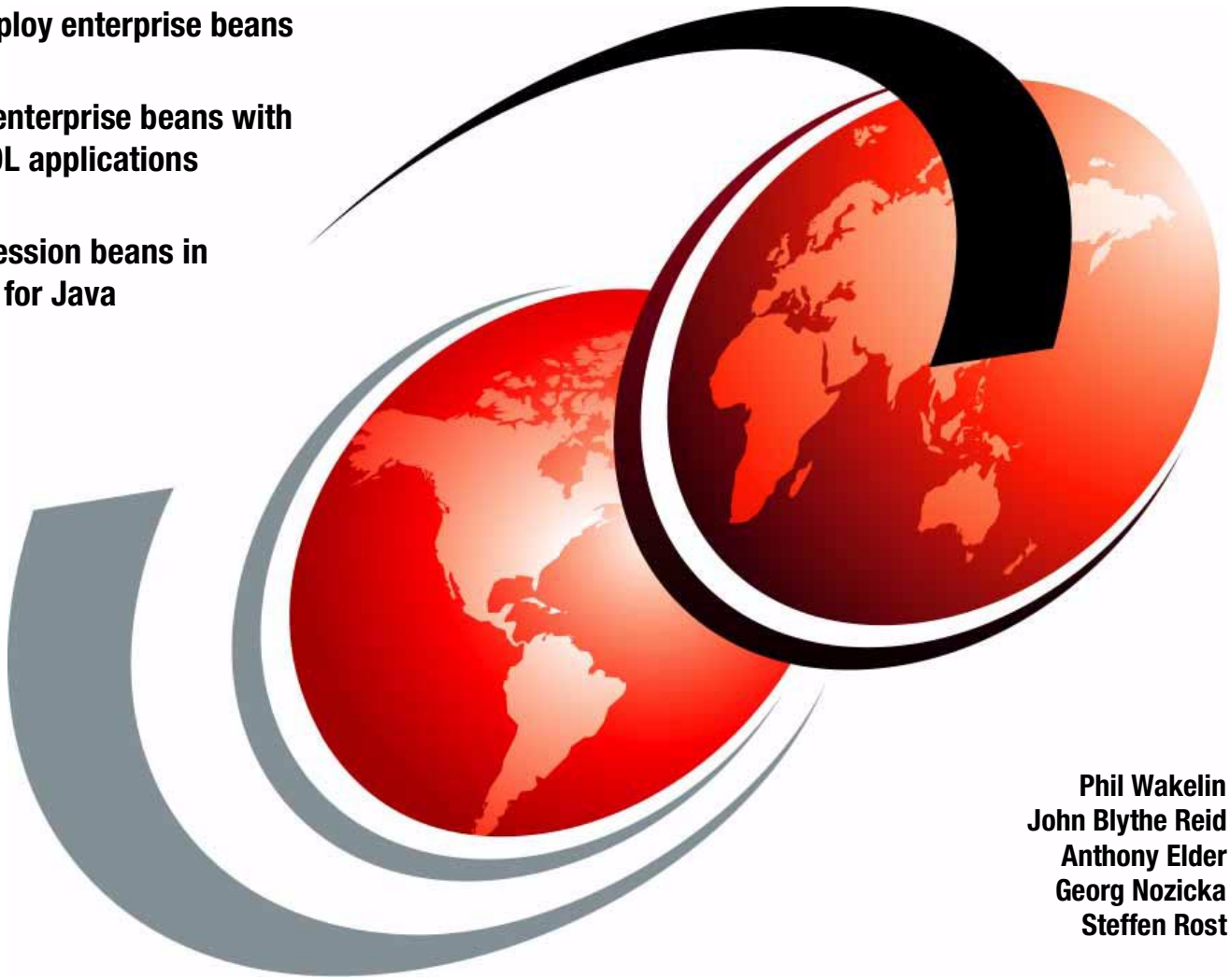
IBM

Enterprise JavaBeans for z/OS and OS/390

CICS Transaction Server V2.1

- Understand the CICS EJB Server and how to deploy enterprise beans
- Integrate enterprise beans with your COBOL applications
- Develop session beans in VisualAge for Java



Phil Wakelin
John Blythe Reid
Anthony Elder
Georg Nozicka
Steffen Rost

Redbooks

ibm.com/redbooks

International Technical Support Organization

Enterprise JavaBeans for z/OS and OS/390 CICS Transaction Server V2.1

July 2001

SG24-6284-00

Take Note! Before using this information and the product it supports, be sure to read the general information in “Special notices” on page 321.

First Edition (July 2001)

This edition applies to CICS Transaction Server for z/OS Version 2 Release 1, for use with the OS/390 Version 2, Release 10 Operating System.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

Contents

Preface

Consistent with its 32-year technical evolution, CICS, the IBM Customer Information Control System, has delivered support for the Enterprise JavaBeans (EJB) technology in its latest release, CICS Transaction Server for z/OS V2.1.

In this IBM Redbook, we first provide an introduction to both EJB and the way it has been implemented within the CICS architecture. We also include a summary of the different configurations in which servlets and enterprise beans can be used to access CICS applications.

Following this, we document how to set up and configure a CICS region to support enterprise beans, how to use the various new tools and features required, and how to deploy and test the product samples. Then we provide information on how to diagnose and fix problems when deploying and testing enterprise beans in CICS.

Finally, we document five scenarios in which we developed enterprise beans and deployed them to CICS. We start with the initial step of creating a simple HelloWorld session bean using the VisualAge for Java Development environment, and then move on to creating a stateful session bean called TraderBean that wraps the existing pseudo-conversational COBOL Trader application.

Following this, we provide details on how to develop new Java versions of COBOL applications using either the JCICS classes, or the SQLJ and JDBC interfaces. We also provide details on how we developed a sample JSP/servlet application to invoke the TraderBean and information on how to deploy this in WebSphere Application Server for Windows NT and WebSphere Application Server for OS/390.

This redbook is part of a two-part series entitled *Enterprise JavaBeans for z/OS and OS/390.* The other book in this series covers WebSphere Application Server V4.0 for z/OS and will be available as SG24-6283.

The CICS evolution continues

CICS first offered on-line transaction processing beginning in 1968, providing support for traditional procedural programming languages such as COBOL and PL/I to be used in a high volume, high performance transaction processing environment. Since then, it has continually enhanced that basic capability in virtually every year of its existence.

In 1972 CICS was one of the first software systems to offer support and exploitation of a new IBM device for commercial data processing, the 3270 Display Station, which has since become a legacy synonymous with on-line transaction processing itself. Data management support with CICS began with standard access methods but today includes the major IBM offerings of VSAM, DL/I and DB2, plus the use of, and co-existence with, major non-IBM relational data base management systems.

In the late 1970s, CICS first offered its support for client-server and distributed systems with the introduction of Intersystem Communication (ISC) and Multi-Region Operation (MRO), and support for communications protocols such as Advanced Program to Program Communications (APPC). In the 1990s this evolved into the CICS client support, and became the basis for today's high function CICS Universal Client, which runs on a wide variety of client platforms including Windows, IBM's AIX, HPUX, and Solaris.

In the mid-1990s CICS first offered its support for the then-burgeoning Internet, with the introduction of the CICS Internet Gateway and the CICS Gateway for Java, both of which now having evolved into the very comprehensive CICS Transaction Gateway. In parallel with the support of Java, CICS/ESA V4.1 developed the CICS Web Interface, in order to provide a CICS application with the ability to directly send and receive HTML pages to a Web browser. This has now itself evolved into the CICS Web support feature of CICS Transaction Server V1.3, and perhaps unknowingly provided the first step towards EJB connectivity by allowing a CICS region to process TCP/IP requests.

In addition to all this technical growth and evolution, CICS has exploited other environmental improvements for the benefit of its customers. CICS offers high availability and high performance with its support of the S/390 Parallel Sysplex architecture, and allows network workloads to be dynamically routed to available CICS regions through the use of the MVS Workload Manager (WLM) and the CICSPlex System Manager (CICSPlex SM).

During its progress through the decades, CICS has evolved to include support for all the popular programming languages (COBOL, PL/I, Assembler, REXX, C) and now includes support for object-oriented languages such as C++ and Java. Java, like the Internet, has been rapidly accepted by the I/T industry as the language of choice. But Java is proving to be rather more than a programming language, as it is continually expanding from the basis of many new computing standards the latest of which is Enterprise JavaBeans (EJB).

The EJB specification was first introduced by Sun Microsystems in 1999, and has now been endorsed and supported by many companies in the computer industry. The EJB specification addresses the need for transactional capabilities for the Java component technology of JavaBeans, and provides for transactional services (sharing, integrity, recoverability, persistence, and so on) unique to the specific platforms. Application developers, using EJB, can concentrate on the business logic and not the physical environment, and should the needs of the business change, the enterprise beans can be ported to other environments which support the EJB standard.

With its tremendous pedigree, the time is now right for CICS to embrace object oriented technology. Now there is an EJB container that runs within a truly proven transaction processing environment. Once again, CICS is on the leading edge.

Bob Yelavich, Yelavich Consulting

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

Phil Wakelin is a Senior IT Specialist at the International Technical Support Organization, San Jose Center, where he has worked since 1999. He joined IBM in 1990, originally working in the System Test department of IBM Hursley. He worked on most platforms and versions of CICS before joining the Installation Support Center, as a pre-sales support specialist for CICS client-server. He is an IBM Certified Solutions Expert – CICS e-business, and holds a BSc degree in Applied Biology from the University of Bath, UK.

Georg Nozicka is a Certified IT Architect for IBM Global Services in Austria. He has over 17 years of experience in the industry. He worked for many years in the IBM Vienna Software Development Laboratory, where he specialized in workstation application development. Georg was one of the key architects of the IBM workflow manager FlowMark, now known as MQ Series Workflow. Since 1996, he has worked for IBM Global Services as an IT Architect in the insurance and banking industries. More recently he has been involved in OS/390 application development projects for e-business enablement. He holds a master's degree in Computer Science from the Technical University of Vienna.

Anthony Elder is a Software Engineer with the CICS Change Team at IBM's Hursley Laboratory in the UK. He has 14 years experience with IBM mainframes using most of IBM's major operating systems and components. He holds a BSc degree in Computer Science from Auckland University, New Zealand, is a Sun certified Java programmer, and is currently studying for an MSc in Software Engineering at Oxford University in the UK.

John Blythe Reid is an IT Specialist with IBM UK, working in Software Group Services at IBM's Hursley Laboratory. He has extensive experience in both software development and systems programming in the IBM mainframe environment. His area of specialization is in the area of transaction processing systems and his most recent assignment was to work on the CICS TS V2.1 beta test. He is currently following a postgraduate software engineering programme at the Open University in the UK.

Steffen Rost is an IT Specialist working at the IBM zSeries Software Development Laboratory in Boeblingen. He has 5 years of experience in computer science and application development. His current areas of expertise include application development with Java and EJB, and developing e-business solutions using WebSphere and VisualAge for Java. He holds a degree in Computer Science from the University of Rostock.

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

Ken Davies, Chris Smith, Geoff Sharman, IBM Hursley UK, for supporting this project. Becca Loader, Richard Chamberlain, Adrian Colyer, Glyn Normignton, John Tiling, John Bond, Richard Chamberlain, Chris Backhouse, Noel Sales, Terry Warren, Shane Babey, Paul Willats, Hilora Munro, Andrew Clement, George Burgess, Daniel McGinnes, Adrian Thomson, IBM Hursley for technical input.

Christopher Farrar, IBM Silicon Valley Laboratory, for advice on JDBC and SQLJ.

Norm Aaronson, Patricia Healy, David Booz, IBM Poughkeepsie, for technical input.

Dennis Weiand, Leigh Compton, IBM Dallas System Center, for reviewing.

Bob Haimowitz, Rich Conway, ITSO Poughkeepsie, for providing excellent systems support.

Yvonne Lyon, ITSO San Jose, for technical editing support.

Special notice

This publication is intended to help systems programmers and application developers to build and deploy enterprise beans in CICS Transaction Server V2.1. The information in this publication is not intended as the specification of any programming interfaces that are provided by IBM CICS Transaction Server for z/OS Version 2 or WebSphere Application Server for z/OS V4. See the PUBLICATIONS section of the IBM Programming Announcements for more information about what publications are considered to be product documentation.

IBM trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

e (logo)® 	Redbooks
IBM ®	Redbooks Logo 
AIX	Parallel Sysplex
CICS	RACF
CICS Connection	S/390
CICS/ESA	SupportPac
CICSPlex	System/390
DB2	VisualAge
FlowMark	VTAM
Language Environment	WebSphere
OS/390	Wizard

Comments welcome

Your comments are important to us!

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

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

  `ibm.com/redbooks`

- Send your comments in an Internet note to:

  `redbook@us.ibm.com`

- Mail your comments to the address on page ii.

Part 1

CICS and EJB

In this part we introduce the Enterprise JavaBeans (EJB) technology and provide details of how this technology has been implemented within the CICS TS V2.1 architecture. We also provide details on the different scenarios in which CICS applications can now be invoked from both servlets and enterprise beans.

1

Enterprise JavaBeans: An introduction

In this chapter we provide a general description of the Enterprise JavaBeans (EJB) technology. We describe what enterprise beans are and how they are beneficial to the application development process; the two types of enterprise beans session beans and entity beans; and finally the manner in which the enterprise beans may inter-operate. An overview of how EJB is used by a Java Client is illustrated in Figure 1-1.

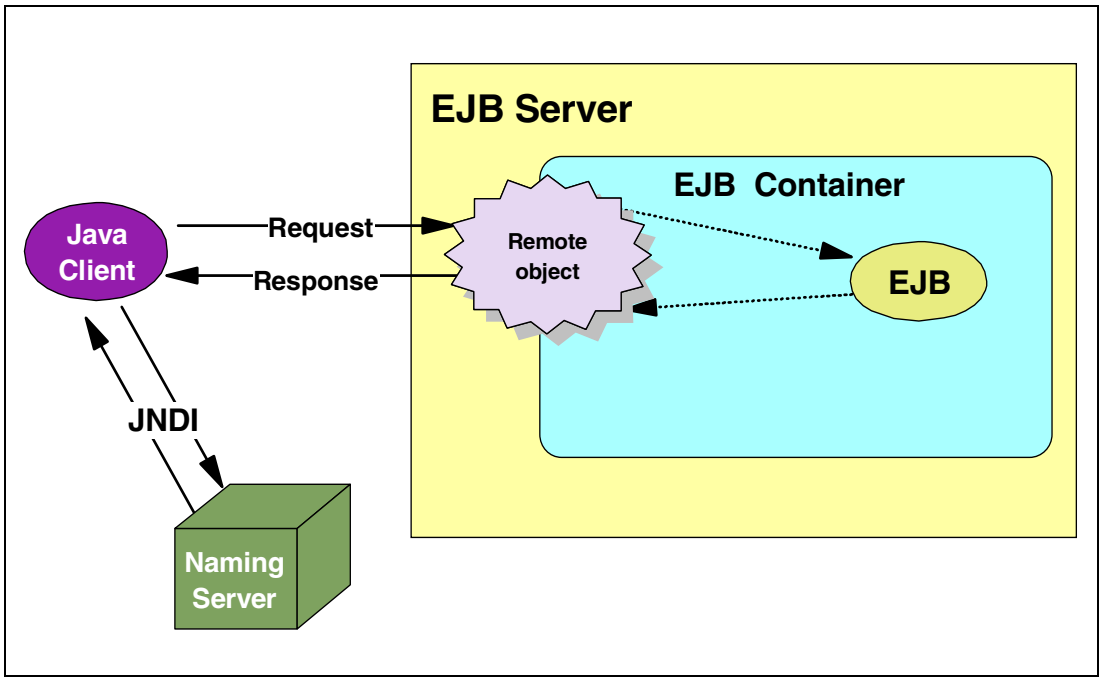


Figure 1-1 EJB overview

1.1 Enterprise JavaBeans

In this section we take a close look at the EJB technology and how it permits the developer to create applications without the need to consider certain execution time aspects such as transactionality and security during the development activity.

1.1.1 Object orientation

Object oriented software engineering brings many advantages to the development process. Application design more realistically attempts to model the real world by defining classes of objects which communicate with one another by sending messages. Each object encapsulates function and has its own state which may only be queried or changed using formally defined protocols, known as the method signatures.

Object orientation permits easier reusability of existing code by a technique known as inheritance and increases reliability by hiding the implementation of function and state from the user of the object.

Developing applications in this component-like way also enhances maintainability. A component may be internally re-engineered or indeed substituted, and providing its external protocol is preserved, then the change will be transparent to the remainder of the application.

The re-use of existing components has been particularly successful for the development of graphical interfaces on the client platform. Libraries of classes defining graphical components such as buttons, frames and text boxes have been available for some time. The *java.awt* and *swing* classes are examples of these class libraries.

Now the focus is moving to providing these same benefits of re-use on the enterprise platform by using object-oriented techniques for server-side development. However, the development of enterprise-wide server-side applications introduces new challenges in areas such as transaction management, security and scalability. The Enterprise JavaBeans technology is rapidly becoming an industry standard to address these challenges.

An enterprise bean is a Java component which is written to conform to the Enterprise JavaBeans specification. It is deployed into an EJB server, and its name is published into a name space using the Java Naming and Directory Interface (JNDI).

A client program locates the enterprise bean by obtaining its reference from the name server, asks the container in the EJB server to make the enterprise bean available and then invokes methods on the bean by sending messages to the object reference returned by the container. The messages sent to the enterprise bean by the client are intercepted by a component of the EJB server known as the container which applies transaction management and security rules to the execution of the enterprise bean's method. When the enterprise bean's method completes execution, a response is returned to the client (see Figure 1-1).

1.1.2 Transactionality

Transaction management is an important aspect of enterprise-wide server-side operation. In order to maintain the integrity of the information held on possibly dispersed databases, it is essential that any changes to persistent data are either all committed on successful termination of a transaction or all rolled back if the transaction fails.

These properties of a transaction are commonly referred to by the acronym ACID, representing *Atomicity, Consistency, Isolation* and *Durability*. The meanings of these terms in this context are as follows:

Atomicity indicates the coordination of all the participants in a transaction, such as enterprise beans, database managers, and servlets, so all must complete successfully for the data to be committed to the database or databases, whereas the failure of any participant to complete successfully will result in all database changes being rolled back.

Consistency means that the execution of the transaction must always leave the database in a consistent state upon completion of the transaction. For example, in a banking application the sum of account balances for a branch must always be equal to the branch total.

Isolation means that the execution of concurrent transactions must produce the same results as if the transactions were executed serially.

Durability means that database integrity will be maintained, even in the event of a system failure during the execution of a transaction.

The transaction management rules which are applied to enterprise bean method invocations are based on the Object Transaction Service (OTS) which is part of the CORBA specification.

An OTS transaction permits transaction management in a distributed heterogeneous environment. The instigator of an OTS transaction registers the start of the transaction with an OTS Transaction Coordinator which returns a transaction context that uniquely identifies the transaction. The transaction context, which includes a reference to the OTS Transaction Coordinator responsible, is passed as part of the protocol for method invocations on remote objects such as enterprise beans.

As shown in Figure 1-2, the transaction context is propagated throughout all method invocations on any enterprise beans which are involved during the life of the transaction.

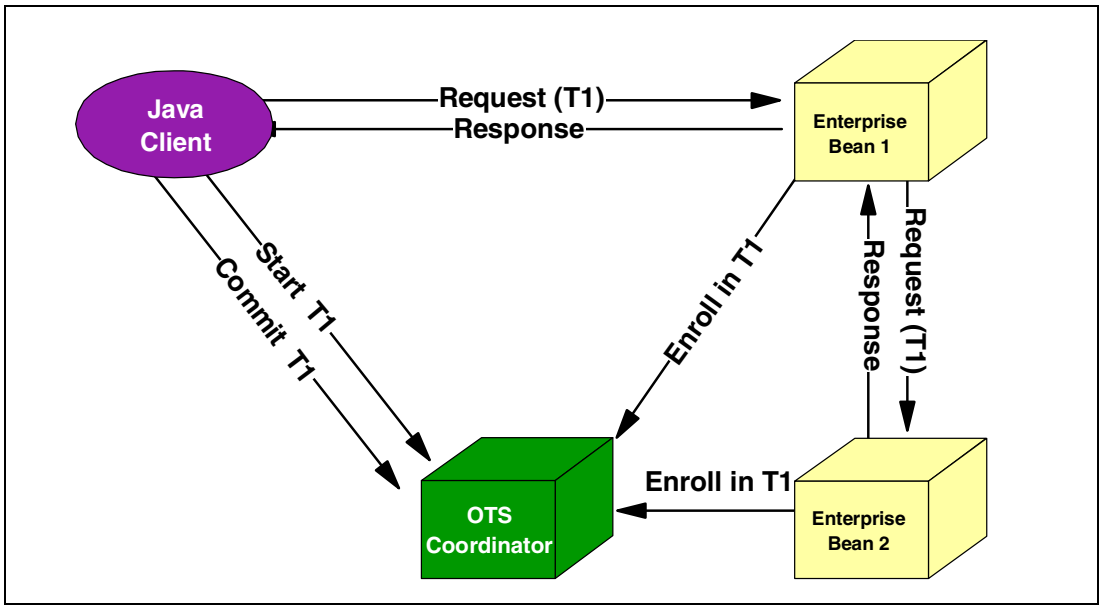


Figure 1-2 EJB transaction propagation

That is not to say that the enterprise beans involved necessarily participate in the transaction. The transaction context which is sent with the method invocations is an invitation to participate in the transaction, but the invitation may be declined. Whether or not to participate in a transaction is defined by the operational attributes of the enterprise bean which are held in an associated deployment descriptor. The transactional attributes of the bean that may be specified in the deployment descriptor are as follows:

Mandatory: The enterprise must be invoked with a transaction context. If there is no transaction context, then an error response is returned.

Never: The enterprise bean must not be invoked with a transaction context. If the method invocation contains a transaction context, then an error response is returned.

NotSupported: The enterprise bean may be invoked with a transaction context, but it will be ignored. The transaction will be suspended for the duration of the method.

Supports: The enterprise bean may be invoked with or without a transaction context. If the method invocation contains a transaction context, then the execution of the method will form part of the transaction; otherwise the method execution will not participate in a transaction.

Required: The enterprise bean requires a transaction. If the enterprise bean is invoked without a transaction context, then a new transaction is started for the execution of the method. The life of this transaction is the duration of the method execution. On the other hand, if the enterprise bean was invoked with a transaction context, then the method will be executed as part of this transaction.

RequiresNew: The enterprise bean will always start a new transaction irrespective of any transaction context passed with the method invocation. If a transaction context was passed with the method invocation, then the transaction will be suspended for the duration of the method execution. A new transaction is started prior to the execution of the method and this new transaction is terminated upon completion of the method.

It is clear that a key role played in this activity is the OTS Transaction Coordinator. It needs to be made aware of all participants in any transaction so that it can effect a two phase commit protocol with the participants upon successful completion, or conversely, notify all the participants to roll back any database changes.

The OTS Transaction Coordinator is made aware of the participating enterprise beans by the EJB containers within the EJB servers. The container code which intercepts the enterprise bean method invocations uses the enterprise bean's transaction attribute to decide if the enterprise bean is to participate in an existing transaction. If it is, then the enterprise bean is registered with the OTS Transaction Coordinator as participating in the transaction. The container locates the OTS Transaction Coordinator by using the reference to it which is passed as part of the transaction context in the method invocation.

Transactions: The term *transaction* is used to describe a recoverable unit of work. This equates to the CICS term *unit of work*, which was previously termed a logical unit of work in earlier CICS releases.

Bean-managed OTS transactions

As a means of simplifying application development it is recommended that transaction management be left to the container as described in the previous section. However, an application programming interface is provided to allow the developer to explicitly set OTS transaction boundaries. This interface is called the Java Transaction API (JTA).

The Java Transaction API is used by obtaining an instance of the class *UserTransaction* from the enterprise bean's session context and then invoking methods on the instance to request transaction management services. Here are four of the methods provided by the JTA:

`begin()`	Start a new transaction
`commit()`	Commit the current transaction
`getStatus()`	Retrieve the status of the current transaction
`rollback()`	Roll back the current transaction

The fragment of code shown in Figure 1-3 is an example of a bean-managed transaction.

```
public void deposit(int amt) throws AccountException{
/* get a user transaction object from the session context */
      javax.transaction.UserTransaction userTran = ctx.getUserTransaction();
/* start the transaction */
      userTran.begin();
/* update the balance */
      balance += amt;
/* code to update the database */
/* commit the transaction */
      try{
          userTran.commit();
      }
      catch (Exception e){
          throw new accountException("error:"+e.toString());
      }
}
```

Figure 1-3 An example of a bean-managed transaction

1.1.3 Isolation

Concurrency control is an important part of transaction processing system. Section 1.1.2, “Transactionality” on page 4 covered the properties of a transaction which ensure that database integrity is maintained. One of these properties is *isolation*. The term isolation refers to the manner in which concurrent access to shared information is controlled.

A problem arises when multiple clients wish to refer to the same item of information and the information is to be updated.

As an example, consider an enterprise bean that carried out the following:

1. Read A from a database
2. Add 5 to A
3. Write back A to the database

If these actions always take place in this order, then there is no problem. However, in a multi-threaded system, this cannot always be guaranteed.

If the enterprise bean were now invoked by two clients without concurrency control, the following sequence of operations could occur:

1. Instance 1 reads A from the database. A holds 0.
2. Instance 2 reads A from the database. A holds 0.
3. Instance 1 adds 5 to A and updates the database with A set to 5.
4. Instance 2 adds 5 to A and updates the database with A set to 5.

Because of the switching that occurs between the two threads running the two enterprise bean instances, the first update of A to the database has been lost.

Transaction management systems prevent this from happening by using a process of locking. In this example when Instance 1 reads A from the database it would be locked; this lock would not be released until Instance 1 had updated the database with the new value of A. Similarly, Instance 2 would request the same lock so that it could read A from the database, but in this case the lock would already be held by Instance 1. This would result in Instance 2 waiting until Instance 1 released the lock. Having obtained the lock, Instance 2 would update A and then release the lock, leaving A set to the correct value of 10.

However, there is a certain amount of overhead in managing locks, and contention for shared resources which are locked causes delays and may cause deadlocks. So the EJB server should only use locks when they are necessary. An enterprise bean that simply reads an item from the database and returns it to its client would not need any locks to be set.

The deployment descriptor of the enterprise bean may be used to communicate the isolation level of the enterprise bean to the EJB server. The isolation levels only apply to enterprise beans that participate in a transaction. The four isolation levels are:

1. **Read uncommitted:** This level does not provide any isolation guarantee, and means that a *dirty* read is performed. Thus the data read may actually have been changed by another user, but may not have yet been committed.
2. **Read committed:** This level ensures that any data item read from the database has been committed. This prevents accessing data which has been changed by another user, but has not yet been committed.
3. **Repeatable read:** This level guarantees that a set of database rows may be read and that upon re-reading the same set, the same values will be returned. In other words, all of the rows in the set are locked when first read.
4. **Serializable:** This level provides full transaction isolation, so that the data is locked before it is accessed.

The four isolation levels may be specified as applying to the entire enterprise bean or they may be specified for individual methods.

1.1.4 Security

The secure use of an application involves the following three considerations:

- **Authentication:** The validation of the identity of the user. The user normally enters a user identification and a password, and once verified, the user is free to use the application.
- **Access Control:** The control of what a user may or may not do within the application.
- **Secure Communication:** This normally involves the use of encryption techniques so that the information flowing across the communication link cannot be understood by a third party.

The EJB specification is concerned with *access control.*

When a user logs on to the system, a process of authentication will take place in order to establish the user's security identity for the duration of the session. The security identity as defined in the EJB specification may be that of an individual user or a *role*.

A role is a logical security identification which is mapped to real users or user groups in the environment where the enterprise beans are to be deployed. For example, in the OS/390 environment, the roles could be mapped to RACF group profiles.

When a security identity has been established it will be implicitly passed with the method invocations.

The EJB 1.1 specification defines the class *java.security.Principal* to represent the security identity. This class is used by the EJB access control architecture running in the EJB server to ensure that method invocations are permitted for the role of the user.

The roles are defined using the XML deployment descriptor. Figure 1-4 shows the definition of two security roles: *Administrator* and *ReadOnly*. At this stage these are simply role names; they will subsequently be used in permission definitions.

```
<security-role>
    <description>
        A user with this role may access any method.
    </description>
    <role-name>
        Administrator
    </role-name>
</security-role>
<security-role>
    <description>
       A user with this role is not allowed to make changes
    </description>
    <role-name>
        ReadOnly
    </role-name>
</security-role>
```

Figure 1-4 Definition of security roles in the XML deployment descriptor

The association between security roles and the enterprise bean methods is defined by using *method-permission tags* in the deployment descriptor. Figure 1-5 shows an example of giving the security role *Administrator* access to all the methods in an enterprise bean named Account.

```
<method-permission>
    <role-name>Administrator</role-name>
        <method>
           <ejb-name>Account</ejb-name>
               <method-name>*</method-name>
        </method>
</method-permission>
```

Figure 1-5 Access rights for the administrator role

Figure 1-6 shows the definition of the methods which the role ReadOnly may access. A user in the ReadOnly role may only invoke the methods `getName()` and `getBalance()` on the Account enterprise bean.

```
<method-permission>
    <role-name>ReadOnly</role-name>
        <method>
           <ejb-name>Account</ejb-name>
               <method-name>getName</method-name>
               <method-name>getBalance</method-name>
        </method>
</method-permission>
```

Figure 1-6 Access rights for the ReadOnly role

The enforcement of these access rights is performed by interposed code in the EJB container when the enterprise bean is invoked. This interposed code is generated automatically when the enterprise bean is deployed into the container. The enterprise bean developer need have no awareness of the security controls that are assigned when the enterprise bean is deployed.

If the enterprise bean invokes methods on other enterprise beans, then the security identity is propagated throughout the entire invocation hierarchy.

Querying security information

The enterprise bean session context object can be queried by the application code to retrieve security information. This should not be necessary except in special circumstances, as using this facility introduces security management function into the application code. The following two methods are provided which return information about the security context:

- **isCallerInRole(String s)** returns *true* if the caller is in the security role specified as the single string argument; otherwise, *false* is returned.
- **getCallerPrincipal()** returns an object of the class java.security.Principal which may then be used to extract more information such as the distinguished name which is associated with this session.

Important: *CICS TS V2.1 provides limited support for security as defined in the EJB specification. In particular, it does not support role base security and therefore the above method* `IsCallerInRole()` *will always return* `true`*. The specification will be supported more fully in a subsequent version of CICS.*

1.2 Enterprise beans

In this section we look at enterprise beans themselves. There are two types of enterprise beans: session beans and entity beans, which we shall now describe.

1.2.1 Session beans

Session beans define the business logic of an application. They are effectively an extension of the client that invokes them. Once instantiated, the client invokes methods on the enterprise bean in the same way as it would on any other object.

Session beans are server-side business components that can be deployed into any J2EE compliant EJB server. They are located by looking up their names in a name server using JNDI. There are two types of sessions beans, stateful and stateless; the differences are described further, later in this section.

Normally an instance of a session bean will be relatively short-lived. It will provide its service to the client and then be removed. The life of the session bean instance will typically be the same duration as the life of the client session, hence the term session bean.

The parts of a session bean

A session bean is composed of the following:

- **Home interface:** This specifies the lifecycle events such as the creation of a session bean instance.
- **Remote interface:** This specifies the business methods that are to be exposed to the clients that will use the bean.
- **Bean implementation:** This contains the actual implementation of the business logic together with the implementation of lifecycle methods that will be invoked by the container.
- **Deployment descriptor:** This gives the container operational information about the enterprise bean, such as security access rules and transactional attributes.

The home interface

The home interface extends the class EJBHome and it must contain at least one `create()` method. Figure 1-7 shows the home interface for the enterprise bean Account.

```
public interface AccountHome extends javax.ejb.EJBHome{
    public Account create() throws java.rmi.RemoteException,
                                   javax.ejb.CreateException;
}
```

Figure 1-7 Home interface for the enterprise bean Account

The remote interface

The remote interface exposes the methods in the enterprise bean implementation that may be used by a client. The remote interface extends the EJBObject class. Figure 1-8 is an example of the remote interface definition for the enterprise bean Account.

```
public interface Account extends javax.ejb.EJBObject{
    public int getBalance() throws java.rmi.RemoteException;
    public void setBalance(float amount) throws java.rmi.RemoteException;
}
```

Figure 1-8 Remote interface for the enterprise bean Account

The bean implementation

The code that actually carries out the business logic is written as part of the bean implementation. By convention the name given to this class is the name of the remote interface with the suffix *Bean* appended to it. The bean implementation class must implement the SessionBean interface. This interface defines the methods which are invoked by the container to notify the enterprise bean of lifecycle events. For example, the method `ejbCreate()` is invoked whenever a new instance of the enterprise bean is created. The Java compiler forces these methods to be implemented in the class. However, if no action is to be taken on these events, then the methods may be implemented with null method bodies.

Figure 1-9 shows the implementation of the Account enterprise bean.

```
public class AccountBean implements javax.ejb.SessionBean{
// instance variable to hold the balance
private int balance;
// management methods invoked by the container
    public void ejbCreate(){}
    public void ejbRemove() {}
    public void ejbPassivate(){}
    public void ejbActivate() {}
    public void setSessionContext(SessionContext ctx){}
// business methods exposed in the remote interface
    public int getBalance(){
        return balance;
    }
    public void setBalance(float amount){
        balance = amount;
    }
}
```

Figure 1-9 Implementation of the enterprise bean Account

The deployment descriptor

The deployment descriptor provides declarative information to the container about the enterprise bean. The EJB 1.1 deployment descriptor is an XML document. Typically the deployment descriptor is generated by the application development tools. The transactional attributes specified by the deployment descriptor are discussed in 1.1.2, “Transactionality” on page 4. The security access definitions which may be specified in the deployment descriptor are discussed in 1.1.4, “Security” on page 8.

For details on how we actually modified the deployment descriptor using the CICS JAR development tool, refer to Figure 10-17, “CICS JAR development tool, environment properties for DB2” on page 273.

Session bean deployment

The developer defines the enterprise bean’s home and remote interfaces and the implementation code for the bean itself (Figure 1-10).

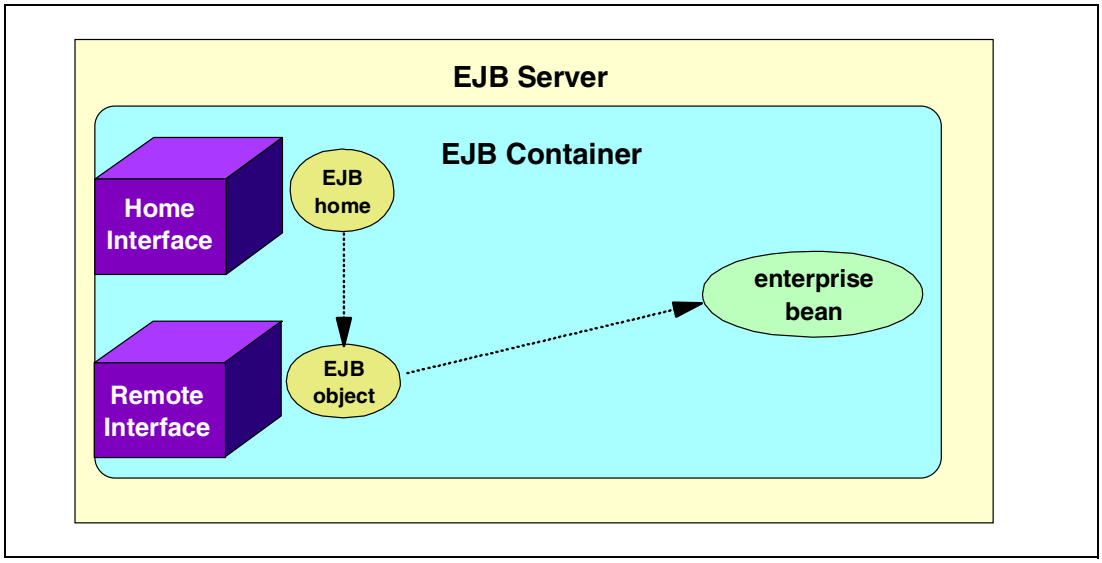


Figure 1-10 Deployed enterprise bean

The deployment process generates additional code that is necessary for the execution of the enterprise bean. The code generated is used both on the client machine and on the EJB server. The definition of the home interface is used to generate an EJB home stub for the client and an EJB home class for the EJB server. The definition of the remote interface is used to generate an EJB object stub for the client and an EJB object class for the EJB server. The class files for the enterprise bean are packaged into a Java Archive File (JAR) and this file is used for deploying into the EJB server.

After the enterprise bean has been deployed into an EJB server, one further step is necessary. The name of the enterprise bean must be published. The EJB server registers the name of the enterprise bean with a naming server using the Java Naming and Directory Interface (JNDI).

The name of the bean is registered using the JNDI with a reference to the enterprise bean's home interface. The reference is known as an interoperable object reference (IOR). The IOR includes the precise network location of the enterprise bean, including the server name and port.

Using a session bean

Figure 1-11 illustrates the control flows that take place in order to locate the session bean, to ask the container to create an instance of it, and then to invoke methods on the instance.

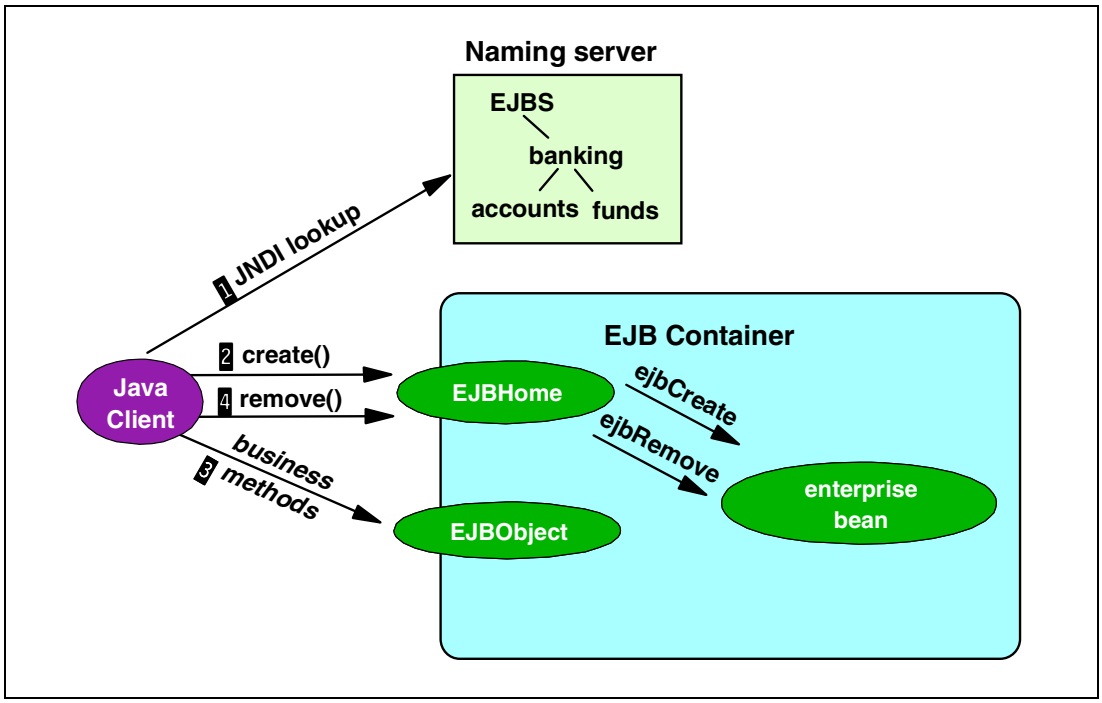


Figure 1-11 Session bean invocation

- 1 The client program, which may be a Java application, an applet, a servlet, or indeed another enterprise bean, must first know the name of the enterprise bean. The client sends the name to a naming service using JNDI in order to locate the enterprise bean. If the naming service has registered the enterprise bean's name, the IOR of the enterprise bean's home interface is returned to the client.
- 2 Using the IOR, the client has an object reference to the enterprise bean's home interface. The `create()` method is invoked on the enterprise bean's home interface, which causes the container to drive the `ejbCreate()` method to instantiate the bean itself. The

container responds with a reference to the enterprise bean's remote interface. The reference to the remote interface is an indirect reference to the enterprise bean's business methods. This indirect reference points at container code that imposes security and transactions and then invokes the real business method in the bean implementation.

- 3 The client then invokes the business method (or methods) on the remote interface, to perform the desired business functions.
- 4 Finally the client calls the `remove()` method on the home interface; this causes the container to drive the `ejbRemove()` method on the enterprise bean itself.

An example of the client code to use the Account enterprise bean is shown in Figure 1-12.

```
/* obtain a JNDI initial context "/
     javax.naming.InitialContext jndiCtx = new InitialContext();
/* look up the home interface of the enterprise bean */
     Object o = jndiCtx.lookup("pfx/Account");
/* convert the object returned to an AccountHome object */
     AccountHome anAccountHome = PortableRemoteObject.narrow(o,AccountHome.class);
/* create an instance of the Account enterprise bean */
    Account anAccount = anAccountHome.create();
/* invoke the business methods of the enterprise bean */
    anAccount.setBalance(5);
System.out.println("Balance: "+anAccount.getBalance());
/* finished, so remove the instance */
    anAccountHome.remove();
```

Figure 1-12 Client access to an enterprise bean

As mentioned earlier, there are two different types of session beans, stateful and stateless, which we will now discuss.

Stateless session beans

Stateless session beans do not preserve any conversational state across method calls. Each method call acts on the arguments passed and returns a response without saving any state in its instance variables.

Stateless session beans must complete their transactions within single method invocations; transactions cannot span method invocations.

As they retain no conversational state, stateless session beans are not tied to any particular client. This means that any client request can be routed to any stateless session bean. There is a benefit here in that stateless session bean instances may be pre-created and pooled so that they are ready when client requests arrive. By eliminating instantiation at the time of the request, the performance can be improved.

