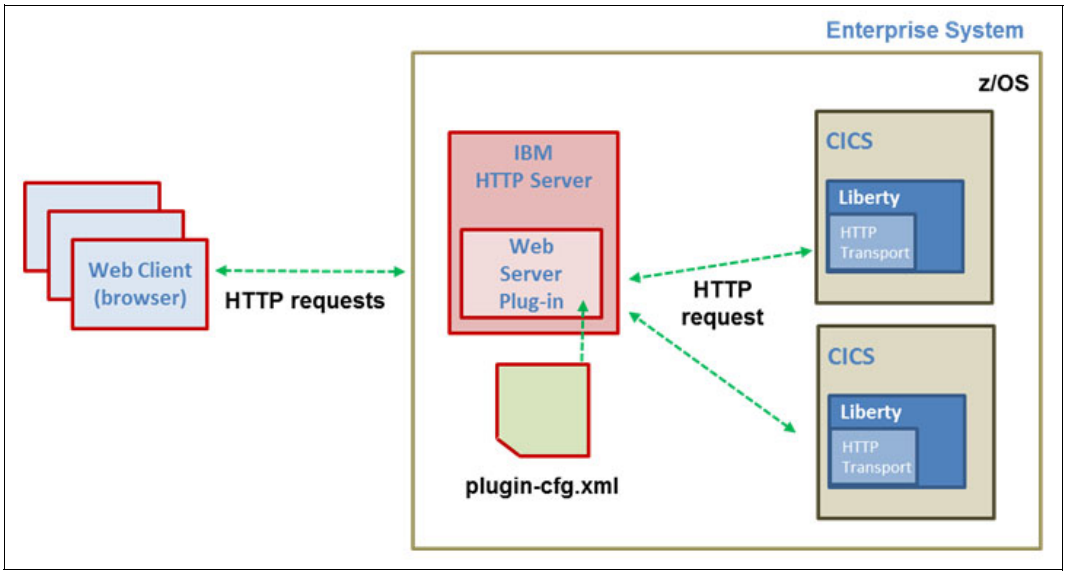


Figure 3-4 Local web server plug-in scenario

There are several reasons why you might want to use a web server and associated plug-in with a web application deployed to a Liberty JVM server:

- The plug-in provides workload management and failover capabilities by distributing requests evenly to multiple Liberty servers and routing requests away from a failed Liberty server.
- The web server provides an additional hardware layer between the web browser and the application server, which can be incorporated into a secure DMZ architecture. Existing

HTTP-based security procedures, such as SSL client authentication, can be used in the DMZ layer to protect the enterprise systems and then valid requests forwarded onto the enterprise system by the plug-in.

- Encrypted SSL connections can be terminated at the web server, thus removing the requirement for SSL processing in the enterprise system.
- The plug-in provides integration with a web server for the serving of static content without doing a full round-trip to the application server and back. For more information about optimizing the serving static content, see Chapter 7, “Configuring the web server plug-in” on page 133.

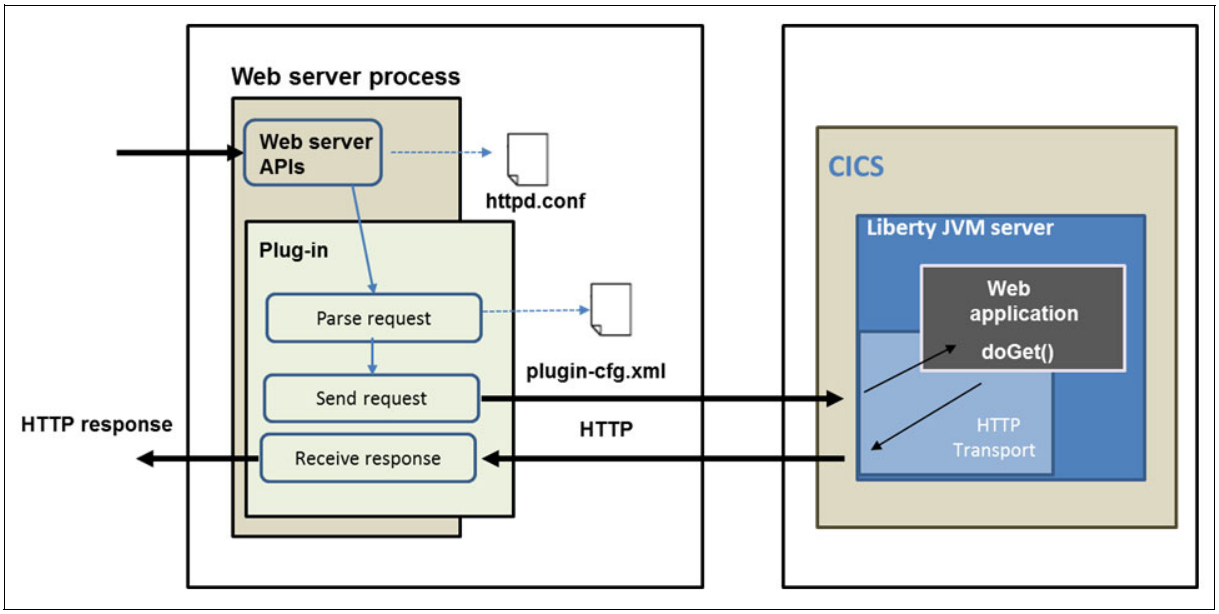


Figure 3-5 Web server plug-in processing

The web server plug-in consists of the following components (Figure 3-5):

- Web server process: This receives the request first
- The `httpd.conf` file: Contains the reference to the `plug-in.so/dll` file and reference to the plug-in configuration file
- The web server plug-in: This is loaded by the web server during startup and runs inside the same process as the web server
- The `plug-in-cfg.xml` file: The plug-in configuration file, which contains information about which URLs are served by the application servers
- The web container: In the CICS scenario, this is the Liberty JVM server that handles the request

The web server gives the plug-in an opportunity to handle every request first. It takes a request and checks the request against the configuration data. If the configuration data maps the URL for the HTTP request to the host name of an application server, the plug-in then uses this information to forward the request on to the application server. But if the plug-in is not configured to forward the request (URL), the request goes back to the web server for servicing.

The plug-in configuration file needs to be regenerated and propagated to the web servers when there are changes to your Liberty server configuration that affect how requests are routed from the web server to the application server. These changes include:

- Installing an application
- Creating or changing a virtual host
- Creating a new Liberty server
- Modifying HTTP transport settings

3.2.1 Load balancing

To do load balancing using plug-in files, the web server maintains information about the back-end web containers and applications. Based on these configured settings, the web server forwards requests to its corresponding back-end web container. The plug-in configuration file contains all of the information about the target servers, such as the application URLs, session affinities, and weightings.

The plug-in can be configured to route requests using one of two load balancing methods:

- Round-robin load balancing is the default algorithm where the workload is distributed across the group of servers based on the configured `LoadBalanceWeights`. The first request that arrives is routed to a randomly selected server. After that, the plug-in uses the next server in the list, and so forth. Each time that a server is chosen, the load balance weight is decremented by one. Any server with a weight below 0 is longer considered for new route requests. But, affinity requests still go through to that server. When all servers have a weight of less than 0, the plug-in resets the weights to the starting values.
- Random load balancing is where the requests are forwarded among the group of servers on a random basis. This approach does not use the load balance weight.

Using the web server plug-in, you can distribute the load between a single web server and multiple Liberty JVM servers running in CICS on z/OS. However, the load balancing options are impacted by the session affinity. After a session is created at the first request, all the subsequent requests have to be served by the same Liberty server. The plug-in retrieves the Liberty server that serviced the previous request by analyzing the session identifier and tries to route to this same server.

3.2.2 Failover

When the plug-in can no longer connect to or receive a response from a specific Liberty server, the session affinity is broken. This is considered a plug-in failover. This results in the Liberty server being marked as down. Therefore, the web server plug-in no longer attempts to route requests to that server. There are three plug-in properties that affect failover directly.

`ServerIOTimeout` is the number of seconds the plug-in waits for a response from the Liberty server before it times out the request. The recommended value is 120 seconds. Failover typically does not occur the first time that this time limit is exceeded for either a request or a response. Instead, the web server plug-in tries to resend the request to the same Liberty server, using a new stream. If the specified time is exceeded a second time, the web server plug-in marks the server as unavailable, and initiates the failover process.

`ConnectTimeout` is the number of seconds the plug-in should wait for a response when trying to open a socket connection to the Liberty server. The default is 0, which means to never time out. It is recommended to keep this value small. If a request connection exceeds this limit, or the Liberty server returns an HTTP 5xx response, the web server plug-in marks the server as down, and attempts to connect to the next application server in the list. If the plug-in

successfully connects to another Liberty server, all requests that were pending for the down Liberty server are sent to this other server.

`RetryInterval` is the number of seconds the plug-in waits before attempting to route requests to a Liberty server that was previously marked as down. The recommended value is 60 seconds as most servers are recoverable.

If a Liberty server responds to an HTTP request with an application failure error, such as the HTTP Not Found error 404, the plug-in does not view this as a permanent failure. This response code will be returned by a Liberty JVM server for unknown URLs during the JVM server startup process as Liberty applications are being started.

3.2.3 HTTP session management

In some web applications, application state date will be dynamically created as users navigate through the website. Where the user goes next, and what the application displays as the user's next page, or next choice, depends on what the user has chosen previously from the site. In order for this to occur, a web application needs a mechanism to hold the user's state information over a period of time. However, HTTP does not recognize or maintain a user's state. It treats each user request as a discrete, independent interaction.

The Java EE specification provides a mechanism known as a *session* for web applications to maintain a user's state information. A session needs to be serviced all within the same JVM server. Otherwise, each JVM will see only one or two requests and get confused. This is known as a *session affinity*. The web server plug-in assures the session affinity is maintained in the following way: Each server Clone ID is appended to the session ID. When an HTTP session is created, its ID is passed back to the browser as part of a cookie or URL encoding. When the browser makes further requests, the cookie or URL encoding is sent back to the web server. The web server plug-in examines the HTTP session ID in the cookie or URL encoding, extracts the unique ID of the cluster member handling the session, and forwards the request.

The application server ID can be seen in the web server plug-in configuration file, as shown in Example 3-1.

Example 3-1 Server ID from the plug-in-cfg.xml file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--HTTP server plug-in config file for the cell ITSOCell generated on 2004.10.15
at 07:21:03 PM BST-->
<Config>
......
   <ServerCluster Name="MyCluster">
      <Server CloneID="vuel491u" LoadBalanceWeight="2" Name="NodeA_server1">
         <Transport Hostname="wan" Port="9080" Protocol="http"/>
         <Transport Hostname="wan" Port="9443" Protocol="https">
         ......
</Config>
```

Liberty keeps information about the user's session on the server. It passes the user a session ID, which correlates an incoming user request with a session object maintained on the server. By default, Liberty stores session objects in memory. When session data must be maintained across a server restart or an unexpected failure, you can configure the Liberty server to persist the session data to a database using the sessionDatabase-1.0 feature. This configuration allows multiple servers to share the same session data, and session data can be recovered if there is a failover.

For more information about how to set up and configure the web server plug-in for usage with CICS Liberty, see Chapter 8, “Implementing security options” on page 143.

3.3 CICSPlex SM workload management

Workload management in CICS can be achieved by dynamically routing work between several CICS regions. You have to manage several CICS address spaces to manage the workload. You can choose to run these regions as separately connected systems in a multiregion operation (MRO) complex, or you could define them as CICSPlex System Manager (CICSPlex SM)-managed regions. Multiple CICS regions can communicate with each other and cooperate to handle inbound work requests. This specialized type of cluster is called a *CICSplex*.

Typically a CICSplex consists of one or more terminal-owning regions (TORs) connected to a group of application-owning regions (AORs). The workload originates in the TORs and is then routed to one of the available AORs. For web applications, the work is not initiated from a terminal, but from a web browser that sends an HTTP request to the HTTP listener in a Liberty JVM server. We term each of these CICS regions hosting a Liberty JVM server as a Liberty-owning region (LOR), each of which can then use dynamic routing of distributed program link (DPL) requests to the business logic in a set of remote AORs.

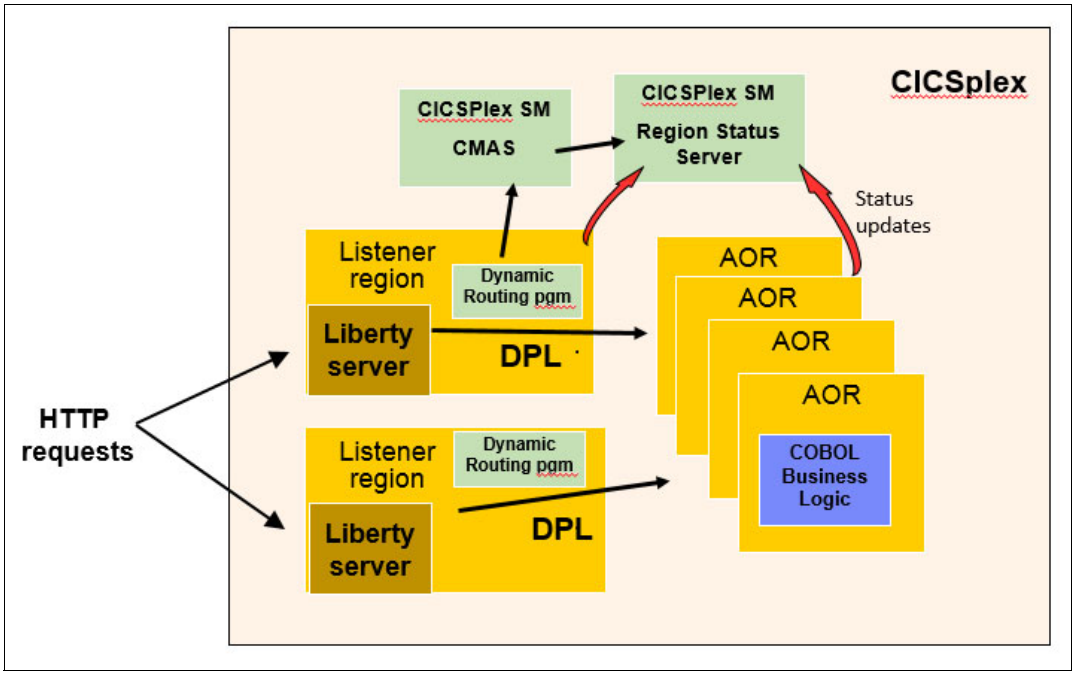


Figure 3-6 CICSPlex SM dynamic routing with Liberty

Using dynamic routing can provide high availability as well as manage the workload within a CICS sub-system. CICSPlex SM Workload Management (WLM) can be used to distribute the workload between multiple target regions. It augments its workload management routing decisions by using the current status information posted from the CICS regions. To optimize the workload routing in a sysplex, you must configure and monitor a region status (RS) server as part of a coupling facility data table.

CICSPlex SM Workload Management facilities create a list of suitable target candidate CICS regions based on:

- The transaction
- The terminal ID, LU-name, user ID, or process type

The list of candidate targets is based on the workload to which the requesting or routing region belongs.

CICSPlex SM does not do the routing. CICS still performs the routing based on information received from CICSPlex SM.

CICSPlex SM Workload Management is based on workloads. The workload determines the routing regions the workload will apply and the behavior of the work entering the routers using routing rules. When you establish a workload, you associate the work itself with the CICS regions to form a single, dynamic entity. Within this entity, you can route the work in two ways:

- Using workload balancing: You can route the task to a target region that is selected based on the availability and activity level of the regions.
- Using workload separation: You can route the task to a subset of the target regions based on a specific criteria.

3.3.1 Workload balancing

Workload balancing is the process that decides which target region is considered the most suitable to route the task to, assuming that it is possible to send work there and that it does not have an affinity in place. It is the most basic configuration because it allows you to balance the dynamic transactions and program links across all of the available target CICS regions that are associated with the workload. It does not distribute the work evenly or consistently. It simply provides CICS with the *best target region* at the moment of the request from all of the possible candidates.

Workload balancing uses a combination of health factors to determine which region is the best fit and requires that three items are in place:

- A workload specification (WLMSPEC) to identify the default target scope for the workload. This is associated with each CICS region that specifies EYU9XLOP as the dynamic routing program or distributed routing program in the system initialization table (SIT).
- A CICS system definition (CSYSDEF) for each CICS system that specifies the Workload Manager status of the system.
- The CICS system connects to a CICSPlex SM address space (CMAS) allowing the workload management facility to initialize for transaction routing.

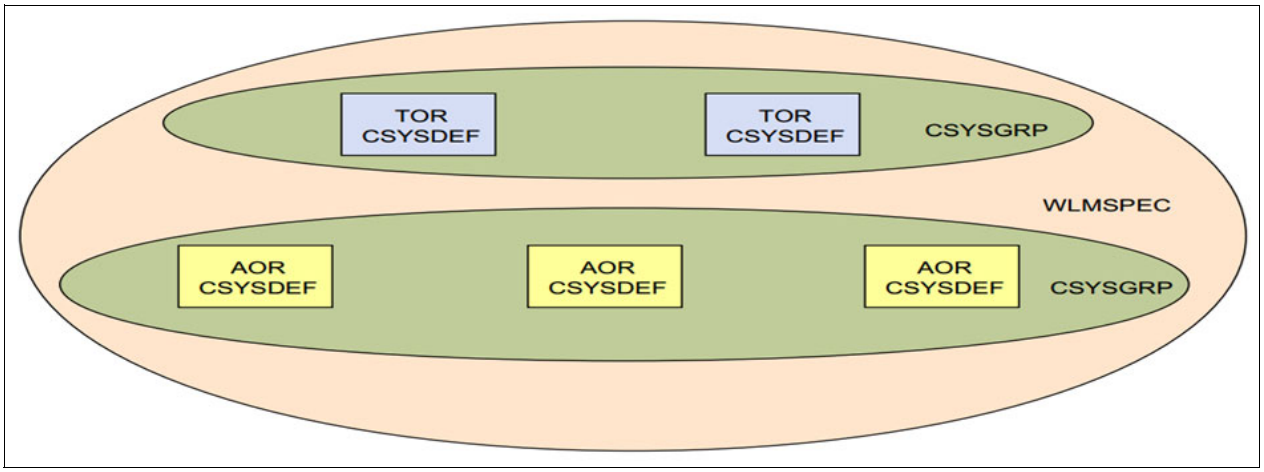


Figure 3-7 Monitoring of health factors for workload balancing

3.3.2 Workload separation

Workload separation allows you to separate tasks and program occurrences and direct them to a subset of target regions. The workload can be balanced across this subset of target regions. The target scope for a separated workload can vary from a single AOR to a large AOR group consisting of multiple CICS regions. If an AOR group is the target, the routing algorithm is applied to select the most suitable region from those defined to it.

The criteria that you use to separate transactions or programs can be based on the following items:

- The terminal ID and user ID that are associated with a transaction or program occurrence
- The process type that is associated with the CICS business transaction services (BTS) activity
- The transaction ID
- A selected target region based on its affinity relationship and lifetime

To perform workload separation, you also need a few more resources:

- A transaction group (TRANGRP) to identify the characteristics of a set of dynamic transactions.
- The WLM definition (WLMDEF) to identify the target scope used to route the transaction in the TRANGRP.
- A WLM group (WLMGROUP) that contains one or more WLMDEFs that are to be installed together. The WLMGROUP can be associated with a WLMSPEC in case all of the child WLMDEFs are installed at workload initialization.

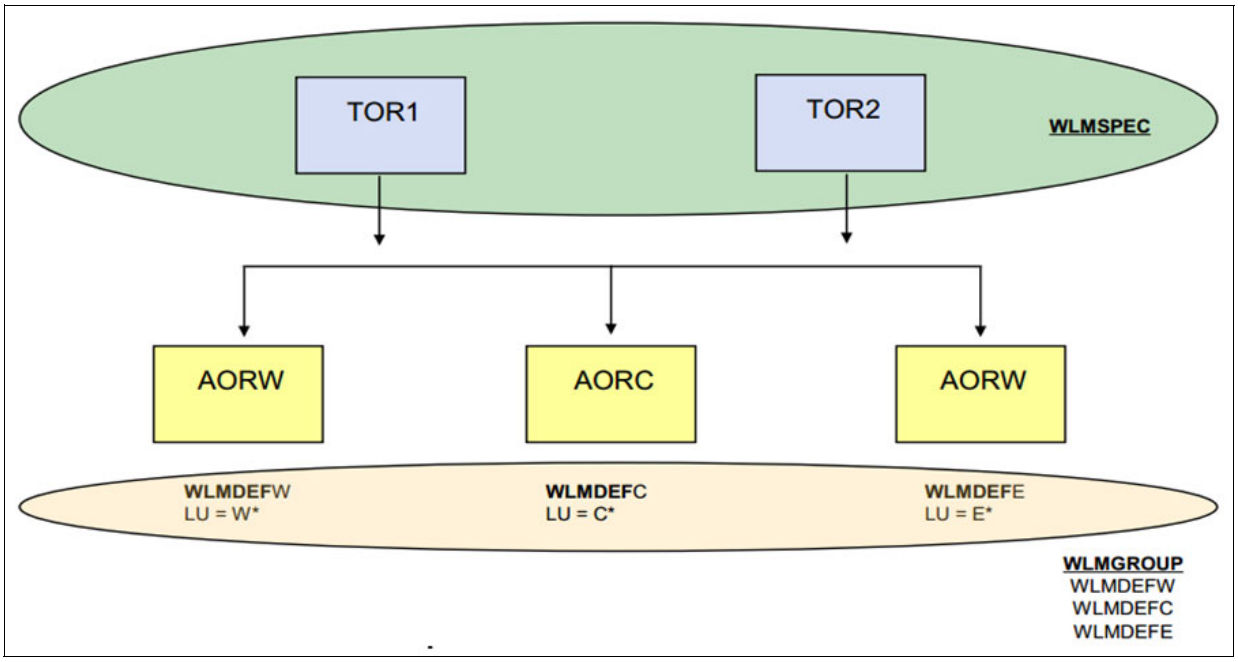


Figure 3-8 Resources needed for workload separation

Workload balancing and workload separation can be active concurrently in the same or different workloads associated with a CICSplex. Workload balancing can be used in addition to, or in place of, workload separation.

4

Security options

This chapter gives an overview of how the Liberty JVM server implements Java Platform, Enterprise Edition security, along with the integration it provides with traditional CICS security. This chapter highlights and explains choices that you can make to tailor Liberty JVM security to meet your needs.

For more information about how to set up and implement CICS Liberty security, see Chapter 8, “Implementing security options” on page 143.

This chapter describes the following topics:

- Java Platform, Enterprise Edition security
- Confidentiality
- Security registries

4.1 Java Platform, Enterprise Edition security

When you click a link to invoke a web application, what happens in CICS Liberty JVM server before that web application gets control? There is a series of steps, defined by Java Platform, Enterprise Edition specifications, which take place in the web container to ensure that the user is authorized to run the web application.

4.1.1 Deployment descriptor

The first item to understand in Java Platform, Enterprise Edition security is the web deployment descriptor. This is the `web.xml` file that is in the `WEB-INF` subdirectory of a web project. Example 4-1 shows our GenAppWeb example.

Example 4-1 Web deployment descriptor

```
<login-config>
   <auth-method>BASIC</auth-method>
</login-config>
   <security-constraint>
   <display-name>Broker Role</display-name>
   <web-resource-collection>
      <web-resource-name>GenAppWeb</web-resource-name>
      <url-pattern>/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
      <role-name> Broker</role-name>
   </auth-constraint>
         <user-data-constraint>
               <transport-guarantee>CONFIDENTIAL</transport-guarantee>
         </user-data-constraint>
</security-constraint>
```

The key items in the `web.xml` file in Example 4-1 are:

- Authentication method

  Using the <login-config> element and its <auth-method> sub element is key to defining how a user of this web application is authenticated by the web container. If the <auth-method> element is present, the user must be authenticated to access any resource that is constrained by a <security-constraint> rule in `web.xml`. The three authentication method options available in Liberty are BASIC for basic authentication, FORM for form-based authentication, or CLIENT-CERT for Secure Sockets Layer (SSL) client certificates. In addition, the Java Platform, Enterprise Edition specification supports DIGEST authentication. However, this is not available in Liberty.

- URL pattern

  The <url-pattern> defines the URLs in the web application to which this security constraint applies. In our example, all URLs are protected by using the /* prefix.

- Role name

  The <role-name> element is the access group the server uses for authorization to this application. In this example, all users need to be in the Broker role.

- Transport guarantee

  The <transport-guarantee> sub element of the <user-data-constraint> defines whether encryption is required when accessing this application. If CONFIDENTIAL is

listed, a protected transport layer connection, such as HTTPS is required when accessing this application. Thus, you need to define an <httpEndpoint> in the server.xml configured to use an httpsPort.

Note: Instead of using the <role-name> element in an authorization constraint in web.xml to grant access to specific roles, it is also possible to programmatically declare which roles are authorized to access Java components. This can be done by using either annotations, such as @RolesAllowed, or the isCallerInRole() method. For more information about securing Java EE components programmatically, see the following tutorial:

https://docs.oracle.com/cd/E19798-01/821-1841/gjgcq/index.html

For general details about Java EE 6 application security, refer to the Oracle Java EE 6 tutorial at:

https://docs.oracle.com/cd/E19226-01/820-7627/bncbk/index.html

4.1.2 Key steps in security processing

Figure 4-1 illustrates the steps in security processing of an HTTP request running in a Liberty JVM server.

The Liberty angel process is a key part of any Liberty server security infrastructure on z/OS. The angel process provides a Liberty server with access to z/OS authorized services, such as System Authorization Facility (SAF) authentication and authorization. Each Liberty server must register with the angel at startup to use authorized services. However, registration is limited to only one process per address space. This means that it is only possible to configure one Liberty JVM server per CICS region to use the function of the cicsts:security-1.0 feature.

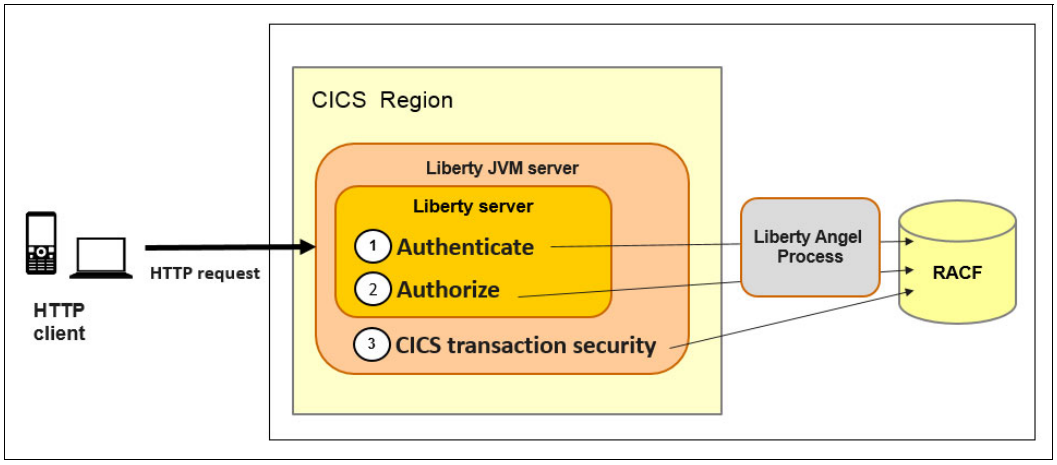


Figure 4-1 Security processing of HTTP requests

The three steps in Liberty server security processing are as follows:

- Authentication by the web container
- Authorization by the web container
- Security processing during CICS transaction attach

Authenticate

The first step is authentication. Authentication is the process of finding out and verifying the identity of the user who is attempting to run the application. This function is performed entirely by the Java Platform, Enterprise Edition security feature in Liberty, by enabling the appSecurity-2.0 feature in `server.xml`. This step also requires access to the angel process to provide access to SAF-authorized services if an SAF registry is used as the registry type.

Liberty can authenticate a user using the following methods:

- *Basic authentication*: The simplest example is basic authentication, whereby the web container rejects the initial request with an HTTP 401 status code. This authentication causes the browser to present a window requesting a user ID and password, which is then used by the browser to resubmit the request.
- *Form logon*: In this scenario, a customized HTML form is created, which is added to the application and used to supply the user ID and password.
- *SSL client authentication*: In this scenario, an SSL client certificate is mapped to a user ID in the local security registry. When this client certificate is presented, the request is authenticated by using the specified user ID.
- *Trust association interceptor (TAI)*: The trust association interface is a service provider API that enables the integration of third-party security services with a Liberty server. This function allows the authenticated subject to be programmatically determined for each request, providing for a simple means of identity assertion.

There are also three different registries available for verifying the user ID and password: SAF, LDAP, and a basic user registry. We discuss the advantages and disadvantages of these different methods of authentication in 4.3, “Security registries” on page 64.

If the user fails authentication, the Liberty server rejects the request. However, if the user passes authentication, a valid Java subject, containing the details of the user ID and authentication realm, is available. This is passed to the next step, authorization.

Authorize

The next step is authorization. At this point, we know who the user is and we have a valid user ID. Authorization determines if the authenticated user has the correct privileges to access the web application. This is determined by using roles. In our example, if the user is in the Broker role, that user is authorized to run the application. If the user is not a Broker, Liberty rejects the request.

To understand what authorization is, it is helpful to remember the security steps that happen before granting authorization:

1. A security constraint in the application `web.xml` file establishes the role or roles that are authorized to access a particular part of the URL path.
2. Authentication establishes the user ID that wants to access that path.
3. Authorization determines if the authenticated user ID belongs to the authorized role.

Authorization uses *user role mapping* to find out if a user, or group of users, belongs to a certain role. There are three options to set up this authorization in a Liberty JVM server:

1. CICS bundles and the cicsAllAuthenticated role

   If you deploy a web application as a CICS bundle, you need to add the cicsAllAuthenticated role to authorization constraint in your web deployment descriptor as follows:

   ```
   <auth-constraint>
      <role-name> Broker</role-name>
   </auth-constraint >
   ```

   CICS then creates the application definition statements in the `installedApps.xml` server configuration file. Then, CICS grants access to this role for all authenticated users using the special subject ALL_AUTHENTICATED_USERS (meaning all users who have passed authentication).

   Example 4-2 shows what the application element in `server.xml` looks like when GenAppWeb is deployed as a CICS bundle.

   Example 4-2 GenAppWeb deployed as CICS bundle in server.xml

   ```
   <application id="GenAppWeb" name="GenAppWeb" type="war"
        location="${server.output.dir}/installedApps/GenAppWeb.war"
        bundle="GenApp" token="22ABF9E000000100" bundlepart="GenAppWeb">
           <application-bnd>
               <security-role name="cicsAllAuthenticated">
                   <special-subject type="ALL_AUTHENTICATED_USERS"/>
               </security-role>
           </application-bnd>
       </application>
   ```

   There are two possible ways that Liberty JVM server in CICS performs authorization. One is called *SAF authorization*, which uses the EJBROLE classes in the SAF registry. The other is *Java Platform, Enterprise Edition role authorization*, which uses the roles defined in the `server.xml` configuration file.

   Using this option requires that you have the ability to update the deployment descriptor in your web application. In addition, this option delegates request authorization to CICS transaction and resource security checking (see , “CICS transaction security” on page 62) as from a Java Platform, Enterprise Edition security point of view. Any authenticated user has access to run the web application. However, it is also possible to combine CICS transaction security authorization with the usage of SAF authorization using EJBROLEs (option 3., “SAF authorization” on page 62). This is because setting the `<safAuthorization/>` element prevents access to the cicsAllAuthenticated role being granted to ALL_AUTHENTICATED_USER. Instead, the EJBROLE permission is used.

2. Liberty security role authorization

   The next option for user role mapping is to use <security-role> elements in the application definition in the `server.xml` or `ibm-application-bnd.xml` file in the enterprise archive (EAR). This method uses a definition in the `application-bnd` element of each application. These definitions establish which users or user groups are authorized to certain roles.

   Say that the web application security constraint defines the Broker role as follows:

   ```
   <role-name> Broker </role-name>
   ```

The application element might look like that shown in Example 4-3 if you wanted to grant access to the authenticated user1 and any user ID in the `webusers` group.

Example 4-3 Security roles defined in server.xml

```
<application id="GenAppWeb"
    name="GenAppWeb" type="war"
    location="/u/reds13/cicsts53/GenAppWebBundle_1.0.0/GenAppWeb.war">
   <application-bnd>
      <security-role name="Broker">
           <user name="user1" />
                <group name="webusers" />
      </security-role>
   </application-bnd>
</application>
```

Note: In a Liberty JVM server, if you want to use Java Platform, Enterprise Edition security roles without SAF authorization, you cannot use CICS bundles to install your applications, as the application-bnd element will always be created with the `cicsAllAuthenticated` role. However, this can be a useful method if you are porting an application directly from a third-party Java Platform, Enterprise Edition application server that uses this method.

3. SAF authorization

   Liberty uses SAF authorization when the `<safAuthorization/>` element is present in `server.xml`, and determines the EJBROLE class to be validated using the <safRoleMapper profilePattern=…> element. Authenticated users are then authorized to access an application by giving them access to the EJBROLE referenced in the profilePattern attribute. This option requires access to the SAF authorized services using the angel process. For more information about configuring SAF authorization, see Chapter 8, “Implementing security options” on page 143.

   EJBROLEs can be used with CICS bundle-deployed applications because the `<application-bnd>` and `cicsAllAuthenticated` role is ignored by Liberty in favor of using the EJBROLE mappings defined by the SAF registry.

CICS transaction security

Authentication and authorization are common to any Java Platform, Enterprise Edition application server. This third step, CICS transaction security, is unique to a Liberty JVM server in CICS TS. This step is where CICS builds the transaction environment that enables the web application to run as a CICS task and invoke CICS services. As such, it is a critical point at which CICS security can be used to ensure that the user ID is authorized to run the transaction and is implemented by using the cicsts:security-.1-0 feature, defined in `server.xml`.

As part of this process, the following key decisions need to be made by the CICS runtime during transaction attach processing:

- The CICS transaction ID must be chosen: The default is CJSA, but this can easily be overridden by using a URIMAP resource, which maps the URI and the protocol (HTTP or HTTPS) to a specified CICS transaction ID.
- The CICS task user ID must be set: The user ID is taken from the authenticated Java subject and used to set the CICS transaction security context during transaction attach processing. In doing so, the user ID needs read access to the SAF resource group (TCICSTRN) for the CICS transaction chosen at Step Ê. If the user ID does not have access to this profile, the request is rejected.

The result of this process gives you the following security choices:

- If there is an authenticated SAF user ID from the Java subject, this is used to run the CICS transaction. This should always be the case if there is a security constraint in the `web.xml` file.
- If there is no authenticated user ID, the USERID specified on the URIMAP definition is used to run the CICS transaction. This is the case, for instance, if there was no security constraint in the `web.xml` file.
- If there is no USERID on the URIMAP definition and no authenticated user ID, the default user ID for that CICS region is used to run the transaction. This is a default setup, and it is unlikely that the default user should be granted access in this scenario in a secure system.

4.2 Confidentiality

You can configure a Liberty JVM server to use SSL for data encryption, and optionally authenticate with the server by using a client certificate. Certificates can be stored in a Java keystore held on the zFS filing system or in a SAF key ring such as IBM RACF®.

If the security constraint in the `web.xml` file defines a transport-guarantee of CONFIDENTIAL, all communication with this web application needs to be using an HTTPS-enabled endpoint. See Example 4-4.

Example 4-4 Excerpt from web.xml

```
<user-data-constraint>
   <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

To make an environment secure, you must be sure that any communication is with trusted sites whose identity you can be sure of. SSL uses certificates for authentication. These are digitally signed documents, which bind the public key to the identity of the private key owner:

- SSL client authentication

  The process of SSL client authentication occurs when the underlying socket is first established. Authentication is performed by an exchange of certificates in X.509 format between the client and the server. A Liberty server allows you to specify that client authentication is required for connections to a particular HTTP endpoint by adding the `clientAuthentication="true"` attribute to the SSL configuration element in `server.xml`.

In addition, if you want requests to run under the identity of the user ID to which the personal certificates are mapped in the keystore, you can specify the value CLIENT-CERT on the auth-method element in the `web.xml` deployment descriptor, as shown in Example 4-5.

Example 4-5 Excerpt from web.xml

```
<login-config>
    <auth-method>BASIC </auth-method>
</login-config>
```

- As extra protection, it is also possible to request Liberty to drive an HTTP basic authentication challenge if the CLIENT-CERT identity is not authorized to the application. This is achieved by adding the following webAppSecurity element to `server.xml`.

  ```
  <webAppSecurity allowFailOverToBasicAuth="true" />
  ```

4.3 Security registries

Liberty provides a choice of security registries, which can be used to authenticate users accessing protected web applications.

4.3.1 Basic user registry

The basic user registry is part of the appSecurity-2.0 feature and provides a simple, text-based registry in the `server.xml` file. This can be found by using the `basicRegistry` element shown in Example 4-6.

Example 4-6 Basic user registry definition in server.xml

```
<basicRegistry id="basic" realm="customRealm">
     <user name="dan" password="p@ssw0rd" />
     <user name="andy" password="pa$$w0rd" />
     <user name="shayla" password="pa$$w0rd" />
     <user name="indi" password="{xor}Lz4sLCgwLTs=" />
     <group name="residents">
         <member name="dan" />
         <member name="andy" />
     </group>
</basicRegistry>
```

Because access to `server.xml` is not controlled by SAF, and because this registry is not integrated with CICS Liberty security or synchronized with SAF, it is not advisable to use this registry type for any purpose other than testing.

4.3.2 System Authorization Facility

Integration with a System Authorization Facility (SAF) user registry, such as RACF, is part of the Liberty zosSecurity-1.0 feature, which is integrated into the CICS Liberty security feature (cicsts:security-1.0). All of the supported Java Platform, Enterprise Edition authentication mechanisms, including basic authentication, form-based authentication, and SSL client authentication, can use an SAF registry to authenticate users. In addition, the EJBROLE support in Liberty can use the SAF registry to authorize access to the roles defined in

`web.xml`, and CICS transaction and resource security can use SAF to authorize access to CICS resources.

4.3.3 LDAP and distributed identities

Many Java Platform, Enterprise Edition application servers on distributed platforms, such as WebSphere Application Server, support the use of an LDAP security registry to authenticate users. Liberty on z/OS also supports this ability. However, it also supports a means to map these distributed identities in X.509 form to RACF user IDs that use mapping known as *distributed identity filters*. These filters describe the mapping association between a RACF user ID and one or more distributed identities.

Example 4-7 shows a RACMAP. This maps the LDAP user with the X.509 distinguished name `'CN=Alice,OU=ITSO,O=IBM,C=US'` to the RACF user ID ALICE.

Example 4-7 RACMAP example

```
RACMAP ID(ALICE)
      USERDIDFILTER(NAME('CN=Alice,OU=ITSO,O=IBM,C=US'))
      REGISTRY(NAME('ldaps://itso.ibm.com'))
      WITHLABEL('Alice')
```

This function is supported in CICS TS V5.3 by using the new cicsts:distributedIdentity-1.0 feature. It can be used with the cicsts:security-1.0 feature to take the RACF user ID from the mapping, then set this as the CICS transaction security context. This allows EJBROLEs and CICS transaction security to be used for authorization and an LDAP registry to be used for Java Platform, Enterprise Edition authentication.

Part 2

Up and running

In this part, we describe how to get your system up and running.

Part 2 contains the following chapters:

5

Developing and deploying applications

This chapter describes how to install and set up your development environment and deploy applications to a Liberty JVM server in CICS. We used the Eclipse integrated development environment (IDE) enhanced with the CICS Explorer software development kit (SDK) and the IBM WebSphere Application Server Liberty Developer Tools plug-in to develop the sample applications.

During testing, we used a local installation of WebSphere Liberty Profile. This chapter also describes the advantages and disadvantages of this installation type. Applications can be tested in the local server if you do not use the CICS specific library (JCICS) to access z/OS resources. Otherwise, testing must be performed in a Liberty Server running in CICS.

This chapter also shows how to build and export your bundles to zFS using the CICS Build Toolkit (CICS BT).

This chapter describes the following topics:

- Creating the development environment
- Creating a HelloWorld application
- Deploying the application to CICS

5.1 Creating the development environment

IBM provides the following tools for developing Liberty applications in CICS:

- CICS Explorer SDK
- WebSphere Developer Tools (WDT), which includes a WebSphere Liberty Profile local server runtime for testing and debugging applications

5.1.1 Installing the components

After you complete the steps in this chapter, you will have a complete workbench in which to develop and test your applications. The base development environment that we use is an installation of Eclipse IDE for Java EE Developers, which is available from the following website:

https://eclipse.org/luna

Follow these steps:

1. Install the CICS Explorer SDK

   The following steps show how to extend Eclipse with the IBM CICS Explorer SDK. IBM CICS Explorer provides tools and interfaces to CICS that help to manage CICS resources and applications, and to develop Java applications for deployment into CICS:

   a. Start your Eclipse and go to **Help** → **Install New Software**. The wizard opens. See Figure 5-1 on page 71.

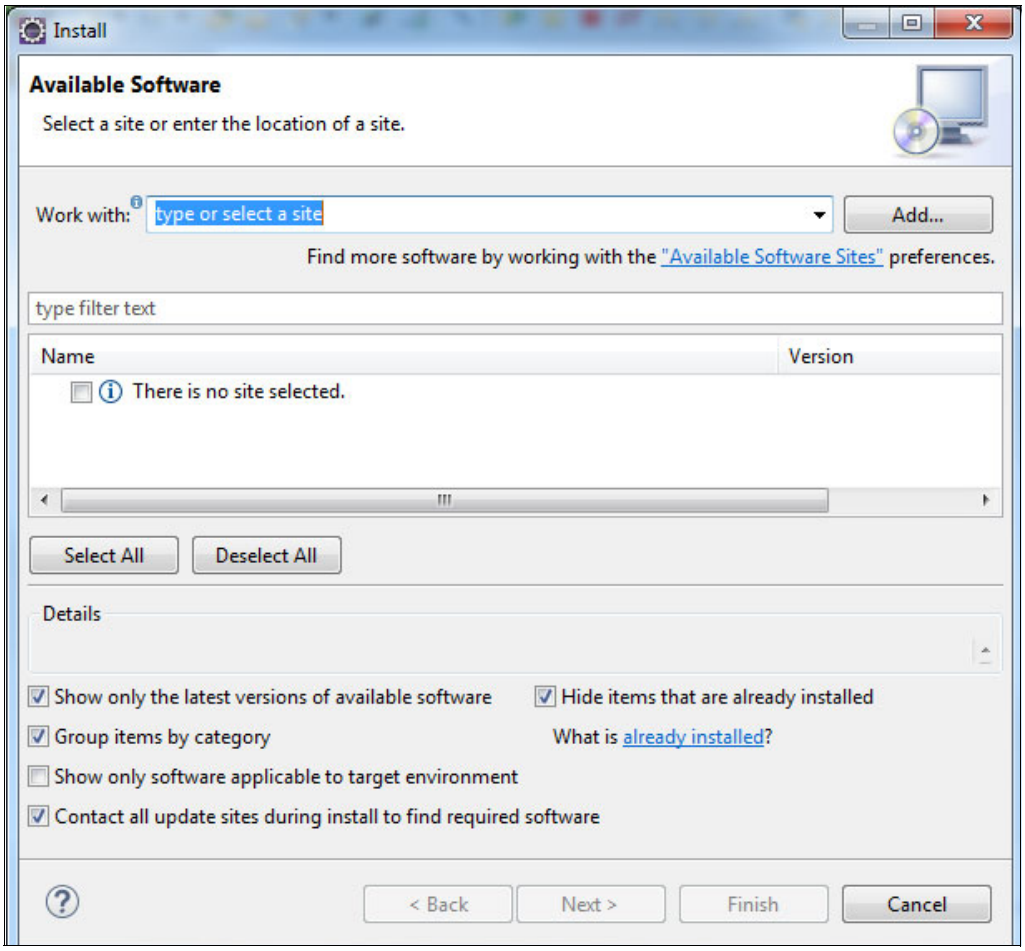


Figure 5-1 Install new software wizard

b. Click **Add**. In the Add Repository window, add the following information about the software update site to be used and click **OK**. See Figure 5-2.

Name: Explorer for z/OS Update Site

Location:
`http://public.dhe.ibm.com/ibmdl/export/pub/software/htp/zos/tools/aqua/`

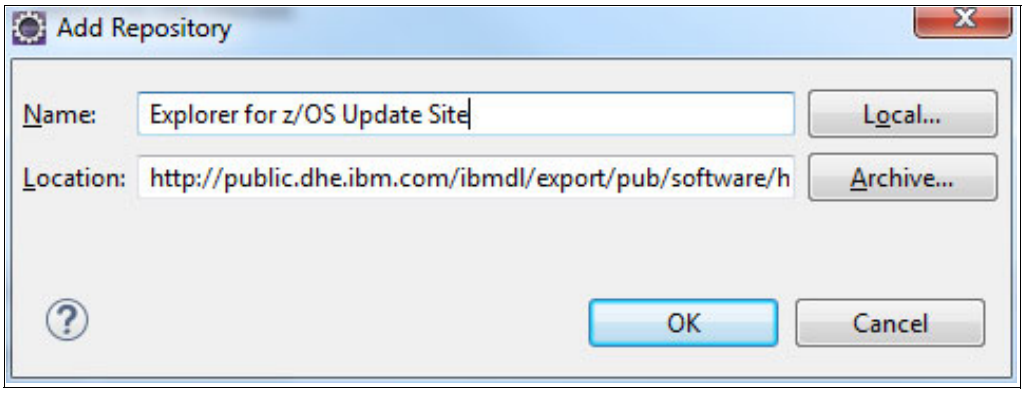


Figure 5-2 Add update site to the repository list

c. In the Available Software window, select **IBM CICS Explorer** and ensure that all the subcomponents are selected including the CICS SDK for Java and the CICS SDK for Servlet and JSP support (this is the component you need for developing web applications). See Figure 5-3.

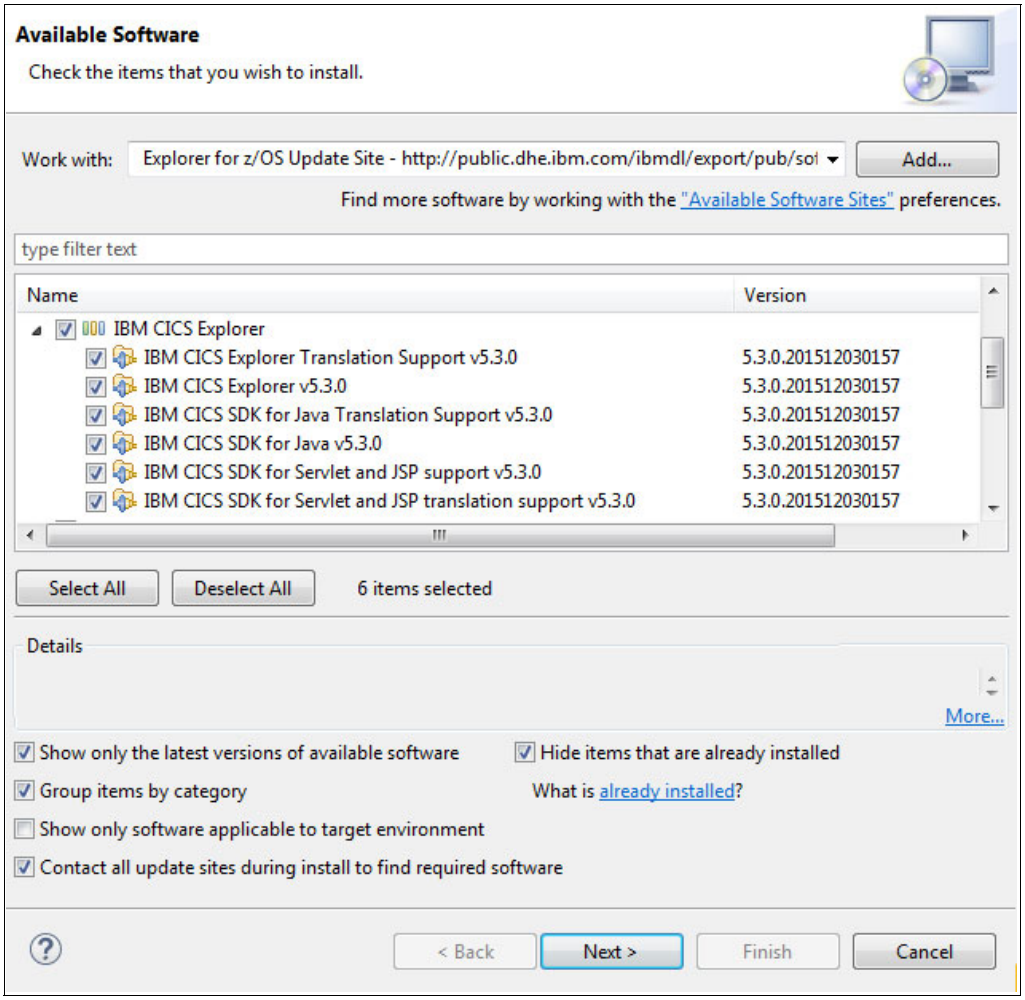


Figure 5-3 Selecting IBM CICS Explorer from the available software

d. Click **Next**, and **Next** again. Accept the license agreement and click **Finish**.

In CICS Explorer V5.3, the WebSphere Development Tools are now installed as part of the CICS SDK for Servlet and JSP translation support. However, you still need to create a local Liberty server to perform local testing of the application.

2. Install the local WebSphere Liberty Profile Server.

   Perform the following steps to create the local WebSphere Liberty Profile Server:

   a. Select **File** → **New** → **Other** and then select **Server** and click **Next**. See Figure 5-4.

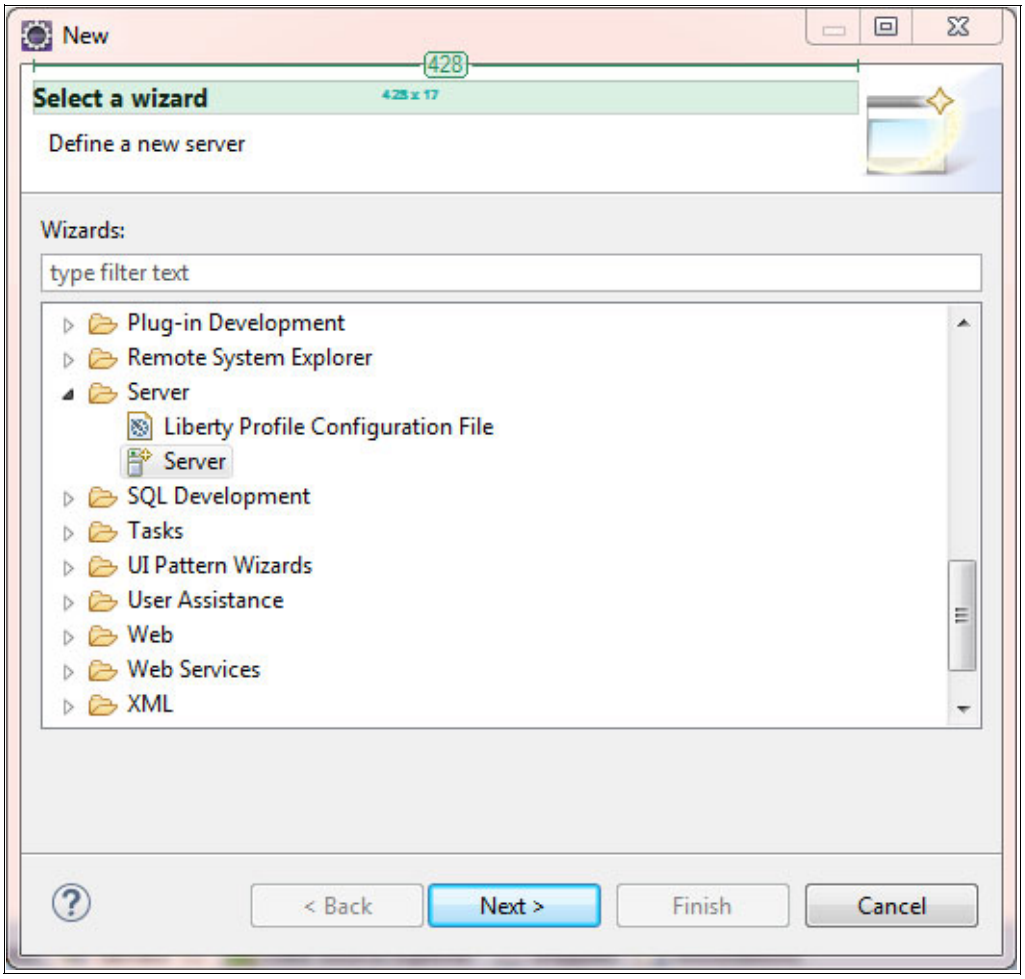


Figure 5-4 Installing a new local server

Choose **WebSphere Application Server Liberty Profile** and click **Next**. See Figure 5-5.

