

Developing a RESTful Web application for Liberty in CICS

This course provides a step-by-step guide for creating a simple RESTful Java application, then deploying it into WebSphere Liberty running in IBM CICS Transaction Server. Topics covered include:

- Concepts and terms used in Java and RESTful applications
- Creating an environment suitable for developing Java applications for deployment into CICS
- Configuring CICS to run Java applications in a Liberty JVM server
- Developing the application, using supplied sample course materials
- Deploying and testing the application in CICS
- Extending the sample application in order to access other CICS applications

<http://www.redbooks.ibm.com/abstracts/crse0300.html?Open>

Section 1 Lecture 1

Hello and welcome to the CICS Java course around developing a RESTful Web Application in Liberty.

My name is Ian Burnett and I'm part of the CICS development team, based in the IBM Hursley lab in the UK. This course will show you how to get quickly started with Java application development in a CICS Liberty environment. We will start by configuring an Eclipse development environment ready to develop Java applications for CICS. Using this development environment, we will create a RESTful web application, and deploy this into a Liberty server running in CICS.

This new application will be able to interact with CICS to obtain some basic information about the environment, and to invoke a CICS COBOL application using the CICS Java API, also known as JCICS.

To follow along with this course, you will need access to:

- A development workstation capable of running an Eclipse-based environment.
- The ability to download and install IBM Explorer for z/OS onto this machine, as well as some sample application code.
- In addition, you will need access to a z/OS system, running CICS Transaction Server, V5.1 or later.

By the end of this course you will be proficient in the development of a simple RESTful Java web service, usage of the CICS Java API, and deployment of web applications to a Liberty server running in CICS. This will provide you with the first step in creating a Java-based, RESTful API for CICS programs.

Thank you for watching.

Section 2 Lecture 1

The Eclipse integrated development environment, or IDE, provides an extensible platform that is used as the basis for thousands of products. This includes the IBM Explorer for z/OS that we shall be using to develop our CICS Java applications.

Eclipse is provided by the Eclipse Foundation and supported by the Eclipse community and its member organizations. The base Eclipse IDE provides hundreds of common services, including a Java editor, with additional functionality provided by plugins. Plugins are commonly collected together into features. A wide variety of plugins are available for Eclipse including the CICS Explorer SDK and the WebSphere Developer Tools.

In this lecture, we will introduce the different Eclipse environments that are available for developing Java applications for CICS, and help you choose between the different options.

There are two main technologies involved when installing or updating an Eclipse platform.

Eclipse p2 is the built-in installation process, which provides an extension mechanism for installing plugins into Eclipse IDEs.

IBM Installation Manager is a standalone installation program that can use local or remote software repositories to install, modify, or update compatible IBM products. It determines and shows available packages, checks prerequisites and interdependencies, and installs or modifies the selected packages. Installation Manager provides greater administrative control for the installation process, however, it is unique to IBM software components whereas other 3rd party Eclipse plugins will be installable using p2. In this course, we will use the IBM installation manager process.

There are 3 different Eclipse environments you can choose between for Java development with CICS:

1. Eclipse from Eclipse.org
2. IBM Explorer for z/OS with CICS Explorer
3. IBM Rational Developer for z Systems, commonly known as RDz

Eclipse from eclipse.org is an open source version of the Eclipse framework that is downloadable from the eclipse.org community without charge. The CICS Explorer SDK can be installed into this platform allowing you to build a custom Eclipse workbench, integrated with the CICS Java development tooling.

Rational Developer for z Systems, is the IBM Eclipse-based integration platform for z/OS, providing a rich set of Java and high level language development tools designed for CICS, DB2, IMS and batch environments. All the features provided by IBM Explorer for z/OS are already integrated into Rational Developer for z Systems, meaning that if you are already licensed to use RDz then you can simply continue to use this environment for your CICS Java development platform.

IBM Explorer for z/OS is an Eclipse-based integration platform for z/OS tools and includes a subset of the features provided by RDz, including the ability to access MVS datasets, zFS, and JES. It is extendable via the IBM repository of compatible plugins enabling you to build a customized workbench for different users. For CICS Java development the CICS Explorer plugin can be easily installed, providing all the necessary capabilities. Entitlement for the IBM Explorer for z/OS is available for all customers with a license for a supported version of z/OS.

In this course, we will focus on using the IBM Explorer for z/OS as this is freely downloadable and requires minimal customization to create a CICS Java development platform.

Section 2 Lecture 2

In this lecture, we will configure your development environment in order to allow the development of applications that use the CICS Java API, commonly known as JCICS. If you already have a CICS Explorer SDK or z/OS Explorer environment, you can safely skip to the next section.

The first step is to download an Eclipse environment. Everything you need to get started can be downloaded from the IBM Mainframe developer center website at developer.ibm.com/mainframe.

Assuming you do not have IBM Installation Manager already installed, then navigate to the URL shown and choose the option "I am starting from scratch" to download a new Installation Manager installer, preconfigured to install IBM Explorer for z/OS.