Stateful session beans

A stateful session bean does retain conversational state. The stateful session bean instance is specific to the client that instantiated it. An OTS transaction may span multiple invocations to methods in stateful session beans. The stateful session bean instance exists for the duration of the conversation with the client.

As there is a one-to-one relationship between clients and stateful session bean instances, this may result in excessive demands on memory if a large number of clients are in session.

To alleviate these memory demands, the conversational state may be written from memory to auxiliary storage pending the next method invocation on the enterprise bean instance. This is termed *passivation*. Different EJB containers can use different algorithms to decide which bean instances to passivate. The CICS container usually passivates beans at the earliest opportunity.

The conversation state is read back into memory when the subsequent method invocation is received. This is termed *activation* and occurs when another method invocation is received for the enterprise bean instance.

The enterprise bean is notified by the container when it is about to be passivated by an invocation of the method `ejbPassivate()`. In a similar fashion the enterprise bean is notified by the container when it has just been activated by an invocation of the method `ejbActivate()`. If the enterprise bean is an active participant in an OTS transaction it will not be a candidate for passivation, however, once the OTS transaction has been committed or rolled back, it again becomes a candidates for passivation.

The conversational state of a stateful session bean is not recoverable. If a system failure occurs during the lifetime of stateful session bean instance, then the conversational state would be lost, even if it had been passivated.

Attention: Note that passivation and activation do not apply to stateless session beans, since stateless session beans do not hold state, and so can simply be created and destroyed rather than passivated or activated.

1.2.2 Entity beans

An entity bean is an object representation of data such as a customer or an account. Usually an instance of an entity bean corresponds to a row in a relational database. This allows the data to be manipulated in a normal object-oriented manner by invoking methods on the entity bean. Unlike session beans, entity beans live for as long as the data they represent lives.

Entity beans are shared by multiple users. For example, more than one user may wish to access the same account information simultaneously.

An entity bean is uniquely identified by a primary key.

Restriction: CICS TS V2.1 does not support entity beans. Session beans are a natural extension to the existing transactional capabilities of CICS; however, entity beans have no obvious mapping to existing CICS functionality.

Bean persistence

Bean persistence means writing the data held in the entity bean to its associated database.

This involves accessing the instance variables in the entity bean and, for example, using them in an SQL statement to update the database. It is a matter of synchronizing the data in the entity bean with the data held in the database.

The synchronization of the entity bean and its associated database row may be performed by either the entity bean itself or by the container running in the EJB server. These alternative ways are called *bean-managed* persistence or *container-managed* persistence.

Entity beans, like session beans, are server-side components that may be deployed into any EJB server that supports them. They are located by name using a name server and the JNDI.

The parts of an entity bean

An entity bean consists of the following:

- **Home interface:** This is used by the client to create, find, and destroy entity bean instances.
- **Remote interface:** This serves the same purpose as in session beans. It specifies the business method signatures that are to be exposed to the clients that will use the bean.
- **Bean implementation:** This is a class which is a model of the persistent data, for example, a row in a relational database table. It includes methods to manipulate the data and lifecycle methods that are invoked by the container.
- **Primary key class:** This class represents a unique identification of the entity bean.
- **Deployment descriptor:** This provides operational information about the entity bean.

The home interface

The home Interface extends EJBHome and is used by the client to create, find and destroy entity beans. Unlike session beans, it is not obligatory to have a `create()` method.

Figure 1-13 shows the home interface for the entity bean Account.

```
public interface AccountHome extends EJBHome{
    public Account create(String id, String name)
                                              throws RemoteException,
                                                     CreateException;
    public Account findByPrimaryKey(AccountPK key)
                                              throws FinderException,
                                                     RemoteException;
}
```

Figure 1-13 Home interface for the entity bean Account

The remote interface

The remote interface exposes the methods that the client may use to manipulate the entity data. The remote interface extends the EJBObject class.

Figure 1-14 is an example of the Remote interface definition for the entity bean Account.

```
public interface Account extends EJBObject{
    public String getName()                              throws RemoteException;
    public void setName(String name)               throws RemoteException;
    public int getBalance()                        throws RemoteException;
    public void setBalance(int amount)             throws RemoteException;
}
```

Figure 1-14 Remote interface for the entity bean Account

The bean implementation

This is a Java class which provides a view into the database table row that it represents.

It must define instance variables for the various elements of data that are to be represented. These instance variables could correspond to columns in a database table row, although that is at the discretion of the application designer.

It must define methods which allow its clients to manipulate the instance variable.

The bean implementation class must implement the EntityBean interface, which enforces the implementation of a series of lifecycle methods that are invoked by the container. Figure 1-15 shows the implementation of the Account entity bean.

```
/* The Account entity bean implementation with container-managed persistence */
public class AccountBean implements EntityBean{
/* instance variable to hold the persistent data*/
    public String accountNumber;
    public String accountName;
    public int      balance;
/* management methods invoked by the container */
    public void ejbLoad(){}
    public void ejbStore(){}
    public void ejbCreate(String id, String name){
        accountNumber = id;
        accountName    = name;
        balance             = 0;

    }
    public void ejbPostCreate() {}
    public void ejbRemove() {}
    public void ejbPassivate(){}
    public void ejbActivate() {}
    public void setEntityContext(EntityContext ctx){}
    public void unsetEntityContext(){}

/* methods to manipulate the persistent data              */
/* these methods are exposed in the remote interface */
    public int getBalance(){
        return balance;
    }
    public void setBalance(int amount){
        balance = amount;
    }
}
```

Figure 1-15 Implementation of the entity bean Account

Primary key class

This is a class which defines a unique identification for the entity beans. The commonly used naming convention for this class is the entity bean name suffixed by “PK”. Figure 1-16 shows the Primary key class for the enterprise bean Account.

```
public class AccountPK implements java.io.Serializable{
    public String accountNumber;
    public AccountPK(String s){
        accountNumber = s;
    }
    public AccountPK() {}
    public String toString(){
        return accountNumber;
    }
}
```

Figure 1-16 Primary key class for the enterprise bean Account

Deployment descriptor

This is used to pass operational information about the entity bean to the container in the EJB server. In the Account entity bean example container-managed persistence is used, so the deployment descriptor also contains object to relational mapping details. How this is done depends on the container provider. For the Account example, the mapping could be as shown inTable 1-1.

Table 1-1 Object to relational mapping

Instance variable name	Table column name
`accountNumber`	ACCOUNT_NUMBER
`accountName`	NAME
`balance`	BALANCE

Bean managed persistence

In this case, explicit code must be written to transfer data between the entity bean and the database. If the database is a relational database, then the JDBC API could be used to achieve this.

How does the entity bean know when to do this? The container tells it by invoking the entity bean methods `ejbLoad()` to read the data from the database and `ejbStore()` to write the data to the database.

Container managed persistence

This technique permits the relationship between the instance variables in the entity bean and the columns in the database table to be defined declaratively. This has an advantage for the developers in that they can concentrate on the business logic without concerning themselves with how the underlying data is held.

The deployment descriptor is used to provide the mapping between the instance variables and the database columns. The container refers to the deployment descriptor to know which database accesses are needed to ensure that the entity bean and its corresponding database row are kept in synchronization.

Entity bean deployment

The steps necessary to deploy an entity bean are outlined in “Session bean deployment” on page 12.

Using an entity bean

The client program, which may be a Java application, an applet, a servlet, or indeed another enterprise bean, must first know the name of the enterprise bean.

The client sends the name to a naming service using JNDI in order to locate the entity bean. The naming service responds with the IOR of the entity bean’s home interface. Using the IOR, the client has an object reference to the enterprise bean’s home interface. The entity bean’s home interface may then be used to create, find or remove entity beans. For example, the `findByPrimaryKey()` method responds with a reference to the entity bean’s remote interface. The reference to the remote interface is an indirect reference to the enterprise bean’s business methods. This indirect reference points at container code that imposes security and transactions and then invokes the real business method in the bean implementation.

The client then invokes the business methods on the remote interface. An example of the client code to use the Account enterprise bean is shown in Figure 1-17.

```
/* obtain a JNDI initial context "/
     javax.naming.InitialContext jndiCtx = new InitialContext();

/* look up the home interface of the enterprise bean */
     Object o = jndiCtx.lookup("pfx/Account");

/* convert the object returned to an AccountHome object */
     AccountHome anAccountHome =PortableRemoteObject.narrow(o,AccountHome.class);

/* Obtain a reference to the entity bean for account number 12345*/
     Account anAccount = anAccountHome.findByPrimaryKey(new AccountPK("12345"));

/* invoke the business methods of the enterprise bean */
    anAccount.setBalance(5);
    System.out.println("Balance: "+ anAccount.getBalance());
```

Figure 1-17 Client access to an entity bean

1.2.3 Database access

An enterprise bean may access a relational database in three different ways. It may use JDBC, SQL, or Data Access beans. The following sections describe these techniques. To find out more about how we used JDBC and SQLJ within a CICS enterprise bean refer to Chapter 10, "Rewriting the Trader session bean using JDBC/SQLJ" on page 253.

Java Database Connectivity (JDBC)

JDBC is a uniform API for accessing relational databases. It is provided by the java.sql package which is one of the core APIs provided by the Java SDK, standard edition and the enterprise bean the JDBC Optional Package API provided by the javax.sql package.

The purpose of JDBC is to allow the developer to issue SQL statements to access a relational database without the need to code the program in a database specific way. This fits with the Java style of "write once, run anywhere".

Using JDBC, the developer is provided with an object-oriented API to access relational data. A selection of the classes and interfaces supplied with JDBC is shown here:

- Database connections
- SQL statements
- Result sets
- Prepared statements
- Callable statements
- Database drivers
- Driver managers

JDBC comprises a Driver Manager and a specific database driver for each type of database accessed. Figure 1-18 shows a schematic view of the JDBC structure.

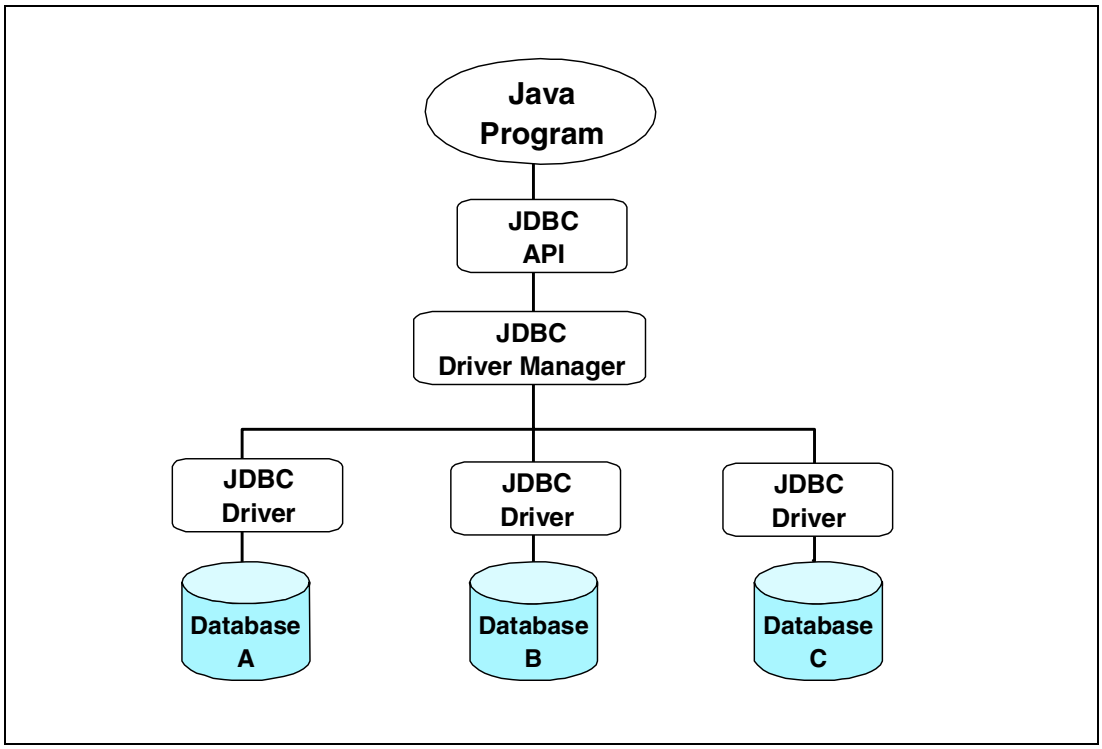


Figure 1-18 JDBC structure

It is the JDBC driver that converts the JDBC API request into the format required by the specific database manager. This may be a direct API call to a local database manager or a network request to a remote database. There are four types of JDBC drivers:

- Type 1: JDBC-ODBC bridge
- Type 2: Java to native API
- Type 3: Java to network protocol
- Type 4: Java to database protocol

The JDBC driver type that has been used in this redbook project (see Chapter 10, “Rewriting the Trader session bean using JDBC/SQLJ” on page 253), is a type 2 driver for access to DB2. This driver type is normally the most efficient, as it uses the Java Native Interface (JNI) to handle the database requests and responses. However, the use of JNI means that the driver is platform specific and that the JNI code must be available on the client machine when database accesses are made.

Restriction: DB2 for OS/390 only provides the type 1 and type 2 JDBC drivers; CICS TS V2.1 only supports the type 2 driver for access to DB2.

The steps that the developer must take in order to access a relational database using JDBC are as follows:

- Register the JDBC driver with the driver manager.
- Obtain a connection to the database.
- Issue an SQL request.
- Process the results of the SQL request.
- Close the connection to the database.

Register the JDBC driver

The JDBC Driver Manager takes charge of all JDBC drivers which are available for use by JDBC client programs. Each JDBC driver must be loaded and it must register itself with the JDBC driver manager. In order to load and register a JDBC driver, the static method `forName()` of the class *Class* is invoked. Each JDBC driver is a Java class and so it must be available in the CLASSPATH when the `forName()` method is invoked. The following statement shows the Java code that is needed to load and register the DB2 driver which is used by the samples in this book:

```
Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
```

Obtain a database connection

Before an application can access a database, it must obtain a reference to a database connection object. It obtains this object from the JDBC Driver Manager using the static method `getConnection()`. The application program identifies the specific database it wishes to access by passing a database URL to the JDBC Driver Manager. The format of the database URL is as follows:

```
jdbc:<subprotocol>:<location name>
```

The meanings of these three URL components are as follows:

jdbc	This indicates that the JDBC API is to be used to access the database.
subprotocol	This tells the JDBC Driver Manager which driver to use.
location name	This identifies the database to be accessed.

The following is an example of the code required to obtain a connection:

```
Connection c = DriverManager.getConnection("jdbc:db2:databaseName");
```

Issue an SQL request

An SQL request is issued by first creating a Statement object and then invoking a method on the Statement object to execute an SQL statement.

The Statement object is obtained by invoking the `createStatement()` method on the Connection object.

The Statement object may then be used to execute a database query or a database update. The following piece of Java code shows an example of a simple database query: The result of the query is returned in another object, which is an instance of the class ResultSet that is described in the next section.

```
Statement stmt = c.createStatement();
ResultSet rs = stmt.executeQuery("select a, b, c FROM Table1"};
```

An SQL statement can also be pre-compiled and used multiple times by the use of a PreparedStatement object. Due to their being pre-compiled, PreparedStatement objects are much more efficient than normal Statement objects. The SQL statement encapsulated by a PreparedStatement object may also contain variables which can be set each time that this object is used. Figure 1-19 shows an example of the use of a PreparedStatement object:

```
/*   create a PreparedStatement ? indicates variable              */
PreparedStatement insert = c.prepareStatement(
     "insert into cars(car_id,make,model,size) values(?,?,?,?)"      );
/* set variable 1 to 4 then update the database */
    insert.setString(1,regNumber);
    insert.setString(2.manufacturer);
    insert.setString(3,carModel);
    insert.setInt(4,cubicCapacity);
    insert.executeUpdate(); /* insert the row into the database */
```

Figure 1-19 PreparedStatement object

Process the results of the SQL request

The ResultSet object returned by the `executeQuery()` method of the ResultSet instance contains the set of database rows that satisfied the selection criteria in the query. A cursor is maintained by the ResultSet object which is initially positioned just before the first row. Each row is retrieved by invoking the `next()` method on the ResultSet object. The individual column values of the current are retrieved by invoking `get`*`xxx`*`()` methods on the ResultSet object. The following Java code illustrates this:

```
while (rs.next()) {
    System.out.println(rs.getString("a"));
    System.out.println(rs.getString("b"));
    System.out.println(rs.getString("c"));
}
```

Close the connection to the database

When no further access to the database is required the connection and its associated objects should be closed by invoking the `close()` on each of the objects. For example:

```
stmt.close();              // close statement and result set objects
c.close();                 // close connection
```

The following Java code (Figure 1-20) shows a complete example of making a query to DB2 on OS/390:

```
try {
    /* load and register the driver, then get connection object    */
    Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
    String url = "jdbc:db2os390sqlj:";
    Connection c = DriverManager.getConnection(url);
    /*     get statement object and execute a query           */
    Statement stmt = c.createStatement();
    ResultSet rs = stmt.executeQuery("select name,balance from Table1");
    /*          display the values in the result set          */
    while (rs.next()){
        String name = rs.getString(1);
        int balance = rs.getInt(2);
        System.out.println(name + " " + balance);
    }
    /*     close the statement and connection            */
     stmt.close();
     c.close();
} catch (Exception e){
     System.out.println(e.getMessage());
}
```

Figure 1-20 Full JDBC query example

Note that in this example, the `get`*xxx*`()` methods which are used to retrieve the value of the columns in the result set, are invoked with a numeric argument to identify the database column. This is an alternative to using the column names. The number refers to the index of the column in the result set, where an index value of one represents the first column.

SQLJ

JDBC is a means of accessing a relational database using dynamic SQL whereas with SQLJ the database is accessed using static SQL, which is also known as embedded SQL.

A program written using static SQL contains SQL statements embedded in the program code. Before compilation the source program is passed through a translator which replaces the SQL statements with standard language statements to call the database services. The translated program is then compiled in the normal way. This is the way that static SQL is normally handled in other languages such as COBOL and PL/I.

One advantage with SQLJ is that type checking can be done during the program preparation process to determine whether table columns are compatible with Java host expressions. SQLJ also provides the advantages of static SQL authorization checking. With SQLJ, the authorization ID under which SQL statements execute is the plan or package owner. The database manager checks table privileges at bind time.

When the SQL statements can be pre-determined, which is the normal case in enterprise system application design, then the precompilation inherent in embedded SQL allows for faster execution at runtime and reduces program size and complexity.

The statements in a Java program which are to be expanded by the translator are identified by the token `#sqlj`.

The basic steps that the developer must take when accessing a database using SQLJ are as follows:

- Register the DB2 driver.
- Obtain a connection.
- Create a SQLJ connection context.
- Issue an SQL request
- Process the results of the SQL request.
- Close the connection to the database.

Registering the driver and obtaining and closing a connection are the same as with JDBC, They are described in “Java Database Connectivity (JDBC)” on page 19.

Create a SQLJ connection context.

Each embedded SQL statement is issued within a SQLJ connection context. This is either specified on the SQL statement itself or a default connection context is taken. In order to create a connection context the connection object returned from the driver manager is passed to the SQLJ connection context constructor as follows:

```
// declare context class, this will be expanded by the translator
#sql context ctx
// load and register driver then get a connection
Class.forName(“COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver”);
Connection c = DriverManager.getConnection(“jdbc:db2:databaseName”);
// get sqlj connection context
ivConCtx = new ctx(c);
```

Issue an SQL request

The SQL statement is embedded in the Java code using the `#sql` token to show that it has to be translated. An example of a SQL SELECT statement follows:

```
#sql [ivConCtx] i={SELECT c_name FROM trader_company};
```

Note that in this example the SQLJ connection context is explicitly specified as `ivConCtx`. The set of rows returned is placed into an object called an iterator which in this example is referenced by the variable `i`.

Process the results of the SQL request

The result set of an SQL request is placed into an iterator. To define an iterator the following construct is used:

```
#sql public static iterator iter_selectCompany (String c_name);
```

The translator generates a class to hold the returned set of the column named `c_name`. The class contains a method with the name `c_name()` which is used to retrieve the value of `c_name` at each position in the set. An implicit cursor is associated with the iterator object which is initially positioned immediately before the first row. The `next()` method is used to step through the rows in the iterator object. The `next()` method returns *false* if there are no more rows in the iterator object.

This example displays the set returned by the SELECT statement in the previous section:

```
while (i.next())
    System.out.println(i.c_name());
```

The following Java code shows a complete example of making a SQLJ query to DB2 on OS/390.

```
#sql context ctx
#sql public static iterator iter_selectCompany (String c_name);
try {
    // load and register the driver for DB2
    Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
    // get a connection object
    String url = "jdbc:db2os390sqlj:";
    Connection c = DriverManager.getConnection(url);
    // create SQLJ connection context
    ctx sqljCtx = new ctx(c);
    // issue a SQL SELECT statement
    iter_selectCompany i;
    #sqlj [sqljCtx] i={SELECT c_name FROM trader_company};
    // display the values returned in the iterator object
    while (i.next())
        System.out.println(i.c_name());
    // close the iterator and the SQLJ connection context
     i.close();
     sqljCtx.close();
} catch (Exception e){
     System.out.println(e.getMessage());
}
```

Figure 1-21 Full SQLJ query example

Program preparation

The actions required to prepare a program containing SQLJ statements for execution are shown in Figure 1-22

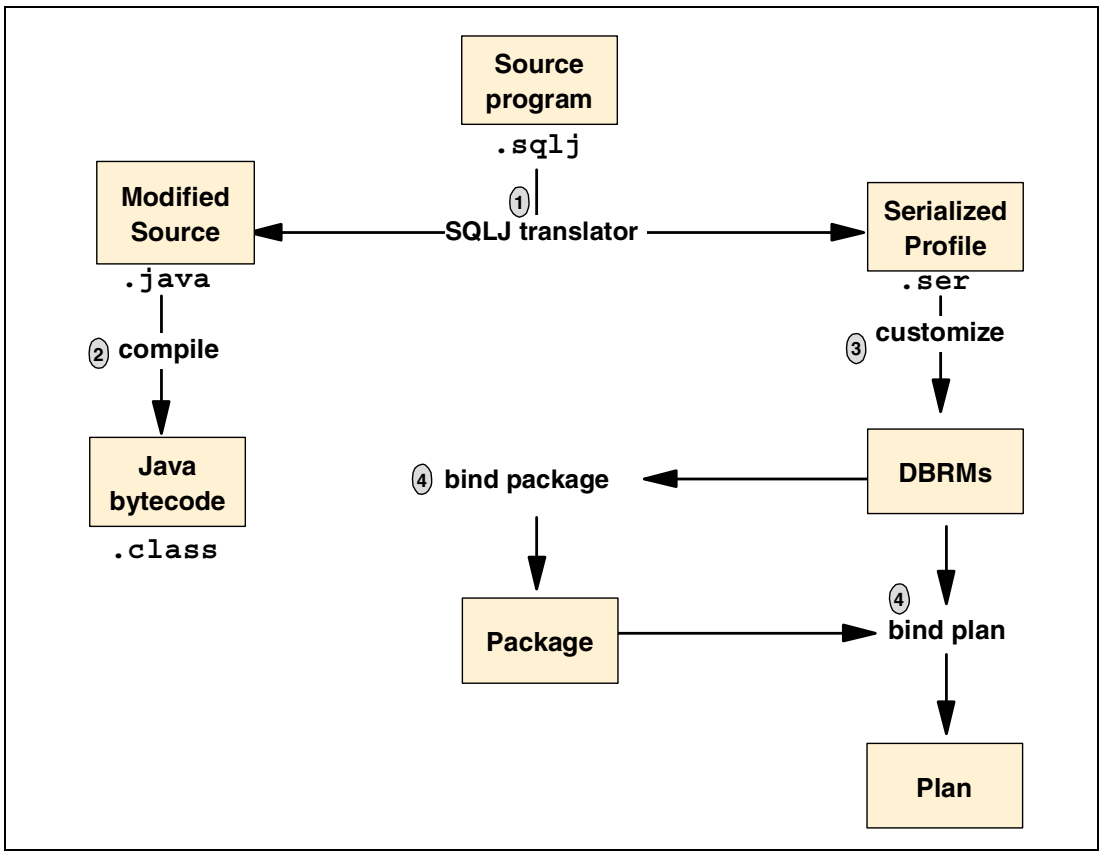


Figure 1-22 SQLJ program preparation

1. Translate the source code to produce modified Java source code and serialized profiles
2. Compile the modified Java source code to produce Java bytecodes
3. Customize the serialized profiles to produce DBRMs
4. Bind the DBRMs into packages and bind the packages into a plan, or bind the DBRMs directly into a plan

Data Access beans

Data Access beans are a level of abstraction for accessing data in relational databases. They are an alternative to using JDBC classes to access relational data. In effect Data Access beans wrap JDBC classes providing more function and making them easier to use. Data Access beans are JavaBeans *not* enterprise beans. They are a feature of VisualAge for Java.

The following types of Data Access beans are provided:

Select bean	Used to query relational data.
Modify bean	Used to modify relational data.
ProcedureCall bean	Used to run a database stored procedure.

The Select, Modify and ProcedureCall beans have properties that contain connection aliases and SQL specifications. These properties allow the user to connect to relational databases and access the data.

Figure 1-23 illustrates a method in an CICS session bean using a Data Access bean to access relational data.

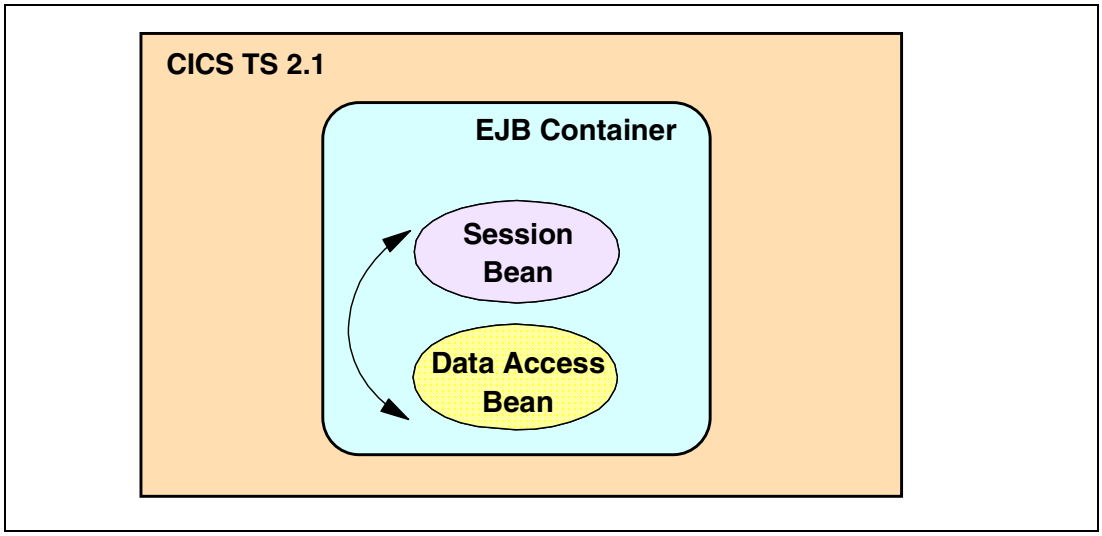


Figure 1-23 An enterprise bean using a Data Access bean to access relational data

The invocation of the methods on the Data Access bean are normal Java method invocations within the same Java Virtual Machine and so take place within the same transaction context as that of the enterprise bean. The resultant JDBC method invocations issued by the Data Access bean are converted by CICS into requests to the attached DB2 database and flow between CICS and DB2 using the CICS DB2 Attachment Facility. This facility uses a two phase commit protocol between CICS and DB2.

If an OTS transaction is in force, then all other participants in the transaction will be coordinated with any database updates made via the Data Access bean. The participating enterprise beans may be dispersed over multiple EJB servers.

In summary the use of Data Access beans to access relational data has the following advantages:

- They are more convenient to use than the JDBC classes.
- They function as part of the enterprise bean that calls them so they fall into its transaction context.

The following considerations apply when using Data Access beans in CICS:

- The installation of Visual Age for Java is a pre-requisite for their use as they are a feature of this product.
- There are special considerations for the use of Data Access beans in CICS. For further information refer to the CICS TS V2.1 supplied readme file `/usr/lpp/cicsts/cicsts21/doc/HOWTO/Data-Access-Beans-HOWTO`.

1.3 Enterprise bean interoperability

This section examines interoperability in a distributed object world. We discuss the mechanism for invoking methods on remote objects and the way that directories are used to locate them.

1.3.1 RMI

Remote Method Invocation (RMI) is the Java language's native mechanism for performing simple, powerful networking. RMI allows you to write distributed objects in Java, enabling objects to communicate in memory, across Java Virtual Machines, and across physical devices.

A remote procedure call (RPC) is a procedural invocation from a process on one machine to a process on another machine. A remote invocation in Java takes the RPC concept one step further and allows for distributed object communications. RMI allows you to invoke methods on remote objects. You can build your networked code as full objects. This yields the benefits of object oriented programming, such as inheritance, encapsulation, and polymorphism.

RMI has to deal with following issues:

- **Marshalling and unmarshalling:**

  To solve the problem that machines often have different data representations, it is necessary to encode and decode the data in such a way, that it can be passed between distributed machines. Marshalling and unmarshalling is the process of packing and unpacking parameters so that they are usable in two heterogeneous environments.

- **Parameter passing conventions**

  RMI supports pass-by-value as well as pass-by-reference when calling a method.

- **Distributed garbage collection**

  The Java language itself has a built-in garbage collection of objects. But in a distributed object system, garbage collection of remote objects cannot be done by the local JVM. Therefore, it is necessary to have RMI handle this issue.

- **Downloadable implementations**

  Since RMI supports the pass-by-value calling convention, objects (not just references) need to be brought from the local to the remote machine. However, it is possible that the class definition of that object is not available on the target machine. RMI allows for such class files to be automatically downloaded behind the scenes.

- **Security**

  RMI has support to restrict possible hostile implementations and grant system level access only to authenticated implementations.

- **Activation**:

  If a method on a remote object that is not in memory is invoked, RMI contains measures to automatically bring the object into memory so that it can service method calls.

1.3.2 RMI and EJB

Enterprise beans are made available for remote clients by using Java RMI. Enterprise beans are wrapped in RMI-aware shells, called EJB objects. EJB objects are the remote objects clients invoke. Therefore, when a client calls an EJB object, it delegates the remote call to the enterprise bean as illustrated in Figure 1-24.

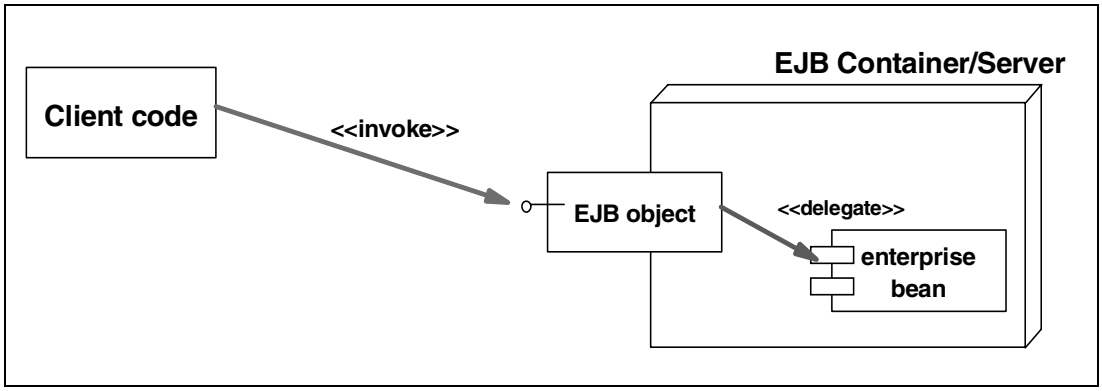


Figure 1-24 Invocation of an enterprise bean

Each enterprise bean must have a remote interface which duplicates every method signature which should be visible to the client. The remote interface has following characteristics:

- An EJB remote interface derives indirectly from java.rmi.Remote.
- Each method in an EJB remote interface must throw a java.rmi.RemoteException.

Each of the parameters of the remote interface's methods must be Java *primitives*, *serializable*, or *remote object*s in order to be valid types for Java RMI. By doing this you can control if the parameter is passed by value or passed by reference. In this manner remote object are passed by reference and Java primitives or serializable parameters are passed by value.

Each EJB object implements the remote interface which is automatically generated by the EJB container tools. This is because EJB objects must contain proprietary logic to:

- Interact with the container itself
- Implement fault tolerance for the case the bean crashes.
- Deal with marshalling and de-marshalling

1.3.3 JNDI

The Java Naming and Directory Interface (JNDI) is a standard Java extension which provides a common API to access naming and directory services. In a similar fashion to the abstraction offered by JDBC when accessing relational databases, JNDI allows the application developer to use the same API irrespective of that used by the underlying naming or directory service.

The actual underlying naming server provider can be any name service that provides an interface to the JNDI. CICS uses the CORBA Object Services (COS) naming service, as provided by WebSphere Application Server Advanced Edition, for this purpose, but other EJB implementations use the Lightweight Directory Access Protocol (LDAP) or a simple directory system. Furthermore, the underlying directory services may be interlinked heterogeneously, but JNDI still presents a common facade to the developer (Figure 1-25).

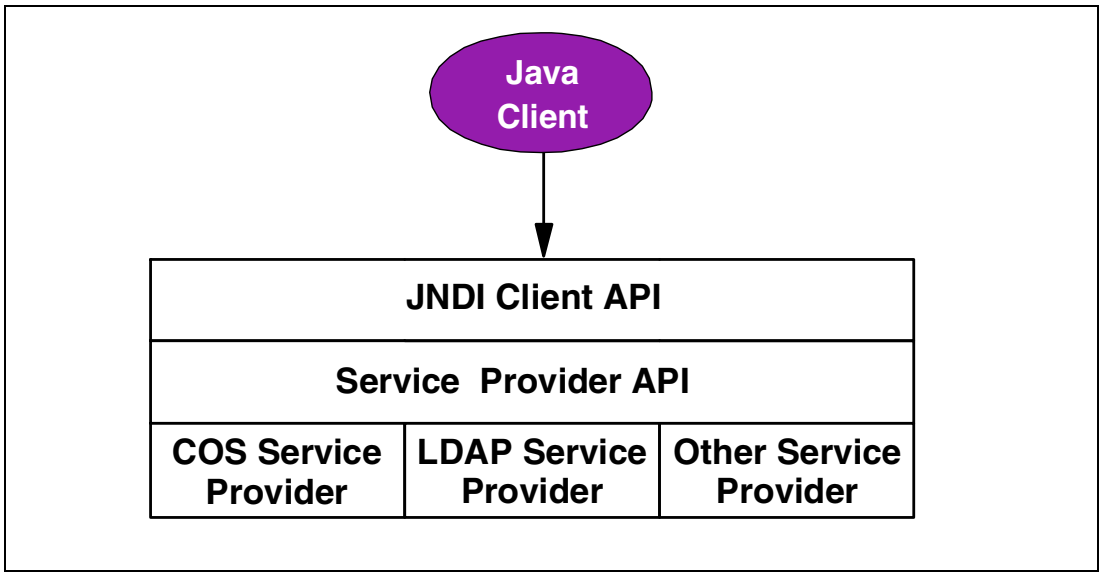


Figure 1-25 JNDI architecture

The structure of JNDI as shown in Figure 1-25 demonstrates that there is one client API which is the view that the client always sees together with a Service Provider Interface and multiple Server Provider plug-ins. The directory service providers write plug-in code compliant with the Service Provider Interface to couple their systems into JNDI. JNDI has already been extended in this way to support all the main directory service providers.

Any distributed object environment needs a way for servers to make known the objects that are available for use and for the clients to locate those objects. The Enterprise JavaBeans technology specifies that JNDI is the system to be used for enterprise beans.

When an enterprise bean is published by an EJB server its external name is associated with its home object reference. This process of association is known as binding into the directory structure. The EJB server sends a bind request to JNDI passing the external name and home object reference as arguments. JNDI then delegates the request to the underlying directory services such as the COS Naming service.

When a client wishes to use an enterprise bean it must first obtain a reference to the enterprise bean's home object. It does this by passing the bean's external name to the JNDI server and receiving the home object reference in response. The home object is then used to create an instance of the bean as covered in previous chapters.

The request to JNDI is passed using a *Context* object which tells JNDI where to find the naming server by specifying its URL, and what type of naming server it is by specifying an initial context factory. Figure 1-26 shows the Java code to use JNDI to obtain the reference to the home object for the Account enterprise bean.

```
//Use JNDI to get home object ref for Account enterprise bean
    Hashtable h = new Hashtable();

//Specify the URL of the name server
    h.put(Context.PROVIDER.URL,"iiop://hostname:900 ");

//Specify the type of name server as COS naming server
    h.put(Context.INITIAL.CONTEXT.FACTORY, "com.sun.jndi.cosnaming.CNCtxFactory ");

//Do the lookup
    try {
    Context ctx = new InitialContext(h);
    Object o =ctx.lookup("Account ");
    AccountHome a =PortableRemoteObject.narrow(o,AccountHome);
}
catch (NamingException e){
    System.err.println("jndi lookup failed ");
    System.exit(1);
}
```

Figure 1-26 Locating an enterprise bean using JNDI

The example code shown in Figure 1-26 shows how to create a JNDI Context object by passing the service provider URL and initial context factory from values stored in a hash table. To avoid hard coding these values in application code these properties may also be specified as an applet parameter, a system property, or in an application resource file. For example, the same result as the code in Figure 1-26 may be achieved by adding the following two lines to the JVM system properties file:

```
java.naming.provider.url=iiop://hostname:900
java.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
```

Then the code relating to the Hashtable can be removed and the Context object created with the default constructor with no arguments.

```
Context ctx =new InitialContext();
```

In addition, the defalt constructor can also be used if an enterprise bean within a container wishes to invoke methods on another enterprise bean within the same container (such as two beans within the same CICS CorbaServer). It is only necessary to explicitly code a name server if you need to invoke methods on an enterprise bean with a home interface located in a different naming server.

2

CICS TS V2.1: The EJB Server

In this chapter we provide an overview of the EJB Server as provided by CICS Transaction Server for z/OS V2.1 (CICS TS V2.1). We first describe the CICS road map, from recent years to the near future. Then we explain how the Java Virtual Machine (JVM) has been exploited to fulfill the special needs for high performance Java in a CICS environment. Next, we show how CICS supports IIOP. Finally, we explain the CICS EJB Server architecture and take a closer look at all its components.

2.1 The CICS Java road map

Java and the Internet have created a wonderful challenge for CICS customers. Many customers see the Internet as the way to directly reach their customers, and they see Java as the language of the future with a very real possibility that the “write once, run anywhere” motto will reduce programming costs and give them the freedom to deploy their applications on any platform.

Many CICS customers see a need to evolve from the 3270 procedural COBOL world to a Web-based, object-oriented Java world. The challenge is to evolve from where they are today to the future. They must effect this transition as quickly as practical in order to maintain their competitive advantage. But all the while they must maintain their current applications without sacrificing any of their mission critical attributes. CICS, the one constant in this transition, is transforming itself to enable customers to evolve (Figure 2-1).

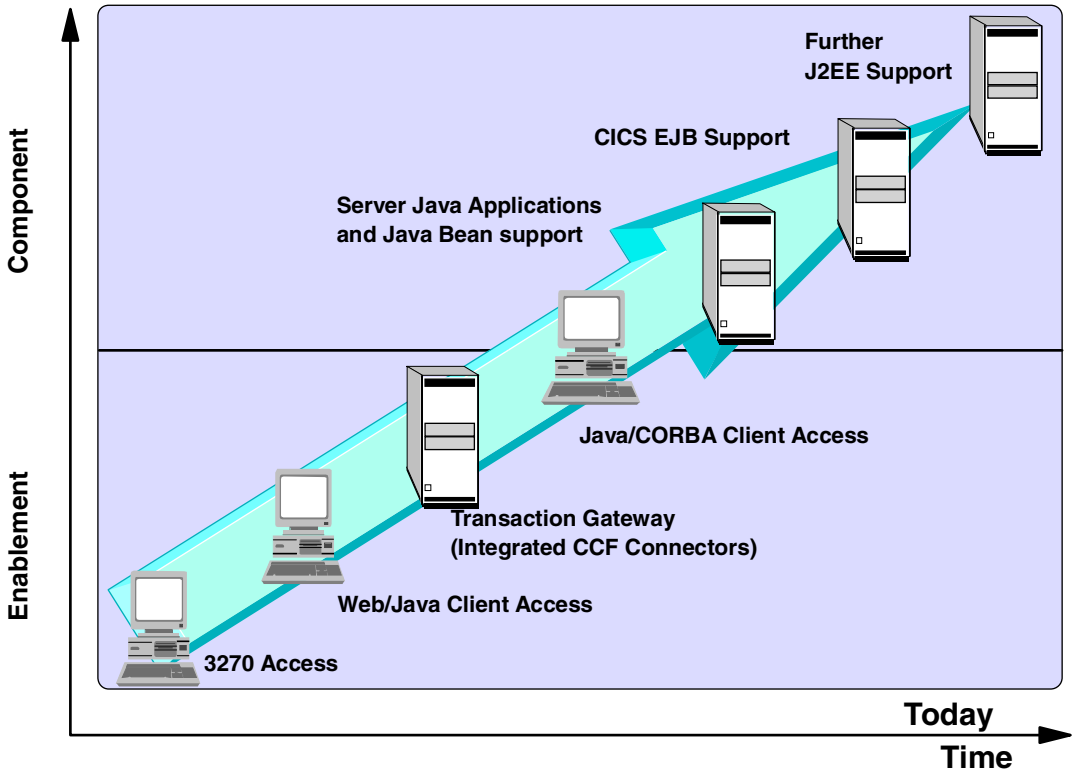


Figure 2-1 From 3270 access to J2EE

CICS TS V1.3, which became generally available in March 1999, already supports Web access to CICS programs, and 3270 transactions either directly through the facilities of CICS Web support (CWS) or through the CICS Transaction Gateway (CTG). It also supports access to CICS programs from CORBA clients using the standard IIOP protocol. Both HTTP and IIOP can exploit SSL for network privacy and integrity and, optionally, for client authentication.