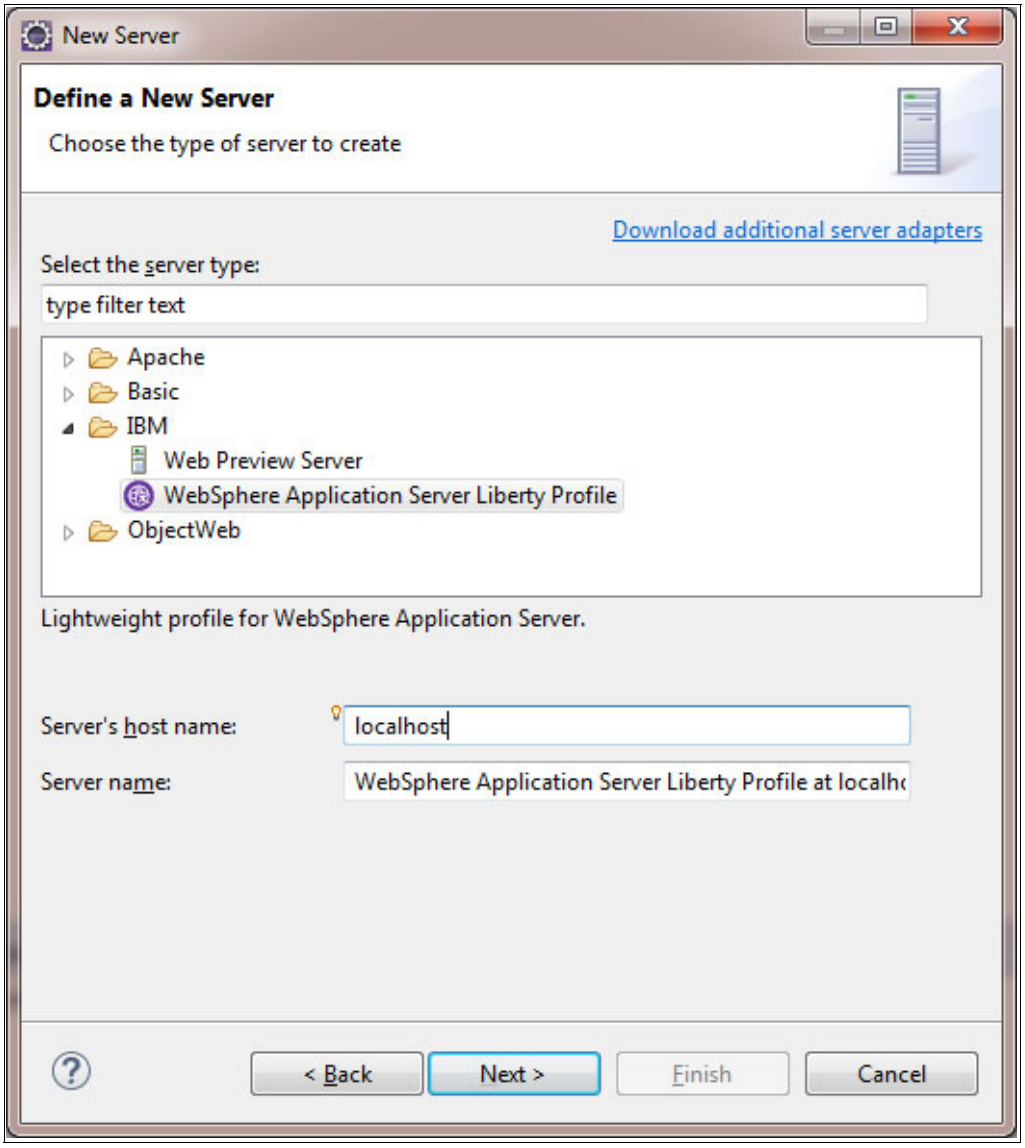


Figure 5-5 Define a new server type

b. Select **Install from an archive or a repository** and click **Next**. See Figure 5-6.

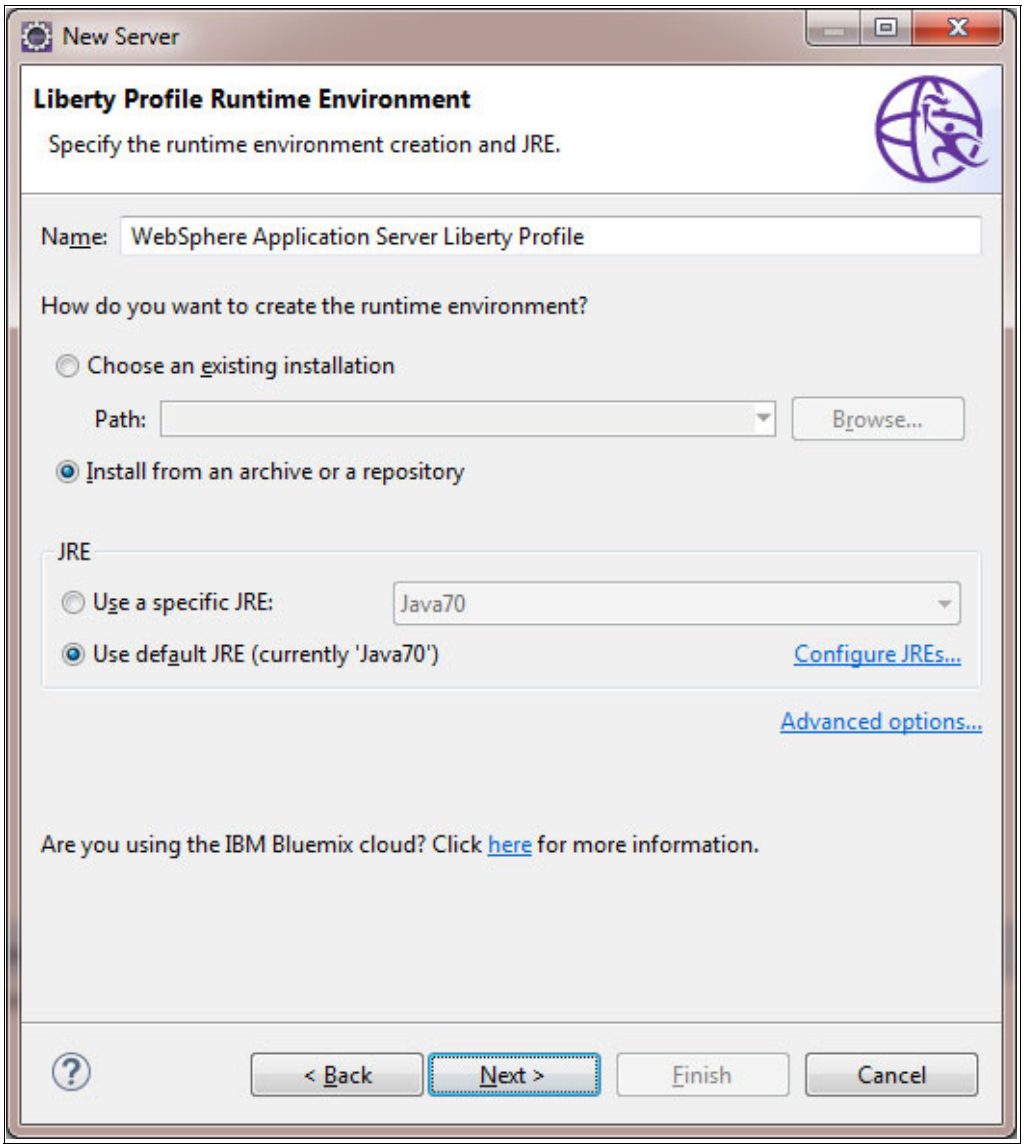


Figure 5-6 Specifying Liberty Profile Runtime Environment

c. Enter a destination path. The destination path is the path where your Liberty server will be installed. Select **WAS Liberty V8.5.5.7 Runtime**, which provides the basic Java EE 6 Web profile environment for testing our servlet, and click **Next** twice. See Figure 5-7.

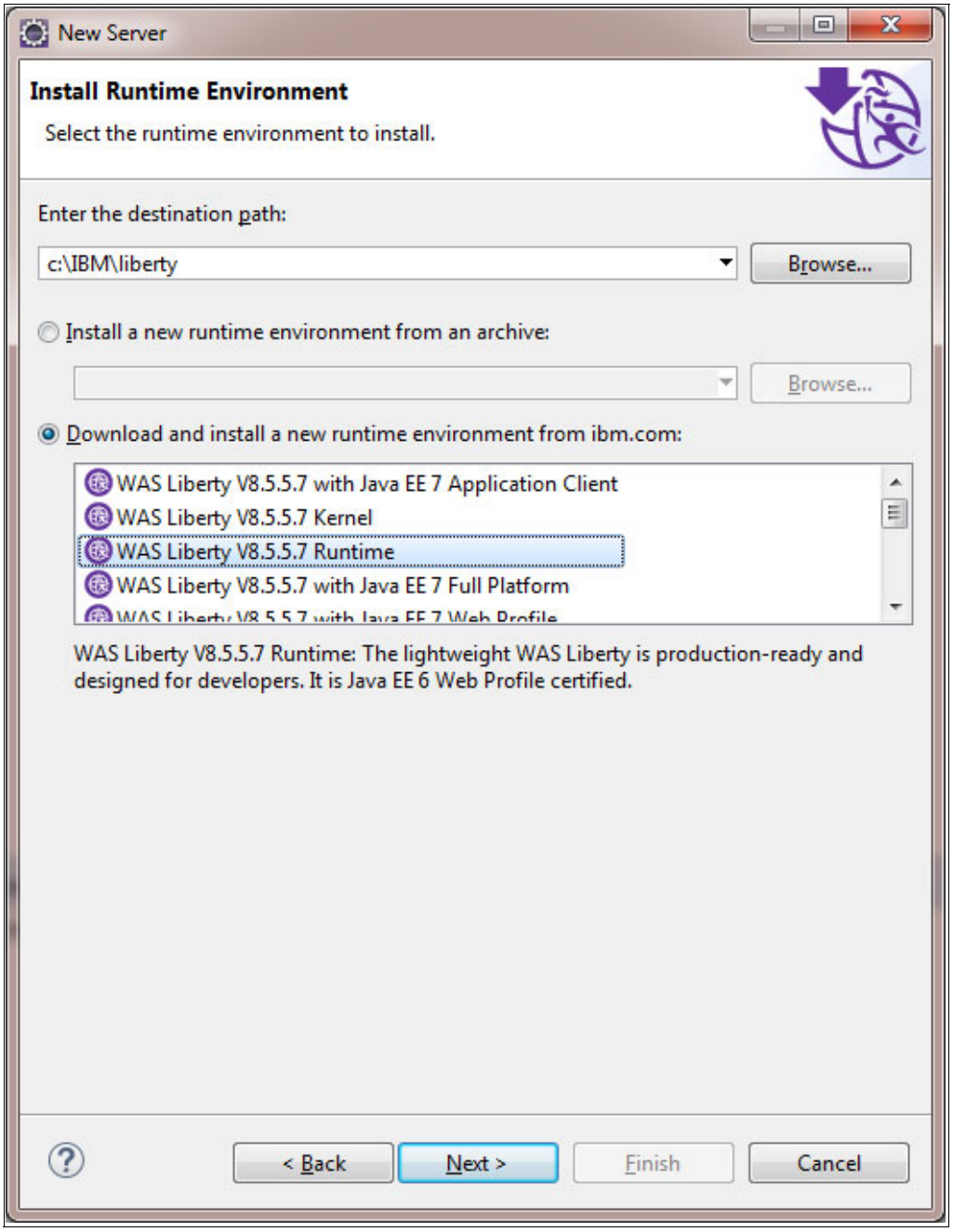


Figure 5-7 Install the runtime environment

d. If you want to change the server name or add an existing application to the server, click **Next**. Otherwise, accept the license agreement and click **Finish** and the Liberty runtime is downloaded.

5.2 Creating a HelloWorld application

After setting up the appropriate development environment, you can now create and deploy a simple Liberty application from scratch. The application includes a simple servlet, which responds to HTTP GET requests, either from a browser or another web client.

Use the following stepwise process for developing your HelloWorld servlet:

1. Create a simple dynamic web project, which produces a web archive (WAR) for deployment to the dropins directory.
2. Convert the web project to an OSGi bundle project.
3. Add the OSGi bundle to an OSGi application project.
4. Exported as an enterprise bundle archive (EBA) for deployment to Liberty within a CICS bundle project.

This method allows you to start out with a simple deployment model using a WAR. Then, migrate to using an OSGi environment for more advanced modular development, and integrate this with the CICS bundle deployment lifecycle.

Follow these steps:

1. Configure the target platform

   Before creating any OSGi applications, you need to first define a target platform that describes the runtime server environment and the interfaces that are available.

   a. Open the Eclipse preference page (**Window** → **Preferences**) and enter `target platform` in the search box and then select it from the list. Next, click **Add** to create a new target platform to represent the CICS runtime. See Figure 5-8.

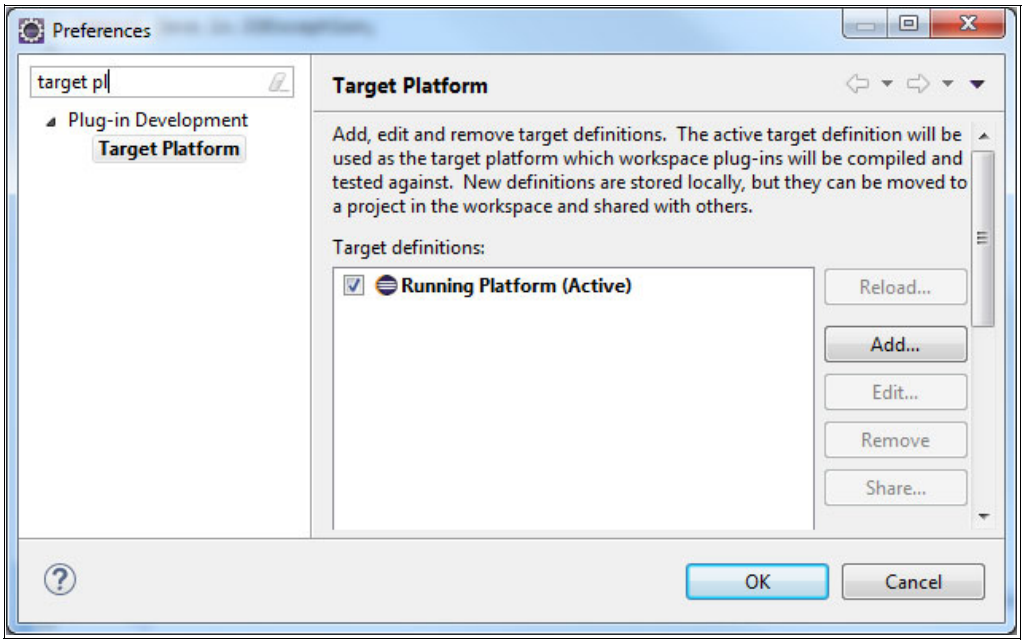


Figure 5-8 Define target platform

b. The CICS Explorer SDK provides useful templates to set up the development environment. Select **Template**, select **CICS TS V5.3 with Liberty and PHP** from the drop-down list, and click **Next**. See Figure 5-9.

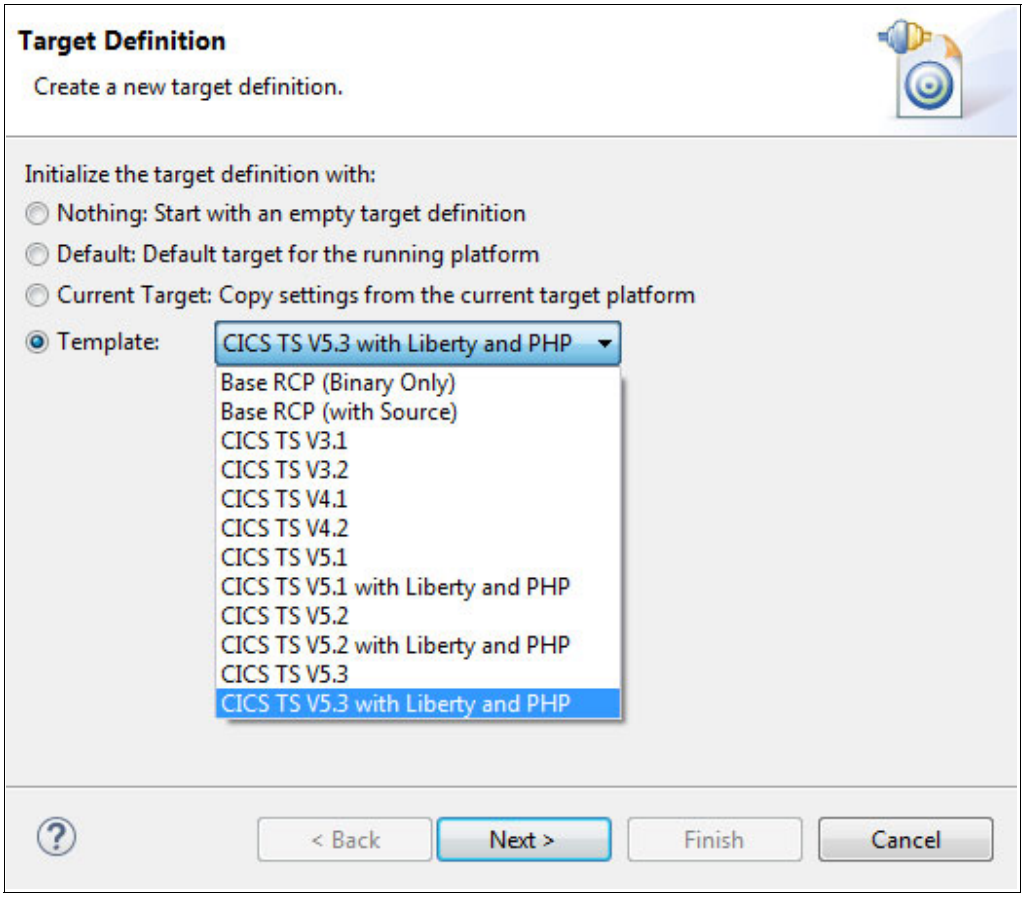


Figure 5-9 Create Target Definition

c. Figure 5-10 shows the contents of the platform and gives you the possibility to extend the platform with your own bundles if required. Click **Finish**.

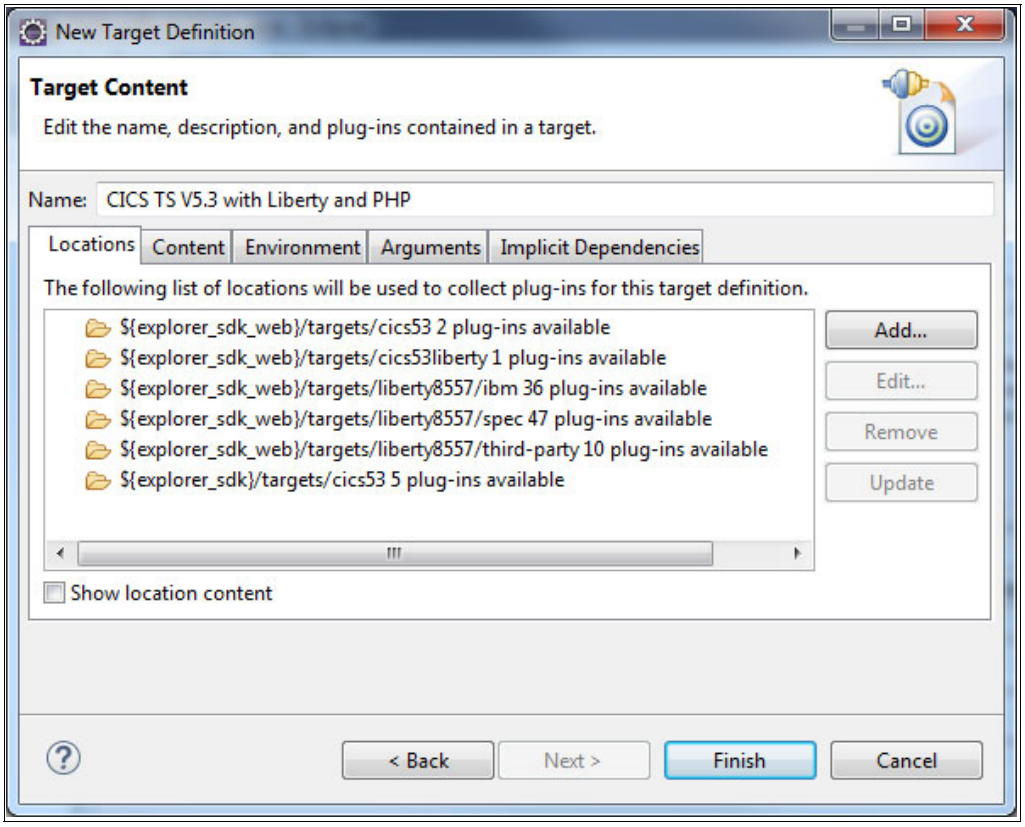


Figure 5-10 Target Content options

d. To finish setting up the development environment, activate the configured platform. Select the **CICS TS V5.3 with Liberty and PHP** check box, and click **OK**. See Figure 5-11.

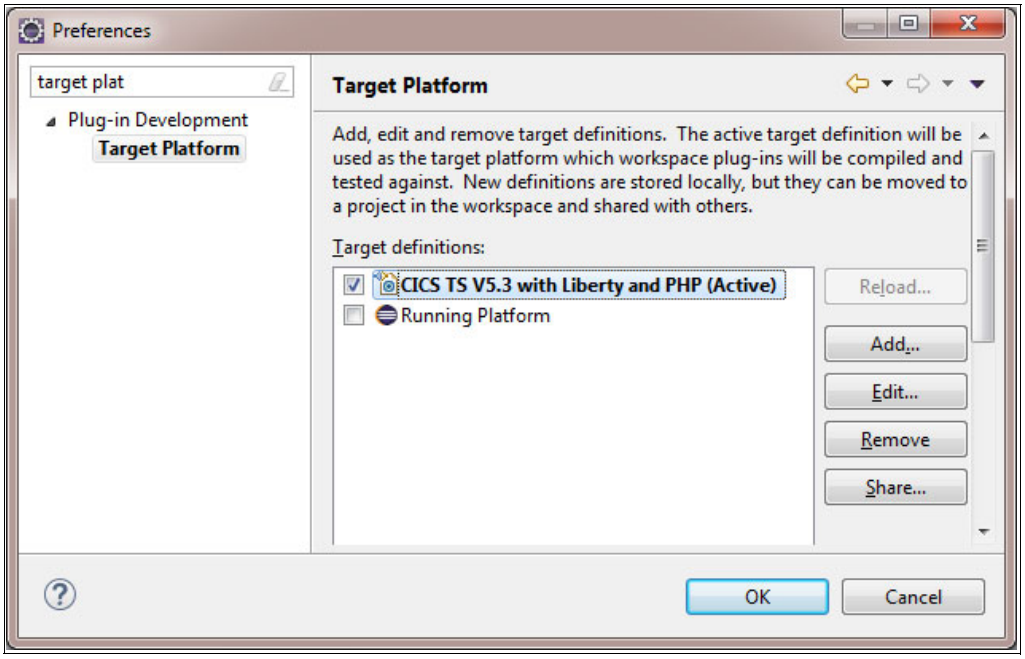


Figure 5-11 Select target platform

2. Create the HelloWorld application.

   Perform the following steps to create the HelloWorld servlet:

   a. If not already done, switch to the Java Platform, Enterprise Edition perspective by selecting **Window** → **Open Perspective** → **Other** → **Java EE (default)**. Click **OK**. See Figure 5-12.

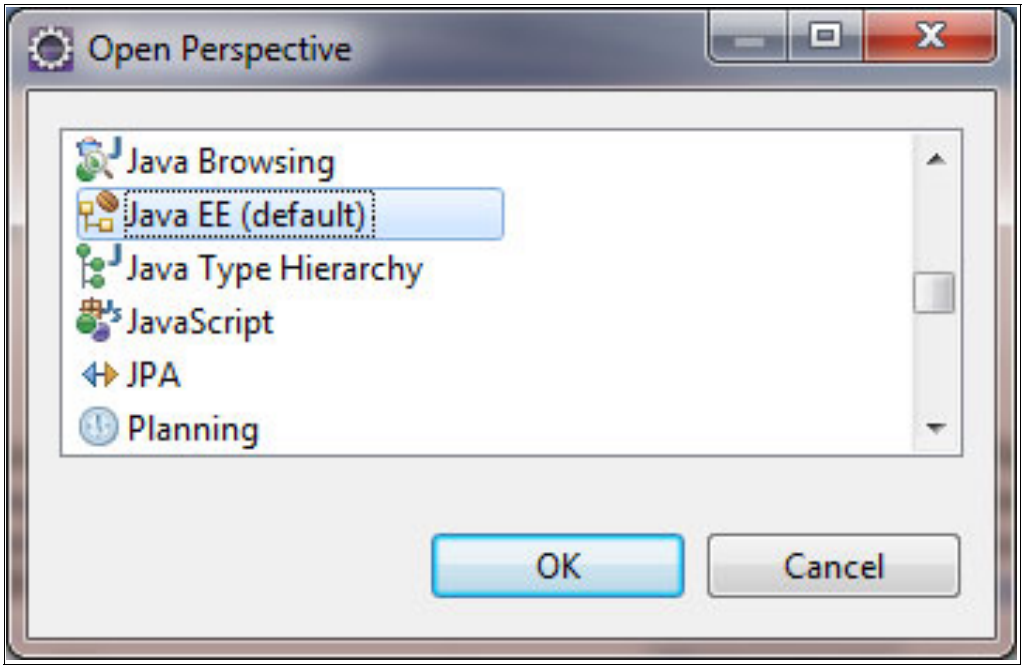


Figure 5-12 Open Java EE perspective

b. Now you can create the HelloWorld project. Click **File** → **New** → **Web** → **Dynamic Web Project**.

Give your project a name (in this sample, enter com.ibm.liberty.HelloWorld) and clear the **Add project to an EAR** option because we are not planning on using EJBs. See Figure 5-13.

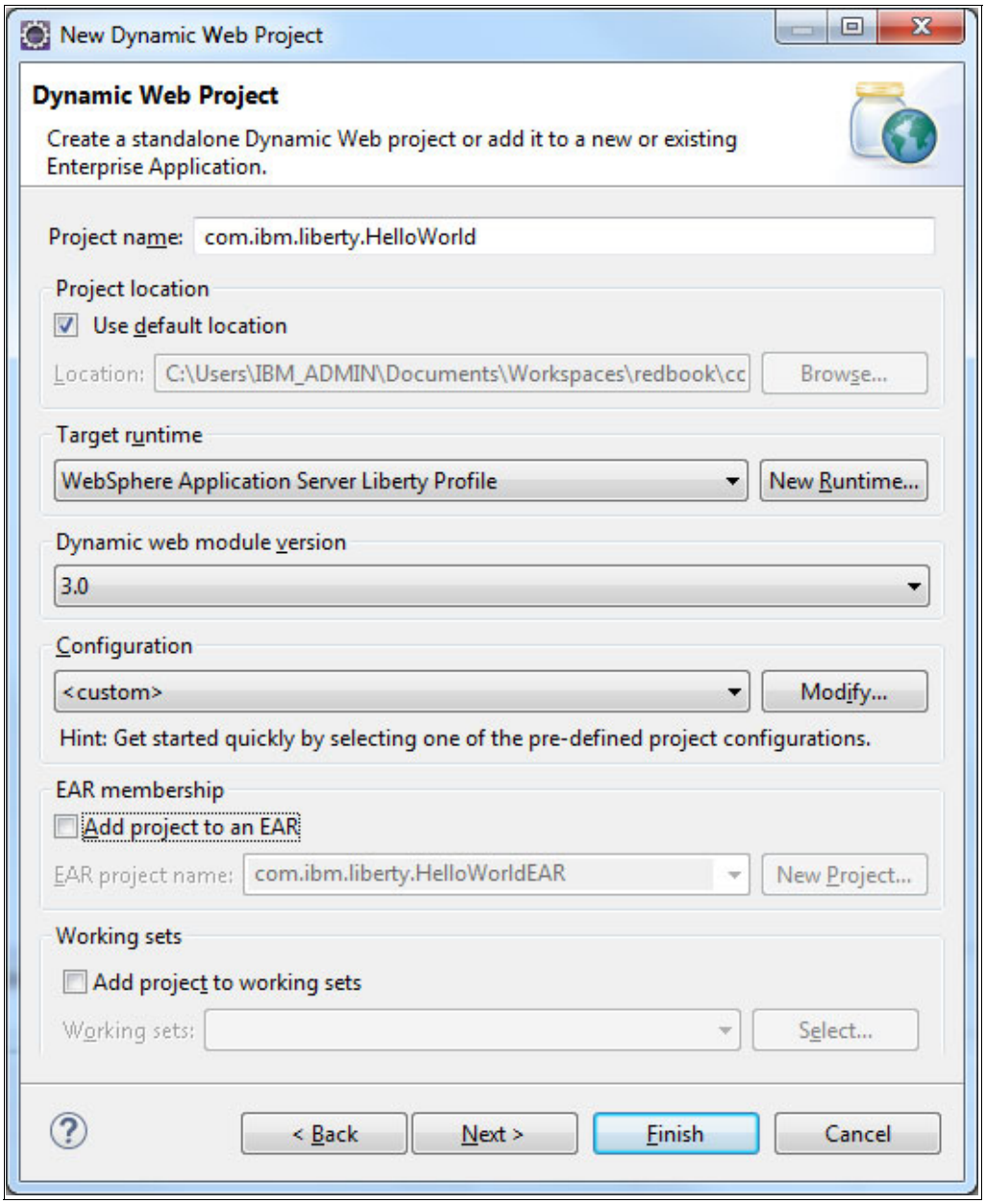


Figure 5-13 Create a Dynamic Web Project

c. Click **Next**. The next page describes the Java source directory. You do not need to change anything on this page so click **Next** again.

d. The next page configures the web module settings. Select the check box to **Generate `web.xml deployment descriptor`**, and click **Finish**. See Figure 5-14.

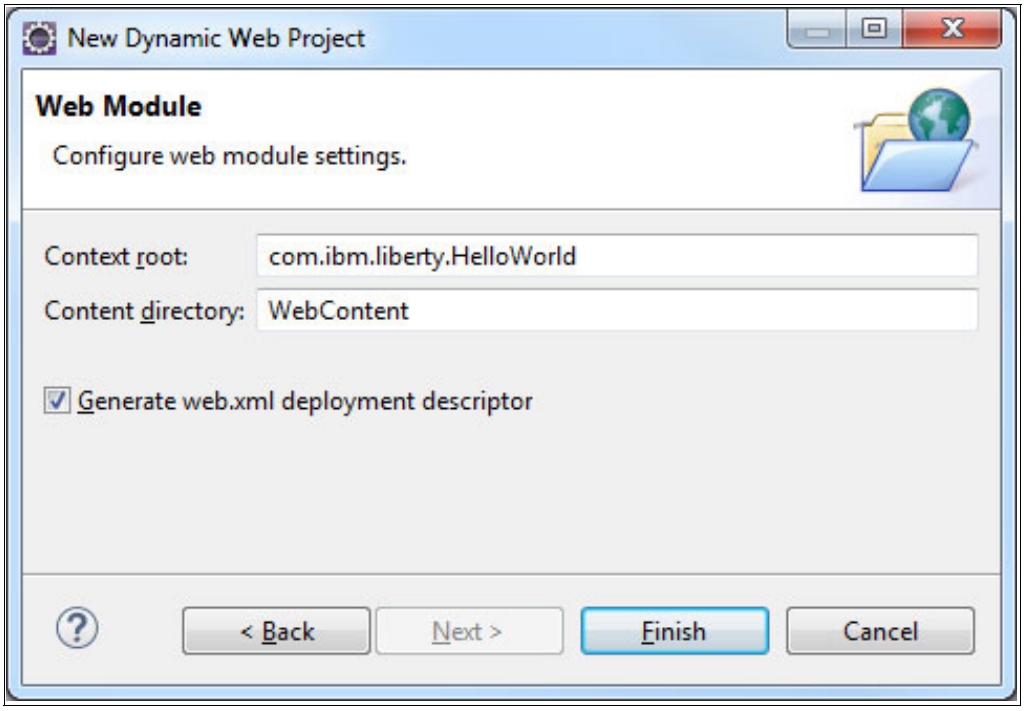


Figure 5-14 Configure web module settings

Now your web project is ready for development work to begin.

Your newly created web-enabled bundle project is now listed in the Enterprise Explorer view of your Eclipse. See Figure 5-15.

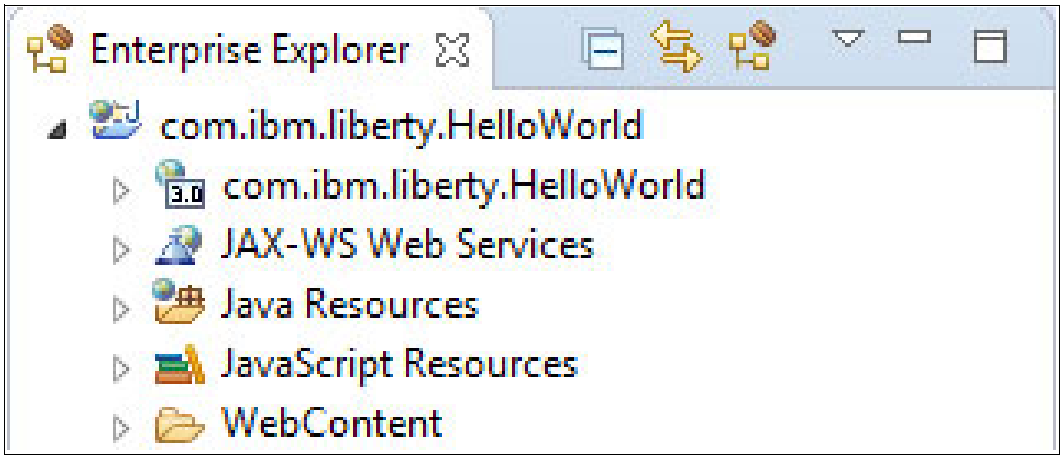


Figure 5-15 Enterprise Explorer view of the project

3. Set the build path.

   Dynamic web projects are not able to use the Target Platform because they are not OSGi components. Therefore, you need to add both the Liberty and CICS development libraries to the project's build path. Complete the following steps:

   a. In the Enterprise Explorer view, right-click the HelloWorld Project and select **Build Path** → **Configure Build Path**. The Java Build Path editor is then displayed. See Figure 5-16.

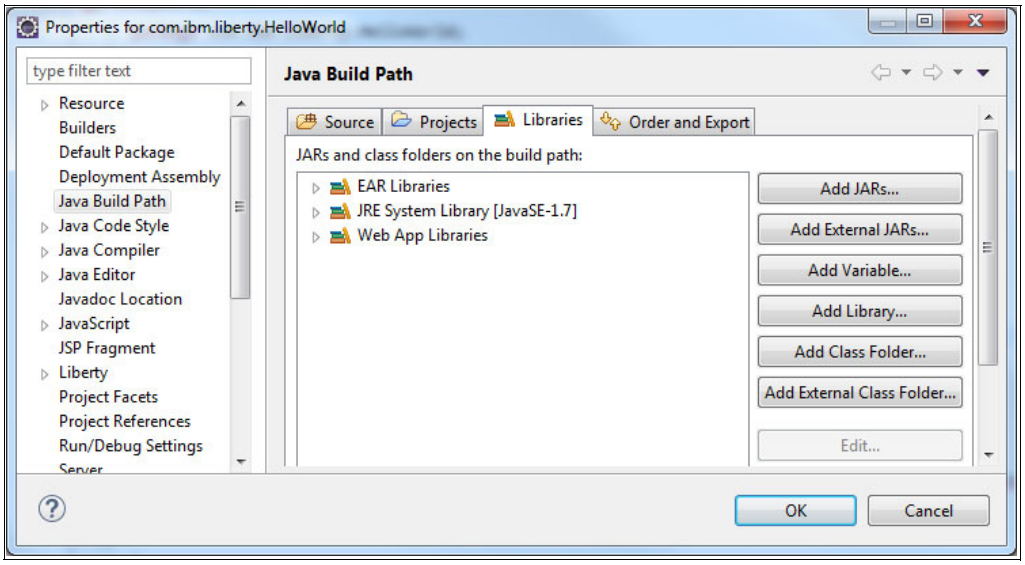


Figure 5-16 Configure Java Build Path

   b. Click the **Libraries** tab and then click **Add Library**.

c. Select **Liberty JVM server libraries** and then click **Next**. See Figure 5-17.

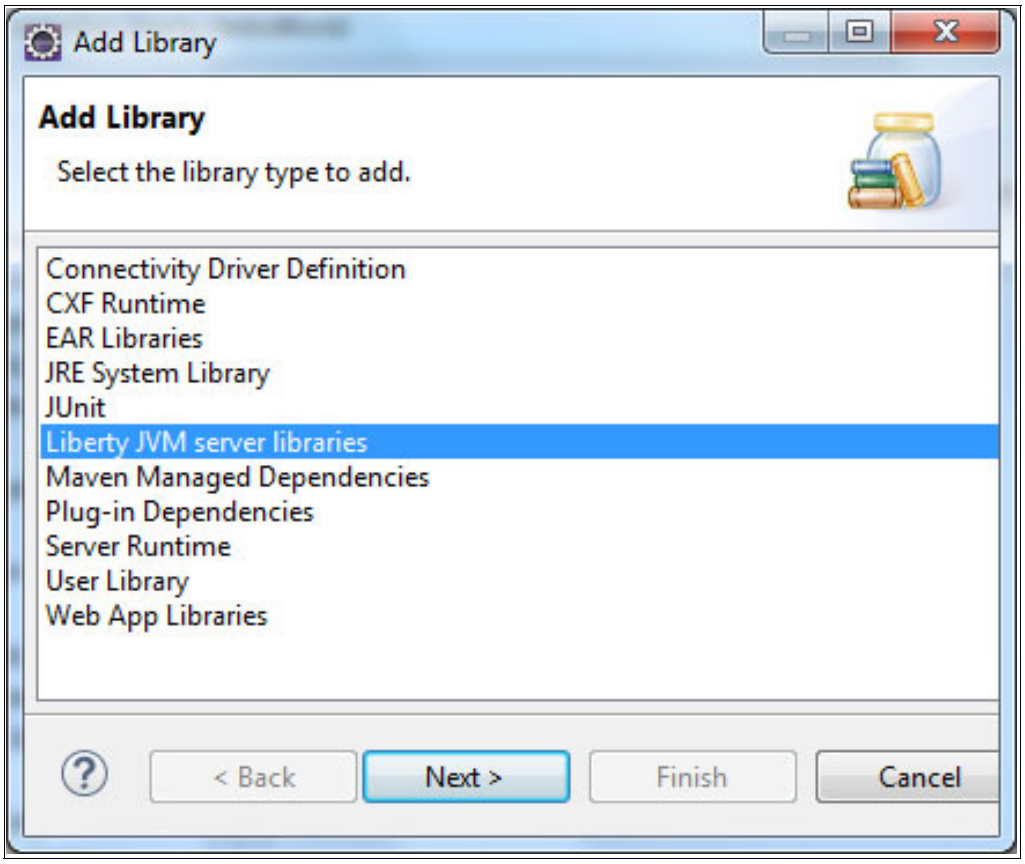


Figure 5-17 Add Liberty JVM server libraries

d. In the next window, select **CICS TS 5.3** and click **Finish**.

Both the CICS and Liberty libraries are now added to the project build path, giving access to both the JCICS and the servlet APIs.

Next, you create the sample servlet class.

e. Select **File** → **New** → **Servlet**. In the Create Servlet window, specify the name for the Java package (com.ibm.liberty.HelloWorld) and for the Class (Hello). Click **Next**. See Figure 5-18.

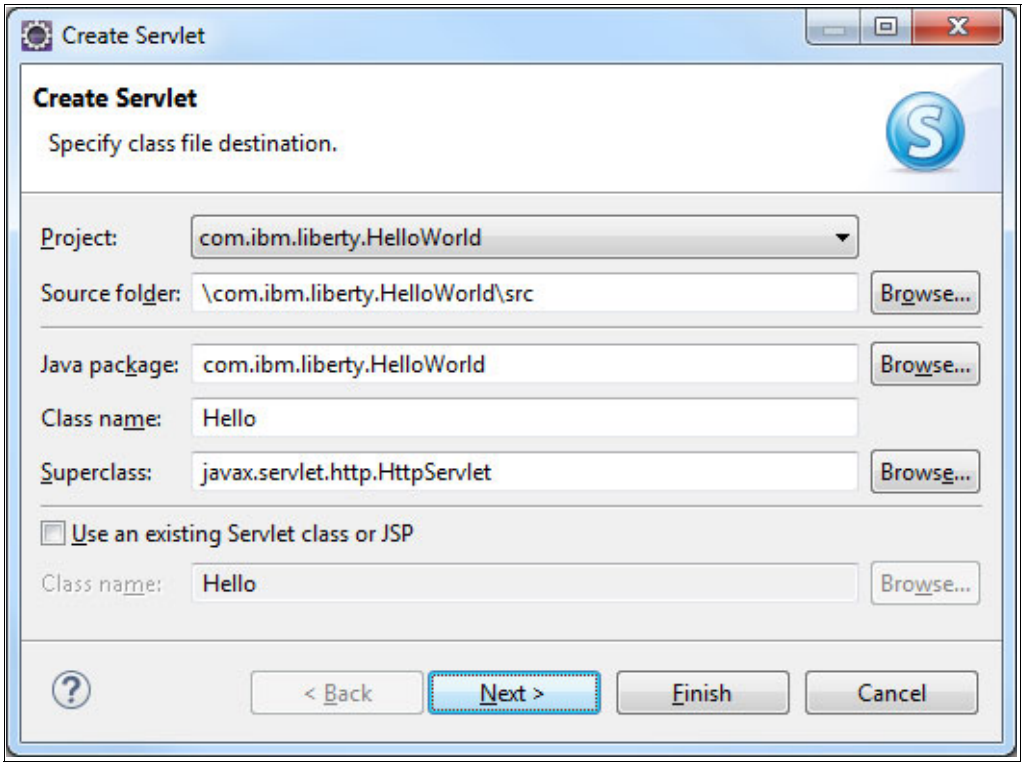


Figure 5-18 Create Servlet window

f. The next page displays information about the servlet. No changes are necessary. Click **Next**. See Figure 5-19.

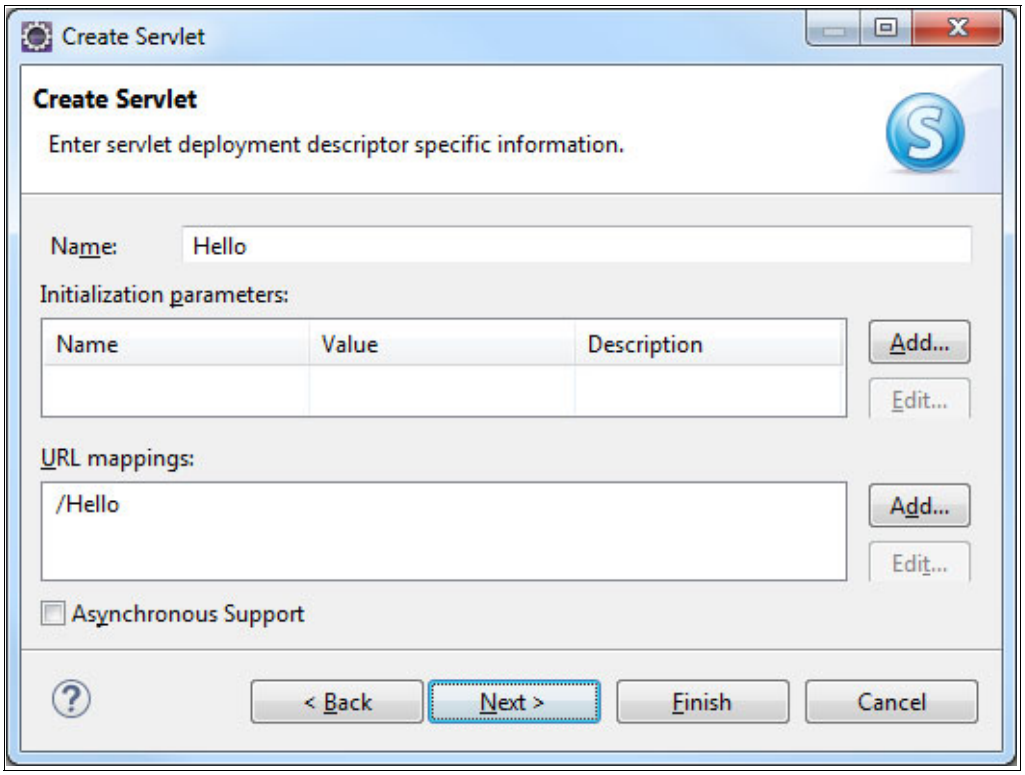


Figure 5-19 Create Servlet default values

g. The last window specifies the HTTP interfaces supported by your servlet. Clear the **doPost** field because we will only be implementing a doGet() method. Then, click **Finish**. See Figure 5-20.

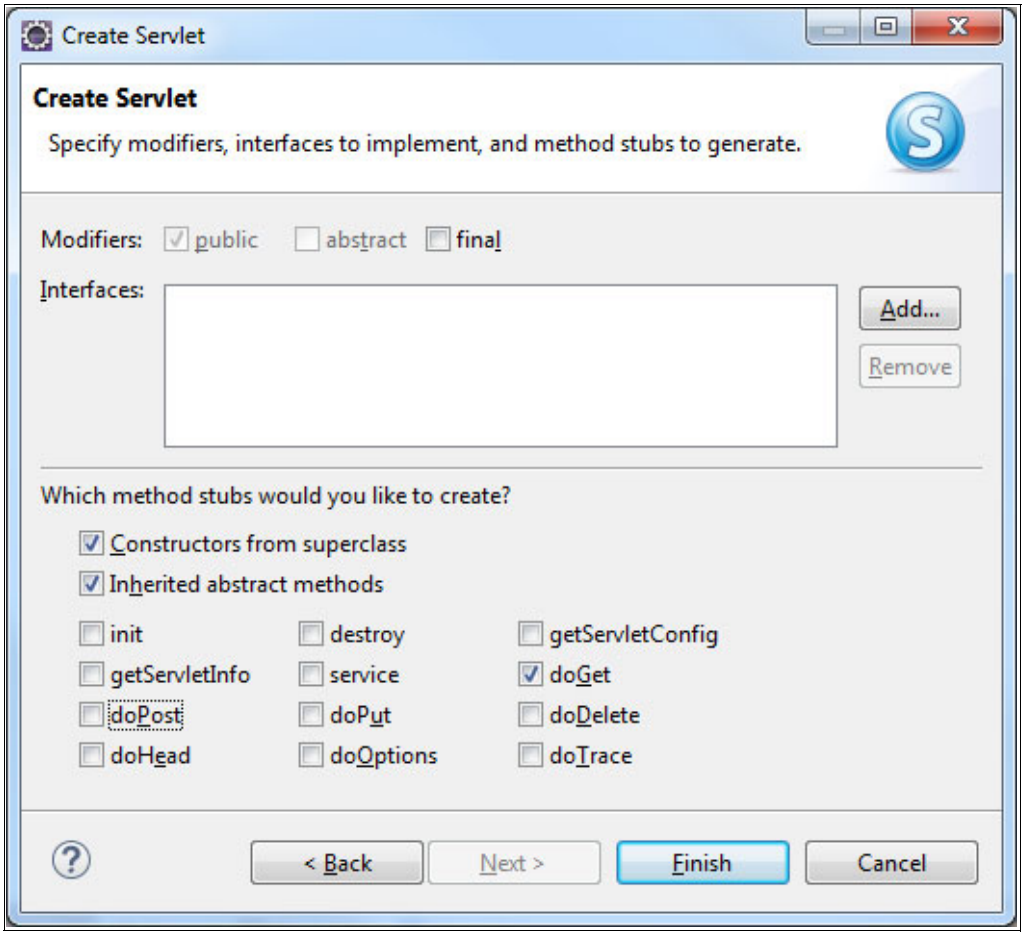


Figure 5-20 Select modifiers

h. The Hello.java class should open automatically. Go to the doGet() method and find this line:

```
// TODO Auto-generated method stub
```

i. Replace it with the following code snippet:

```
ServletOutputStream output = response.getOutputStream();
output.println("Hello World!");
```

ii. Next, click **Source** → **Organize Imports** to add the missing imports. Your Hello World servlet should look like what is shown in Example 5-1.

Example 5-1 Hello source

```
@WebServlet("/Hello")
public class Hello extends HttpServlet {
   private static final long serialVersionUID = 1L;

    public Hello() {
    super();
    }
```

```
   /**
    * @see HttpServlet#doGet(HttpServletRequest request,
HttpServletResponse response)
    */
   protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

    ServletOutputStream output = response.getOutputStream();
    output.println("Hello World!");

   }

}
```

Lastly, you need to add the Hello servlet to the welcome file list for the web application.

i. To do this, open the `web.xml` file in the `WebContent/WEB-INF` folder of the dynamic web project. Click **Welcome File List** for the web application (see Figure 5-21), and then in the Details tab, click **Add**. Select **Hello**, which is the name of the servlet class, and then close the editor to save the `web.xml` file.

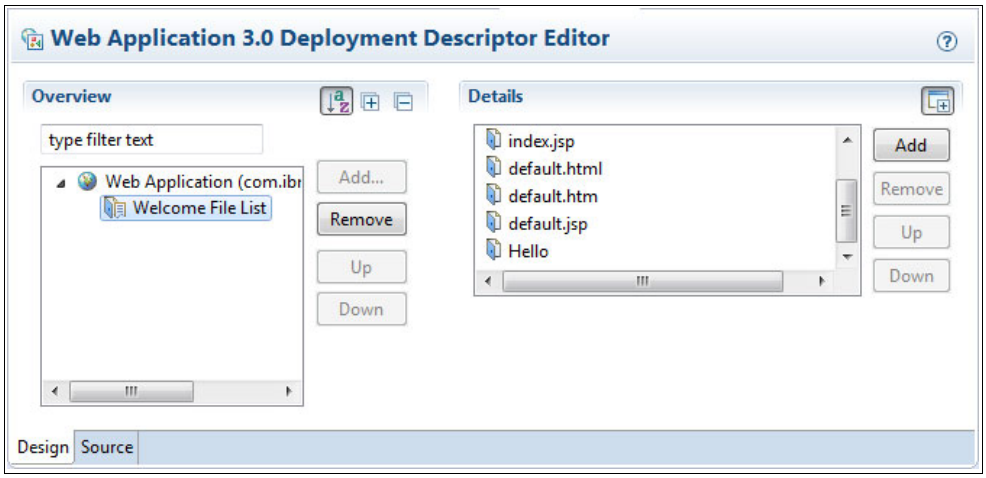


Figure 5-21 Updating the deployment descriptor

Now you are ready to test the application locally on the Liberty server.

j. Right-click your com.ibm.liberty.HelloWorld project and select **Run As** → **Run on Server** (Figure 5-22).