Unzip the downloaded package and run the installer. You can choose between a user install, or a centralized shared administrator install. We will use the simpler user install so unzip this and run the `IMLauncherWindowsUser.bat` file. This will launch the installation manager install process, and you will need to select the packages to be installed.

IBM Explorer for z/OS will already be selected but scroll down the list and also select the IBM CICS Explorer SDK packages for installation.

Accept the license and click through the installation screens until the "Install Packages" screen is displayed.

Expand the "CICS Explorer SDK" package and ensure the "IBM CICS SDK for Web and JSP support" is selected, as this provides the web APIs for Java development that we will require.

Click Next and allow the installation process to complete.

The Installation Manager installer should then terminate, and you now are ready to start the process of developing CICS Java applications on your workstation.

In the next lecture, we'll discuss how to configure CICS Explorer connections.

Section 2 Lecture 3

Now that we have the IBM Explorer for z/OS installed, we need to configure the connectivity to our z/OS system and our CICS region. In this lecture, we will show you how to configure an FTP connection from Explorer to z/OS in order to support the editing and exporting of files to the z/OS UNIX File System and how to create a connection from CICS Explorer into a CICS region in order to define resources and manage operations.

Open the CICS SM perspective in CICS Explorer and then select the Host Connection view. There are 3 options presented for connecting to z/OS: FTP, z/OS Remote System, and z/OSMF. We will use the FTP option and will assume the FTP daemon has already been configured on z/OS. Select z/OS FTP and then click Add.

Enter the TCP/IP hostname for the z/OS FTP server and modify the name if desired. Click Save and Connect and you will be presented with a dialog to create a new user ID. Enter the required credentials and click OK.

The Host Connections view should now display a green icon for the new FTP connection we have just created. This will now allow access to zFS files, MVS datasets, and JES jobs within the z/OS perspective in CICS Explorer.

Next, we need to create a connection to our CICS region to allow us to create and manage resource definitions. We could use the traditional 3270-based CICS management utility transactions CEDA and CEMT to do this, but for this example, we will use the CICS Explorer.

To manage CICS resources using the CICS Explorer we need to create a CICS management client interface, or CMCI connection, to our stand-alone CICS region. This requires that the SEYUAUTH and SEYULOAD libraries have been added to the CICS region JCL. It also requires that URIMAP and TCPIPSERVICE resource definitions have been installed and enabled. Alternatively, you can also connect into a CICSplex SM WUI server if using a CICSplex.

For further details on configuring these resources search for the topic “Setting up access for CICS Explorer” in the CICS Knowledge Center.

In the CICS Explorer Host Connections view select CMCI and click Add. The “Edit CMCI Connection” window will be displayed. Enter the hostname and port that the CICS CMCI TCP/IP service is listening on. Click Save and Connect and you will be prompted to enter new credentials or you can reuse the existing credentials from the FTP connection.

Click OK and your CMCI connection should become active and display a green icon. If the CMCI connection is successful, the CICS Explorer view on the left will now be populated with details of the CICS region, and the Operations views will display resources such as Regions, Task, Programs etc.

In this section, we have successfully installed our CICS Explorer environment and created FTP and CMCI connections. In the next section, we will show how to use CICS Explorer to configure and manage a Liberty JVM server in CICS.

Section 3 Lecture 1

WebSphere Liberty is a Java Enterprise Edition application server. It provides, amongst other things, a web container for Java applications such as servlets, JSPs, and web services. CICS TS V5.1 and later provides a version of Liberty that supports web application deployment, and is also tightly integrated with the CICS runtime, to provide integration with the CICS qualities of services, such as security and transactionality.

In this section, we will introduce the JVMSERVER resource that runs within a CICS region, and explain how the Liberty server is configured within this. We then go on to show you the basic steps of how to configure Java in CICS, create a Liberty JVM server, and enable the drop-ins function for deploying applications.

Resources in a CICS region are configured using definitions stored in a CICS system definition dataset, or CSD. The JVMSERVER is the resource used to manage and control a JVM within CICS. The JVM profile is the zFS configuration file that is used to configure the JVM server, and defines the type of JVM server and the options that are passed to it. It is encoded in EBCDIC and can be configured using environment variables and system properties.

A Liberty JVM server is a specific type of JVM server which hosts a Liberty server. It is configured using the Liberty server configuration file, known as server.xml which is stored in zFS. In the following lectures we'll discuss configuring, starting, and validating a Liberty JVM server.

Section 3 Lecture 2

To configure a JVM server inside CICS, we will assume that Java support has been installed into your CICS region, and that the region is already started.

The CICS STEPLIB concatenation requires the SDFJAUTH library, and Language Environment support must be installed. The CICS system initialization, or SIT, parameter USSHOME should reference the CICS installation directory on zFS. The SIT parameter JVMPROFILEDIR references a zFS directory to which you will need to have read and write access, and will be used to store the JVM profile we will create. We will use the location `/u/cics1`.