In addition, CICS applications can be written in Java. The Java bytecodes can be interpreted by the standard OS/390 JVM or they can be compiled to S/390 machine code by the HPJ compiler that is provided with VisualAge for Java Enterprise Edition. The CICS Java programs can access CICS services and LINK to CICS programs in other languages, by means of the provided JCICS classes.

These facilities, combined with CICS scalability, availability and systems management have enabled many CICS customers to quickly Web-enable many of their applications and put them into production.

IBM's strategy is to transform CICS into an Enterprise Server for Java. By this we mean an e-business application server supporting, among other programming models, Enterprise JavaBean (EJB). In addition, support is given for Enterprise Java APIs and Common Connector Framework (CCF) connectors to important existing data sources and application servers. IBM intends to achieve this transformation, allowing customers to leverage their existing applications and retain all the mission critical characteristics CICS customers expect.

The first step along the path is to Web-enable applications, possibly on an intranet initially, to get the benefit of a familiar end user interface on a standard thin client.

CICS TS offers a number of ways to access CICS applications from a Web browser. It offers direct access to CICS without use of a Gateway or intermediate Web server. And it offers indirect access via the CICS Transaction Gateway for those customers who prefer a 3-tier solution.

CICS TS continues to evolve, constantly adding to its repertoire those aspects of the new technologies that are valuable to customers and that it can support well. Version 2 of CICS TS delivers a production ready implementation of an enterprise Java server, together with further support for TCP/IP connectivity, 3270 application reuse, and Parallel Sysplex technology. And, further down the road, the strategy is to support additional aspects of J2EE that are natural extensions for CICS TS (Figure 2-2).

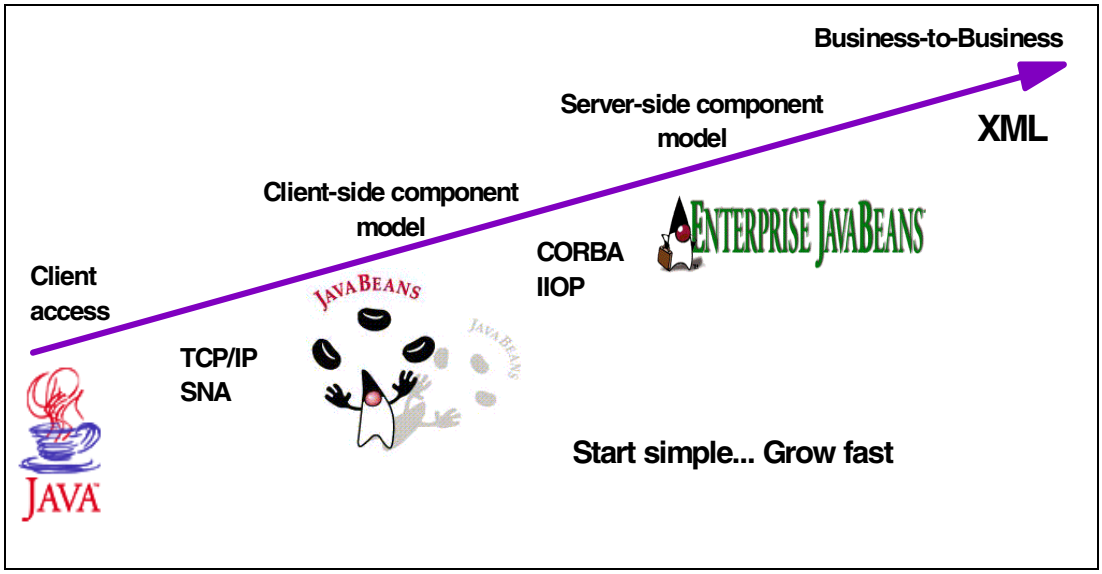


Figure 2-2 CICS e-business roadmap

2.2 The Java Virtual Machine

The main reason why it is possible to fulfill the “write once, run everywhere” paradigm of Java is the use of a Java Virtual Machine (JVM). A Java program is compiled with a Java compiler to a bytecode which is then interpreted by a JVM. Therefore, if you have a JVM for a specific operating system running on a dedicated hardware, you can in most cases run your Java program on that machine without adapting it to that platform.

A Java program is a long-running task, which means, at least a couple of seconds, but usually minutes to hours or days. Therefore, the time used to start up and initialize a JVM is relatively little in contrast to the time the actual Java program is running.

For enterprise beans, this behavior is different, especially in a CICS environment. Traditionally CICS is used as a high performance system handling hundreds to thousands of transactions per second. To accomplish this behavior, CICS applications should be designed in such a way that they perform their work in a minimum amount of time. Therefore, it is important to have as small as possible an overhead to run a CICS transaction. This behavior is in marked contrast to the Java programming model, because it would take longer just to start and initialize a JVM than to run the typical business logic of a CICS transaction.

A way to avoid a full JVM initialization every time is to reuse the JVM on subsequent invocations. This means that multiple transactions are executed serially in a single long-running JVM. This can be achieved because enterprise beans run as part of one transaction and do not interfere with any other transaction. This transaction isolation allows scaling to high volume workloads while retaining high levels of system integrity. This is the CICS quality of service. But to achieve transaction isolation for enterprise beans, each enterprise bean must be presented with a “clean” environment when it is run. This is provided by the OS/390 *persistent reusable JVM* which is the subject of the following section.

2.2.1 Features of the persistent reusable JVM

High performance Java is achieved with the following concepts:

- The serial reuse of a JVM for multiple transactions, while resetting the JVM to a known state between each transaction. This provides isolation without paying the high cost of a full JVM initialization for each transaction.
- An optimized garbage collection scheme, enabled by the clean separation of short-lived application objects from the long-lived classes, objects and native code (that is, non-Java or C language).

Serial reuse of JVMs

A JVM is dedicated to each CICS program for the lifetime of the CICS transaction. Initializing a JVM for each CICS transaction is very expensive, so the JVM is serially reused. There is only one transaction using a JVM at any one time. But when the transaction terminates, the JVM is made available for reuse by another transaction, for the same or a different end user and for the same or different CICS program.

From the application's perspective, the JVM appears to be short-lived even though it lives on to be reused by another Java program. In fact, the instances are isolated into separate Language Environment (LE) enclaves to eliminate interference between them and to enable hardware transaction isolation at some time in the future.

Serial reuse of a JVM is enabled by dividing the classes contained in the JVM into three parts:

- The OS/390 *JVM code*, which provides the base services in the JVM.
- The *middleware*, which provides services that access resources. These include the JCICS interfaces classes, JDBC, JNDI, and the enterprise bean runtime environment.
- The *user application* code.

Middleware classes have privileges that are not available to the application, and which enable optimizations through the caching of state (loading of classes and native libraries, for example) to be used by multiple applications. However, middleware is also responsible to reset itself correctly at the end of a transaction and, if necessary, to reinitialize at the beginning of a new transaction in order to isolate different applications from each other.

The trusted middleware class path property is built automatically by CICS from the paths specified on the CICS_DIRECTORY, JAVA_HOME, TMPREFIX, and TMSUFFIX parameters defined in the JVM profile (see the *CICS System Definition Guide*, SC34-5725 for details of these parameters and the trusted middleware class path). Generally speaking, only classes supplied by IBM, or your chosen middleware vendor, should be placed in the trusted middleware class path.

The reusable JVM manages run-time storage in several segregated heaps. The characteristics of these heaps differ considerably, as follows:

- **The transient heap:**

  The transient heap contains *objects* constructed by application classes. It also contains any non-shared application classes including their static data; non-shared application classes are those loaded from the CLASSPATH. This heap is subject to JVM garbage collection, which involves clearing the heap when the application completes. This is much quicker than traditional garbage collection.

- **The middleware heap:**

  This heap contains *objects* constructed by classes defined in the trusted middleware classpath. The objects in this heap are subject to normal garbage collection when no longer used. Note, however, that middleware code is designed to have a minimum of garbage collection; for example, object instances are pooled rather than used and discarded.

- **The system heap:**

  This is composed of two distinct heaps, both of which are never garbage collected:

  - **Main system heap:**

    This is composed of *system classes*, and *classes* loaded from the trusted middleware classpath.

  - **Application class system heap:**

    The application class system heap, or (shareable application system heap) contains classes loaded from the `ibm.jvm.shareable.application.class.path` and all classes loaded from the deployed JAR file. Since this is never garbage collected, this means that applications do not have to be reloaded from the HFS if they are subsequently reused by another CICS transaction that reuses the JVM.

Each JVM heap has its own set of tuning parameters; these parameters can be used to balance storage usage and garbage collection for each heap. For further details, refer to the *CICS System Definition Guide*, SC34-5725.

JVM garbage collection

Optimized garbage collection is performed in such a way that the middleware heap and system heaps are reset and the transient heap is garbage collected. Not all applications are able to exploit serial reuse, so that the JVM cannot be reused. For example, this can happen when the application performs one of the following actions:

- It modifies the state of the JVM in a way that cannot be safely reset, such as:
  - Changing system properties
  - Closing the standard output stream
  - Loading a native library
- It uses multithreading.

Note that the EJB V1.1 specification restricts use of these functions and the following additional functions:

- Utilizing read/write static fields
- AWT functions
- Utilizing the java.io package to access files and directories
- Socket operations
- Accessing information about other classes

For a full list of these restrictions, refer “Programming restrictions” in the *Enterprise JavaBeans Specification, V1.1*, available from `http://www.javasoft.com/products/ejb/.`

If an unsafe operation does occur, the storage used by the JVM is recovered and a new JVM is initialized to provide a safe environment for subsequent applications. The JVM monitors the use of interfaces that prevent safe resetting, and the events that prevent reuse are logged. Reinitializing the whole JVM rather than simply resetting the middleware heap and system heap is slower overall and costs performance. Enterprise beans and CICS Java programs that execute on a single Java thread using interfaces defined by the EJB specification, or by the JCICS classes, are normally able to exploit serial reuse of the JVM.

Figure 2-3 summarizes the heap management in the persistent reusable JVM.

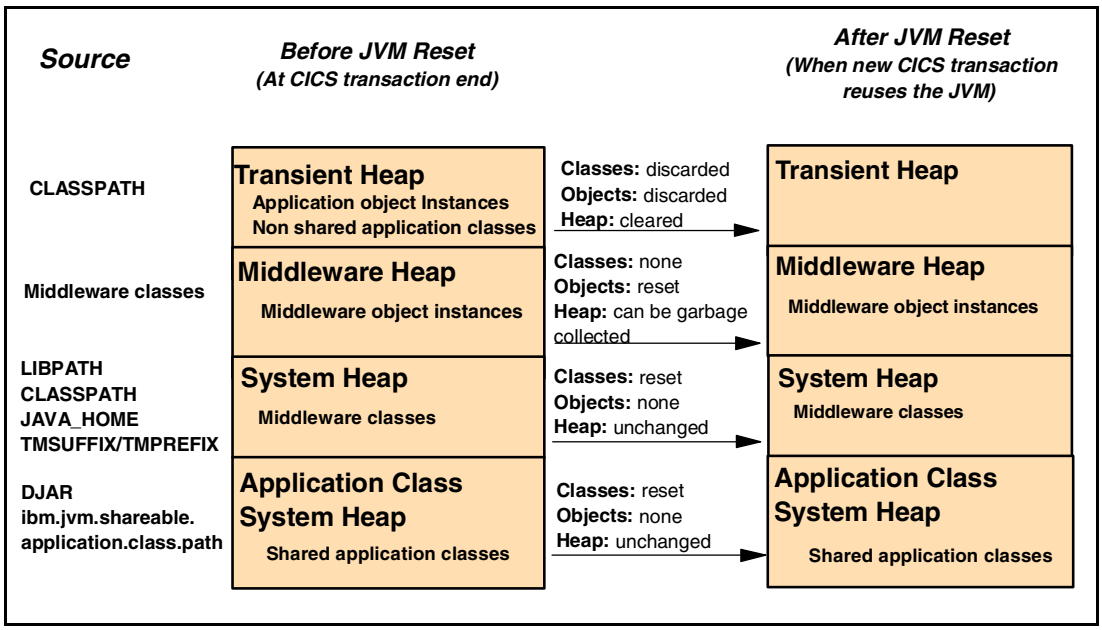


Figure 2-3 Heap management for the persistent reusable JVM

2.2.2 Exploitation of the persistent reusable JVM

CICS exploits the persistent reusable JVM using the following features:

- JVM pools
- JVM selection
- JVM pool management

We shall look at each of these in the following sections.

JVM pools

CICS exploits the persistent reusable JVM by managing a pool of pre-initialized JVMs and selecting an appropriate JVM from the pool when a Java program is invoked. Each JVM is essentially single threaded and runs under its own Open Transaction Environment (OTE) TCB. CICS controls the numbers of TCBs of each type and adjusts the number in response to the work load. You should adjust the MAXOPENTCBS setting according to the amount of storage below 16M that is available in your system.

You should also restrict the number of active transactions in the system to maintain a JVM pool that always has Java Virtual Machines free to satisfy new requests. CICS reduces the number of active JVMs automatically if the work load does not require them. One side-effect of this implementation is, that it enables improved multiprocessor utilization in a single CICS region.

How CICS maintains a pool of JVMs, in which JVMs may be in use or available for reuse is shown in Figure 2-4.

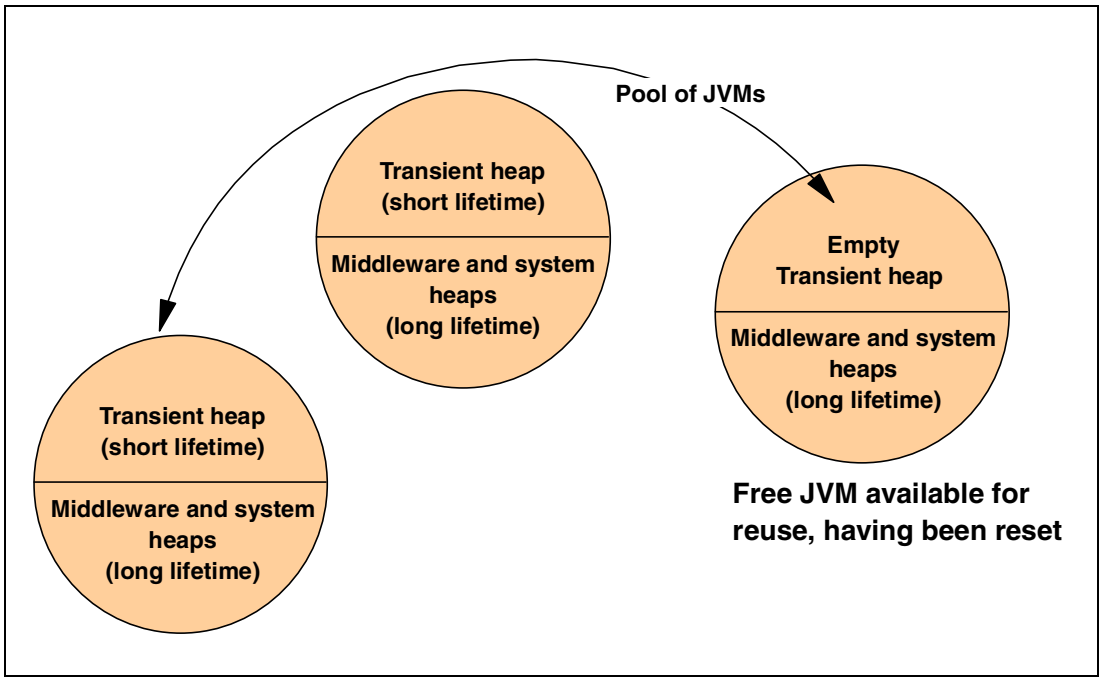


Figure 2-4 The reuse optimization from a JVM pool

In the figure, two of the JVMs shown are in use and contain application objects in the transient heap, which is separated from long-lived objects in system and middleware heaps. The third JVM contains only long-lived objects and is available for reuse.

JVM selection

JVMs are allocated to a CICS transaction that requests execution of a Java program. The JVMs in the pool are not necessarily identical because they may have been initialized with different parameters, such as different heap sizes or debugging capability. These characteristics required by the Java program are defined by naming a JVM profile on the CICS PROGRAM resource definition, which also indicates the static main method that is the entry point of the application program. In the case of enterprise beans, the entry point is the CICS request processor (DFHIIRP).

The initialization parameters are specified in a member in the DFHJVM PDS named by the JVMPROFILE parameter on a PROGRAM definition. All JVMs initialized with the parameters from a particular JVMPROFILE file are identical.

When a program is invoked, CICS selects from the pool a JVM that was initialized using the JVMPROFILE file specified on the program definition. If no such JVM is available, one of the other JVMs will be reinitialized with the parameters specified in the JVMPROFILE file. If there are no free JVMs, the transaction will wait for a JVM to become free.

Selecting a JVM from the pool to match a request is illustrated in Figure 2-5.

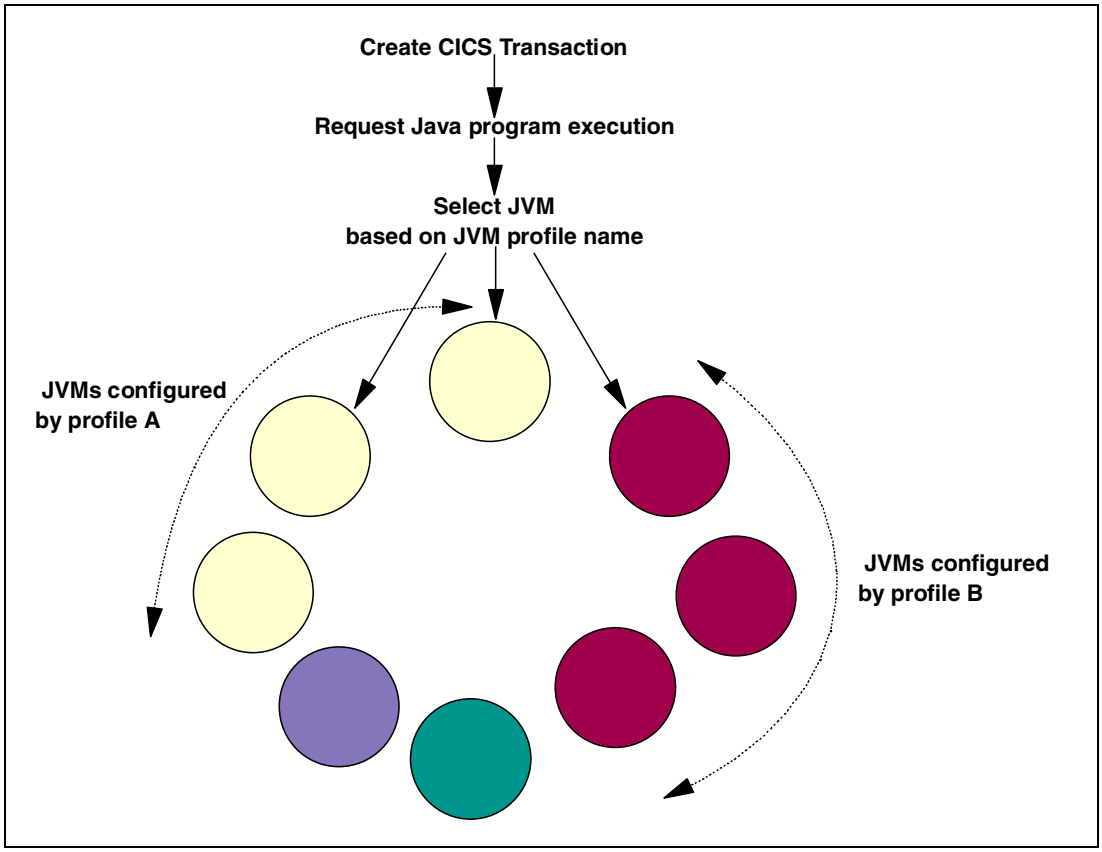


Figure 2-5 Selecting a JVM from the pool to match a request

JVM management

Functions are provided to manage the pool of JVMs. In particular, if it becomes necessary to reinstall the JVMs, because some of the pre-loaded classes have been modified, then the current JVM instances in the pool can be *phased out*. This means that as the transactions the JVMs are running terminate, the JVMs are reinitialized so that they reload the latest version of the classes. This action can be performed using the command **`CEMT SET JVM PHASEOUT.`**

2.3 IIOP support in CICS

The Internet Inter-ORB protocol (IIOP) is a TCP/IP based implementation of the General Inter-ORB Protocol (GIOP) that defines formats and protocols for distributed applications. It is part of the Common Object Request Broker Architecture (CORBA). Both client and server systems require a CORBA Object Request Broker (ORB) to implement IIOP interoperability.

CORBA was defined by a consortium of over 500 information technology organizations called the Object Management Group (OMG). CORBA is OMG's open, vendor-independent specification for an architecture and infrastructure that computer applications use to work together over networks. Interoperability results from two key parts of the specification: OMG Interface Definition Language (OMG IDL), and the standardized protocols GIOP and IIOP. These allow a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, to interoperate with a CORBA-based program. This CORBA-based program can be from the same or another vendor, on almost any other computer, operating system, programming language, and network. You can read the CORBA architecture and specification document at http://www.omg.org/, their Web site. CICS provides an ORB and support for IIOP defined by the CORBA 2.1 specification.

2.3.1 The Object Request Broker

An Object Request Broker or ORB is the facilitator by which objects on the CORBA network can communicate. An ORB enables disparate applications to communicate without being aware of the underlying communication mechanism. In this way, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments, and interconnects multiple object systems.

Object Request Brokers have the following main responsibilities:

- To allow objects to dynamically discover each other
- To allow objects to call methods on each other
- To handle the passing of parameters to objects
- To return results

The client does not need to know where the object is located, its programming language, its operating system, or any other system aspects that are not part of the object's interface. The broker chooses the best server to meet the client's request and separates the interface that the client sees from the implementation of the server.

The CICS ORB

The CICS ORB implements the following level of function:

- Support for the CORBA 2.1 API, except for Dynamic Invocation Interface (DII)
- Dynamic Skeleton Interface (DSI), and GIOP fragments
- Support for IIOP 1.1
- Support for both inbound and outbound IIOP requests; IIOP applications can act as both client and server
- Support for transactional objects

  CICS TS V2.1 method invocations may participate in Object Transaction Service (OTS) distributed transactions. If a client calls an IIOP application in the scope of an OTS transaction, information about the transaction flows as an extra parameter on the IIOP call. If the client ORB sends an OTS Transaction Service Context and the target stateless CORBA object implements CosTransactions::TransactionalObject, then the object is treated as transactional.

2.4 The CICS EJB Server architecture

This section describes the architecture of the CICS EJB server.

2.4.1 Components of the CICS EJB Server

A CICS EJB Server is made up of following components:

- TCP/IP listener
- Request receiver
- Request models
- Request stream
- Request stream directory
- Request processor
- EJB container
- Object store
- CorbaServers
- Java Virtual Machine
- Deployed JAR files

The flow of data through the components is illustrated in Figure 2-6, and the relationship between the different resources is shown in Figure 2-7.

Each of the components is discussed further in the following sections.

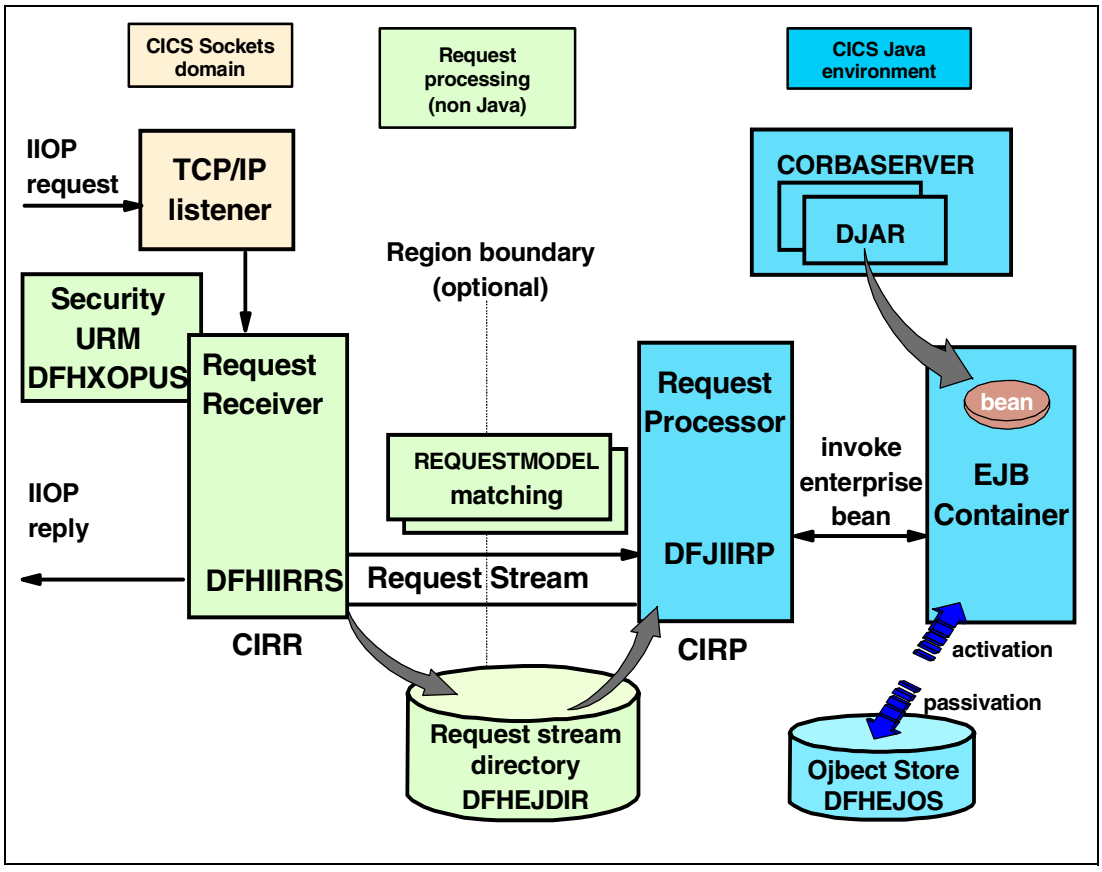


Figure 2-6 Data flow through the CICS EJB server

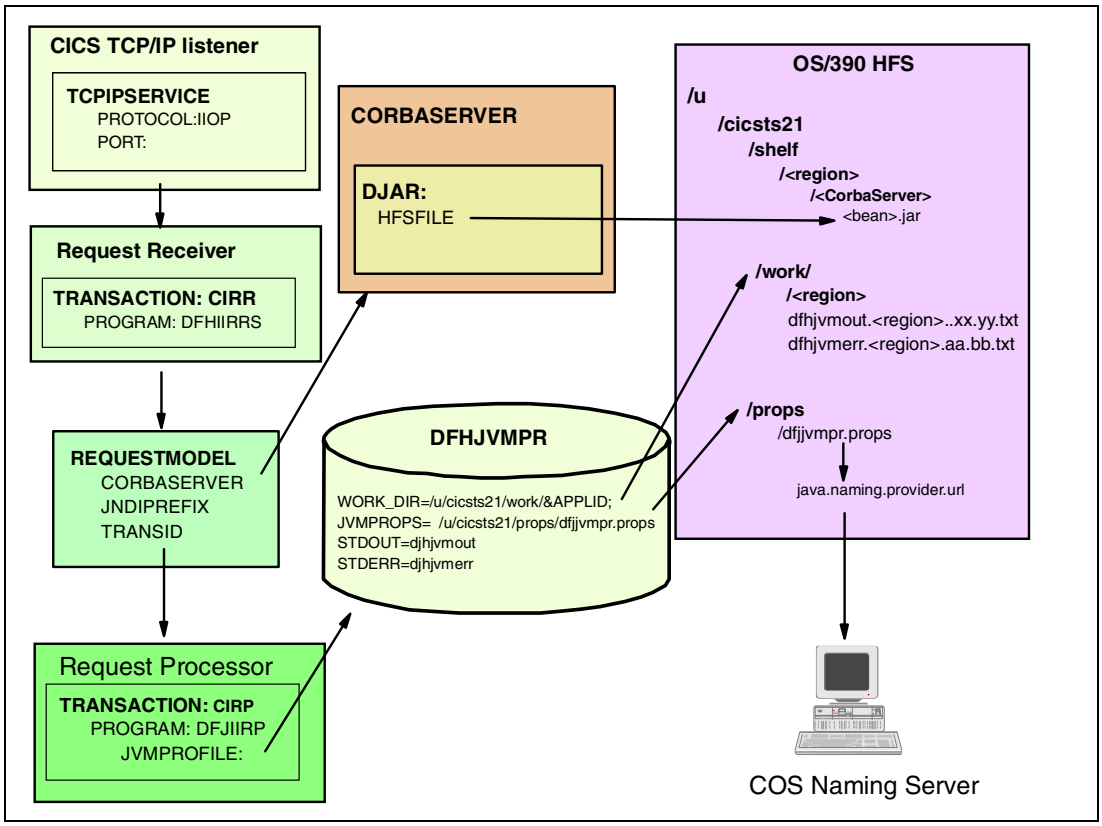


Figure 2-7 The CICS EJB server — how resources fit together

TCP/IP listener

The CICS TCP/IP listener monitors specified ports for inbound requests. You specify IIOP ports and configure the listener by defining and installing TCPIPSERVICE resources.

For a TCPIPSERVICE, you define:

- The port number to listen on.
- The protocol, which must be set to IIOP for enterprise beans.
- The transaction ID of the request receiver. The default transaction ID is CIRR.
- Whether SSL is enabled or not. If SSL is enabled, you can choose between client and server authentication, or server authentication only.
- The name of the user replaceable security module. The default module is DFHXOPUS.
- The name of the DNS connection optimization group for workload management (optional).

The TCP/IP listener receives the incoming request and starts the transaction specified. For IIOP services, this corresponding transaction resource definition must have the program attribute set to DFHIIRRS, the request receiver program.

Request receiver

The request receiver transaction (CIRR) is attached by the IIOP domain and invokes the program DFHIIRRS. The request receiver retrieves the incoming message and examines the contents of the IIOP formatted message stream. If a message is processed, a response is sent to the client. The request receiver transaction stays running while the IIOP connection is open, and terminates when it has no further work to do.

In order to process the request, the request receiver must set a CICS user ID to be used as the security context. This user ID can be determined in the following two ways:

- Through the use of Secure Sockets Layer (SSL) client certificates
- By calling a CICS User Replaceable Module (URM) specified in the TCPIPSERVICE resource definition, for which CICS supplies the sample DFHXOPUS

A request is passed to a request processor using an associated request stream. A request stream is a new task-to-task communication mechanism in CICS and is explained in more detail in “Request stream” on page 43.

If it is a new request for the server, a request processor is determined and passed the associated request stream. If it is a follow-on request, the existing request processor is determined and passed the existing request stream. Once the request is forwarded to a request processor, the request receiver waits for the return of data on the request stream.

Request models

To associate the incoming IIOP formatted request with a CICS transaction, you need to provide and install REQUESTMODEL resource definitions for all the possible requests that CICS can process. CICS compares fields in the method request against values defined in the REQUESTMODELs, to find the best match.

A request model is required for:

- Each method in the bean’s remote interface, including the methods inherited from the EJBObject interface
- Each method in the bean’s home interface, including methods inherited from the EJBHome interface

A request model specifies:

- The CorbaServer for matching DJAR files.
- Bean names for matching enterprise beans. Bean names can also be generic.
- Operation patterns to match against a bean method name. These patterns can also be generic.
- Interface type. Valid types are Home, Remote, or Both.
- The CICS transaction to be started when a matching request is received. The default is CIRP, which specifies the default request processor, DFJIIRP. If you choose to use your own transaction definition, you should base it on CIRP.

Note: If you do not define a REQUESTMODEL, CICS uses the default request model, which always maps to transaction ID CIRP.

Some IIOP and EJB requests are processed using an existing request processor transaction. This is the case when a request is processed which belongs to an open OTS transaction, and one of the previous requests of this open OTS transaction has also been processed by this EJB server. Such requests run in existing JVM environments, even if a REQUESTMODEL matches the requests. The transaction ID in any matching REQUESTMODEL is used only when a new request processor transaction is required.

For further details on request model selection refer to Section 2.4.2, “Selecting a new request processor” on page 47.

Request stream

Requests are passed from the request receiver to the request processor using an associated request stream. Request streams are a new task-to-task communication mechanism in CICS. They are used in the distributed routing of method requests for enterprise beans. The request receiver and request processor can either be:

- In the same CICS region
- On different CICS regions in the same OS/390 system
- On different CICS regions in different OS/390 systems

To achieve scalability, the different transport protocols used for these topologies are dynamically selected as shown in Table 2-1.

Table 2-1 Transport protocols for request streams

Topology	Transport type
Same CICS region	Within region
Different CICS regions, same OS/390 system	MRO/XM
Different CICS regions, different OS/390 system	MRO/XCF

Request stream directory (DFHEJDIR)

Request streams are stored in the request stream directory, DFHEJDIR. This is a recoverable VSAM KSDS file and must be shared among all application-owning regions (AORs) in the logical EJB Server. It can be shared using one of the following options:

- VSAM RLS
- Function shipping
- Coupling facility data tables

It contains a table of OTS TIDs and object keys, mapped to public IDs (Figure 2-8). The OTS TID and the object key in the request determine if the request must be sent to an existing processor. If the request is not found in the request stream directory, then the request must be the first request for the EJB server concerning a specific OTS transaction and object key. In this case, a new request stream is created and passed to a request processor.

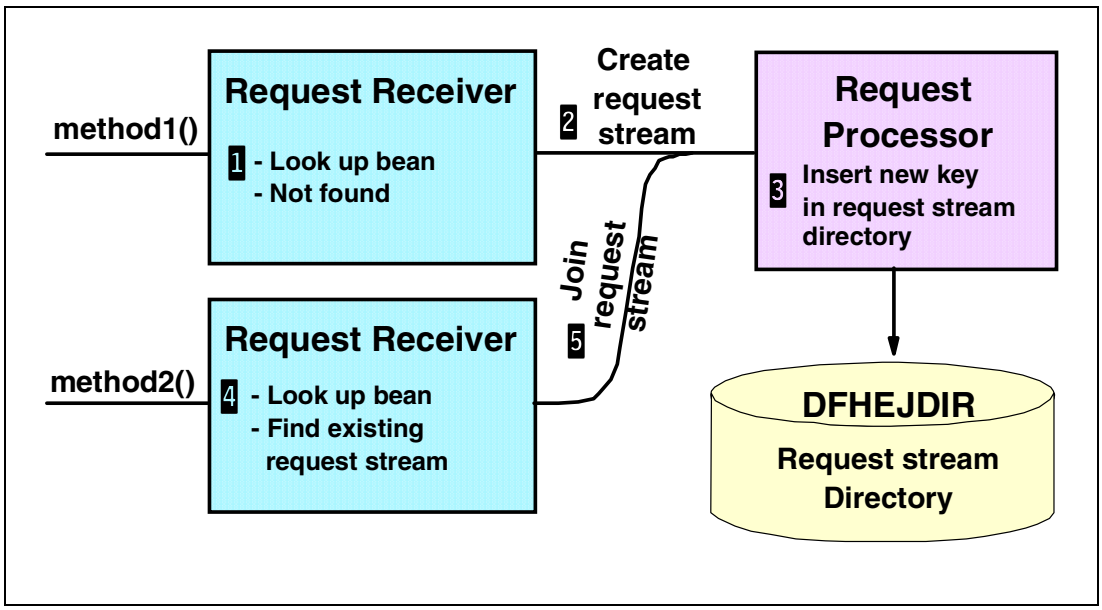


Figure 2-8 Lookup of request stream

If the request stream is found in the request stream directory, then a previous request has been processed previously by this EJB server concerning the specific OTS transaction and object key. The existing request stream key is taken and passed to the specified existing request processor. At the end of the OTS transaction, the entry is removed from the request stream directory.

Request processor

The request processor manages the execution of the IIOP request and is responsible for:

- Locating the object identified by the request
- Calling the container to process the bean method for an enterprise bean request
- Processing the request itself for a request for a stateless CORBA object (although the transaction service may also be involved)

The request processor instance that handles each IIOP request is configured by a CORBASERVER resource definition in CICS. Its default transaction ID is CIRP and has the default program DFJIIRP associated. DFJIIRP must specify the JVMClass com.ibm.cics.iiop.RequestProcessor.

When the request processor receives a request stream, it locates the object identified by the object key of the request. If it identifies an enterprise bean, it calls the container to process the bean method. If it is a stateless CORBA object, it invokes the method directly.

The request processor waits for the return of the method call and passes the result back to the request receiver. Because all method invocations of one specific enterprise bean within one OTS transaction must be processed by the same request processor, the request processor's lifetime is the same as the OTS transaction lifetime. For a bean having the transaction attribute RequiresNew set, a new request processor task is used.

Note: To utilize a user-defined request processor, you will need to specify the new request processor transaction ID in a REQUESTMODEL definition. For further details on how a request processor is selected, refer to 2.4.2, "Selecting a new request processor" on page 47.

EJB container

Whereas desktop JavaBeans usually run within a visual container such as an applet or JVM application, an enterprise bean runs within a container provided by the enterprise Java server. The EJB container creates and manages enterprise bean instances at run-time, and isolates the enterprise beans from direct client access.

The EJB container supports a number of implicit services, including lifecycle, state management, security, transaction management, and persistence. These services are required by each enterprise bean running in the container.

Lifecycle: Individual enterprise beans do not need to manage process allocation, thread management, object activation, or object passivation explicitly. The EJB container automatically manages the object lifecycle on behalf of the enterprise bean.

State management: Individual enterprise beans do not need to save or restore object state between method calls explicitly. The EJB container automatically manages object state on behalf of the enterprise bean.

Security: Individual enterprise beans do not need to authenticate users or check authorization levels explicitly. The EJB container can automatically perform all security checking on behalf of the enterprise bean.

Transaction management: Individual enterprise beans do not need to specify transaction demarcation code to participate in distributed transactions. The EJB container can automatically manage the start, enrollment, commitment, and rollback of transactions on behalf of the enterprise bean.

Persistence: Individual enterprise beans do not need to retrieve or store persistent data from a database explicitly. The EJB container can automatically manage persistent data on behalf of the enterprise bean.

Restriction: CICS TS V2.1 does not support entity beans and so does not support the persistence of enterprise beans.

Object store (DFHEJOS)

The object store is a non-recoverable shared file used by CICS to store stateful session beans that have been passivated. In CICS enterprise beans are passivated at the earliest opportunity, which will occur for all enterprise beans which are not within an OTS transaction and have finished executing method requests.

The object store can be a VSAM file or a coupling facility data table, but must be shared between all regions (both listener regions and AORs) in a CICS EJB Server. Each object store is logically divided up into many stores. Each store is associated with a named CorbaServer. The contents of the object store are not deleted upon request. Instead, the entries are marked as deletable, but not actually removed until an internal garbage collection operation is performed. The garbage collection operation is initiated by CICS and is not tunable. However, garbage collection is controlled by an adaptive mechanism that ensures optimum performance.

CICS supplies sample JCL to help you create this file, in the DFHDEFDS member of the SDFHINST library.

CorbaServer

Before enterprise beans can be deployed into an EJB server, their execution environment must be configured. In CICS, this is achieved by installing a CORBASERVER resource definition. A CORBASERVER defines an execution environment for enterprise beans and CORBA stateless objects. For convenience, we shall refer to the execution environment defined by a CORBASERVER definition as a CorbaServer.

The CORBASERVER resource defines the following attributes:

- The name of the CorbaServer
- The JNDI prefix
- The session bean timeout
- The shelf directory, this is a HFS directory where deployed JAR files can be stored
- Server ORB attributes, which are published to the COS Naming Server and are used by a client application to connect to CICS TCP/IP listener, therefore, it is important, that these settings match with the settings of the TCPIPSERVICE controlling the TCP/IP listener:
  - IPaddress
  - PortNumber
  - SSL (flag for SSL client or server authentication)
- The SSL certificate to use for outbound messages.

You can think of a CorbaServer as a means to logically group enterprise beans in a CICS EJB server. The following rules apply to CorbaServers:

- A CICS EJB server may contain more than one CorbaServer.
- A CorbaServer may contain more than one enterprise bean.
- An enterprise bean cannot be deployed multiple times into the same CorbaServer.
- An enterprise bean can be deployed multiple times to different CorbaServers per region.

There are several reasons why you might want to have more than one CorbaServer in an EJB server, which are as follows:

- To deploy the same enterprise bean with different characteristics to your CICS EJB server. These different characteristics could be:
  - Different deployment descriptor attributes, such as transaction attributes
  - Different timeout settings
  - Different SSL settings
  - Different JNDI prefixes; allows two CorbaServers containing the same enterprise bean
- To implement application separation. You might want to separate enterprise beans in different CorbaServers based on application criteria.

Java Virtual Machine

A new persistent-reusable JVM has been introduced for OS/390, designed to for high-volume execution of Java. To read more about the JVM, refer to 2.2, “The Java Virtual Machine” on page 34.

Deployed JAR files

A DJAR resource defines the file location of a CICS deployed JAR file. A CICS-deployed JAR file is an EJB-JAR file, containing enterprise beans, on which code generation has been performed and which has been stored in the HFS used by the CICS region.

The DJAR resource defines the following attributes:

- The name of the DJAR resource.
- The CORBASERVER into which to install the DJAR.
- The location of the DJAR file in the HFS.

When the DJAR resource definitions are installed into a CICS region, this will cause CICS to:

- Copy the deployed JAR file (and the classes it contains) to the CICS shelf directory in the HFS.
- Read the deployed JAR from the shelf, parse its XML deployment descriptor, and store the information it contains.

2.4.2 Selecting a new request processor

Request models are used to set the properties of the execution environment for enterprise beans. In CICS terms, they set the CICS request processor transaction that will be used to execute the function within a JVM which has the required properties. Their function is illustrated in Figure 2-9.