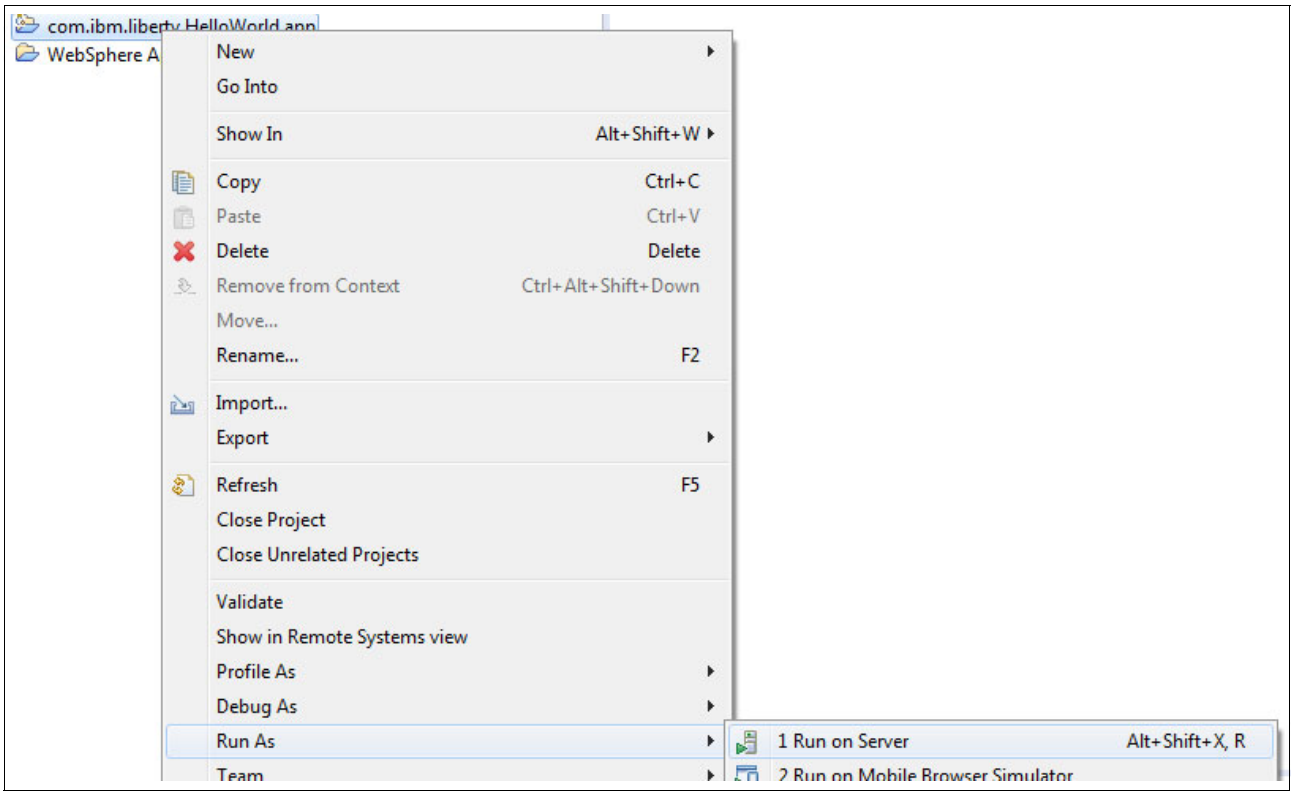


Figure 5-22 Selecting Run on Server option

k. Select your local Liberty server and click **Finish**. See Figure 5-23.

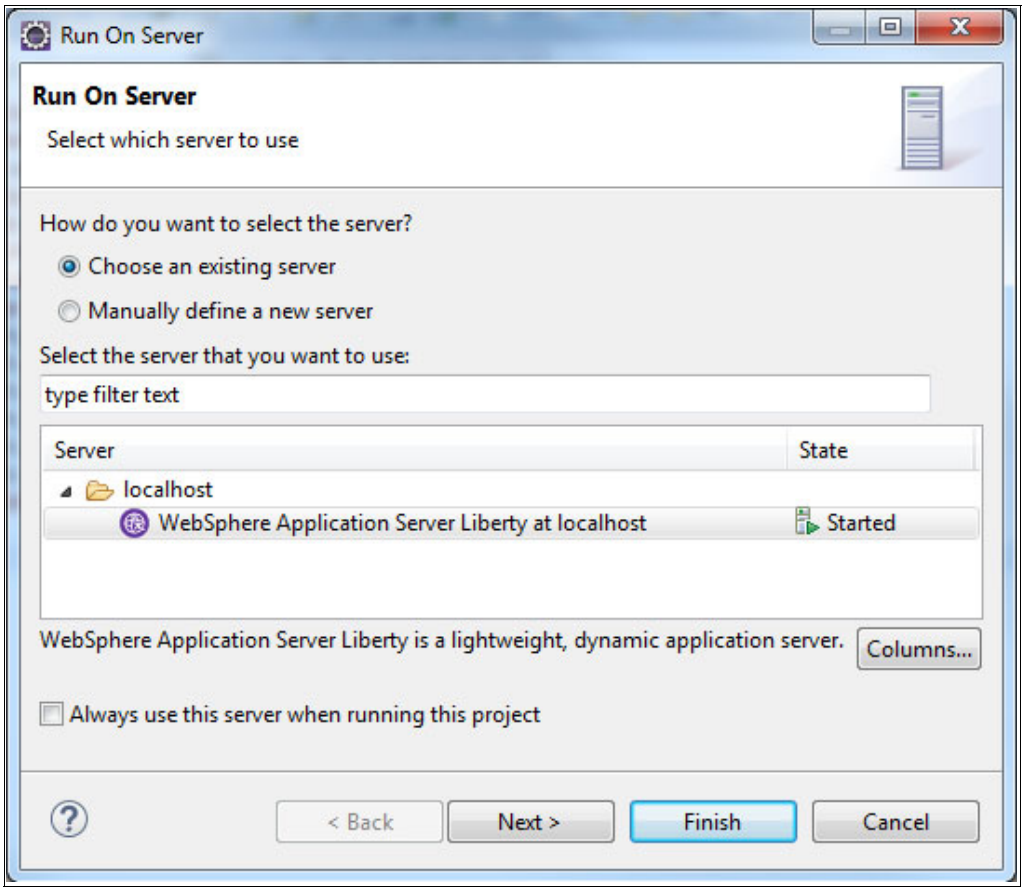


Figure 5-23 Select local server

l. Go to your web browser and open `http://localhost:9080/com.ibm.liberty.HelloWorld/Hello`. See Figure 5-24.

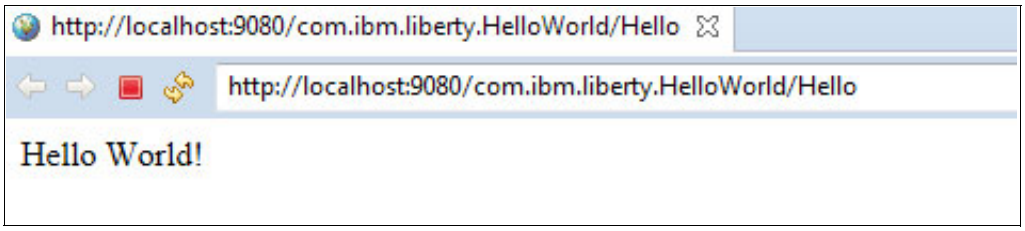


Figure 5-24 Result of running HelloWorld

You have now created and executed the HelloWorld application.

5.2.1 Local Liberty server

Development on a local Liberty server has many advantages and some disadvantages. The following list shows the advantages and disadvantages for a local installation:

- Advantages:
  - Fast deployment: After your application is added to the local server, it is automatically updated.
  - Debugging: The complete range of Java debugging functions is available.
  - Set up by the developer: No need for a system programmer during the development process.
- Disadvantages:
  - Performance: It is not valid to do performance tests on a local server.
  - CICS API: The CICS features are not available, so it is not possible to use the JCICS API. Although, you can use the JCA via the CICS Transaction Gateway to invoke CICS programs, and then port the same application into a CICS Liberty JVM server.

5.3 Deploying the application to CICS

There are various ways to export and deploy your web application to CICS. The simplest option is to use the `dropins` directory, which is automatically scanned by the Liberty server. In this section, we describe the following three options:

- Deployment of a WAR via the `dropins` directory
- Deployment of an web-enabled OSGi bundle as an EBA via a CICS bundle
- Scripting the building of an application by using the CICS Build Toolkit

5.3.1 Dropins directory

The dropins method is an easy way to deploy an application in your Liberty server and is ideal in a test environment, but is not recommended for a production environment.

The first step when using the dropins is to export your web application and copy it to the `dropins` folder of the Liberty server. Next, you must modify your `server.xml` file. The benefit of this option is that Liberty reloads your application automatically without requiring any further definitions.

The following steps illustrate the end-to-end flow for configuring and deploying applications using dropins:

1. Complete the following steps to create a WAR export from your web project:
   a. Right-click your project and select **Export** → **WAR file**. See Figure 5-25.

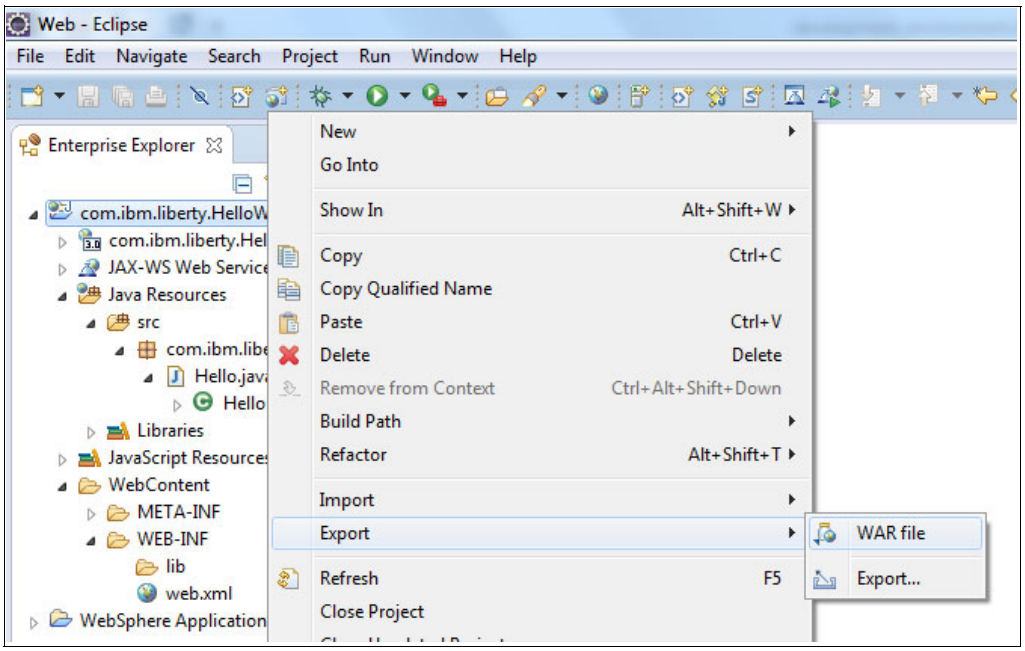


Figure 5-25 Export options on the right-click menu

   b. Select a local destination for your WAR file and click **Finish**. See Figure 5-26.

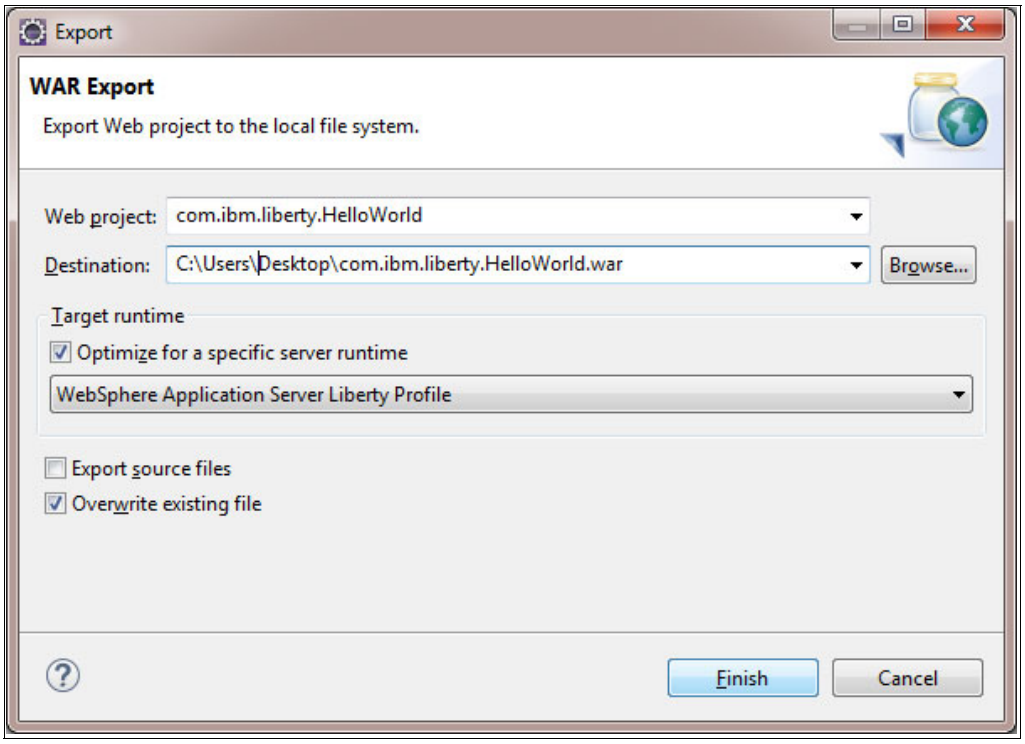


Figure 5-26 Export WAR file

2. Enable dropins in the Liberty JVM server.

   To use the `dropins` directory in a Liberty server, it must be enabled in the `server.xml` file by adding the following directive:

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

   Note: If you use the CICS autoconfigure defaults, the `dropins` directory is not automatically created. In this case, create the directory with this path: `WLP_USER_DIR/servers/server_name/dropins`, where `server_name` is the value of the com.ibm.cics.jvmserver.wlp.server.name property.

   With dropins enabled, the Liberty server automatically detects deployed WAR, EBA, or EAR files in the `dropins` folder and automatically enables them.

3. Copy to zFS.

   Now you can copy your WAR file by using FTP to your `dropins` folder. For this step, you can use any FTP client. The following sample uses the Microsoft Windows command-line FTP tool. Perform the following steps:

   a. Open a command line and connect to your host with the ftp **HOSTNAME** command:

      ```
      C:\Users\Desktop>ftp <hostname>
      ```

   b. Log in with your user ID and password:

      ```
      User (<hostname>:(none)): reds12
      331 Send password please.
      Passwd:
      230 REDS12 is logged on.  Working directory is "REDS12.".
      ```

   c. Change your working directory to your `dropins` folder:

      ```
      cd DROPINSFOLDER
      ftp> cd /…/dropins/
      250 HFS directory /…/dropins/ is the current working directory
      ```

   d. Set the transfer type to binary:

      ```
      ftp> binary
      200 Representation type is Image
      ```

   e. Send your WAR file in the `dropins` folder: `send WARfile`

      ```
      ftp> send C:\...\com.ibm.liberty.HelloWorld.war
      200 Port request OK.
      125 Storing data set /…/dropins/com.ibm.liberty.HelloWorld.war
      250 Transfer completed successfully.
      FTP: 2977 Bytes gesendet in 0,42Sekunden 7,02KB/s
      ```

   f. Close and exit the FTP connection:

      ```
      ftp> close
      ftp> bye
      ```

      Note: Be sure to use a binary transfer mode. Otherwise, the Liberty cannot read the WAR file.

5.3.2 Deploying to CICS as an EBA

This section describes the conversion of the web project to an OSGi project, and the deployment in a CICS bundle project. The supported deployment artifact for OSGi bundles in Liberty is an enterprise bundle archive (EBA), which is created in Eclipse using an OSGi Application Project.

First, you need to create a new OSGi Application Project. Proceed as follows:

1. In the Enterprise Explorer view, select your dynamic web project com.ibm.liberty.HellowWorld, right-click, and select **Configure** → **Convert to OSGi Bundle Projects**.
2. Click **OK** on the next window, which warns about the modifications to the project classpath.

   This converts the project to an OSGi bundle project, adding the OSGi bundle facet, and importantly adding the `MANIFEST.MF` file to the `WebContent/META-INF` folder, which contains the bundle dependencies. These are dynamically created for us so they do not need editing in our example.

   Next, we need to create an OSGi Application Project, which references the OSGi bundle project.
3. Right-click in the Enterprise Explorer and go to **New** → **OSGi Application Project**. If this option does not appear, you can find it through **New** → **Other** by using the filter bar. See Figure 5-27.

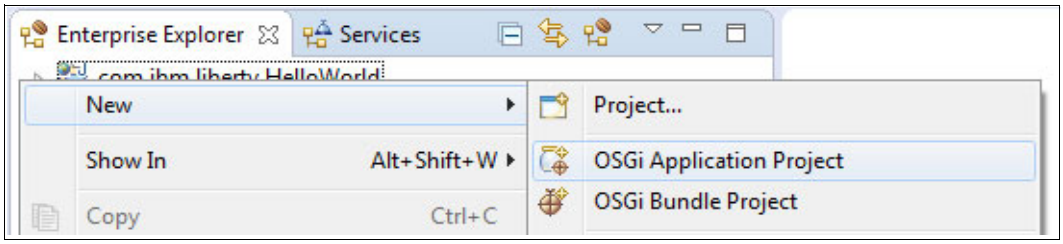


Figure 5-27 Creating the OSGI Application Project

4. Complete the project name. In this example, use com.ibm.liberty.HelloWorld.app. Click **Next**. See Figure 5-28.

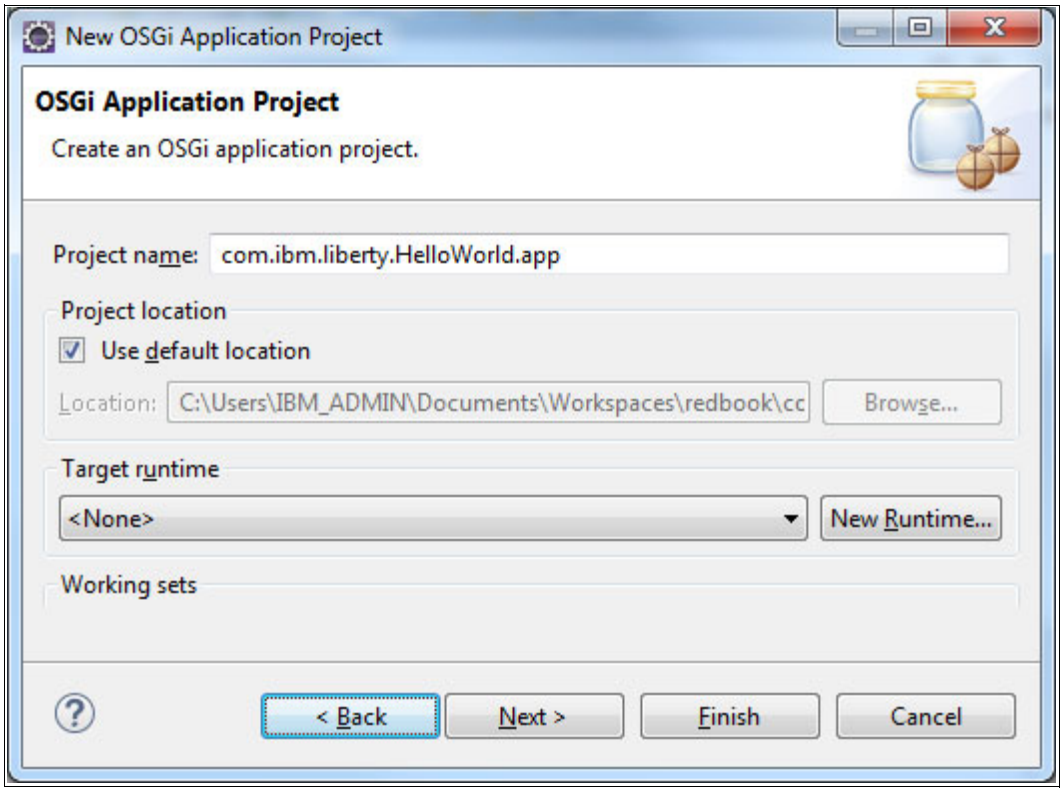


Figure 5-28 Define OSGI Application Project name

5. Now you should see the converted OSGi bundle that you created earlier. It is listed under the Contained Bundles and Composite Bundles (Application-Content) section. Select the check box next to the bundle name (`com.ibm.liberty.HelloWorld 1.0.0`) to add it to your OSGi Application Project content. Click **Finish**. See Figure 5-29.

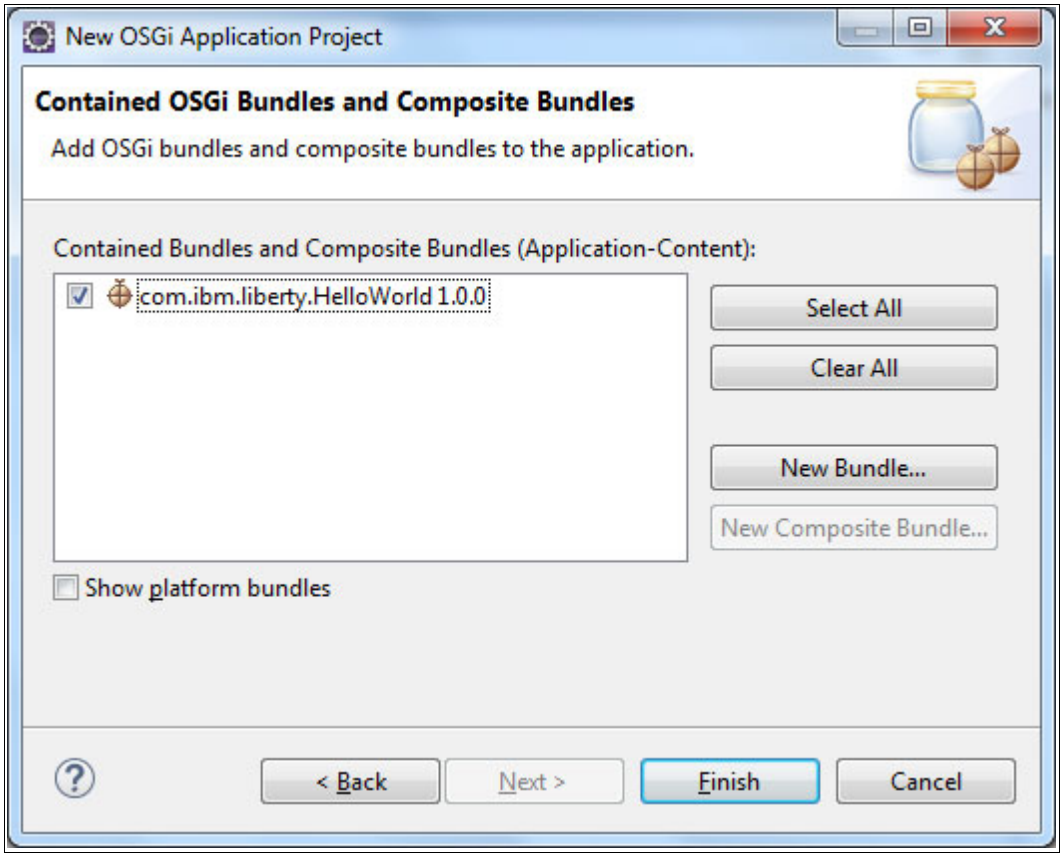


Figure 5-29 Add OSGi bundle and composite bundles to the application

Next, we need to create a CICS bundle project to reference the OSGi Application Project.

6. Click **File** → **New** → **Other**. Select **CICS Resources** → **CICS Bundle Project** and click **Next**. See Figure 5-30.

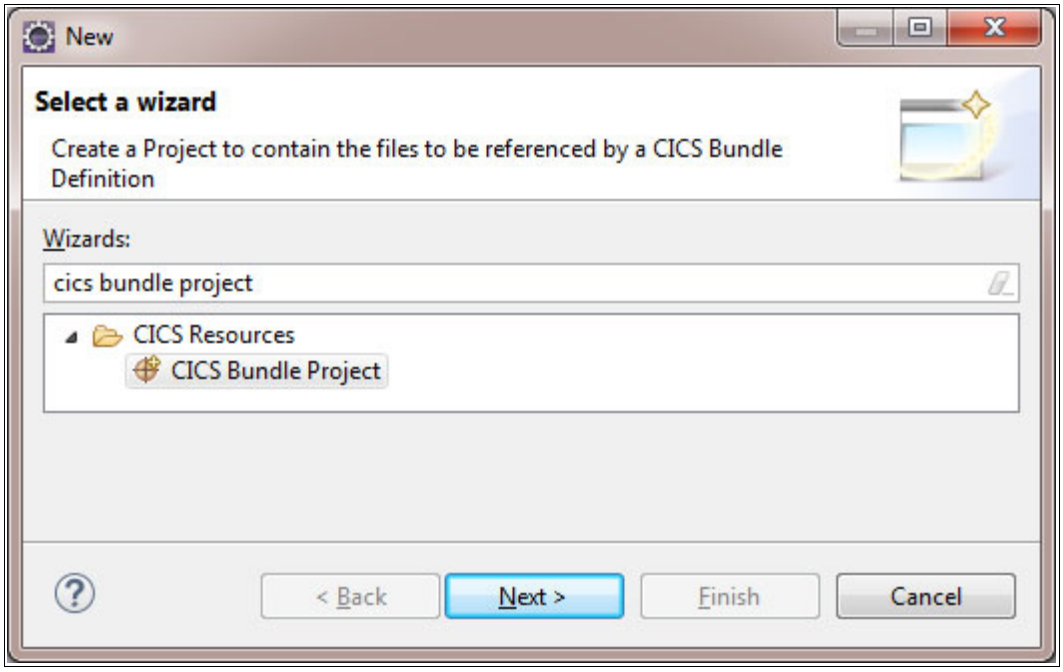


Figure 5-30 Selecting the CICS bundle project wizard

7. Specify the name of your project, in this case `com.ibm.liberty.HelloWorld.cics`, and modify the properties if wanted. Click **Finish**. See Figure 5-31.

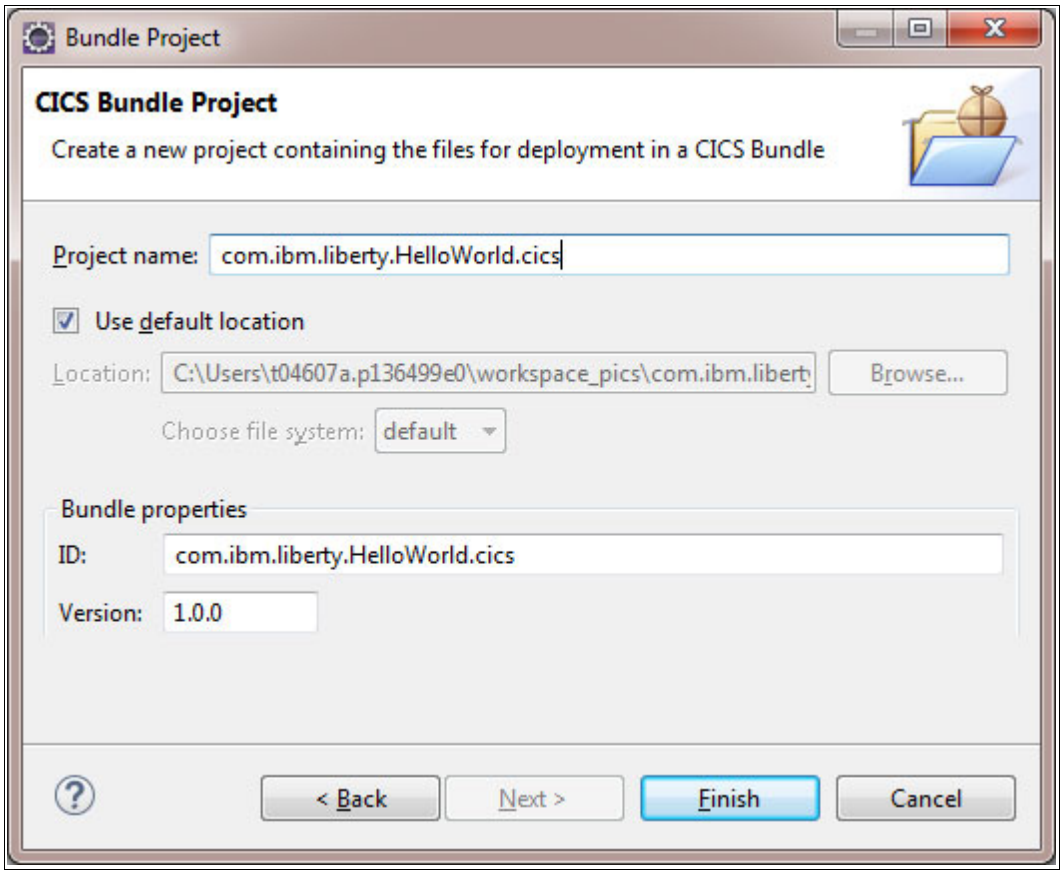


Figure 5-31 Defining the CICS bundle project name

8. Expand the newly created CICS bundle project, including the `META-INF` folder, then double-click the `cics.xml` file to open the CICS bundle manifest editor. This view shows some basic information about your bundle and gives you options for adding resources and applications to your CICS bundle. The section that we are interested in is Defined Resources. Click **New** to get a list of resources that you can add to the bundle. Then, select **OSGi Application Project Include** from the list. See Figure 5-32.

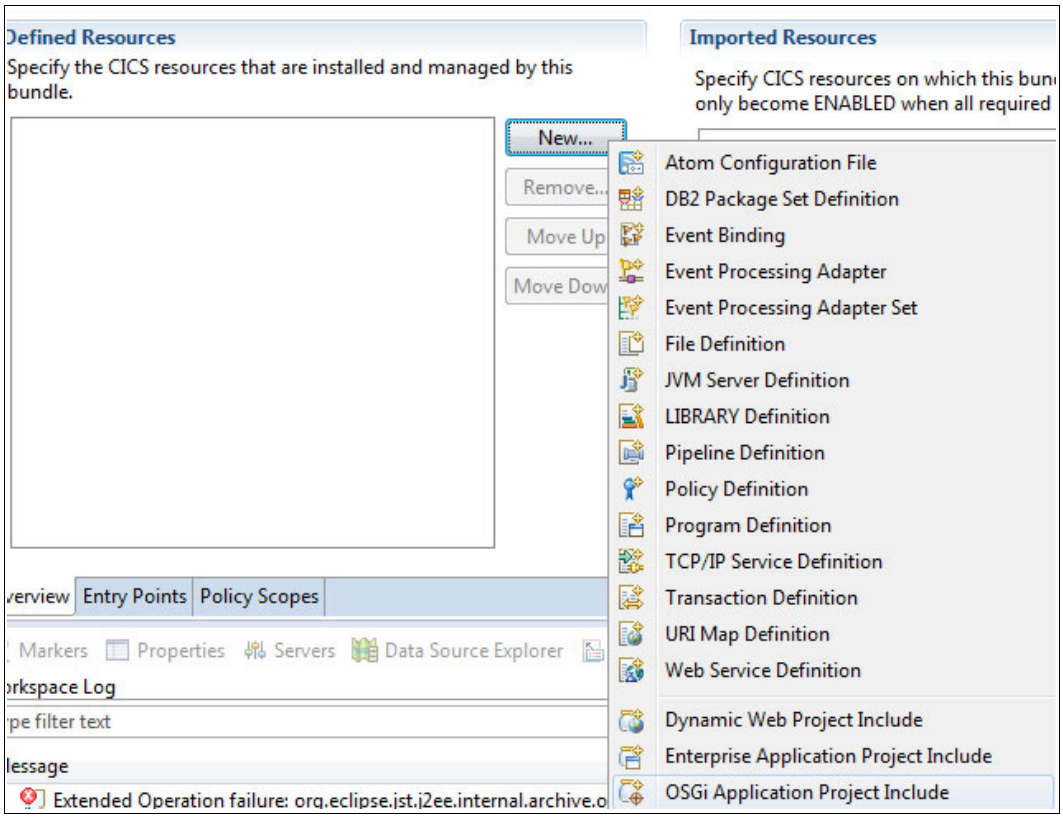


Figure 5-32 Adding the OSGi Application Project Include

9. Select your OSGi application project. Specify the name of your Liberty JVM server. Click **Finish**. See Figure 5-33.

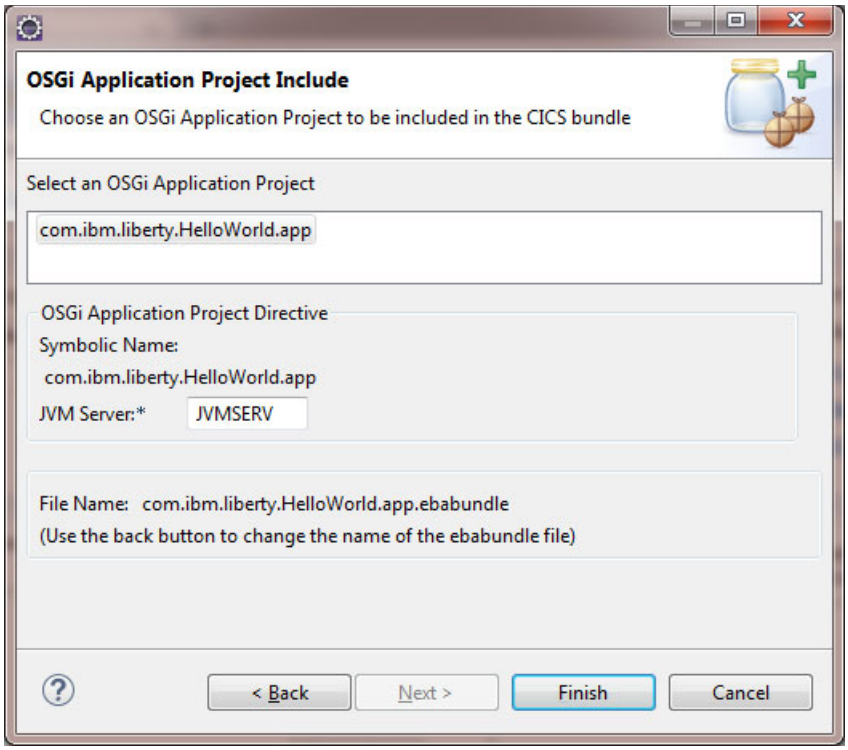


Figure 5-33 Selecting OSGi Application Project

5.3.3 Deploy CICS bundle with CICS Explorer

This section describes the deployment workflow with the CICS Explorer. Complete the following steps:

1. Right-click your CICS bundle project and select **Export Bundle Project to z/OS UNIX File System**. See Figure 5-34.

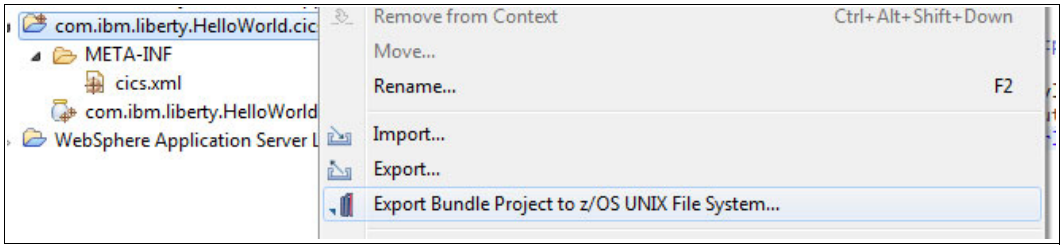


Figure 5-34 Export bundle project

2. Select **Export to a specific location in the file system**. See Figure 5-35.

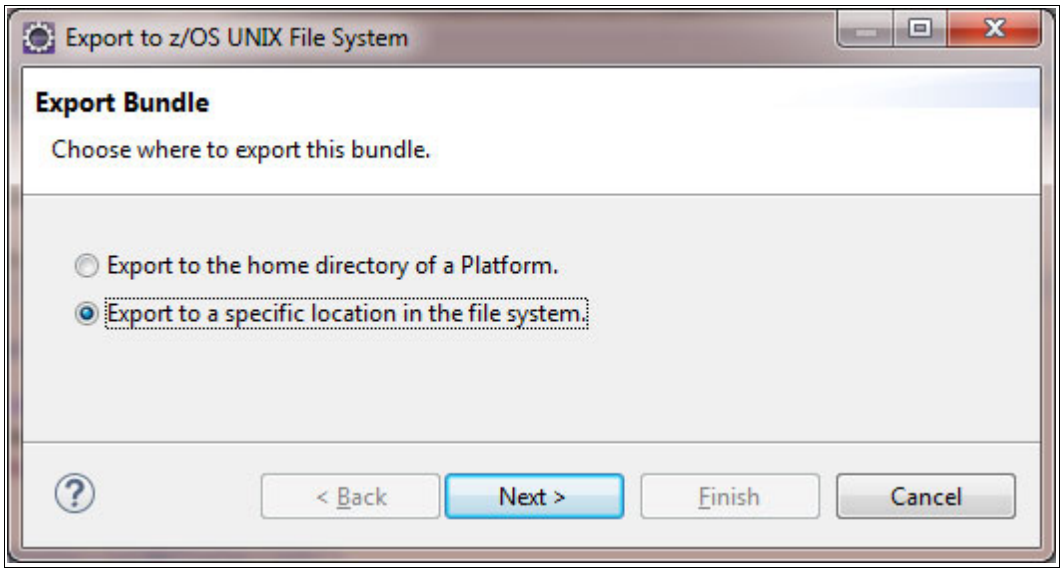


Figure 5-35 Export bundle project to a specific location

3. Specify your Bundle Directory and click **Finish** (Figure 5-36).

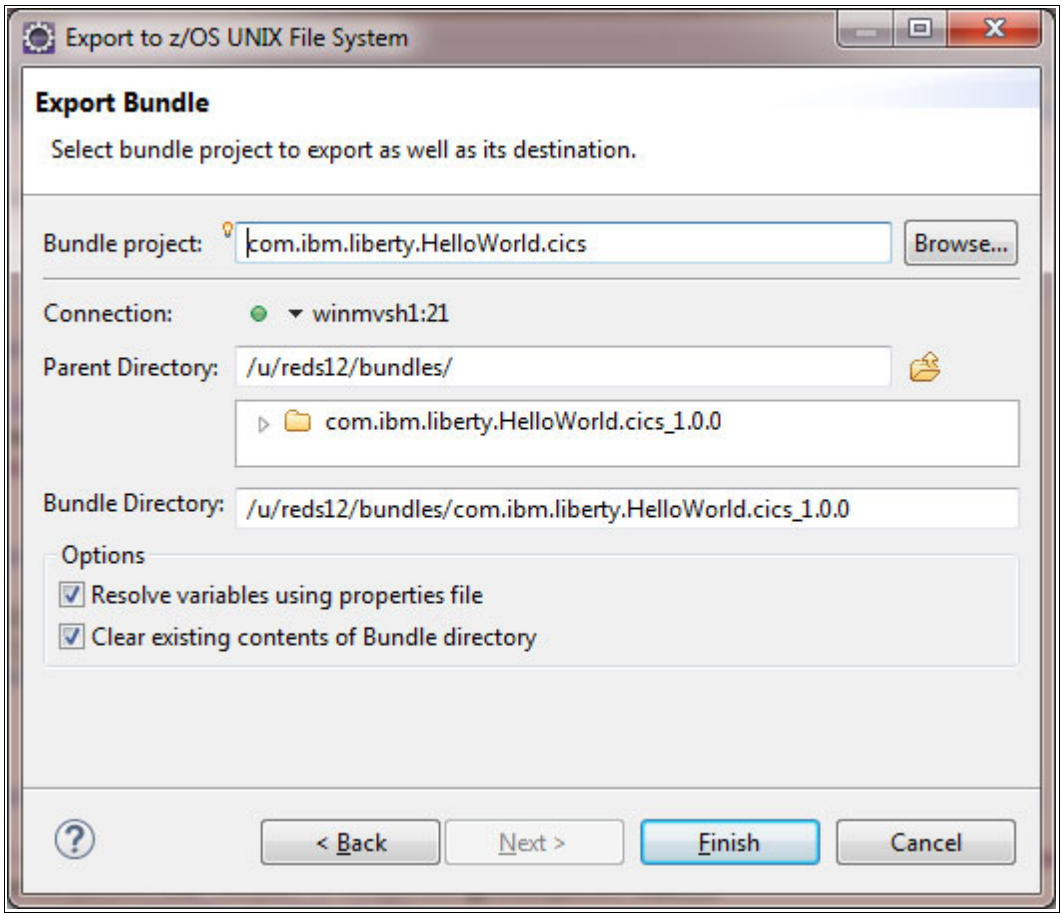


Figure 5-36 Specify the bundle to export

After you complete these steps, you need to install your CICS bundle by defining a CICS BUNDLE definition that refers to the export location on zFS. For more information, see Chapter 6, “Configuring a Liberty server in CICS” on page 103.

5.3.4 Using CICS Build Toolkit

The CICS Build Toolkit (CICS BT) provides a command-line interface for build automation of CICS bundle projects, such as web projects, OSGi bundles, CICS applications, and CICS application bindings and can take prebuilt OSGi bundles and Liberty applications as input.

In this section, we show a small sample of how to build a CICS bundle with a command line in Microsoft Windows. You can also run the Build Toolkit under Linux and z/OS because the only requirement is to have a Java Runtime Environment.

After you download and extract the CICS Build Toolkit, open a Microsoft Windows command line, change to the directory where you installed the CICS Build Toolkit, and type the following command:

```
cicsbt
--input path\to\your\project\*
--build my.bundle.name(1.0.0)
--target com.ibm.cics.explorer.sdk.web.liberty53.target
--output path\to\your\output\directory
```

The parameter descriptions for the **`cicsbt`** command are as follows:

- input: Specifies the local path to the project source
- build: Specifies which project should be built from the `input` directory option
- target: Specifies the target runtime. We used the com.ibm.cics.explorer.sdk.web.liberty53.target to support CICS TS V5.3 web applications
- output: Specifies the directory for your deployable CICS bundle project

For more information about using the CICS Build Toolkit, see *CICS and DevOps: What You Need to Know*, SG24-8339, which is available at:

`http://www.redbooks.ibm.com/abstracts/sg248339.html`

6

Configuring a Liberty server in CICS

This chapter shows you how to configure the Liberty JVM server in CICS to run a web container. This can be used to deploy Java applications that use a range of web technologies including Java servlets and JavaServer Pages (JSP) technology. The web container is based on the WebSphere Liberty profile (WLP) technology.

In this chapter, we show you a simple way to set up a Liberty server in CICS Transaction Server (TS) and how to configure it in only a few steps. This chapter takes you through setting up CICS and zFS, including all the steps required to get you started with a JVM server in CICS, and to deploy a sample Hello World web application. For more information about how to create a Hello World example, see Chapter 5, “Developing and deploying applications” on page 69.

You can define all of the Liberty JVM server resources in a classic way by using these tools:

- Resource definition online (RDO) using the CEDA transaction
- Submitting offline batch jobs (using the DFHCSDUP program provided by CICS)
- Using CICS Explorer

This chapter describes the use of both CEDA and CICS Explorer.

This chapter describes the following topics:

- Getting your CICS region ready
- zFS file system configuration
- Setting up a Liberty JVM server
- Defining and installing a CICS bundle
- Verifying the configuration
- Setting up a Liberty JVM server using CICS Explorer

6.1 Getting your CICS region ready

The first step to run Java in CICS TS is to ensure that the UNIX System Services and Java environment are enabled in CICS. To do this, follow these steps:

1. Add the SDFJAUTH library to the CICS region's STEPLIB:
   - `Syntax//STEPLIB  DD DISP=SHR,DSN=HLQ.CICS_VERSION.SDFJAUTH`
   - `Example //STEPLIB  DD DISP=SHR,CICSDPP.BETA14R.CTS53BT.CICS.SDFJAUTH`

   Note: In addition, you need to ensure that the `SDFJAUTH` library is APF-authorized, in the same manner as used for the `SDFHAUTH` library.

2. Set the **`USSHOME SIT`** parameter to the location where the CICS UNIX System Services libraries are installed. In this example, we use the following:
   - `SyntaxUSSHOME=</usr/lpp/cicsts/cicsts53|directory|NONE>`
   - `Example USSHOME=/usr/lpp/cicsts/cicsts53`
3. Determine where your Java runtime is installed. For the purposes of this example, we use:

   `Example </usr/lpp/java/J7.0_64>`

6.2 zFS file system configuration

Several zFS files are used by the JVM server in CICS. There are two for configuration and logs. You need to copy a delivered JVM profile and edit it. This topic explains how to do this.

6.2.1 zFS configuration files

Only two mandatory configuration files are required to run a Liberty JVM server in CICS:

- JVM profile: This is the configuration file for the JVM server. It must be Extended Binary Coded Decimal Interchange Code (EBCDIC) encoded and contain the environment variables and JVM settings that are necessary to support the JVM server.
- server.xml: This is the configuration file for the Liberty run time. It is encoded in ASCII and is an XML-based configuration file. It is normally created on the first startup of the Liberty JVM server using autoconfigure support, and then it can be edited.

The JVM profile is located by using the JVMPROFILEDIR system initialization parameter for your CICS region. CICS provides a set of fully commented examples in the `USSHOME/JVMProfiles zFS` directory. In this example, we show you how to configure your Liberty JVM server using the sample JVM profile and editing it for a first startup. We used the JVMPROFILEDIR location `/u/reds08/cicsts53/JVMProfiles` as the JVM profile directory.

The `server.xml` file defines the Liberty configuration that you want to use for your Liberty JVM server. If you create a Liberty JVM server for the first time with the autoconfigure option turned on in your JVM profile, this file is created for you. We suggest that you start your Liberty JVM servers with this option turned on when you start it for the first time, and then turn it off for any subsequent restarts of that JVM server. This chapter uses autoconfigure to create a `server.xml` file.

6.2.2 zFS output files

Several output files are used by the JVM server, such as the `stdout`, `stderr`, `JVM server trace` and Liberty first-failure data capture (FFDC) and messages logs. These are in region-specific subdirectories based on the WORK_DIR setting in the JVM profile. We used the location `/<home directory>/<version>/<workdir>`. This means that for our example server called JVMWLP on our CICS region with applied CREDS08A, the zFS output files are in the following directory: `/u/reds08/cicsts53/workdir/CREDS08A/JVMWLP`.

Note: In a production environment, we suggest that the zFS configuration and output files be in a separate dedicated zFS file system to provide the best recoverability of the system.

6.2.3 zFS file permissions

Access to zFS files in a JVM server is controlled by using the CICS region user ID and the UNIX file permission bits. You must ensure that the CICS region user ID has a UNIX System Services segment, and either this user ID or a group it belongs to has the following permissions:

- Read and execute access to all directories in the tree so that entry is permitted to all directories
- Read access to the JVM profile, which is used for reading the configuration
- Read and write access to the working directory, for creating log files and the Liberty server configuration files

By using the examples, you can issue the following UNIX System Services commands to set these permissions where `<CICSGID>` is the group ID that the CICS region user ID belongs to:

- `chgrp <CICSGID> /u/reds08/cicsts53/JVMProfiles`
- `chmod 750 /u/reds08/cicsts53/JVMProfiles`
- `chgrp <CICSGID> /u/reds08/cicsts53/workdir`
- `chmod -R 770 /u/reds08/cicsts53/workdir`

Note: UNIX file permissions only allow for one owner and one group owner. This can make it difficult to grant access to multiple writers if the user IDs are not in the same group. As a solution to this, you can use UNIX System Service file ACLs, which provide a more flexible security model.

6.3 Setting up a Liberty JVM server

We are now ready to create and start the JVM server. This section shows what you must define, edit, and configure to set up a Liberty JVM server in CICS using the CEDA tool to create the JVMSERVER resource definition. If you prefer to use CICS Explorer to do this, refer to 6.6, “Setting up a Liberty JVM server using CICS Explorer” on page 124.

6.3.1 JVM profile

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

You can copy the sample JVM profiles provided by CICS and customize them for your own applications. The sample JVM profile supplied with CICS is in the `USSHOME/JVMProfiles` directory on zFS. Copy the sample from the installation directory to the directory that you specified in the JVMPROFILEDIR system initialization parameter. The sample JVM profile in the installation location is overwritten if you apply an APAR that includes changes to these files. To avoid losing your modifications, always copy the sample to a different location before adding or changing any options.

For reference material about all of the properties you can edit in a Liberty JVM server profile, see *Options for JVMs in a CICS environment*:

`http://www.ibm.com/support/knowledgecenter/SSGMCP_5.3.0/com.ibm.cics.ts.java.doc/topics/dfha2_jvmprofile_options.html?cp=SSGMCP_5.3.0%2F8-7-0-3-4-2`

6.3.2 Tailoring the JVM profile

To begin, follow these steps:

1. Copy the sample JVM profile for a Liberty JVM server, which is delivered with your CICS installation and is at:

   `/USSHOME/JVMProfiles/DFHWLP.jvmprofile`

2. Copy it to your `JVMPROFILEDIR` directory as defined in your SIT:

   `/u/reds08/cicsts53/JVMProfiles/JVMWLP.jvmprofile`

Note: You can rename the JVM profile if you do not want to use the delivered default. In this example, we named it JVMWLP.

6.3.3 Java home directory

The `JAVA_HOME` directory specifies the installation location for the IBM Java SDK for z/OS. 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_HOME</usr/lpp/java/J7.0_64
```

WORK_DIR

WORK_DIR is the working directory in zFS that the CICS region uses for activities that are related to the JVM server. The CICS JVM server uses this directory for both configuration and output. A period (.) is defined in the supplied JVM profiles, indicating that the home directory of the CICS region user ID is to be used as the working directory. If the directory does not exist or if `WORK_DIR` is omitted, `/tmp` is used as the zFS directory name. You can also specify your own zFS directory for the working directory. In this situation, you must also ensure that the relevant directory is created on zFS, and that access permissions are given to the correct CICS regions.

We specified the following pattern for our WORK_DIR, under which CICS dynamically creates subdirectories for each region and JVM server.