A JVM server in CICS requires a file on zFS to provide configuration parameters. This is known as the JVM profile. We will use a sample profile supplied with CICS to quickly get started. The sample Liberty JVM profile is called `DFHWLP.jvmprofile`, and can be found in the `JVMProfiles` subdirectory of `USSHOME` as shown.

Copy the sample `DFHWLP.jvmprofile` file described earlier to the `JVMPROFILEDIR` directory specified by your SIT parameter. Now open the EBCDIC file in an editor.

There are only a few modifications needed to this file, in order to produce a fully-working Liberty JVM server. Within this file, first confirm that the `WLP_INSTALL_DIR` option is set, as this controls whether or not the JVM server is a Liberty JVM server.

Next, confirm that the `file.encoding` system property is set to the ASCII code page of ISO-8859-1. Next, edit the value of the `JAVA_HOME` variable to reference the zFS path containing the IBM Java Runtime Environment installed on your z/OS system. In this example, we will use Java 8.

You can optionally update the `WORK_DIR` variable to reference a zFS path where output files from the JVM server will be stored. We will use `/u/cics1`, which is the home directory of the userid under which our CICS region is running.

This userid requires read, write, and execute access to this directory so that the CICS region can create output files here.

Using the “autoconfigure” property, we can request that CICS creates a basic Liberty server configuration file, based on information we supply here. We shall enable this auto-creation by uncommenting the line and changing the value of the “autoconfigure” property to “true”.

There are two final configuration parameters which we need to supply to CICS before we can start the Liberty server. These are both related to TCP/IP: “host”, and “HTTP port”. Use values suitable for your environment.

Now that we have configured the JVM profile, we save and close the file. Lastly, we will require a `JVMSERVER` resource to be defined in the CICS system definition dataset to refer to the JVM profile that we just created.

For the purpose of this lecture we will use the supplied `DFHWLP` definition that can be found in the sample CSD group `DFH$WLP`, and can be used without modification, as it refers to the JVM profile named `DFHWLP.jvmprofile`.

To view this definition using the CICS SM perspective in CICS Explorer, open the JVM server Definitions view under the menu Definitions, JVM Server Definitions.

Select the target CICS region in the CICSplex Explorer view, which will then populate the JVMSERVER resources defined in the CSD for this CICS region. You will then see the JVMSERVER named DFHWLP that we will use to start our JVM server in the next lecture.

In this lecture, we used the CICS-supplied template to produce a customized JVM profile on zFS. We then created a JVM server resource definition which referenced this profile. In the next lecture, we will show how to install and enable this resource into our CICS region.

Section 3 Lecture 3

In this lecture, we will discuss the steps for starting and validating the Liberty JVM server in our CICS region.

Using the JVM profile we created earlier, we can now start the JVM server we will be using, by installing and enabling the JVMSERVER resource definition. Within the CICS SM perspective in CICS Explorer, open the JVM Server Definitions view under the menu Definitions, JVM Server Definitions.

Select the target CICS region in the CICS Explorer view, which will then populate the JVMSERVER resources defined in the CSD for this CICS region. To install the definition into our CICS region, right-click on the sample JVM server resource definition DFHWLP in the sample DFH\$WLP group and select "Install". Choose the target CICS region from the list presented and click "OK".

To verify the JVM server has been installed correctly, switch to the JVM Servers Operations view using the menu "Operations", "Java", "JVM Servers". This will display the status of the JVM server definitions that have been installed in the CICS region. Here we can see that the JVM server DFHWLP has been installed into the correct CICS region and is shown as "Enabled", signifying that it has started successfully.

Having verified that our Liberty JVM server is now running, we can verify the Liberty server is active by opening the default Liberty welcome page. Type in the hostname and port into your web browser and you should see the Liberty welcome page displayed. It is sometimes worth checking the Liberty message log, which can be useful to determine what is happening within the Liberty server. This file is called messages.log.

You can use the CICS Explorer z/OS UNIX files view in the z/OS perspective to view the logs using the FTP connection we configured earlier.

If using CICS TS V5.3, to locate the message log, open the JVM server working directory. In our case this is /u/cics1, and then navigate to the folder named after the APPLID of the CICS region, and then the name of the JVM server. Within this directory there is a symbolic link, called wlp.defaultServer.logs. Follow this link to the Liberty server logs directory, and within this location you will find the Liberty messages log.

The message log has a header containing information about the CICS Liberty environment and contains the Liberty server output messages. Here we can see the messages showing the server is in the process of starting up. Timestamps and other diagnostic information have been removed for clarity.

The application we are creating provides a RESTful interface. In specific, we will be using the Java API for RESTful Web Services, or JAX-RS. We will cover JAX-RS in detail later. For now, though, we need to enable this feature in the Liberty server configuration file as it is not enabled by default.

To add Liberty features we will need to edit the Liberty server configuration file, server.xml, which is by default located in a sub directory of the JVM server working directory. Navigate again to the base JVM server directory.