The actual transaction ID selection part of a request model is not very important for the function of the enterprise bean. Whatever transaction is used, the bean will be executed. The real benefit that it provides is the ability to use standard CICS tuning, measurement, and resource control on a JVM execution. This transaction ID selection can also be used to route request to CICS regions, using the DFHDSRP user-replaceable module.

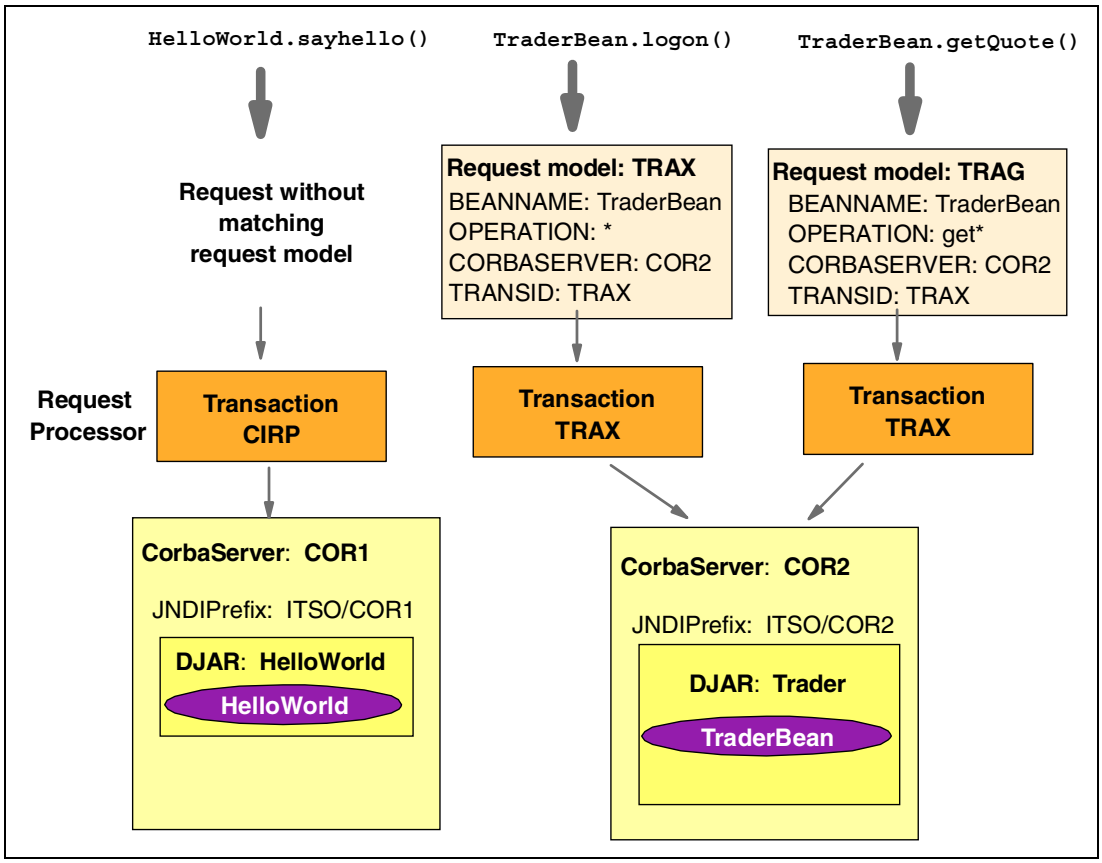


Figure 2-9 Relation between CorbaServers, DJARs, and request models

The transaction ID of the request processor that will be used to process an incoming method request is determined by matching the method request with the request model definitions installed in that region. The bean name, the interfacetype (home or remote), the CorbaServer name, and the operation (method) are all matched against the information in the request stream. The rules for request model matching are as follows:

- A more specific match overrides a generic match.
- The order of precedence is:

  a. CorbaServer
  b. Bean name
  c. Interface type
  d. Operation

Tip: Always end the method name in the REQUESTMODEL operation field with a “*”. This is because this setting includes name-mangling so that a given method with parameter over-loading can be selective.

If you want your bean to be executed with another JVM profile, you have two options:

1. You can define a new request model and specify a different CorbaServer, bean name or transaction ID. This is useful for two reasons:
   – You can allow different method requests to invoke different CICS transactions, which is very useful for CICS monitoring purposes. If so you would also need to define the new request processor transaction and associated request processor program.
   – You can cause different method requests to run in different Java environments, since the request processor specifies the name of the JVM profile in the program definition.
2. You can specify another JVM profile in the CICS program definition for DFJIIRP. This is not the preferred method, because all other beans which match to the default request model will also be executed with this new profile.

In Figure 2-9 on page 47 the two request models TRAX and TRAG are used to separate the different method requests on the TraderBean into two request processor transactions, TRAX, and TRAG. This is achieved by specifying *get** in the Operation field of the request model TRAG, so that methods `getQuote()`, `getUserID()`, and `getCompany()` will run under a separate request processor transaction, which could use a different JVM environment. However, the actual TraderBean is only defined in the one DJAR Trader, which is defined in the COR2 CorbaServer, so both request processor transactions will invoke methods in the same DJAR.

Other requests such as the `sayhello()` method invoked on the HelloWorld bean, will not match request models TRAX or TRAG and so will use the default request processor, CIRP. Executing the bean with another JVM profile.

Tip: A good way of achieving a distinct set of properties for a set of CorbaServers is to associate a unique request model with each CorbaServer and then use distinct DFHJVM members and HFS files to set the JVM properties. Doing this creates a simple one-to-one mapping and permits easy changes to the environment.

For further details on how we defined a new REQUESTMODEL for our Trader bean, refer to Figure 4-8, “REQUESTMODEL resource definition” on page 80.

2.4.3 Object Transaction Service

An Object Transaction Service (OTS) transaction is a distributed unit of work. Method invocations may participate in OTS transactions or bean-managed OTS transactions. The setting of a method's transaction attribute determines whether or not the CICS task under which the method executes makes its own unit of work, or is part of a wider, distributed OTS transaction.

Transaction attributes are specified in the enterprise bean's deployment descriptor. Table 2-2 illustrates how CICS behaves for a specific attribute.

Table 2-2 Transaction attributes

Attribute	If client has OTS transaction	If client has no OTS transaction
Mandatory Bean must execute within context of client's OTS transaction	CICS registers interest in the transaction with the coordinator.	CICS throws TransactionRequired exception.
Never Bean must not been invoked in context of an OTS transaction.	CICS throws RemoteException exception.	CICS starts a unit of work, and commits at end of method. Equivalent to SYNCONRETURN behavior.
NotSupported Bean cannot execute within context of an OTS transaction.	OTS transaction is suspended for duration of the method call. CICS starts a unit of work and commits at end of method. OTS transaction is resumed when method is completed. Equivalent to SYNCONRETURN behavior.	CICS starts a unit of work and commits at end of method. Equivalent to SYNCONRETURN behavior.
Supports Bean can execute with or without an OTS transaction context.	CICS registers interest in the transaction with the coordinator.	CICS starts a unit of work and commits at end of method. Equivalent to SYNCONRETURN behavior.
Required Bean must execute within context of an OTS transaction.	CICS registers interest in the transaction with the coordinator.	Container starts a new OTS transaction; CICS functions as the OTS coordinator and commits or rolls back the OTS transaction when the bean method ends.
RequiresNew Bean must execute with context of a new OTS transaction.	OTS transaction is suspended for duration of the method call. Container starts a new OTS transaction. CICS functions as the OTS coordinator and commits or rolls back the OTS transaction when the bean method ends. OTS transaction is resumed when method is completed.	Container starts a new OTS transaction; CICS functions as the OTS coordinator and commits or rolls back the OTS transaction when the bean method ends.

2.4.4 Workload balancing

You can implement a CICS EJB server in a single CICS region. However, in a sysplex environment you can create a CICS EJB server consisting of multiple CICS regions, which behave like one logical server. Using multiple regions makes failure of a single region less critical and enables you to balance work across the multiple systems in the sysplex, thus providing for greater throughput than a single system could offer.

Typically, a CICS logical server consists of:

- A set of cloned listener regions defined by identical TCPIPSERVICE definitions to listen for incoming IIOP requests. These listener regions do not need to be Java-enabled since only the request receiver (CIRR) runs within the listener region and routes the request stream to the request processor in the AORs. Therefore such listener regions do not need configuring with a JVMProfile or JVM TCBs.
- A set of cloned application-owning regions (AORs), each of which supports an identical set of IIOP applications or enterprise bean classes in an identically-defined CorbaServer. Multiple methods for the same OTS transaction are directed to the same AOR.

Workload balancing of IIOP requests

To workload balance IIOP requests in CICS there are two options. First you can balance incoming IIOP connections across listener regions. Second you can use CICSPlex SM or use your own CICS distributed routing program to balance OTS transactions across a set of cloned AORs.

Balancing client connections across listener regions

To balance clients across listener regions you have the following four options:

1. **Sysplex Distributor**

   This technique offers effective distribution of workload throughout a Parallel Sysplex without requiring any non-OS/390 technology. Sysplex Distributor is provided in OS/390 Communications Server V2R10 and combines XCF and VIPA functions to provide new levels of availability and workload balancing in a Parallel Sysplex. A new sysplex-wide VIPA address enables workload distribution to multiple server instances without requiring changes to clients or networking hardware, and without delays in connection setup. Sysplex Distributor resides in the Parallel Sysplex and has the ability to factor “real-time” information; including server status, quality of server (QoS), and Policy information (provided by the Service Policy Agent). Combining real-time factors with the information obtained from MVS WLM, uniquely ensures that the best destination server instance is chosen for a particular client and that Service Level Agreements are maintained.

2. **DNS connection optimization**

   Connection optimization is a technique that uses the OS/390 dynamic DNS server in conjunction with MVS WLM to balance IP connections in a sysplex domain. With DNS connection optimization, multiple CICS systems in different OS/390 systems can be started to listen for IIOP requests using the same generic hostname and port. Each region is automatically registered with MVS WLM which provides information to the OS/390 dynamic DNS server, which resolves client IIOP request containing the generic host name and port number to the IP address of the least loaded CICS system. This function is implemented in OS/390 V2R5.

3. **TCP/IP port sharing**

   TCP/IP port sharing is a feature of OS/390 Communications Server V2R5, that allows multiple regions in the same OS/390 system to listen for incoming IP requests on the same IP socket. It can be used to easily and efficiently balance requests across multiple listener regions in the same LPAR. The balancing of IP requests across regions is based on the number of socket connections both active and in the backlog.

4. **Network Dispatcher**

   IBM Network Dispatcher is the load balancing component of IBM WebSphere Edge Server and can run on a variety of platforms including Windows NT, AIX, and Solaris. For OS/390, a version of Network Dispatcher can also be deployed within the 2216 router.

   Network Dispatcher provides one network visible cluster address to which all IP requests are routed. The dispatcher then intercepts packets sent to this cluster address and routes them to a destination host based on the current load information.

You can read more about workload balancing TCP/IP requests in *Workload Management for Web Access to CICS*, SG24-6118.

Distributed routing

Distributed routing is used to balance method call invocations from listener regions across CICS application owning regions (AORs). The dynamic selection of the target can be made automatically using CICSPlex SM Workload Management, or using a customized CICS dynamic routing program. CICS invokes the distributed routing program, DFHDSRP, for method requests that will run under a new OTS transaction, or outside the scope of an OTS transaction. However, requests that will run under an existing OTS transaction are not dynamically routed; these are directed automatically to the AOR in which the existing OTS transaction is already running.

For further information on writing a customized distributed routing program, refer to the *CICS Customization Guide*, SC34-5706. For further information on CICSPlex SM Workload Management refer to the manual *CICSPlex System Manager Managing Workloads*, SC34-5735.

The diagram illustrated in Figure 2-10 shows a CICS logical EJB server. Sysplex distributor or DNS connection optimization could be used to balance client connections across the listener regions, and distributed routing is used to balance request streams across the AORs.

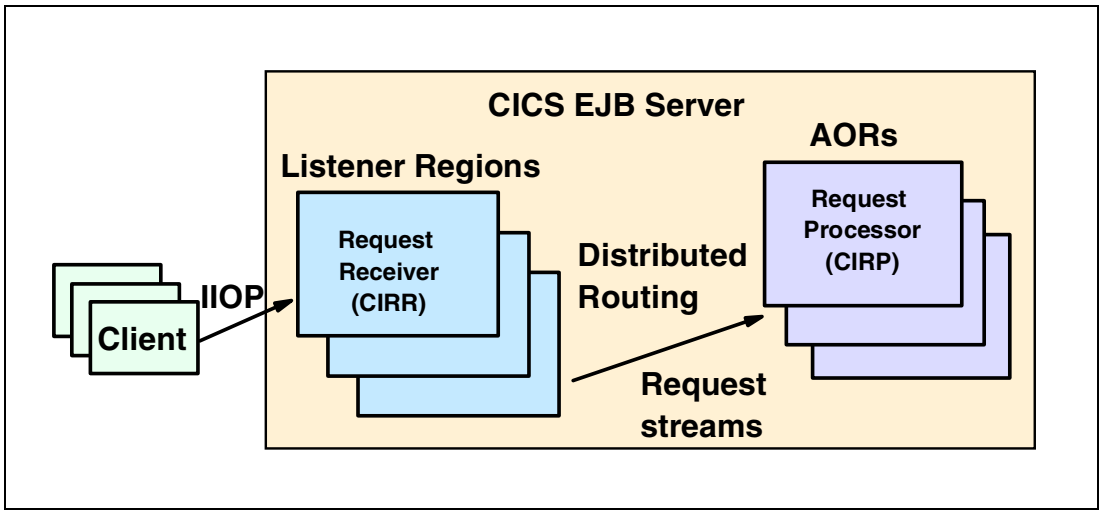


Figure 2-10 Workload balancing in a sysplex

3

Accessing CICS from servlets and enterprise beans

In this chapter we provide a summary of the different configurations in which servlets and enterprise beans can be used to access CICS applications on OS/390. For each scenario we give a brief outline of the configuration and provide a discussion of the pertinent issues when using this configuration.

In this chapter we shall consider the following IBM products, all of which provide an EJB container for the running of enterprise beans:

- IBM WebSphere Application Server, V3.5 Advanced Edition (available for Windows NT, AIX, and Solaris)
- IBM WebSphere Application Server for z/OS and OS/390, V4.0
- CICS Transaction Server for z/OS, V2.1

In addition, we consider the following IBM Web application servers which provide a servlet engine for the execution of servlets and JSPs:

- IBM WebSphere Application Server, V3.5 Standard Edition (available for Windows NT, AIX, and Solaris)
- IBM WebSphere Application Server Standard Edition for OS/390 V3.02
- IBM WebSphere Application Server Standard Edition for OS/390 V3.5

3.1 From a servlet — Using the CICS connectors

All versions of IBM's WebSphere Application Server include an environment for the execution of servlets, the servlet engine. A CICS application program may be executed from a servlet using the CICS connector facilities. A typical use of a servlet is to access information held on OS/390 by invoking a CICS application program and then to pass the data returned to a Java Server Page (JSP). The JSP constructs the contents of the dynamic Web page to be returned to the browser.

The CICS Transaction Gateway (CTG) Java classes *ECIRequest* and *EPIRequest* may be used to invoke the CICS application program programmatically, or the connection and invocation may be defined to the CTG using the Common Connector Framework (CCF).

The CICS application may be a COMMAREA based program, in which case the CICS External Call Interface (ECI) is used; or it may be a 3270 data stream based program, in which case the External Presentation Interface (EPI) is used. In order to use the EPI, the CICS Transaction Gateway (CTG) must to be run on a distributed platform. There is a also third interface to CICS, the External Security Interface (ESI). This allows the client program to request security information from CICS, such as verifying or modifying passwords.

The Common Connector Framework (CCF) is an IBM specification supported by the CTG which provides a Java based infrastructure for providing access to existing Enterprise Information Systems such as CICS. The Enterprise Access Builder of Visual age for Java assists the developer in using the CCF.

WebSphere (distributed platform)

The versions of WebSphere that run on distributed platforms are WebSphere Application Server Standard Edition and WebSphere Application Server Advanced Edition. With both of these versions of WebSphere the CTG V3.1 may be used to connect to a CICS application running on OS/390.

Figure 3-1 illustrates a servlet on a distributed platform making an ECI call to invoke a CICS COMMAREA based application program using either a CTG running on OS/390 or on the distributed platform.

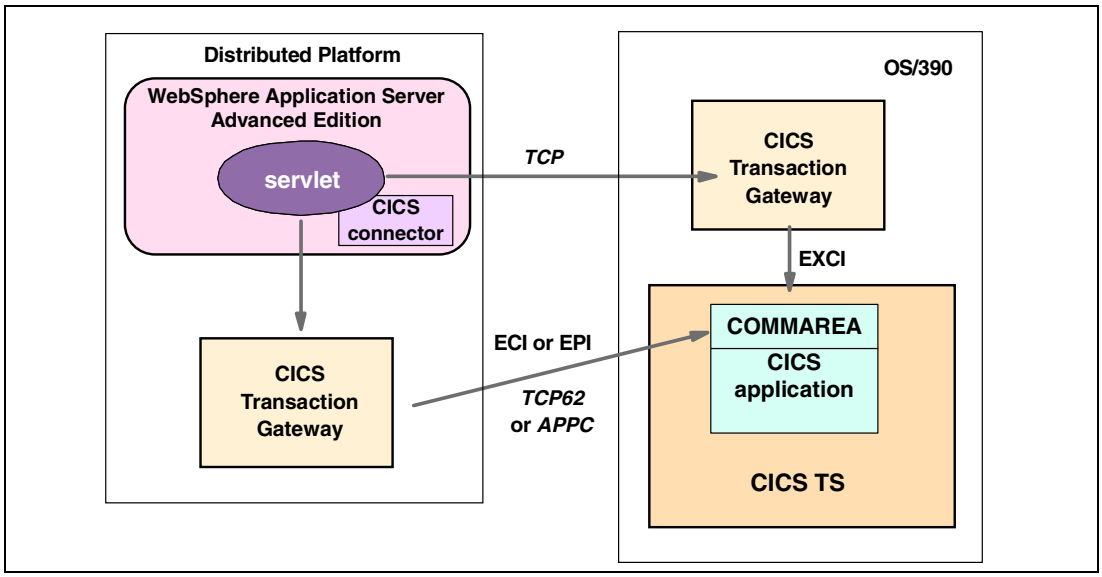


Figure 3-1 Servlet, distributed platform — using CICS connector

There is no transactional co-ordination between the servlet and the execution of the application program in CICS. A CICS logical unit of work is created at the start of execution of the application program and this logical unit of work is committed at successful completion or rolled back (aborted) if the program fails. In the event of a failure, status information is returned to the servlet, which may take appropriate action. However, there is no automatic system action.

In summary, using the CICS connector from WebSphere Application Server on a distributed platform provides the following useful features:

- Ability to use the ECI, EPI, and ESI.
- Ability to communicate with CICS/ESA V4.1 and CICS VSE region as well as CICS TS V1 and CICS TS V2 regions on OS/390.
- Ability to develop a servlet on a distributed platform and subsequently move it to WebSphere on OS/390.

The following limitations of using the CICS connector from a servlet should also be considered:

- The CICS logical unit of work is not coordinated with the servlet if it is running in WebSphere V3.02 or V3.5.
- Using the servlet on a distributed platform requires the use of a TCP62 or SNA connection if you wish to use the EPI or ESI interfaces.

WebSphere (OS/390 and z/OS)

There are three versions of WebSphere that are currently are available on OS/390. They are WebSphere Application Server V3.02 and V3.5 Standard Edition, and WebSphere Application Server V4.

Figure 3-2 illustrates a servlet in WebSphere Application Server for OS/390 V3.02 or V3.5 invoking a COMMAREA based CICS application program using the CICS connector as provided by CTG for OS/390. As the protocol is specified as *local:* by the servlet, a local CTG is used, and the CTG Java classes are invoked within the WebSphere Application Server address space.

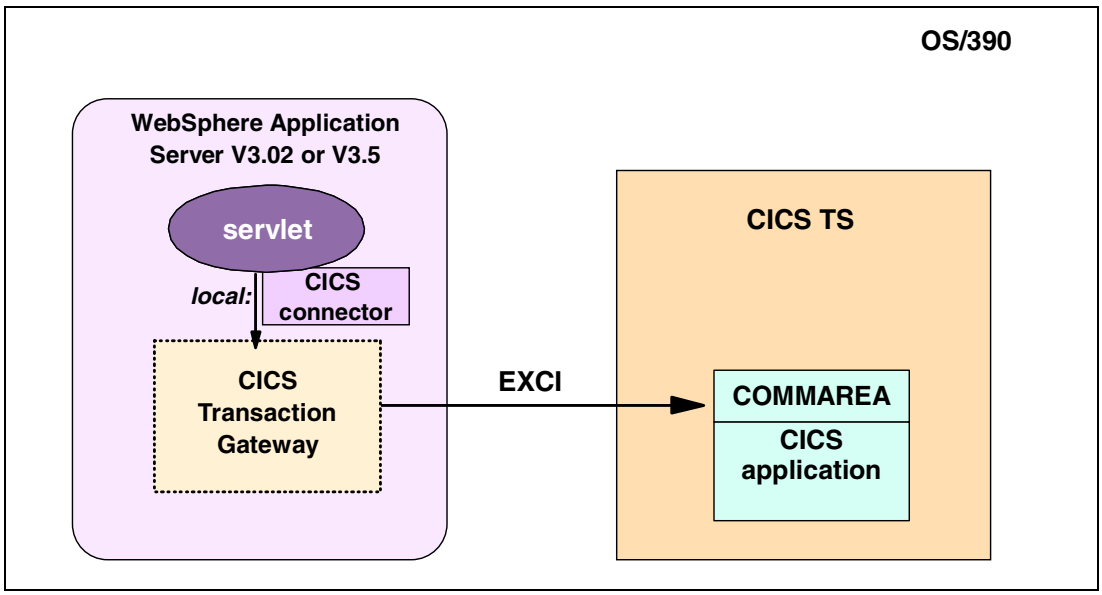


Figure 3-2 Servlet, OS/390 — using CICS connector

A servlet running on either version of WebSphere connects to a CICS application program in the same way as it would on a distributed platform, using the CTG and either the CCF or an *ECIRequest* object.

As on the distributed platform, there is no transactional coordination between the servlet and the program running in CICS. However, if both WebSphere and CICS are running in the same OS/390 LPAR, the CTG V3.1.2 can exploit transactional EXCI with MVS Resource Recovery Service (RRS) as the coordinator and therefore participate in a global unit of work with CICS.

A feature of WebSphere V4.0 is a new connector to CICS which is compliant with the Common Client Interface of the J2EE Connector Architecture specification. This feature is called the *WebSphere/390 CICSEXCI connector*. It does not use the CTG to connect to CICS, but uses its own interface to the External CICS Interface (EXCI) and can provide transactional coordination between WebSphere and CICS if the CICS region and WebSphere are in the same OS/390 image. Note that this feature is provided as a beta in WebSphere V4.0 and will be replaced by the CICS connector in CTG V4.1 when it becomes available for OS/390. CTG V4.1 will be compliant with the J2EE Connector Architecture specification.

J2EE Connector architecture: The J2EE Connector architecture is an industry wide standard for connecting the J2EE to heterogeneous Enterprise Information Systems. The CTG V4 and Visual Age for Java V3.5.3 will support the J2EE Connectors.

Figure 3-3 illustrates a servlet in WebSphere Application Server V4.0 invoking a COMMAREA based CICS application program using the WebSphere/390 CICSEXCI connector.

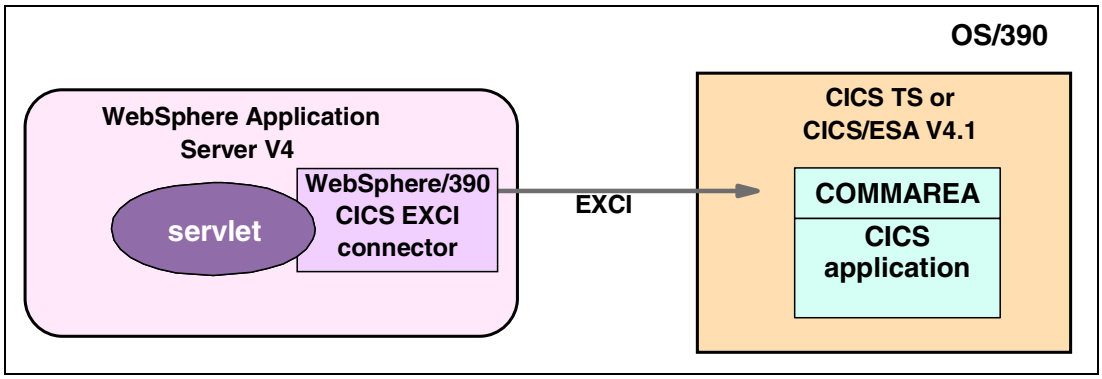


Figure 3-3 Servlet, OS/390 — using WebSphere/390 CICSEXCI connector

In summary, using the CTG or the WebSphere/390 CICSEXCI connector from a servlet running in WebSphere Application Server on OS/390 provides the following useful features:

- Ability to develop to the J2EE connector architecture and then deploy to any Web application server that has J2EE connector runtime support.
- Ability to develop a servlet on a distributed platform and subsequently move it to WebSphere on OS/390.

The following limitations of using the CTG or the WebSphere/390 CICSEXCI CICS connector from a servlet should also be considered:

- The CICS logical unit of work is not coordinated with the servlet if it is running in WebSphere V3.02 or V3.5.
- The EPI and ESI interfaces are not supported, so only COMMAREA based CICS applications can be invoked.

3.2 From a session bean — Using the CICS connectors

IBM's WebSphere Application Server Advanced Edition provides a container for the deployment of enterprise beans. The CICS connector classes provided by CTG can be used within enterprise beans to invoke applications in a connected CICS region. If the CTG runs on a distributed platform, then both EPI and ECI calls can be made to invoke COMMAREA or 3270 based CICS applications. If the CTG runs on OS/390, then only the ECI can be used.

Figure 3-4 illustrates a session bean making an ECI call to invoke a CICS COMMAREA based COBOL application, using a CTG running on OS/390. The CTG on OS/390 makes an EXCI call to the CICS region, which can be either a CICS TS V1 or V2 system.

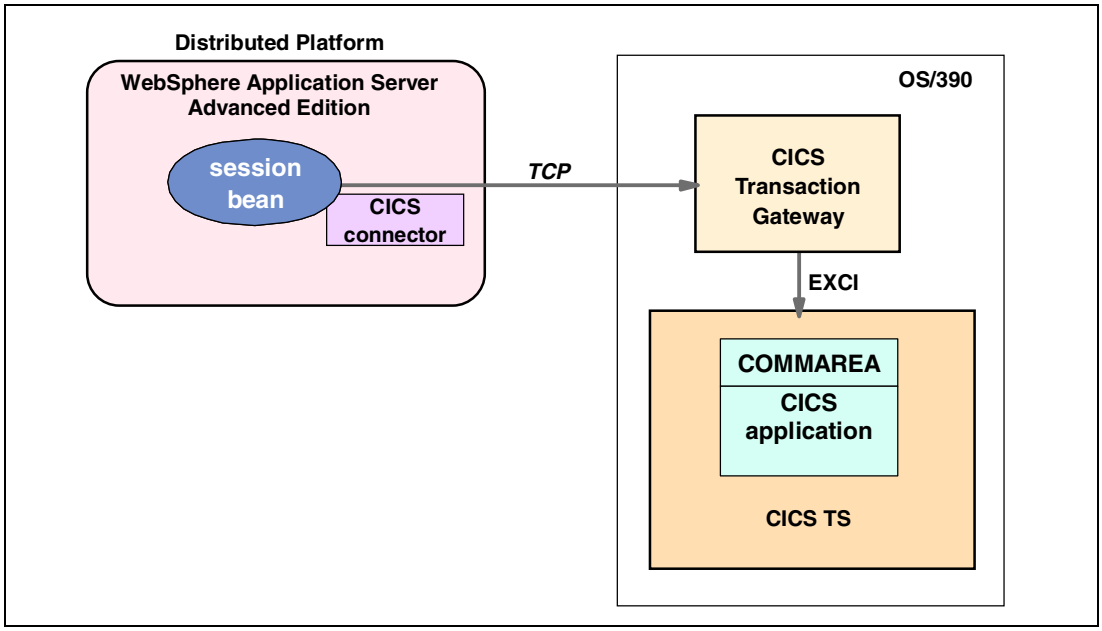


Figure 3-4 Session bean, distributed platform — using CICS connector

Although an enterprise bean runs in the transactional environment provided by the container of the Enterprise Java Server, the work requested in CICS does not participate in the same logical unit of work (or transaction in Java terminology) as the enterprise bean. Both EPI and ECI calls work the same as when used in any Java program. An enterprise bean can be involved in a logical unit of work, but an ECI call (or series of extended ECI calls) to CICS causes a different logical unit of work to start in CICS.

If the called CICS program abends, the logical unit of work in CICS will be terminated, and the abend information will be sent back to the enterprise bean. However, the enterprise bean transaction is separate from the abended CICS logical unit of work, so it will not be automatically influenced by the outcome of the CICS unit of work.

This is also true if the transactional EXCI feature of the OS/390 CTG is used, since this only provides the ability for extending the logical unit of work across multiple extended ECI calls from the CTG to CICS.

Figure 3-5 illustrates a session bean running in WebSphere V4.0 using the WebSphere/390 CICSEXCI connector to invoke a COMMAREA based program in CICS.

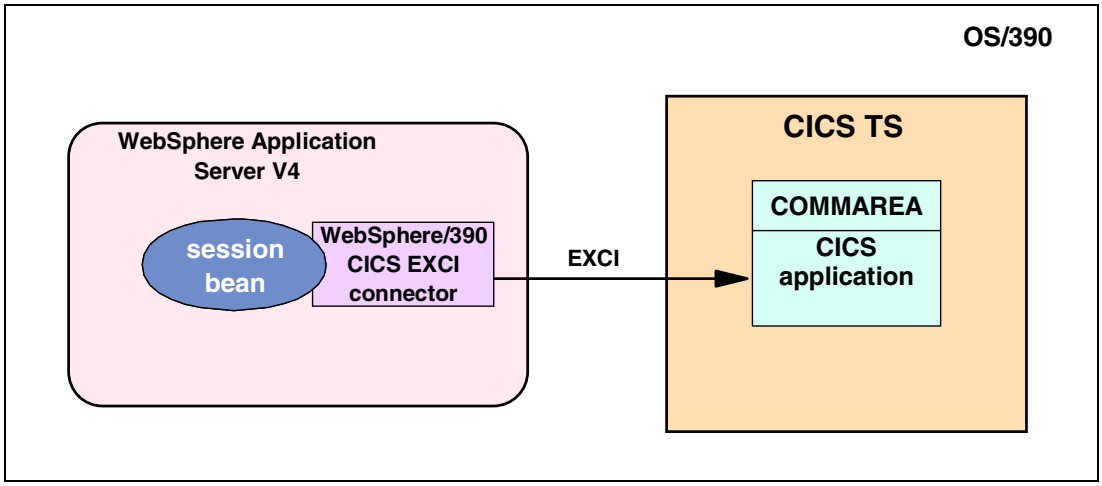


Figure 3-5 Session bean, WebSphere V4.0 — using WebSphere/390 CICSEXCI connector

In summary, using the CICS connector from an enterprise bean provides the following useful features:

- Ability to communicate with CICS/ESA V4.1 (if using a distributed CTG), as well as CICS TS V1 and CICS TS V2 regions.
- Ability to develop the enterprise bean on a distributed platform and then deploy it to WebSphere Application Server on OS/390 or CICS Transaction Server V2.1 if required.

The following limitations of using the CICS connector from an enterprise bean should also be considered:

- The use of the ECI or EXCI limits the maximum data transfer size to 32K as defined in a CICS COMMAREA.
- The CICS logical-unit of work is not coordinated with the enterprise bean transaction, unless the WebSphere/390 CICSEXCI connector is used.
- Using the CTG on a distributed platform requires the use of a TCP62 or SNA connection if you wish to use the EPI or ESI interfaces.

3.3 From a servlet — Invoking a CICS session bean

All versions of IBM's WebSphere Application Server include an environment for the execution of servlets, a servlet engine. A servlet may act as a client to locate, instantiate and execute a session bean in CICS TS V2.1. The session bean could be a self-contained application or it could invoke an existing CICS application. To invoke an existing CICS application program the session bean could use either the CICS connector or the JCICS classes.

The servlet locates the session bean in CICS by using the Java Naming and Directory Interface (JNDI) to obtain a reference to the home interface of the session bean. This reference is then used to create an instance of the session bean and the client is then able to invoke business logic methods on that instance. The Java code used to locate, instantiate and invoke methods on a session bean is not platform specific; the Java code required is the same for both WebSphere on a distributed platform and WebSphere on OS/390.

If the session bean needs to access an existing CICS application program, it can link to it using either the JCICS `link()` method or the new *CICS connector for CICS TS* which provides both a CCF interface or the ECIRequest object. All of these options use a CICS LINK call, rather than the EXCI, to access the back-end server program. Both local links and distributed program link (DPL) calls are supported.

Figure 3-6 illustrates a servlet invoking a session bean in CICS TS V2.1. The protocol used to communicate with the session bean is RMI over IIOP.

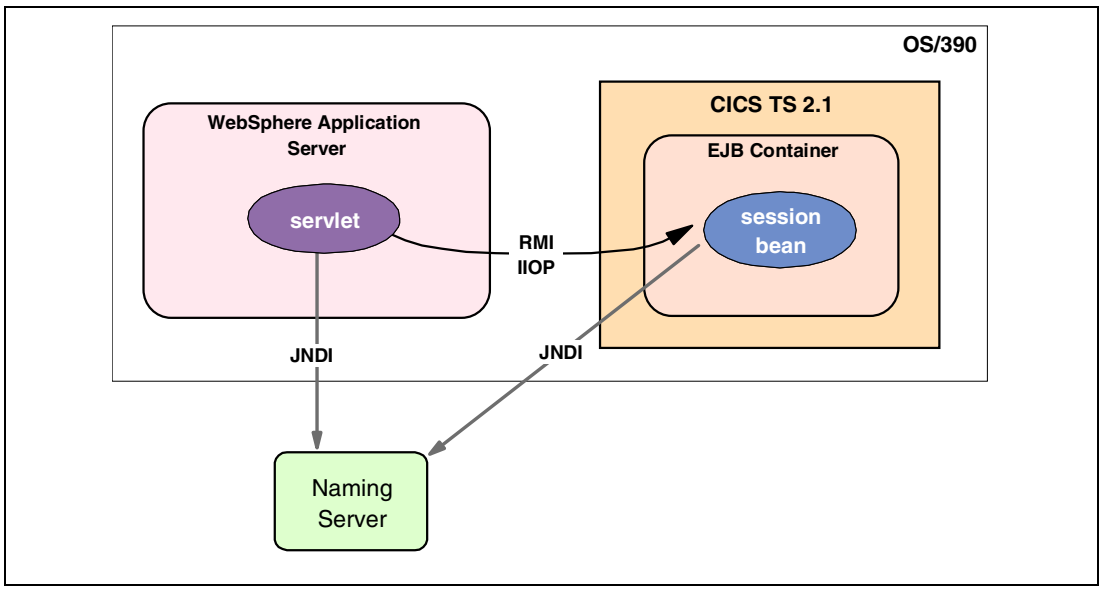


Figure 3-6 Servlet, any platform — invoking a session bean in CICS

In summary, invoking a session bean from a Servlet provides the following useful features:

- This offers the ability to execute a business component by reference to its external name only. There is no need to know where it is in the network.
- The Servlet is portable; it may be developed on one application server and deployed on any server which supports the Java Servlet specification.

The following limitation of invoking a session bean in CICS should also be considered:

- When using servlets, there is no transaction coordination between the Servlet and the session bean running in CICS, as the transactional environment is provided by the EJB container in an Enterprise Java Server.

3.4 From a session bean — Invoking a CICS session bean

WebSphere Application Server Advanced Edition, WebSphere Application Server for OS/390 V4.0 and CICS TS V2.1 all provide containers for the deployment of enterprise beans. A session bean can be deployed in any of these environments and invoke another session bean which may be located in any EJB server.

To invoke a method on another bean, the Java Naming and Directory Interface (JNDI) is called to obtain a reference to the home interface of the required session bean. This reference is then used to create an instance of the session bean. Method invocations are then made on the session bean instance.

Each session bean called may in turn instantiate other session beans and invoke methods on them. If an Object Transaction Services (OTS) transaction is started, then all the enterprise beans called may participate in the transaction. Full transaction coordination among all enterprise beans involved takes place using a two-phase commit protocol.

If the session bean needs to access an existing CICS application program, it can link to it using either the JCICS `link()` method or the new CICS connector for CICS TS which provides both a CCF interface or the ECIRequest object. All of these options use a CICS LINK call, rather than the EXCI, to access the back-end server program. Both local links and distributed program link (DPL) calls are supported

Figure 3-7 illustrates a session bean running in WebSphere Application Server for OS/390 V4.0 invoking a method on a session bean running in CICS TS V2.1.

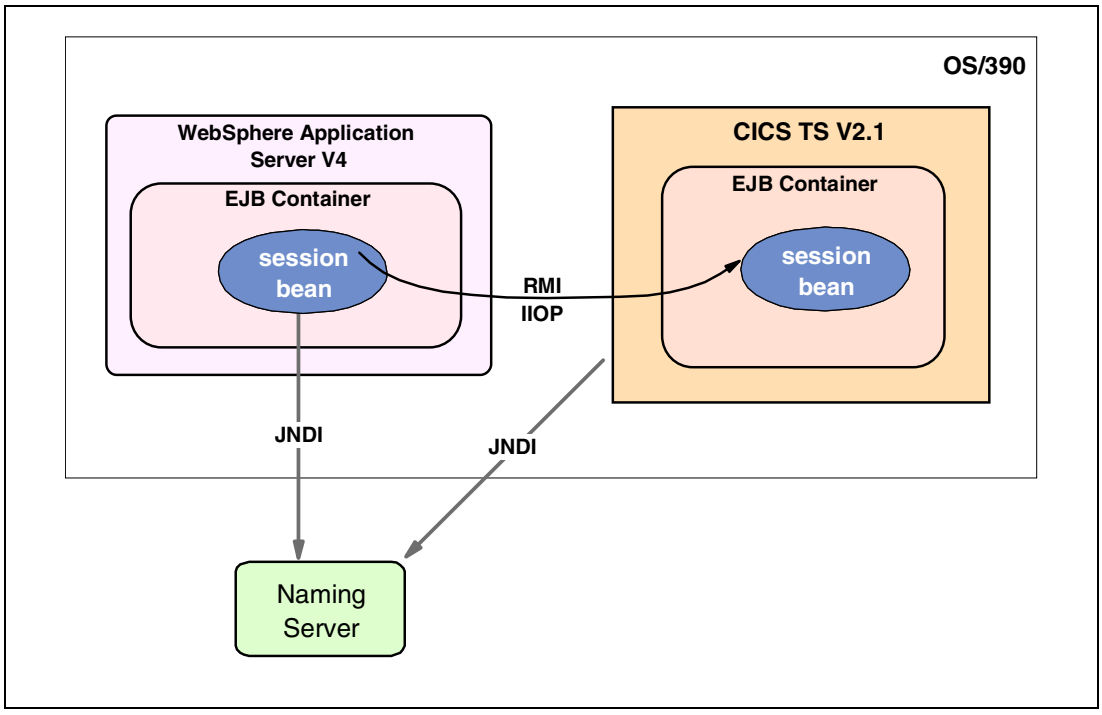


Figure 3-7 Session bean, WebSphere V4.0 — invoking session bean in CICS TS V2.1

The method calls between the enterprise beans use the communication protocol RMI over IIOP. The transaction context is passed in the IIOP header on a method call, and this is used by the enterprise bean container to notify the transaction coordinator of transactional events.

In summary, invoking a session bean in CICS from a session bean in WebSphere provides the following useful features:

- Adherence to the standard J2EE component model is ensured.
- Full two-phase commit protocol between all resource managers is provided.
- Invocations may be in both directions from WebSphere to CICS and from CICS to WebSphere. Furthermore, a session bean in CICS may invoke either a session bean or an entity bean in WebSphere.

The following limitations of invoking enterprise beans in CICS from a session bean in WebSphere must also be considered:

- The CICS region must be at version CICS TS V2.1 or later.
- Only session beans may be invoked in CICS; entity beans are not supported.

Part 2

CICS TS V2.1: Systems programming

In this part we detail how to set up and configure the enterprise bean support in a CICS TS V2.1 region, how to use the various new tools and features required, and how to deploy and test the product samples. Following this, we provide information on how to diagnose and fix problems when deploying and testing enterprise beans in CICS.

4

Installation considerations for CICS TS V2.1

This chapter describes how we installed and configured the EJB support in CICS Transaction Server for z/OS V2.1. It is divided into two sections, the first section describing how we defined and configured the various components required for EJB support, and the second section explaining how we verified that each component was functioning correctly.

The various components of the CICS EJB server are illustrated in Figure 4-1.