```
WORK_DIR=/u/reds08/cicsts53/workdir
```

Note: Do not define your working directories in the CICS USSHOME directory, as this is the home directory for CICS installation files and is typically mounted read-only.

Log files

Output from Java applications that are running in a Liberty JVM server can be written to either zFS or to JES. The destination for the logs is defined using the STDOUT, STDERR, and JVMTRACE options in the JVM profile.

- STDERR

  STDERR specifies the location to which the `stderr` stream is directed from the JVM server, and often contains useful output such as Java exception messages. By default files are placed in the `WORK_DIR/<applid>/<jvmserver>` directory in zFS.

  In our example, we left STDERR unset so that the output was written to the following default location in our working directory:

  `/u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/Dyyyymmdd.Thhmmss.dfhjvmerr`

- STDOUT

  STDOUT specifies the location to which the `stdout` stream is directed from the JVM server. By default, files are placed in the `WORK_DIR/<applid>/<jvmserver>` directory in zFS. CICS JVM servers do not log output to this file, but it can be used by Java applications or other components.

  In our example, we left STDOUT unset so that the output was written to the following location in our working directory:

  `/u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/Dyyyymmdd.Thhmmss.dfhjvmout`

Note: By default, there are multiple versions of the log files stored in the zFS, but the current version in use is easily identified using the symbolic link `CURRENT.<file>` for instance `/u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/CURRENT.STDOUT.`

Redirection of log files to JES

In CICS TS V5.2 and above, JVM server log files can now be directed to JES instead of zFS.

For example, to log output to the following JCL DD members

```
// LOGOUT  DD SYSOUT=*
// LOGERR  DD SYSOUT=*
```

Specify the following options for the STDOUT and STDERR in the JVM profile:

```
STDOUT=//DD:LOGOUT
STDERR=//DD:LOGERR
```

Maximum number of zFS log or trace files

The LOG_FILES_MAX value specifies the number of old log files kept on the system. A default setting of 0 ensures that all old versions of the log file are retained. This value can be modified to specify how many old log files you want to remain on the file system:

```
LOG_FILES_MAX=<n>
```

If STDOUT, STDERR, and JVMTRACE use the default scheme, or if it is customized, include the &DATE;.&TIME; pattern, then only the newest instance of each log type is kept on the system. If your customization does not include any variables that make the output unique, the files are appended to, and there is no requirement for deletion. This clean up function does not apply if the output variables have been customized to route output to JES.

Note: The USEROUTPUTCLASS, which can be specified in an OSGi JVM server in CICS, is not supported for Liberty JVM servers.

If the variable **`LOG_PATH_COMPATABILITY=TRUE`**, the output files are placed in the `WORK_DIR` directory, which is the same behavior as in CICS TS V5.2 and earlier releases.

6.3.4 Time zone

The time zone (TZ) environment variable specifies the `local time` of a system. You can set this for a JVM server by adding it to the JVM profile.

When setting the time zone for a JVM server, be aware of the following issues:

- The TZ variable in your JVM profile needs to match your local IBM MVS™ system offset from Greenwich mean time (GMT).
- If you do not set the TZ variable, the system defaults to Coordinated Universal Time (UTC).
- Customized time zones are not supported and result in failover to UTC or a mixed time zone output in the `JVMTRACE` file.
- If you see `LOCALTIME` as the time zone string, there is an inconsistency in your configuration. This can be between your local MVS time and the TZ you are setting, or between your local MVS time and your default setting in the JVM profile. The output will be in mixed time zones, although each entry will be correct.
- The short form of Portable Operating System Interface (POSIX) TZ can be used and reduces the chances of input errors.

Short format:

```
TZ=CET-1CEST
```

Long format:

```
TZ=CET-1CEST,M3.5.0,M10.5.0
```

6.3.5 Liberty-specific options

The variables described in this section are specific to the Liberty server environment:

- `WLP_INSTALL_DIR`

  This environment variable is required if you want to start a Liberty JVM server. The Liberty profile installation files are installed in the `CICS USSHOME` directory in a subdirectory called `wlp`. The default installation directory is `/usr/lpp/cicsts/cicsts53/wlp`. Always use the &USSHOME symbol to set the correct file path and append the `wlp` directory:

  ```
  WLP_INSTALL_DIR=&USSHOME;/wlp
  ```

  If you set this environment variable, you can also supply other environment variables and system properties to configure the Liberty JVM server. The Liberty specific environment variables are all prefixed with WLP.

- `WLP_USER_DIR`

  This 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 defaults to the following subdirectory of the JVM server: working directory:

  ```
  WLP_USER_DIR=./&APPLID;/&JVMSERVER;/wlp/usr
  ```

  We used this default location and so our Liberty user directory was as follows:

  ```
  /u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/wlp/usr
  ```

- `WLP_OUTPUT_DIR`

  This specifies the directory that contains output files for the Liberty profile. By default, the Liberty profile stores logs, the work area, and configuration files 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 the following subdirectory of the JVM server working directory:

  ```
  WLP_OUTPUT_DIR=./&APPLID;/&JVMSERVER;/wlp/usr/servers
  ```

  We used this default location and so our Liberty output directory was as follows:

  ```
  /u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/wlp/usr/servers
  ```

The following system properties are specific to the Liberty JVM server environment:

- Encoding

  A Liberty JVM server requires that JVM encoding be set to ASCII rather than the usual EBCDIC default for CICS. This is controlled by using the file.encoding JVM system property. You need to set your encoding as follows to start a Liberty JVM server.

  ```
  -Dfile.encoding=ISO-8859-1
  ```

- Autoconfigure

  For the first startup of the Liberty JVM server in CICS, enable autoconfigure, which builds the CICS Liberty server environment. This requires setting the following system properties:

  ```
  -Dcom.ibm.cics.jvmserver.wlp.autoconfigure=true
  ```

  This property instructs CICS to build the Liberty server and configuration file (`server.xml`) during startup. The default JVM profile already supplies this line and you need to remove the '#' comment and set the value to true.

- HTTP/HTTPS port

  Now change your default ports so that they do not conflict with any existing port usage. Delete the '#' before the following two lines and change '9080' and '9443' to valid free ports:

  ```
  -Dcom.ibm.cics.jvmserver.wlp.server.http.port=53088
  -Dcom.ibm.cics.jvmserver.wlp.server.https.port=53089
  ```

This is a summary of all content that is configured and enabled in the `DFHWLP.jvmprofile` by default:

```
JAVA_HOME=/usr/lpp/java/J7.0_64
WORK_DIR=.
WLP_INSTALL_DIR=&USSHOME;/wlp
-Xms128M
-Xmx256M
-Xmso128K
-Xgcpolicy:gencon
-Xscmx128M
-Xshareclasses:name=cicsts530%g,groupAccess,nonfatal
-Dcom.ibm.tools.attach.enable=no
-Dfile.encoding=ISO-8859-1
```

This is how the `JVMWLP.jvmprofile` looked after editing:

```
JAVA_HOME=/usr/lpp/java/J7.0_64
WORK_DIR=/u/reds08/cicsts53/workdir
LOG_FILES_MAX=5
-Xms128M
-Xmx256M
-Xmso128K
-Xgcpolicy:gencon
-Xscmx128M
-Xshareclasses:name=cicsts530%g,groupAccess,nonfatal
-Dcom.ibm.tools.attach.enable=no
TZ=CET-1CEST
#                WebSphere Liberty Profile server (WLP)
#                --------------------------------------
WLP_INSTALL_DIR=&USSHOME;/wlp
-Dcom.ibm.cics.jvmserver.wlp.autoconfigure=true
-Dcom.ibm.cics.jvmserver.wlp.server.http.port=53088
-Dcom.ibm.cics.jvmserver.wlp.server.https.port=53089
#
-Dcom.ibm.ws.logging.max.files=5
-Dfile.encoding=ISO-8859-1
```

6.3.6 Creating a JVMSERVER

Now it is time for you to create the CICS JVMSERVER resource definition. This is used to define the JVM profile location and control the lifecycle of the JVM server within the CICS runtime environment.

Copy the sample JVMSERVER DFHWLP from sample CSD group DFH$WLP with a new name to a CSD group of your choice to enable it to be modified. We used the following values:

Table 6-1 Values for the new JVM server

ATTRIBUTE	DEFAULT	VALUE	Description
JVMSERVER	DFHWLP	JVMWLP	Name of the Liberty JVM server in CICS
Group	DFH$WLP	REDS08LP	Name of the CSD group defined in CICS
Jvmprofile	DFHWLP	JVMWLP	Name of the JVM profile in zFS

Create a JVMSERVER definition by using CEDA with your user-specific changes as shown in Example 6-1.

Example 6-1 Create a JVM server resource definition

```
OVERTYPE TO MODIFY                                          CICS RELEASE = 0700
   CEDA  ALter JVmserver( JVMWLP   )
    JVmserver      : JVMWLP
    Group          : REDS08LP
    DEScription  ==> CICS JVM server to run WLP samples
    Status       ==> Enabled            Enabled | Disabled
    Jvmprofile   ==> JVMWLP                                    (Mixed Case)
    Lerunopts    ==> DFHAXRO
    Threadlimit  ==> 015                1-256
   DEFINITION SIGNATURE

                                                      SYSID=ISV  APPLID=CREDS08A

 PF 1 HELP 2 COM 3 END            6 CRSR 7 SBH 8 SFH 9 MSG 10 SB 11 SF 12 CNCL
```

6.3.7 Install a JVMSERVER

To install a JVMSERVER:

1. Install and enable the JVMSERVER resource into your CICS region using CEDA using the following syntax:

   ```
   CEDA EXPAND GROUP(GROUP)
   ```

   For example:

   ```
   CEDA EXPAND GROUP(REDS08LP)
   ```

   See Example 6-2.

   Example 6-2 Install JVM server resource

   ```
   CEDA EX GR(REDS08LP)
    ENTER COMMANDS
      NAME     TYPE         GROUP                                  LAST CHANGE
      JVMWLP   JVMSERVER    REDS08LP install                     28/10/15 19:28:14
   SYSID=ISV  APPLID=CREDS08A
     RESULTS: 1 TO 7 OF 7                          TIME: 17.38.09  DATE: 29/10/15
   PF 1 HELP 2 SIG 3 END 4 TOP 5 BOT 6 CRSR 7 SBH 8 SFH 9 MSG 10 SB 11 SF 12 CNCL
   ```

2. Then, check the status of the JVM server using CEMT as follows:

   `CEMT INQUIRE JVMSERVER`

   See Example 6-3.

Example 6-3 Check JVM server status

```
CEMT I JVMS
 STATUS:  RESULTS - OVERTYPE TO MODIFY
  Jvm(JVMWLP  ) Ena     Prf(JVMWLP  ) Ler(DFHAXRO )
     Threadc(011) Threadl( 015 ) Cur(61580456)

                                                      SYSID=ISV
APPLID=CREDS08A
  RESPONSE: NORMAL                                TIME: 17.49.26  DATE:
29/10/15
PF 1 HELP       3 END       5 VAR        7 SBH 8 SFH 9 MSG 10 SB 11 SF
```

If the status shows enabled, you know that your JVM environment is running. You completed your first step and have successfully validated the Java environment on CICS. If it fails to enable, check the `CSMT` log and the `stderr` log in the zFS working directory for further errors.

If you want to restart the Liberty JVM server in CICS, disable it by typing over `Ena` with `Dis`.

6.3.8 Manually tailoring server.xml

We show how you can manually configure your `server.xml` file and how to set up the parameters in it. Performing a manual configuration is especially important in a production environment, in which you might want to more tightly control the server configuration.

One of the major advantages of Liberty is its composability, based on features. You can add all the functionality that you need for your specific set of applications by choosing the features that they require. Being composable allows you to keep the server lightweight because you add only the features that you need.

Example 6-4 illustrates how to compose the `server.xml` to load specific features.

Example 6-4 Example of loading specific features on server.xml

```
<server description="ComposabilityIsTheKey">

   <featureManager>
      <feature>cicsts:core-1.0</feature>
      <feature>jsp-2.2</feature>
      <feature>servlet-3.0</feature>
      <feature>restConnector-1.0</feature>
   </featureManager>

</server>
```

6.3.9 Adding new features

You can add a wide variety of features in the <featureManager> list of features. These are the key features that we used in our initial configuration:

- The CICS feature cicsts:core-1.0 enables the CICS Liberty profile integration. This feature is required for the JVM server to start.
- The ssl-1.0 feature enables Secure Sockets Layer (SSL) support for the HTTP listeners, using either Java keystores or RACF key rings.
- The servlet-3.0 feature enables support for servlet applications written to the Java Platform, Enterprise Edition 6 web profile standard.
- 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.
- 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.

See Example 6-5.

Example 6-5 Adding new features on server.xml

```
<featureManager>
    <feature>cicsts:core-1.0</feature>
    <feature>ssl-1.0</feature>
    <feature>servlet-3.0</feature>
    <feature>jsp-2.2</feature>
    <feature>wab-1.0</feature>
</featureManager>
```

CICS supports a wide range of features from the WebSphere Application Server Liberty profile enabling Java Platform, Enterprise Edition web applications to be deployed into a Liberty JVM server. For more information about supported features, see Chapter 1, “Introduction” on page 1.

6.3.10 Configuring the HTTP and HTTPS endpoint

If you want to change the port or IP address that is used by the Liberty HTTP listener, update the httpEndpoint attribute with the host name and port numbers that you require, as shown in Example 6-6.

Example 6-6 HTTP and HTTPS port definition on server.xml

```
<httpEndpoint host="<hostname>" httpPort="53088" httpsPort="53089"
 id="defaultHttpEndpoint"/>
```

Use a port number that is not in use anywhere else, for example, by a TCPIPSERVICE in CICS.

HTTPS is available only if SSL is configured.

Note: If you are using autoconfigure, the HTTP ports are configured by using a system property and should not be modified directly in the server.xml file.

6.3.11 CICS bundle deployed applications

If you want to deploy Liberty applications that use CICS bundles, the `server.xml` file must include the following entry. See Example 6-7.

Example 6-7 Set the resource location on server.xml

```
<include location="${server.output.dir}/installedApps.xml"/>
```

The included file is used to define CICS bundle deployed applications.

6.3.12 Bundle repository

Share common OSGi bundles between applications deployed in EBAs by placing them in a directory and referring to that directory in a bundleRepository element, as in Example 6-8.

Example 6-8 List file sets in the bundle repository

```
<bundleRepository>
    <fileset dir="directory_path" include="*.jar"/>
</bundleRepository>
```

6.3.13 Global library

Share common Java archive (JAR) files between web applications (WARs and EARs) by placing them in a directory and referring to that directory in a global library definition, as in Example 6-9.

Example 6-9 Define path for common libraries

```
<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.

6.3.14 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 in Example 6-10.

Example 6-10 Set monitoring intervals

```
<config monitorInterval="5s" updateTrigger="polled"/>
```

The Liberty JVM server 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 shown in Example 6-11.

Example 6-11 Disable monitoring of dropins

```
<applicationMonitor dropins="dropins" dropinsEnabled="false" pollingRate="5s"
updateTrigger="enabled"/>
```

Note: If you disable configuration monitoring, CICS bundle installation does not work.

6.3.15 CICS default web application

The CICS default web application, `CICSDefaultApp`, illustrated in Figure 6-10 on page 123, is a built-in configuration service that validates the Liberty JVM server has started. To make the application available, add the following statement to your `server.xml` file. See Example 6-12.

Example 6-12 Enable the default web application

```
<cicsts_defaultApp deployed="true" />
```

6.3.16 JTA transaction log

When Liberty applications use 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 the `server.xml` file, as shown in 6.3.17, “Sample server.xml” on page 115. You only need to consider setting this location if your application uses either EJB container-managed transactions or the Java Transaction API using the UserTransaction interface. See Example 6-13.

Example 6-13 Define directory of the transaction log

```
<transaction transactionLogDirectory="/cicsts53/CICSPRD/JVMWLP/tranlog/"/>
```

6.3.17 Sample server.xml

Example 6-14 shows what the `server.xml` file looks like when the configuration is complete.

Example 6-14 Sample server.xml with our updates

```
<server description="JVMWLP sample">

<featureManager>
    <feature>cicsts:core-1.0</feature>
    <feature>ssl-1.0</feature>
    <feature>servlet-3.0</feature>
    <feature>jsp-2.2</feature>
    <feature>wab-1.0</feature>
</featureManager>
...
<httpEndpoint host="<hostname>" httpPort="53088" httpsPort="53089"
 id="defaultHttpEndpoint"/>
...
<include location="${server.output.dir}/installedApps.xml"/>
...
<cicsts_defaultApp deployed="true" />
...
<bundleRepository>
    <fileset dir="/u/user/bundles/" include="TESTBUND.jar"/>
</bundleRepository>
...
```

```
<library id="global">
    <fileset dir="/u/user/bundles/global/" include="GLOBUND.jar"/>
</library>
...
<config monitorInterval="5s" updateTrigger="polled"/>
...
<applicationMonitor dropins="dropins" dropinsEnabled="false" pollingRate="5s"
updateTrigger="enabled"/>
...
<transaction transactionLogDirectory="/u/cics/CICSPRD/JVMWLP/tranlog/"/>
...
</server>
```

6.3.18 Check welcome page

A quick way to check your Liberty JVM server status is to use the home page created by the Liberty server. This page is a simple welcome page, which is displayed by Liberty when a browser attempts to access its HTTP or HTTPS port without a valid URI. You can access this page by visiting the URL *<hostname>*:*<port>*.

In our example, we used: http://*<hostname>*:53088/, which displayed the following page. For more information about how to verify your configuration, see 6.5, “Verifying the configuration” on page 122.

Figure 6-1 Liberty welcome page

For more information, see *Configuring a Liberty JVM server*, at:

https://www.ibm.com/support/knowledgecenter/SSGMCP_5.3.0/com.ibm.cics.ts.java.doc/JVMserver/config_jvmserver_liberty.html

6.4 Defining and installing a CICS bundle

A CICS BUNDLE resource definition is required to install Liberty applications via a CICS bundle project. The BUNDLE resource definition then refers to the deployed versions of these components in a zFS directory. This directory simply contains the deployable artifacts and a manifest that describes the bundle and its potential dependencies. CICS bundles provide a useful way of grouping and managing related resources and providing a versioning schema and the ability to declare dependencies on other resources outside the bundle. CICS does not dynamically create prerequisite systems, but can check that they exist in the CICS region. Application developers can use CICS bundle project for packaging Java and Web applications, and for creating cloud-based applications and business events. System programmers need to use CICS bundles for system events and policies.

In this section, we install CICS bundle projects using both the CEDA transaction server and CICS Explorer. We use the bundle project to install our sample Hello World web application into the Liberty JVM server and control its availability. For more information about creating this application, see Chapter 5, “Developing and deploying applications” on page 69.

6.4.1 BUNDLE resource definition

We are ready to create the BUNDLE resource definition. You have to copy the sample BUNDLE definition ‘WLPHELLO’ from sample group ‘DFH$WLP’ with a new name to a group of your choice. These are the settings that we used. See Table 6-2.

Table 6-2 Values for the new bundle resource definition

ATTRIBUTE	DEFAULT	VALUE	Description
BUNDLE	WLPHELLO	HELLOWO	Name of the BUNDLE definition in CICS
Group	DFH$WLP	REDS12LP	Name of the CSD group defined in CICS
BUndledir	/u/reds12/bundles/com.ibm.liberty.HelloWorld.cics_1.0.0		zFS directory of the deployed CICS Bundle

Create a BUNDLE definition using CEDA and set the system-dependent attributes (Group, BUndledir), as shown in Example 6-15.

Example 6-15 Create a bundle definition with CEDA

```
OVERTYPE TO MODIFY                                        CICS RELEASE = 0700
   CEDA  ALter Bundle( HELLOWO )
    Bundle         : HELLOWO
    Group          : REDS12LP
    DEScription  ==> CICS BUNDLE FOR LIBERTY HELLO TEST
    Status       ==> Enabled            Enabled | Disabled
    BUndledir    ==> /u/reds12/bundles/com.ibm.liberty.HelloWorld.cics_1.0.0
    (Mixed Case) ==>
    BAsescope    ==>
```

```
    (Mixed Case) ==>

   DEFINITION SIGNATURE
+  CHANGETime     : 30/10/15 18:40:12

 PF/PA KEY UNDEFINED                                SYSID=ISV  APPLID=CREDS12A

 PF 1 HELP 2 COM 3 END            6 CRSR 7 SBH 8 SFH 9 MSG 10 SB 11 SF 12 CNCL
```

6.4.2 Install a BUNDLE resource definition

To install a BUNDLE resource definition:

1. Install and enable the CICS bundle into your CICS region using CEDA using the following syntax:

   `CEDA EX GR(GROUP) BUNDLE(BUNDLENAME)`

   For example:

   `CEDA EX GR(REDS12LP) BUNDLE(HELLOWO)`

   See Example 6-16.

   Example 6-16 Install and enable CICS bundle into CICS region

```
CEDA EX GR(REDS12LP) BUNDLE(HELLOWO)
 ENTER COMMANDS
  NAME     TYPE         GROUP                                  LAST CHANGE
  HELLOWO  BUNDLE       REDS12LP install                     30/10/15 18:42:48

                                                        SYSID=ISV  APPLID=CREDS12A
  RESULTS: 1 TO 1 OF 1                             TIME: 20.35.20  DATE: 30/10/15
PF 1 HELP 2 SIG 3 END 4 TOP 5 BOT 6 CRSR 7 SBH 8 SFH 9 MSG 10 SB 11 SF 12 CNCL
```

2. Then, check the status of the Liberty JVM server using the CEMT transaction, as follows:

   `CEMT INQUIRE BUNDLE`

   See Example 6-17.

   Example 6-17 Check bundle status

```
CEMT I BUNDLE
 STATUS:  RESULTS - OVERTYPE TO MODIFY
  Bun(HELLOWO ) Ena         Par(00001) Tar(00001)
     Enabledc(00001) Bundlei(com.ibm.liberty.HelloWorld)

SYSID=ISV  APPLID=CREDS12A
  RESPONSE: NORMAL                                TIME: 20.38.33  DATE: 30/10/15
PF 1 HELP       3 END       5 VAR        7 SBH 8 SFH 9 MSG 10 SB 11 SF
```

If the status shows `enabled`, that means that the bundle resource is enabled. Check the Liberty `messages.log` file to see whether the Liberty application has deployed and is available. To check that the Liberty application is running, access it by using the HTTP or HTTPS port of the Liberty JVM server (Figure 6-2 on page 119). For example:

`http://<hostname>:53120/com.ibm.liberty.HelloWorld/Hello`

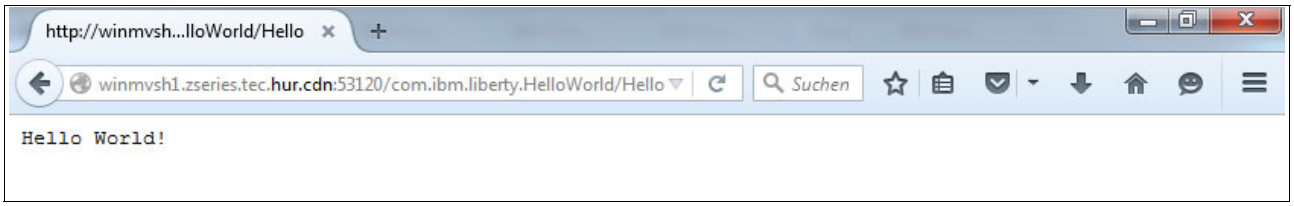


Figure 6-2 Hello World application

6.4.3 BUNDLE definition by CICS Explorer

The next steps explain how you can define a CICS Bundle with CICS Explorer. Switch to the CICS SM perspective in your Eclipse Environment:

1. Click **Window** → **Open Perspective** → **Other** → **CICS SM**. See Figure 6-3.

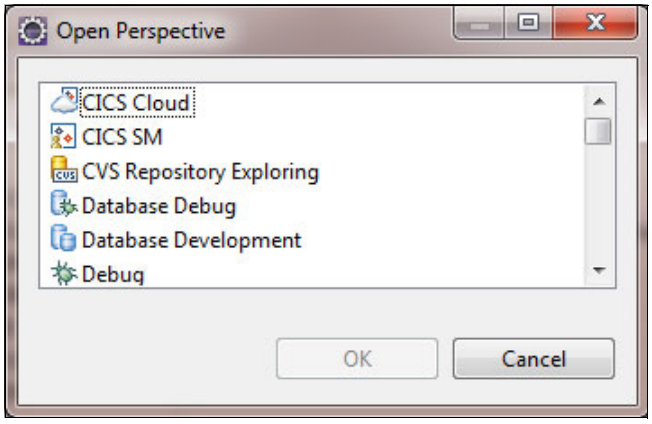


Figure 6-3 Open Perspective window

2. Select your **CICS region** from the list on the left side on the window. See Figure 6-4.

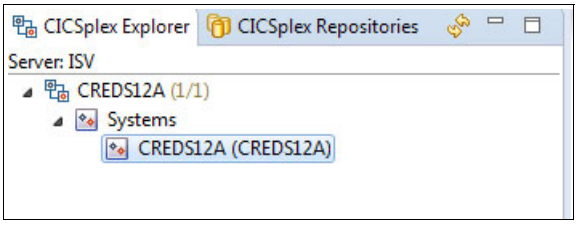


Figure 6-4 Select your CICS region

3. Go to **Window** → **Show View** → **Other** and select **Bundle Definitions**. Click **OK**. See Figure 6-5 on page 120.

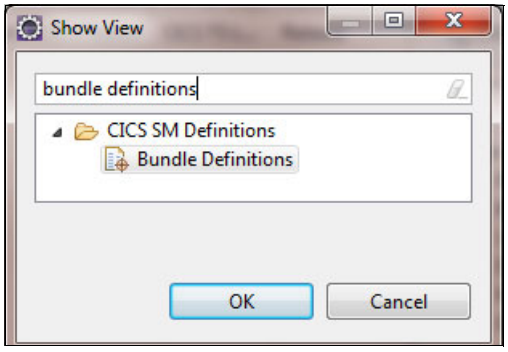


Figure 6-5 Bundle definition

4. Right-click anywhere in the view and select **New**. See Figure 6-6.

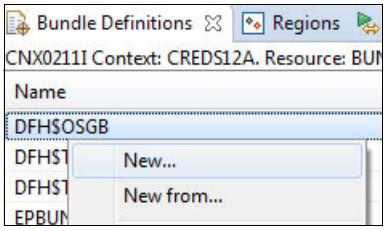


Figure 6-6 New bundle definition menu

5. Now you can edit the details of your CICS bundle. We use the same parameters as defined in Chapter 5, “Developing and deploying applications” on page 69. Create your bundle by clicking **Finish**. See Figure 6-7 on page 121.

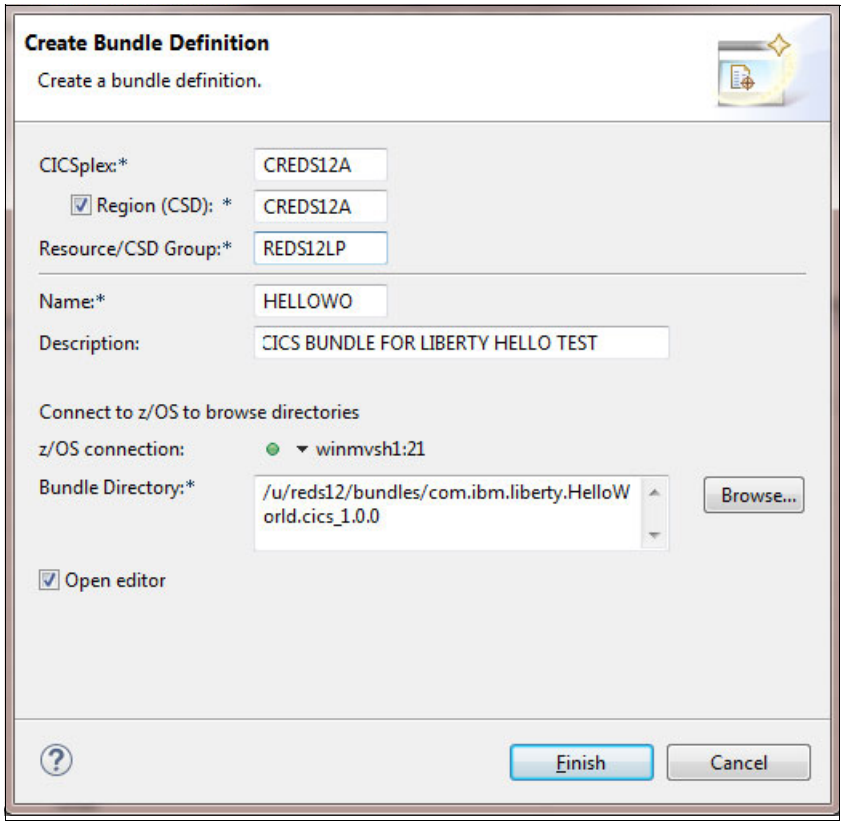


Figure 6-7 Create Bundle Definition window

6.4.4 BUNDLE installation by CICS Explorer

The next step explains how to install the CICS Bundle with CICS Explorer. Switch to the CICS SM perspective in your CICS Explorer.

1. Right-click **CICS Bundle definition** → **Install** → **select CICS region** → **OK**. See Figure 6-8.

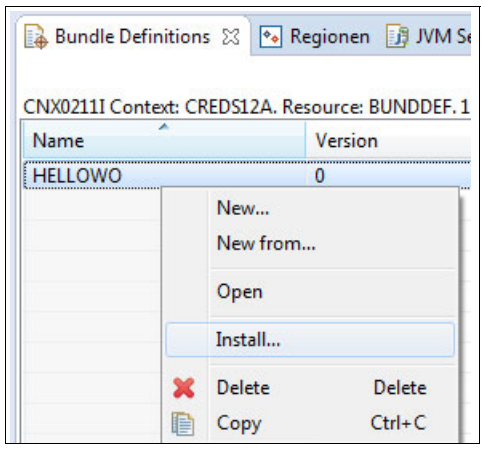


Figure 6-8 Install CICS Bundle window

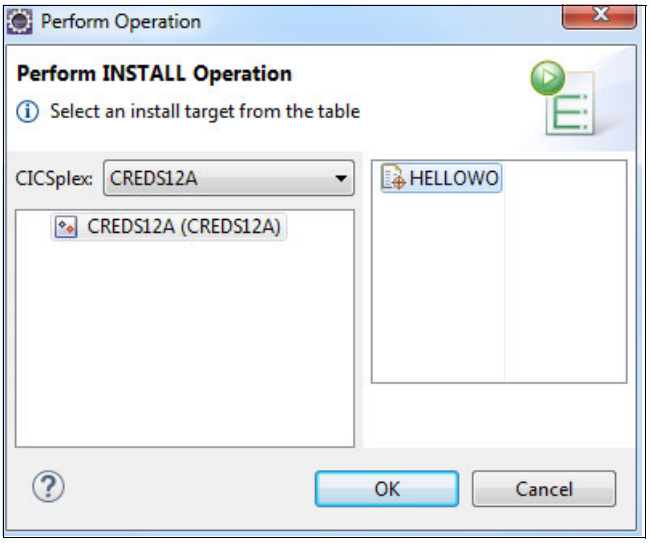


Figure 6-9 Select CICS region

6.5 Verifying the configuration

The default web application is a built-in web application that is activated in the `server.xml` file. It delivers helpful information about the Liberty JVM server, such as:

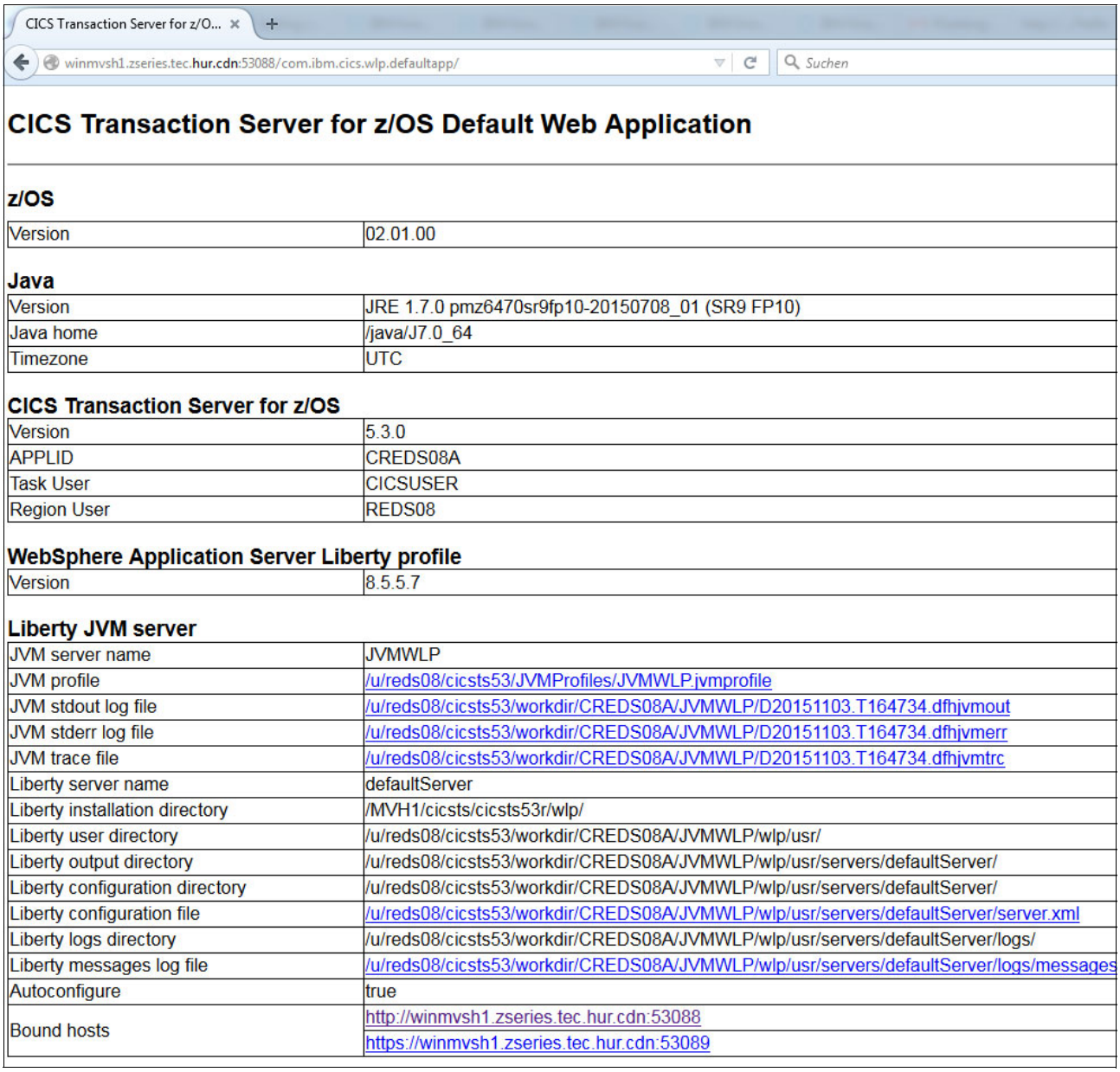
- Reports on runtime configuration
- Provides access to log files
- Defaults to SSL if security set-up

You can run the application by using the following style of URL:

`http://<server>:<port>/com.ibm.cics.wlp.defaultapp/`

Figure 6-10 on page 123 shows the results of entering the following URL:

`http://<hostname>:53088/com.ibm.cics.wlp.defaultapp/`



CICS Transaction Server for z/OS Default Web Application

z/OS

Version	02.01.00

Java

Version	JRE 1.7.0 pmz6470sr9fp10-20150708_01 (SR9 FP10)
Java home	/java/J7.0_64
Timezone	UTC

CICS Transaction Server for z/OS

Version	5.3.0
APPLID	CREDS08A
Task User	CICSUSER
Region User	REDS08

WebSphere Application Server Liberty profile

Version	8.5.5.7

Liberty JVM server

JVM server name	JVMWLP
JVM profile	/u/reds08/cicsts53/JVMProfiles/JVMWLP.jvmprofile
JVM stdout log file	/u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/D20151103.T164734.dfhjvmout
JVM stderr log file	/u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/D20151103.T164734.dfhjvmerr
JVM trace file	/u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/D20151103.T164734.dfhjvmtrc
Liberty server name	defaultServer
Liberty installation directory	/MVH1/cicsts/cicsts53r/wlp/
Liberty user directory	/u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/wlp/usr/
Liberty output directory	/u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/wlp/usr/servers/defaultServer/
Liberty configuration directory	/u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/wlp/usr/servers/defaultServer/
Liberty configuration file	/u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/wlp/usr/servers/defaultServer/server.xml
Liberty logs directory	/u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/wlp/usr/servers/defaultServer/logs/
Liberty messages log file	/u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/wlp/usr/servers/defaultServer/logs/messages
Autoconfigure	true
Bound hosts	http://winmvsh1.zseries.tec.hur.cdn:53088
	https://winmvsh1.zseries.tec.hur.cdn:53089

Figure 6-10 CICS TS default web application

6.5.1 Log file analysis

The most important log file created by each startup of Liberty JVM server is the `messages.log` file. The `messages.log` file can be found by default in the logs directory in the same zFS directory as your Liberty server configuration. To locate this directory, the root directory for your Liberty JVM server log files is in the working directory used by the JVM server.

In our example, the CICS APPLID is CREDS08A. The Liberty JVM Server is called JVMWLP. And the Liberty log files can be found in this location:

```
/u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/wlp/usr/servers/defaultServer/logs
```

Liberty outputs useful information about application and system activity to this `messages.log` file. This file serves as one of the main ways to monitor your Liberty JVM server. This section briefly shows several generated messages written to the `messages.log` file. See Figure 6-11.

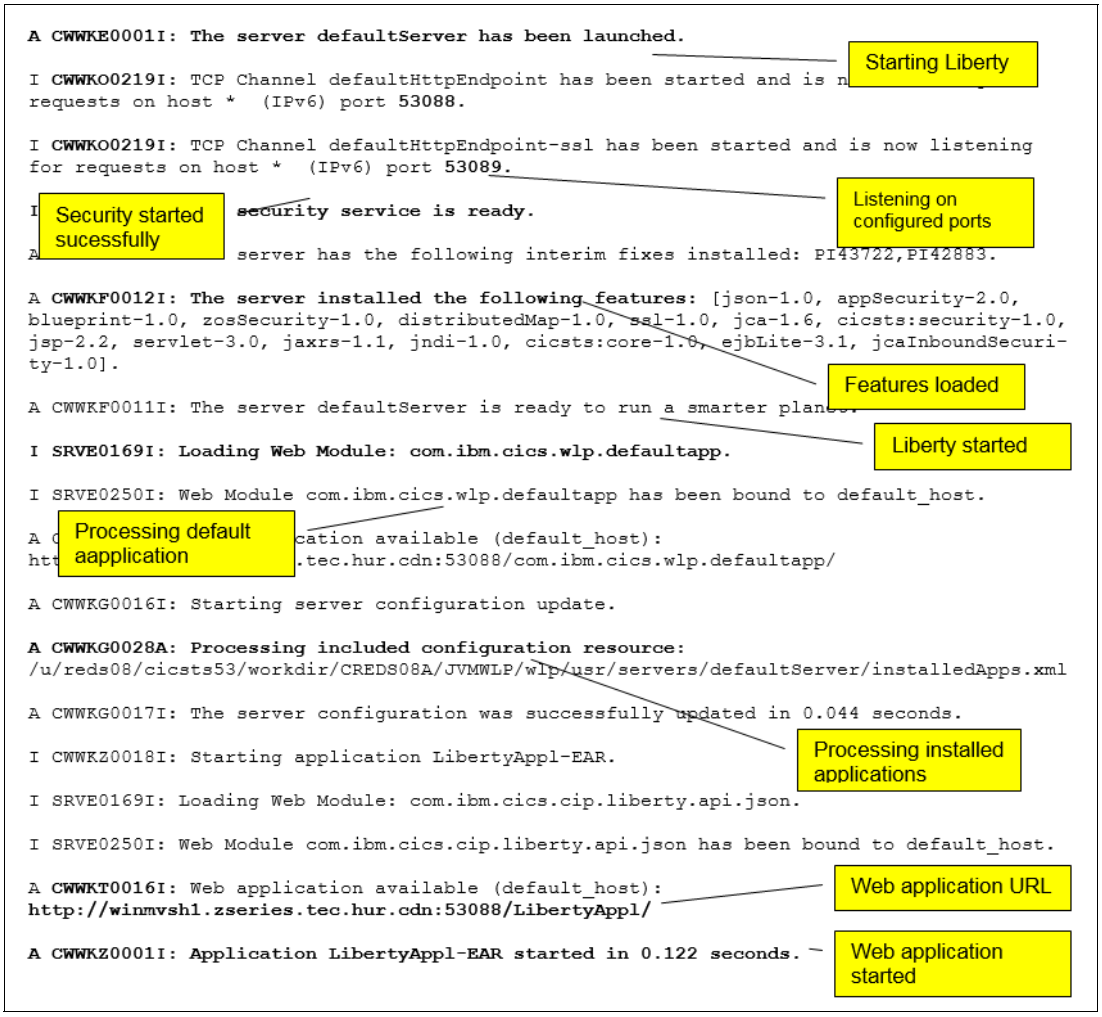


Figure 6-11 Sample from messages.log

For more information about logs, see Chapter 12, “Troubleshooting” on page 211.

6.6 Setting up a Liberty JVM server using CICS Explorer

In this section, we show you how to create and install a Liberty JVM server using CICS Explorer.

To start a Liberty JVM server in CICS, you need a JVM profile that describes your server. In our example, we use the example server provided with the CICS installation as the basis for our server. We are using FTP to connect to zFS on the hosting mainframe. This chapter requires a few technical requirements.

You need to install these items:

- CICS Explorer and FTP

  The IBM Knowledge Center for CICS TS V5.3 has a section that shows you how to install and configure the CICS Explorer.

  Open “IBM Knowledge Center for CICS Transaction Server 5.3.0” – “Installing” – “CICS Explorer installation.”

  http://www.ibm.com/support/knowledgecenter/SSGMCP_5.3.0/com.ibm.cics.ts.installation.doc/topics/explorer_configure_chapter.html

- SMSS CICS region or WUI Server

  The IBM Knowledge Center for CICS TS V5.3 has a section that shows you how to set up an SMSS CICS region or a CICSPlex SM WUI Server.

  Open “IBM Knowledge Center for CICS Transaction Server 5.3.0” – “Configuring” – “Setting up access for CICS Explorer.”

  http://www.ibm.com/support/knowledgecenter/SSGMCP_5.3.0/com.ibm.cics.ts.clientapi.doc/topics/clientapi_setup.html

6.6.1 CICS Explorer connection

Ensure that your CICS Explorer is connected to the CICS system using a CMCI connection to CICS and an FTP connection to your z/OS FTP server.

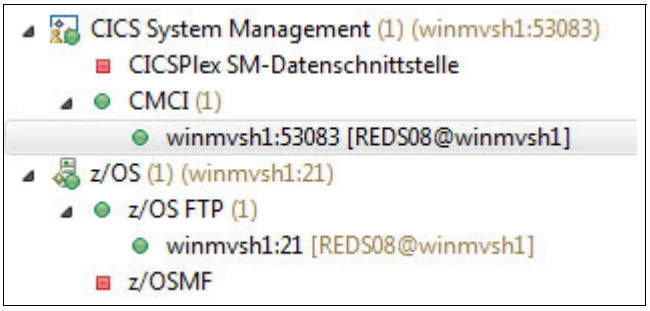


Figure 6-12 CICS Explorer connections

6.6.2 Locate the zFS directories

CICS provides several example JVM profiles to help you get started with JVM servers. We used the Liberty sample profile to help us set up our profile. To locate the directories, locate the UNIX System Services home and the JVM profiles directory. The **USSHOME** and **JVMPROFILEDIR** system initialization parameters list these locations.

You can find out these values for any CICS region from the CICS Explorer Regions view using **Show SIT parameters** → **Combined**:

```
USSHOME=/usr/lpp/cicsts/cicsts53
JVMPROFILEDIR=/u/reds08/cicsts53/JVMProfiles/
```

6.6.3 Copy the JVM profile

In the z/OS UNIX Files view in your Eclipse environment, enter your `USSHOME` directory into the filter box, and then select the `JVMProfile` subdirectory. You will see a number of sample profiles listed. The one that we need is `DFHWLP`. Copy this file to your `JVMPROFILEDIR` location with an appropriate name. In our example, we named it `JVMWLP.jvmprofile`.

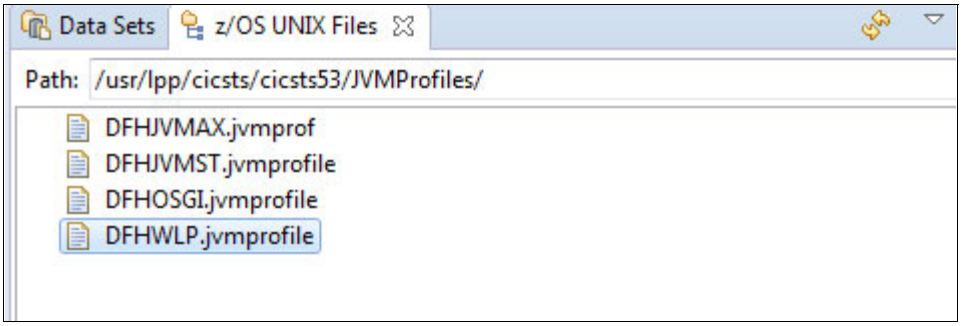


Figure 6-13 Default JVMPROFILE

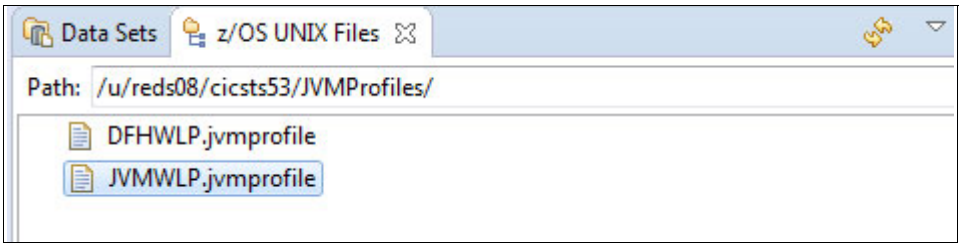


Figure 6-14 Example JVMPROFILE

6.6.4 Edit the JVM profile

The default profile contains most of the settings that are required to run a Liberty JVM server in CICS. Here, we add in only the necessary options to enable the autoconfigure feature. For the remainder of the profile, retain the default values:

```
-Dcom.ibm.cics.jvmserver.wlp.autoconfigure=true
-Dcom.ibm.cics.jvmserver.wlp.server.http.port=53088
-Dcom.ibm.cics.jvmserver.wlp.server.https.port=53089
```

For more information about editing the profile, see 6.3.1, “JVM profile” on page 106.

6.6.5 Create the Liberty JVM server definition

We now need to create and install a JVM server definition in CICS. Follow these steps:

1. Select the connected **CICS region** in the CICSplex Explorer window on the left. See Figure 6-15.

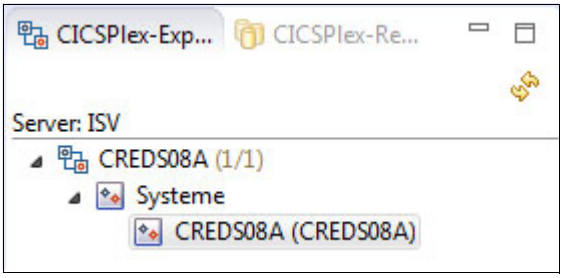


Figure 6-15 CICS region selected

2. Open the JVM Server Definitions view in Explorer, and select **Window** → **Show View** → **Other**. See Figure 6-16.

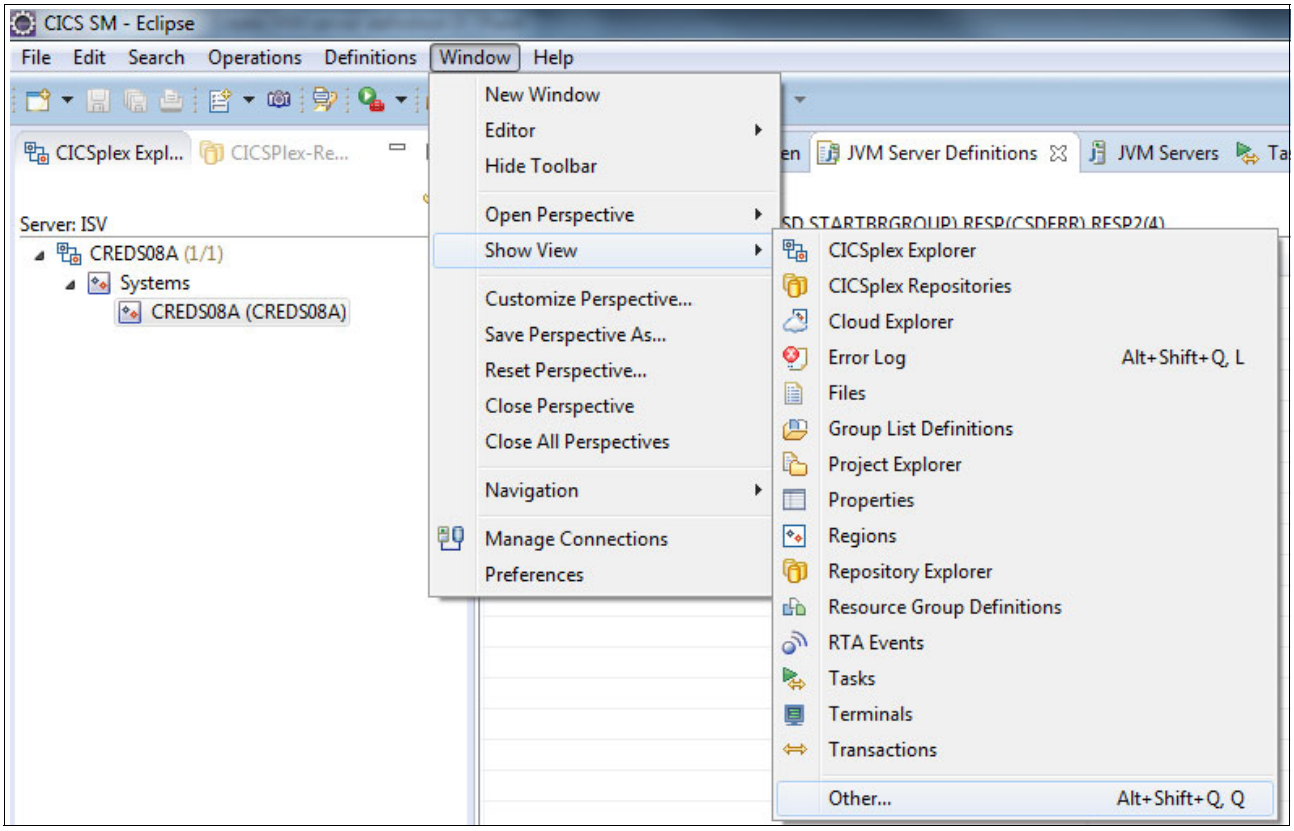


Figure 6-16 Show view menu

3. Enter **JVM Server Definitions** in the filter box, select **Search**, and click **OK**. See Figure 6-17.

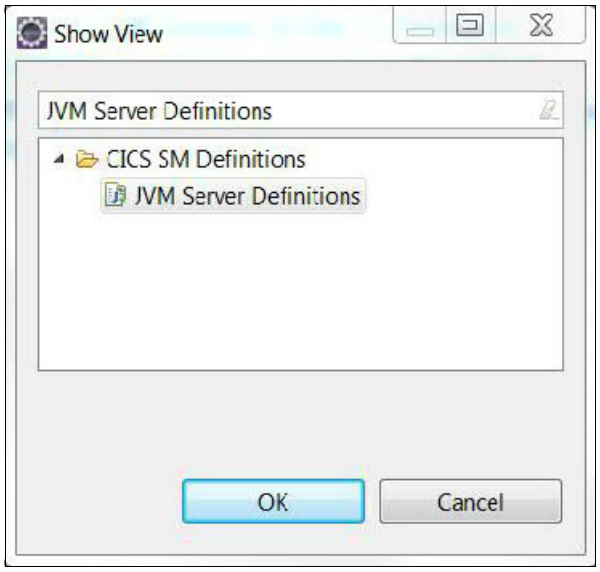


Figure 6-17 Choose JVM server definitions

4. In the newly opened JVM Server Definitions window, right-click anywhere and select **New** to open the New JVM Server Definition wizard, which is shown in Figure 6-18.
5. Complete the fields for the JVM server, pointing the profile to the one created in `JVMWLP`. Click **Finish** (Figure 6-18).

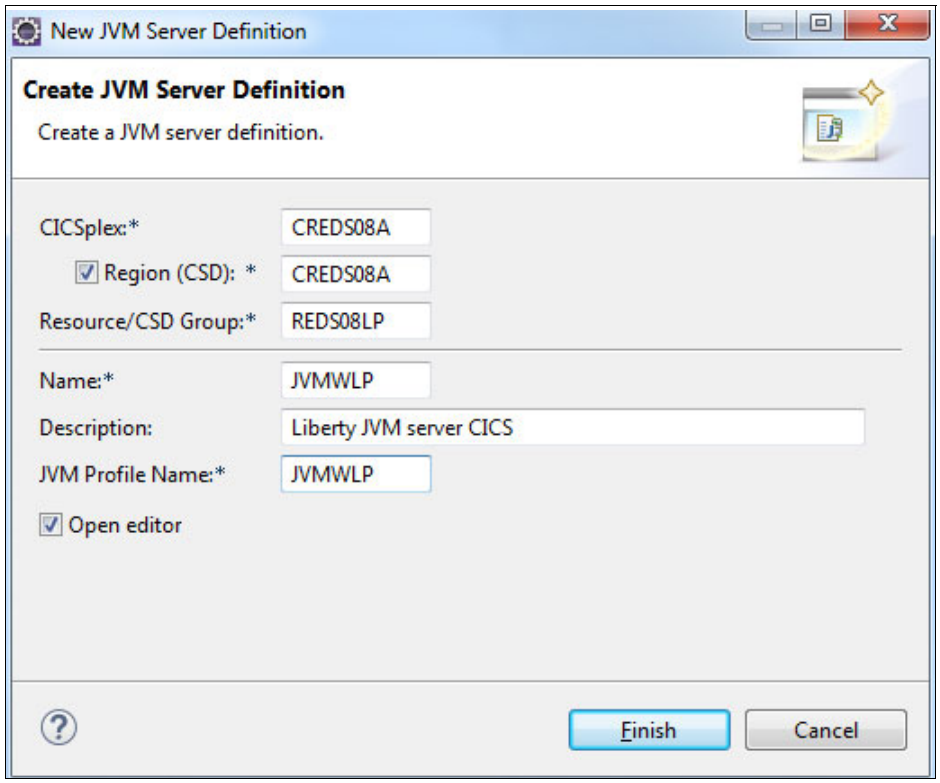


Figure 6-18 JVM server definitions

6.6.6 Install Liberty JVM server definition

You now see the JVM server definition that you created listed in the JVM Server Definitions view.