Within this directory there is a link called wlp.defaultServer.config, which points to the Liberty configuration directory. Within this directory, you will find the file server.xml. Edit server.xml and locate the <featureManager> element.

This lists all the Liberty features that are enabled in the Liberty server. By default, this only includes the CICS core feature and JSP 2.3, which provides JSP support. Add the JAX-RS 1.1 feature to this list as shown.

Before we finish, we will need to enable the dropins function in our Liberty server. This will enable us to deploy our web application by copying our WAR file into the dropins folder. The dropins directory itself will be dynamically created and by default will be found in the Liberty server configuration directory.

To enable dropins we will need to edit the Liberty server configuration file again. Edit server.xml and locate the <applicationMonitor> XML element.

Edit this statement to add "dropins enabled" equals "true" and "updateTrigger" equals "polled" as shown. This will ensure that the Liberty server scans the dropins directory every 5 seconds, looking for updates to applications that are deployed there.

Now save the changes to this file and navigate back to the message log and you should see the following messages to confirm the JAX-RS feature has been installed, and the dropins folder is being monitored for deployed applications. We have now completed the installation, enablement, and configuration of our Liberty JVM server.

In the next section, we will begin the development of our web application.

Section 4 Lecture 1

The Representational State Transfer, or REST, architecture pattern is widely used in modern web applications, as a means of accessing data using a simple, but well-defined interface. The Java API for RESTful Web Services, or JAX-RS, is an API specification for the Java programming language. It provides support for efficiently creating web services using the REST architecture pattern. Java annotations are used during development of an application to simplify both coding and deployment.

The JAX-RS API V1.1 is part of the Java Enterprise Edition V6 platform specification, and is supported in CICS when using a Liberty JVM server. RESTful applications frequently use the JavaScript Object Notation, or JSON, data-interchange format. JSON is lightweight, and easy for machines to parse and generate. JSON is also easy for humans to read and write.

This section looks at the creation of a simple Java application that uses the REST architecture pattern, using the JAX-RS API. The application will extract some basic information from the CICS Java runtime environment, and return this information in JSON format. By the end of this section, we will have created, deployed, and tested this application running in a Liberty JVM server.

Let's now take a look at a diagram of the application we will be creating.

We will use a web browser as an HTTP client to issue an HTTP GET request to our Liberty server running in CICS. This HTTP request will be received by the JAX-RS runtime in the Liberty environment, and converted into the invocation of a Java method in our application. Our application method will return the required information, encapsulating the data in a Java object. This Java object is then passed to the JAX-RS runtime in Liberty, which constructs a full HTTP response containing the returned data in the JSON format and sends it back to the browser.

In the next lecture, we will create the web application in our development environment, ready for deployment into CICS.

Section 4 Lecture 2

Earlier we configured our Eclipse development environment and we are now ready to create our Java application. Full source code for our sample RESTful application is available for download from the CICSdev GitHub using the URL shown. Go to this address now to download the source we need to get started.

To deploy our application into a Java EE server, we need a delivery vehicle, and for this example we will use a simple Web Application Archive, or WAR file. We can create a suitable project in our environment by selecting File, New, and then Project from the menu bar.

In the resulting New project dialog, select Dynamic Web Project from the Web folder, and click Next.

In my workspace I am using a project name of `com.ibm.cicsdev.restapp`. We do not require a target runtime selected, and we wish to use the default dynamic web module version of 3.0. Deselect the option to add the project to an EAR. An Enterprise Archive, or EAR, file is an advanced deployment option that is not required for our example.

Click Next to complete this page, and then accept the defaults on the following page by clicking Next again. In the final dialog box, select the checkbox to generate a `web.xml` deployment descriptor, and click Finish.

You may be prompted to switch to the Java EE perspective. Use of the Java EE perspective is not required to complete development of this application, but I will be using it here, because it offers some useful views into the state of our development environment, along with providing some helpful context-sensitive menus.

Now we have an encapsulating project, we can start building the application. First, we will create the Java classes that will contain the response data we will return to the caller. These classes will be simple Java beans, which use the JAXB interface.

JAXB, or the Java Architecture for XML Binding, is a framework used to map Java classes to XML. It can also be used to map Java classes to a JSON format, and it is for this reason we will use it here.

In this example, we will use two classes: `CICSInformation` and `CICSEnvironment`.

The `CICSInformation` class contains three pieces of information related to the CICS JVM server. It also contains a reference to an instance of the `CICSEnvironment` class, which itself holds information relating to the CICS runtime environment.

We will now add code to the application. Using the code downloaded from the CICSdev GitHub repository, drag and drop the top-level folder named `com` onto the `src` folder. When prompted, select the option to Copy files and click OK.

You should now have two Java packages, as shown here: `com.ibm.cicsdev.restapp`, and a related `.bean` package.