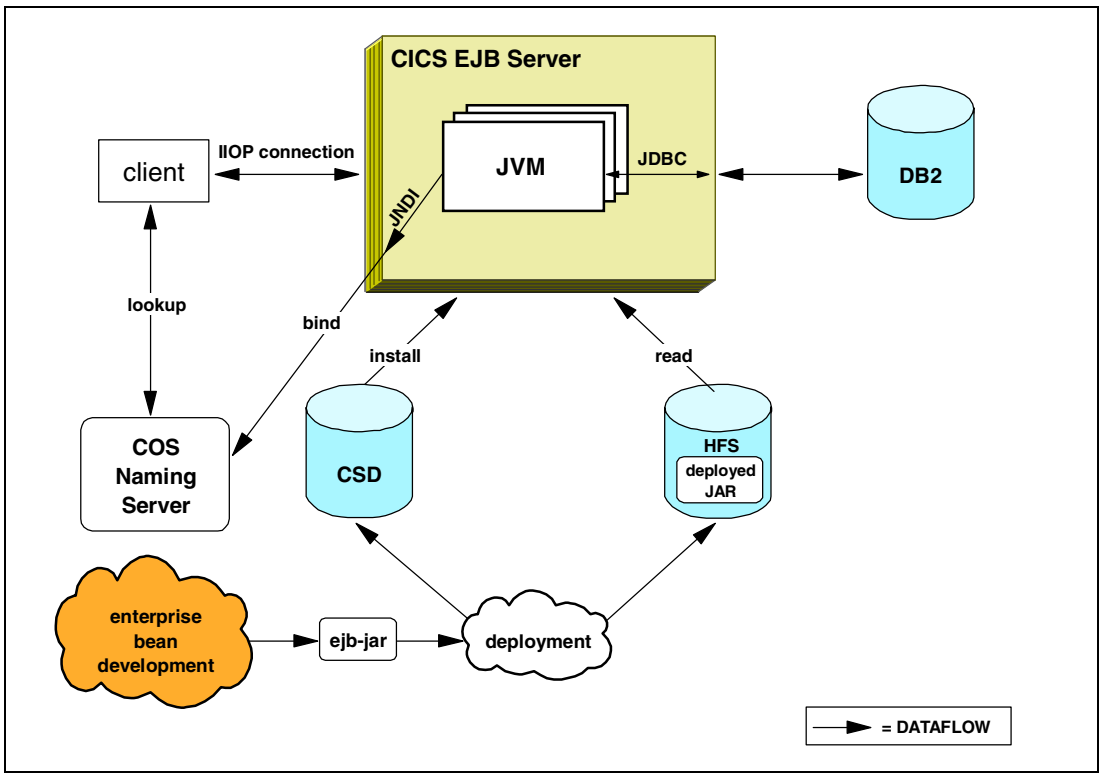


Figure 4-1 Overview of the CICS TS V2.1 EJB components

4.1 Installation and configuration

CICS TS V2.1 as an EJB server makes use of the following components:

- Traditional OS/390 resources such as partitioned data sets (PDS) and VSAM data sets.
- UNIX System Services (USS) resources for files and directories, the persistent, reusable Java Virtual Machine (JVM) and TCP/IP.
- WebSphere Application Server for Windows NT for running the COS Naming Server, and for running Web applications such as the CICS development deployment tool.
- Individual development workstations requiring the CICS deployment tool and an integrated development environment such as provided by VisualAge for Java.

Therefore, in order to successfully install and configure CICS TS V2.1, along with the traditional OS/390 based skills, it is helpful to also have basic UNIX skills, familiarity with Windows NT, and knowledge of WebSphere Application Server administration.

In this chapter we show you how to come to grips with these various components from the point of view of a traditional CICS systems programer who may have less skill in the areas outside of OS/390.

We do not describe how to install the CICS base product from scratch, but assume that the base code has already been installed and that a basic CICS system is operational.

The following list summarizes the tasks we completed to enable EJB support in CICS, each of which is described in the sections that follow:

1. Do the initial preparation.
2. Create HFS directories and files.
3. Define MVS VSAM and PDS data sets.
4. Tailor the CICS startup JCL.
5. Install CICS resource definitions.
6. Build a COS Naming Server.
7. Set up the deployment tools.

4.1.1 Initial preparation

Before beginning the installation process we did some initial preparation.

We made sure that we had the following resources available:

- OS/390 V2.8 configured with UNIX System Services and TCP/IP. Note that OS/390 includes as base elements many products required by CICS TS. See the *Program Directory for CICS Transaction Server for z/OS*, GI10-2525, for information on the requirements for running CICS TS V2.1.
- The IBM Developer Kit for OS/390 Java 2 Technology Edition installed.
- A Windows NT workstation with Service pack 6 installed to be used to run WebSphere Application Server for the COS Naming Server and the Web applications.
- The CICS supplied CD-ROMs for WebSphere Application Server, the CICS deployment tools, and the CICS Information Center.

We established the following information about the environment and its configuration parameters that we would need to use during the configuration (Table 4-1).

Table 4-1 Information needed prior to starting installation

OS/390	Workstation	value we used
CICS_HOME *(CICS install location)*		`/usr/lpp/cicsts/cicsts21`
JAVA_HOME *(Java install location)*		`/usr/lpp/java213d/J1.3`
CICS user home *(CICS working directory)*		`/u/cicsts21`
OS/390 TCP/IP Hostname		`wtsc690e.itso.ibm.com`
TCP/IP free ports		10500-10599
CICS APPLID		SCSCPJA5
	TCP/IP Host Name	`hecate.almaden.ibm.com`

4.1.2 Creating HFS directories and files

This section describes how we defined a new HFS to OS/390 UNIX System Services, and how we used this HFS for the directories and files needed by CICS.

Important: We allocated a single underlying dataset for the HFS for our CICS user home directory `/u/cicsts21`, which was therefore shared among all our CICS regions. In a more active environment, consideration should be given to having separate datasets for individual regions, and also for the `work` subdirectory (which can contain numerous `stdout` and `stderr` files) in order to prevent a full HFS affecting multiple CICS regions.

Creating a new HFS

CICS requires several HFS directories. We allocated a new HFS dataset called `OMVS.SC69.CICSTS21` and mounted this at the directory `/u/cicsts21`.

The dataset for the HFS can be created from the TSO ISHELL environment. Select the **File_systems** pull-down menu and choose option 2, see Figure 4-2 and Figure 4-3.

```
 File  Directory  Special_file  Tools  File_systems  Options  Setup  Help
------------------------------------ ┌──────────────────┐ ----------
                          OpenMVS I  │ 2 _1. Mount table...  │
                                     │   2. New...           │
Enter a pathname and do one of these:│   3. Mount(O)...      │
                                     └──────────────────┘

    - Press Enter.
    - Select an action bar choice.
    - Specify an action code or                  command line.
                          /u/cicsts21

Return to this panel to work with a different pathname.
                                                          More:     +
   /u/cicsts21
   ___________________________________________________________________
   ___________________________________________________________________
```

Figure 4-2 Creating an HFS with TSO ISHELL — 1

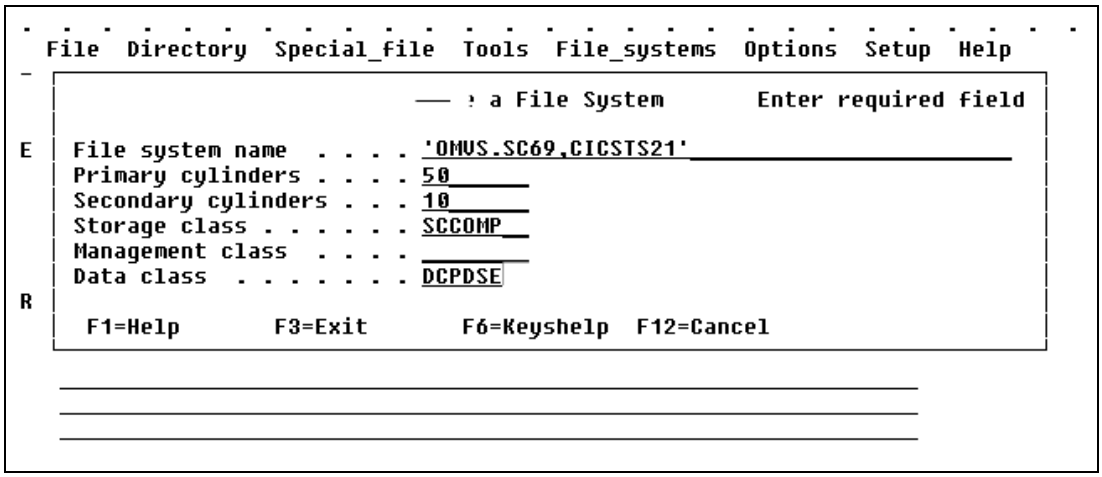


Figure 4-3 Creating an HFS with TSO ISHELL — 2

Once the HFS data set has been allocated, you must mount it at a mount point (an HFS directory), either with a TSO MOUNT command, or by using option 3 from the ISHELL File_systems menu. The mount point should be an empty directory, otherwise its contents will be hidden (but not deleted). We used the USS **mkdir** command to create the directory `/u/cicsts21` and mounted our HFS at this directory using the ISHELL File_systems menu option 3, as shown in Figure 4-4.

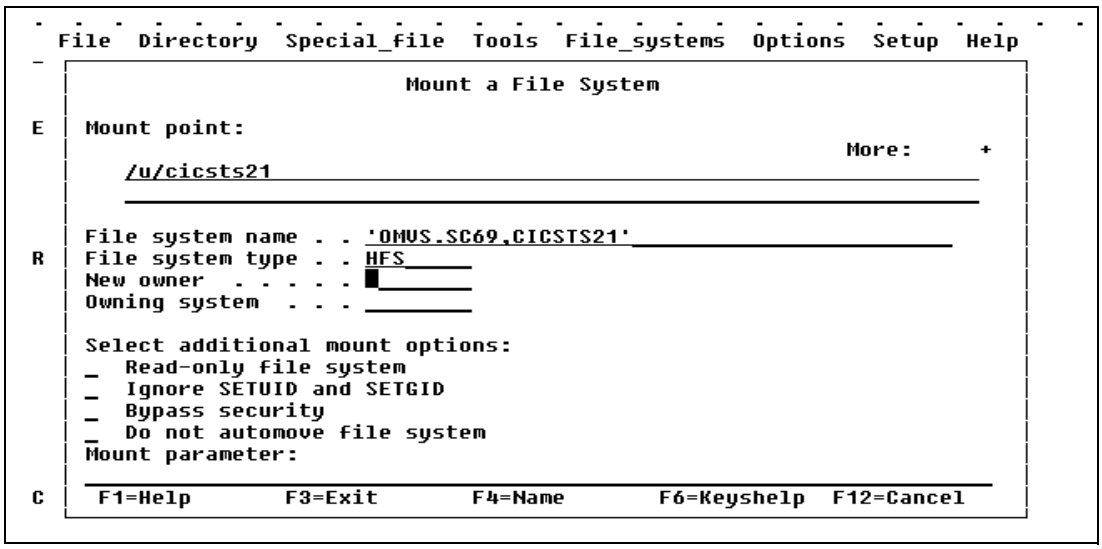


Figure 4-4 Mounting an HFS with TSO ISHELL

Tip: The USS shell command **df -k** can be used to provide a list of mounted file systems, and also provides a useful display of space utilization in 1KB blocks.

To have the mounted HFS file systems available after the OS/390 system is IPLed, a MOUNT statement needs to be added to the BPXPRMXX member in SYS1.PARMLIB. Our additions to BPXPRMXX were as follows:

```
MOUNT FILESYSTEM('OMVS.SC69.CICSTS21')
    TYPE(HFS)
    MODE(RDWR)
    MOUNTPOINT('/u/cicsts21')
```

Defining the CICS HFS directories

Within this newly created HFS we created several subdirectories for use by CICS. We used the TSO **OMVS** command to enter a USS shell and used the UNIX **mkdir** command to create the directories. We also copied the CICS supplied JVM system property file, `dfjjvmpr.props`, from the CICS install directory to our newly defined props subdirectory. This default CICS installation directory is `/usr/lpp/cicsts/cicsts21/props`.This is shown in Example 4-1.

Example 4-1 Defining the CICS HFS directories

```
CICSRS1 @ SC69:/u/cicsrs1>cd /u/cicsts21
CICSRS1 @ SC69:/u/cicsts21>mkdir props
CICSRS1 @ SC69:/u/cicsts21>mkdir work
CICSRS1 @ SC69:/u/cicsts21>mkdir shelf
CICSRS1 @ SC69:/u/cicsts21>mkdir lib
CICSRS1 @ SC69:/u/cicsts21>mkdir djars
CICSRS1 @ SC69:/u/cicsts21>ls
djar   lib    props  shelf  work
CICSRS1 @ SC69:/u/cicsts21>cd work
CICSRS1 @ SC69:/u/cicsts21/work>mkdir SCSCPJA5
CICSRS1 @ SC69:/u/cicsts21/work>ls
SCSCPJA5
CICSRS1 @ SC69:/u/cicsts21/work>cd ../props
CICSRS1 @ SC69:/u/cicsts21/work>cp /usr/lpp/cicsts/cicsts21/props/dfjjvmpr.props
/u/cicsts21/props
CICSRS1 @ SC69:/u/cicsts21/props>ls
dfjjvmpr.props
CICSRS1 @ SC69:/u/cicsts21/props>exit
>>>> FSUM2331 The session has ended.  Press <Enter> to end OMVS.
```

These different directories are used as follows:

cicsts21/props This is used to hold the JVM system properties files; for further details, see “JVM system properties files” on page 69.

cicsts21/work This has a subdirectory for each CICS region’s APPLID, where the java `stdin`, `stdout`, and `stderr` files are stored. It is referenced in the WORK_DIR parameter in the JVM system properties file as shown in Example 4-5 on page 74.

cicsts21/shelf CICS automatically creates a subdirectory in the shelf named after the region’s APPLID. and within that directory a subdirectory for each CorbaServer. This is used primarily for storing deployed JAR files, but can also be used to store an enterprise beans’s Interoperable Object Reference (IRO) in place of a Naming Server.

cicsts21/lib We used this directory to hold the utility JAR files required by the applications we developed later in this book. For example we copied here the Common Connector framework JAR file `ccf.jar`, for the example we used later in the redbook.

cicsts21/djars This is where we kept the CICS-deployed JAR files for our enterprise beans.

JVM system properties files

The `dfjjvmpr.props` file defines the basic information that the JVM needs to know about its environment. Since the CICS default installation only allows system administrator user IDs to update the supplied file `/usr/lpp/cicsts/cicsts21/props/dfjjvmpr.props`, we created a copy to be shared by all of our regions by copying this `/u/cicsts21/props/dfjjvmpr.props`.

For now, the only change we need to make to the system properties file is to add the URL of our WebSphere COS Naming Server (`iiop://hecate.almaden.ibm.com:900`) which will be used for storing the IORs of the enterprise bean we publish to the COS Naming server. We add the following line to our file dfjjvmpr.props:

```
java.naming.provider.url=iiop://hecate.almaden.ibm.com:900
```

CICS security authorization to HFS files

The CICS region user ID needs to be given access to the various HFS files and directories. On our system we ran CICS as a started task, so CICS ran under the user ID STC. User STC was in the same RACF CICS group name as the user we used to define the directories therefore, we were able to give CICS access to our HFS directories with a single command as follows:

```
chmod -R 775 /u/cicsts21
```

Chapter 21, “Managing security for enterprise beans” in *Java applications in CICS*, SC34-5881, gives more complete information on security considerations. A diagram of the completed directory hierarchy and file locations of our HFS is shown in Figure 4-5.

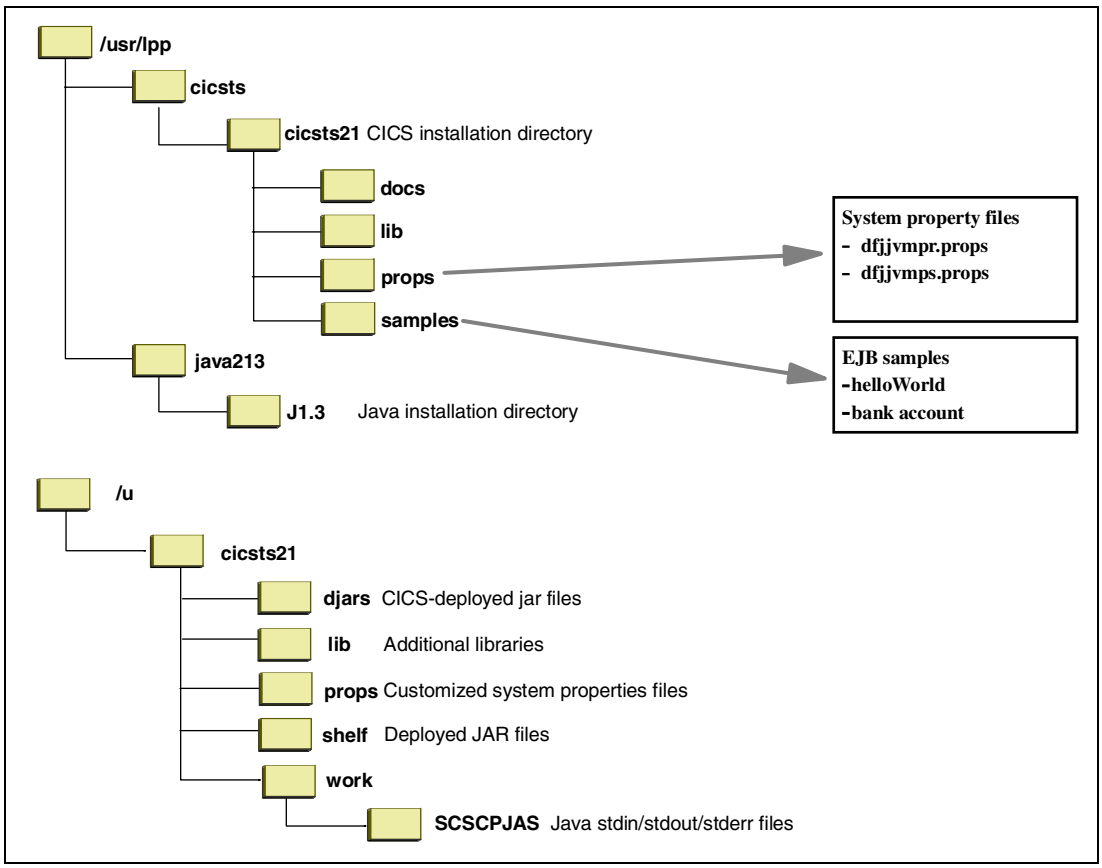


Figure 4-5 The OS/390 HFS directory structure

4.1.3 Defining OS/390 data sets

The following sections describe the changes we made to the following OS/390 data sets:

- CICS System Definition File
- EJB Object Store
- EJB Directory
- DJAR mapping data set
- JVM Profile PDS

The CICS System Definition (CSD) file

With CICS TS V2.1 the CSD file maximum record size has been increased from 500 to 2000 bytes. This is to accommodate the new resources, such as CORBASERVER and REQUESTMODEL, which can have parameters containing HFS paths. As each HFS path can be up to 255 bytes long, the CICS TS V1.3 CSD record size cannot accommodate these resources.

As we wanted to reuse the CSD from our previous CICS TS V1.3 systems, this meant we had to define a new VSAM CSD data set with the larger record size, copy the old CSD into the new data set, and then run a DFHCSDUP UPGRADE against this new data set. The JCL we used to do this is shown in Example 4-2.

Example 4-2 JCL to migrate a CICS TS V1.3 CSD to CICS TS V2.1

```
//CICSRS1C JOB (999,POK),NOTIFY=&SYSUID,
//       CLASS=A,MSGCLASS=T,MSGLEVEL=(1,1),TIME=1440
//*   CREATE A NEW CSD FOR CICS TS 2.1
//ALTERDEF EXEC PGM=IDCAMS,REGION=0M
//SYSPRINT DD SYSOUT=*
//AMSDUMP DD SYSOUT=*
//SYSIN DD *
 DELETE CICSSYSF.CICSTS21.DFHCSD
 DEFINE CLUSTER ( -
   NAME(CICSSYSF.CICSTS21.DFHCSD) -
   VOLUMES(TOTCI3) -
   KEYS(22 0) -
   INDEXED -
   RECORDS(6000 1000) -
   RECORDSIZE(200 2000) -
   FREESPACE(10 10) -
   SHAREOPTIONS(2) ) -
DATA -
   (NAME(CICSSYSF.CICSTS21.DFHCSD.DATA) -
   CONTROLINTERVALSIZE(8192)) -
  INDEX -
   (NAME(CICSSYSF.CICSTS21.DFHCSD.INDEX))
/*
//* COPY THE OLD CICS TS1.3 CSD CONTENTS INTO THE NEW FILE
//REPROCSD EXEC PGM=IDCAMS,REGION=0M,COND=(5,LT,ALTERDEF)
//SYSPRINT DD SYSOUT=*
//AMSDUMP DD SYSOUT=*
//SYS01    DD DSN=CICSSYSF.CICSTS13.DFHCSD,DISP=SHR
//SYSIN    DD *
    REPRO INFILE(SYS01) -
    OUTDATASET(CICSSYSF.CICSTS21.DFHCSD )
/*
//* UPGRADE THE NEW CSD FOR CICS TS 2.1
//CSDUP    EXEC PGM=DFHCSDUP,REGION=4M
//STEPLIB   DD DISP=SHR,DSN=CICSTS21.CICS.SDFHLOAD
//DFHCSD    DD DISP=SHR,DSN=CICSSYSF.CICSTS21.DFHCSD
```

```
//SYSPRINT  DD SYSOUT=*
//SYSIN DD *
  UPGRADE REPLACE
/*
```

Note: Even with this increase in CSD record size, it is still possible to share a CICS TS V2.1 format CSD with earlier CICS releases. See Chapter 3, “Resource Definition (online) changes”, of the *CICS Migration Guide*, GC34-5699, for instructions on how to do this.

The EJB object store and EJB data sets

There are two VSAM data sets, DFHEJOS and DFHEJDIR used by CICS to internally manage the running of enterprise beans. The function of these data sets is described in detail in 2.4, “The CICS EJB Server architecture” on page 40.

DFHEJOS and DFHEJDIR are VSAM KSDS data sets that require no initialization. CICS supplies JCL to create these files in member DFHDEFDS of the XDFHINST data set. We copied these supplied definition statements to our own job, which is shown in Example 4-3, the only change being to modify the VOLUME ID to match our systems. In other environments some consideration may need to be given the DFHEJOS record size, for details see Chapter 15, “Defining the EJB data sets”, in the *CICS System Definition Guide,* SC34-5725.

Example 4-3 JCL to define the DFHEJOS and DFHEJDIR VSAM data sets

```
//EJBFILES JOB (999,POK),'CICSTS21',MSGCLASS=T,CLASS=A,
//             NOTIFY=&SYSUID
//EJBDEF EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
 /* DEFINE EJB DIRECTORY */
 DELETE CICSSYSF.CICS610.PJA5.DFHEJDIR
 DEFINE -
   CLUSTER(NAME(CICSSYSF.CICS610.PJA5.DFHEJDIR) -
     INDEXED -
     LOG(UNDO) -
     CYL(2 1) -
     VOLUME(TOTCI3) -
     RECORDSIZE(1017 1017 ) -
     KEYS(16 0 ) -
     FREESPACE (10 10 ) -
     SHAREOPTIONS(2 3)) -
   DATA (NAME(CICSSYSF.CICS610.PJA5.DFHEJDIR.DATA) -
     CONTROLINTERVALSIZE(1024)) -
   INDEX (NAME(CICSSYSF.CICS610.PJA5.DFHEJDIR.INDEX))
 /* DEFINE EJB OBJECT STORE */
 DELETE CICSSYSF.CICS610.PJA5.DFHEJOS
 DEFINE -
   CLUSTER(NAME(CICSSYSF.CICS610.PJA5.DFHEJOS)-
     INDEXED -
     LOG(NONE) -
     CYL(2 1) -
     VOLUME(TOTCI3) -
     RECORDSIZE(8185 8185 ) -
     KEYS(16 0 ) -
     FREESPACE (10 10 ) -
     SHAREOPTIONS(2 3)) -
   DATA (NAME(CICSSYSF.CICS610.PJA5.DFHEJOS.DATA) -
     CONTROLINTERVALSIZE(8192)) -
```

```
   INDEX (NAME(CICSSYSF.CICS610.PJA5.DFHEJOS.INDEX))
/*
//*
```

The DFHADJM data set (optional)

DFHADJM is used by the CICS development deployment tool as the DJAR mapping dataset. It is used to store mappings between the CICS DJAR resource definitions created by the CICS development deployment tool, and the JAR files on HFS. It is only required in a CICS region where the CICS development deployment tool will be used to deploy enterprise beans.

As we were using the CICS development deployment tool in our CICS region, we defined this data set as shown in Example 4-4. The VOLID parameter is the only change from the installation supplied definition, and no initialization of the data set is required.

Example 4-4 JCL to define the DFHADJM data set

```
//EJBFILES JOB (999,POK),'CICSTS21',MSGCLASS=T,CLASS=A,
//             NOTIFY=&SYSUID
//EJBDEF EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
/*DEFINE A JAR MAPPING DATA SET */
 DELETE CICSSYSF.CICS610.PJA5.DFHADJM
 DEFINE -
   CLUSTER(NAME(CICSSYSF.CICS610.PJA5.DFHADJM)-
     INDEXED -
     LOG(NONE) -
     VOLUME(TOTCI3) -
     RECORDSIZE(263 263 ) -
     RECORDS(12000 00 ) -
     KEYS(8 0) -
     SHAREOPTIONS(2 3)) -
   DATA (NAME(CICSSYSF.CICS610.PJA5.DFHADJM.DATA) -
     CONTROLINTERVALSIZE(1024)) -
   INDEX (NAME(CICSSYSF.CICS610.PJA5.DFHADJM.INDEX))
/*
//*
```

DFHJVM and JVM profiles

The DFHJVM data set contains members (JVM profiles) that are referenced by the JVMPROFILE attribute of a CICS PROGRAM resource definition. All CICS Java programs require a valid JVM profile. JVM profiles can be shared by CICS Java programs or individual profiles can be defined for different purposes such as debugging and production.

Important: The JVMPROFILE name determine if a Java program can reuse an existing CICS JVM. If many different JVMPROFILES are defined, much of the performance benefits of the persistent reusable JVM may be lost.

The CICS install image provides the CICSTS21.CICS.XDFHENV data set, which contains sample JVM profile members. Since this data set will be overwritten on a CICS refresh, we used ISPF 3.2 to allocate a new data set CICSSYSF.CICS610.DFHJVM and copied the samples to our new data set.

We then tailored the following parameters of the supplied profile member, DFHJVMPR, to match the HFS directories we created in 4.1.2, “Creating HFS directories and files” on page 67, and the information we gathered in Table 4-1 on page 67. Our DFHJVMPR member after tailoring is shown in Example 4-5 with the changes we made in bold type. Note that other profile members can be defined as required, for example, we define a profile for use when debugging an enterprise bean in Chapter 5, “Troubleshooting enterprise beans in CICS TS V2.1” on page 103.

Attention: You should be careful when modifying the DFHJVMPR profile, as it is used by CICS internal functions. A better approach is to copy this and use a new REQUESTMODEL definition to point the required method invocations at this JVM profile.

Example 4-5 Our tailored DFHJVMPR member

```
WORK_DIR=/u/cicsts21/work/&APPLID;
INVOKE_DFHJVMAT=NO
JVMPROPS=/u/cicsts21/props/dfjjvmpr.props
#VMPROPS=/usr/lpp/cicsts/cicsts21/samples/ejb/helloworld/IVP.properties
LIBPATH=\
   /usr/lpp/cicsts/cicsts21/lib: /usr/lpp/cicsts/cicsts21/ctg:\
   /usr/lpp/java213d/J1.3/bin:/usr/lpp/java213d/J1.3/bin/classic
#
STDIN=dfhjvmin
STDOUT=dfhjvmout -generate
STDERR=dfhjvmerr -generate
CICS_DIRECTORY=/usr/lpp/cicsts/cicsts21
JAVA_HOME=/usr/lpp/java213d/J1.3
SHOWVERSION=YES
VERBOSE=NO
CLASSPATH=/u/cicsts21/lib
#
Xcheck=NO
Xdebug=NO
Xms=1M
Xmx=32M
Xnoagent=NO
Xnoclassgc=NO
Xoss=4M
Xss=512K
Xresettable=YES
Xverify=none
```

In later chapters we customize this profile further by adding supporting files to the trusted middleware classpath using the TMSUFFIX parameter. For further details refer to:

- 7.3.6, “Adding the supporting JAR files to the trusted middleware classpath” on page 194 for details on adding the classes for the Java Record Framework and Enterprise Access builder.
- “TMSUFFIX” on page 280, for details on adding the files for DB2 and SQLJ support.

Tip: Note you should ensure that the LIBPATH, CLASSPATH or TMSUFFIX paths do not contain a space character before a trailing continuation character (\) — otherwise, the subsequent directories will not be found.

4.1.4 Tailoring the CICS startup JCL

We made several changes to our existing CICS startup JCL to enable EJB support. These included increasing the CICS region size, tailoring the storage limits, adding DD statements for the new data sets, and several changes to the CICS SIT overrides.

The CICS region size

Using a JVM within CICS considerably increases the region size required. It is recommended that as a minimum, a region size of 1000M is used. To accommodate this, we used REGION=0M in our CICS startup JCL, which ensures that MVS gives all available private storage to the job.

Adding DD statements

The new data sets created in the previous steps need DD statements added to the CICS JCL. We added DD statements for all the new data sets to our JCL as follows:

```
//DFHJVM   DD  DSN=CICSSYSF.CICS610.DFHJVM,DISP=SHR
//DFHEJDIR DD  DSN=CICSSYSF.CICS610.PJA5.DFHEJDIR,DISP=SHR
//DFHEJOS  DD  DSN=CICSSYSF.CICS610.PJA5.DFHEJOS,DISP=SHR
//DFHADJM  DD  DSN=CICSSYSF.CICS610.PJA5.DFHADJM,DISP=SHR
```

In a more complex environment with separate development and production regions, and with production regions split into listener regions and AORs, then the following would apply:

- All regions running Java programs require DFHJVM.
- All listener regions require DFHEJDIR.
- All AORs require DFHEJOS.
- Only regions using the CICS development deployment tool require DFHADJM.

Our complete CICS startup JCL is shown in Example 4-6.

Example 4-6 Our CICS startup JCL

```
//CICSTSEJB PROC START='INITIAL',REG='0M',OUTC='*'
// COMMAND 'V NET,ACT,ID=APCPJA5,ALL'
//CICS610  EXEC PGM=DFHSIP,REGION=&REG,TIME=1440,
//             PARM=('START=&START','SYSIN')
//STEPLIB  DD  DSN=CICSTS21.CICS.SDFHAUTH,DISP=SHR
//SYSABEND DD  SYSOUT=&OUTC
//SYSIN    DD  DSN=CICSSYSF.CICSTS21.SYSIN(PJA5SIT),DISP=SHR
//DFHRPL   DD  DSN=CICSSYSF.APPL61.LOADLIB,DISP=SHR
//         DD  DSN=CICSTS21.CICS.SDFHLOAD,DISP=SHR
//         DD  DSN=CEE.SCEECICS,DISP=SHR
//         DD  DSN=CEE.SCEERUN,DISP=SHR
//         DD  DSN=CICSSYSF.APPL61.LOADLIB,DISP=SHR
//DFHCXRF  DD  SYSOUT=&OUTC
//DFHAUXT  DD  DISP=SHR,DCB=BUFNO=5,
//             DSN=CICSSYSF.CICS610.PJA5.DFHAUXT
//DFHBUXT  DD  DISP=SHR,DCB=BUFNO=5,
//             DSN=CICSSYSF.CICS610.PJA5.DFHBUXT
//DFHDMPA  DD  DSN=CICSSYSF.CICS610.PJA5.DFHDMPA,DISP=SHR
//DFHDMPB  DD  DSN=CICSSYSF.CICS610.PJA5.DFHDMPB,DISP=SHR
//DFHINTRA DD  DSN=CICSSYSF.CICS610.PJA5.DFHINTRA,DISP=SHR
//DFHTEMP  DD  DSN=CICSSYSF.CICS610.PJA5.DFHTEMP,DISP=SHR
//DFHGCD   DD  DSN=CICSSYSF.CICS610.PJA5.DFHGCD,DISP=SHR
//DFHLCD   DD  DSN=CICSSYSF.CICS610.PJA5.DFHLCD,DISP=SHR
//DFHLRQ   DD  DSN=CICSSYSF.CICS610.PJA5.DFHLRQ,DISP=SHR
//DFHCSD   DD  DSN=CICSSYSF.CICSTS21.DFHCSD,DISP=SHR
//DFHJVM   DD  DSN=CICSSYSF.CICS610.DFHJVM,DISP=SHR
//DFHEJDIR DD  DSN=CICSSYSF.CICS610.PJA5.DFHEJDIR,DISP=SHR
```

```
//DFHEJOS  DD  DSN=CICSSYSF.CICS610.PJA5.DFHEJOS,DISP=SHR
//DFHADJM  DD  DSN=CICSSYSF.CICS610.PJA5.DFHADJM,DISP=SHR
//DFHCMACD DD  DSN=CICSSYSF.CICSTS21.DFHCMACD,DISP=SHR
```

SIT parameter changes

We made the following changes to our CICS system initialization parameters:

- **EDSALIM=500M:** This is the recommended minimum value.
- **TCPIP=YES:** This is mandatory for EJB support.
- **MAXOPENTCBS=10:** This is the recommended maximum value.

We also had to remove the DCT parameter, as this is no longer supported in CICS TS V2.1.

Our complete SIT overrides member is shown in Example 4-7.

Example 4-7 CICS TS V2.1 SIT overrides

```
AICONS=YES,            AUTOINSTALL FOR CONSOLES
AKPFREQ=1000,
APPLID=SCSCPJA5
AUXTR=OFF,
AUXTRSW=NEXT,
CMDPROT=NO,
CSDBUFND=6,
CSDBUFNI=5,
CSDDISP=SHR,
CSDDSN=CICSSYSF.CICSTS21.DFHCSD,
CSDFRLOG=NO,
CSDINTEG=UNCOMMITTED,
CSDJID=NO,
CSDLSRNO=NONE,
CSDRECOV=BACKOUTONLY,
CSDRLS=NO,
CSDSTRNO=4,
DSRTPGM=NONE,            DISTRIBUTED ROUTING PROGAM
DB2CONN=YES,
EDSALIM=500M,
FCT=NO,
FTIMEOUT=30,
IRCSTRT=YES,
ISC=YES,
GMTEXT='SCSCPJA5 CICS610 SG24-6284',
GRPLIST=(DFHLIST,TS21LIST,PJALIST,PJA5LIST),
MAXOPENTCBS=10,
MN=ON,
MNCONV=YES,
MNPER=ON,
MNEVE=ON,
MROLRM=YES,
MNSUBSYS=PJA1,
MXT=200,
PGAIPGM=ACTIVE,
RENTPGM=NOPROTECT,
RLS=YES,
RRMS=YES,
SEC=NO,
SIT=6$,
SPOOL=YES,
STNTR=ALL,
```

```
SYSIDNT=PJA5
GTFTR=OFF,
STATRCD=OFF,
TCPIP=YES,
TST=NO,
XLT=NO,
XRF=NO,
START=INITIAL,
.END
```

4.1.5 Installing CICS resource definitions

There are several supplied resource definitions groups that are required for EJB support. However, to start with, we brought up CICS as it was, and installed the groups discussed below manually using the command **CEDA INS GROUP**. Later, having verified that everything worked as we expected, we added the groups to a list specified in our SIT GRPLIST parameter to automate the installs on subsequent cold starts.

CICS EJB support resource definitions

The transaction and program resource definitions required for EJB support are included in the supplied group DFHIIOP. This group is included in the default list DFHLIST, so the resources will be installed automatically if DFHLIST is used.

The EJB object store and directory data sets need file definitions in CICS. Three groups are supplied with default definitions depending on how the files need to be shared. These are DFHEJVS, DFHEJVR, and DFHEJCF. As we were building a single region CICS EJB server we used group DFHEJVS. See “Defining EJB directory and object store data sets” in Chapter 15 of the *CICS System Definition Guide*, SC34-5725, for information on which group to use for a multi-region EJB server.

We installed the DFHEJVS group without making any modifications to it with the following command:

```
CEDA INS GROUP(DFHEJVS)
```

We then added it to one of our lists specified in the SIT GRPLIST parameter, using the following command:

```
CEDA ADD GROUP(DFHEJVS) LIST(PJALIST)
```

TCPIPSERVICE resource definition

Enterprise beans running in CICS need a TCPIPSERVICE, to enable the CICS region to listen for incoming IIOP method requests and pass them on to the request receiver transaction, CIRR. We defined the TCPIPSERVICE PJA5IIOP for our CICS region (Figure 4-6), that we shared among all the CORBASERVER definitions in the region.

```
 OVERTYPE TO MODIFY                                        CICS RELEASE = 0610
  CEDA  ALter TCpipservice( PJA5     )
   TCpipservice   : PJA5
   GROup          : PJA5IIOP
   DEscription  ==> SHARED TCPIPSERVICE FOR SCSCPJA5
   Urm          ==>
   POrtnumber   ==> 10500              1-65535
   STatus       ==> Open               Open | Closed
   PRotocol     ==> Iiop               Iiop | Http
   TRansaction  ==> CIRR
   Backlog      ==> 00005              0-32767
   TSqprefix    ==>
   Ipaddress    ==>
   SOcketclose  ==> No                 No | 0-240000 (HHMMSS)
  SECURITY
   SSl          ==> No                 Yes | No | Clientauth
   Certificate  ==>
   Authenticate ==> No                 No | Basic | Certificate | AUTORegister
+                                      | AUTOMatic

                                                    SYSID=PJA5 APPLID=SCSCPJA5

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

Figure 4-6 TCPIPSERVICE resource definition

The important parameters in the TCPIPSERVICE are as follows:

- **TCpipservice** is the name of this resource definition.
- **POrtnumber** (10500) was chosen from one of the spare ports documented 4.1.1, “Initial preparation” on page 66.
- **PRotocol** must specify IIOP.
- **TRansaction** should specify CIRR (or an alias of CIRR), which is the CICS supplied transaction for receiving requests.
- **Ipaddress** we left as blank, meaning the default of *INADDR_ANY* will be used. This signifies that the TCPIPSERVICE will listen on all of the IP addresses known to TCP/IP for this OS/390 host. This has the advantage that the TCPIPSERVICE definition is not specific to a CICS region and can be shared among CICS regions on different OS/390 hosts.

It is possible to have multiple TCPIPSERVICE definitions installed within a CICS region at the same time. This can allow you to use different request receiver transactions for different ports, or to use different security levels for different ports. Security is controlled on each TCP/IP listener by the user exit DFHXOPUS; for further details refer to Appendix A, “Security customization: DFHXOPUS” on page 303.

CORBASERVER

The CORBASERVER resource describes the execution environment for enterprise beans and stateless CORBA objects. We defined the CORBASERVER PJA5; see Figure 4-7.

We used several CORBASERVER resources, one for each person developing enterprise beans, and a shared one for the CICS region that we all deployed our enterprise beans to after we had finished testing them. This allowed us avoid naming conflicts when deploying different enterprise beans of the same name within the same CICS region. This works by defining a unique JNDI prefix for each CORBASERVER.

Tip: The CEOT transaction has been changed with CICS TS V2.1 to aid the entering of lower case characters in HFS path parameters on CICS resource definitions. We used the command `CEOT TRANID` before entering the CEDA panels to allow us to easily enter lower case characters. See the manual *CICS Supplied Transactions*, SC34-5724, for more details.

```
OVERTYPE TO MODIFY                                        CICS RELEASE = 0610
 CEDA  ALter CORbaserver( PJA5 )
  CORbaserver    : PJA5
  Group          : PJA5IIOP
  Description  ==> SHARED CORBASERVER FOR SCSCPJA5
  Jndiprefix   ==> ITSO/PJA5
               ==>
               ==>
  SEssbeantime ==> 00 , 00 , 10       0-99 (Days,Hours,Mins)
  SHelf        ==> /u/cicsts21/shelf
               ==>
 SERVER ORB ATTRIBUTES
  Host         ==> wtsc69oe.itso.ibm.com
               ==>
  Port         ==> 10500              1-65535
  SSL          ==> No                 Yes | No | Clientcert
  SSLPort      ==> No                 No | 1-65535
 CLIENT ORB ATTRIBUTES
  Certificate  ==>

                                                  SYSID=PJA5 APPLID=SCSCPJA5
```

Figure 4-7 CORBASERVER resource definition

The important CORBASERVER parameters are as follows:

- **CORbaserver** is name of this resource definition.
- **Jndiprefix** should match the JNDI prefix defined for the enterprise bean in the client code. We used the prefix `ITSO/` followed by the CORBASERVER name PJA5.
- **Sessbeantime** is the time-out for stateful session beans that have been passivated, and for which no `remove()` method has yet been invoked. You should always code a time-out value to prevent the CICS object store from inadvertently filling up.
- **Host** is the OS/390 TCP/IP hostname.
- **Port** must match the port defined in the TCPIPSERVICE definition.

REQUESTMODEL

The REQUESTMODEL resource maps incoming requests to a particular request processor transaction ID. The CORBASERVER and REQUESTMODEL resources are described in more detail in Chapter 2, “CICS TS V2.1: The EJB Server” on page 31.

For each of the CORBASERVER resources we defined an associated REQUESTMODEL resource. We defined the REQUESTMODEL PJA5REQ associated with the PJA5 CORBASERVER resource (Figure 4-8).