1. Right-click your **JVM server** and select **Install**. See Figure 6-19.

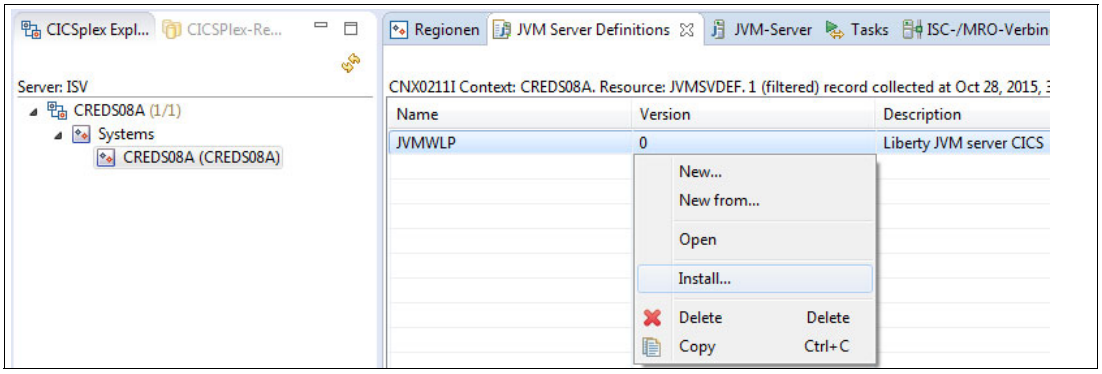


Figure 6-19 Select the JVM server to install

2. In the window that displays, select your **CICS region** and then click **OK**.

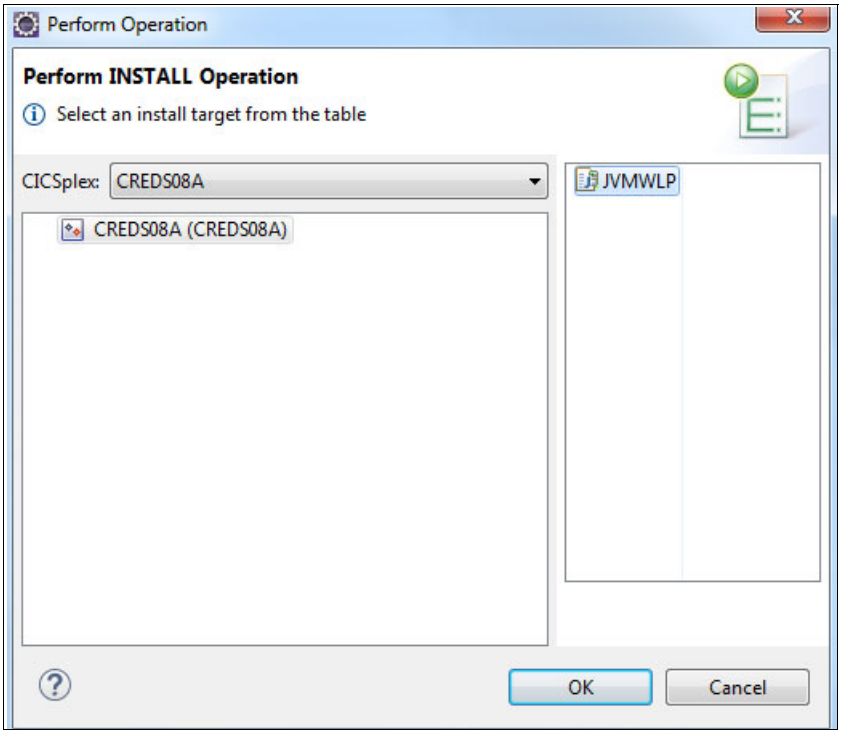


Figure 6-20 Install the chosen JVM server

3. CICS installs your JVM server definition.

4. To see your JVM server installation, open the JVM Servers view, and go to **Window** → **Show View** → **Other**. Type `JVM Servers` into the filter box and click **OK**.

   Your JVM server is listed in this view with a status of Enabled. Liberty is now available for use. See Figure 6-21.

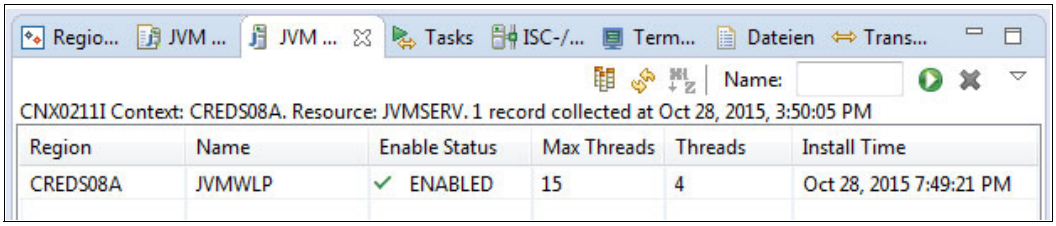


Figure 6-21 JVM server is ENABLED

6.6.7 Liberty JVM server log files

Liberty prints a message to its `messages.log` file when it is available for use. Having the status of Enabled confirms that the JVM server is ready for use. However, look at the logs anyway because logging is slightly different from other JVM servers in CICS. Liberty JVM servers still produce `dfhjvmout`, `dfhjvmerr`, and `dfhjvmtr` files when run. However, Liberty provides its own logging file named `messages.log`.

The `messages.log` file (shown in Figure 6-22 on page 131) can be found by default in the logs directory in the same zFS directory as your server configuration. To locate this directory, first open the z/OS UNIX Files view in your Explorer environment.

The directory for your Liberty JVM server output files is in the working directory of the JVM server. In our example, the CICS APPLID is CREDS08A and the Liberty JVM Server is named JVMWLP so our `message.log` is in this location:

`/u/reds08/cicsts53/workdir/CREDS08A/JVMWLP/wlp/usr/servers/defaultServer/logs`

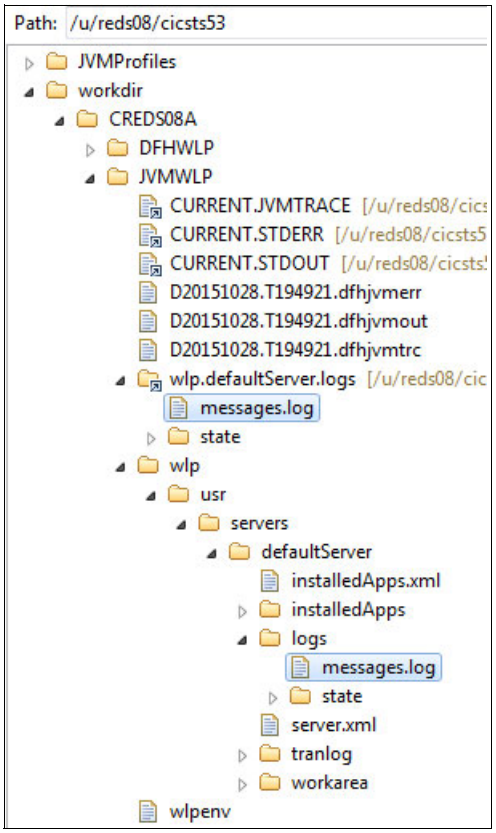


Figure 6-22 Path to messages.log

Liberty outputs a large amount of useful information about application and system activity to the `messages.log` file. It serves as one of the main ways to monitor your Liberty JVM server.

For now, we are looking for a message that reads as follows and indicates that Liberty started successfully:

```
A CWWF0011I: The server defaultServer is ready to run a smarter planet.
```

For more information about verifying the configuration and analyzing the log file, see 6.5, “Verifying the configuration” on page 122.

7

Configuring the web server plug-in

In this chapter, we describe how we built a web server plug-in configuration file and deployed the plug-in to a web server to load balance requests across a set of Liberty servers. For more information about the workload management options available with Liberty in CICS, see Chapter 4, “Security options” on page 57.

This chapter describes the following topics:

- Configuring the web server plug-in
- Install the HTTP Server and plug-in
- Liberty server configuration
- Configure the plug-in
- Test the configuration

7.1 Configuring the web server plug-in

The web server plug-in is configured by generating a `plugin-cfg.xml` file on the Liberty server that is then copied to the machine hosting the web server. This configuration file can be generated by calling the generateDefaultPluginConfig or the generatePluginConfig MBean operation that is provided by Liberty. This JMX MBean can either be invoked remotely, by using the JConsole utility supplied with the IBM Java SDK in combination with the Liberty server restConnector-1.0 feature. Or, the JMX MBean can be invoked by developing a custom JMX application to invoke the required operation on the MBean.

The plug-in must be installed into each web server that redirects requests to the target Liberty servers. See Figure 7-1.

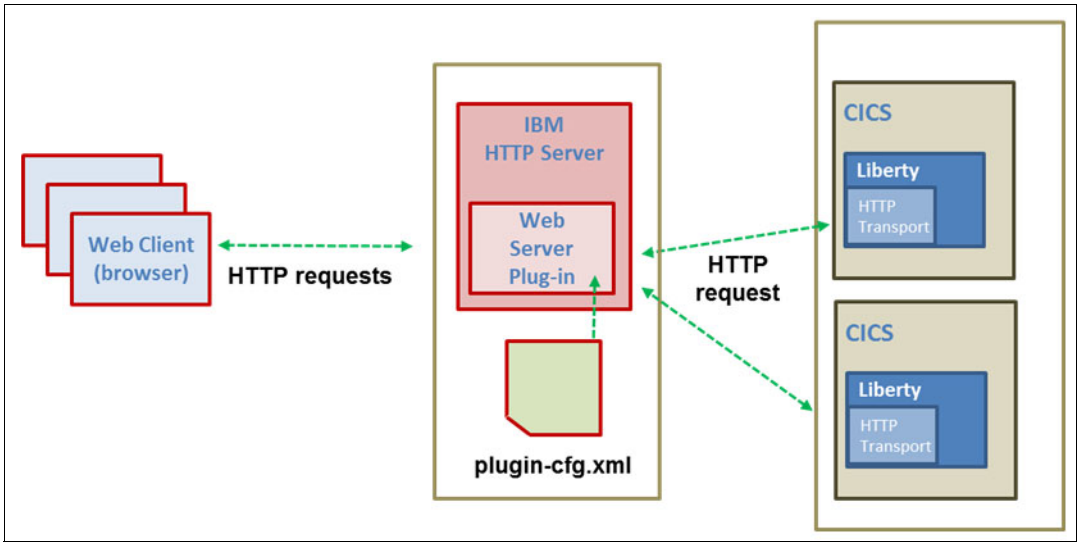


Figure 7-1 Web server plug-in configuration

Several steps are needed to configure the IBM HTTP Server to forward HTTP requests to the Liberty JVM server running in CICS. In this scenario, we set up a remote web server on a Microsoft Windows PC and configure it to route requests to the CICS region running on z/OS. These are the high-level steps:

1. Install the IBM HTTP Server and the plug-in
2. Configure the Liberty server
3. Configure the plug-in configuration file
4. Test the configuration

7.2 Install the HTTP Server and plug-in

You can install a web server and web server plug-in by using IBM Installation Manager. You can perform the installation by using the graphical user interface (GUI), command-line interface (CLI), console mode, or silently. To install the IBM HTTP Server and web server plug-in, complete the following steps:

1. Using IBM Installation Manager, configure the Installation Manager repository to point to the WebSphere supplements package.

   The IBM HTTP web server and the web server plug-in are provided as part of the WebSphere Application Server supplements to the WebSphere Application Server. You can download the fixes for WebSphere Application Server V8.5.5 at:

   `http://www.ibm.com/support/docview.wss?uid=swg24040533`

2. Click the **Install wizard** and select the following packages:

   a. IBM HTTP Server for WebSphere Application Server

   b. Web server plug-ins for IBM WebSphere Application Server

3. Click **Next**.
4. Read and accept the license agreement, and click **Next**.
5. Specify the installation directory for each package.
6. Ensure the “Create a new package group” option is selected, then click **Next**.
7. In the Features window, verify the selected packages, and click **Next**.
8. Configure a port number for the IBM HTTP Server to communicate. By default, it listens on ports 80 and 443. Click **Next**.
9. Review the settings in the Summary window and click **Install**.
10. When the installation is complete, review the summary and click **Finish**.

Now that everything is installed, we need to configure the web server plug-in. Because the Liberty profile is running in CICS, we generate the `plugin-cfg.xml` file for the Liberty profile server by calling the WebSphere:name=com.ibm.ws.jmx.mbeans.generatePluginConfig MBean using a remote Representational State Transfer (REST) connection from the JConsole tool running on our workstation.

Remote access through the REST connection is protected by a single administrator role. Also, Secure Sockets Layer (SSL) is required in order to keep the communication confidential.

7.3 Liberty server configuration

The first step is to configure the ports to be used by the plug-in to communicate with the Liberty server:

1. Specify the web server HTTP and HTTPS ports in the `server.xml` file. By default, the values are 80 and 443:

   ```
   <pluginConfiguration webserverPort="80" webserverSecurePort="443"/>
   ```

2. In the `server.xml` file, make the following changes:
   a. Ensure that the `httpEndpoint` element specifies `host="*"`
   b. Enable the REST connector in the `server.xml` file

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

   c. Enable SSL communication for the Liberty profile
   d. Enable the SSL Liberty feature in `server.xml`

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

      Add the keystore service object entry to the `server.xml` file. If you let CICS autoconfigure the `server.xml` file, this element has been added for you. Note the keystore password:

      ```
      <keyStore id="defaultKeyStore" password="defaultPassword"/>
      ```

   e. Map a user to the administrator role for Liberty. Because we are mapping a z/OS user to an administrator role, we must configure the SSL definitions with SAF and define the administrator in IBM Resource Access Control Facility (RACF):

      Configure SSL definitions with SAF:

      ```
      <safRegistry id="saf"/>
      <safCredentials profilePrefix="CREDS1OA"/>
      <safAuthorization id="saf"/>

      <ssl id="defaultSSLConfig" keyStoreRef="defaultKeyStore" sslProtocol="TLS"/>
      <keyStore id="defaultKeyStore" password="defaultPassword"/>
      <trustStore id="defaultTrustKeyStore" password="defaultTrustPassword"/>
      ```

   f. Configure the administrator role through RACF authorization. By default, the SAF profile name for the administrator role is as follows: BBGZDFLT.com.ibm.ws.management.security.resource.Administrator.

      That profile must exist in the EJBROLE SAF class, and the admin user must be granted READ access to it. For example:

      ```
      RDEFINE EJBROLE
      BBGZDFLT.com.ibm.ws.management.security.resource.Administrator UACC(NONE)
      PERMIT BBGZDFLT.com.ibm.ws.management.security.resource.Administrator
      ID(MSTONE1) ACCESS(READ) CLASS(EJBROLE)
      ```

   g. Start the Liberty JVM server in CICS by installing the JVMSERVER resource definition.
   h. Access the REST connector.
   i. Start the Liberty JVM server in CICS. In the `messages.log` file, you see the following messages:

      ```
      CWWKT0016I: Web application available (default_host):
      http://<hostname>:53103/IBMJMXConnectorREST/
      ```

   j. Go to this URL in a browser to verify that you have set up a successful connection to the REST connector. You need to input your z/OS user ID and password.

7.4 Configure the plug-in

To configure the `plugin-cfg.xml` file, we will be calling the generatePluginConfig MBean. We use the jconsole utility that is provided with Java to generate this plug-in. However, we must set some environment variables when we call this utility on the command line:

1. In the `messages.log` file, look for the following message and note the URL:

   ```
   CWWKX0103I: The JMX REST connector is running and is available at the following
   service URL:

   service:jmx:rest://<hostname>:53104/IBMJMXConnectorREST
   ```

2. Create the `C:\restclient` directory on your workstation:
   a. In the CICS region on z/OS:
      i. Use FTP to transfer $USSHOME/clients/restConnector.jar to C:\restClient
      ii. Use FTP to transfer <WLP_USER_DIR>/resources/security/key.jks to C:\restClient
   b. On your workstation:
      i. Copy <IBM_JDK_root>/jre/lib/security/java.policy to C:\restClient
      ii. Replace the ssl.SocketFactory.provider and ssl.ServerSocketFactory.provider with the following, if they exist:

      ```
      ssl.SocketFactory.provider=
      ssl.ServerSocketFactory.provider=
      ```

3. In a command window, type SET JAVA_HOME=<IBM_JDK_root>\java on your workstation. This is the directory where you install the WebSphere Customization Toolbox in step 2 above.
4. Change to the java directory: `CD %JAVA_HOME%\bin`
5. From the command window, run the jconsole Java utility. The full command follows:

   ```
   jconsole -J-Djava.security.properties=C:\restClient\java.policy
   -J-Djava.class.path=%JAVA_HOME%\lib\jconsole.jar";%JAVA_HOME%\lib\tools.jar";
   "C:\restClient\restConnector.jar"
   -J-Djavax.net.ssl.trustStore=C:\restClient\key.jks
   -J-Djavax.net.ssl.trustStorePassword=defaultPassword
   -J-Djavax.net.ssl.trustStoreType=jks
   -J-Dcom.ibm.ws.jmx.connector.client.disableURLHostnameVerification=true
   ```

Note: This command must be all on one line.

- The `trustStorePassword` is the password specified on the keystore element in `server.xml`.
- If jconsole fails to connect over SSL, add the following line to open a diagnostic window when trying to connect:

  ```
  -J-Djavax.net.debug=ssl
  -J-Djava.util.logging.config.file=C:\temp\logging.properties
  ```

6. In the jconsole window, do a remote connection to the CICS system on z/OS:
   a. Enter the `service:jmx:rest://<hostname>:53104/IBMJMXConnectorREST` you noted in Step 2 on page 134 for the remote process.
   b. Enter your z/OS **user ID** and **password** and click **Connect**. See Figure 7-2.

Figure 7-2 Establishing a remote connection

7. In the MBeans tab, go to **WebSphere** → **com.ibm.ws.jmx.mbeans.generatePluginConfig** → **Operations** → **generatePluginConfig**.
8. Under Operation Invocation, click the **`generatePluginConfig`** button:

   This creates a `plugin-cfg.xml` file on z/OS within the `${server.output.dir}` directory.
9. Use FTP to transfer this `plugin-cfg.xml` file to the machine where the web server resides.

7.4.1 Configure the plug-in to forward requests

Plug-in properties used for mapping HTTP requests within the configuration file are:

- *Route*: Identifies the server for the plug-in to send a request to. The plug-in checks each route to find the appropriate route for the request.
- *VirtualHost*: Contains multiple defined host:port pairs for the web servers and application servers.
- *URIGroup*: Specifies a group of URIs that are indicated on the HTTP request line. The route compares the incoming URI with the URIs in the group to determine if the request should be forwarded to the application server for handling.
- *Transport*: Provides the information that is necessary to determine the location of the application server to which the request is sent. If the server has multiple transports that are defined to use the same protocol, the first one is used.
- *Log*: Is useful for identifying where the web server log is written.

Example 7-1 shows what the `plugin-cfg.xml` file contains.

Example 7-1 plugin-cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?><!--HTTP server plugin config file for String generated on
2015.11.12 at 23:23:56 GMT-->
<Config ASDisableNagle="false" AcceptAllContent="false" AppServerPortPreference="HostHeader"
ChunkedResponse="false" FIPSEnable="false" IISDisableNagle="false" IISPluginPriority="High"
IgnoreDNSFailures="false" RefreshInterval="60" Re-sponseChunkSize="64" SSLConsolidate="false"
TrustedProxyEnable="false" VHostMatchingCompat="false">
   <Log LogLevel="Error" Name="String/logs/String/http_plugin.log"/>
   <Property Name="ESIEnable" Value="true"/>
   <Property Name="ESIMaxCacheSize" Value="1024"/>
   <Property Name="ESIInvalidationMonitor" Value="false"/>
   <Property Name="ESIEnableToPassCookies" Value="false"/>
   <Property Name="PluginInstallRoot" Value="String"/>
<!-- Configuration generated using httpEndpointRef=defaultHttpEndpoint-->
<!-- The default_host contained only aliases for endpoint defaultHttpEndpoint.
    The generated VirtualHostGroup will contain only configured web server ports:
      webserverPort=8080
      webserverSecurePort=443 -->
   <VirtualHostGroup Name="default_host">
      <VirtualHost Name="*:8080"/>
      <VirtualHost Name="*:443"/>
   </VirtualHostGroup>
<ServerCluster CloneSeparatorChange="false" GetDWLMTable="false" IgnoreAffinityRequests="true"
LoadBalance="Round Robin" Name="String_default_node_Cluster" PostBufferSize="0"
PostSizeLimit="-1" RemoveSpecialHeaders="true" RetryInter-val="60">
      <Server CloneID="a33546e6-6399-4fea-a865-2546d224fe44" ConnectTimeout="5"
ExtendedHandshake="false" MaxConnec-tions="-1" Name="default_node_String" ServerIOTimeout="900"
WaitForContinue="false">
         <Transport Hostname="<hostname>" Port="53103" Protocol="http"/>
         <Transport Hostname="<hostname>" Port="53104" Protocol="https">
            <Property Name="keyring" Value="keyring.kdb"/>
            <Property Name="stashfile" Value="keyring.sth"/>
            <Property Name="certLabel" Value="LibertyCert"/>
         </Transport>
      </Server>
      <PrimaryServers>
         <Server Name="default_node_String"/>
      </PrimaryServers>
   </ServerCluster>
   <UriGroup Name="default_host_String_default_node_Cluster_URIs">
      <Uri AffinityCookie="JSESSIONID" AffinityURLIdentifier="jsessionid"
Name="/com.ibm.cics.wlp.defaultapp/*"/>
      <Uri AffinityCookie="JSESSIONID" AffinityURLIdentifier="jsessionid"
Name="/IBMJMXConnectorREST/*"/>
   </UriGroup>
   <Route ServerCluster="String_default_node_Cluster"
UriGroup="default_host_String_default_node_Cluster_URIs" Virtu-alHostGroup="default_host"/>
</Config>
```

Notice the CICS `com.ibm.cics.wlp.defaultapp` is included within the URIGroup. We use this application to test this scenario.

Enable the plug-in in the `httpd.conf` file of the web server using the LoadModule phrase, and specify the location of the `plugin-cfg.xml` file using the `WebspherePluginConfig` phrase. For example, on a Microsoft Windows system, see Example 7-2.

Example 7-2 LoadModule excerpt from httpd.conf

```
LoadModule was_ap22_module
"C:\Users\IBM_ADMIN\IBM\WebSphere\Plugins2\bin\32bits\mod_w
as_ap22_http.dll"
WebSpherePluginConfig "c:\Program Files\IBM\HTTPServer\conf\plugin-cfg.xml"
```

7.4.2 Merging plug-in configuration files

In an environment where you have more than one CICS region each running a Liberty JVM server, you might want to use the same HTTP server to send and receive HTTP and HTTPS requests to each region. To accomplish this, you must merge the web server plug-in configuration files that are used for each Liberty JVM server into a single web server plug-in configuration file.

Use the preceding steps to generate a separate plug-in configuration file for each Liberty profile. Then, combine all of these files into a single configuration. Install the generated merged `plugin-cfg` file on the web server. Typically, you have to enable the plug-in in the `httpd.conf` file of the web server. Use the `LoadModule` phrase to enable the plug-in, and specify the location of the configuration file using the `WebSpherePluginConfig` phrase.

7.5 Test the configuration

As a first step in testing the plug-in, you can test that the web server is active:

1. Go to the web server host name and port number it is listening to in a browser (for example, enter `http://localhost:8080`.
2. In Figure 7-3, you can see the default welcome page for the IBM HTTP Server.

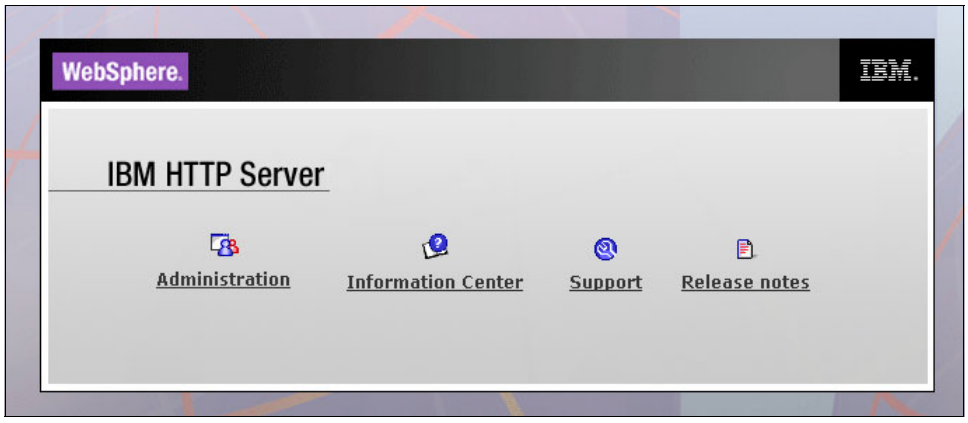


Figure 7-3 IBM HTTP Server welcome page

As the next step, you create a static web page hosted on the Microsoft Windows PC that will be serviced by the IBM HTTP Server.

SSL between the plug-in and the Liberty server might not be necessary, especially if the plug-in is running on the same machine as the application server. However, in this case it is not, and so we removed the following security features from the `server.xml` file:

```
<feature>cicsts:security-1.0</feature>
<feature>ssl-1.0</feature>
```

Now you can test the plug-in configuration. There are three ways to validate the configuration. Choose one of the following options:

- Send an HTTP request to the Liberty JVM server port to validate that the server is up and the application is running:

  `http://<libertyHost>:libertyPort/com.ibm.cics.wlp.defaultapp/`

- Send an HTTP request to the IBM HTTP Server to validate the web server is listening on the post and capable of handling requests:

  `http://<webserverHost>:webserverPort`

- Test the application URL at the IBM HTTP Server to validate the service statement and `plugin-cfg.xml` statements:

  `http://<webserverHost>:webserverPort/com.ibm.cics.wlp.defaultapp/`

In this section, you successfully configured the web server plug-in to forward requests sent to the IBM HTTP Server on Microsoft Windows to the Liberty profile running in CICS on z/OS.

8

Implementing security options

This chapter shows you how to get up and running with security in Liberty Java virtual machine (JVM) server. The first sections are about basic security infrastructure that you need. Later, we cover the details about several options for securing your web applications.

This chapter describes the following topics:

- Start the angel process
- Server.xml security elements
- Set up the SAF unauthenticatedUser user ID
- SAF definitions and permissions
- Web application security
- SAF authorization
- Liberty security-role authorization

For more information about the different security authentication and authorization options that are available in the Liberty server environment on CICS, refer to Chapter 4, “Security options” on page 57.

8.1 Start the angel process

The angel process is a long-running started task required for the Liberty JVM server in CICS Transaction Server (CICS TS) to use System Authorization Facility (SAF)-authorized services. There needs to be one, and only one, angel process started in each logical partition (LPAR). In addition to Liberty JVM server in CICS TS, two other products also use the angel process. These are WebSphere Liberty Profile on z/OS, and zOSMF. If there is already an angel process started for either of those products, Liberty JVM server can use that angel process also.

Note: Only one Liberty JVM server per CICS region can connect to the angel process. Therefore, you can have only one Liberty JVM server per CICS region that integrates with CICS security.

Next, we describe how to start the angel process.

8.1.1 Copy and modify the angel process procedure

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

`/usr/lpp/cicsts53/wlp/templates/zos/procs/bbgzangl.jcl.`

Example 8-1 shows the procedure.

Example 8-1 bbgzangl.jcl

```
//BBGZANGL PROC PARMS='',COLD=N
//*-------------------------------------------------------------------
//  SET ROOT='/u/MSTONE1/wlp'
//*-------------------------------------------------------------------
//* Start the Liberty angel process
//*-------------------------------------------------------------------
//* This proc may be overwritten by fixpacks or iFixes.
//* You must copy to another location before customizing.
//*-------------------------------------------------------------------
//STEP1   EXEC PGM=BPXBATA2,REGION=0M,
//  PARM='PGM &ROOT./lib/native/zos/s390x/bbgzangl COLD=&COLD &PARMS'
//STDOUT    DD SYSOUT=*
//STDERR    DD SYSOUT=*
//* ================================================================ */
//* PROPRIETARY-STATEMENT:                                           */
//* Licensed Material - Property of IBM                              */
//*                                                                  */
//* (C) Copyright IBM Corp. 2011, 2012                               */
//* All Rights Reserved                                              */
//* US Government Users Restricted Rights - Use, duplication or      */
//* disclosure restricted by GSA ADP Schedule Contract with IBM Corp.*/
//* ================================================================ */
```

We now begin the copy and modify procedure:

1. Copy this JCL to a JES procedure library.
2. On the line where it reads, `// SET ROOT='/u/MSTONE1/wlp'`, change `ROOT` to `USSHOME/wlp`.
   `//  SET ROOT='/usr/lpp/cicsts53/wlp'`

8.1.2 Specify the user ID under which the angel process runs

The following command adds an SAF STARTED profile to specify the user ID that the angel process runs under. For example, in RACF, the following commands specify that user ID WLPUSER will be the user ID for the angel process. WLPUSER must already be a valid SAF user ID:

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

8.1.3 Start the angel process and verify that it is up

Example 8-2 shows the console command to start the angel process.

Example 8-2 Start command for the angel process

```
START BBGZANGL
```

If BBGZANGL started successfully, it will be a visible long-running job. If it started and then ended immediately, then likely the problem is that there is already an active angel process in the LPAR. Example 8-3 is a sample of this situation.

Example 8-3 Sample console message

```
23.03.15 STC47564  IEF403I BBGZANGL - STARTED
23.03.15 STC47564  CWWKB0062E A WEBSPHERE FOR Z/OS ANGEL PROCESS IS ALREADY
STARTED
23.03.15 STC47564  CWWKB0063I ACTIVE ANGEL ASID 94 JOBNAME IZUANG1
23.03.15 STC47564  CWWKB0058E WEBSPHERE FOR Z/OS ANGEL PROCESS ENDED ABNORMALLY,
701
   701             REASON=14
```

Note: In the messages, the active angel process is jobname **IZUANG1**. That is the name of the angel process when started with instructions for zOSMF. There is no problem with that. A Liberty JVM server in CICS can connect to, and use, this angel process with no problem.

8.2 Server.xml security elements

There are several necessary `server.xml` elements required for security. This section describes how to add these to the `server.xml` file:

1. Add the cicsts:security-1.0 and the ssl-1.0 features. With autoconfigure=true in the jvmprofile, these features are added automatically if CICS security is active. See Example 8-4.

   Example 8-4 Enabling features in server.xml

   ```
   <featureManager>
   …
   <feature>cicsts:security-1.0</feature>
   <feature>ssl-1.0</feature>
   …
   </featureManager>
   ```

2. Ensure that there is an httpsPort. See Example 8-5.

Example 8-5 Setting HTTP and HTTPS ports in server.xml

```
<httpEndpoint host="*" httpPort="53130" httpsPort="53131"
id="defaultHttpEndpoint"/>
```

3. Specify the following SSL-related elements. These elements are added automatically when using autoconfigure and CICS security is active. This is shown in Example 8-6.

   This causes the Liberty server to dynamically configure an SSL keystore using a Java keystore on zFS.

Example 8-6 Configuring SSL in server.xml

```
<ssl id="defaultSSLConfig" keyStoreRef="defaultKeyStore" sslProtocol="TLS"/>
<keyStore id="defaultKeyStore" password="defaultPassword"/>
```

4. Activate SAF authorization with the element shown in Example 8-7.

   With this element, Liberty JVM server uses SAF EJBROLE profiles to authorize access to web applications, and does not use the application-bnd element. Without this element, it is the opposite. Liberty JVM server does not use SAF EJBROLE profiles to authorize access to web applications. Instead, Liberty JVM server uses the application-bnd element.

Example 8-7 Activate SAF authorization in server.xml

```
<safAuthorization id="saf"/>
```

5. Specify the server name. The angel process will know the Liberty JVM server by this name. This name will be used for the APPL class profile and as the prefix for EJBROLE profiles. The default name is BBGZDFLT. Specify a name that is unique to this Liberty JVM server, such as the CICS applid.

   Example 8-8 shows the Liberty server name defined as LIBSRV01. This name is used in further examples in this section.

Example 8-8 Specify Liberty server name in server.xml

```
<safCredentials profilePrefix="LIBSRV01" />
```

For more information about configuring SSL in Liberty with SAF key rings, see the IBM Knowledge Center at this link:

https://www.ibm.com/support/knowledgecenter/SSEQTP_8.5.5/com.ibm.websphere.wlp.doc/ae/twlp_sec_ssl.html

8.3 Set up the SAF unauthenticatedUser user ID

If you are using an SAF user registry, it is necessary to specify an SAF user ID that represents the unauthenticated state. The name of the unauthenticated user ID is specified on the unauthenticatedUser attribute of the SAFCredentials element in `server.xml`. See Example 8-9.

Example 8-9 Define unauthenticated user in server.xml

```
<safCredentials unauthenticatedUser="WSGUEST"/>
```

This establishes the user ID that is treated as the unauthenticatedUser. The user ID you specify here needs to have been established in the following way.

It is important to define this user ID correctly in your SAF registry. If you are using a RACF SAF user registry, the unauthenticated user (default WSGUEST) needs a unique default group (DFLTGRP) with no other user IDs connected to that group, an OMVS segment, but not a TSO segment, and the options NOPASSWORD, NOOIDCARD, and RESTRICTED. If you have another SAF user registry, instead of RACF, find the user ID options that are provided by that SAF registry that are equivalent to these RACF options.

By running the appropriate commands, you can correctly set up an unauthenticated user in your SAF user registry. An unauthenticated user that is set up incorrectly can cause a security exposure.

Note: An unauthenticated user that is set up incorrectly can cause a security exposure.

The procedure for setting up the SAF unauthenticatedUser user ID follows:

1. Run the **ADDGROUP** command. Use WSGUESTG as the group name, as shown in Example 8-10.

   Example 8-10 Add group command

   ```
   ADDGROUP WSGUESTG SUPGROUP(SYS1)OWNER(SYS1)
               DATA('WAS Unauthenticated User Group')
               OMVS(AUTOGID)
   ```

2. Run the **ADDUSER** command. Use WSGUEST as the user ID name, as shown in Example 8-11.

   Example 8-11 Add user command

   ```
   ADDUSER WSGUEST DFLTGRP(WSGUESTG) OWNER(SYS1)
               OMVS(AUTOUID
               HOME(/u/WSGUEST)
               PROGRAM(/bin/sh))
               NAME('WAS unauth')
               NOPASSWORD NOOIDCARD
               RESTRICTED
   ```

   The **NOPASSWORD** and **NOOIDCARD** options protect this user ID from being revoked by repeated attempts to guess the password.

   The **RESTRICTED** option means that this user ID cannot gain access to protected resources unless it is explicitly permitted to that resource, even is that resource has a general access setting of `UACC(READ)`.

Note: After the unauthenticated user ID (WSGUEST) is defined to the SAF registry, ensure that the user ID is permitted to only the minimum number of SAF resources. If the Liberty server is using SAF APPL resource check to control which users can connect to the Liberty z/OS System Security Access Domain, the unauthenticated user ID must be given access to the APPL profile.

Run the **PERMIT** command:

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

If you receive the RACF authorization failure message ICH408I because the unauthenticated user (WSGUEST) does not have access to a RACF resource, it is almost always incorrect to permit the unauthenticated user ID to the resource profile to resolve the problem. This message usually means that the request is running in an unauthenticated state, when it should be running in an authenticated state. The actual problem is probably a failure to authenticate properly. Whenever it appears necessary to permit the unauthenticated user ID to an SAF resource profile, consider carefully whether that is the correct action to take. Permitting the unauthenticated user ID to any SAF resource profile makes that resource available to everyone, including users who are not authenticated. There are almost no instances where that is required. However, the APPL profile that controls access to the WZSSAD is one exception.

8.4 SAF definitions and permissions

Set up the following SAF definitions and permissions.

8.4.1 Server class authorizations

The CICS region user ID must be authorized to connect to and access necessary services provided by the angel process. The commands in Example 8-12 show how to do this with RACF.

Example 8-12 Server class authorizations

```
RDEFINE SERVER BBG.ANGEL UACC(NONE)
PERMIT BBG.ANGEL CLASS(SERVER) ACCESS(READ) ID(cics_region_userid)

RDEFINE SERVER BBG.AUTHMOD.BBGZSAFM.SAFCRED UACC(NONE)
PERMIT BBG.AUTHMOD.BBGZSAFM.SAFCRED CLASS(SERVER) ACCESS(READ)
ID(cics_region_userid)

RDEFINE SERVER BBG.AUTHMOD.BBGZSAFM UACC(NONE)
PERMIT BBG.AUTHMOD.BBGZSAFM CLASS(SERVER) ACCESS(READ) ID(cics_region_userid)

RDEFINE SERVER BBG.AUTHMOD.BBGZSAFM.PRODMGR UACC(NONE)
PERMIT BBG.AUTHMOD.BBGZSAFM.PRODMGR CLASS(SERVER) ACCESS(READ)
ID(cics_region_userid)

RDEFINE SERVER BBG.SECPFX.LIBSRV01 UACC(NONE)
PERMIT BBG.SECPFX.LIBSRV01 CLASS(SERVER) ACCESS(READ) ID(cics_region_userid)
```

8.4.2 APPL class authorizations

Every user ID needs read access to the Liberty server APPL class profile. The profile name in this example is LIBSRV01 because that is what was specified earlier in the <safCredentials> element. Example 8-13 shows how to define and activate the profile in RACF.

Example 8-13 Define Liberty server APPL class profile

```
RDEFINE APPL LIBSRV01 UACC(NONE)
SETROPTS CLASSACT(APPL)
```

Permit read access for every user ID to this profile. In the following examples, ALLUSERS is a RACF group that all users are connected to. WSGUEST is the unauthenticated user ID. It needs access too. See Example 8-14.

Example 8-14 Permit read access to Liberty server APPL class profile

```
PERMIT LIBSRV01 CLASS(APPL) ACCESS(READ) ID(ALLUSERS)
PERMIT LIBSRV01 CLASS(APPL) ACCESS(READ) ID(WSGUEST)
SETROPTS RACLIST(APPL) REFRESH
```

8.5 Web application security

Controlling access to a web application involves a combination of Java Platform, Enterprise Edition authorization and CICS transaction security. There are two ways to do Java Platform, Enterprise Edition authorization. You can use SAF authorization or Liberty security-role authorization:

- SAF authorization uses SAF profiles in the EJBROLE class. An EJBROLE profile represents a role in a web application. By permitting groups and users access to those profiles, you authorize those groups and users to those roles in the web application.
- Liberty security-role authorization takes place outside of SAF. Security-role information is present in the application element that defines each web application in `server.xml` or in other `.xml` files that are logically part of `server.xml`. This security-role information maps an application role with the groups and users who are authorized to those roles in the web application.

You must set up your Liberty JVM server to do either SAF authorization or Liberty security-role authorization. It cannot do both. You control which one to use by the `<safAuthorization/>` element in `server.xml`. In Example 8-7 on page 146, you were directed to include the `<safAuthorization id="saf"/>` element in `server.xml`.

With that element present in `server.xml`, your Liberty JVM server will use SAF authorization. Any security-role information in the application elements in `server.xml` is ignored.

Similarly, if the `safAuthorization` element is not present in `server.xml`, Liberty JVM server will use Liberty security-role authorization. That is, Liberty JVM server will map roles to users and groups using security-roles information in the application element. Any SAF EJBROLE profiles and permissions are ignored.

The method of Java Platform, Enterprise Edition authorization affects how you set up CICS transaction security. Sometimes, Java Platform, Enterprise Edition authorization totally controls access to the web application and you do not need to use CICS transaction security to control access. Other times, Java Platform, Enterprise Edition authorization is effectively bypassed, and you need to use CICS transaction security to control access.

See 8.6, “SAF authorization” on page 150 for assistance in getting up and running with securing your web applications using SAF authorization.

See 8.7, “Liberty security-role authorization” on page 152 for assistance in getting up and running with securing your web applications using Liberty security-role authorization.

Each section uses several web applications as examples:

com.ibm.cics.server.examples.wlp.hello.war	One of the CICS SDK servlet example applications.
com.ibm.cics.wlp.defaultapp	The Liberty JVM server default application.
GenAppWeb	A sample application that is provided as a SupportPac.
CatalogBrowse	A fictional web application that does not have a security constraint.

8.6 SAF authorization

When using SAF authorization, you need to do the following steps to secure your web applications:

1. Set up EJBROLE profiles and permissions
2. Set up a TCICSTRN profile and permissions for transaction CJSA
3. Define a URIMAP to specify a static user ID for applications with no security constraints

8.6.1 Set up EJBROLE profiles and permissions

For each web application, set up appropriate EJBROLE class definitions and permissions. The examples in this section show how to do this by using RACF commands for sample applications. First, we cover some background information.

Note: Unlike other SAF profiles, changes to EJBROLE profiles do not take effect until the address space is stopped and restarted. Therefore, you need to restart your CICS region if adding or changing EJBROLE profiles in this section.

The EJBROLE profiles defined in this section are based on the following `server.xml` elements:

```
<safCredentials profilePrefix=“LIBSRV01”/>
<safRoleMapper profilePattern="%profilePrefix%.%resource%.%role%"
 toUpperCase=“false“ />
```

In the safRoleMapper profilePattern, %profilePrex%, %resource% and %role% are substitution variables. At run time, they are substituted as follows:

%profilePrefix%	The name specified in safCredentials profilePrefix.
%resource$	The name of the application.
%role%	The name of the role.

So, if you are checking authorization for the role **Broker** in application GenAppWeb, the name of the EJBROLE profile is LIBSRV01.GenAppWeb.Broker.

The specification of the safRoleMapper element is the default. If you specify something different for safRoleMapper, you change the EJBROLE profile. For example, if you specify the following, when checking for the role of Broker in application GenAppWeb, the name of the EJBROLE profile is LIBSRV01.GenAppWeb.allRoles:

```
<safRoleMapper profilePattern="%profilePrefix%.%resource%.allRoles"
 toUpperCase="false" />
```

The following examples describe how to set up EJBROLE class definitions and permissions using RACF commands for sample applications.

com.ibm.cics.server.examples.wlp.hello.war

The security constraint in the deployment descriptor for com.ibm.cics.server.examples.wlp.hello.war declares that role cicsAllAuthenticated is authorized to this application. Rather than permit all users to this role, you can specify UACC(READ) on the definition for the profile. This allows all authenticated users to run this web application. See Example 8-15.

Example 8-15 Define read access to the Hello World application

```
RDEFINE EJBROLE
LIBSRV01.com.ibm.cics.server.examples.wlp.hello.war.cicsAllAuthenticated
UACC(READ)
SETROPTS RACLIST(EJBROLE) REFRESH
```

com.ibm.cics.wlp.defaultapp

The default application, com.ibm.cics.wlp.defaultapp, has a security constraint that authorizes the role of User to the entire application. The following RACF commands map the role of User to a group called USERGRP. This allows all users who are associated with group USERGRP to run the default application. See Example 8-16.

Example 8-16 Define read access to the default application

```
RDEFINE EJBROLE LIBSRV01.com.ibm.cics.wlp.defaultapp.User UACC(NONE)
PERMIT LIBSRV01.com.ibm.cics.wlp.defaultapp.User CLASS(EJBROLE) ACCESS(READ)
ID(USERGRP)
SETROPTS RACLIST(EJBROLE) REFRESH
```

The default application has an additional security constraint that controls the ability to access links to some of the key files, such as `server.xml` and `messages.log`. Access to those files is authorized to the role of Administrator. The following RACF commands authorize an appropriate group, ADMINS, to that role. Any user connected to the ADMINS group is able to use those links in the default application. See Example 8-17.

Example 8-17 Define read access to the default application for the administrator role

```
RDEFINE EJBROLE LIBSRV01.com.ibm.cics.wlp.defaultapp.Administrator UACC(NONE)
PERMIT LIBSRV01.com.ibm.cics.wlp.defaultapp.Administrator CLASS(EJBROLE)
ACCESS(READ) ID(ADMINS)
SETROPTS RACLIST(EJBROLE) REFRESH
```

GenAppWeb

The GenAppWeb deployment descriptor has a security constraint that authorizes the role of Broker to the entire application. The following commands map a RACF group called BROKERS to that role. Any user connected to group BROKERS is authorized to run GenAppWeb. See Example 8-18.

Example 8-18 Define read access to the GenAppWeb application

```
RDEFINE EJBROLE LIBSRV01.GenAppWeb.Broker UACC(NONE)
PERMIT LIBSRV01.GenAppWeb.Broker CLASS(EJBROLE) ACCESS(READ) ID(BROKERS)
SETROPTS RACLIST(EJBROLE) REFRESH
```

Note: At the time of this writing, changes to the EJBROLE profiles do not take effect until the CICS region is stopped and restarted.

BrowseCatalog

There is no security constraint in the BrowseCatalog deployment descriptor. There will be no authentication and no authorization on a request to run this application. Therefore, you do not have to define any EJBROLE profiles.

8.6.2 Set up a TCICSTRN profile and permissions for transaction CJSA

In 8.6.1, “Set up EJBROLE profiles and permissions” on page 150, we used EJBROLE class profiles to control access to web applications. Therefore, you do not need transaction attach security to further limit who can access the web applications. The following RACF commands permit all users access to the default Liberty JVM server transaction, CJSA. See Example 8-19.

Example 8-19 Permit access to Liberty JVM transaction

```
RDEFINE TCICSTRN CJSA UACC(NONE)
PERMIT CJSA CLASS(TCICSTRN) ACCESS(READ) ID(ALLUSERS)
SETROPTS RACLIST(TCICSTRN) REFRESH
```

Define a URIMAP to specify a static user ID for applications with no security constraints.

Having completed all of these steps, and if you have the appropriate authority, you should be able to access the three web applications.

8.7 Liberty security-role authorization

Liberty security-role authorization does not use SAF EJBROLE profiles. Liberty security-role authorization uses role-to-user and role-to-group security information that is specified inside of the application element of each application. Who supplies the application element for each web application, and the security information within it, depends on how you define the application to Liberty JVM server.

One method of defining a web application to Liberty JVM server is to deploy it in a CICS bundle. When you do it that way, CICS builds the application element and puts it in `installedApps.xml` which is included in, and logically part of, `server.xml`. CICS also builds role-to-user and role-to-group security information inside that application element. You have no control over it. See 8.7.1, “Deploying your web application in a CICS bundle” on page 153 for a discussion about securing web applications that are deployed in a CICS bundle.

Another method of defining a web application to Liberty JVM server is for you to build the application element and put it into `server.xml`. When you do it that way, you supply the security information that maps the web application roles to the users and groups authorized to those roles. Section 8.7.2, “Defining a web application in server.xml” on page 156 discusses how to secure web applications that you define that way.

Whichever method you use to get Liberty JVM server to use, the security information in the application element, you must ensure there is no <safAuthorization/> element in `server.xml`.

8.7.1 Deploying your web application in a CICS bundle

When you deploy a web application to Liberty JVM server in CICS in a CICS bundle, CICS automatically builds the application element for the application in a file called `installedApps.xml`, which is included in, and logically part of, `server.xml`. This application element always specifies just one security role, cicsAllAuthenticated, and maps it to a special subject that stands for all authenticated users. There are two main implications of this.

- If your application has a security constraint, the only role that can pass authorization is cicsAllAuthenticated.
- Your application will be authorized to all authenticated users. You cannot use Java Platform, Enterprise Edition authorization to authorize the application to a smaller subset of users.

The effect of this use of cicsAllAuthenticated on bundle-installed web applications is that Java Platform, Enterprise Edition security role authorization is effectively bypassed. This means that you need to use transaction security to control the access to each web application.

To use transaction security to control access to a web application:

1. Define a new transaction for the web application. The attributes should be the same as those of transaction CJSA.
2. Define a URIMAP whose Path attribute specifies the generic context root of the application. On the URIMAP transaction attribute, specify the transaction ID of the transaction defined in step 1 above.
3. Define a SAF TCICSTRN profile for the transaction. Permit read access to that profile for whatever group is appropriate to run the web application.

The following are specific instructions for setting up security for the example transactions when you are not using SAF authorization, and you are installing and deploying the applications in a CICS bundle.

com.ibm.cics.server.examples.wlp.hello.war

The deployment descriptor for this application authorizes a role of cicsAllAuthenticated to the entire application. That matches perfectly with the security role built automatically when you deploy a web application as a CICS bundle. See Example 8-20.

Example 8-20 Authorizing roles to application

```
<auth-constraint>
   <role-name>cicsAllAuthenticated</role-name>
</auth-constraint>
```

The next step is to define a transaction ID and URIMAP for this web application.

Define and install a transaction definition

Define and install a transaction definition with the following attributes:

Description	Copy of transaction CJSA for servlet hello example
Name	HIEX
Program Name	DFHSJTHP
Task Data Location	Any
System Purge	Yes
CmdSec	Yes
ResSec	Yes

Define and install a URIMAP definition

Define and install a URIMAP definition with the following attributes:

Description	URIMAP for servlet hello example
Name	HELLOSRV
Host	*
Path	/com.ibm.cics.server.examples.wlp.hello.war/*
Transaction	HIEX
Usage	JVMSERVER

TCICSTRN definition and permission

The following RACF commands show how to give access to group ALLUSERS to the new transaction. See Example 8-21.

Example 8-21 Give transaction access to the ALLUSERS group

```
RDEFINE TCICSTRN HIEX UACC(NONE)
PERMIT HIEX CLASS(TCICSTRN) ACCESS(READ) ID(ALLUSERS)
SETROPTS RACLIST(TCICSTRN) REFRESH
```

GenAppWeb

The deployment descriptor for this application authorizes a role of Broker to the entire application. Since, when deployed as a CICS bundle, the application element will not specify a security-role for Broker, you are not able to run GenAppWeb. The solution is to alter the deployment descriptor to add the cicsAllAuthenticated role.

Alter the deployment descriptor

Follow these steps to alter the deployment descriptor:

1. In the development environment, expand the GenAppWeb project, as shown in Figure 8-1.

Figure 8-1 View of the GenAppWeb project in the Java EE perspective

2. Double-click the highlighted `3.0 GenAppWeb` directory. This opens the `web.xml` file in the deployment descriptor editor.
3. Add a security role for cicsAllAuthenticated, as shown in Figure 8-2.

Figure 8-2 Add security role in the deployment descriptor

4. Highlight **Security Constraint**, and then add cicsAllAuthenticated as an extra authorization constraint, as shown in Figure 8-3.

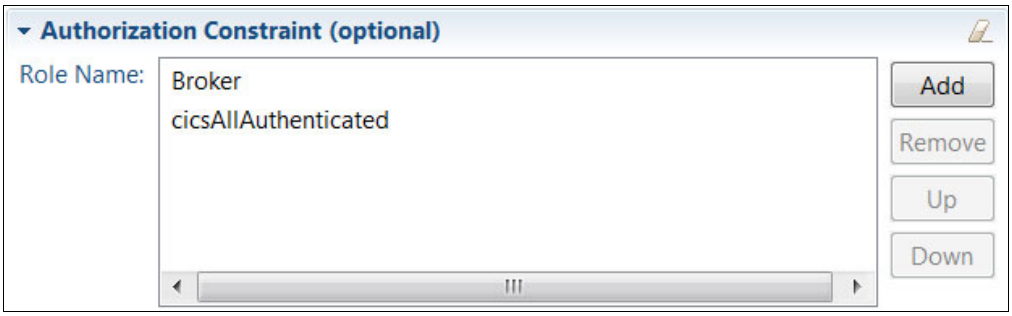


Figure 8-3 Add additional authorization constraints

5. The resulting `web.xml` source looks like that shown in Example 8-22.

Example 8-22 Sample of updated web.xml