Importing the Java classes will produce some compile errors, due to missing classes in the build path. We will fix the compile errors now. Right-click on the project, and select Build path, and then Configure Build Path. In the resulting dialog, select the Libraries tab and click Add Library.

In the next dialog box, select "Liberty JVM server libraries" and click Next again. For this example application, we do not need functionality over and above that provided by CICS TS V5.1. Select CICS TS 5.1 from the menu and click Finish.

After the project has been recompiled, all the errors should now have disappeared.

Let's now take a look at the code by opening the imported `CICSEnvironment` class from the `.bean` package. Here we can see use of the JAXB annotations, applied to both the Java class, and the fields within. The fields `prod` and `cicsVer` represent the CICS product ID and CICS version string respectively.

The annotation `XMLAccessorType` of `NONE` indicates that the JAXB framework will only create bindings for explicitly annotated fields. The `XMLType` annotation indicates the name of the top-level XML element type, while the `XMLElement` annotations on the fields, indicate the name of the element created when this instance is serialized to XML.

As described earlier, JAXB is primarily a framework for accessing and producing XML data from Java, but it can also be used to produce and consume JSON data. We will be using this JSON functionality in our application.

The remainder of the `CICSEnvironment` class is a simple collection of getter and setter methods. The `CICSInformation` class uses a similar pattern of fields, getter and setter methods, and JAXB annotations. Now we have the Java classes defined to encapsulate the required data, we can continue building our application.

Now we will create the JAX-RS application itself. Open the `CICSApplication` class to view the JAX-RS source. The `JAX-RS ApplicationPath` annotation defines the component of the URI that will be used to access all resources within this WAR file. By extending the JAX-RS Application class, we can configure and deploy a simple JAX-RS application, without the need to add configuration information to the `web.xml` file. No further methods are required, and the class body is empty.

Now open the `InfoResource` class. This class represents a resource which may be accessed in our application. It contains the main business logic we will be using to obtain information about the current CICS system.

The `Path` annotation defines the URI path used to access this resource.

The `Produces` annotation indicates that this resource will return JSON data.

The `getCICSInformation` method is annotated with a single `Get` annotation, to represent the HTTP method that will be accepted. An object of type `CICSInformation` will be returned.

Now let's look at the `getCICSInformation` method body. The method creates an instance of the `CICSEnvironment` class, and populates it with data obtained from the standard Java `System.getProperty` method. Here, we highlight the CICS product name and CICS product version properties which provide the information required.

Similarly, an instance of the `CICSInformation` class is also created and populated using the `applid` and `JVM server name system` properties. A timestamp is also created and stored in the `CICSInformation` instance, and the method completes by returning the collated information in our Java beans.

You will notice there are other Java source files in the sample application that we have not yet used. These files will be useful in a future lecture, where we will be extending the application.

This completes the coding of our application, and we are now ready to deploy it into our CICS Liberty JVM server environment, which we'll cover in the next lecture.

Section 4 Lecture 3

In this lecture, we will be taking the RESTful web application we have created, and then deploying and testing the application in a CICS Liberty environment.

As described in an earlier lecture, Liberty offers a quick and simple means of deploying and updating applications into the server with no restart required. The dropins directory is the location in zFS that the Liberty server periodically scans for updates, and modifies the status of running applications dynamically. This method of deployment is not recommended when deploying applications in a production environment, but is ideal for our purposes, providing a rapid application development and test cycle. The dropins directory needs to be explicitly enabled, so check that the entry in the Liberty configuration file is correctly specified.

When we initially created the project in our development workspace, we decided our delivery vehicle for the application would be a Web Application Archive, or WAR, file. We now need to export the completed application as a WAR file and copy it to the dropins directory.

Creating a WAR file from our environment is a simple process – right-click on the project, select “Export”, and then “WAR file”. You will then be prompted to enter an export destination. Specify a filename on your local workstation and click Finish.

The exported WAR file now needs to be copied to the dropins directory we discussed earlier. Copy the exported WAR file to the dropins directory using any method that can transfer binary files. In my environment, I use FTP.

Now the file has been transferred to the Liberty dropins directory, the Liberty server will detect there is a new application to install, and will do so automatically. To confirm this has happened correctly, we look at the Liberty messages.log file we used earlier when verifying the server had started correctly.

Firstly, we can see the message that confirms that our web application has been detected by the dropins monitoring logic. This is quickly followed by a confirmation message to show the application is being loaded.

The next message shows the web application is bound to the default_host – an internal identifier representing any hostname and port combination that the Liberty server may be using.

The most interesting message is next, and this confirms the web application is now available, and provides the base URL we can use to access it. A final confirmation message shows the application started correctly.

To test the application, we need to determine the URL required to access the InfoResource class we coded earlier. The URL is composed of three parts.

The first is the base URL of the web module we have just seen in the Liberty messages log file.

The second part is the application path, which was specified in the ApplicationPath annotation on the CICSApplication class.

The final part is the resource-specific component, which was specified in the Path annotation on the InfoResource class.