```
OVERTYPE TO MODIFY                                          CICS RELEASE = 0610
 CEDA  DEFine Requestmodel( PJA5REQ  )
  Requestmodel   : PJA5REQ
  Group          : PJA5IIOP
  Description  ==>
  Corbaserver  ==> PJA5
  TYpe         ==> Ejb               Corba | Ejb | Generic
 EJB PARAMETERS
  Beanname     ==> *
               ==>
  INTFacetype  ==> Both              Both | Home | Remote
 CORBA PARAMETERS
  Module       ==>
               ==>
  INTErface    ==>
               ==>
 COMMON PARAMETERS
   OPeration    ==> *
                ==>
 TRANSACTION ATTRIBUTES
   TRansid      ==> 5IRP

                                                SYSID=PJA5 APPLID=SCSCPJA5
PF 1 HELP 2 COM 3 END            6 CRSR 7 SBH 8 SFH 9 MSG 10 SB 11 SF 12 CNCL
```

Figure 4-8 REQUESTMODEL resource definition

The parameters we defined for our REQUESTMODEL PJA5REQ are described bellow:

- **Requestmodel** is name of this resource definition.
- **Corbaserver** is the CORBASERVER resource this REQUESTMODEL is associated with.
- **TYpe** specifies if this REQUESTMODEL is used for CORBA requests, EJB requests, or both types of requests. We use the value `Ejb`, note that this value defaults to `Generic`, and if left to default, the CORBA parameters Module and Interface must also be specified.
- **Beanname** specifies which enterprise beans this REQUESTMODEL is applicable to. An asterisk(*) means all enterprise beans.
- **INTFacetype** specifies if this REQUESTMODEL applies to the enterprise beans Home interface, Remote interface, or both types of interface.
- **OPeration** specifies which operation (or Java methods) that this REQUESTMODEL applies to. A asterisk(*) means all methods.
- **TRansid** is the CICS transaction to run as the request processor if this REQUESTMODEL matches the incoming IIOP request. We used 5IRP which was defined as an alias of CIRP.

Request processor aliases

The REQUESTMODEL maps an incoming IIOP request to a request processor transaction. CICS defines a default request processor transaction, CIRP. CIRP runs the default request processor Java program DFJIIRP.

We defined individual aliases for the CIRP transaction and the DFJIIRP program for each of our CORBASERVERs. The reason for this is that it then allows each program alias to use a different JVMPROFILE. Each developer can then change the JVMPROFILE to point to their own profile, for example, to specify a different CLASSPATH or for specifying debugging options. See “DFHJVM and JVM profiles” on page 73 for more information on the JVMPROFILE.

We created these aliases by copying the supplied CIRP and DFJIIRP resource definitions with the commands:

```
CEDA COPY PROG(DFJIIRP) GROUP(DFHIIOP) TO(PJA5IIOP) AS(PJA5IIOP)
CEDA COPY TRANS(CIRP)   GROUP(DFHIIOP) TO(PJA5IIOP) AS(5IRP)
```

The only change to these definitions is then to change the new transaction, 5IRP, to use the new program by changing its PROGram attribute to PJA5IIOP.

Attention: At this stage it is possible to skip forward to 4.3.1, “Running the IVP OS/390 USS client application” on page 94 in order to test the basic CICS EJB server configuration. The CICS deployment tools are optional utilities and the COS Naming Server is not required to run the IVP, as the enterprise beans can be published to the HFS shelf directory.

CICS development deployment tool resource definitions

The following groups are only required for regions using the CICS development deployment tool. The two groups DFHADPD and DFHADFD require no modifications and can be installed as is. The group DFHADBD incudes definitions for a TCPIPSERVICE and CORBASERVER pair which need tailoring to match the environment.

We first copied the resources in the DFHADBD group to our own group with the command:

```
CEDA COPY GR(DFHADBD) TO(PJA5ADBD)
```

Then we changed the PORTNUMBER, HOST, and SHELF attributes as described previously in , “CORBASERVER” on page 79. We used the value `10599` for the PORTNUMBER, `wtsc69oe.itso.ibm.com` as the HOST, and `/u/cicsts21/shelf` as the SHELF.

We then installed the groups with the commands:

```
CEDA INS GROUP(DFHADPD)
CEDA INS GROUP(DFHADFD)
CEDA INS GROUP(PJA5ADBD)
```

We then added them to a list in our CICS GRPLIST as follows:

```
CEDA ADD GROUP(DFHADPD)  LIST(PJA5LIST)
CEDA ADD GROUP(DFHADFD)  LIST(PJA5LIST)
CEDA ADD GROUP(PJA5ADBD) LIST(PJA5LIST)
```

Finally, the DJAR resource defined within the *PJA5ADBD* group must be published to the COS Naming Server with the command:

```
CEMT PERFORM DJAR(DFHADJAR) PUBLISH
```

4.2 Setting up the workstation tools

CICS TS V2.1 supplies a set of workstation tools on CD-ROMs that accompany the product. The workstation tools comprise the following components:

- WebSphere Application Server Advanced Edition for Windows NT.

  This provides the COS Naming Server and also the servlet engine for the use of the CICS development deployment tool.
- The CICS Information Center
- The CICS JAR development tool for EJB Technology
- The CICS production deployment tool for EJB Technology
- The CICS development deployment tool for EJB Technology

4.2.1 WebSphere Application Server

CICS requires a COS Naming Server, which it uses to publish references to the home interfaces of enterprise beans. A CD-ROM containing WebSphere Application Server Advanced Edition for Windows NT Version 3.5.3 is shipped with CICS TS V2.1 for this purpose.

Inserting the CD-ROM should bring up the WebSphere InstallShield window as shown in Figure 4-9. We chose the *Full Installation* option and accepted all other defaults.

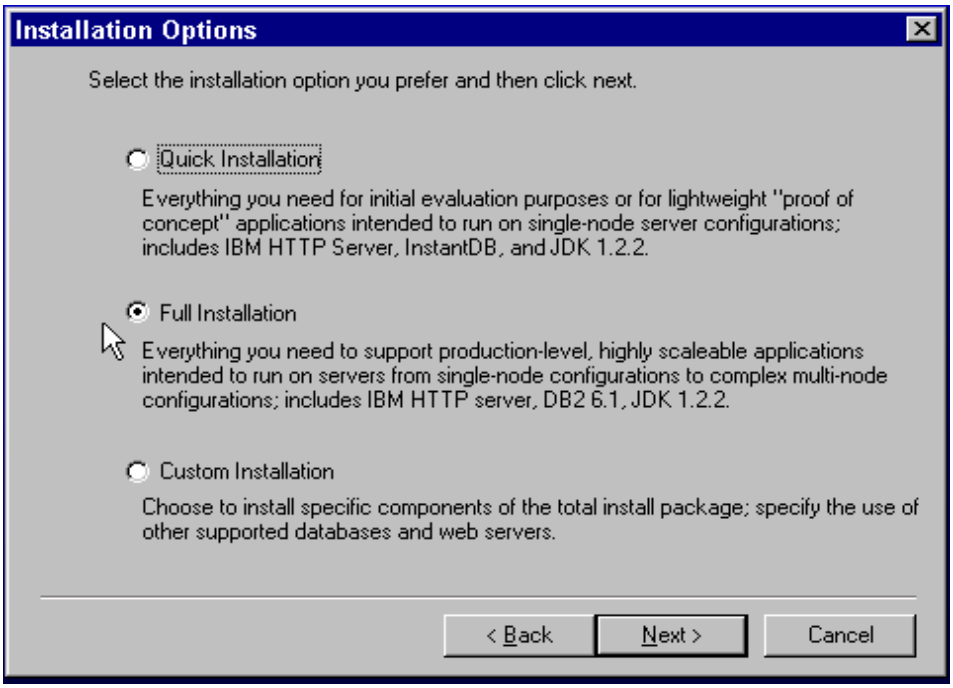


Figure 4-9 WebSphere InstallShield screen

It is recommended that the latest fixpack is applied to WebSphere Application Server after the install has completed. We applied fixpack 3, which is available for download from:

`http://www.ibm.com/software/webservers/appserv/efix.html`

To successfully apply the Fixpack, we had to first stop several WebSphere services in Windows NT. To do this from the Start button, choose **Start -> Settings -> Control Panel**, then double-click on the **Services** icon. In the Services window, we stopped the following three services:

- IBM HTTP Administration
- IBM HTTP Server
- IBM WS AdminServer

We then extracted the Fixpack files from the downloaded Zip file and ran the `Install.bat` file. This opens a Windows command prompt which asks several questions about the location of the WebSphere and HTTP directories, and then installs the fixes. An abbreviated listing of this is shown in Example 4-8 with our answers in bold, which match the locations used if all the defaults are taken during the WebSphere installation.

Example 4-8 Applying the WebSphere Fixpack

```
Enter the directory where WebSphere Application Server is installed:
c:\WebSphere\AppServer
...
Do you wish to upgrade the WebSphere Application Server samples?(Yes/No)
yes
In order to update the WebSphere Application Server Samples
the currently configured WebServer's doc root must be specified
Eg. C:\IBM HTTP Server\htdocs
Please enter your webserver's doc root path:
c:\IBM HTTP Server\htdocs
c:\IBM HTTP Server\htdocs
Is this correct?(Yes/No)
yes
142 File(s) copied
...
WARNING: If you install IBM HTTP Server PTF, you may not be able to uninstall
it cleanly.
Do you wish to upgrade the IBM HTTP Server 1.3.12:(Yes/No)
Yes
Enter the directory where the IBM HTTP Server 1.3.12 is installed:
c:\IBM HTTP Server
Upgrading IHS
...
2001/03/19 13:15:50
2001/03/19 13:15:50 Installation completed with no errors.
2001/03/19 13:15:50 Please view the activity log for details.
Press any key to continue . . .
241 File(s) copied
IBM WebSphere Application Server V3.5.3 Advanced Fixpack install
complete
```

After WebSphere was installed and the Fixpack applied, there was no configuration required to activate the COS Naming Server. The COS Naming Server function is provided by the WebSphere Admin Server component of WebSphere. This is an NT service that by default is set to be started manually. This behavior can be changed by changing **Startup Type** to **Automatic** from **Manual** for the *IBM WS AdminServer* Windows NT service.

As there is no tool provided with WebSphere to verify that the COS Naming Server is really being used when enterprise beans are published, we wrote a small Java utility program, JNDIList, that uses the Java Naming and Directory Interface (JNDI) to interrogate the COS Naming Server and display the results. This utility is described in more detail in 5.2, “WebSphere diagnostic aids” on page 115, and further details on how to obtain the code are provided in Appendix C, “Using the additional material” on page 315.

The first three of these tools are to be installed on development workstations. The last tool, the CICS development deployment tool, should be installed on a WebSphere Application Server and also requires definitions within the CICS region.

4.2.2 CICS Information Center

The CICS Information Center is a set of searchable online resources including all the product documentation for CICS TS V2.1. It is supplied with CICS on a single CD-ROM labeled *CICS TS V2.1 Publications*.

Running the **`setup.exe`** program on the CD-ROM starts the InstallShield Wizard, which prompts for:

- User information
- Install location (locally or on a server)
- The target installation directory

We install it locally on each of our workstations, letting all the options default. The default install directory is `C:\Program Files\IBM\CICS TS 2.1 Tools`.

After the Install has completed, the CICS Information Center can be started by choosing **Start -> Programs -> CICS TS 2.1 Tools -> CICS Information Center**. This starts your Web browser at the Information Center home page, as shown in Figure 4-10.

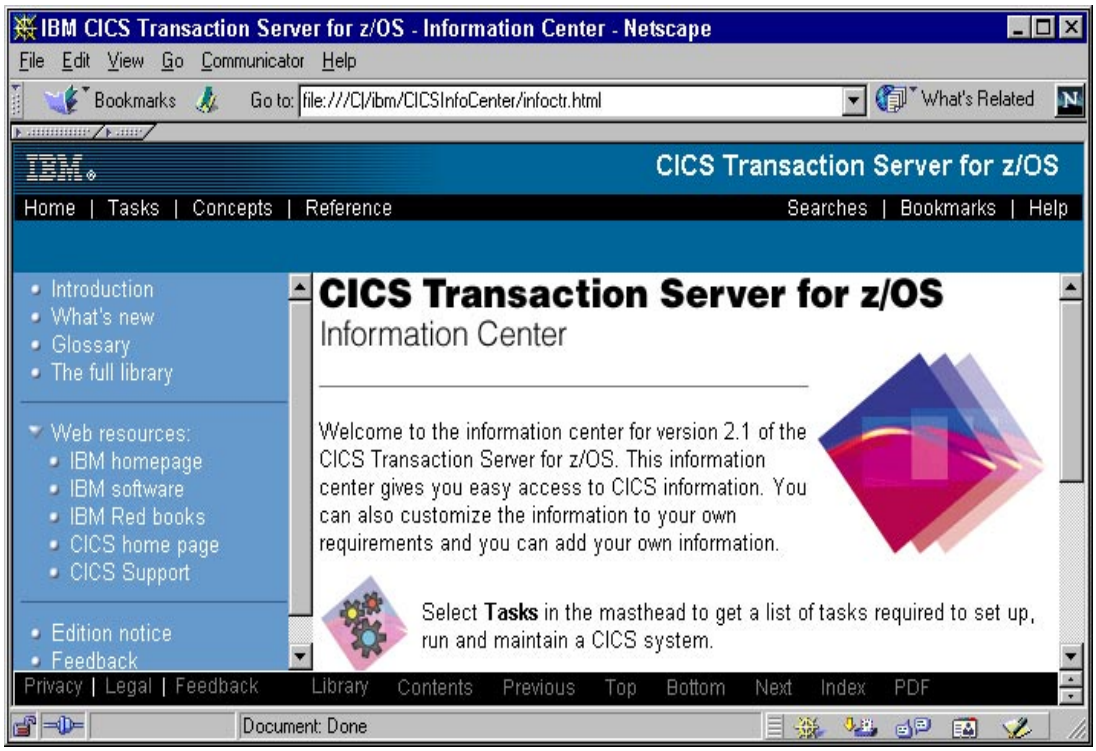


Figure 4-10 The CICS Information Center

4.2.3 CICS JAR development tool and production deployment tool

One of the CD-ROMs supplied with CICS is labeled *CICS TS V2.1 PC Tools*. This includes the *CICS JAR development tool*, the *CICS development deployment tool*, and the *CICS production deployment tool*.

We installed the *CICS JAR development tool* and the *CICS production deployment tool* on each workstation we were using for development. From the CD-ROM we ran the **`setup.exe`** program which starts the InstallShield Wizard. We chose the Custom install option, and *excluded* the CICS development deployment tool from being installed, leaving the other two tools selected. We then completed the install taking all the default options (Figure 4-11).

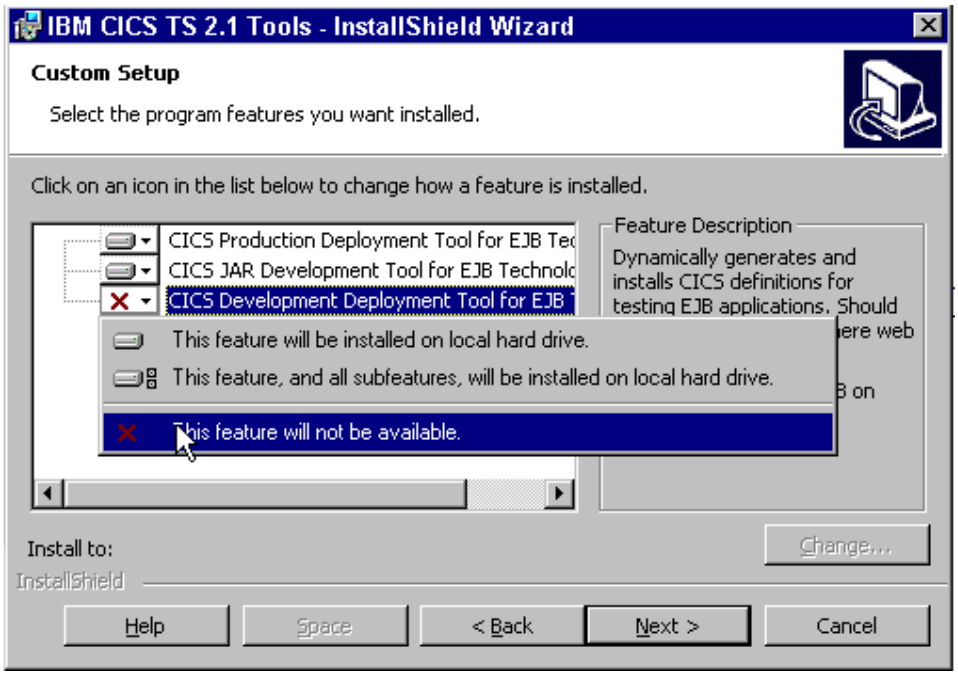


Figure 4-11 Installing the CICS workstation tools

CICS JAR development tool

The *CICS JAR develop tool* provides a useful graphical interface to the *CICS code generation utility*, and as such can be used to easily convert an undeployed JAR file into a deployed JAR file that can be used within a CICS EJB server. It can also be used to edit the deployment descriptor and to add CICS specific information about transaction IDs. For further details on how we used this tool, refer to 6.3.2, “Generating a CICS deployed JAR file” on page 148.

CICS production deployment tool

The *CICS production deployment tool* provides a mechanism for automating the creation of multiple CICS resource definitions when deploying enterprise beans into a CICS region. This is achieved by producing CICS resource definitions that can be used as input to the off-line utility DFHCSDUP or the CICSPlex SM BATCHREP utility. Resource definitions can either be added manually or can be input from the CICS-deployed JAR file output by the CICS development deployment tool.

Tip: If you have just a few enterprise beans and are familiar with CICS systems programming, you may find it easier to deploy enterprise beans *manually* into CICS, by just defining the required resource definitions using CEDA and then manually transferring the deployed JAR file (output from the CICS JAR development tool) to the OS/390 system. For details on how we used this technique, refer to “Deploying manually” on page 150.

4.2.4 CICS development deployment tool

The *CICS development deployment tool* is a servlet based Web application for workstation developers to allow beans developed on a workstation to be deployed into a CICS test environment, without knowledge of OS/390 or CICS system programming.

An HTML form is used to logon to the CICS development deployment tool Web page, where the developer can enter details about the JAR file they wish to deploy. The CICS development deployment tool then automatically transfers the deployed JAR file to the HFS on OS/390, deploys the JAR file to CICS, and creates the necessary CICS definitions through using EXEC CICS CREATE CICS System Programming Interface (SPI) commands. This means a CICS region must be auto-started to retain these definitions, since the definitions will be lost on a cold start.

Tip: The transaction ID definitions created by the CICS JAR development tool will be processed by the CICS development deployment tool to create CICS REQUESTMODEL definitions. These definitions are stored in the `cics-ejb-jar-ext.xmi` file in the deployed JAR file and can be used as input to the CICS production deployment tool in order to create definitions in the CICS CSD.

Note: The CICS development deployment tool can only work with one CICS region at a time, as it can only be configured with one JNDI prefix which references the bean deployed in a specific CorbaServer. Further information on how we used the tool to deploy our HelloWorld bean is provided in 6.3, “Deploying the HelloWorld session bean to CICS” on page 147.

We have already defined the CICS VSAM file (DHFADJM) required by the CICS development deployment tool in the previous sections in this chapter. We now need to define the CICS development deployment tool’s WebSphere components. We made these definitions on the same WebSphere Application Server workstation that we used for our COS Naming server.

Install the CICS tools software

We installed the CICS TS V2.1 tools on to the WebSphere workstation as described on page 85, but this time we took the complete install option to install all of the component tools. By default, the CICS development deployment tool files are installed into the directory `C:\WebSphere\AppServer\hosts\default_host\CICS_EJB`. A diagram of the layout of the directories and files is shown in Figure 4-12.

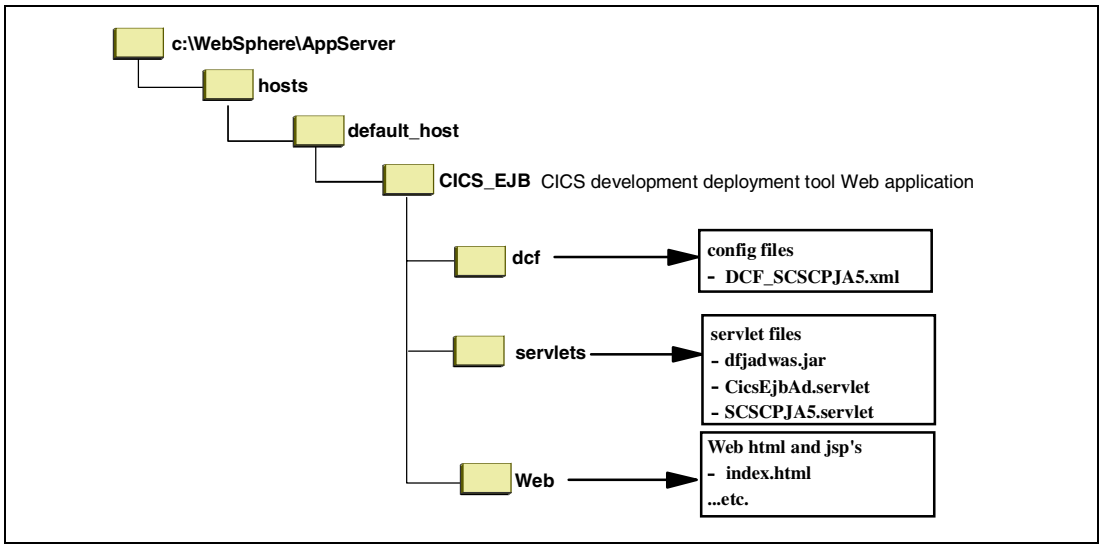


Figure 4-12 CICS development deployment tool directory structure

The steps required to configure the CICS development deployment tool are as follows:

1. Create an XML deployment configuration file (DCF)
2. Define the Web application in WebSphere
3. Create a servlet for the Web application
4. Copy the `.servlet` file
5. Tailor the `index.html` file
6. Test the CICS development deployment tool

Create an XML deployment configuration file

The CICS development deployment tool is controlled by a deployment configuration file (DCF). The DCF is an XML file which defines the Host system, userids, and CORBASERVERS associated with each userid.

DCF sample files are provided with the CICS development deployment tool in the `C:\WebSphere\AppServer\hosts\default_host\CICS_EJB\dcf` subdirectory. We copied the `DCF_sample.xml` file to a new file named `DCF_SCSCPJA5.xml` and then used the Windows Notepad editor to modify it.

Our DCF file is shown in Example 4-9 with the changes we made in bold type.

Example 4-9 Deployment configuration file for the SCSCPJA5 region

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DeploymentConfig SYSTEM "DCF.DTD">
<DeploymentConfig>
  <ConfigDefaults MaxJARSize="1000" LocalJARBase="C:/WebSphere/DJARS"
    AdminContact="ant" MasterTrace="OFF"
    TraceLogPath="C:/WebSphere/AppServer/hosts/default_host/CICS_EJB/logfile.log"
    MaxActionWaitPeriod="600"/>
  <OS390Server DeployJarBase="/u/cicsts21/djars"
    ServerName="wtsc69oe.itso.ibm.com" UserIDIgnoreCase="true" FTPPort="21"
    NamingServiceURL="iiop://hecate.almaden.ibm.com:900/" JNDIPrefix="DFHD"/>
  <CorbaServers>
    <CorbaServer CICSName="PJA5"
      FriendlyName="Shared CORBASERVER on SCSCPJA5 (PJA5)" TransID="5IRP"/>
    <CorbaServer CICSName="ANT"
      FriendlyName="Ants CORBASERVER on SCSCPJA5 (ANT)" TransID="AIRP"/>
    <CorbaServer CICSName="GEOR"
      FriendlyName="Georgs CORBASERVER on SCSCPJA5 (GEOR)" TransID="GIRP"/>
    <CorbaServer CICSName="JOHN"
      FriendlyName="Johns CORBASERVER on SCSCPJA5 (JOHN)" TransID="JIRP"/>
    <CorbaServer CICSName="PHIL"
      FriendlyName="Phils CORBASERVER on SCSCPJA5 (PHIL)" TransID="PIRP"/>
    <CorbaServer CICSName="STEF"
      FriendlyName="Steffens CORBASERVER on SCSCPJA5 (STEF)" TransID="SIRP"/>
  </CorbaServers>
  <Users>
    <User Userid="CICSRS1" Trace="ALL">
        <CorbaServerRef Name="ANT"/>
        <CorbaServerRef Name="PJA5"/>
    </User>
    <User Userid="CICSRS2" Trace="ALL">
        <CorbaServerRef Name="GEOR"/>
        <CorbaServerRef Name="PJA5"/>
    </User>
    <User Userid="CICSRS3" Trace="ALL">
        <CorbaServerRef Name="STEF"/>
```

```
        <CorbaServerRef Name="PJA5"/>
    </User>
    <User Userid="CICSRS4" Trace="ALL">
        <CorbaServerRef Name="JOHN"/>
        <CorbaServerRef Name="PJA5"/>
    </User>
    <User Userid="CICSRS5" Trace="ALL">
        <CorbaServerRef Name="PHIL"/>
        <CorbaServerRef Name="PJA5"/>
    </User>
  </Users>
  <Bindings>
    <ResourceMapping LogicalName="Resource1" Value="jdbc/Resource1"/>
    <ResourceMapping LogicalName="Resource2" Value="jdbc/Resource2"/>
  </Bindings>
</DeploymentConfig>
```

The following are descriptions of the parameters we changed in our DCF file:

- **LocalJARBase**

  This is the path name of the directory used to hold the uploaded JAR files on the WebSphere machine. This directory needs to be defined on the WebSphere machine. We created the directory `C:\WebSphere\DJARS`.

- **DeployJarBase**

  This is the path name of the base directory used to hold the uploaded JAR files on OS/390. This path name is extended by the Web application to include additional directories based on the CorbaServer, user ID and APPLID of the CICS region (this is the `djars` subdirectory from “Defining the CICS HFS directories” on page 69).

- **ServerName**

  This is the hostname of the OS/390 system on which the CICS region is running. This is the OS/390 TCP/IP hostname from Table 4-1 on page 67.

- **NamingServiceURL**

  This is the URL of the COS Naming Server used to look up the location of the beans used by this tool. This must be the Naming Server used when the bean is published from CICS. This is the URL we also defined in our `dfjjvmpr.props` file.

- **CorbaServers**

  This includes details of all the CORBASERVERs into which beans can be deployed. We defined all the CORBASERVER resources that we had defined to CICS in “CORBASERVER” on page 79.

- **Users**

  This contains the list of users able to use the CICS development deployment tool. These user IDs are used to FTP the JAR file from the workstation to OS/390, so the user IDs must also be defined to RACF on OS/390.

Create the Web Application

The WebSphere Admin Server must be started before a Web Application can be created. If the *IBM WS AdminServer* service is not already running it can be started using **Start -> Programs -> IBM WebSphere -> Application Server V3.5 -> Start Admin Server**.

Now the Admin Server GUI can be run with **Start -> Programs -> IBM WebSphere -> Application Server V3.5 -> Administrator's console**. Once the WebSphere Administrator console is open, perform the following steps to create the CICS development deployment tool application:

1. Select **Console -> Tasks -> Create a Web Application**. Enter 'CICS Dev Deployment Tool' as the Web Application Name, select the **Enable JSP .91** radio button, and click **Next**. This is shown in Figure 4-13.

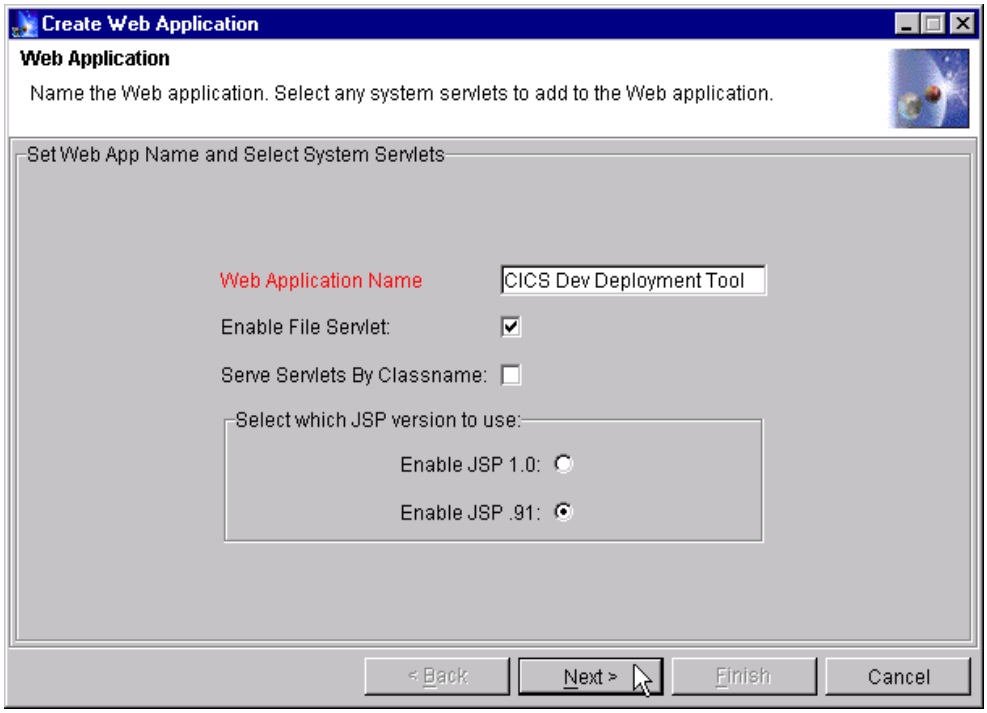


Figure 4-13 Creating the Web application — 1

2. The next window is used to choose the WebSphere servlet engine for the Web application. Expand all the nodes to get to the Default Servlet Engine node, click on this to select it, then click **Next**. This is shown in Figure 4-14.

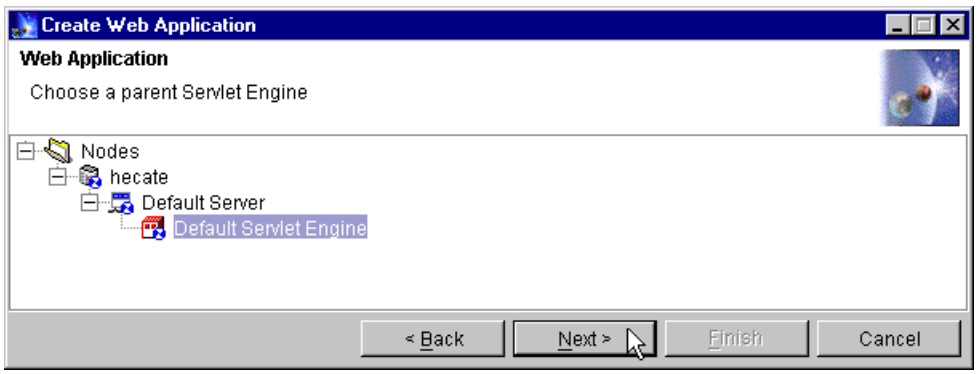


Figure 4-14 Creating the Web application — 2

3. Next, you will name the Web application and specify the path (URL) for invoking it. Change the Web Application Web Path to contain `/CICS_EJB` and click **Next** (Figure 4-15).

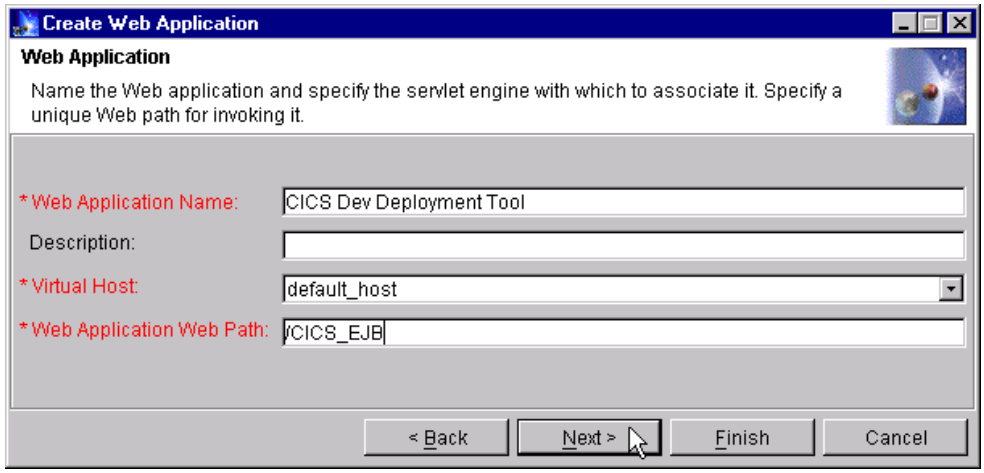


Figure 4-15 Creating the Web application — 3

4. The next window is used to set the Document Root and Classpath. Change the Document Root to contain the following directory:

```
C:\WebSphere\AppServer\hosts\default_host\CICS_EJB\web
```

Also change the Classpath to contain the following five directories:

```
C:\WebSphere\AppServer\hosts\default_host\CICS_EJB\servlets
C:\WebSphere\AppServer\hosts\default_host\CICS_EJB\servlets\dfjadwas.jar
C:\Program Files\IBM\CICS TS 2.1 Tools\Common\xml4j.jar
C:\Program Files\IBM\CICS TS 2.1 Tools\Common\log.jar
C:\Program Files\IBM\CICS TS 2.1 Tools\Common\j2ee.jar
```

Once the Classpath is updated, click **Finish**. This is shown in Figure 4-16.

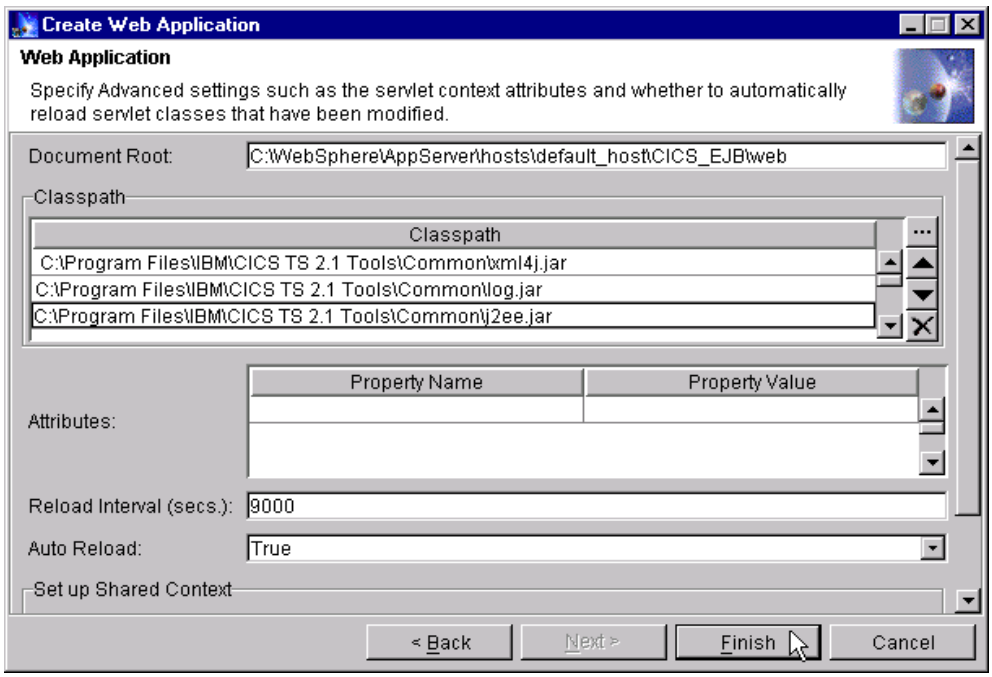


Figure 4-16 Creating the Web application — 4

Creating a servlet for the Application

1. From the WebSphere Administrative console menu bar choose **Console -> Tasks -> Add a Servlet**. In the pop-up window select **Yes** and then click **Next**.
2. The next window asks 'Please select a Web App to contain this servlet'. Expand all the nodes until 'CICS Dev Deployment Tool' is shown, select it, and click **Next**. (Figure 4-17).

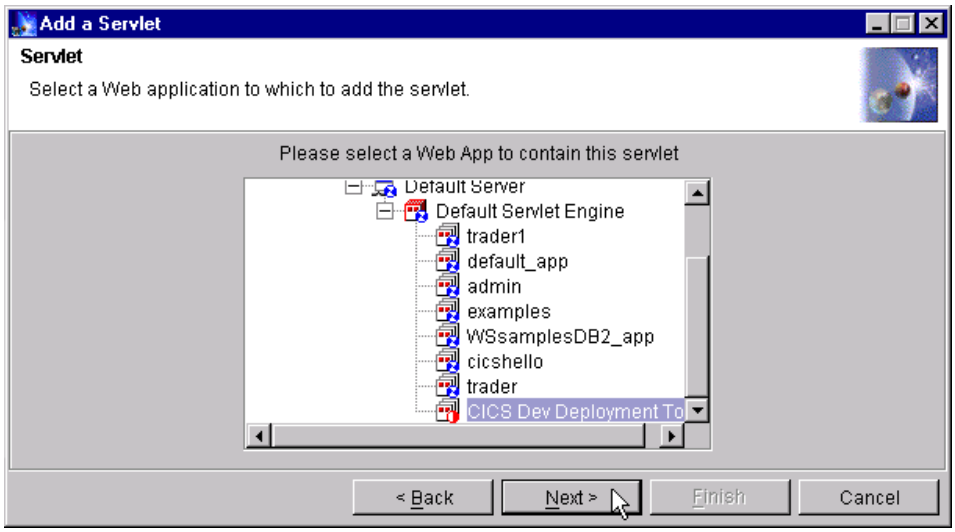


Figure 4-17 Creating the servlet — 1

3. In the next window, click **Browse** to open the file chooser dialog, select the `dfjadwas.jar` file in the directory `C:\WebSphere\AppServer\hosts\default_host\CICS_EJB\servlets`, click **Select** to return to the select jar window, and then click **Next**.
4. The next window is to specify the type of servlet. Choose the **Create User-Defined Servlet** radio button, and then click **Next**.
5. The next window is used to set the servlet name and class. We set the servlet name to the APPLID of our CICS region, SCSCPJA5. The class name should be *com.ibm.cics.addeploy.servlet.CicsEjbAdServlet.* Click **Add**, and in the Add Web Path to Servlet window, change the Servlet Path to be */CICS_EJB/SCSCPJA5.* Now click **OK**, which returns you to the Servlet Name and Class window shown in Figure 4-18.

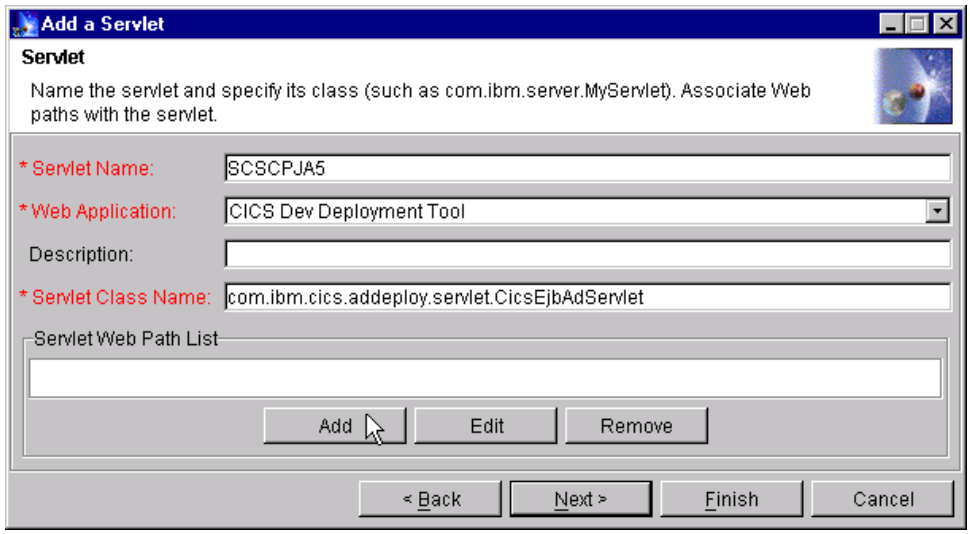


Figure 4-18 Creating the servlet — 2

6. Click **Next** to go to the Servlet Initial parameters window. In the Init Parm Name field, enter `configDefLoc`, and in the Init Parm Value field enter the path to the DCF file, we used the value `C:\WebSphere\AppServer\hosts\default_host\CICS_EJB\dcf\DCF_SCSCPJA5.xml`. Click **Finish**. This is shown in Figure 4-19.

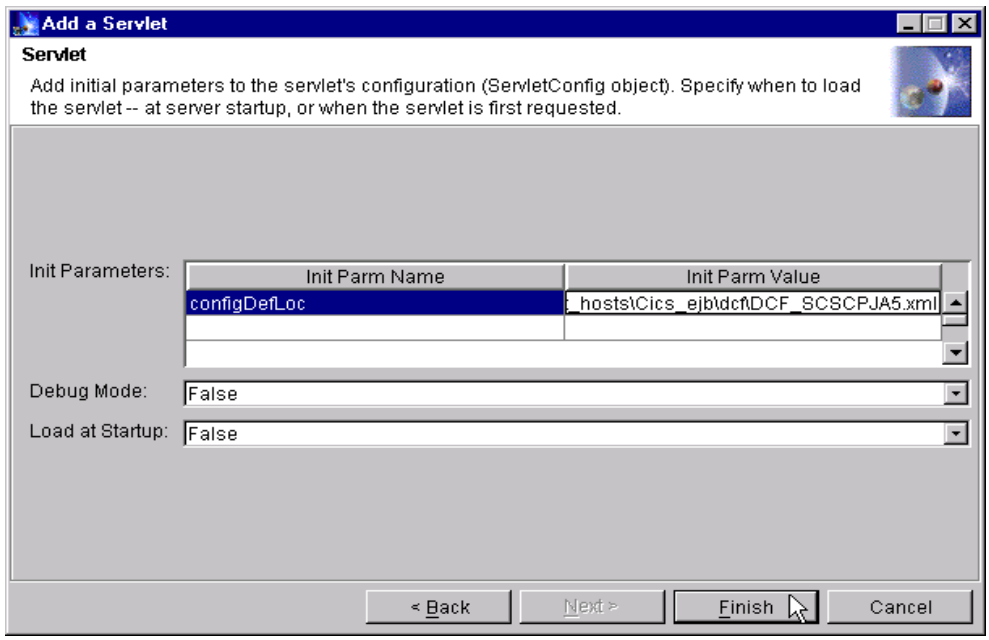


Figure 4-19 Creating the servlet — 3

Copy the servlet file

The servlet we created, SCSCPJA5, must have a .servlet file of the same name in the servlets directory. This .servlet file defines the various components that make up the servlet. A default file is supplied which just needs to be copied. In the directory `C:\WebSphere\AppServer\hosts\default_hosts\CICS_EJB\servlets`, copy the `CicsEjbAd.servlet` file to `SCSCPJA5.servlet` in the same directory.

Tailor the index.html file

The `index.html` file now needs to be tailored to match the servlet name. Use Notepad to edit the file `C:\WebSphere\AppServer\hosts\default_hosts\CICS_EJB\web\index.html` and change the line:

```
<frame src="CicsEjbAd" name="servlet">
```

Change it to be:

```
<frame src="SCSCPJA5" name="servlet">
```

Test the CICS development deployment tool

To start up the application server, in the left panel of the WebSphere Administrator console, right click on Default Server, and select **Start**.