```
<security-role>
   <role-name>Broker</role-name>
</security-role>
<security-constraint>
   <web-resource-collection>
      <web-resource-name>full app</web-resource-name>
      <url-pattern>/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
      <role-name>Broker</role-name>
      <role-name>cicsAllAuthenticated</role-name>
   </auth-constraint>
   <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
   </user-data-constraint>
</security-constraint>
<security-role>
```

```
    <role-name>cicsAllAuthenticated</role-name>
</security-role>
```

When you have more than one role in an auth-constraint, you only need to belong to one role in order to satisfy the constraint. So with these changes, all authenticated users are able to run GenAppWeb. But that is not what you want. You only want Brokers to be able to run GenAppWeb. To enforce that, you need to use a transaction, URIMAP, and TCICSTRN permissions.

Define and install a transaction definition

Define and install a transaction definition with the following attributes:

```
Description        - Copy of transaction CJSA for GenAppWeb
Name               - GENA
Program Name       – DFHSJTHP
Task Data Location – Any
System Purge       - Yes
CmdSec             - Yes
ResSec             - Yes
```

Define and install a URIMAP definition

Define and install a URIMAP definition with the following attributes:

```
Description        - URIMAP for GenAppWeb
Name               - GENAPPWB
Host               – *
Path               - /GenAppWeb/*
Transaction        - GENA
Usage              - JVMSERVER
```

TCICSTRN definition and permission

The following RACF commands show how to give access to group BROKERS to the new transaction. See Example 8-23.

Example 8-23 Give transaction access to the BROKERS group

```
RDEFINE TCICSTRN GENA UACC(NONE)
PERMIT GENA CLASS(TCICSTRN) ACCESS(READ) ID(ALLUSERS)
SETROPTS RACLIST(TCICSTRN) REFRESH
```

8.7.2 Defining a web application in server.xml

When you define the application in `server.xml`, and you are not using SAF authorization, you can fully control access to your web application. Nevertheless, you still need to be able to pass transaction security. Here, for each of the sample applications, we provide examples of how to do that.

com.ibm.cics.server.examples.wlp.hello.war

The deployment descriptor for this application authorizes a role of cicsAllAuthenticated to the entire application so you need to make the appropriate updates to the server. This example application element in `server.xml` maps the cicsAllAuthenticated role to RACF group ALLUSERS. Any authenticated user that is connected to group ALLUSERS is authorized to the application. See Example 8-24 on page 157.

Example 8-24 Mapping groups to security role for the hello.war application in server.xml

```
<application id="com.ibm.cics.server.examples.wlp.hello.war"
 Location="/u/reds13/cicsts53/com.ibm.cics.server.examples.wlp.hello.war.war"
 name="com.ibm.cics.server.examples.wlp.hello.war" type="war">
   <application-bnd>
      <security-role name="cicsAllAuthenticated">
         <group name="ALLUSERS"/>
      </security-role>
   </application-bnd>
</application>
```

Note: RACF Groups and user IDs specified in a security-role must have an OMVS segment and must have a unique GID or UID.

TCICSTRN definition and permission

Ensure that the group ALLUSERS has access to the TCICSTRN profile for CJSA. Here are the RACF commands to do that. See Example 8-25.

Example 8-25 Define and permit ALLUSERS group access to the TCICSTRN profile

```
RDEFINE TCICSTRN CJSA UACC(NONE)
PERMIT GENA CLASS(TCICSTRN) ACCESS(READ) ID(ALLUSERS)
SETROPTS RACLIST(TCICSTRN) REFRESH
```

GenAppWeb

The GenAppWeb deployment descriptor has a security constraint that authorizes the role `Broker`. The following example application element maps the Broker role to RACF group BROKERS. Any authenticated user that is connected to group BROKERS is authorized to the application. See Example 8-26.

Example 8-26 Map Broker role to BROKERS group in the server.xml

```
<application id="GenAppWeb"
location="/u/reds13/cicsts53/GenAppWebBundle_1.0.0/GenAppWeb.war" name="GenAppWeb"
type="war">
   <application-bnd>
      <security-role name="Broker">
         <group name="BROKERS"/>
      </security-role>
   </application-bnd>
</application>
```

Note: RACF Groups and user IDs specified in a security-role must have an OMVS segment and must have a unique GID or UID.

TCICSTRN definition and permission

Make sure group BROKERS has access to the TCICSTRN profile for CJSA. Here are the RACF commands to do that. See Example 8-27 on page 158.

Example 8-27 Define and permit BROKERS group access to the TCICSTRN profile

```
RDEFINE TCICSTRN CJSA UACC(NONE)
PERMIT GENA CLASS(TCICSTRN) ACCESS(READ) ID(BROKERS)
SETROPTS RACLIST(TCICSTRN) REFRESH
```

BrowseCatalog

The deployment descriptor for `BrowseCatalog` does not specify any security constraints. Therefore, a request to run `BrowseCatalog` will not go through authentication or authorization. So there is no need for security-role information in the application element in `server.xml`. That application element might look like that in Example 8-28.

Example 8-28 BrowseCatalog application without security-role information in server.xml

```
<application id="BrowseCatalog"
location="/u/reds13/cicsts53/BrowseCatalog.war"
name="BrowseCatalog" type="war">
</application>
```

Because there is no security constraint, a request to run this application needs to run the CJSA transaction using a static user ID.

URIMAP definition

Define and install a URIMAP definition with the following attributes:

```
Description        - URIMAP for BrowseCatalog
Name               - BROWSCAT
Host               – *
Path               - /BrowseCatalog/*
Usage              - JVMSERVER
Userid             - PUBLIC01
```

TCICSTRN definition and permission

Ensure that the user ID `PUBLIC01` has access to the TCICSTRN profile for transaction CJSA. Example 8-29 shows the RACF commands to do that.

Example 8-29 Define and permit PUBLIC01 user ID access to the TCICSTRN profile

```
RDEFINE TCICSTRN CJSA UACC(NONE)
PERMIT CJSA CLASS(TCICSTRN) ACCESS(READ) ID(PUBLIC01)
SETROPTS RACLIST(TCICSTRN) REFRESH
```

Part 3

Scenarios

In this part, we describe scenarios.

Part 3 contains the following chapters:

9

Porting a web application

In this chapter, we show how to take a presentation logic web application from a third-party Java EE web application server, and migrate it into the Liberty profile JVM server in CICS with minimal changes. This is sometimes referred to as *lift* and *shift* of an application to Liberty in CICS.

This chapter describes the following topics:

- Lifting or shifting an application to Liberty
- The migration process
- Configuring the Liberty environment
- Package and deploy the application
- Package the existing application WAR file and deploy
- Complete deployment and test the application

9.1 Lifting or shifting an application to Liberty

The application can be one that already uses CICS services remotely, using for example:

- JCA connectors
- CICS Transaction Gateway (CICS TG) ECI requests
- Web services (SOAP or JSON)
- HTTP or socket-based connectors

The application can also use other non CICS services such as:

- Web services (XML-based or JSON)
- JDBC database connections
- JMS or IBM MQ messaging

Alternatively, the application can be one that does not use CICS services, which are being migrated for platform consolidation, or perhaps with the intention of extending it to exploit integration with CICS applications in the future.

The following diagram illustrates the scenario that we walk through in this chapter. It is for an application using servlets and JavaServer Pages (JSP) and an external call interface (ECI) Java EE Connector Architecture (JCA) call to a CICS program using the CICS Transaction Gateway.

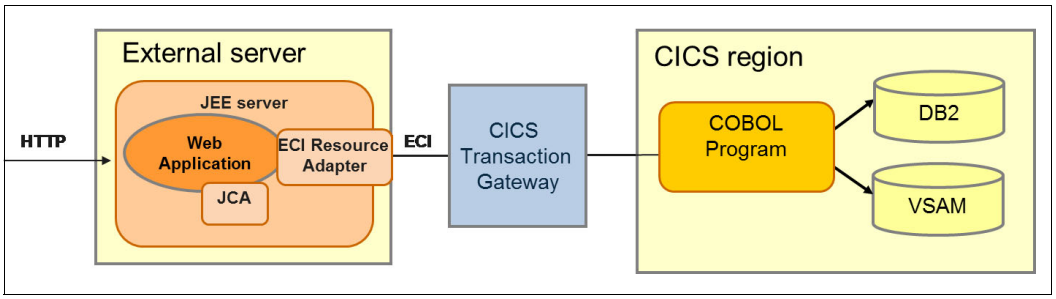


Figure 9-1 Application running on an external server before migration

When this application is migrated into a Liberty JVM server, the application can remain unchanged and the function of the CICS Transaction Gateway can be replaced by the local JCA ECI feature running within CICS.

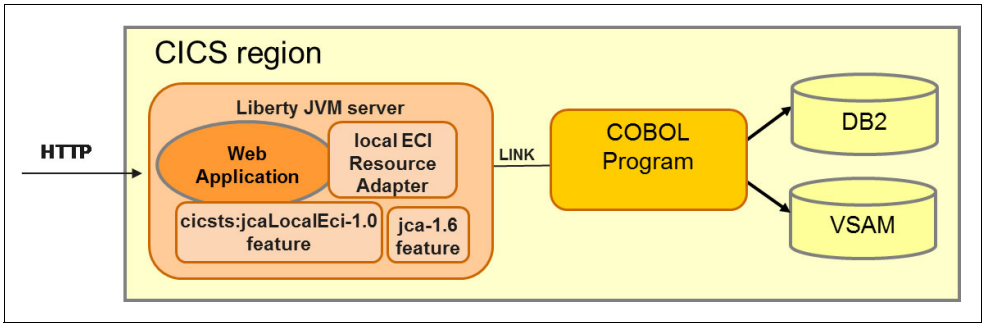


Figure 9-2 Application running in Liberty in CICS after migration

9.2 The migration process

There are potentially several stages to the application migration process, some of which are optional, depending on the level of feasibility validation, environment configuration, and deployment repackaging required. Following are the three main stages:

1. Validate the eligibility of the application for migration to Liberty in CICS. If issues are discovered, remediate the migration inhibitors. Toolkit support is available to assist with this validation.
2. Configure the Liberty in CICS environment for the application. Ensure that the features and resources required by the application are available, in addition to any other application-specific requirements. Toolkit support is available to assist with this configuration.
3. Deploy and test the application in Liberty in CICS.

This Redbooks publication uses an example web application (GenAppWeb), which is a front end to the CICS GenApp sample COBOL application. In the following sections, we walk through the process that we used to migrate the sample GenApp web application to run in Liberty in CICS. This is a fairly simple case of a stand-alone web application with no external dependencies, and in the real world, the process might be more involved, but you can still follow the same steps.

Figure 9-3 shows the initial page for the GenAppWeb application running on a local WebSphere Application Server, and connecting to CICS using a local CICS Transaction Gateway instance.

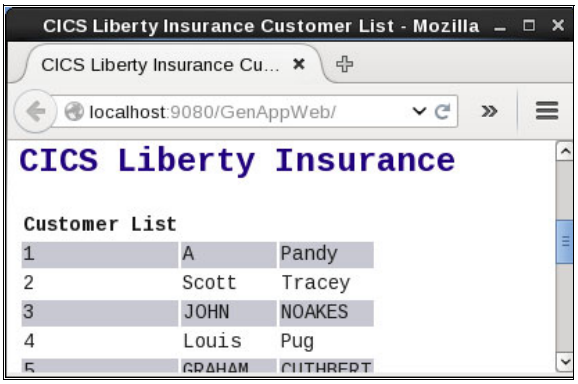


Figure 9-3 GenAppWeb running on a local application server

In the rest of this chapter, we assume that the GenAppWeb web archive (WAR) file containing just the complied Java classes is available on the local file system.

9.2.1 Validating the application for migration

A Java EE application is eligible for migration to Liberty in CICS if all of its environment requirements can be satisfied. That is, the Java EE features required by the application are supported by Liberty in CICS, and configured in `server.xml`. The significant items to consider in this process are:

- JDBC data sources

  These are configured in `server.xml`, and you need to consider whether the associated database should remain in its current environment or should also be migrated to z/OS.

- JCA connection factories

  These are configured in the `server.xml` file. If you are using the ECI resource adapter, it is simple to migrate using the local ECI JCA resource adapter supplied with CICS Liberty. If you are using another JCA resource adapter, such as WebSphere Optimized Local Adapter (WOLA), you need to determine if the resource adapter is portable to the CICS Liberty environment.

- JMS resources

  JMS applications use connection factories for outbound calls and activation specifications for inbound calls, both of which need to be configured in the `server.xml` file. IBM MQ JMS connections are not currently supported in CICS Liberty.

- Java system properties

  These are configured in the JVM profile for the Liberty JVM server, and are used by many applications to define specific properties.

- Vendor-specific application programming interfaces (APIs)

  These must be removed from the application to run within Liberty, and so manual rework is required.

- Unsupported components from Java Platform, Enterprise Edition

  These must be removed from the application, and you need to find an alternative for these components.

9.2.2 Using the dropins for validation

This section describes how to use dropins for validation:

1. As a first step in validating an application, a possible approach is simply to copy the WAR file into the `dropins` directory of a suitably configured Liberty JVM server. To use the `dropins` directory, add the following message to the `server.xml` file:

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

   The application might run without any further changes, although this is unlikely. However, any error messages that are produced can be useful in understanding further migration actions that might be necessary. When we copied our `GenAppWeb.war` file to the `dropins` directory, the following messages were produced in the `messages.log` file for the server, indicating that the web application could be loaded by Liberty:

   ```
   CWWKZ0018I: Starting application GenAppWeb.
   SRVE0169I: Loading Web Module: GenAppWeb.
   SRVE0250I: Web Module GenAppWeb has been bound to default_host.
   CWWKT0016I: Web application available (default_host):
   http://<hostname>:65432/GenAppWeb/
   CWWKZ0001I: Application GenAppWeb started in 3.221 seconds.
   ```

2. The next step is to access the application from a browser by using the URL from the message CWWKT0016I produced:

   `http://<hostname>:65432/GenAppWeb/`

   The `index.jsp` page is displayed, showing that the application is accessible at that URL, but the customer list does not appear. The `messages.log` file reports an `Exception: ja-va.lang.ClassNotFoundException: javax.resource.ResourceException`. This indicates that the JCA feature is not available in the Liberty environment.

One approach to the migration is to correct this by adding the jca-1.6 feature to the server configuration, then disable, and re-enable the JVM server, and retry. The next error would then give a hint as to further migration action. However, we preferred to use a more targeted method by using the WebSphere Application Migration Toolkit.

9.2.3 Application Migration Toolkit

WebSphere Liberty provides a suite of tools to assist in migrating web applications from WebSphere Application Server full profile or other Java EE servers to the Liberty profile. This section shows how to use these tools to help migrate the example GenApp front-end web application to Liberty in CICS.

Migration report using application binaries technical preview

1. The first step is to use the application binaries technology preview to scan the application WAR file. Run the following command, assuming that the WAR file has been copied into the same directory as the scanner Java archive (JAR) file:

```
java -jar binaryAppScanner.jar GenAppWeb.war
The tool will produce command output similar to:
Scanning files.....................
The report was saved to the following file:
/home/jlawrence/LibertyRedbook/MigrationToolkit/wamt/TechnologyReport.html
The report was saved to the following file:
/home/jlawrence/LibertyRedbook/MigrationToolkit/wamt/TechnologyReport.html
```

2. When finished, the tool writes an HTML report file named `TechnologyReport.html` to the current directory and displays the file in your default browser, as shown below in Figure 9-4. Technologies used by the application are shown in the left column, and WebSphere product editions each have their own column. For Liberty in CICS, the most appropriate column is the one with the heading *WebSphere Application Server for z/OS 8.5.5 Liberty profile*.

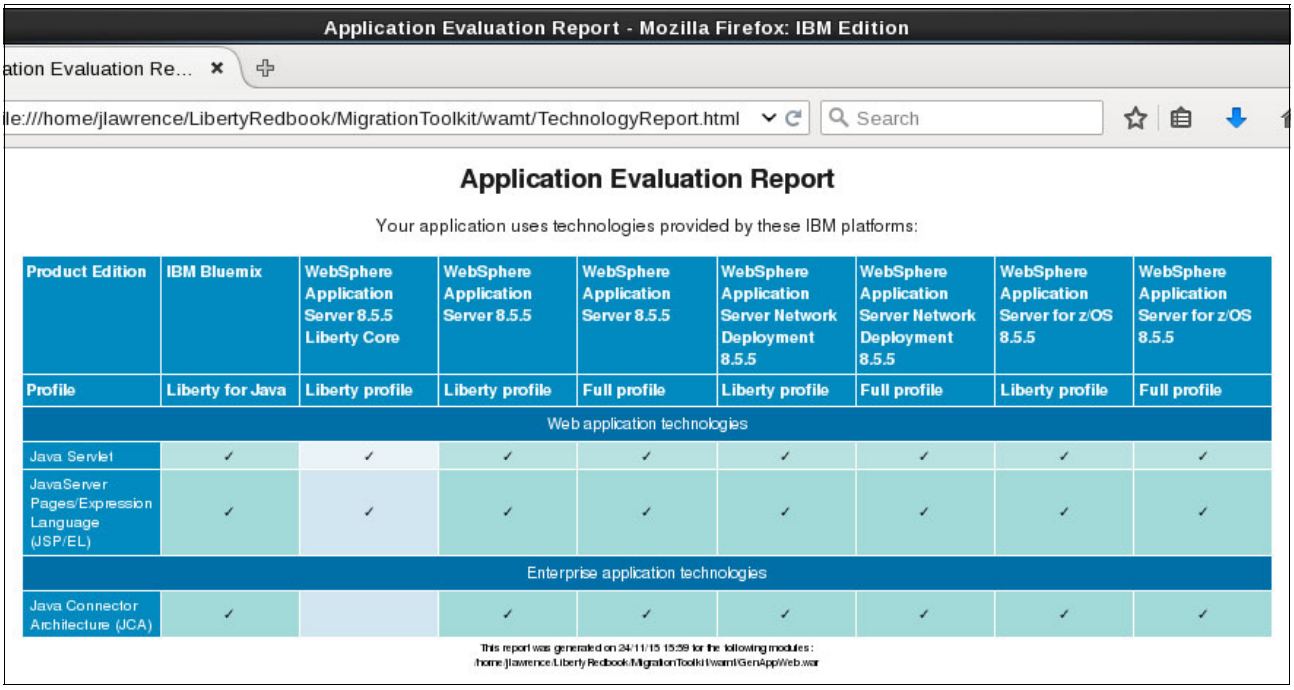


Figure 9-4 Output from the Technology Report tool

3. For `GenAppWeb.war`, the tool has determined that the application uses servlets, JSP technology, and JCA, and that all of these technologies are supported in WebSphere Application Server for z/OS 8.5.5 Liberty profile. The corresponding features must be configured in Liberty for the application to run.

Detailed migration report using the application analysis tool

The next level of analysis uses the same tool with the option to produce the more detailed form of migration report. Run the following command in the same directory: It is necessary to specify the full details (as shown) for the target server, because the defaults are Java 8 and Java Platform, Enterprise Edition 7:

```
java -jar binaryAppScanner.jar GenAppWeb.war --analyzeMigrationDetails
--sourceAppServer=fullProfile855 --sourceJava=ibm6
--targetAppServer=libertyProfile --targetJava=ibm7 –targetJavaEE=ee6
```

The following output is displayed at the command line. The `AnalysisReport.html` file is written to disk and opened in your default browser:

```
Scanning files.....................
The report was saved to the following file:
/home/jlawrence/LibertyRedbook/MigrationToolkit/wamt/AnalysisReport.html
```

In this case, a detailed report is saved to the `AnalysisReport.html` and displayed in the default browser as illustrated in Figure 9-5.

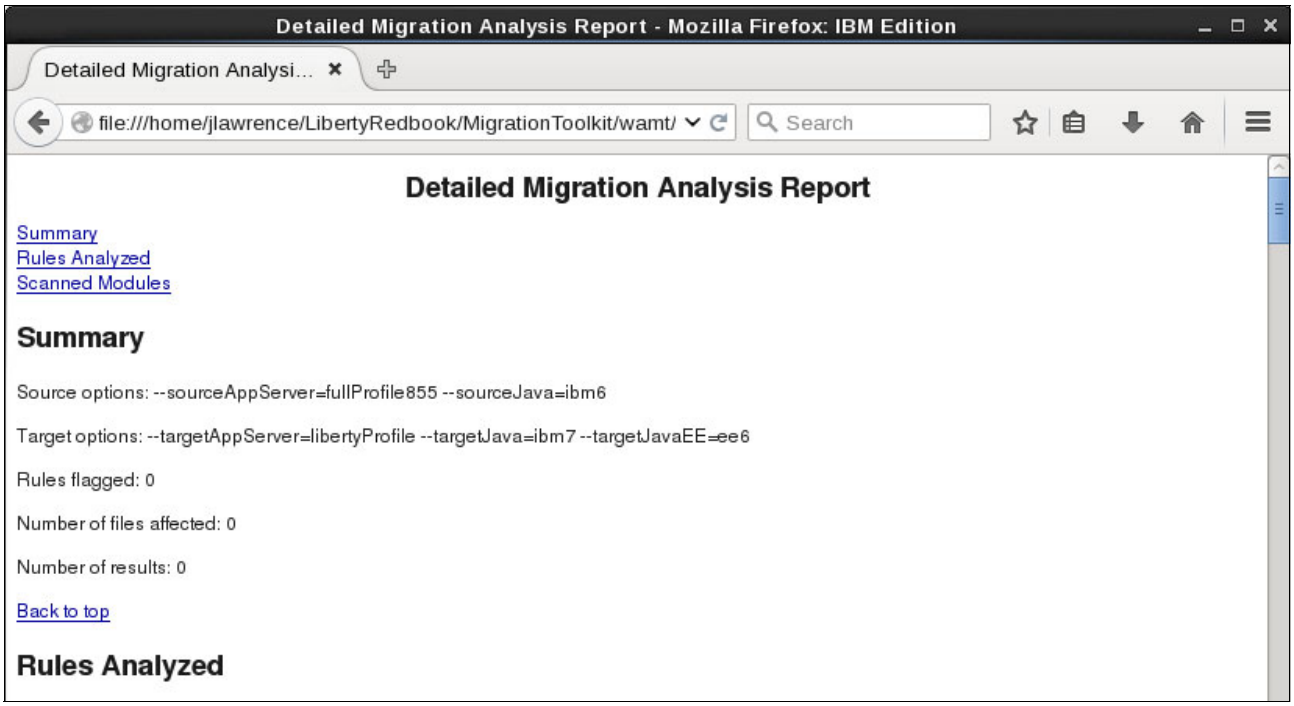


Figure 9-5 Output from the Application Analysis tool

The report shows that no rules were flagged for the migration to Liberty. That means that in principle, there is no obstacle to the migration. The application is suitable provided that the target server environment is configured appropriately.

For more information, see the following resources:

- Move applications to Liberty by using the Migration Toolkit, a blog about the Application Migration Toolkit, and links to download the tool:

  https://developer.ibm.com/wasdev/docs/move-applications-liberty-using-migration-toolkit

- Migration Toolkit for Application Binaries to download the command-line tool:

  https://developer.ibm.com/wasdev/downloads/#asset/tools-Migration_Toolkit_for_Application_Binaries

- WebSphere Application Server Migration Toolkit to download the Eclipse-based tools:

  https://developer.ibm.com/wasdev/downloads/#asset/tools-WebSphere_Application_Server_Migration_Toolkit

9.2.4 Other migration considerations

There might be migration considerations that will not be detected by the tooling, but that will need to be addressed before an application can be migrated. For example, an application currently connecting to a local database using JDBC might need the database to be migrated so that the application can continue to access the data after migration. Such additional considerations are not detected by the toolkit and need to be determined by checking the existing application design.

9.3 Configuring the Liberty environment

For a migrated application to run in Liberty in CICS, the Liberty server must be configured to provide an equivalent environment for the application as the original server environment. The same facilities, APIs, and data must be accessible to the application.

9.3.1 Migrating Java Platform, Enterprise Edition resources

The Java Platform, Enterprise Edition resources required by the application must be added to `server.xml` for the CICS Liberty JVM server. You can see from the Application Migration Technology report that servlet, JSP, and JCA are required by the GenAppWeb application.

1. Add the following feature elements into the featureManager section of `server.xml`:

   ```
   <feature>jsp-2.2</feature>
   <feature>jca-1.6</feature>
   <feature>jndi-1.0</feature>
   ```

   The jsp-2.2 feature automatically includes servlet capability, so this does not need to be explicitly included.

2. Next, you need to configure instance elements for some of the features. This applies to JDBC, JCA, and JMS. Information on the required configuration can be extracted from the configuration of the original server environment.

For a WebSphere full profile server, this can be done by navigating the Admin console and copying relevant configuration parameters for the JCA resource adapter connection factories, and so on. However, this is inconvenient. An alternative is to extract the entire WebSphere configuration as a properties file and browse the properties to determine the parameters. The command to extract the configuration properties must be run while the WebSphere server is running:

```
wsadmin.(sh/bat) -lang jython -c
"AdminTask.extractConfigProperties(['-propertiesFileName websphere.props'])"
```

The `wsadmin` command must be run as a Windows Administrator, and needs to be run from the `<Profile_Home>\bin directory`. On Linux, the command is suitable from any location, as follows:

```
sudo '/opt/IBM/WebSphere/AppServer/bin/wsadmin.sh' -lang jython -c
"Ad-minTask.extractConfigProperties(['-propertiesFileName websphere.props'])"
```

The following output is produced and the `websphere.props` file is written to disk:

```
WASX7209I: Connected to process "server1" on node localhostNode01 using SOAP
connector;  The type of process is: UnManagedProcess
```

3. Next, we examine the contents of the `websphere.props` file to find the configuration parameters for the JCA connection factories required by the GenAppWeb application. By browsing the file, we locate the following entry:

```
#
#Properties
#
name=CICS ECI Connection Factory #required
.. .. ..
J2EEResourceProvider=ECIResourceAdapter #ObjectName(J2CResourceAdapter)
jndiName="eis/ECI" #required
.. .. ..
```

This tells us the Java Naming and Directory Interface (JNDI) lookup location for the connection factory. Another subsection with the title “J2CConnectionFactory Properties”, contains the connection factory-specific attributes. In this case, for a CICS TG ECI Connection Factory. This subsection reads as follows:

```
#
# SubSection 1.0.2.0.0.2 # J2CConnectionFactory Properties
#
.. .. ..
#
#Properties
#
keyRingClass=null
tranName=null
password=null
cipherSuites=null
portNumber=2006 #String
xaSupport=off #String
clientSecurity=null
serverName=winmvsh1 #String
ipicSendSessions=100 #integer
userName=null
tPNName=null
serverSecurity=null
socketConnectTimeout=0 #String
```

```
applidQualifier=null
connectionURL=tcp://localhost #String
requestExits=null
keyRingPassword=null
traceLevel=1 #integer
applid=null
```

These attributes need to be configured on the connection factory instance specification in the Liberty `server.xml` file, although with modification because `tcp://localhost` refers to the local host for the original server.

Continuing to search in `websphere.props` for "J2CConnectionFactory Properties" finds other hits, one for the `builtin_rra resource` adapter, and there might be other third-party resource adapters configured in the system. You need to determine which ones are being used by the application. A good way to do this is to match the JNDI lookup locations for the connection factories with those used by the application.

The manual process just described is one option. WebSphere Liberty provides a Beta tool, called the *WebSphere Configuration Migration Tool*, which is intended to read the configuration files for other servers and produce appropriate configuration fragments to include in the `server.xml` of a Liberty server.

For more information, see the following tools:

- WebSphere Configuration Migration Tool

  https://developer.ibm.com/wasdev/downloads/#asset/tools-WebSphere_Configuration_Migration_Tool

- Making the move to the Liberty profile, Part 2: Migrating Java EE resources with the Configuration Migration Toolkit

  http://www.ibm.com/developerworks/websphere/library/techarticles/1404_vines2/1404_vines2.html

9.3.2 External dependency migration

In the original environment, the J2C connection factory used by GenAppWeb is configured to connect to a CICS Transaction Gateway instance, in this case on the system local to the WebSphere Application Server. This continues to work after migration to Liberty in CICS, but would make little sense because JCA requests would be transmitted to the external Gateway, only to be forwarded back to CICS.

Therefore, we changed the parameters used for the J2C connection factory to use the new local ECI adapter provided with CICS TS V5.3. The following `server.xml` changes are required for this:

1. Add the JCA local ECI feature, which manages the local ECI resource adapter, to the featureManager list in `server.xml`:

   ```
   <feature>cicsts:jcaLocalEci-1.0</feature>
   ```

2. Define a connection factory instance for the JCA local ECI adapter, published at the correct JNDI location, such as:

   ```
   <connectionFactory id="eciLocal" jndiName="eis/ECI">
      <properties.com.ibm.cics.wlp.jca.local.eci/>
   </connectionFactory>
   ```

With these changes, JCA ECI requests from the application are processed within the CICS region where the Liberty JVM is running. CICS dynamic routing of the LINK request can then be used to forward these to other CICS regions if required.

Note: The CICS local ECI resource adapter provides a default connection factory that can be used without an instance configuration. This is always available when the cicsts:jcaLocalEci-1.0 feature is enabled, and is configured at the JNDI lookup location `eis/defaultCICSConnectionFactory`. Therefore, if the application is changed or can be configured to look up its ECI Connection Factory at this location, there is no need for a connection factory to be configured in `server.xml`.

For more information, see the following sites:

- CA local ECI support

  http://www.ibm.com/support/knowledgecenter/SSGMCP_5.3.0/com.ibm.cics.ts.java.doc/topics/dfhpj_jcadoc.html

- Porting JCA ECI applications into a CICS Liberty JVM server

  https://developer.ibm.com/cics/2015/07/23/porting-jca-eci-applications-into-a-cics-liberty-jvm-server

9.3.3 Other configuration parameters

For a complex application or environment, it is possible that using the toolkit and general guidelines above might not be sufficient to guarantee that the application will be successfully migrated at the first attempt. Some additional configuration and migration actions might be required.

An example of such additional configuration is Java system properties. Some applications require specific system properties to be configured, and so these also need to be set in the Liberty JVM server environment. For Liberty in CICS, system properties are specified in the JVM profile for the Liberty JVM server with -D declarations.

If you are migrating from WebSphere full profile, system properties are reported in the configuration properties exported by the wsadmin AdminTask.extractConfigProperties tool. For example:

```
#
# SubSection 1.0.12.0.2 # Environment properties
#
.. .. ..
*#
#Properties
#
com.ibm.cics.genapp.jca.jndiname=eis/ECI
```

The GenAppWeb application uses the system property `com.ibm.cics.genapp.jca.jndiname` to determine the JNDI lookup location for the JCA connection factory, so this had to be configured in the JVM profile.

We added the following entry into the JVM profile for the Liberty JVM server:

```
-Dcom.ibm.cics.genapp.jca.jndiname=eis/ECI
```

This configures the JNDI lookup location for the JCA connection factory used by the GenAppWeb application. In fact, because of the default connection factory provided by the CICS Local ECI adapter, we can change this to the following property and remove the connection factory configuration from `server.xml` altogether:

```
-Dcom.ibm.cics.genapp.jca.jndiname=eis/defaultCICSConnectionFactory
```

9.4 Package and deploy the application

Having completed any necessary application remediation and configured the Liberty in CICS server as required, we are ready to deploy and test the migrated application in its new environment.

There are several possible deployment options for a Liberty application in CICS. Here, we choose the option to deploy as a CICS bundle, using CICS Explorer. There are several options to package an existing application as a CICS bundle. Two options are highlighted here:

- Option 1: Import the application including its source code into your development environment, then rebuild and package. This option is similar to the process described for the Hello World starter application in Chapter 5, “Developing and deploying applications” on page 69.
- Option 2: Package the existing application WAR file and deploy as-is. This is slightly different from the usual process because there is no source code in the WAR.

Option 1 is covered in detail in Chapter 5, “Developing and deploying applications” on page 69, so here we show how to package an existing binary WAR file for deployment by using CICS Explorer.

9.5 Package the existing application WAR file and deploy

To package an existing binary WAR file for deployment using CICS Explorer, we use the example of the GenAppWeb application. Following are the main stages:

1. Import the binary WAR file as a web project into your development environment.
2. Create a CICS Bundle project, and include the imported web project as a WAR bundle part.
3. Export the CICS bundle to the z/FS file system, then define and install a CICS bundle definition.

Complete the following steps:

1. In CICS Explorer, click **File** → **Import**.
2. In the **Import Select** dialog, select **WAR file** from the web section, as shown in Figure 9-6, and click **Next**.

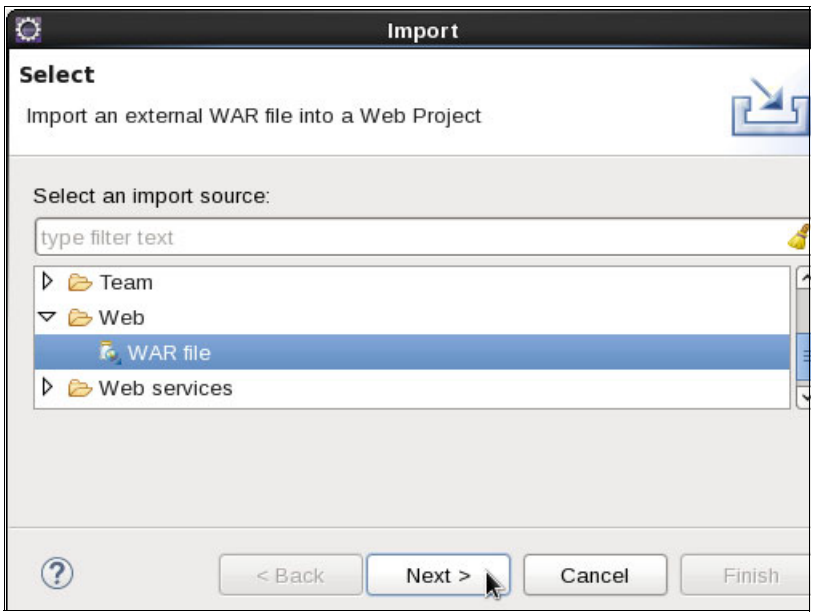


Figure 9-6 Select the option to import a WAR file

3. Browse for your file and click **Next**.
4. Accept the default web project name, which is the same as the WAR file name. And clear the “Add project to an EAR” check box.
5. Click **Finish** to complete the import, as shown in Figure 9-7.

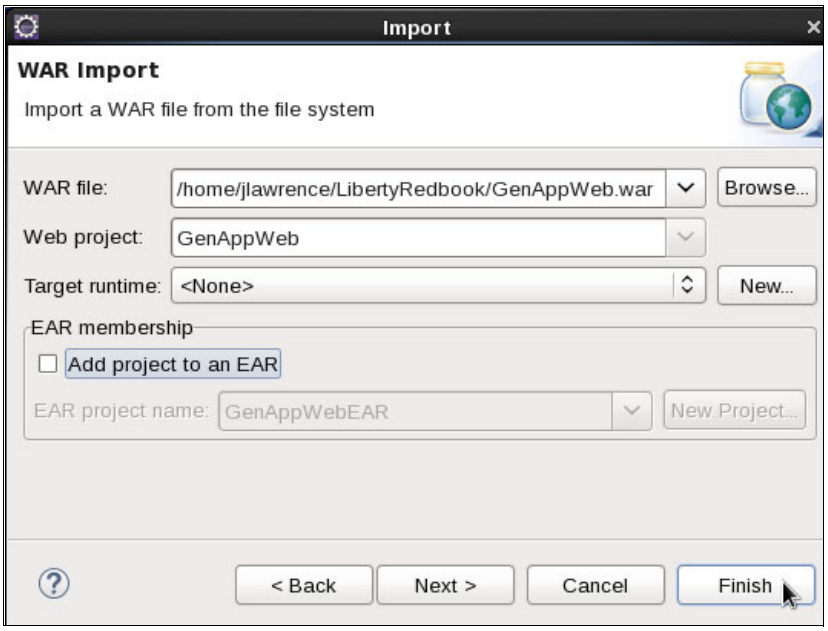


Figure 9-7 Importing a WAR file from the local file system

The `.war` file is imported and is displayed in your Eclipse workspace as a dynamic web project, including any Web App Library JAR files, in this case, `GenAppBli.jar`. However, there is no available source for the project.

> **Note:** If you click **Next** *instead of Finish* at this point, you will have the option to import web library JAR files as separate projects. In our case, because this is a binary-only JAR file, there is no point in doing so because there is no source available in the .war or included `.jar` files.

If errors are displayed in the new project after importing the WAR file, this can be because the Liberty JVM server libraries are not in the build path for the project. These libraries provide both the web APIs for the Liberty features, and also the JCICS APIs, if these are needed. To add these libraries, right-click the web project, and select **Build path** → **Configure Build Path** → **Add Library** → **Liberty JVM server libraries**. Ensure that all errors are corrected before continuing.

6. In CICS Explorer, create a new CICS Bundle project by selecting **File** → **New** → **Other**, and then select the **CICS Bundle Project** from the CICS Resources section of the New Project wizard.
7. Click **Next**.

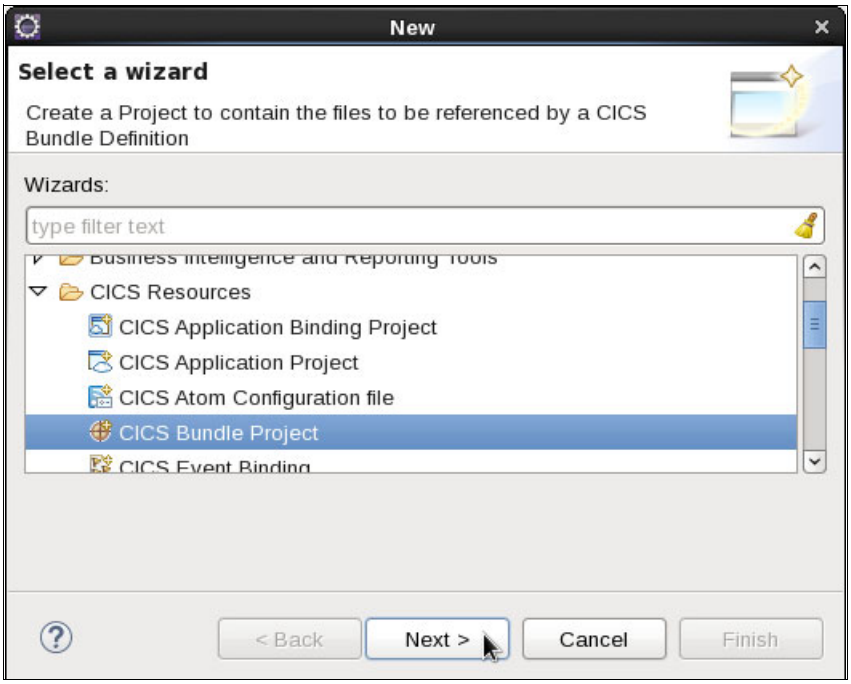


Figure 9-8 Create a new CICS Bundle project

8. Specify your new bundle project name, **GenAppBundle**, and click **Finish**, as shown in Figure 9-9. The new, empty bundle project is created in your workspace.

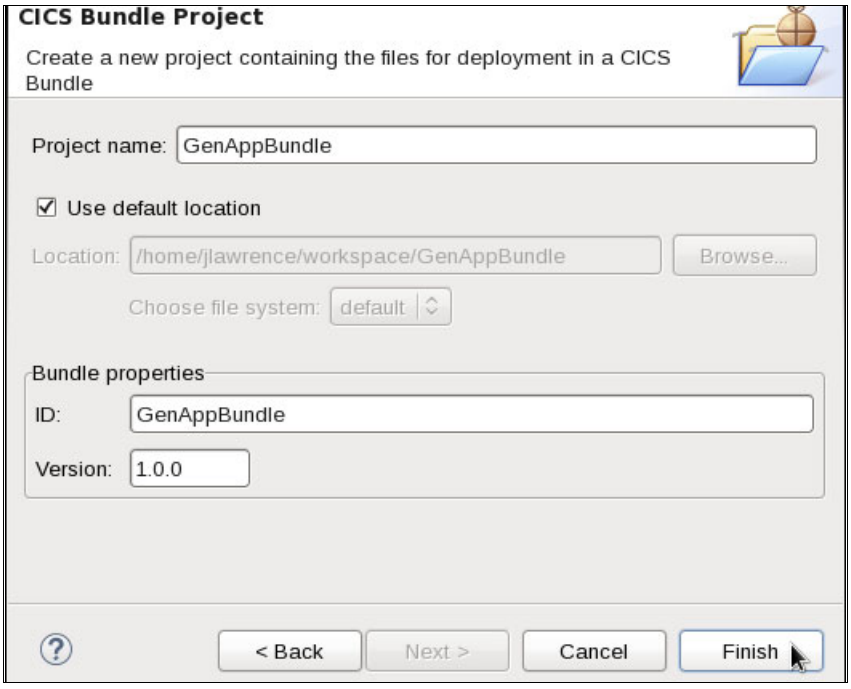


Figure 9-9 Specify the details for the new CICS bundle project

9. Add your previously imported GenAppWeb dynamic web project as a warbundle resource to the new CICS bundle project.
10. Expand the Bundle project and open the **`cics.xml` Bundle definition file** in the `META-INF` directory.

11. In the Defined Resources area, click **New**, then select **Dynamic Web Project Include** in the lower section of the list. A Dynamic Web Project Include dialog window opens, as shown in Figure 9-10.

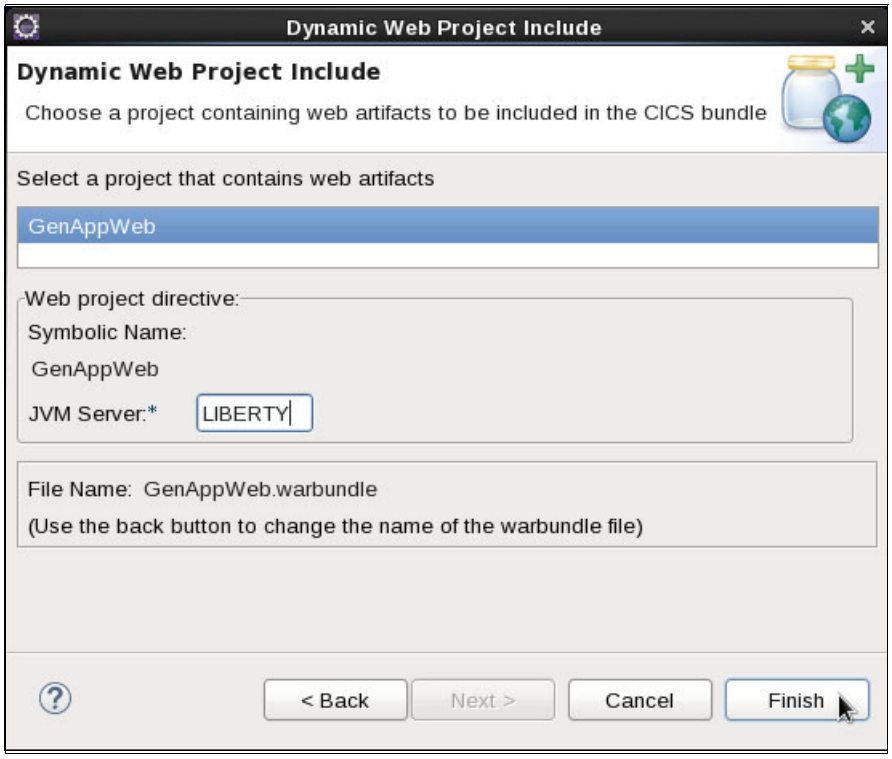


Figure 9-10 Specify the name of the target JVM server for Dynamic Web Project Include

12. Specify the name of the **JVM SERVER** in which the web application is to be deployed (see Figure 9-10) and **click** Finish. The new GenAppWeb.warbundle include appears in the GenAppBundle project.

9.6 Complete deployment and test the application

The final stage of the migration is to complete the deployment to the target CICS Liberty JVM Server and then to test the application. For this, you need an active z/OS FTP connection to the zFS file system of the target. Ensure that you have a connected z/OS FTP connection to the target in the z/OS perspective.

From this point, the deployment procedure is the same as documented for deploying the Hello World CICS Bundle project, documented in Chapter 5, “Developing and deploying applications” on page 69, so the description is not repeated here. Follow the equivalent steps as for the Hello World example application to export the CICS bundle to the z/OS UNIX file system, then define and install a CICS bundle definition for the application.

9.6.1 Test the web application

Finally, test the migrated application by accessing its URL from a browser. This will be the host name for the CICS region, with the HTTP or HTTPS port for the Liberty JVM server, with a URL, depending on the deployment descriptor for the web application. The full URL is in the `messages.log` file of the Liberty server when the application is installed, for example:

A CWWKT0016I: Web application available (default_host):

`http://<hostname>:65432/GenAppWeb/`

Figure 9-11 shows the initial page of the migrated GenAppWeb application, running in the target CICS Liberty JVM server.

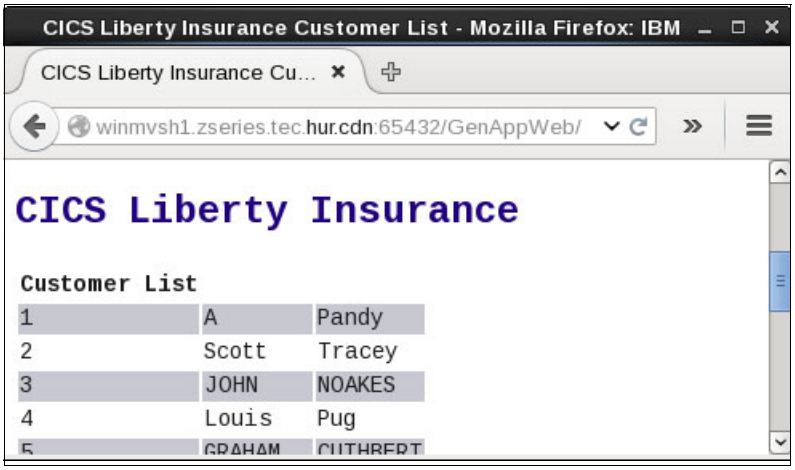


Figure 9-11 The migrated GenAppWeb application running in Liberty in CICS

In the case of unresolved build or deployment issues with the application, error messages are issued to indicate the cause, allowing you to resolve any problems.

For example, if you do not configure a JCA connection factory instance in `server.xml`, or if there is a mismatch between the JNDI lookup names used by the application and the connection factory instance definition, a `javax.naming.NameNotFoundException` is thrown, such as:

```
javax.naming.NameNotFoundException: eis/ECIJNDI
at com.ibm.ws.jndi.internal.ContextNode.lookup(ContextNode.java:218)
at com.ibm.ws.jndi.internal.WSContext.lookup(WSContext.java:295)
at com.ibm.ws.jndi.WSContextBase.lookup(WSContextBase.java:58)
at org.apache.aries.jndi.DelegateContext.lookup(DelegateContext.java:161)
at javax.naming.InitialContext.lookup(InitialContext.java:423)
at com.ibm.cics.genapp.bli.GenAppJCAImpl.<init>(GenAppJCAImpl.java:94)
... 35 more
```

This might be displayed in the `messages.log` file for the Liberty JVM server, depending on how the application handles exceptions from the JNDI lookup.

Or, if you are using the CICS JCA local ECI adapter, and an error occurs making the call to the CICS program, an exception is thrown, such as:

```
com.ibm.connector2.cics.CICSTxnAbendException:
CTG9638E Transaction Abend occurred in CICS: Abend Code=: AEIO, error code: AEIO
at com.ibm.connector2.cics.ECIManagedConnection.checkReturnCode
at com.ibm.connector2.cics.ECIManagedConnection.call
```

```
at com.ibm.connector2.cics.ECIConnection.call(ECIConnection.java:133)
at com.ibm.connector2.cics.ECIInteraction.execute(ECIInteraction.java:346)
at com.ibm.cics.genapp.bli.GenAppJCAImpl.invokeChannelProgram
... 36 more
Caused by:
com.ibm.cics.server.InvalidProgramIdException:
CICS PGMIDERR Condition(RESP=PGMIDERR, RESP2=2)
at com.ibm.cics.server.DTCProgram.LINK(Native Method)
at com.ibm.cics.server.Program.link(Program.java:477)
at com.ibm.ctg.client.LocalCICSJavaGateway.flow(JavaGateway.java:953)
at com.ibm.connector2.cics.ECIManagedConnection.flowGatewayRequest
at com.ibm.connector2.cics.ECIManagedConnection.call
... 39 more
```

The underlying InvalidProgramIdException is wrapped by an exception thrown from the CICS TG code used by the CICS local ECI adapter.

10

Creating an integration logic application

In this chapter, you learn how to use IBM WebSphere Liberty Profile (Liberty) within CICS Transaction Server (TS) for IBM z/OS V5.3 (CICS) to develop a new service-based interface into an existing CICS application. We use the general insurance application for all of our coding examples. Although there are many options available when choosing to expose applications, we use the representational state transfer (RESTful) and JavaScript Object Notation (JSON) technologies within Liberty.

This chapter covers only the process of constructing and exposing a RESTful web service with JSON data formats. It assumes the existence of a ready-made Java application programming interface (API) to service the requests. In this case, the Java API that we use is an interface to the CICS GenApp insurance application. This interface, referred to as the *GenApp Java business logic interface* (BLI), encapsulates access to the underlying CICS GenApp programs and resources.

The GenApp Java BLI is implemented by using Java EE Connector Architecture with external call interface (JCA ECI) requests to invoke the GenApp COBOL programs, and uses JZOS toolkit-record generated Java classes internally to marshal Java data into record structures, to transmit to and from CICS. The details of the implementation of the BLI are not described here. We treat BLI implementation as a black box, which provides the business methods required to fulfill the web service requests.

This chapter describes the following topics:

- JSON and the json-1.0 feature
- RESTful and the jax-rs-1.1 feature
- Exposing GENAPP through a RESTful JSON interface
- Package and deploy the application

10.1 JSON and the json-1.0 feature

JavaScript Object Notation (JSON) is an open standard format for data interchange. Although originally used in the JavaScript scripting language, JSON is now language-independent, with many parsers available in many languages.

Compared to Extensible Markup Language (XML), JSON has many advantages. Most predominantly, JSON is well-suited to easy integration for client-side software. It is based on the JavaScript syntax, however it is distinct from it. Suitable for any data exchange, JSON is often used within RESTful web services because of its easy consumption. XML is an extremely verbose language. Every element in the tree has a name, and the element must be enclosed in a matching pair of tags. Alternatively, JSON expresses trees in a nested array format, similar to JavaScript. This enables the same data to be transferred in a far smaller data package with JSON than with XML.

The Java API for JSON (json-1.0) feature provides a standardized method for constructing and manipulating data to be rendered in JSON.

10.2 RESTful and the jax-rs-1.1 feature

Representational State Transfer (REST) is a defined set of architectural principles by which you can design web services that focus on service resources. The REST architectural pattern takes advantage of the technologies and protocols of the World Wide Web to describe how data objects can be defined and modified.

In contrast to a request-response model, such as the SOAP protocol specification, which focuses on procedures made available by the system, REST is modeled around the resources in the system. Each resource is globally identifiable through its uniform resource identifier (URI). Because REST does not focus on the procedures and services provided by a system, a few actions are defined based on existing HTTP methods: GET, POST, PUT, DELETE, HEAD.

The <jaxrs-1.1> feature provides technologies that simplify the creation of RESTful-like web services. This is achieved by the use of Java annotations. For example, Table 10-1 shows annotation declarations before a Java method, in a jax-rs application, can be called directly when the respective HTTP method is received.

Table 10-1 A mapping of Java annotations to HTTP methods and their expected behaviors

Annotation	HTTP method	RESTful behavior expected
@GET	GET	Retrieve a resource representation
@PUT	PUT	Modify a resource representation
@POST	POST	Create a new resource representation
@DELETE	DELETE	Delete a resource representation
@HEAD	HEAD	Retrieve a resource's metadata

10.3 Exposing GENAPP through a RESTful JSON interface

We now outline the steps that are needed to expose services from the GENAPP application through a RESTful JSON interface using the <jax-rs-1.1> and <json-1.0> features.

To expose a CICS application through a RESTful service, you need to follow a number of steps. At a high level, follow these steps:

- Mapping a URI to your application in the Liberty RESTful servlet
- Writing the application logic that handles the RESTful requests
- Packaging and deploying the application to Liberty running inside CICS
- Testing the RESTful service

We now review those steps in detail, using the sample application ported in Chapter 9, “Porting a web application” on page 161 as a starting point.

You now need to create the local project that we will use to build your application. Perform the following steps in your Eclipse environment to create a new Dynamic Web project:

1. Switch to the Java EE perspective by selecting **Window** → **Open Perspective** → **Other** → **Java EE**. Click **OK**.
2. Select **File** → **New** → **Other** → **Dynamic Web Project**.
3. Give the project a name and ensure that Dynamic web mobile version has the value of 3.0.
4. Clear **Add project to an EAR** in the EAR membership section.
5. Click **Next** and **Next** again.
6. Select **Generate web.xml deployment descriptor**.
7. Click **Finish**.

Before you start to configure and write your RESTful application, you need to configure the build path of the project. We know that we will use the libraries supplied by Liberty and the GenApp Java business logic interface, which provides access to the underlying CICS GenApp application. Add these libraries to the project by following these steps:

1. Right-click your Integration logic application package and select **Build Path** → **Configure Build Path**.
2. On the Projects tab, click **Add**.
3. Select the **Business logic application project** and click **OK**.
4. On the Libraries tab, ensure that Liberty JVM server libraries are present. If they are not, click **Add Library**. Select **Liberty JVM server libraries**. Click **Next** and select **CICS TS 5.3** in the Version drop-down box. Click **Finish**.
5. Open **`web.xml`** and click **Manage Utility Jars** on the Design tab.
6. Click **Add**.
7. Select **Project** and click **Next**.
8. Select the Business logic application project **GenAppBli** and click **Finish**.

We have now extended our project build path to include the business logic project that we imported earlier, in addition to the Liberty JVM server libraries. We have also configured Eclipse to ensure that when we build and package our Integration logic application, the business logic package is built and deployed with our project as a utility JAR.

10.3.1 Mapping a URI to your application

We will map a URI pattern to the application that we will write to handle incoming RESTful requests. We do this by configuring the IBM JAX-RS REST servlet, which is provided by the Liberty web container. This mapping is created with the steps shown in Example 10-1. Add the following to `WebContent/WEB-INF/web.xml`.

Example 10-1 Excerpt of web.xml