Combining these components produces a URL we can use to access the RESTful resource we have created in CICS. Returning to our high-level diagram, we can see that we need to send in an HTTP GET request to the application, and we will use a web browser as a simple HTTP client.

Enter the URL into the address bar, and press enter.

You should receive back a response in JSON format, providing the information we have obtained from the CICS runtime. All of the returned data is enclosed in an outer pair of braces.

The `getCICSInformation` method in the `InfoResource` class returned a `CICSInformation` object, which contained four pieces of information: the CICS application ID, the name of the CICS JVM server, a timestamp, and a nested `CICSEnvironment` object.

Correspondingly, the `CICSEnvironment` object contains two pieces of information: the CICS product name, and product version.

By following these lectures, you have now coded, deployed, and tested a RESTful application in a Liberty JVM server in CICS.

If you want to modify the application, then update the Java source in your development environment and re-export the project as a WAR file. Transfer the updated WAR file to the `dropins` directory and simply overwrite the old WAR file. When the updates are detected by Liberty, the log file will reflect this with a new set of messages to indicate the application has been restarted.

In the next lecture, we will look at how to install our RESTful web application in CICS, in a manner that is more suitable for an enterprise environment, through the use of CICS bundles.

Section 4 Lecture 4

In the previous lecture, we deployed a RESTful web application into a Liberty environment using the dropins directory. While the dropins directory is useful for a rapid application development and test cycle, it lacks certain capabilities expected in a CICS environment. Deployment of an application into the Liberty dropins directory will limit the security available, and CICS will not be able to provide application lifecycle management.

In this lecture, we will deploy the WAR file into CICS using a collection of resources known as a CICS bundle. You should not deploy an application through both the dropins and the CICS bundle mechanisms, so before continuing, you should delete any existing WAR files from the dropins directory to avoid conflicts later.

A CICS bundle is a collection of resources that are installed into CICS as a unit. One or more resources of different types may be installed using a single CICS bundle. A CICS bundle is a directory containing XML files that describe resources. The resources are referenced from a manifest file, called `cics.xml`, in the `META-INF` subdirectory.

All of the mechanisms used to deploy Java applications into a Java EE server are supported by CICS bundles, including WAR, EAR, and EBA files.

Our first task is to create a CICS bundle project in the workspace. From the z/OS Explorer workspace, choose the File, New, and Project option. Choose the CICS Bundle Project wizard, and click Next.

You will need to supply a project name, and I chose `com.ibm.cicsdev.restapp.bundle`. We can revisit the version number field later.

Completing the wizard produces a CICS bundle project in the workspace, with an empty manifest file. If it isn't open already, double-click on the `cics.xml` manifest file to open the CICS bundle editor. The CICS bundle editor provides several useful tools to edit the manifest file and manage the accompanying resource description XML files. Here we see the editor for an empty manifest file.

We need to add a reference from our CICS bundle, to our web project, and this is achieved by clicking New and selecting the "Dynamic Web Project Include" option. In the dialog shown, select the dynamic web project we created in the previous lecture, and enter the name of our JVM server. In our case, this is DFHWLP.

When the "Dynamic Web Project Include" dialog is closed, you can see the editor has been updated with a reference to our web application project, and this corresponds to the warbundle file shown in the project tree.

Our next step is to export the CICS bundle structure to zFS, which will also include the WAR file for deployment into Liberty. The CICS Explorer environment provides a wizard to automatically build a CICS bundle, along with all other related projects, and then transfer all required files to the zFS environment.

Right click on the CICS bundle project and select the "Export Bundle Project to z/OS UNIX File System" option. Now select the option to export to a specific location in the file system, and click Next.

If you correctly configured the FTP connection, then you will be presented with a dialog like this, where a connection to the z/OS system is active, and allows you to browse the zFS file system. I have

also enabled the checkbox to clear existing contents of the bundle directory in order to overwrite anything already on the file system.

Clicking Finish will begin the process of transferring the project to the remote file system. The console window will provide a report on the operations as they complete.

Switching to the CICS SM perspective we used earlier when creating the JVMSERVER resource, we will now create a resource definition to reference the zFS location we have just exported to. Open the bundle definitions view under the menu Definitions, Bundle Definitions.

Select the target CICS region in the CICSplex Explorer view, which will then populate the BUNDLE resources defined in the CSD for this CICS region. To create a new bundle definition, right click in the "Bundle definitions" view, and select "New".

You will need to supply the name of a CSD group in which to create the resource. Here, I've used the group LIBERTY. See your CICS system programmer for a suitable group name.

Specify the name of the definition. In this example, I have used RESTAPP.

The final piece of information required is to reference the zFS path to your application from the bundle definition. This is the path you exported the CICS bundle project to earlier.

Either enter the path manually, or use the z/OS Connection to allow remote browsing of the file system. Once the required information has been specified, click Finish.

As with the JVMSERVER resource created earlier, we now install the resource into our CICS system. Find the new bundle definition we have just created, right-click, and select Install. Select the CICS region into which the resource should be installed, and click OK.