Now, on a Web browser, enter the URL of the WebSphere machine appended with the servlet name. We used `http://hecate.almaden.ibm.com/CICS_EJB`, which displayed the CICS development deployment tool login page, as shown in Figure 4-20.

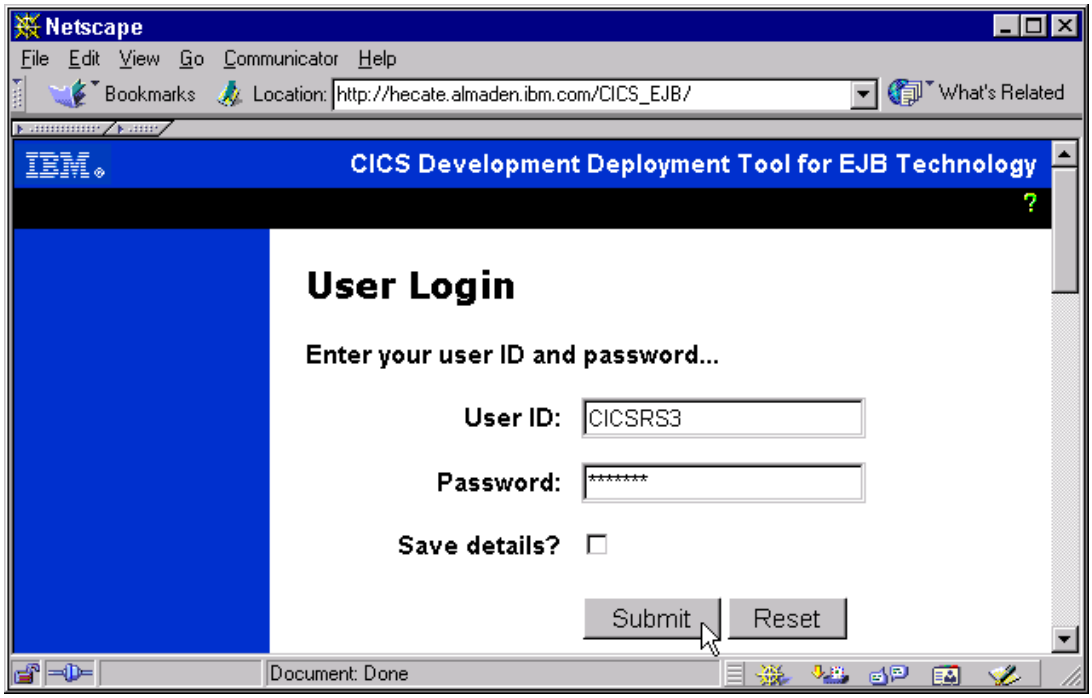


Figure 4-20 CICS development deployment tool login page

More information on how we used the CICS development deployment tool can be found in 6.3, “Deploying the HelloWorld session bean to CICS” on page 147, and in Chapter 17, “Installing and configuring CICS deployment tools for EJB technology”, *Java applications in CICS*, SC34-5881. For help with common problems, refer to “CICS development deployment tool application problems” on page 124.

4.3 Installation verification

CICS provides an Installation Verification Program (IVP) to test the CICS TS V2.1 EJB server configuration. This is a simple *HelloWorld* enterprise bean that returns the string that a client passes it. This is provided as a set of ready to use JAR files with no compiling or editing of deployment descriptors required. This makes it very easy to install and run even for someone without Java programing skills.

There are two clients provided for use with the CICS HelloWorld enterprise bean sample:

1. A stand-alone Java application that runs from the OS/390 USS command prompt
2. A Web application that runs as a servlet initiated from a Web browser

The stand-alone client is the easiest to set up and use, but it does not use WebSphere Application Server or verify that the COS Naming Server is working. The Web application client provides more complete function verification, but as it requires setting up a servlet application in WebSphere, it is more time-consuming to configure.

4.3.1 Running the IVP OS/390 USS client application

For someone with OS/390 skills, probably the simplest way to test the CICS EJB server is to to run the IVP test client (runEJBIVP) from the OS/390 USS environment. This allows you to call the HelloWorld enterprise bean in CICS without using the COS Naming Server, as the bean is published to the CICS *shelf*. To run the client, perform the following steps:

1. Modify the CICS JVM profile member
2. Define the CICS resources
3. Modify the USS runEJBIVP script
4. Run the runEJBIVP script

To publish an enterprise bean to the CICS shelf rather than the COS Naming Server requires special parameters to be set in the JVM system properties file. A file named `IVP.properties` is supplied for this purpose, in the CICS samples directory, `/usr/lpp/cicsts/cicsts21/samples/ejb/helloworld.`

When using this system properties file, publishing an enterprise bean causes the IOR to be written to the CICS shelf directory (which is an OS/390 HFS directory) instead of the COS Naming Server. The USS client program can then access the IOR in the shelf, bypassing the COS Naming Server.

To use this system properties file requires the default JVM profile member to be changed to point to this system properties file. In “DFHJVM and JVM profiles” on page 73, we described how to create a DFHJVMPR JVM profile. We changed the JVMPROPS parameter of this member to point at the HelloWorld system properties file, which is:

```
JVMPROPS=/usr/lpp/cicsts/cicsts21/samples/ejb/helloworld/IVP.properties
```

In order to pick up the change to JVMPROFPS, we then reset all the CICS JVMs using the command **`CEMT SET JVM PHASEOUT`**.

Define the CICS resources

The HelloWorld sample requires TCPIPSERVICE, CORBASERVER, and DJAR resource definitions. Samples are supplied in group DFH$EJB. We copied this group to our own group with the command:

```
CEDA COPY GROUP(DFH$EJB) TO(IVP$EJB)
```

We then made the following changes to the resources:

- **CORBASERVER**

```
SHelf ==> /u/cicsts21/shelf
Host ==> wtsc69oe.itso.ibm.com
Port ==> 10555
```

- **DJAR**

```
Hfsfile ==> /usr/lpp/cicsts/cicsts21/samples/ejb/helloworld/HelloWorld
        ==> EJB.jar
```

- **TCPIPSERVICE**

```
POrtnumber ==> 10555
STatus     ==> Open
```

We then installed the group and published the enterprise bean to the HFS shelf with the following commands:

```
CEDA INS GROUP(IVP$EJB)
CEMT PERFORM DJAR(HELLO) PUBLISH
```

Modify the USS runEJBIVP script

CICS provides a USS shell script called `runEJBIVP` to automate running the USS HelloWorld client. This needs tailoring to define the path of the Java install, and the path to the CICS shelf.

We used the TSO ISHELL command to locate and edit the runEJBIVP. This script can be found in `/usr/lpp/cicsts/cicsts21/samples/ejb/helloworld/runEJBIVP`. The relevant modifications to this script are highlighted in Example 4-10.

Example 4-10 modified runEJBIVP script

```
# ---------------------------------------------------------------------
# CICS EJB IVP run script
#
# Modify the following to match your IBM SDK 1.3 installation directory:
JAVA_HOME=/usr/lpp/java130s/J1.3
#
# Modify the following to match your CICS TS 2.1 installation directory:
CICS_HOME=/usr/lpp/cicsts/cicsts21
#
# Modify the following to match your CICS region shelf directory, region
# name and sample CORBASERVER name:
CICS_SHELF=/u/cicsts21/shelf/SCSCPJA5/EJB1
#
#----------------------------------------------------------------------
#
CLASSPATH=./HelloWorldCLI.jar:$JAVA_HOME/standard/ejb/1_1/ejb11.jar
#
#
echo "CICS EJB IVP: Querying the Java SDK level"
if $JAVA_HOME/bin/java -version
then
else echo "CICS EJB IVP: Failed, possible cause:"
     echo "    Java support not found at $JAVA_HOME"
     echo "    Check the JAVA_HOME setting in the runEJBIVP script"
     exit
fi
#
echo ""
echo "CICS EJB IVP: Starting the EJB client program"
if $JAVA_HOME/bin/java -classpath $CLASSPATH  HelloWorldIVP $CICS_SHELF
then echo "CICS EJB IVP: Completed successfully"
else echo "CICS EJB IVP: Failed"
     echo "    Check the JAVA_HOME, CICS_SHELF and CLASSPATH settings in the runEJBIVP"
     echo "    script, and the CICS server installation steps of the EJB IVP."
fi
```

Attention: Note that this script will be overwritten if the CICS install is refreshed. In order to preserve the customization, it should be copied to a different user directory before any changes are made.

Executing the runEJBIVP script

We started a USS shell with the TSO OMVS command, changed to the CICS samples directory, and ran the **runEJBIVP** script; the output is shown in Example 4-3.

Example 4-11 runEJBIVP script output

```
CICSRS5@SC69:/u/cicsrs5>cd /usr/lpp/cicsts/cicsts21/samples/ejb/helloworld
CICSRS5@SC69:/usr/lpp/cicsts/cicsts21/samples/ejb/helloworld>runEJBIVP
CICS EJB IVP: Querying the Java SDK level
java version "1.3.0"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.0)
Classic VM (build 1.3.0, J2RE 1.3.0 IBM OS/390 Persistent Reusable VM build
CICS EJB IVP: Starting the EJB client program
HelloWorld client program started
Located home interface for HelloWorld bean
You said: Hello from CICS EJB IVP client
HelloWorld client program ended
CICS EJB IVP: Completed successfully
CICSRS5 @ SC69:/usr/lpp/cicsts/cicsts21/samples/ejb/helloworld>
```

4.3.2 The HelloWorld Web application

In this section we describe how we used the CICS supplied HelloWorld sample Web application to verify both the CICS EJB Server and the WebSphere COS Naming Server.

The steps we used to install and run this sample were these:

1. FTP the sample files from OS/390 to the WebSphere workstation.
2. Copy the sample files into the correct WebSphere directories.
3. Define the sample Web Application to WebSphere.
4. Create a servlet for the Web application.
5. Install the required CICS resource definitions.
6. Test using by invoking the configured servlet from a Web browser.

Tip: Further details on running the HelloWorld Web application sample can be found in the HFS file `/usr/lpp/cicsts/cicsts21/samples/ejb/helloworld/readme.txt`

FTP the sample files from OS/390 to the WebSphere workstation

CICS supplies three JAR files for the HelloWorld sample in the CICS samples subdirectory. We used the Windows NT FTP program to copy these files to our WebSphere workstation. This is shown in Example 4-12.

Example 4-12 FTP the HelloWorld sample jar files to the workstation

```
C:\TEMP>mkdir HelloWorld
C:\TEMP>cd helloWorld
C:\TEMP\HelloWorld>ftp wtsc69oe.itso.ibm.com
Connected to wtsc69oe.itso.ibm.com.
220-FTPDOE1 IBM FTP CS V2R10 at wtsc69oe.itso.ibm.com, 00:34:34 on 2001-03-16.
220 Connection will not timeout.
User (wtsc69oe.itso.ibm.com:(none)): cicsrs1
331 Send password please.
230 CICSRS1 is logged on.  Working directory is "/u/cicsrs1".
ftp> cd /usr/lpp/cicsts/cicsts21/samples/ejb/helloworld
250 HFS directory /usr/lpp/cicsts/cicsts21/samples/ejb/helloworld is the current working
directory
```

```
ftp> bin
200 Representation type is Image
ftp> get HelloWorldCLI.jar
200 Port request OK.
125 Sending data set /usr/lpp/cicsts/cicsts21/samples/ejb/helloworld/HelloWorldCLI.jar
250 Transfer completed successfully.
9093 bytes received in 0.36 seconds (25.19 Kbytes/sec)
ftp> get HelloWorldEJB.jar
200 Port request OK.
125 Sending data set /usr/lpp/cicsts/cicsts21/samples/ejb/helloworld/HelloWorldEJB.jar
250 Transfer completed successfully.
34844 bytes received in 0.75 seconds (46.40 Kbytes/sec)
ftp> get HelloWorldWeb.jar
200 Port request OK.
125 Sending data set /usr/lpp/cicsts/cicsts21/samples/ejb/helloworld/HelloWorldWeb.jar
250 Transfer completed successfully.
5887 bytes received in 0.23 seconds (25.60 Kbytes/sec)
ftp> bye
221 Quit command received. Goodbye.
C:\TEMP\HelloWorld>
```

Copy the sample files into the correct WebSphere directories

Within the `C:\WebSphere\AppServer\hosts\default_host` subdirectory we created a directory call `cicshello`, and extracted the files from the `HelloWorldWeb.jar` file into this directory. We used Winzip to extract the files and this created two subdirectories within the `cicshello` directory called `web`, and `servlets`. The servlets subdirectory contains further subdirectories for the servlet classes.

We then also copied the `HelloWorldCLI.jar` file from OS/390 into the `cicshello\servlets` subdirectory on our Windows NT machine. The servlet also uses the `j2ee.jar` which contains classes required for EJB support. This is installed on the workstation as part of the CICS tools installation. We copied it from the tools install directory `C:\Program Files\IBM\CICS TS V2.1 Tools\Common` into the `cicshello\sevlets` subdirectory.
The resulting directory structure and file locations are shown in the diagram in Figure 4-21.

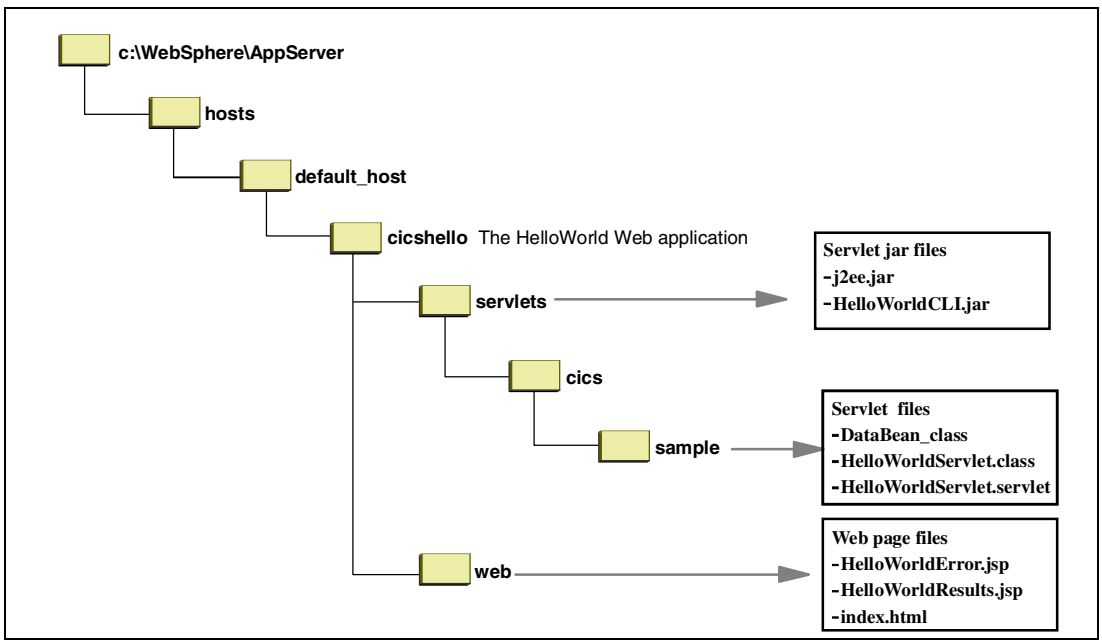


Figure 4-21 The HelloWorld sample directory structure

Define the sample Web Application to WebSphere

We now created a Web application and servlet within WebSphere Application Server for Windows NT. This is done in a manner similar to the Web application defined for the CICS development deployment tool in 4.2.4, “CICS development deployment tool” on page 86.

We started the WebSphere Admin Server GUI with **Start -> Programs -> IBM WebSphere -> Application Server V3.5 -> Administrator’s console**. Once the WebSphere Advanced Administrator console opened, we defined a Web application and then added a servlet to it with the following steps:

Create a cicshello web application

1. From the WebSphere Advanced Administrator Console menu bar, select **Console -> Tasks -> Create a Web Application**. This opens the Create Web Application window. Enter *cicshello* as the Web Application Name, select **Enable File Servlet**, **Serve Servlets By Classname**, and **Enable JSP 1.0**. Click **Next**. This is shown in Figure 4-22.

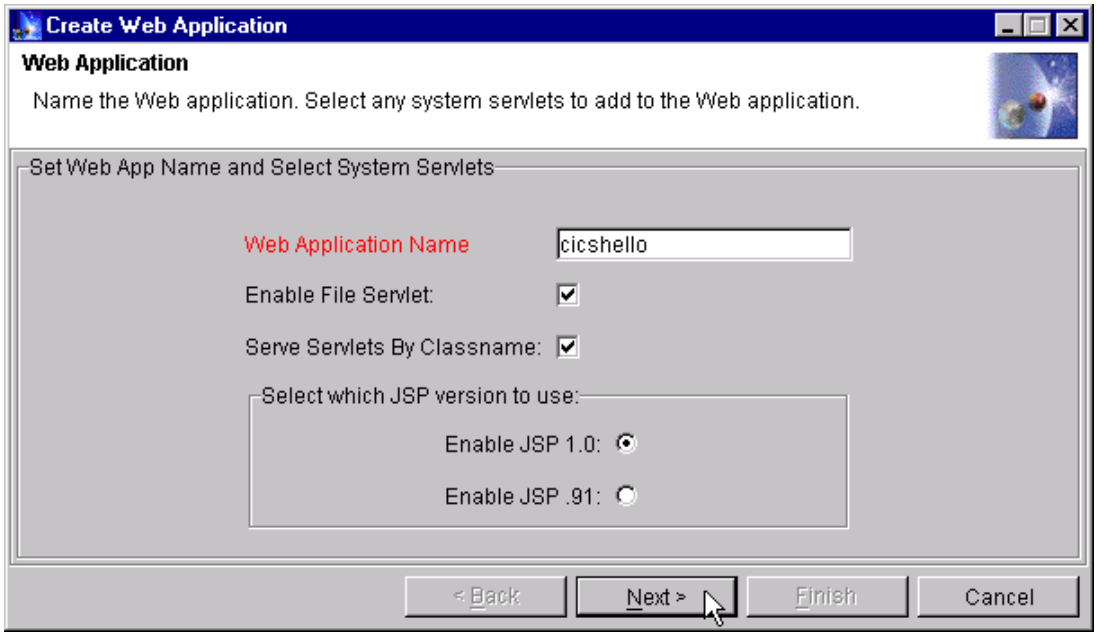


Figure 4-22 Creating the HelloWorld Web Application — 1

2. In the Choose a Parent Servlet Engine window, expand the tree of Nodes until *Default Servlet Engine* is reached. Select this, and click **Next**.
3. Next is the Name the Web Application window. Change the *Web Application Web Path* to `/cicshello`, and click **Finish**. This is shown in Figure 4-23.

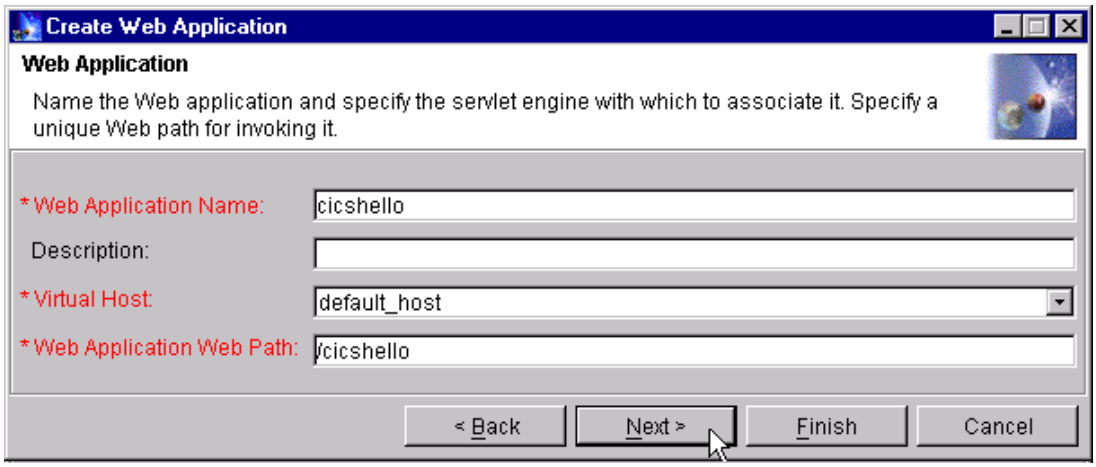


Figure 4-23 Creating the HelloWorld Web Application — 2

A dialog box is now displayed saying that “Command WebApplication.create completed successfully”. The next step is now to define a servlet for the Web application.

Create a servlet for the Web application

1. In the left of the WebSphere Advanced Administrator console expand the tree of nodes to show the new cicshello application beneath the Default Servlet Engine node. Right-click on *cicshello*, and select **Create -> Servlet**.
2. In the Create Servlet window that opens, set the Servlet Name field to Hello, and the Servlet Class Name to *cics.sample.HelloWorldServlet*. Click **Add**.
3. In the Add Web Path to Servlet, set the Servlet Path to `/cicshello/Hello`. Click **OK**. The resulting window is shown in Figure 4-24.

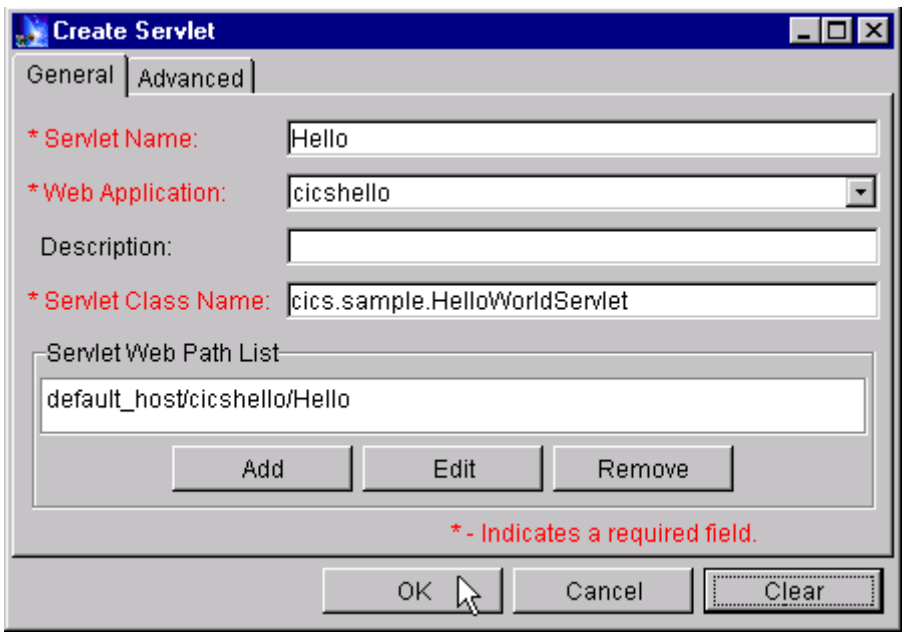


Figure 4-24 Creating the HelloWorld servlet

A dialog box is displayed saying that the “Command Servlet.create completed successfully”. The Web application is now complete. To enable it for use, in the left of the WebSphere Administrator console right click on cicshello and select **Restart Web App.**

Install the required CICS resource definitions

The CICS resource definitions we defined in earlier in “Define the CICS resources” on page 94 can be used in this scenario with no changes. However, since we had previously changed the JVM profile in “Running the IVP OS/390 USS client application” on page 94 we first undid that change by setting JVMPROPS parameter back to the following value:

```
JVMPROPS=/u/cicsts21/props/dfjjvmpr.props
```

Note: After changing the JVMPROPS parameter, we recommended that all the JVMs are re-initialized in order to be sure that the new JVM properties file is used. This can be achieved using the command **CEMT SET JVM PHASEOUT**.

Test the cicshello sample

To test the HelloWorld Web application, on our Web browser we entered the URL, `http://hecate.almaden.ibm.com/cicshello` and were presented with the HTML form shown in Figure 4-25.

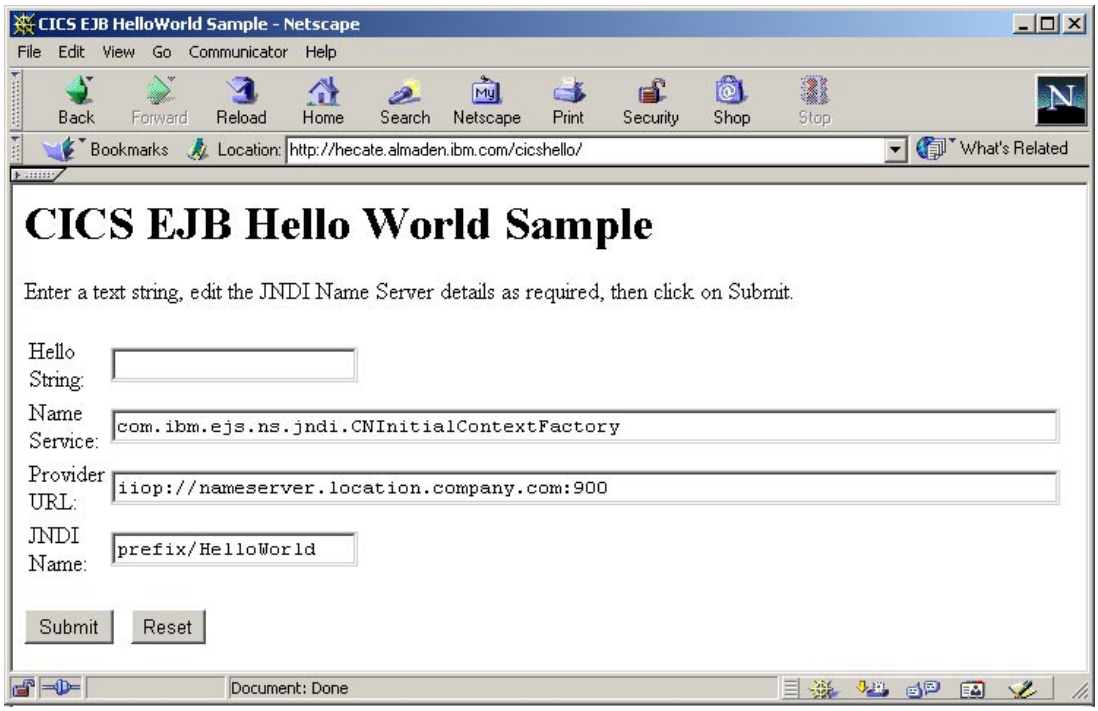


Figure 4-25 HelloWorld Web application, initial screen

We entered the following parameters:

Hello String	Hello CICS
Provider URL	`iiop://hecate.almaden.ibm.com:900`
JNDI Name	HelloWorld

We left the Name Service parameter to default and clicked **Submit**. The HelloWorld enterprise bean then replied with the output shown in Figure 4-26.

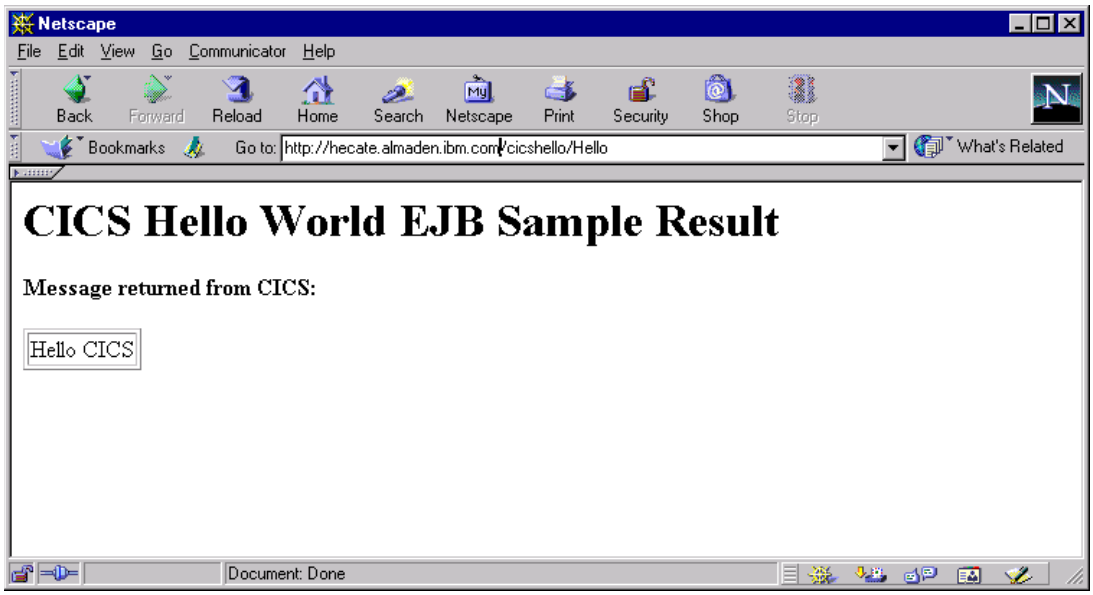


Figure 4-26 HelloWorld Web application, output

Installation completed

We have now completed the configuration and verification of the EJB Server supplied with CICS TS V2.1.

For further information on trouble-shooting errors within CICS TS V2.1, refer to Chapter 5, “Troubleshooting enterprise beans in CICS TS V2.1” on page 103.

For further practical examples on using the CICS JAR development tool, the CICS production deployment tool, and the CICS development deployment tool refer to Chapter 6, “Developing a HelloWorld session bean for CICS” on page 135.

Following on from this, the chapters in Part 3,on page 133 describe details on how to write and deploy various enterprise beans for use within CICS.

5

Troubleshooting enterprise beans in CICS TS V2.1

This chapter describes the various options and tools available for diagnosing problems with an application which uses enterprise beans in CICS. We first give an overview of the various tools and facilities available for capturing diagnostic information at various points. This is followed by some debugging scenarios using the sample applications developed in this book.

Compared with debugging a regular 3270 CICS COBOL application, debugging a Web application using enterprise beans in CICS can seem complicated. When something goes wrong there are several systems in the path from the browser to the enterprise bean which can make it difficult to know where to look to find the problem. Knowing where to look for diagnostic information can make all the difference.

This chapter does not give in-depth information on how to debug all the systems in the path of a Web application. Rather, we attempt to give enough information on where to find diagnostic information to get the sample programs described in this book running and overcome some of the more common problems.

5.1 Diagnosing Java problems in CICS

This section covers what options are available for collecting information and debugging problems relating to the execution of Java code within CICS.

First we explain where diagnostic information is output by the JVM and how to control what information is recorded.

We then describe the Java Platform Debugger Architecture and discuss how to use this to debug a Java program running in CICS.

5.1.1 Gathering diagnostic information

The JVM records information in several places which are controlled by options in the JVM profile and the system properties file. The following are options that we found useful while we developed the applications in this redbook.

For more information about these and other JVM parameters, see Chapter 18, “Defining the JVM initialization options” in the *CICS System Definition Guide*, SC34-5725.

The Java stdin/stdout/stderr files

These standard Java files are in located in the directory specified by the WORK_DIR parameter of the JVM profile. For our installation, they were written to the directory `/u/cicsts21/work/SCSCPJA5` which we specified as our WORK_DIR in our JVMProfile.

When a Java program calls one of the System.out class print methods, such as `System.out.println()`, the output is written to a `stdout` file. This file will be located in the directory specified by the WORK_DIR parameter in the JVM profile. This is one of the most common ways of debugging a Java program, and we used it frequently while developing the sample programs in this book. A new `stderr` file is created each time a JVM is initialized. These are unique from other files, since the timestamp is appended to the file name. This makes it easy to locate the most recent file, as it will be at the bottom of a TSO ISHELL directory list. Note, however, that since JVMs can be reused this means the files can contain information relating to the execution of a previous program.

Any un-caught exceptions that are thrown while executing a Java program cause a stack trace to be written to the `stderr` file located in the WORK_DIR directory. This is one of the first places we checked when we had unexpected problems with a program. The stack trace provides an easy way to see which class and method is having a problem.

Note: To have the next execution of a Java program or enterprise bean create new `stdin/stderr` files, the command **`CEMT SET JVM PHASEOUT`** can be used. This marks all JVMs for deletion, causing the next request to initialize a new JVM. Also, Chapter 21 of the *CICS Customization Guide*, SC34-5706 describes how to use the user-replaceable program DFHJVMAT to create unique `stdout` and `stderr` files by appending the task number to the files name.

The system properties event logging file

The system properties file specifies where to write event logging information with the `ibm.jvm.events.output` parameter. Setting this system property enables event logging in the JVM.

It defines whether the text records describing the event are stored in a file described by its full path name, or whether the events are logged in the `stderr` or `stdout` files. Our system properties file is described in “JVM system properties files” on page 69, and in we use the value:

```
ibm.jvm.events.output=ure.log
```

This creates the file named `ure.log` in our work directory, which is defined by the WORK_DIR parameter, and on our system is at `/u/cicsts21/work/SCSCPJA5`.

The `ure.log` file is written to when a JVM is initialized. It contains useful information about the parameters being used for the JVM initialization, such as the Java build level, the class and lib paths, and the trusted middleware class path that is generated by CICS.

We used this information to verify that the JVM was using the profiles and system properties file we expected and that the various directory paths were being built correctly.

The JVM profile VERBOSE parameter

This parameter indicates whether or not the JVM should issue messages containing information about its activities. The information gets written to the Java `stderr` file.

Our JVM profile is described in “DFHJVM and JVM profiles” on page 73, and has VERBOSE set to NO because requesting this information can produce a lot of output.

We did occasionally use VERBOSE=class to aid with diagnosing some problems. This causes messages to be written as each class is loaded and initialized and any error information such as *class not found* messages. It can sometimes help with locating precisely where a problem is occurring.

The system properties JVM trace

The `ibm.dg.trc.external` parameter of the system property file enables the internal JVM trace facility to aid in the diagnosis of problems within the Java Virtual Machine itself.

Our system properties file is described in “JVM system properties files” on page 69, and it has the `ibm.dg.trc.external` parameter commented out. This is because JVM tracing should only be used under the direction of IBM support personnel as it can have a major impact on performance and produce huge amounts of output.

5.1.2 The Java Platform Debugger Architecture

In this section we describe the Java Platform Debugger Architecture (JPDA). We first give an overview of what JPDA is and how it works, followed by instruction on how to configure CICS for the JPDA. We then describe how to debug an enterprise bean using a remote JPDA client.

Overview of JPDA

JPDA is the standard debugging mechanism of the Java 2 Platform. It enables a debugger client program on a remote workstation to control the execution of a client application’s Java classes in the CICS JVM. Breakpoints can be set at individual method calls or line numbers within the code, and variables can be displayed or altered. This all happens in real time, as the CICS JVM is actually executing the Java byte code.

A diagram of how this works is shown in Figure 5-1.

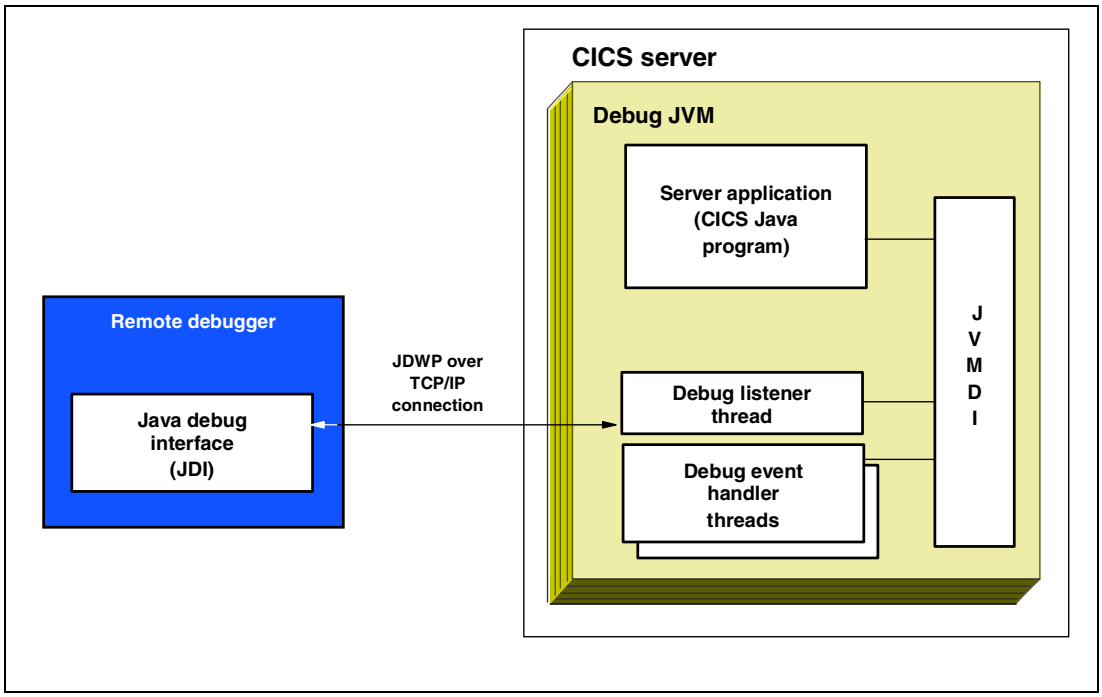


Figure 5-1 Overview of the CICS JVM and JPDA

- An incoming request starts a request processor alias transaction. That alias transaction specifies a request processor PROGRAM with a JVMPROFILE attribute that has special debug options causing a Debug JVM to start.
- The JVM starts debug threads to manage the debugging session. These threads use the Java Virtual Machine Debug Interface (JVMDI) to communicate with the JVM and control the execution of the application threads.
- A remote client uses the Java Debug Interface (JDI) to communicate with the debugging threads within the JVM. The actual communication between the debugger client and the remote JVM uses the Java Debug Wire Protocol (JDWP).

The examples we used in this section are for debugging with the Trader application. They should also apply to debugging with any other enterprise bean; only the names of the JARs, beans, and methods need to be changed.

Configuring CICS for JPDA

Configuring CICS to use JPDA requires three things:

- A JVM profile specifying the correct debug options
- A request processor transaction alias that uses the debugging JVM profile
- A REQUESTMODEL resource definition that maps incoming request to use this transaction.

JVM profile debugging options

A JVM profile needs to be created that specifies various Java options to enable JPDA. One of these parameters specifies the TCP/IP port number that CICS uses to communicate with the remote debugger client. Because only one client can connect to this port at a time, as many profiles are needed as there are developers using JPDA. Each of these profiles must specify a different port number.

In “DFHJVM and JVM profiles” on page 73 we described how we used the PDS data set CICSSYSF.CICS610.DFHJVM to store our JVM profiles and by default used the profile member name DFHJVMPR. We now copied this default member to a member with a new name for each of us wanting to use JPDA, for example ANTJVMDB, and then set the various debug options in the new member.

An extract of one of our debugging profiles is shown in Example 5-1, with the changes we made to enable debugging in bold.

Example 5-1 Extract of our JVM profile used for JPDA debugging

```
# ********* Java non-standard options **********
Xdebug=YES
Xnoagent=YES
Xresettable=NO
Xrunjdwp=(transport=dt_socket,server=y,address=10177)
Xcheck=NO
Xms=1M
Xmx=192M
Xnoclassgc=NO
Xoss=4M
Xss=512K
Xverify=none
```

Each of these parameters is now described:

- **Xdebug=YES** specifies that the JVM should be started in debug mode
- **Xnoagent=YES** disables Java 1.1 compatibility which is not available when the JVM is in debug mode
- **Xresettable=NO** ensures that the JVM is discarded after use, since JVMs which have been used in debug mode are not eligible for reuse.
- **Xrunjdwp=(transport=dt_socket,server=y,address=10177)** defines the protocol to communicate with the debugger client (transport=dtsocket), if CICS initiates the communication (server=n) or waits for the debugger client to connect (server=y), and the TCP/IP port (address=port) that the remote debugger must user for this connection.

Note: The Xrunjdwp parameter we specified in the profile causes CICS to stop after initializing the JVM and wait for the remote debugger client to initiate a debugging session.

CICS definitions for using JPDA

In “Request processor aliases” on page 81 we described how to define aliases for the request processor transaction, CIRP, and the request processor program, DFJIIRP. We now copied these TRANSACTION and PROGRAM definitions to use as an alias transaction which uses a debugging JVM profile.

As with the debugging JVM profile, there needs to be a separate request processor alias transaction and program for each developer using JPDA.

We created these aliases by copying the supplied CIRP and DFJIIRP resource definitions with the commands:

```
CEDA COPY PROG(DFJIIRP) GROUP(DFHIIOP) TO(PJA5IIOP) AS(ANTDIIRP)
CEDA COPY TRANS(CIRP)   GROUP(DFHIIOP) TO(PJA5IIOP) AS(ADRP)
```

The only change to these definitions is to set the PROGRAM attribute of the new transaction definition, ADRP, to use the new program ANTDIIRP, and to set the JVMPROFILE attribute of the new program ANTDIIRP to use the new JVM profile, ANTJVMDB.

To use these new request processor alias transactions requires a REQUESTMODEL resource definition to map incoming requests to the new transactions. This can be done automatically by the CICS development deployment tool, or they could be defined manually as described in “REQUESTMODEL” on page 80.

Using the CICS development deployment tool with JPDA

The CICS development deployment tool will automatically generate a REQUESTMODEL resource definition when deploying a JAR file. By changing the DCF configuration file, these can be used to map a request to a debugging request processor transaction.

The changes required are to add CORBASERVER statements to the DCF configuration file specifying the new transactions. The DCF configuration file is described in 4.2.4, “CICS development deployment tool” on page 86. An extract of our configuration file with these changes in bold, is shown in Example 5-2.

Note: The only difference between the new CorbaServers definition for debugging, and the existing CorbaServers definition, is the TransID parameter. As both definitions have the same CICSName parameter, any User definition that already specifies this name in its CorbaServerRef statement will pick up both definitions.