```
<servlet>
   <description>RESTful JSON service for the GENAPP application</description>
   <servlet-name>GenAppService</servlet-name>
   <servlet-class>com.ibm.websphere.jaxrs.server.IBMRestServlet</servlet-class>
   <init-param>
      <param-name>javax.ws.rs.Application</param-name>
      <param-value>com.ibm.cics.genapp.services.ServiceConfig</param-value>
   </init-param>
/servlet>
```

Example 10-1 defines our servlet to the Liberty web container. Each individual field is explained in Table 10-2.

Table 10-2 Description of the parameters when creating a servlet in the web.xml configuration file

Field name	Example value	Field description
servlet	Not applicable	The root node. All fields encapsulated within this XML node define the servlet that we are creating.
servlet-name	GenAppService	The name used for internal reference to this servlet.
servlet-class	com.ibm.websphere.jaxrs.server.IBMRestServlet	The type of servlet we want to create. In this instance, we are extending the IBM JAX-RS REST servlet.
init-param	Not applicable	The initialization parameters required for this type of servlet. All parameters must be encapsulated inside this node.
param-namej	javax.ws.rs.Application	The name of an initialization parameter. In this case, we are describing where the IBM JAX-RS REST servlet will look for the configuration class of our service.
param-value	com.ibm.cics.genapp.services.ServiceConfig	The value of the initialization parameter. In this case, we are defining a class that we will create, which will define to the IBM JAX-RS REST servlet which entry point Java classes are used in our RESTful application.

When deployed, this will define the existence of our servlet to the Liberty web container. Before we begin to write the code for our service, it is important to map the invocation of our service to an inbound URI request. To do this, insert the XML shown in Example 10-2 on page 183 into the `web.xml` file.

Example 10-2 Excerpt of web.xml

```
<servlet-mapping>
<servlet-name>GenAppService</servlet-name>
<url-pattern>/service/*</url-pattern>
</servlet-mapping>
```

The fields in Example 10-2 are described in Table 10-3.

Table 10-3 Description of the parameters required to map a servlet to an inbound URL

Field name	Example value	Field description
servlet-mapping	Not applicable	The root node used for mapping a servlet to an inbound URL.
servlet-name	GenAppService+	The name of the servlet we want to map to a URI. It is important that this value matches the value of servlet-name that was used when defining the servlet to the IBM JAX-RS REST servlet.
url-mapping	/services/*	The URL pattern to map to the servlet. This maps our servlet to a URI that matches this URI pattern suffixed to the domain address of our Liberty application.

With this XML, we defined a servlet mapping between our GenAppServlet servlet and the URI mapping of <domain & port>/GenAppService/services/*. Where GenAppService is the URI of our Liberty application and /services/* maps all requests to our RESTful service.

10.3.2 Writing the application logic that handles the RESTful requests

At this point, we have only defined the existence of one Java class to the IBM JAX-RS REST servlet. The value that we entered for <param-value> in the servlet definition is the subclass that will be used when configuring which classes are available as entry points to our REST service. It is important that we create this class exactly as named. Create `com.ibm.cics.genapp.services.ServiceConfig` with the Java code that is shown in Example 10-3 on page 184.

Example 10-3 ServiceConfig class

```
package com.ibm.cics.genapp.services;

import java.util.Set;
import java.util.HashSet;

public class ServiceConfig extends javax.ws.rs.core.Application {

      public Set<Class<?>> getClasses()
      {
            Set<Class<?>> classes = new HashSet<Class<?>>();
            classes.add(com.ibm.cics.genapp.services.CustomerService.class);
            return classes;
      }
}
```

This subclass extending javax.ws.rs.core.Application returns a set of classes that the IBM JAX-RS REST servlet will use when attempting to match incoming URI and HTTP method combinations for your RESTful service. In our example, we defined one class: com.ibm.cics.genapp.services.CustomerService.class. You can provide more classes here if you require multiple classes to handle your RESTful service requests. We demonstrate how this is useful when creating a service that interacts with multiple resources. Follow these steps to create your first service:

1. Create the class `com.ibm.cics.genapp.services.CustomerService`.
2. Above the class declaration, add, `@javax.ws.rs.Path("/customer")`. This defines a further URI mapping to this class. In our example, with the configuration in `web.xml`, all URIs with the mapping <domain & port>/GenAppWeb/services/customer will use this class to resolve the request. It is at this point that you can define a different @javax.ws.rs.Path annotation and add a new class to ServiceConfig if you want to create multiple classes to handle different resources. For example, you can create a class with annotation @jvax.ws.rs.Path(“/policy”) to handle requests against policies in the GENAPP application.
3. Create a new method in CustomerService named FetchCustomerById with the definition shown in Example 10-4.

Example 10-4 FetchCustomerById method

```
@javax.ws.rs.GET
@javax.ws.rs.Produces("application/json")
@javax.ws.rs.Path("/{customerNumber : \\d+}}")
public JSONObject FetchCustomerById(
      @javax.ws.rs.PathParam("customerNumber")String customerNum)
{
```

This defines to the IBM JAX-RS REST servlet that this method is driven when the following statements are true:

- The URI <domain & port>/GenAppWeb/services/customer/<data> was received, where data matches the regular expression of `\\d+`, digits only.
- An HTTP GET request was received.
- The request produces a MIME media type of application/json as a response.

This definition also defines that the method driven will produce a response of JSONObject, which is the Java JSON object provided by the Liberty feature <json-1.0> for parsing and serializing JSON data. In Example 10-4 on page 184, we used a variable value in the javax.ws.rs.Path annotation for the first time. This allows you to define URI mappings that will extract data from the URI and input those as path parameters for use in your Java code. In this example, we used the {customerNumber} variable as input data for the FetchCustomerById method, usable with the Java variable customerNum within the method. This means that a request that looks like the following, results in the method `FetchCustomerById` being driven with the input variable customerNumber defined as the value "1". :

```
GET .../GenAppWeb/services/customer/1
```

The annotation `@javax.ws.rs.Path` supports regular expressions. The annotation @javax.ws.rs.Consumes is also available. This annotation defines which Content-Type the method receives. This can be useful if you want to create services that handle multiple different types, such as `application/json` and `application/xml`.

We are now in a position to create the logic for our interface. For our sample application, create a new class named `CustomerService` in the `com.ibm.cics.genapp.services package`.

Use the following package declaration for the application.

```
package com.ibm.cics.genapp.service;
```

The following imports are for the classes needed to use the GenApp Java Platform, Enterprise Edition business logic interface. These classes must also be available at run time, by importing the classes into your project, and including the `GenAppBli.jar` as a utility JAR as described earlier in 10.3, "Exposing GENAPP through a RESTful JSON interface" on page 181.

The source code for the GenApp Java BLI is provided with the additional materials for this book at the following URL:

http://www.redbooks.ibm.com/abstracts/sg248335.html

The interface is specified by `com.ibm.cics.genapp.bli.GenAppInterface`, and is self-documenting, though Javadoc comments are also included.

The interface uses some auxiliary *holder* classes for customer and policy data passed to and from the interface methods. You need to import the following holder classes. These classes have getters and setters for the individual data items that they encapsulate.

Example 10-5 Import statements for auxiliary classes

```
import com.ibm.cics.genapp.GenAppException;
import com.ibm.cics.genapp.bli.Customer;
import com.ibm.cics.genapp.bli.CustomerData;
import com.ibm.cics.genapp.bli.GenAppInterface;
import com.ibm.cics.genapp.bli.GenAppJCAImpl;
import com.ibm.cics.genapp.bli.PolicyData;
import com.ibm.cics.genapp.bli.PolicyType;
```

Import the following IBM JSON support classes, which are used to build the object model for JSON. These are provided by the json-1.0 feature, which must be included in `server.xml` for your Liberty server. See Example 10-6.

Example 10-6 Import statements for IBM JSON support classes

```
import com.ibm.json.java.JSONArray;
import com.ibm.json.java.JSONObject;
```

The example application that we create uses variables and static strings, among other things that the labels used for individual data items within the JSON. They are shown in Example 10-7.

Example 10-7 Declaration of variables and static strings

```
     private boolean error = false;
     private static final String ERROR_MSG_LABEL = "errorMsg";
     private static final String ERROR_CODE_LABEL = "errorCode";
     private static String errorMsg = "";
     private static int errorCode = 0;
     private static final String ERROR_NOT_INTEGER = "Customer number must
  contain only numerical characters";
     private static final String ERROR_GENAPP_ERROR = "Error retrieving response
  from the GenApp application.";

     private static final int ERROR_NOT_INTEGER_CODE = 1;
     private static final int ERROR_GENAPP_ERROR_CODE = 2;

     private static final String CUSTOMER_NO_LABEL = "customerNumber";
     private static final String FIRST_NAME_LABEL = "firstName";
     private static final String LAST_NAME_LABEL = "lastName";
     private static final String BIRTH_YEAR_LABEL = "birthYear";
     private static final String BIRTH_MONTH_LABEL = "birthMonth";
     private static final StringBIRTH_DAY_LABEL = "birthDay";
     private static final String HOUSE_NAME_LABEL = "houseName";
     private static final String HOUSE_NUMBER_LABEL = "houseNumber";
     private static final String POST_CODE_LABEL = "postCode";
     private static final String MOBILE_PHONE_LABEL = "mobilePhone";
     private static final String EMAIL_ADDRESS_LABEL = "emailAddress";

     private static final String POLICY_NO_LABEL = "policyNumber";
     private static final String POLICY_TYPE_LABEL = "policyType";
     private static final String POLICY_ISSUE_DATE = "issueDate";
     private static final String POLICY_EXPIRY_DATE = "expiryDate";

     private static final String HTTP_CODE_LABEL = "httpCode";
     private static int httpResponseCode = 0;
     private static final int HTTP_OK = 200;
     private static final int HTTP_BAD_REQUEST = 400;
     private static final int HTTP_NOT_FOUND = 404;
     private static final int HTTP_INTERNAL_ERROR = 500;

     private static final String DATA_LABEL = "data";
     private static final String RECORD_COUNT_LABEL = "recordCount";
```

Now that the infrastructure for our application is in place, we can focus on the logic and configuration of the RESTful service. The Java code shown below uses the Genapp Business logic interface to fetch a single customer record from Genapp running inside a CICS region. It then returns that customer record as a JSON response to an HTTP client that made the request:

```
JSONObject cJson = new JSONObject();
   try {
```

Access to CICS data is by means of JCA to the GenApp COBOL programs. In this case, the JCA communications, and marshalling of the required record structures for the commareas and containers, have been encapsulated within the GenApp Java BLI component.

To access the underlying CICS GenApp programs, first instantiate an instance of the GenAppJCAImpl class, which is the live JCA implementation of the GenAppInterface Java interface:

```
GenAppInterface genAppJcaImpl = new GenAppJCAImpl();
```

To look up a specific GenApp customer record and obtain the customer details, invoke the getCustomer() method, passing the required customer number. This method builds a commarea in the appropriate format, invokes the LGICUS01 GenApp COBOL program using JCA, and extracts the customer details returned in the commarea.

The getCustomer() method shown in the following example returns a CustomerData object, which is a holder for the customer details extracted from the customer record returned from the JCA call:

```
CustomerData cdata = genAppJcaImpl.getCustomer(customerNumber);
```

If the call was successful and the customer was found (meaning, no GenAppException has been thrown (this is guaranteed by reaching this point in the code)), we can obtain the individual customer attributes by using getter methods on the returned CustomerData object, cdata. We extract each required attribute in turn, adding to the JSONObject Map with an appropriate label for each attribute. See Example 10-8.

Example 10-8 Sample of getter methods to retrieve customer attributes

```
cJson.put(CUSTOMER_NO_LABEL, cdata.getCustomerNumber());
cJson.put(FIRST_NAME_LABEL, cdata.getFirstName());
cJson.put(LAST_NAME_LABEL, cdata.getLastName());

if(cdata.getBirthDay() > 0)
{
   cJson.put(BIRTH_DAY_LABEL, cdata.getBirthDay());
   cJson.put(BIRTH_MONTH_LABEL, cdata.getBirthMonth());
   cJson.put(BIRTH_YEAR_LABEL, cdata.getBirthYear());
}
cJson.put(HOUSE_NAME_LABEL, cdata.getHouseName());
cJson.put(HOUSE_NUMBER_LABEL, cdata.getHouseNumber());
cJson.put(POST_CODE_LABEL, cdata.getPostCode());
cJson.put(MOBILE_PHONE_LABEL, cdata.getMobilePhone());
cJson.put(EMAIL_ADDRESS_LABEL, cdata.getEmailAddress());
```

If an exception is thrown by the mainline processing, we catch it here and set an appropriate error message and HTTP response code, depending on the type of exception that is caught. See Example 10-9.

Example 10-9 Sample code for catching exceptions

```
        } catch (NumberFormatException e) {
              errorMsg = ERROR_NOT_INTEGER;
              errorCode = ERROR_NOT_INTEGER_CODE;
              httpResponseCode = HTTP_BAD_REQUEST;
              error = true;
        } catch (GenAppException e) {
              error = true;
              errorMsg = ERROR_GENAPP_ERROR;
              httpResponseCode = HTTP_NOT_FOUND;
              e.printStackTrace();
  }
```

Instantiate a top-level JSONObject to return to the caller. This will be a holder for the response, representing either the customer data or an error message. See Example 10-10.

Example 10-10 Instantiating a new JSON object

```
           JSONObject jsonResponse = new JSONObject();
```

If the method is successful and we have valid customer data to return, add this code to the response object. See Example 10-11.

Example 10-11 Adding to response object if return is successful

```
        if(!error)
        {
              jsonResponse.put(DATA_LABEL, cJson);
        }
```

Otherwise, add the error code and message to the response object. See Example 10-12.

Example 10-12 Adding to response object if an error is returned

```
        else
        {

              jsonResponse.put(ERROR_CODE_LABEL, errorCode);
              jsonResponse.put(ERROR_MSG_LABEL, errorMsg);
        }
```

Add the appropriate HTTP code to the response object. See Example 10-13.

Example 10-13 Set the HTTP code to the response object

```
        jsonResponse.put(HTTP_CODE_LABEL, httpResponseCode);
```

Finally, return the top-level JSONObject response to our caller, the Liberty JAX-RS framework, which marshals the JSON into an HTTP response to the web service client. See Example 10-14.

Example 10-14 Return the JSON response

```
        return jsonResponse;
```

10.4 Package and deploy the application

We have shown you the steps that are needed to create and configure a JAXRS application. Follow the steps in 5.3.2, “Deploying to CICS as an EBA” on page 94 to create a bundle project with our application and deploy that to CICS. Ensure that you select the **Dynamic Web Project Include** instead of OSGi Application Project described in step 3 on page 94.

10.4.1 Testing the RESTful service

Our JAXRS application has been configured and deployed to CICS. The next step is to test that our service is available and works correctly. Our application so far has only created a simple GET method for fetching one customer. This takes no HTTP body input from the client and can therefore be tested by using a standard web browser. If you want to test an application that uses an HTTP body, such as the POST methods that are available in the full Genapp sample, you might need to download a RESTful client. The Chrome web browser has one such client named “Simple REST Client.”

To test our application, we can send an HTTP GET request to the domain and path of our application. In this example, that is <domain port>/<application_root>/service/customers/8. This attempts to find a customer with customerNumber 8. Figure 10-1 on page 190 shows an HTTP GET request being made to the application and running a successful retrieval of customer 8 in JSON format.

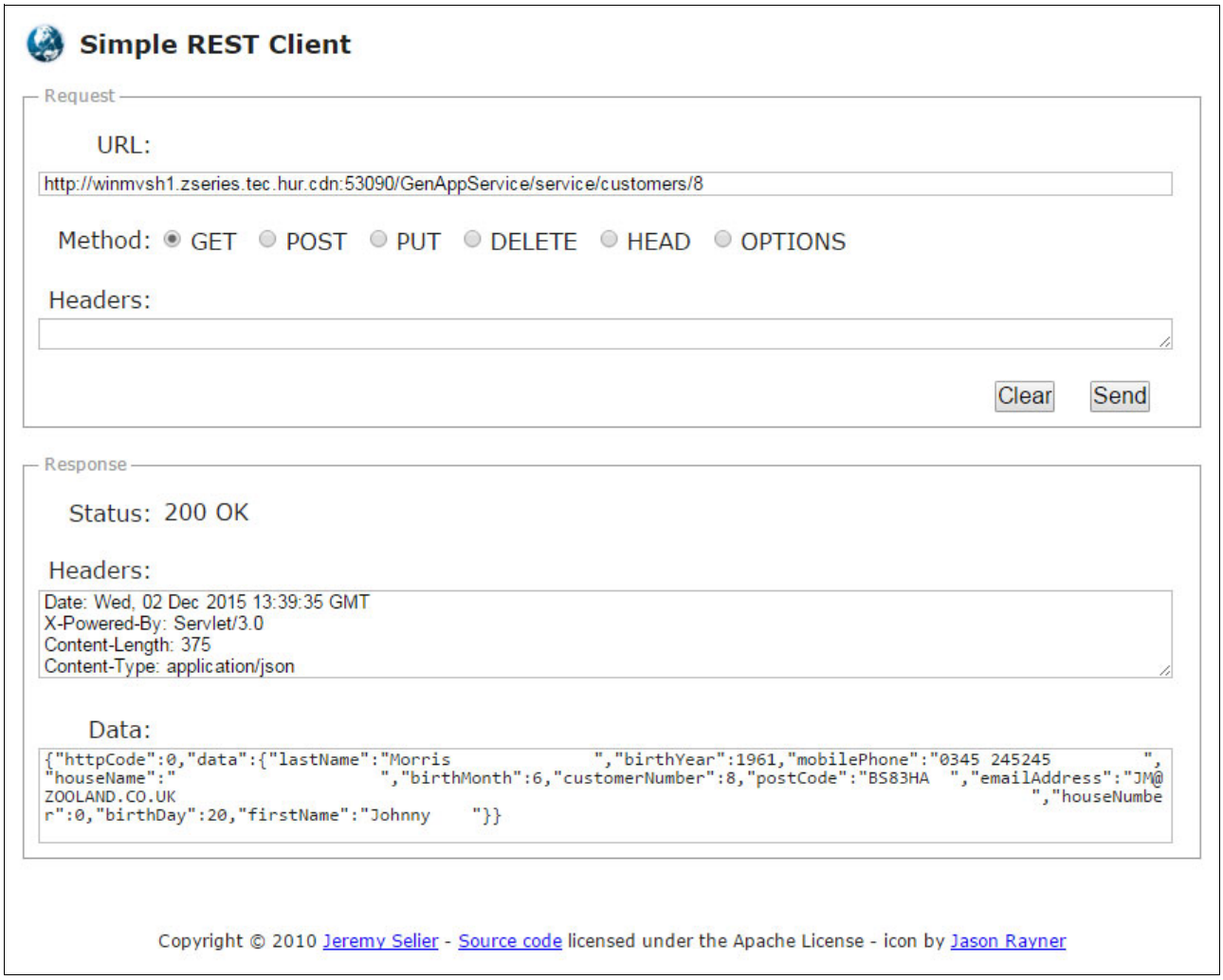


Figure 10-1 The Simple REST Client Chrome add-on making a successful GET request to the Genapp JSON application

11

Creating a business logic application

In Chapter 9, “Porting a web application” on page 161, you saw how to take an existing Java application using Java EE Connector Architecture (JCA) to access CICS programs, migrate, and deploy it into Liberty in CICS. This is the first stage in exploiting the capability of Liberty in CICS for Java EE applications. The chapter that followed, Chapter 10, “Creating an integration logic application” on page 179, showed how to develop a new service-based interface into an existing CICS application by building a JavaScript Object Notation (JSON) Representational State Transfer (REST) web service for the Java business logic interface.

In this chapter, we show how to further exploit Liberty in CICS to extend the flexibility and function of an application by using more of the capabilities of the Liberty server. We first cover conversion of the existing application to Open Services Gateway initiative (OSGi) and then show how to extend the function in two ways:

- Adding a new custom COBOL program to implement a new business function and invoking it from the Liberty application via JCA.
- Using JCICS to implement another new business function, directly accessing a Virtual Storage Access Method (VSAM) key sequenced data set (KSDS) from Java.

There are many other ways in which the Liberty server capabilities can be used to enhance and extend an application and in the conclusion of this chapter, we give a brief summary of a few of these.

The following topics are covered in this chapter:

- Using OSGi with Liberty in CICS
- Extending the business logic application
- Summary

11.1 Using OSGi with Liberty in CICS

In previous chapters, the applications have been developed and deployed as Java EE applications, using non-OSGi Java or dynamic web projects. In this section, we show how to take a non-OSGi web application, and convert and deploy it as an assembly of OSGi bundles by creating a shared OSGi bundle and an enterprise bundle archive (EBA).

Liberty in CICS is an OSGi-enabled Java EE application server environment. There are a number of advantages to deployed applications as OSGi bundles, including improved encapsulation of separate components, a clear version management capability, and easy sharing of common components between applications.

In this section, we show how to take the existing GenApp Java EE projects, convert them to OSGi bundles, and deploy them to Liberty in CICS.

There are four main stages of this conversion:

1. Copying the original Eclipse project to create a new version (optional, though strongly suggested).
2. Removal of the existing build path entries and utility JARs.
3. Conversion of the projects to OSGi bundles using the Eclipse tooling.
4. Editing of the OSGi manifests to correct errors and ensure correct runtime resolution of dependencies.

First, we need to create an OSGi bundle to contain the `ibmjzos.jar` utility JAR.

11.1.1 Create an OSGi bundle project for ibmjzos.jar

The GenAppBli Java project, which provides the Java business logic interface used by the web and service front end modules, interacts with the GenApp base COBOL programs using JZOS record-generated classes to marshall the records to pass to the CICS programs.

These JZOS-generated classes depend on utility classes in the `com.ibm.jzos.fields` package from the `ibmjzos.jar` file included with the IBM Java SDK on z/OS. In previous chapters, the classes were simply added to the build path for the GenAppBli project because they are automatically available in the z/OS Java environment. When the application projects are converted to OSGi, however, they can only access packages from other OSGi bundles, so `ibmjzos.jar` needs to be converted to an OSGi bundle.

This process is achieved by creating a wrapper project containing the required JAR and exposing the `com.ibm.jzos.fields` package for use by other OSGi bundles as follows:

1. Download the `ibmjzos.jar` file from the JRE on z/OS. The JAR file is in the `/lib/ext` directory of the Java runtime installation, for example:

   `/java/J7.0_64/lib/ext/ibmjzos.jar`
2. Download the JAR file as a binary file using FTP or another mechanism to a suitable directory on your workstation.
3. Create a new OSGi bundle to contain `ibmjzos.jar`. In the Eclipse development environment, click **File → New → OSGi Bundle Project**. The New OSGi Bundle Project dialog opens.
4. Enter a name for the new bundle, for example **ibmjzosgi**. Accept the default location for the project, but clear the check box to “Add bundle to application”. Click **Next →** to continue.

5. This OSGi bundle project does not contain any source or built classes because it is just a holder for `ibmjzos.jar`. Therefore, you can remove the `src` folder from the build path if you want. Select the `src` source folder and click **Remove**.

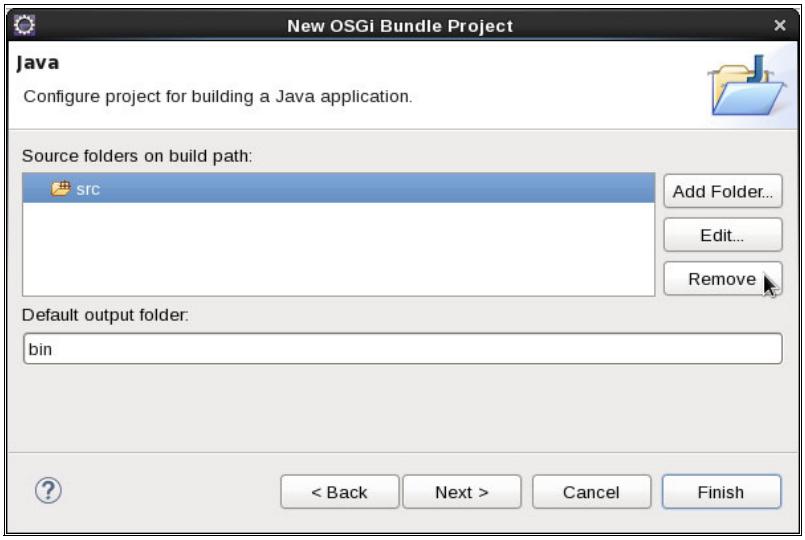


Figure 11-1 Remove the source folder from the OSGi bundle project

6. On the final dialog to create the OSGi bundle, you can remove the `.qualifier` element from the bundle version. Again, this is optional because it does not affect the use of the bundle.

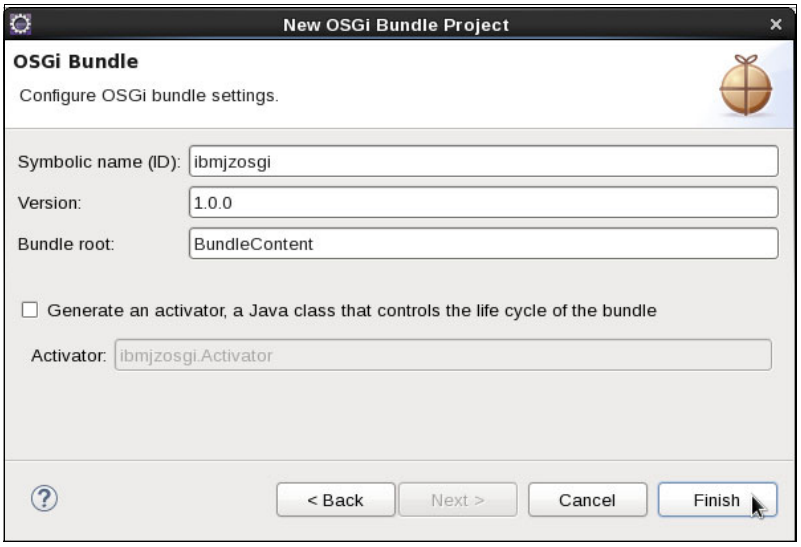


Figure 11-2 Finalize the creation of the OSGi bundle

7. Click **Finish** to complete creating the new OSGi bundle. The new bundle project is created in the workspace.
8. The next stage is to import `ibmjzos.jar` into the bundle, and then export the `com.ibm.jzos.fields` package. Right-click the **ibmjzosgi** project and click **Import**. The import dialog opens. Expand the OSGi section and select **Java Archive into an OSGi Bundle** as shown in Figure 11-3.

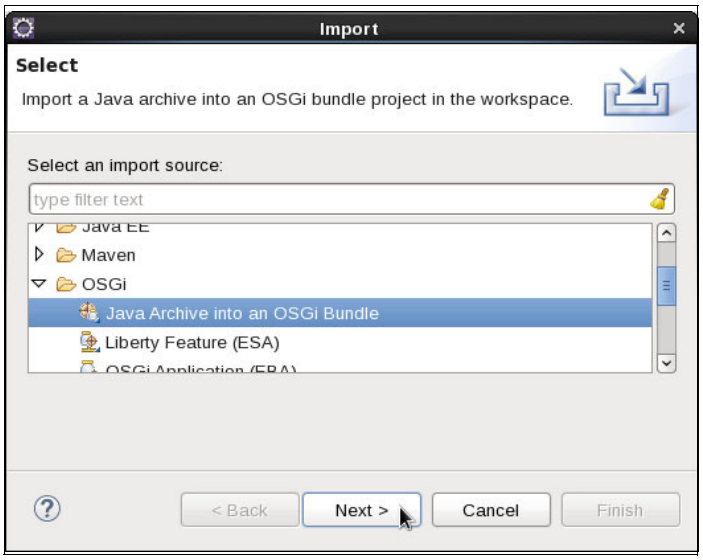


Figure 11-3 Import a Java archive into an OSGi bundle

9. Click **Next** to continue. The Java Archive Import dialog is displayed.
10. Click **Browse** and select the `ibmjzos.jar` file that you downloaded in step 1 above, then click **Next** to continue.

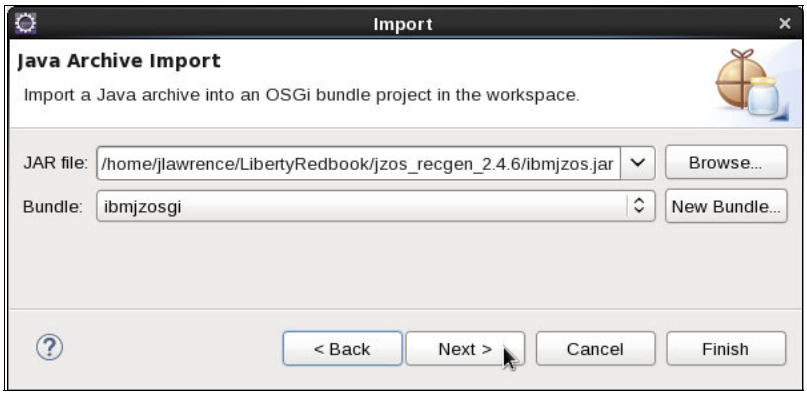


Figure 11-4 Import the JAR file into the OSGi project

11. The final dialog for the import (Figure 11-5 on page 195) allows the selection of the packages to be exported by the OSGi bundle. Only the `com.ibm.jzos.fields` package needs to be visible, so clear the boxes for the others and then click **Finish** to complete.

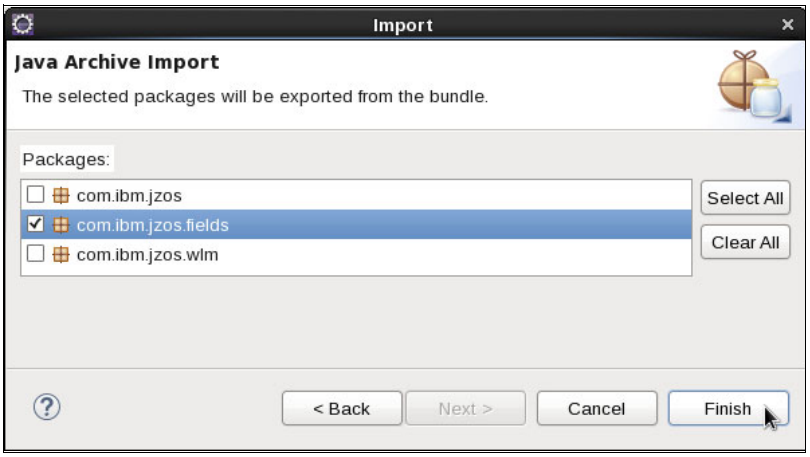


Figure 11-5 Selection of packages to export

The `ibmjzos.jar` file is now imported into the OSGi bundle and the bundle manifest is updated to export the `com.ibm.jzos.fields` package from the bundle. The final `MANIFEST.MF` file for the bundle contains the content as shown in Example 11-1.

Example 11-1 OSGi bundle manifest

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: ibmjzosgi
Bundle-SymbolicName: ibmjzosgi
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Bundle-ClassPath:  ibmjzos.jar
Export-Package: com.ibm.jzos.fields
```

11.1.2 Convert the existing Java projects to OSGI

Next, we need to convert the existing Java projects to use OSGi.

Complete the following steps to convert the GenAppBli project to use OSGI:

1. In your Eclipse development environment with the GenApp projects imported and correctly building, right-click the **GenAppBli** project and select **Copy** (or use Ctrl-C). Then, click **Edit → Paste** (or Ctrl-V). A Copy Project dialog is displayed with a suggested new project name.

2. In the Copy Project dialog, replace the suggested name (Copy of GenAppBli) with a new name **GenAppBliV2** as shown in Figure 11-6.
3. Click **OK** and the new project is created as a copy of the original.

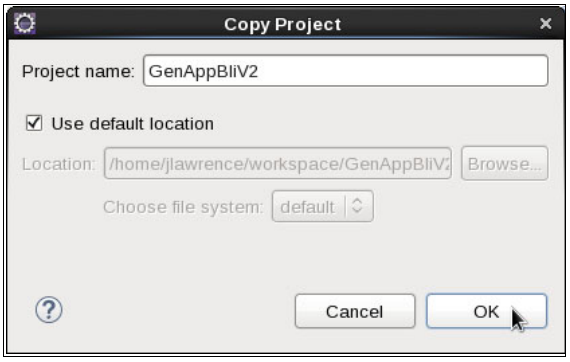


Figure 11-6 Copying the GenAppBli project to GenAppBliV2

4. Right-click the new **GenAppBliV2** project and select **Configure → Convert to OSGi Bundle Project**. The “Project conversion to OSGi” dialog window appears.
5. Because the project contains no binary files, which might have hidden dependencies, you can clear the check box to search binary files. Click **OK** as shown in Figure 11-7.

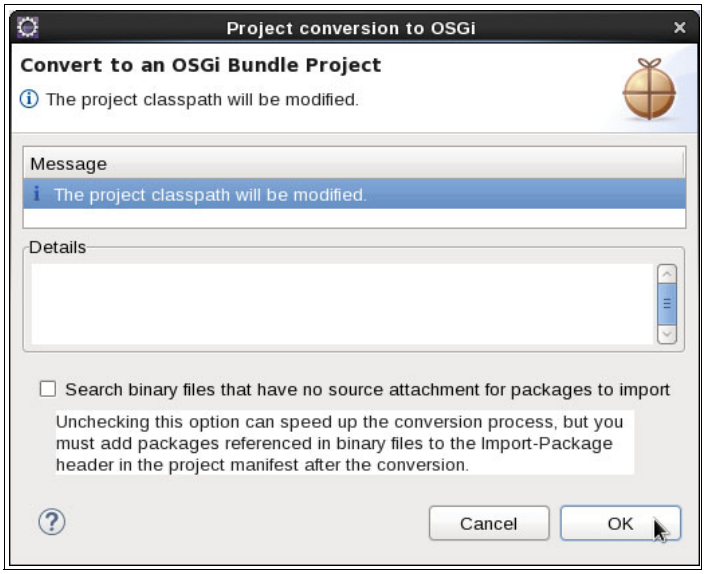


Figure 11-7 The Project conversion to OSGi dialog window

A progress bar might appear for a few seconds while the OSGi project facet is installed. Then, the project is converted and it might now contain errors because the classpath has been updated and no longer includes all required dependencies. These errors now need to be repaired.

The errors should be confined to the OSGi manifest `MANIFEST.MF` for the bundle project, as shown in Figure 11-8 on page 197 because not all the import dependencies can be found.

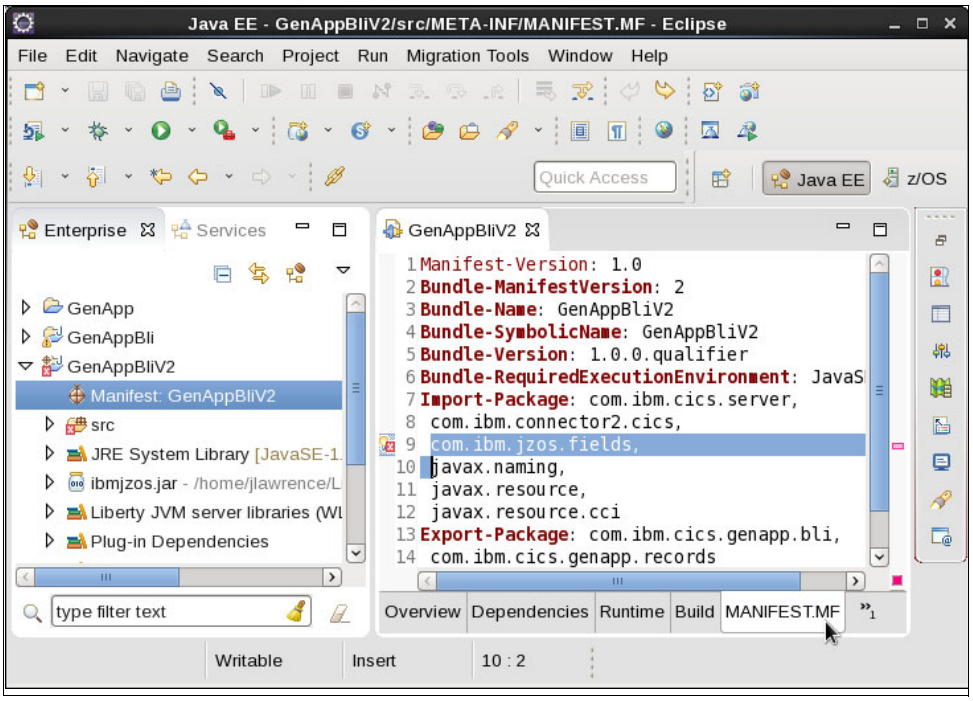


Figure 11-8 Source tab of the manifest editor

To fix the error in the `MANIFEST.MF`:

1. Expand the project **GenAppBliV2** and open **Manifest: GenAppBliV2**.
2. Select the `MANIFEST.MF` tab to edit the source for the manifest, as shown in Figure 11-8. The **`Import-Package`** entry for package `com.ibm.jzos.fields`, highlighted in Figure 11-8, is required as this package is one of the platform extensions in the CICS Liberty JVM server.
3. Also, delete the **`Export-Package`** entry for `com.ibm.cics.genapp.records`. This package is internal to the GenAppBli implementation and does not need to be exported.
4. Remove **`.qualifier`** from `Bundle-Version` and add `;version="1.0.0"` to the exported package. See Example 11-2 on page 197. These changes simplify some of the later steps, and allow version dependence to be specified for users of this bundle.

   Your OSGi manifest should now look like as shown in Example 11-2. The entry `Import-Package: com.ibm.cics.server` is needed for the JCICS API used later in this chapter so it is included now for convenience.

Example 11-2 Final OSGi manifest

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: GenAppBliV2
Bundle-SymbolicName: GenAppBliV2
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Import-Package: com.ibm.cics.server,
 com.ibm.jzos.fields,
 com.ibm.connector2.cics,
 javax.naming,
 javax.resource,
 javax.resource.cci
Export-Package: com.ibm.cics.genapp.bli;version="1.0.0"
```

5. Save the changes and close the manifest editor.

Next, complete a similar process for the GenAppWeb dynamic web project:

1. Copy the **GenAppWeb** project to a new project **GenAppWebV2** and convert to an OSGi bundle in the same way as for **GenAppBli**. Errors will again appear in the project, which will be corrected by editing the manifest.
2. Expand project **GenAppWebV2** and open the OSGi manifest for the bundle in the manifest editor and select the `MANIFEST.MF` tab.
3. Remove the entry `WEB-INF/lib/GenAppBli.jar` from the `Bundle-Classpath`. This is not needed because the dependency will instead be satisfied by using OSGi import from the shared bundle.

11.1.3 Deploying the OSGi version to Liberty in CICS

The original projects comprising the GenAppWeb application have now been converted to OSGi bundles. We now show you how to package and deploy these to the Liberty JVM server in CICS to allow you to test whether the conversion was successful.

The common business logic component GenAppBli will be deployed as a shared OSGi bundle, while the application-specific web component GenAppWeb will be deployed as an enterprise bundle archive (EBA) embedded in a CICS bundle:

1. In the Eclipse development environment, right-click project **GenAppBliV2** and select **Export → OSGi Bundle or Fragment**. The Export dialog appears.
2. Use the **Browse** option to select an appropriate directory and file name for the OSGi JAR file as in Figure 11-9.
3. Click **Finish** and the OSGi bundle is packaged and placed in the designated location.

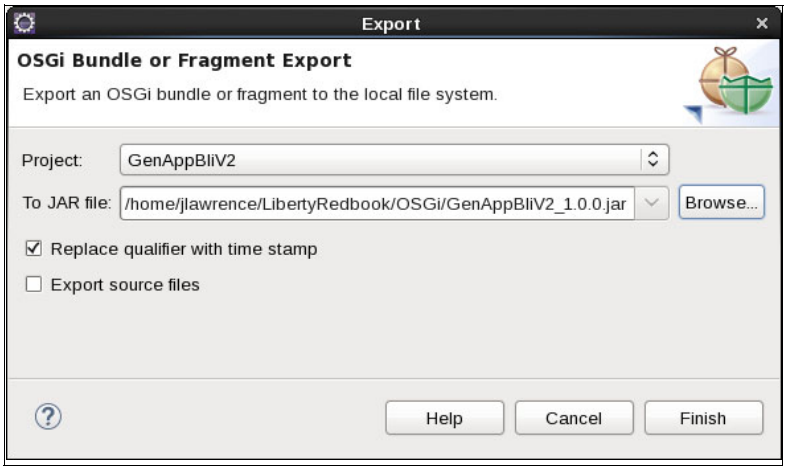


Figure 11-9 Export dialog for the OSGi shared bundle

4. Transfer the packaged OSGi bundle as a binary file to a suitable location in zFS via FTP or another mechanism.
5. Edit the `server.xml` configuration file for the target Liberty JVM server and add a `<bundleRepository>` element for the OSGI bundle transferred in Step 4, for example:

```
<bundleRepository>
  <fileset dir="/u/redg11/cicsts53/OSGi" include="GenAppBliV2_1.0.0.jar"/>
</bundleRepository>
```

This makes the exported packages from the bundle available to all installed OSGi applications in the Liberty server runtime.

The OSGi version of the GenAppWeb application is packaged as an OSGi application and deployed as a CICS bundle. We start by creating a new OSGi application project as follows:

1. In the Eclipse development environment, click **File → New → OSGi Application Project**. A New OSGi Application Project dialog is displayed.
2. Enter a name for the new project, such as **GenAppWebV2App** and click **Next**.
3. Select the check box for the OSGi project for the GenAppWeb OSGi version created earlier, but not for the shared OSGI bundle, as shown in Figure 11-10.

Figure 11-10 Adding the web OSGi bundle to the OSGi application project

4. Click **Finish** and the new OSGi application project is created in the workspace.

Create a CICS bundle project to deploy the OSGi application as follows:

1. Click **File → New → Other**. A new project dialog opens.
2. Expand **CICS Resources** and select **CICS Bundle Project** and click **Next**. A new CICS Bundle Project dialog opens as shown in Figure 11-11.

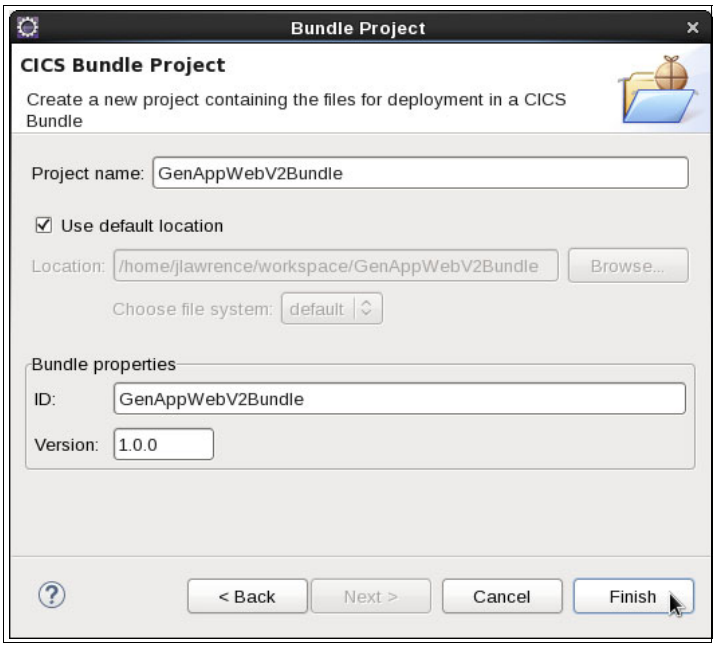


Figure 11-11 CICS Bundle Project dialog window

3. In the CICS bundle project dialog, enter a name for the new CICS bundle, such as **GenAppWebV2Bundle**.
4. Click **Finish** and the new CICS bundle project is created in the workspace and opened in a CICS Bundle Manifest editor.
5. On the **Overview** tab in the CICS Bundle Manifest editor, click **New** next to the Defined Resources list and select **OSGi Application Project Include**. An OSGi Application Project Include dialog window opens, as shown in Figure 11-12.

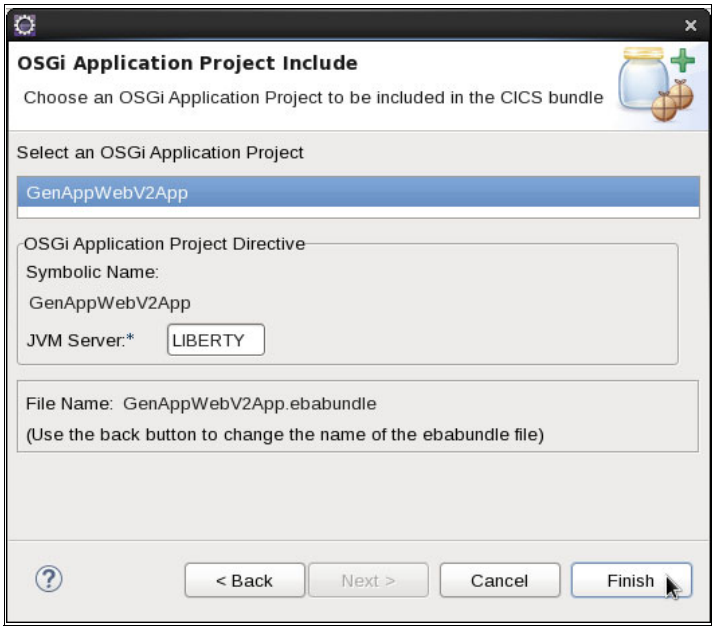


Figure 11-12 OSGI Application Project Include dialog window

6. In the **OSGi Application Project Include** dialog, select the OSGi project for the GenApp web application, **GenAppWebV2App**, and enter the name of the JVM server for the target Liberty instance.
7. Click **Finish** to confirm, and the GenApp web-enabled OSGi application project is displayed in the **Defined Resources** list in the CICS Bundle Manifest. You can now close the Bundle Manifest editor.
8. Export the CICS bundle project to the zFS by right-clicking the project and selecting **Export Bundle Project to z/OS UNIX File System**.

Your CICS bundle project containing the EBA archive is now ready for installation into CICS for testing. This requires the installation of a CICS BUNDLE resource definition that refers to the zFS location of the exported bundle in the BUNDLEDIR attribute. For more information, see Chapter 6, “Configuring a Liberty server in CICS” on page 103.

In this section, you learned how to convert a regular Java Platform, Enterprise Edition web application with an embedded utility JAR to OSGi, using a shared OSGi bundle for the common component and an EBA for the web application.

We now build on this infrastructure to extend the Java business logic with new functions, adding a new COBOL program for one new function, and using JCICS for another.

11.2 Extending the business logic application

So far in this chapter, we repackaged and redeployed the existing GenApp web application by using the OSGi tooling in Eclipse, and deployed it into a Liberty server in CICS. Now we look at a couple of options available to extend the business logic with new functions.

There are two techniques to be considered:

- Adding a COBOL program to implement the function and linking to it using JCA.
- Implementing the new function directly in Java using the JCICS API.

11.2.1 Adding new custom CICS programs

Perhaps the most obvious way to add new function to the Java business logic API is to create a new CICS program to perform the required function and invoke it by the same method as for the base application, that is via the JCA local CICS external call interface (ECI) adapter.

Note: You can use the JCICS **`Program.link()`** method to link to other CICS programs; however, we used JCA because this method was also used in the base GenAppWeb application.

This section provides an example of creating a new COBOL program to implement a function to return a list of customers for the GenApp Insurance application, using a channel and container interface and invoked using JCA. The results of invoking this program are then made available through a new method on the Java business logic interface.

Writing a new CICS COBOL program to provide the `listCustomers` function is outside of the scope of this book, but we give an overview of the process and some code snippets.

Add the new wanted method declaration to the GenAppInterface Java interface:

Example 11-3 Declare method

```
public Customer[] listCustomers(int startCustomer, int maxCount)
throws GenAppException;
```

where:

- `startCustomer` is the customer number to start the list of returned customers.
- `maxCount` is the maximum number of customers to return from the call.

These allow the client to page through many customers, for example, to display on a web page.

The returned `Customer[]` is an array of Customer objects. Each includes the customer number and first and surname, for summary purposes, not the full details for each customer.

For the new COBOL program intended to implement this function, we decided to use the channel and container interface, using `CHAR` type containers. We also decided to implement the marshalling of the container contents using hand coded Java rather than the JZOS record-generated classes.

The channel interface to the new program uses three containers as shown in Table 11-1.

Table 11-1 Interface details

Name	I/O	Content
REQUEST	Input	`StartCustomer`, `maxCount` as character string.
RESPONSE	Output (success)	Customer count, then count * (custnum, firstname, lastname) as CHAR string.
ERROR	Output (failure)	Error message as character string.

After the COBOL program that implements the new business function has been written, we know the format and content of the containers that it uses, so the Java code to implement the listCustomers() method can be completed. The skeleton method created by the Eclipse source tool is replaced with the implementation outlined below.

The method signature, which matches the Java interface declaration, was created by the Eclipse source tool:

Example 11-4 Interface declaration

```
public CustomerData[] listCustomers(int startCustomer, int maxCount)
   throws GenAppException {
```

Create a StringBuffer to build the request container contents, and insert the input parameters `startCustomer` and `maxCount` converted to strings (these also need to be padded with zeros to the correct length. The details of this are omitted for brevity).

Example 11-5 StringBuffer sample