As with the JVMSERVER resource created earlier, we can look at the state of the installed bundle resource we have just created. From the main menu, select Operations and Bundles. The RESTAPP resource we have just installed should show a status of Enabled.

Given that a CICS bundle can contain multiple resources for installation, we may wish to look at all the parts within it. Right-click on our RESTAPP bundle and select Show bundle parts. This produces a view to show the components of the bundle we have created.

We can verify the application has again been installed into Liberty by returning to the Liberty messages.log file as we used earlier. Regardless of the installation mechanism, we can see that Liberty has performed the same installation processing, and provides the same output messages as when we used the dropins directory earlier.

Using your web browser, test the application again to verify the server has been correctly updated.

By completing this section, you have now coded and deployed a RESTful web application in CICS. The application used standard Java programming practices to obtain information about the CICS environment, and returned the data in a JSON format. We also looked at two methods of application deployment – the dropins directory and CICS bundles.

In the next section, we will expand this application – connecting to a COBOL program from our RESTful web interface.

Section 5 Lecture 1

In this section, we will introduce some CICS concepts, and demonstrate how to use the JCICS API, to call an existing COBOL program from a Java application. Note that to complete this section, you will need to download and compile the sample COBOL program EDUCHAN which is available for download from the CICS Developer Center section of the GitHub website.

Our first piece of CICS terminology to explore is the concept of a CICS task. From a high level, a task is analogous to a thread, created as a result of a request into CICS. When CICS receives a request for a piece of work to be executed, a new task is created.

A task encapsulates all the work that is performed as a result of that request. The work performed by a task is defined by a transaction, which specifies properties such as the program entry point, scheduling, and security.

In traditional languages, a program contains the application logic used to fulfil the request. A single task can encompass control being transferred between multiple programs, with all code being executed as part of the same task. When calling one program from another in CICS, we use the LINK command.

This example shows program A performing a LINK operation to program B, passing some input data to the called program. Program B will perform some business logic and then return data back to program A before terminating. These programs can be written in Java, as well as COBOL, PL/I, and other languages through the mixed-language application support provided by CICS.

There are two main methods to pass data between programs, either through the use of a communication area, or through the use of channels and containers.

A communication area, or COMMAREA, is a piece of storage, which contains a data structure defined by a program. Most commonly this is a COBOL copybook. In this example, you can see a COMMAREA containing three fields.

This COMMAREA is passed from program A to program B when a CICS program LINK occurs. Program B can use this data, and then modify the COMMAREA to return information back to program A.

There are a few limitations with COMMAREAs. The first is that a COMMAREA has a maximum size of 32 KB. The second issue is that both the calling and called programs must have exactly the same definition of the COMMAREA layout, to enable correct intercommunication. This results in a tightly-coupled application structure.

Where large COMMAREAs are used, this can be inefficient, as the whole COMMAREA is always transferred between programs, regardless of the scope of changes made in the target program. With these limitations, and the growing need for larger data structures, a new solution was required to supplement the use of COMMAREAs. Channels and containers were introduced to address the limitations with COMMAREAs.

A channel is a named area of CICS managed storage associated with a task. Within a channel, you can store multiple pieces of data in the form of named containers. Containers are classified as holding either CHAR data, where CICS manages any codepage conversion required; or as holding BIT data, where no conversion takes place.

In our example, program A creates a channel associated with the task, then creates a container to store some input data for program B. Program A then performs a link to program B. Program B uses the channel reference passed from program A to retrieve the input container and read its contents. Program B then performs some business logic, and creates a new container to store the output data. Control is returned to program A, along with any new or modified containers. Finally, program A will retrieve the data from the output container.

Channel and container names are limited to 16 characters but can otherwise be named using any convention suitable for your application.

When developing CICS applications in Java, we need to use the JCICS API to invoke CICS commands. This API provides the capability to interact with most resources that are controlled by CICS using a set of dedicated CICS classes such as Task, Program, Container, or File. The JCICS API is analogous to the EXEC CICS API available in other CICS supported languages.

One element of this API is the Task class. This class contains a number of methods to get information about our execution environment. This includes the transaction name, the user ID that the task is running under, and the program name.

To obtain an instance of the Task class, use the static `getTask` method on the Task class to reflect the underlying CICS task in which our application is executing.

The link command in the JCICS API can be found in the Program class. To link to a named program, create a new instance of the Program class, and use the `setName` method to specify the name of the target program. Invoking the link method will then pass control to the specified program.

This example shows how you to instantiate a Program object. The only mandatory method is to set the program name that you want to invoke. In this case, we have specified PROGRAMB. Next, setup any container data to pass to PROGRAMB within a channel. Now invoke the link method, which will call PROGRAMB, passing the channel data we have created.

This lecture has introduced the CICS link API and shown an example of using the JCICS API to invoke another CICS program.