```
StringBuffer request = new StringBuffer(12);

String cstring = Integer.toString(startCustomer);
request.append(cstring);
```

```
String cmaxstr = Integer.toString(maxCount);
request.append(cmaxstr);

// Create a channel to pass to the JCA adapter:
ECIChannelRecord channel;
try {
      channel = new ECIChannelRecord("GENAPP");
   } catch (ResourceException e) { throw new GenAppException(e); }

// Create and populate the request container in the channel.
channel.put("REQUEST", request.toString());

// Invoke the target program passing the channel as input.
invokeChannelProgram(ListCustomersPgm, channel);
```

If an error container has been returned, throw an exception with the contents of the error message.

Example 11-6 Sample to handle exception

```
if (channel.containsKey("ERROR")) // Error container exists.
    {
      throw new GenAppException((String) channel.get("ERROR"));
    }
```

Otherwise, there is no error container so the RESPONSE container is valid. Extract the contents into a String (Example 11-7).

Example 11-7 Extract response into a string

```
String response = (String) channel.get("RESPONSE");
```

Extract the customer count from the first two characters of the response. Then, iterate to extract each customer's detail from the response, and add to a new array `CustomerData[]` of the required size. See Example 11-8.

Example 11-8 Sample to extract customer count

```
int ccount = Integer.parseInt(response.substring(0, 2));
CustomerData[] carray = new CustomerData[ccount];

int coffset = 2; // Start offset for customer data.
for (int cindex = 0; cindex < ccount; cindex++) {
   CustomerData cdata = new CustomerData();

   int cnumber = Integer.parseInt(response.substring(coffset, coffset + 10));
   cdata.setCustomerNumber(cnumber);
   cdata.setFirstName(response.substring(coffset + 10, coffset + 20));
   cdata.setLastName(response.substring(coffset + 20, coffset + 30));

   carray[cindex] = cdata; // Store new CustomerData in array.
   coffset += 30; // Shift forward by length of each customer data.
}
```

Finally, return the completed array of CustomerData (Example 11-9).

Example 11-9 Return array

```
return carray; // Return full CustomerData[] to caller.
```

The private method `invokeChannelProgram()`, which uses JCA to invoke the target CICS COBOL program, passing the channel containing the request container, is outlined in Example 11-10. The input parameters are the name of the target CICS program, and the channel, which has been created by the caller.

Example 11-10 invokeChannelProgram() method

```
private void invokeChannelProgram(String program, ECIChannelRecord channel)
      throws GenAppException {
```

Create a new `ECIInteractionSpec` to define the JCA interaction to be performed, and set the name of the target program. The interaction will be a synchronous send and receive.

Example 11-11 ECIInteractionSpec sample

```
ECIInteractionSpec eSpec = new ECIInteractionSpec();
eSpec.setFunctionName(program); // Name of target COBOL program
eSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);

// Create a new ECI Interaction to execute the call
Interaction eciInt = eciConn.createInteraction();
```

Execute the interaction using the InteractionSpec, and passing the channel as input. The same channel object is used to receive the response channel content from the CICS program call (Example 11-12).

Example 11-12 Execute interaction

```
eciInt.execute(eSpec, channel, channel);
eciInt.close();
```

The Java implementations outlined above have been condensed from the complete code, which is available in the additional materials supplied with this book at the following URL:

`http://www.redbooks.ibm.com/abstracts/sg248335.html`

These two new Java methods, `listCustomers()` and `invokeChannelProgram()`, when added to the existing business logic interface class, complete the implementation of the new function to provide a list of known customers. This function can then be used by the GenApp web front end to display a list of customers on a web page, or exposed as a web service (such as a JSON REST web service) as described in Chapter 10, “Creating an integration logic application” on page 179.

11.2.2 Using the JCICS API

The Liberty server in CICS is fully integrated with its host CICS environment and this allows you to use the native CICS Java programming capability, including JCICS. In this section, we show how JCICS can be used to extend the functions of the Java business logic interface very quickly and easily.

To demonstrate this, we choose to add another new method on the GenAppBli interface, in this case to obtain a list of all the policies for a given customer.

Add the method declaration shown in Example 11-13 to the GenAppInterface Java interface.

Example 11-13 Declare method

```
public PolicyData[] listPolicies(int custNumber) throws GenAppException;
```

where

- `custNumber` is the customer number for which the list of policies is required.

The returned `PolicyData[]` is an array of PolicyData objects. Each contains the policy number and an enumerated type indicating the type of policy represented, for example, house or car.

Saving the new method declaration in the Java interface definition causes the implementation class to be invalidated because it no longer fully implements its declared interface. The Eclipse coding assist tools can be used to create a new skeleton implementation method.

Edit the implementation class `GenAppJCAImpl` and position the cursor at an appropriate point in the source where the new method skeleton should be added.

Right-click and select **Source → Override/Implement Methods**. A dialog is displayed as shown in Figure 11-13. After checking details, click **OK** to finish and add the new method.

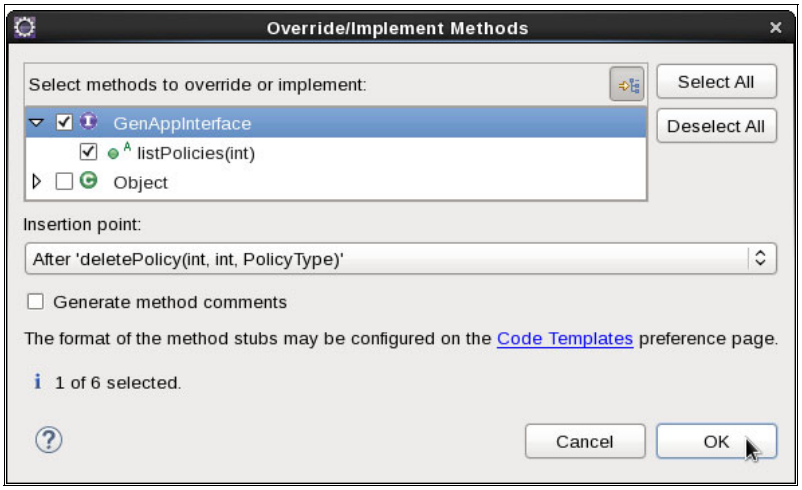


Figure 11-13 Override/Implement Methods dialog after adding listPolicies()

Now the actual implementation for the method has to be added. The following is the condensed and annotated Java source code for the `listPolicies()` method. The method signature matches the declaration in the interface, and the final statement builds the `PolicyData` array for return (Example 11-14).

Example 11-14 Declare a listPolicies() method

```
public PolicyData[] listPolicies(int custNumber) throws GenAppException {
...
return policyList.toArray(new PolicyData[0]);
}
```

The main JCICS class used to implement `listPolicies()` is the KSDS class, which is used to browse the "KSDSPOLY" VSAM file to obtain a list of policies that match the input customer number. The key length for this file is 21 so we declare a byte array of size 21 to contain the key (Example 11-15).

Example 11-15 Declare a byte array for KSDS class

```
KSDS ksdspoly = new KSDS();
ksdspoly.setName("KSDSPOLY");
byte[] key = new byte[21]; // Null value array, key length = 21.
```

Create a new `ArrayList` to hold the list of PolicyData objects as they are created from matching file records. Declare and start a browse of the KSDS file, starting with the low valued key so that all records are seen, and catching any exceptions returned from JCICS (Example 11-16).

Example 11-16 Create a new ArrayList

```
ArrayList<PolicyData> policyList = new ArrayList<PolicyData>();
KeyedFileBrowse policyBrowse;
try {
   policyBrowse = ksdspoly.startBrowse(key); // Start with low values.
...
  } catch (CicsException e) { throw new GenAppException(e); }
```

Create `RecordHolder` and `KeyHolder` objects to use as holders for the returned file record and key respectively during the browse, as shown in Example 11-17.

Example 11-17 Create RecordHolder and KeyHolder objects

```
RecordHolder recordholder = new RecordHolder();
KeyHolder keyholder = new KeyHolder(key);
```

Loop indefinitely (until an exception is thrown), obtaining the next file record and key from the browse, and terminate when the end of file is encountered (Example 11-18).

Example 11-18 Create a loop

```
while (true) { // Browse until end of file (EndOfFileException)
try {
policyBrowse.next(recordholder, keyholder);
...
  } catch (EndOfFileException endfile) {
    break; // End browse loop normally - expected.
  }
} // while()
policyBrowse.end(); // End the file browse normally at end of file.
```

If this point in the method is reached, no exception was thrown, so the next record was found. Extract the key from its holder and convert it to a string for subsequent processing, as shown in Example 11-19.

Example 11-19 Extract keys

```
key = keyholder.getValue();
String keystring = new String(key, ccsid);
```

```
// If the key matches the requested customer number,
// process the key to extract policy number & type.
PolicyData pdata = null;
```

If the key represents a file record for the requested customer, check the type of policy to see if it is one of the types that we are interested in, that is a house or a car policy. If so, create a `PolicyData` object of the appropriate type. If `pdata` is created, extract the policy number from the key string and store it in `pdata`. Then, add it to the list of policies in `policyList`. Continue browsing to inspect the next file record (Example 11-20).

Example 11-20 Create a PolicyData object and extract policy data

```
if (Integer.parseInt(keystring.substring(1, 11)) == custNumber) {

switch (keystring.charAt(0)) {
   case 'M':
      pdata = new PolicyData(PolicyType.Car);
      break;
   case 'H':
      pdata = new PolicyData(PolicyType.House);
      break;
   default: // other
   break; // Just ignore other policy types
} // switch()

   if (pdata != null) { // Correct customer *and* accepted type.
      pdata.setPolicyNumber(Integer.parseInt(keystring.substring(11)));
      policyList.add(pdata);
   }
}
```

When `EndOfFileException` is caught and terminates the loop normally, this is an expected exception (Example 11-21). Any other exception also terminates the loop, but it will be an error resulting in an exception to the caller.

Example 11-21 Exiting the loop

```
   } catch (EndOfFileException endfile) {
   break; // End browse loop normally - expected.
   }
} // while()
policyBrowse.end(); // End the file browse normally at end of file.
} catch (CicsException e) { throw new GenAppException(e); }
```

Finally, convert the `ArrayList` to an array of the correct type, `PolicyData[]`, and return it to the caller, as shown in Example 11-22.

Example 11-22 Convert array and return to caller

```
return policyList.toArray(new PolicyData[0]);
```

The single short Java method shown here is all that is needed to implement the new business logic function using JCICS to obtain a list of active policies for a given customer. This function can then be used by a dynamic web page or exposed as a web service in the same way as for the `listCustomers()` function.

The full source is available with the additional materials for this book at.

http://www.redbooks.ibm.com/abstracts/sg248335.html

11.3 Summary

This chapter has shown how to convert an existing Java Platform, Enterprise Edition application to exploit OSGi and provided examples of two different approaches to extending the business logic interface for a CICS Java Platform, Enterprise Edition application: JCA with an auxiliary COBOL program, versus JCICS. Consider the following factors when deciding between these the following alternatives:

- Portability
- Implementation
- Performance

The main advantages and disadvantages of these approaches, which are equivalent from a functional perspective, include:

- Using the JCICS APIs to extend the business logic of an application prevents the application from being deployed in a non CICS environment. Therefore, if you ever want to port the application to another server, these would have to be changed to use a different implementation such as COBOL via JCA as for the listCustomers() method.
- The JCA/COBOL approach requires a new additional COBOL program to be written, compiled, and installed in CICS alongside existing programs.
- The JCICS implementation requires (in this case) much less code because it is a single short Java method as opposed to a COBOL program plus a more extended Java method.
- The JCA/COBOL is likely to have better performance because it requires only a single interaction across the Liberty CICS interface, versus many interactions (in this case) for the JCICS performing the file browse. However, the performance gains have not been measured during this project.

Part 4

Reference

In this part, we provide reference information.

Part 4 contains Chapter 12, “Troubleshooting” on page 211.

12

Troubleshooting

Problems can occur when running Java web applications inside of CICS Transaction Server (CICS TS). Additionally, with several different components running in tandem, the amount of documentation to review can be confusing. When you have CICS TS, the Liberty Profile, and the Java virtual machine (JVM) running and gathering documentation, where should you begin?

If you encounter problems, you can use the diagnostic tools provided by CICS, the JVM, and the Liberty Profile to determine the cause of the problem. CICS provides statistics, messages, and tracing to help you diagnose problems related to your Java applications. The IBM Java software development kit (SDK) diagnostic tools give more detailed information about the internals of the JVM, including the garbage collection and just-in-time (JIT) daemon threads running in a JVM. The Liberty Profile tools provide information about its inner workings.

This chapter discusses some of the diagnostics and tools available that can be obtained from your Liberty JVM server environment and applications, mainly focusing on diagnosing problems within a CICS region.

This chapter describes the following topics:

- Diagnostics
- Messages and log files
- Dumps and trace
- Additional documentation
- Debugging tools
- JVM Health Center
- Liberty debug tools
- Symptoms and user actions

12.1 Diagnostics

In addition to the traditional CICS administration facilities, such as the CICS EXEC Master Terminal (Customer Information Control System (CEMT)) transaction, the system programming interface (SPI), transaction monitoring, and statistics, there are a number of unique interfaces and diagnostic logs that are specific to the CICS Liberty JVM environment that you can use for problem determination with Java web applications.

12.2 Messages and log files

Several logs can be used by the JVM server environment on CICS. These logs vary from z/OS file system (zFS) files to traditional DD destinations in the CICS job. The zFS files are located, by default, in the working directory of the JVM server.

12.2.1 CICS logs

Most errors that occur in a JVM server result in a message being issued to the CICS `MSGUSR DD` log. This is one of the first places to look for any CICS related errors, warnings, or abends that might have occurred.

The `SYSPRINT DD` log is the stdout destination normally used by C-language routines. In a CICS region in particular, it might contain errors from the initialization of the JVM server. If omitted, individual `SYSPRINT DD` logs are dynamically allocated for each JVM server instance, in the pattern SYSnnnnn.

12.2.2 Java logs

When the JVM server has initialized, output from the System.out stream is written to the `STDOUT` destination set by the `STDOUT` option within the JVM server profile. If the file exists, output is appended to the end of the file.

If you specify the USEROUTPUTCLASS option in a JVM profile, the Java class named on that option handles the System.out requests instead. CICS supplies a sample class, `com.ibm.cics.samples.SJMergedStream`, which can be used to redirect output from stdout back to the `MSGUSR` log.

The `STDERR` destination is the primary location to which Java exceptions and stack traces are written. These exceptions are typically a result of errors in Java applications or components.

The location is set by using the `STDERR` option in the JVM server profile. If the file exists, output is appended to the end of the file.

For JVM servers, the default file name is `<applid>.<jvmserver>.dfhjvmout`, which means that a unique file is created for each JVM server and CICS region. In addition, further unique files can be made by using the `&APPLID; &JVMSERVER, &DATE;` and `&TIME;` symbols.

12.2.3 Liberty Profile server logs

Three primary log files are written by the Liberty Profile server:

- The `messages.log` file contains all messages except trace messages that are written or captured by the logging component. All messages that are written to this file contain more information, such as the message time stamp and the ID of the thread that wrote the message. This file does not contain messages that are written directly by the JVM process. Each message has a unique message identifier, and includes an explanation of the problem, and details of any action that you can take to resolve the problem. Liberty profile system messages are logged from various sources, including application server components and applications.
- The `console.log` file contains the redirected standard output and standard error from the JVM process. The console output contains major events and errors if you use the default consoleLogLevel configuration. The console output also contains any messages that are written to the System.out and System.err streams if you use the default copySystemStreams configuration. The console output always contains messages that are written directly by the JVM process, such as -verbose:gc output. By specifying the following parameter within the JVM profile, you can control which messages that Liberty writes to the stdout file:

  ```
  com.ibm.ws.logging.console.log.level={INFO|AUDIT|WARNING|ERROR|OFF}
  ```

- The `trace.log` file contains all the messages that are written or captured by the product. This file is created only if you enable additional trace. This is usually at the request of Level 2 Support. This file does not contain messages that are written directly by the JVM process. Within the JVM profile, selectively enable this trace and specify the file name by using the following parameters:

  ```
  com.ibm.ws.logging.trace.specification=*=all=enabled
  com.ibm.ws.logging.trace.file.name=trace.log
  ```

12.3 Dumps and trace

Several sources of detailed diagnostic information can be collected from a JVM server. This includes several types of dumps as well as tracing that can be activated in each of the components.

12.3.1 Java dumps and trace

Three types of Java dumps can be produced by the JVM.

The Javadump is often referred to as a *Javacore* or *thread dump* in some JVMs. This dump is in a human-readable format. It is produced by default when the JVM terminates unexpectedly because of an operating system signal, an `OutOfMemoryError` exception, or when the user enters a reserved key combination (for example, Ctrl-Break on Microsoft Windows). You can also generate a Javadump by calling a method from the Dump API, for example `com.ibm.jvm.Dump.JavaDump()`, from within the application. A Javadump summarizes the state of the JVM at the instant the signal occurred. Much of the content of the Javadump is specific to the IBM JVM.

The JVM can generate a *Heapdump*, which is a snapshot of JVM memory. It shows the live objects on the heap along with references between objects. It is used to determine memory usage patterns and memory leak suspects. A Heapdump can be taken at the request of the user by calling `com.ibm.jvm.Dump.HeapDump()` from inside the application or when the JVM ends because of an `OutOfMemoryError` exception. You can specify finer control of the timing of a Heapdump with the `-Xdump:heap` option. For example, you can request a Heapdump after a certain number of full garbage collections have occurred. The default Heapdump format (.phd files) is not human-readable and must be processed by using available tools, such as Memory Analyzer.

System dumps are platform-specific files that contain information about the active processes, threads, and system memory. System dumps are usually large. By default, system dumps are produced by the JVM only when the JVM fails unexpectedly because of a general protection fault (GPF) or a major JVM or system error. You can also request a system dump by using the Dump API. You can call the `com.ibm.jvm.Dump.SystemDump()` method from your application. You can use the `-Xdump:system` option to produce system dumps when other events occur.

IBM JVM tracing allows execution points in the Java code and the internal JVM code to be logged. The `-Xtrace` option allows the number and areas of trace points to be controlled, as well as the size and nature of the trace buffers maintained. The internal trace buffers at a time of failure are also available in a system dump and tools are available to extract them from a system dump. Generally, trace data is written to a file in an encoded format and then a trace formatter converts the data into a readable format. However, if small amounts of trace are to be produced and performance is not an issue, trace can be routed to `STDERR` and will be preformatted.

12.3.2 Liberty Profile server dumps and trace

Within a running server, there might be occasions when a dump is needed for further analysis. Within the Liberty Profile server, use the server script command to capture status information for a Liberty profile server.

The server dump command is useful for problem diagnosis of a Liberty profile server, because the result file contains server configuration, log information, and details of the deployed applications in the `workarea` directory. This command can be applied to either a running or a stopped server.

The `server javadump` command is useful for diagnosing problems at the JVM level, such as hung threads, deadlocks, excessive processing, excessive memory consumption, memory leaks, and defects in the virtual machine. The server must be running to use this command. Each dump type creates a file, but not all dump types are supported by all virtual machines. By default, the files are written to the `${server.output.dir}` directory.

For a running server, the following information is also included:

- State of each Open Service Gateway initiative (OSGi) bundle in the server
- Wiring information for each OSGi bundle in the server
- Component list that is managed by the Service Component Runtime (SCR) environment
- Detailed information about each component from SCR
- Configuration administration data of each OSGi bundle
- Information about registered OSGi services
- Runtime environment settings, such as JVM, heap size, operating system, thread information, and network status

Within a CICS environment, use the `wlpenv` script to request these dumps. We discuss this script in 12.7.1, “Using the wlpenv script to run Liberty commands” on page 224.

12.3.3 CICS dumps and trace

When errors occur, transactions abend and messages are written to the appropriate log. Messages related to Java are in the format `DFHSJxxxx`. You can also turn on tracing to produce more diagnostic information. When the first JVM is started in a CICS region after initialization, CICS issues message `DFHSJ0207`, which shows the version of Java that is being used:

```
DFHSJ0207 date time applid JVMSERVER jvmserver is running Java version version
```

In addition to the logging produced by Java, CICS, and the Liberty server, CICS also provides some standard trace points in the SJ (JVM) and AP domains. These trace points trace the actions that CICS takes in setting up and managing JVM servers, installing and running applications, and making JCICS calls. JVM server tracing does not use auxiliary trace 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 and created within the `WORK_DIR` directory when you enable the JVMSERVER resource. The default file name has the format `applid.jvmserver.dfhjvmtrc`. 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.

This file does not wrap so you must manage its size and the number of unique entries in zFS. You can manage the number of files by setting the LOG_FILES_MAX option in the JVM profile to control the number of old trace files that are retained on the JVM server startup. If you disable the JVMSERVER resource, you can delete old entries or rename them if you want to retain the information separately. When you enable the JVMSERVER resource, CICS appends the new entries to the trace file if it already exists and creates a file.

You can activate JVM server tracing by turning on `SJ` and `AP` component tracing by using either the CETR transaction or the SIT STNTRxx parameter. The SJ component traces exceptions and processing in the SJ domain to the internal trace table. The AP component traces the installation of OSGi bundles in the OSGi framework.

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 produce 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.

12.4 Additional documentation

In addition to the above sets of documentation, there are other ways that are available to help monitor and tune the JVM server. CICS statistics and garbage collection can be used to properly tune the JVM.

12.4.1 CICS statistics

CICS collects statistics for the JVM servers and for Java programs that run in the JVMs. You can use these statistics to manage and tune the Java workloads that are running in your CICS regions. The JVM server statistics tell you about the activity of the JVM that is used by a particular JVM server. The JVM program statistics tell you about Java programs that run in the JVM servers. You can use the `CEMT` transaction to inquire on the JVMSERVER. You can also access JVMSERVER statistics online by using the `EXEC CICS EXTRACT STATISTICS JVMSERVER()` command, by using the STAT transaction, or through the CICS Explorer `JVM Servers` view. JVMSERVER statistics are mapped by `DFHSJSDS DSECT`.

12.4.2 Garbage collection data

Garbage collection and heap expansion are an essential part of the operation of a JVM. When a JVM runs out of space in the storage heap and is unable to allocate any more objects, an allocation failure occurs and a garbage collection is triggered. To help diagnose problems in the garbage collection, add the `-verbose:gc` option to the JVM profile. This produces output in XML format that can be used to analyze problems in the Garbage Collector itself or problems in the design of user applications.

12.5 Debugging tools

In this section, we discuss the tools that are available that allow you to interactively debug a Java program running in CICS.

12.5.1 Execution diagnostic facility

CICS provides the Command Execution Diagnostic Facility (CEDF) as a basic debugging tool.

You can use the execution diagnostic facility (EDF) to test an application, which can be used to test application programs online without modifying the program or the program-preparation procedure. When used, CEDF intercepts a transaction when each program starts and finishes, before and after any `EXEC CICS or EXEC` SQL commands are executed, and when the task ends. Using CEDF gives you a chance to interact with your transaction and view the parameters and responses that are passed back and forth as well as storage areas when each command is executed.

When used with a JCICS program, CEDF intercepts the CICS commands that the Java Native Interface (JNI) programs start, which provide the interface between JCICS classes and CICS. There might not be an obvious relationship between the JCICS class method and the CICS command. For example, the HelloCICSWorld sample uses the `Task.out.println()` method to send a message to the user terminal. The JNI program starts the `SEND` command to write to the panel.

To use EDF on the same terminal as the transaction you are testing, follow these steps:

1. Type `CEDF` and press Enter to turn it on.

   This results in the following message being printed to the terminal:

   ```
   THIS TERMINAL: EDF MODE ON
   ```

2. Type `CEDF,,OFF` and press Enter to turn it off. (notice the double commas)

   This results in the following message being printed to the terminal:

   ```
   THIS TERMINAL: EDF MODE OFF
   ```

To use EDF on a different terminal than where the transaction is located:

1. Type `CEDF,####` where `####` is the other terminal ID and press Enter to turn it on.
2. Type `CEDF,####,OFF` and press Enter to turn it off.

12.5.2 Java debugger

When you use the CICS Explorer SDK to develop your web applications and then deploy them in the Liberty JVM server, if a failure occurs in the application it can be difficult to debug because there is no development environment there. This is where remote debugging comes in handy. The Java Platform Debugger Architecture (JPDA) is a useful tool for debugging applications that are failing in a deployment environment remotely. The JPDA 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 JPDA connects your running application with your development environment (such as Eclipse).

Java provides this debugger using a listener mechanism. The application that you want to debug attaches a socket to itself where it listens for any debug instructions. The JPDA then uses this socket to connect to the JVM. To debug an application remotely, it must be running in debug mode. This requires that the JVM is started with the following debug argument:

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=port
```

In a CICS environment, this means adding the `–agentlib:jdwp` option to the JVM profile for the JVM server and using the CICS Explorer SDK to debug your applications.

To configure Eclipse to remotely debug CICS web applications, follow these steps:

1. Ensure that the Liberty JVM server is either DISABLED or not installed in your CICS region.
2. Add the following option to your JVM profile to attach the remote debugger:

   ```
   -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=53107
   ```

3. Set the `address=` option to an available port number that the CICS region user ID is authorized to open. This is the port that Eclipse will connect to during the debug session.
4. Install your JVMSERVER resource definition (or set it to Enable).
5. Install the CICS BUNDLE resource definition for your CICS web application.
6. In the CICS Explorer SDK, switch to the debug perspective by going to **Window** → **Open Perspective** → **Other** → **Debug** → **OK**.
7. Select **Run** → **Debug Configurations** to set up the JPDA configuration.
8. Right-click **Remote Java Application** and click **New**.
9. Give the configuration a meaningful name.
10. Select the **Project** you want to debug.

11. Select **Socket Attach** to attach the debugger to the application.
12. Specify the host name of your z/OS system and the port from step 2.
13. Click **Debug** to start the session. See Figure 12-1.

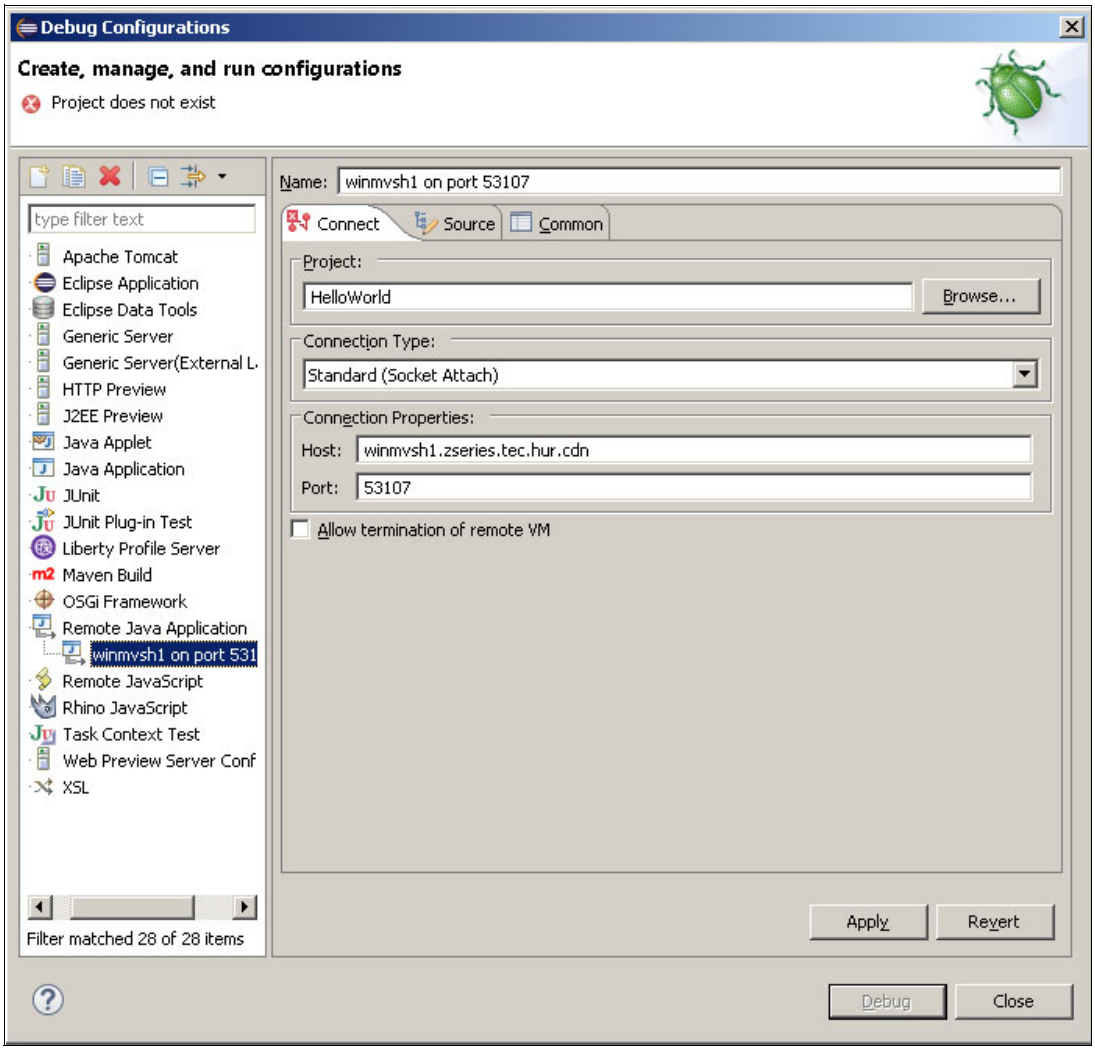


Figure 12-1 Debug configurations

Because the JVM is running in server mode, it must be started first in debug mode. Then, the debugger is started. Otherwise, you would get a “Failed to connect to remote VM” message.

14. Switch to the web perspective and go to your application class that you need to debug.
15. Set your breakpoints by double-clicking in the margin.
16. Save your changes and export the CICS Bundle to zFS.
17. Install CICS BUNDLE resource to deploy it to the Liberty JVM.
18. In your web browser, enter the URL for the web application.
19. When the first breakpoint hits, the Eclipse perspective switches to the debug perspective. You can now step through your application code.

12.6 JVM Health Center

The IBM Health Center is a diagnostic tool for monitoring the status of a JVM and can be used with any IBM JVM, including the CICS Liberty server. It provides recommendations and analysis that help you improve the performance and efficiency of your application. In particular, it is good for deep dives into issues with performance, high CPU, and monitor contention. In this section, we look at the Health Center and how to integrate it with CICS Explorer for monitoring.

12.6.1 What the Health Center is

The Health Center has a low runtime overhead and can be used to profile and tune your CICS Java application. The Health Center is a monitoring tool that consists of two components, which are installed separately:

- The agent, which is shipped with the IBM JVM
- The client, which is an Eclipse-based GUI perspective

Figure 12-2 shows where the Health Center agent and client are installed when using CICS.

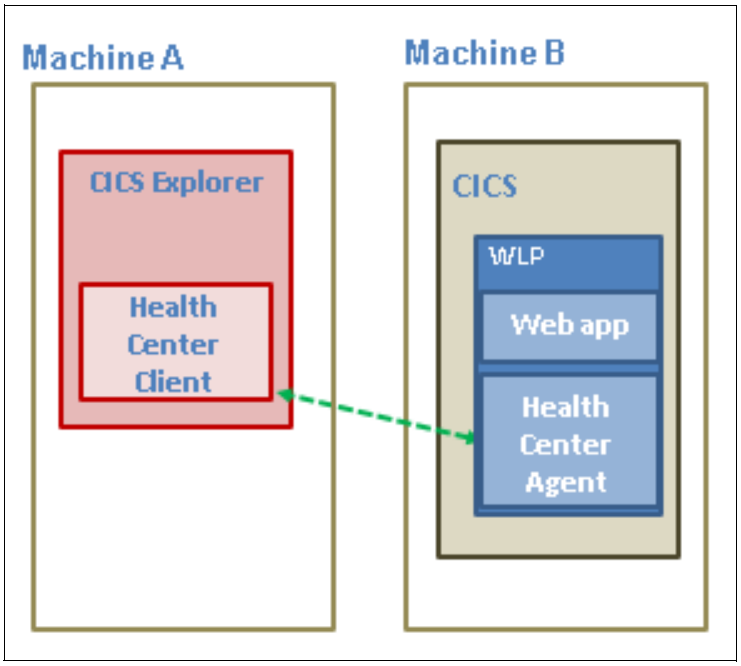


Figure 12-2 Location of Health Center agent and client

The Health Center agent is a JVMTI native library that must be enabled in the JVM (by default, it is `off`). It monitors the Java applications and collects data from the running application. The agent can be enabled in three modes:

- Socket communication, by specifying `–Xhealthcenter:port=xxxx`: The agent connects to a client through a socket. It communicates either in IIOP (default) or JRMP.
- Headless, by specifying `–Xhealthcenter:level=headless`. Instead of opening a socket, it stores the agent data to a zFS file system in an HCD file.
- Late attach, where no data is collected until the client connects.

It is always a preferred practice to update the agent because it is independent of the JVM and the latest version has bug fixes, additional collection, better options, and so on.

The Health Center client is used to interpret the data collected by the agent. It provides recommendations to improve the performance of the monitored application. The Health Center client has several different views, each providing data on a different aspect of the JVM. Following are some of the available views:

- Classes: Provides information about classes being loaded
- CPU: Provides processor usage for the application and the system on which it is running
- Environment: Provides details of the configuration and system of the monitored application
- Garbage Collection: Provides information about the Java heap and pause times
- I/O: Provides information about I/O activities that take place
- Locking: Provides information about contention on inflated locks
- Method Trace: Provides information about method use over time
- Native Memory: Provides information about the native memory usage
- Profiling: Provides a sampling profile of Java methods including call paths
- Threads: Provides information about the live threads of the monitored JVM

12.6.2 Install and Configure the Health Center and CICS Explorer

Because the Health Center client is Eclipse-based, it can be installed and utilized by using the CICS Explorer. We do not look at how to monitor a CICS web application using the Health Center and CICS Explorer.

To monitor a CICS Liberty JVM using the Health Center, follow these steps:

1. Install the Health Center client

   The easiest way to install the Health Center client is by using the Eclipse Marketplace. In your Eclipse client, follow these steps:

   a. Select **Help** → **Eclipse Marketplace**.

   b. In the Search tab, enter `Health Center` in the “Find” field.

   c. Select **IBM Monitoring and Diagnostic Tools for Java - Health Center** from the list and click **Install**.

   d. Confirm that the Health Center and Health Center Core will be installed and click **Confirm**.

   e. Read and accept the license terms and then click **Finish**.

2. Enable the Health Center agent in the JVM:

   a. Ensure that the JVMSERVER resource ID is DISABLED.

   b. Add the following arguments to the JVM profile:

```
-Xhealthcenter:port=53108
-Dcom.ibm.java.diagnostics.healthcenter.agent.iiop.port=53109
```

Note: Information that will help you follows:

- The default port for the Health Center agent is 1972. You could just add `–Xhealthcenter` to use the default port 1972.
- The secondary port is used for the JVM over RMI-IIOP connection. The default is to choose this port randomly. It can be configured by specifying the iiop port.
- If you use a non IBM JDK to run the Eclipse client, there can be a conflict in protocols. Set the following option instead so that the Liberty server uses RMI-JRMP to connect:

  `-Xhealthcenter:transport=jrmp,port=xxxxx`

c. Enable or start the JVM server

The JVM server `jvmerr` file will show the following messages indicating the Health Center agent was successfully started:

```
Nov 15, 2015 6:55:11 AM
com.ibm.java.diagnostics.healthcenter.agent.mbean.HCLaunchMBean <init>
INFO: Agent version "3.0.4.20150622"
Nov 15, 2015 6:55:12 AM
com.ibm.java.diagnostics.healthcenter.agent.mbean.HCLaunchMBean
createJMXCon-nector
INFO: IIOP will be listening on port 53109
Nov 15, 2015 6:55:12 AM
com.ibm.java.diagnostics.healthcenter.agent.mbean.HCLaunchMBean startAgent
INFO: Health Center agent started on port 53108.
```

3. Connect the Health Center Client to the JVM:

   a. In the CICS Explorer SDK, go to the Health Center perspective by going to **Window** → **Open Perspective** → **Other** → **Health Center Environment** → **OK**.

   b. Click **File** → **New** connection to open the Health Center Connection wizard. Click **Next**.

c. In the JMX connector tab, enter the z/OS host name and port number that the Health Center agent is listening on as specified in the JVM profile (`-Xhealthcenter:port=`).

d. Click **Next**. See Figure 12-3.

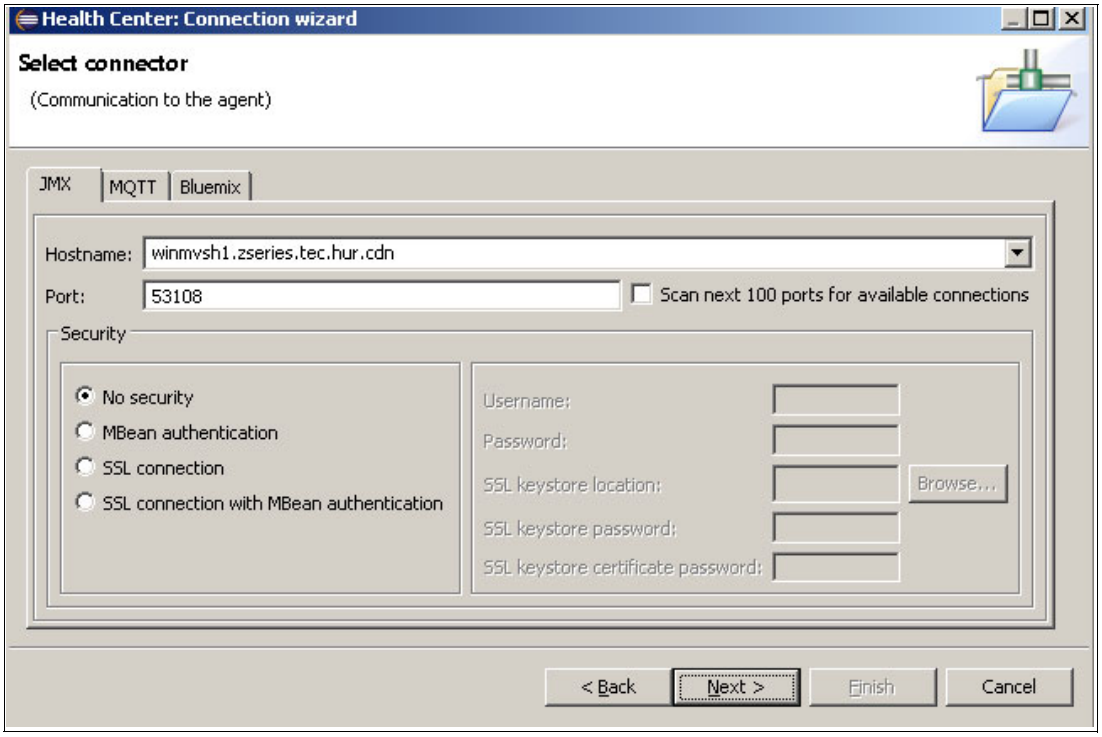


Figure 12-3 Connecting the Health Center client

4. Select the detected agent and click **Finish**. See Figure 12-4 on page 223.

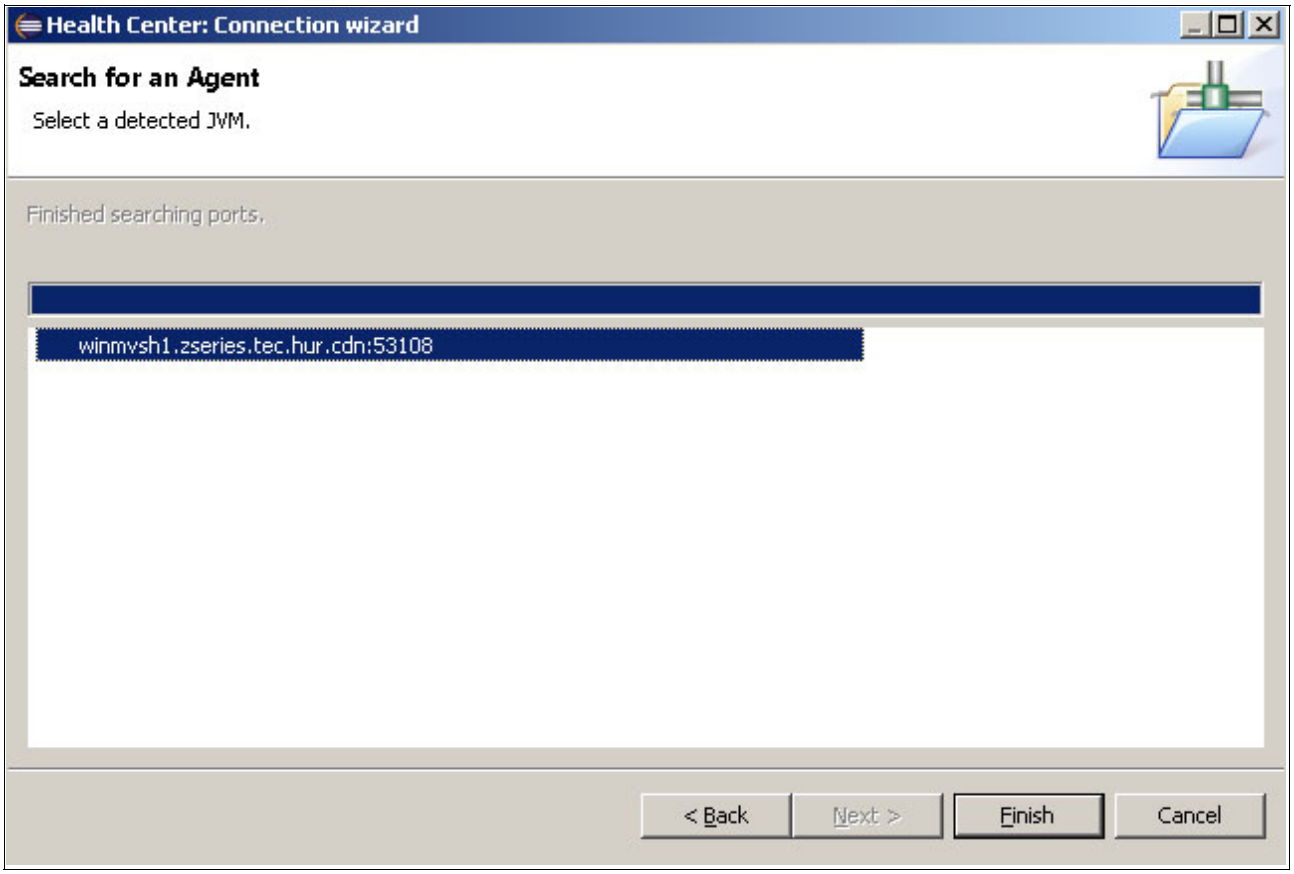


Figure 12-4 Selecting an agent

You are now successfully connected. The Configuration view is now displayed, providing useful information about all the Java parameters, dump options, and classpath settings in the JVM. By using the Status view, you can access the different functions of the Health Center.

For more information about how some of these features can be used to analyze the CICS JVM server, see Integrating IBM Health Center and CICS Explorer at the following site:

`https://ibm.biz/BdHfmw`

12.7 Liberty debug tools

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, use the **`wlpenv`** script as a wrapper to set the required environment for running the Liberty commands. This script is created and updated every time the Liberty JVM server is enabled after the JVM profile has been successfully parsed. This script is unique to each JVM server in each CICS region. Therefore, it is created in the `WORK_DIR` directory as specified in the JVM profile.

12.7.1 Using the wlpenv script to run Liberty commands

To run the **wlpenv** script in the UNIX System Services shell, change to the `WORK_DIR` directory, and run the script with the Liberty profile command as an argument. For example:

```
./wlpenv productInfo version
```

You can verify the integrity of the Liberty profile installation after you install CICS or apply service by using the **`productInfo`** script.

The following examples demonstrate the correct syntax for using the Liberty command scripts and parameters using the CICS **`wlpenv`** script:

- compare: Allows you to compare APAR fixes installed in the current installation with a different version of the Liberty profile:

  ```
  ./wlpenv productInfo compare --target=C:\wlp\newInstall\wlp
  ./wlpenv productInfo compare --target=C:\wlp\newInstall.jar
  —-output=C:\wlp\compareOutput.txt
  ./wlpenv productInfo compare
  —-apars=com.ibm.ws.apar.PM39074,com.ibm.ws.apar.PM39075
  ```

- featureInfo: Lists all features installed on the current Liberty profile server:

  ```
  ./wlpenv productInfo featureInfo --output=c:\wlp\featureListOutput.txt
  ```

- validate: Validates the Liberty profile server:

  ```
  ./wlpenv productInfo validate
  ```

- version: Displays the product name and version:

  ```
  ./wlpenv productInfo version
  ```

- viewLicenseAgreement and viewLicenseInfo: Displays the license agreement and information for the Liberty profile edition that is installed:

  ```
  ./wlpenv productInfo viewLicenseAgreement
  ./wlpenv productInfo viewLicenseInfo
  ```

- help: Displays help information for the specific action:

  ```
  ./wlpenv productInfo help compare
  ```

- server: Captures a snapshot of the server or JVM status:

  ```
  ./wlpenv server dump --archive=package_file_name.dump.pax --include=heap
  ./wlpenv server javadump —-include=heap
  ```

12.8 Symptoms and user actions

This section describes the basic failure scenarios that might occur during JVM server set up, the deployment of user applications, or at application runtime.

12.8.1 Unable to start the Liberty JVM server

If you are unable to start a Liberty JVM server, check that your setup is correct. Use the CICS messages and `SYSPRINT` data for any errors in the Liberty `messages.log` file that is below `WLP_OUTPUT_DIR` to determine what might be causing the problem.

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.

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

If you receive a CWWK0221E error message in dfhjvmerr, check that you set the right values for the host name and port number in the JVM profile and server.xml. 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.

12.8.3 CICS CPU usage is increased after a Liberty JVM server is enabled

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 to disable application monitoring:

```
<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.

12.8.4 Application not available

You copy a WAR file into the dropins directory but your application is not available. Check the Liberty messages.log file for error messages. If you receive the CWWKZ0013E error message, you already have a web application running in the Liberty JVM server with the same name. To fix this problem, change the name of the web application and deploy to the dropins directory.

12.8.5 Web application is not requesting authentication

You configured security, but the web application is not requesting authentication. Follow these steps:

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 cicsts:security-1.0 is specified. If CICS security is switched off, check that you specified a basic user registry to authenticate application users.
3. Check that the server.xml file 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 within the <application> element in server.xml or installedApps.xml. The default security-role added by CICS for a local role mapping is 'cicsAllAuthenticated'.

12.8.6 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, ensure that the user ID has a valid password and is authorized to run the transaction.
2. If the application is returning 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.

12.8.7 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 `DFHSJxxxx` 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, ensure that the URIMAP is using the correct transaction.
3. If the URIMAP is using the correct transaction, ensure 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.

12.8.8 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 `DFHSJxxxx` messages for additional information.
2. Ensure 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 error messages in the log describe why the request was purged.

12.8.9 The web application is returning exceptions

The web application is returning exceptions in the web browser. For example, the application is returning 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.

12.8.10 Error message CWWKB0109E

If the Liberty server fails to shut down cleanly, the next Liberty JVM server with `WLP_ZOS_PLATFORM=TRUE` writes the error message `CWWKB0109E` to the `messages.log` file. You do not need to fix this error and it can be ignored.

12.8.11 JVM server set-up failures

If you are unable to start a JVM server, check the available documentation for an indication of the problem. The CICS message log will provide the following message indicating the reason for the failure:

`DFHSJ0210` *date time applid* `An attempt to start a JVM for the JVMSERVER resource jvmserver has failed. Reason code: reason`

One of the following reasons will be given:

- `ATTACH_JNI_THREAD_FAILED`

  An attempt to attach a thread and run setup or termination classes in the JVM server has failed. Contact IBM Support for further assistance.

- `CHANGE_DIRECTORY_CALL_FAILED`

  An attempt to change the zFS working directory has failed. Check that the CICS job has read, write, and execute access to the directory specified by WORK_DIR in the JVM profile.

- `CREATE_JVM_FAILED`

  An attempt to create a JVM has failed. Additional diagnostic messages have been output to the standard error stream. Use these messages to determine the cause of the problem.

- `ENCLAVE_INIT_FAILED`

  The IBM Language Environment® enclave failed to initialize successfully. Check that the SDFJAUTH library is in the STEPLIB concatenation of the CICS region. Check SYSPRINT, or the CICS log, for any error messages that are output by the Language Environment. It is likely that either the SDFJAUTH PDSE has not been included in the APF-authorized STEPLIB concatenation, or there is not enough storage available to Language Environment. Additional information might be available in message DFHSJ0216.

- `ERROR_CODE_UNRECOGNIZED`

  `START_JVM` returned an error that was not handled. Contact IBM Support for further assistance.

- `ERROR_LOCATING_MAIN_METHOD`

  An attempt to locate the main method within a setup or termination class has failed. An exception has been output to the standard error stream. Use the exception to determine the cause of the problem.

- `JNI_CREATE_NOT_FOUND`

  JNI Create has not been found. This error might occur because the JAVA_HOME value in the JVM profile does not specify the correct Java installation location. Check that the JAVA_HOME value specifies the correct installation directory.

- `JVMPROFILE_ERROR`

  An error occurred when processing the JVM profile. Additional diagnostic messages have been output to the standard error stream. The standard error stream is usually redirected to the location on zFS specified by the **`WORK_DIR`** parameter of the JVM profile. However, for early failures before redirection, the standard error stream might be in SYSPRINT, or in the CICS log as a dynamically generated DD name. Use the additional messages to determine the cause of the problem.

- `OPEN_JVM_ERROR`

  An error occurred when opening the JVM DLL. This error might occur because the JAVA_HOME value in the JVM profile does not specify the correct Java installation location. Check that the JAVA_HOME value specifies the correct installation directory.

- `REDIRECT_IO_FAILED`

  An attempt to redirect stdout, stderr, or jvmtrace output for the JVM has failed. Additional diagnostic messages have been output to the standard error stream either on zFS, in SYSPRINT, or in the CICS log as a dynamically generated DD name. Use the messages to determine the cause of the problem.

- `SETUP_CLASS_NOT_FOUND`

  A setup class specified in the JVM profile cannot be found. An exception has been output to the standard error stream. Check that the directory or archive containing the setup class is added to the classpath using the CLASSPATH_SUFFIX JVM Profile option and that the setup class is fully qualified.

- `SETUP_CLASS_TIMEDOUT`

  A setup class did not return in a reasonable amount of time and was therefore canceled. Ensure that your setup classes are not long running and return from the JVM within a reasonable amount of time.

- `STDOUT/STDERR_ACCESS_FAILED`

  An attempt to open the stdout or stderr stream for output has failed. The most likely reason is that the CICS job has read-access only to the directory specified by WORK_DIR in the JVM profile. Check that the CICS job has read, write, and execute access to the directory specified by WORK_DIR in the JVM profile.

- `TERMINATION_CLASS_NOT_FOUND`

  A termination class specified in the JVM profile cannot be found. An exception has been output to the standard error stream. Check that the directory or archive containing the termination class is added to the classpath using the CLASSPATH_SUFFIX JVM Profile option and that the termination class is fully qualified.

- `USS_VOLUME_CHECK_FAILED`

  An attempt to create stdout and stderr files for the JVM has failed because the UNIX System Services file system is full. You need to allocate more space to the file system used for stdout and stderr. It might also be useful to remove files that are no longer needed.

12.8.12 Deploying from Explorer SDK failures

If you are unable to deploy your web application using CICS Explorer SDK, check if there are errors in the CICS bundle project. If there are errors, the CICS Explorer SDK will not export files to zFS.

Redbooks

IBM CICS and Liberty: What You Need to Know

SG24-8335-00

ISBN 0738441368

IBM

(0.5” spine)
0.475”<->0.873”
250 <-> 459 pages

SG24-8335-00

ISBN 0738441368

Printed in U.S.A.