The next lecture will show you how to use these commands to invoke a CICS program from our RESTful web application, passing data to this program and returning a result.

Section 5 Lecture 2

In the previous lecture, we showed how to link to a CICS program from Java, and access task information, using the JCICS API. This lecture will use both these concepts, and show how to use the JCICS API to provide additional functionality to the RESTful application by linking to a CICS program.

Let's now take a look at a diagram of the changes we are going to make to our web application. The flow of JSON data into and out of the web application follows the same path as in the previous section. The web application will now obtain its return data from a COBOL program running within CICS, and use this to produce the HTTP response.

Now we are ready to add some more code to our RESTful application, by creating a new resource class. Open the ReverseResource class, which contains the Java logic to link to our COBOL program.

As with our previous InfoResource class, we have a Java class defined with JAX-RS annotations. These annotations include a high-level path for all methods in the class, as well as the Produces annotation to signify the resource will again return JSON data.

We then have a new method named reverse, again with JAX-RS annotations. This method declares the path annotation value of /text, where the braces around the word "text" signify this is a path parameter.

This path parameter is then referenced in the method signature, using the PathParam annotation. On each invocation of the method, the value supplied in the URI is passed as the argument to the reverse method. The method will now take the input string parameter, and pass this to the EDUCHAN program, which will reverse the characters and return them to the application.

In the method body, we declare and initialize the Program and Task objects we will use in the rest of the method. The program name is set using the setName method, and this is set to EDUCHAN, our sample COBOL program.

The Task object is obtained using the static getTask method, as we saw in the previous lecture. The chan, inputContainer, and outputContainer local variables are defined as shown. These are used when building the input data and reading the output data to the EDUCHAN program.

To construct the input data, we create a channel. We do this by calling the createChannel method on the task object obtained previously. This creates a channel associated with the current task. The only required parameter is a channel name.

Next, we create the input container, by calling the createContainer method on the newly-created channel object. A name of "INPUTDATA" is specified for the new container.

We then put data into our container by calling the putString method on the container. The putString method indicates this will be a container holding data of type CHAR, and therefore CICS will perform any codepage conversion required. Remember the inputStr variable is the method argument populated from the HTTP request URL.

Now we are ready to invoke the program we declared earlier in the method. We do this by using the program link method, and passing the channel object as a parameter.

Most JCICS methods throw checked exceptions. These exceptions map to the underlying CICS response codes, and suitable try-catch logic is required.

When the link method returns, we can get the output container, by calling the channel `getContainer` method. The container returned will have been created within the called program and in our example is the `OUTPUTDATA` container created by the `EDUCHAN` COBOL program.

Lastly, we assign the contents of this output container to the `String` `outputStr`. We can now return the output of the `EDUCHAN` program as the result of this method.

As part of our project we have a `ReverseResult` Java bean that we annotated using the `JAXB` annotations. We assign the values for the original text, reversed text, and time in `ISO-8601` format. We then return this object from our method to the `JAX-RS` runtime.

If you need to refresh the web application in CICS before testing, use the same method detailed earlier and export the CICS bundle project to zFS. To refresh the application when using CICS bundles, simply disable the bundle, and then re-enable it, using the CICS Explorer bundle operations view.

To test this new part of the application, we can determine the URL required to access the `ReverseResource` class, using the same methodology as we used for the first resource. The URL is composed of the base URL of the web module, `com.ibm.cicsdev.restapp`, the application path, `rest`, and the resource path, `reverse`. Combining these components produces the URL we can use to access the `JAX-RS` `ReverseResource` we have created in CICS.

The `reverse` method in the `ReverseResource` class is invoked when an additional path parameter is supplied on the `HTTP GET` request. Using a URI ending with `reverse/ilovecics`, we should see `JSON` data returned as shown here.

This section has shown how we can use the `JCICS` API to access CICS tasks and call programs, including how to pass data using the channels and containers feature of CICS. We can now expose an existing COBOL program as a RESTful web application.

In the final section, we will summarize what we have learned in this course, and highlight the steps we have taken in creating this RESTful web application.

Section 6 Lecture 1

This course has covered, from first principles, the creation and deployment of a RESTful web application into a CICS Liberty environment.

We started by establishing our development environment, and configuring it for CICS Java development. We then defined and configured a Liberty JVM server that we could use during testing. We then developed a simple JAX-RS resource, which returned some basic information about the CICS runtime environment.

Another resource was added, demonstrating several CICS concepts, including tasks, programs, channels and containers, and use of the LINK command. The JCICS API was covered, and how it maps to the underlying CICS commands.

The application was deployed initially using the Liberty dropins directory, and later using the CICS bundle mechanism. Using a web browser, we tested the application, returning information in a JSON format.

Many code samples, including the ones used in this course, are available on our CICSdev GitHub site. Together with the topics covered in this course, these will enable you to quickly get started with your first RESTful web application in CICS Liberty.

This concludes our CICS Java course around developing a RESTful Web Application in Liberty.

Thank you for watching.