Example 5-2 DCF configuration file extract showing JPDA debugging definitions

```
...
  <CorbaServers>
    <CorbaServer CICSName="PJA5"
      FriendlyName="SCSCPJA5 (PJA5)" TransID="5IRP"/>
    <CorbaServer CICSName="PJA5"
      FriendlyName="Debugging SCSCPJA5 (PJA5)" TransID="5DRP"/>
    <CorbaServer CICSName="ANT"
      FriendlyName="SCSCPJA5 (ANT)" TransID="AIRP"/>
    <CorbaServer CICSName="ANT"
      FriendlyName="Debugging SCSCPJA5 (ANT)" TransID="ADRP"/>
  </CorbaServers>
...
```

When not using the CICS development deployment tool we manually created REQUESTMODEL definitions for debugging. For example, to use JPDA to debug the Trader sample program, we used the command:

```
CEDA DEFINE REQ(TRADDBG) GROUP(TRADDBG)
```

This is shown in Figure 5-2 with the attributes we added in bold.

```
CEDA DEF REQ(TRADDBG) GR(TRADDBG)
  OVERTYPE TO MODIFY                                      CICS RELEASE = 0610
   CEDA  ALter Requestmodel( TRADDBG    )
    Requestmodel   : TRADDBG
    Group          : TRADDBG
    Description  ==>
    Corbaserver  ==> PJA5
    TYpe         ==> Ejb               Corba | Ejb | Generic
   EJB PARAMETERS
    Beanname     ==> Trader
    INTFacetype  ==> Remote            Both | Home | Remote
COMMON PARAMETERS
    OPeration    ==> logon*
                 ==>
   TRANSACTION ATTRIBUTES
    TRansid      ==> ADRP

SYSID=PJA5 APPLID=SCSCPJA5
PF 1 HELP 2 COM 3 END            6 CRSR 7 SBH 8 SFH 9 MSG 10 SB 11 SF 12 CNCL
```

Figure 5-2 REQUESTMODEL for using JPDA with the Trader bean

The description of these parameters is as follows:

- **Corbaserver** specifies which CORBASERVER resource definition this request model applies to. Our Trader bean DJAR is defined in the PJA5 CORBASERVER, so that is the name we define here.
- **TYpe** specifies which request types this request model applies to. We only used EJB requests, so that is what we defined here.
- **Beanname** is the name of the bean that this request applies to. We want to debug the Trader sample bean, so this is the name we use here.
- **OPeration** defines the method requests that this request model applies to; an asterisk(*) can be used alone or with a prefix for generic method names. In this example we use a generic pattern matching all methods starting with logon. If a specific method name is used we also recommend appending it with an asterisk(*) to avoid problems with name-mangling. Note also that for stateful session beans, the REQUESTMODEL is only used for the first request in an OTS transaction.
- **TRansid** specifies the name of the transaction to be used when an incoming request matches all the previously defined parameters. This is the name of a request processor transaction we just defined that uses a debugging JVM profile.

After these changes are complete, CICS is ready to use a remote JPDA debugger client to debug an enterprise bean running within CICS.

Debugging an enterprise bean using JPDA

This section describes how to use JPDA to debug an enterprise bean running within CICS. The steps necessary to debug the Java program are as follows:

- Compile the Java program to be debugged with the debug option.
- Deploy the enterprise bean JAR file to CICS specifying a request processor program which has a JVM profile with the special debug options.
- Make the Java source code of the class to be debugged available to the debugger.
- Use a JPDA debugger client to connect to CICS and debug the program.

Compiling programs with the debug option

The Java compile debug option tells the compiler to store extra symbol table information within the class file it produces. Without this extra information, the class can still be debugged using JPDA, but some things are not available to the remote debugging client, such as access to the classes local variables.

When using the command line Java compiler, the debug option is specified by the `-g` parameter. For example:

```
javac -g HelloWorld.java
```

If using VisualAge for Java, the debug option is specified when exporting the class files. This can be set by using the VAJ *Export SmartGuide* **FILE -> Export** and selecting **Include debug attributes in .class file**. This is shown in Figure 5-3.

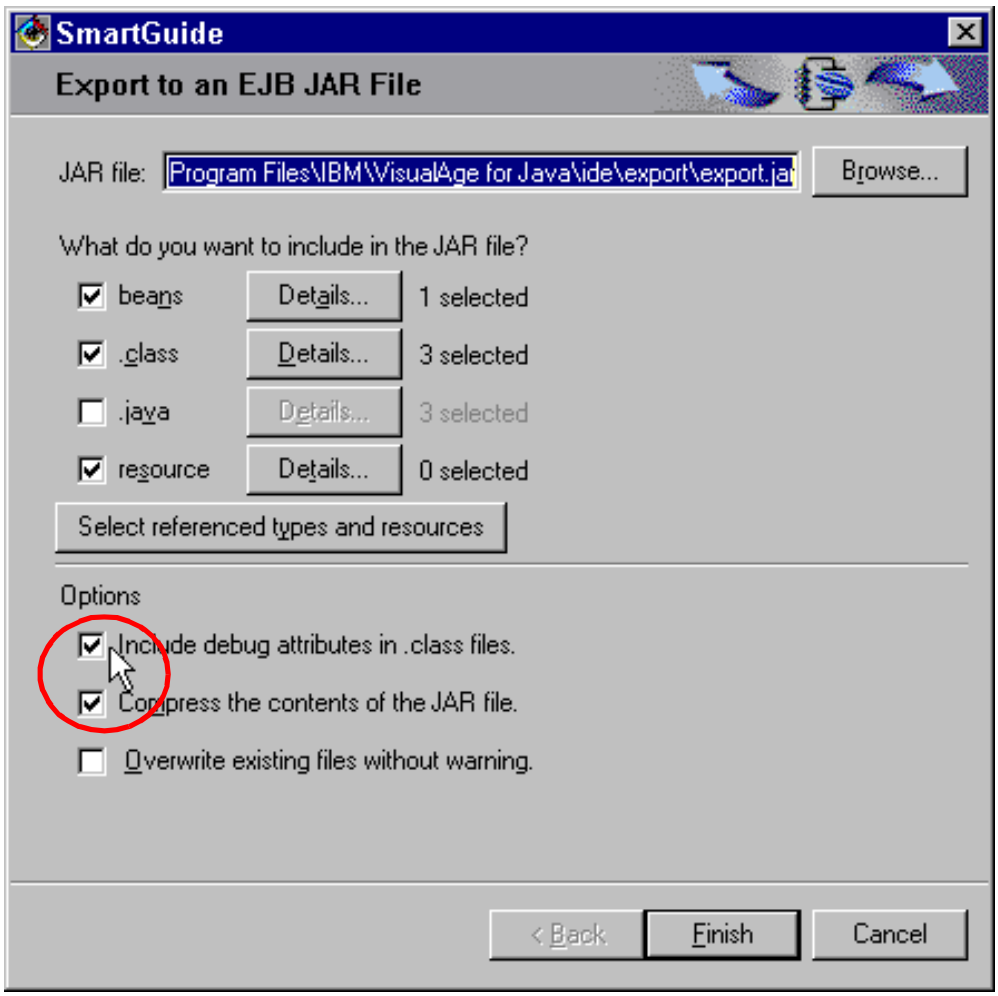


Figure 5-3 Exporting classes from VisualAge with the debug attributes

Deploying a JAR file to be debugged in CICS

We deployed our JAR files for debugging in CICS using the CICS development deployment tool that we set up in “Configuring CICS for JPDA” on page 106. All that is required is to logon to the tool, specify the JAR file name, and select the debugging CORBASERVER. This is shown in Figure 5-4.

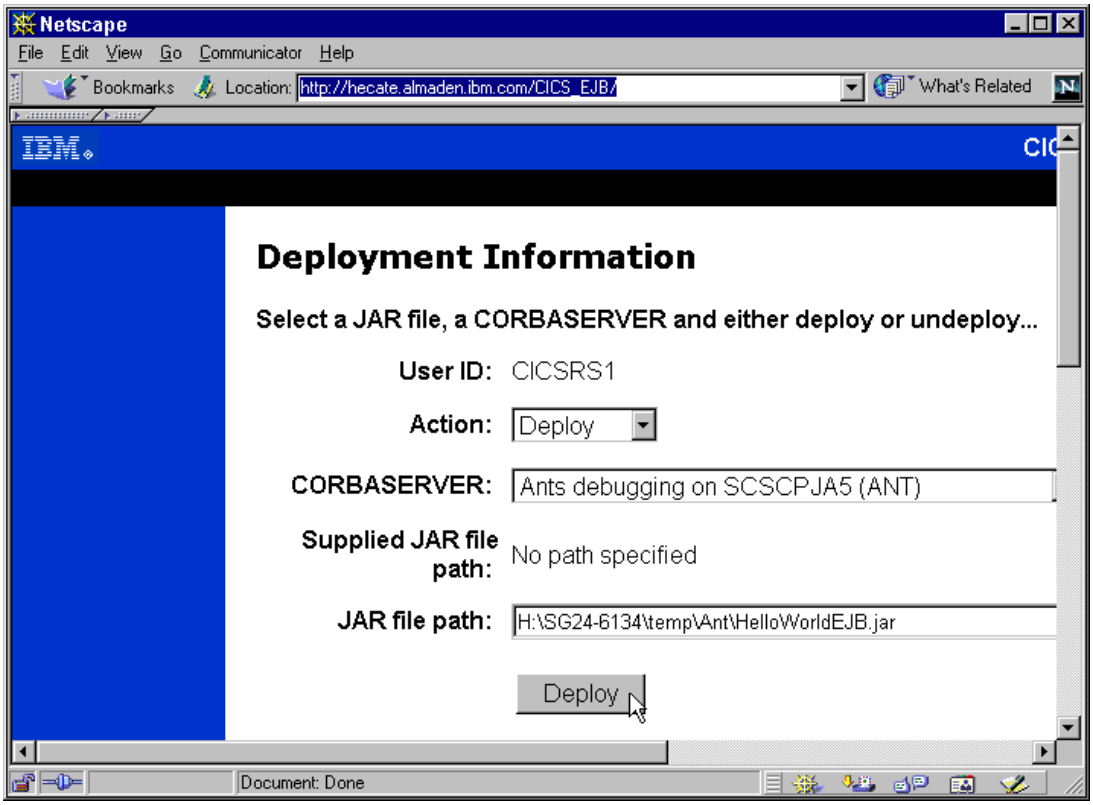


Figure 5-4 Deploying a JAR to a debugging CorbaServer

If the CICS development deployment tool is not being used, the enterprise bean needs to be deployed to CICS as normal, and then the new REQUESTMODEL should be installed.

The Trader application is described in Chapter 7, “Wrapping the Trader application: JCICS link” on page 171. We then installed our debugging REQUESTMODEL with:

```
CEDA INS GROUP(TRADDBG)
```

Now all requests for the trader bean will run under the ADRP request processor transaction, which uses a JVM with the debugging options.

To stop using the debugging request processor transaction, the REQUESTMODEL can be deleted with the command:

```
CEMT DISC REQ(TRADDBG)
```

The **CEDA INS** command and **CEMT DISC** command can be used to toggle back and forth between debugging and normal operation.

JPDA debugging clients and debugging a program

The strategic IBM debugging tools for the CICS EJB server are the IBM Distributed Debugger and object-level trace.

Attention: At the time of writing this redbook, the IBM Distributed Debugger did not support Java 1.3 and the JPDA, so we could not use it with CICS TS V2.1. However, subsequent to this redbook project, we found that CICS TS V 2.1 does work with V9.1.4 (or later) of the IBM distributed debugger. Version 9.1.4 and subsequent versions of the IBM Distributed debugger offer support for JPDA debugging.

Other debugging clients are available which do support Java 1.3 and JPDA. The standard Java installation comes with a command line debugger called `jdb` which works with the CICS TS V2.1. There are also commercial or Open Source debug clients available which support JPDA and provide a more friendly GUI than the `jdb` command line interface. Simply entering `JPDA client` on your favorite Internet search engine should provide links to more information on these.

Our examples here are shown using the `jdb` command line client but as all the clients use the JPDA architecture, all will have very similar commands and capabilities.

Making the Java source code available to the debugger

While debugging a program the debugger client can display the lines of source code as it steps through them. This can make it much easier to see what is going on in the program being debugged. To do this the Java source must be available to the debugger.

In the example that follows we show `jdb` running from the root directory of the workstation C: drive, so to debug the Trader application we put the source code here. We used Winzip to extract all the files from the `TraderAll.jar` file provided with this Redbook (see Appendix C, “Using the additional material” on page 315 for information about how to obtain this file). Winzip preserves the directory structure of the JAR file which is required as the debugger client uses the package name of a class to locate the source code.

Using jdb

We now describe a simple debugging session using the `jdb` client to debug the Trader client we deployed in the previous section.

We first start the Trader Web application as described in “Testing the Trader servlet” on page 212 and click on the Logon button. This should hang as the CICS JVM waits for a remote debugger to connect. Running the CEMT transaction to show the active tasks should verify that the correct request processor transaction (ADRP) has indeed been started; this is shown in Figure 5-5.

```
I TAS
 STATUS:  RESULTS - OVERTYPE TO MODIFY
  Tas(0000037) Tra(CIRR)           Sus Tas Pri( 001 )
     Sta(U ) Use(CICSUSER) Uow(B5A4E62CB94FB1C4) Hty(SOCBNOTI)
  Tas(0000038) Tra(ADRP)           Run Tas Pri( 001 )
     Sta(U ) Use(CICSUSER) Uow(B5A4E62CB9D47302)
  Tas(0000041) Tra(CEMT) Fac(TC44) Run Ter Pri( 255 )
     Sta(TO) Use(CICSUSER) Uow(B5A4E63D410783C2)




                                                SYSID=PJA5 APPLID=SCSCPJA5
  RESPONSE: NORMAL                          TIME:  00.41.47  DATE: 04.06.01
PF 1 HELP       3 END       5 VAR       7 SBH 8 SFH 9 MSG 10 SB 11 SF
```

Figure 5-5 CEMT I TAS showing the waiting request processor transaction ADRP

The `jdb` client is started from a Windows command prompt on a workstation with Java installed and in the workstations PATH. It is started by entering `jdb` followed by the various startup parameters.

As we have set up our CICS system to wait for the remote debugger to initiate the connection, we use the `-attach` parameter to tell `jdb` to initiate the connection to the CICS JVM. The attach parameter specifies the hostname of the CICS system followed by the port number specified in the JVMPROFILE of request processor program. For example we used:

```
jdb -attach wtsc69oe.itso.ibm.com:10177
```

Information on the various `jdb` parameters can be displayed by entering the *help* option, for example, `jdb -help`.

Once a connection to the remote JVM is established breakpoints can be set with the `stop` command, and execution of the remote JVM controlled with the `cont` and `next` commands.

A basic debugging session is shown in Example 5-3. The CICS JVM is in debug mode waiting for a debugger client to initiate a connection. We connect to it `jdb`, set a breakpoint in the Trader `logon()` method, and when that break point is hit, we alter the name of the user logging on.

Example 5-3 Using jdb to step through the hello() method of the TraderBean sample

```
C:\>jdb -attach wtsc69oe.itso.ibm.com:10177
Initializing jdb...
VM Started: main[1] No frames on the current call stack

main[1] stop in *.TraderBean.logon
Deferring breakpoint *.TraderBean.logon.
It will be set after the class is loaded.
main[1] cont
> Set deferred breakpoint *.TraderBean.logon

Breakpoint hit: thread="ANTDIIRP.TASK200.ADRP", itso.ejb390.trader.TraderBean.logon(),
line=114, bci=0
  114   ivUserID = userID;

ANTDIIRP.TASK200.ADRP[1] list logon
110        }
111        public void logon(String userID, String password, String connectURL, String
cicsServer) throws Exception {
112
113             // save userID
114     =>      ivUserID = userID;
115
116             // now logon to system
117             ivTraderBackend.logon(userID, password, connectURL, cicsServer);
118
119        }
ANTDIIRP.TASK200.ADRP[1] locals
Method arguments:
  userID = "ANT"
  password = "ELDER"
  connectURL = "local:"
  cicsServer = ""
Local variables:
ANTDIIRP.TASK200.ADRP[1] set userID="cicsrs1"
 userID="cicsrs1" = "cicsrs1"
ANTDIIRP.TASK200.ADRP[1] next

Step completed: ANTDIIRP.TASK200.ADRP[1] thread="ANTDIIRP.TASK200.ADRP",
itso.ejb390.trader.TraderBean.logon(), line=117, bci=5
  117   ivTraderBackend.logon(userID, password, connectURL, cicsServer);
```

```
ANTDIIRP.TASK200.ADRP[1] next

Step completed: ANTDIIRP.TASK200.ADRP[1] thread="ANTDIIRP.TASK200.ADRP",
itso.ejb390.trader.TraderBean.logon(), line=111, bci=19
  111 public void logon(String userID, String password, String connectURL, String
cicsServer) throws Exception {

ANTDIIRP.TASK200.ADRP[1] cont
>
The application exited

C:\>
```

The details of each of these steps are as follows:

- First the **stop** command is used to set a breakpoint in the `logon()` method of the TraderBean class. We use an asterisk(*) to prefix the class name to avoid entering the full package name of the class.
- When the breakpoint is hit we display the source of the method with the **list** command.
- The **locals** command is then used to show the values of the local variables. This displays each local variable along with its current value.
- The userID local variable is altered with the **set** command. This is the local variable holding name of the user logging on which we change with the **set** command to be the name of a different user.
- We then use the **next** command to step through the lines of code as they execute. The display shows the `logon()` method of the Trader backend class being called.
- Finally, the **cont** command is used to let execution continue normally.

Entering **help** at the **jdb** command prompt causes **jdb** to print out all its commands with a short description of what they do. Some of the more useful commands are:

- **threads** to list the running threads
- **where <thread id>** to print a stacktrace of a thread
- **eval** to evaluate an expression, which can use any variables and call any methods that are in scope.
- **classes** and **class <classname>** to show details of classes
- **methods <classname>** to list methods of a class
- **fields <classname>** to list fields (variables) of a class
- **locals** to list all local variables
- **set** to change the value of a field or variable

5.2 WebSphere diagnostic aids

A complete description on how to use and debug Web applications on WebSphere Application Server is out of the scope of this redbook and there have been many other publications on this topic already. We can only mention here some of the places within WebSphere which we found particularly useful in getting our Web applications running.

In-depth information on the using WebSphere Application Server can be found in the WebSphere InfoCenter, at this URL:

`http://www.ibm.com/software/webservers/appserv/library.html`

More information on using and debugging Web applications in WebSphere can be found in the redbook, *WebSphere Application Servers: Standard and Advanced Editions*, SG24-5460.

5.2.1 WebSphere logs

The WebSphere Advanced Administrator Console contains a Console Messages window which can be useful while developing servlets. This displays messages about servlet activity such as when servlets are initialized, and any uncaught exceptions thrown by the servlet code.

The WebSphere Application server logs record useful information to aid with debugging applications. In a default installation, the logs are located in the directory:

```
C:\WebSphere\AppServer\logs
```

A number of logs are stored in this directory; the most useful are `default_server_stdout.log` and `default_server_stderr.log`. These are the Java `stdout` and `stderr` files, and they are where messages and uncaught exceptions from servlets are recorded.

An example of the type of information found in the `stderr.log` is shown in Example 5-4. This error was caused by specifying a non-existent prefix in the JNDIName field of the HelloWorld sample Web application. The HelloWorld sample is described in “Test the cicshello sample” on page 100.

Example 5-4 WebSphere default_server_stderr.log example

```
...
javax.naming.NameNotFoundException: XXX/HelloWorld
   at javax.naming.NamingException.<init>(NamingException.java:104)
   at javax.naming.NameNotFoundException.<init>(NameNotFoundException.java:40)
   at com.ibm.ejs.ns.jndi.CNContextImpl.doLookup(CNContextImpl.java:729)
   at com.ibm.ejs.ns.jndi.CNContextImpl.lookup(CNContextImpl.java:584)
   at javax.naming.InitialContext.lookup(InitialContext.java:349)
   at helloworld.sample.DataBean.hello(DataBean.java:43)
   at helloworld.sample.HelloWorldServlet.performTask(HelloWorldServlet.java:128)
   at helloworld.sample.HelloWorldServlet.doPost(HelloWorldServlet.java:54)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:566)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:627)
...
```

5.2.2 COS Naming Server

CICS uses the COS Naming Server facility provided by WebSphere when publishing its resources, and the EJB clients use it to locate the location of the enterprise beans. WebSphere provides no tools for the COS Naming Server, and it can be difficult to verify that the IORs are being correctly published to it by CICS.

Verifying that the COS Naming Server is running

The COS Naming Server function is provided by the AdminServer part of WebSphere. By default, the AdminServer is not started automatically when the WebSphere workstation is booted. To manually start AdminServer on the workstation, use **Start -> Programs -> IBM WebSphere -> Application Server V3.5 -> Start Admin Server**.

The AdminServer can be set to start automatically from the Services window of the Windows NT Control Panel. The service name is IBM WS AdminServer.

The **netstat** command can be used to verify that the workstation is listening on the correct TCP/IP port. From a Windows command prompt, enter **netstat -a** and it should show that the workstation is listening on port 900. This is shown in Example 5-5 which shows an extract of the **netstat** command running on our WebSphere workstation.

Example 5-5 Using netstat to list active TCP/IP connections

```
C:\>netstat -a
Active Connections
Proto  Local Address          Foreign Address        State
  TCP    hecate:80            0.0.0.0:0              LISTENING
  TCP    hecate:135           0.0.0.0:0              LISTENING
  TCP    hecate:135           0.0.0.0:0              LISTENING
  TCP    hecate:900           0.0.0.0:0              LISTENING
  TCP    hecate:1027          0.0.0.0:0              LISTENING
  TCP    hecate:1029          0.0.0.0:0              LISTENING
```

Querying the COS Naming Server

We found it useful to be able to query the WebSphere COS Naming Server to see if our enterprise beans really were published to it. To do this we wrote a simple utility program, called *JNDIList* to list out the contents of the JNDI directory. Appendix C, “Using the additional material” on page 315 contains information on how to create this program and the other sample programs provided with this redbook.

The JNDIList program takes two parameters, the URL of the COS Naming Server, and optionally the literal IOR. Appending the JNDI prefix to the URL restricts the output to only show references for that JNDI prefix. Using a specific JNDI prefix makes the utility run faster and limits the number of references displayed. This makes it much easier to locate a particular enterprise bean reference.

The optional second parameter, IOR, causes the stringified IOR for each bean to also be listed. This could then be cut-and-pasted into another utility which formats the IOR. There are many such utilities available including Web pages where you paste the IOR into a form and a CGI program formats it. Searching for *IOR parser* with your favorite Internet search engine should locate several of these sites. The IOR contains information such as the host address and port number and the complete object key.

Example 5-6 shows the JNDIList utility being run against our COS Naming Server for the JNDI prefix /ITSO/PJA5.

Example 5-6 JNDIList utility showing the samples published to the COS Naming Server

```
C:\JNDIList>java JNDIList iiop://hecate:900/ITSO/PJA5
Trader.itso\.ejb390\.trader\.EJSRemoteTraderHome
HelloWorldSession.itso\.ejb390\.helloworld\.EJSRemoteHelloWorldSessionHome
HelloWorld.helloworld\.sample\.EJSRemoteHelloWorldHome


C:\JNDIList>
```

5.3 Traditional CICS diagnostic aids

This section describes the various CICS options available to aid with diagnosing problems with enterprise beans in CICS. We only describe the areas which we found particularly useful for solving the general configuration problems we had during the development and testing of our sample programs. For more complete information on debugging CICS problems, see the *CICS Problem determination Guide*, GC33-5719.

5.3.1 CICS job log and console messages

The CICS job log and OS/390 console contain useful messages about CICS activity. This was one of the first places we would check when a problem had been narrowed down to somewhere within the CICS region. Example 5-7 shows some typical messages produced. In this example CICS has published an enterprise bean and the message indicates the URL of the COS Naming Server.

Example 5-7 Extract of the CICS job log show messages about EJB activity

```
...
DFHAD2000 I 04/11/2001 21:20:06 SCSCPJA5 A DJAR named J2DJ0013 was created by CICSRS1.
DFHEJ1540 04/11/2001 21:20:18 SCSCPJA5 DJar J2DJ0013 and the Beans it contains are now
accessible.
DFHEJ5009 04/11/2001 21:20:24 SCSCPJA5 Published bean HelloWorld to JNDI server
iiop://hecate.almaden.ibm.com:900 at location ITSO/PJA5.
...
```

5.3.2 CICS auxiliary trace

The CICS auxiliary trace can be useful in some circumstances. However, in many problem situations, once the JVM starts executing the enterprise bean, then the Java code problems do not cause entries to be written to the trace file. For example, if the Java program catches its own exceptions, then no messages appear in the CICS trace file about the exceptions.

The trace file can be useful for CICS configuration problems relating to enterprise beans. For example, Example 5-8 provides an edited extract of a CICS auxiliary trace which shows the exception CICS gets when receiving an incoming request for an enterprise bean that has not been installed.

Example 5-8 Trace extract showing HelloWorld bean not defined to CICS

```
00128 J8005 EJ 0602 JRAS  ENTRY com.ibm.cics.ejs.csi.EJJOGate getBeanMetaData
00128 J8005 EJ 0E01 EJJO  ENTRY GET_BEAN_DD           PJA5,HelloWorld,352026B8 ,
00128 J8005 EJ 0B01 EJBG  ENTRY GET_BEAN_DD           PJA5,352026B8 , 00000000 ,
00128 J8005 EJ 0701 EJCG  ENTRY INQUIRE_CORBASERVER   PJA5
00128 J8005 EJ 071A EJCG  EXIT  INQUIRE_CORBASERVER/OK INSERV
00128 J8005 EJ 0B02 EJBG  EXIT  GET_BEAN_DD/EXCEPTION BEAN_ABSENT,,352026B8 , 00
00128 J8005 EJ 0E0B EJJO  EXIT  GET_BEAN_DD/EXCEPTION BEAN_ABSENT,,0,352026B8 ,
00128 J8005 EJ 0701 EJCG  ENTRY INQUIRE_CORBASERVER   PJA5
00128 J8005 EJ 071A EJCG  EXIT  INQUIRE_CORBASERVER/OK INSERV
00128 J8005 EJ 0901 EJDG  ENTRY COUNT_FOR_CS          PJA5
00128 J8005 EJ 0911 EJDG  EXIT  COUNT_FOR_CS/OK       2,0,0,0,0,0,0,2,0
00128 J8005 EJ 0911 EJDG  EXIT  WAIT_FOR_USABLE_DJARS/OK
00128 J8005 EJ 0E0B EJJO  EXIT  WAIT_FOR_USABLE_DJARS/OK
00128 J8005 EJ 0B01 EJBG  ENTRY GET_BEAN_DD           PJA5,352026B8 , 00000000 ,
00128 J8005 EJ 0701 EJCG  ENTRY INQUIRE_CORBASERVER   PJA5
00128 J8005 EJ 071A EJCG  EXIT  INQUIRE_CORBASERVER/OK INSERV
00128 J8005 EJ 0B02 EJBG  EXIT  GET_BEAN_DD/EXCEPTION BEAN_ABSENT,,352026B8 , 00
00128 J8005 EJ 0E0B EJJO  EXIT  GET_BEAN_DD/EXCEPTION BEAN_ABSENT,,0,352026B8 ,
00128 J8005 EJ 0603 JRAS  *EXC* com.ibm.cics.ejs.csi.EJJOGate getBeanMetaData
```

```
00128 J8005 EJ 0603 JRAS  *EXC* com.ibm.cics.ejs.csi.CICSBeanMetaDataStore getIn
00128 J8005 EJ 0603 JRAS  *EXC* com.ibm.cics.ejs.csi.CICSBeanMetaDataStore get
00128 J8005 EJ 0603 JRAS  *EXC* com.ibm.cics.ejs.csi.CICSObjectAdapter keyToObje
00128 J8005 II 1003 JRAS  *EXC* com.ibm.cics.iiop.orb.ExtendedServerDelegate get
00128 J8005 II 1003 JRAS  *EXC* com.ibm.cics.iiop.orb.CICSConnection doWork (Req
00128 J8005 II 1002 JRAS  ENTRY com.ibm.rmi.iiop.IIOPOutputStream writeTo (java.
00128 J8005 II 0700 IIRP  ENTRY SEND_REPLY            0D362840,1,35254F80 , 0000
00128 J8005 II 0200 IIRH  ENTRY PARSE                35254F80 , 00000014
00128 J8005 II 0201 IIRH  EXIT  PARSE/OK             146,00000000 , 00000000,00
00128 J8005 II 0714 IIRP  EVENT IIOP_DATA            ABOUT TO SEND GIOP REPLY
00128 J8005 RZ 0120 RZTA  ENTRY SEND_REPLY           35254F80 , 00000014,1
```

5.3.3 Verifying that the request receiver transaction runs

One thing we found useful early-on in the setting up of our CICS system with enterprise bean support was to determine if a request had even reached CICS or not. The first thing that happens when a request is received by CICS is that the request receiver transaction is run. Using **`CEMT I PROG(DFHIIRRS)`** to display the use count of the request receiver program provides a quick way to verify this without having to resort to running a trace.

This technique does not work with the request processor program because CEMT does not show a use count for Java programs. However, it is possible to use CEDF to determine if the correct request processor transaction is running, and this is described further in the following section.

5.3.4 Using EDF with enterprise beans

The CEDX transaction is used to invoke the CICS execution diagnostic facility (EDF) for testing application programs that are associated with non-terminal transactions. CEDX can be used with the request processor transaction CIRP or one of its aliases.

While testing our enterprise beans, we found two ways that using CEDX on a request processor transaction could be useful:

- To step through the EXEC CICS commands used by an enterprise bean. This can be done both for a called a program in another language, or for a CICS Java program using the JCICS classes.
- To verify that the REQUESTMODEL was mapping our requests to the correct request processor alias transaction.

  When used with CIRP, EDF will always show the final task terminate screen. We found this useful when testing that a particular request processor alias has indeed been selected by the REQUESTMODEL.

Restriction: There are restrictions on having multiple request processor transactions using EDF. This means that the TRANSCLASS resource definition of the request processor transaction alias needs to have the MAXACTIVE attribute set to a value of 1. For more information on this, see the section “Using EDF with enterprise beans’ in *Java applications in CICS*, SC34-5881.

5.4 Debugging common errors

In this section we describe how to diagnose some common problems with Web applications using enterprise beans in CICS. The first part of this section gives an overview of the method we use when trying to diagnose problems. The second part of the section gives some examples of the symptoms of various problems and what is done to resolve them.

5.4.1 Overview of debugging a Web application

We suggest using the following strategy when attempting to diagnose a problem with a Web application using an enterprise bean in CICS:

1. Verify that the IBM HTTP server is serving Web pages.
2. Verify WebSphere Application server is starting the servlet.
3. Check for exceptions and messages in the WebSphere logs.
4. Verify that the request reached CICS.
5. Verify that the expected request processor transaction is called.
6. Check for any messages on the OS/390 console or CICS job log.
7. Check for any messages in the Java `stderr` files in the HFS.
8. Use JPDA to step through the code of the enterprise bean.

Verify that the IBM HTTP server is serving Web pages

Verify that the IBM HTTP server is running and serving Web pages. On the WebSphere workstation, starting a Web browser and entering `http://localhost/` as the URL should show the default index.html page. Repeating this on another workstation but specifying the WebSphere Application server hostname as the URL should also show this page; this verifies that the Web server is running and that there is remote connectivity to it.

Verify that the WebSphere Application server is starting the servlet

The Console Messages window of the WebSphere Administrators console will show messages when a servlet is loaded and initialized. This verifies that the HTML page is causing a servlet to be started. For example, the first time we use our CICS development deployment tool, the SCSCPJA5 servlet is initialized, as shown in Example 5-9.

Example 5-9 WebSphere initializing the CICS development deployment tool servlet

```
4/8/01 7:56 PM : AUDIT [hecate/Default Server]: Loading.servlet:."SCSCPJA5"
4/8/01 7:56 PM : AUDIT [hecate/Default Server]:[Servlet.LOG]:
                         "com.ibm.cics.addeploy.servlet.CicsEjbAdServlet: [SCSCPJA5]init"
4/8/01 7:57 PM : AUDIT [hecate/Default Server]:Servlet.available.for.service:."SCSCPJA5"
4/8/01 7:57 PM : AUDIT [hecate/Default Server]:[Servlet.LOG]:
                         "pagecompile._loginIBM01_xjsp: init"
4/8/01 7:57 PM : AUDIT [hecate/Default Server]:[Servlet.LOG]:
                         "com.ibm.servlet.jsp.http.pagecompile.JSPState: init"
```

Check for exceptions or messages in the WebSphere logs

Many Web application problems will cause messages to be written to the WebSphere Application Server log files. These files are described in 5.2, “WebSphere diagnostic aids” on page 115.

An example of the type of messages written to the WebSphere logs is shown in Example 5-10. Here the HelloWorld sample Web application is failing with a Java NameNotFound exception. The cause of this problem is that a DJAR resource in CICS has not been published, so when the servlet does a JNDI lookup for the bean it gets this exception.

Example 5-10 HelloWorld naming exception due

```
javax.naming.NameNotFoundException: ITSO/PJA5/HelloWorld
   at javax.naming.NamingException.<init>(NamingException.java:104)
   at javax.naming.NameNotFoundException.<init>(NameNotFoundException.java:40)
   at com.ibm.ejs.ns.jndi.CNContextImpl.doLookup(CNContextImpl.java:729)
   at com.ibm.ejs.ns.jndi.CNContextImpl.lookup(CNContextImpl.java:584)
   at javax.naming.InitialContext.lookup(InitialContext.java:349)
   at helloworld.sample.DataBean.hello(DataBean.java:43)
   at helloworld.sample.HelloWorldServlet.performTask(HelloWorldServlet.java:128)
   at helloworld.sample.HelloWorldServlet.doPost(HelloWorldServlet.java:54)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:566)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:639)
...
```

Verify that the request reached CICS

When an IIOP request is received by CICS the request receiver program should be invoked. The request receiver program is the program associated with the TRANSID attribute of the TCPIPSERVICE definition, which by default is CIRR.

CEMT can be used to verify that the use count increases for the request receiver program each time a client program tries to invoke a method on an enterprise bean in CICS. If this use count does not increase, then the client requests have probably not reached CICS. This is shown in Figure 5-6.

```
CEMT I PROG(DFHIIRRS)
STATUS:  RESULTS - OVERTYPE TO MODIFY
Prog(DFHIIRRS) Len(0000001032) Ass Pro Ena Pri
     Res(000) Use(0000000150) Any Cex Ful Qua           Nat




                                                  SYSID=PJA5 APPLID=SCSCPJA5
  RESPONSE: NORMAL                           TIME:  23.21.20  DATE: 04.08.01
PF 1 HELP       3 END       5 VAR        7 SBH 8 SFH 9 MSG 10 SB 11 SF
```

Figure 5-6 Displaying the request receiver programs use count

Verify that the expected request processor transaction is called

It can be useful to verify that the correct request processor transaction is being selected by the REQUESTMODEL resource definitions. For example, we used the command **CEDX ADRP** to verify that our debugging request processor was being called for the correct requests. The next time that an incoming request is matched to this transaction, we get the EDF initiation screen, as shown in Figure 5-7.

This EDF program initiation screen is displayed even when using the default request receiver program which defines the EDF attribute set to NO.

```
TRANSACTION: ADRP PROGRAM: ANTDIIRP TASK: 0000542 APPLID: SCSCPJA5 DISPLAY:  00
  STATUS:  PROGRAM INITIATION

     EIBTIME      = 232852
     EIBDATE      = 0101098
     EIBTRNID     = 'ADRP'
     EIBTASKN     = 542
     EIBTRMID     = '....'

     EIBCPOSN     = 0
     EIBCALEN     = 0
     EIBAID       = X'00'                                        AT X'0DBB00EA'
     EIBFN        = X'0000'                                      AT X'0DBB00EB'
     EIBRCODE     = X'000000000000'                              AT X'0DBB00ED'
     EIBDS        = '........'
 +   EIBREQID     = '........'

ENTER:  CONTINUE
 PF1 : UNDEFINED           PF2 : SWITCH HEX/CHAR     PF3 : END EDF SESSION
 PF4 : SUPPRESS DISPLAYS   PF5 : WORKING STORAGE     PF6 : USER DISPLAY
 PF7 : SCROLL BACK         PF8 : SCROLL FORWARD      PF9 : STOP CONDITIONS
 PF10: PREVIOUS DISPLAY    PF11: EIB DISPLAY         PF12: UNDEFINE
```

Figure 5-7 Verifying that a request processor alias is used with EDF

Check for messages in the OS/390 console or CICS job log

Many errors will cause messages to be written to the CICS job log. The messages typically seen here will occur when the problem is drastic enough to prevent the JVM from initializing or running the request receiver program. An example of this type of problem is shown in Example 5-11, which is caused by the request receiver program specifying a JVMPROFILE that pointed to a non-existent system properties file.

Example 5-11 Sample messages from the CICS job log

```
DFHSJ0509 04/09/2001 00:05:50 SCSCPJA5 Attempt to open JVM system properties file
/u/cicsts21/props/Xfjjvmpr.props has failed. Runtime error message is EDC5129I No such
file or directory.
DFHAC2016 04/09/2001 00:05:50 SCSCPJA5 Transaction CIRP cannot run because program
DFJIIRP is not available.
DFHII0108 04/09/2001 00:05:50 SCSCPJA5 9.1.150.233 PJA5 The request receiver was
notified that a reply could not be delivered for requestId 274. Reason: Request Stream
closed.
```

Check for any messages in the Java stderr files in the HFS

The Java `stderr` files will record JVM problems and exceptions that are not caught by the enterprise bean code. Sometimes these messages can be very specific about what the problem is. Example 5-12 shows the messages you get in the Java `stderr` file when an enterprise bean tries to use JDBC but the DB2 library is no defined in the JVMPROFILE LIBPATH parameter.

Example 5-12 Java stderr file in HFS showing DB2 missing from the JVMPROFILE

```
Can't find library db2os390j2comp
Make sure that the library is in your path
java.lang.UnsatisfiedLinkError: no db2os390j2comp  (libdb2os390j2comp.so) in
java.library.path
SQLException loading DLL/registering JDBC Driver
```

```
SQLSTATE is FFFFF
SQLCODE  is -1
java.sql.SQLException: Error: DB2 JDBC Driver was unable to load the DLL db2os390j2comp
java.lang.Exception: AIIA
```

Use JPDA to step through the code of the enterprise bean

We found that where possible, it is much easier to test and debug enterprise bean logic using the WebSphere Test Environment which is described in 6.2.2, “Testing in VAJ” on page 142. Sometimes this is not possible, such as when using the JCICS classes, and in these cases a JPDA debugger can be used to debug the Java code within CICS. See “Debugging an enterprise bean using JPDA” on page 109 for more details.

5.4.2 Common problems

This section shows examples of some of the problems that we encountered while developing the applications in this redbook. We show the symptoms of the problem occurring and describe what we did to resolve the problem.

Problems with the IVP HelloWorld application

The CICS supplied HelloWorld program used in the installation verification has limited error handling. Most problems result in the program returning an HTML page showing a blank Sample Results page (Figure 5-8).

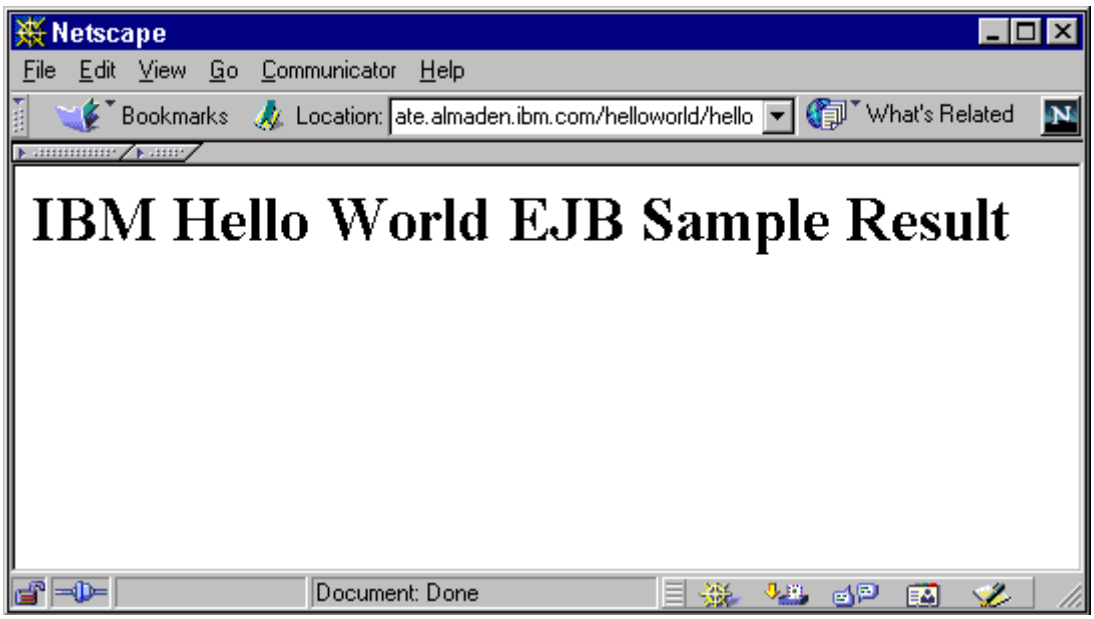


Figure 5-8 HelloWorld sample blank error page

When this blank page is shown, the HelloWorld Web application will have written exception information to the WebSphere `stderr` log. An example of this is shown in Example 5-13.

Example 5-13 HelloWorld naming exception due to the DJAR not published

```
javax.naming.NameNotFoundException: ITSO/PJA5/HelloWorld
   at javax.naming.NamingException.<init>(NamingException.java:104)
   at javax.naming.NameNotFoundException.<init>(NameNotFoundException.java:40)
   at com.ibm.ejs.ns.jndi.CNContextImpl.doLookup(CNContextImpl.java:729)
   at com.ibm.ejs.ns.jndi.CNContextImpl.lookup(CNContextImpl.java:584)
   at javax.naming.InitialContext.lookup(InitialContext.java:349)
   at helloworld.sample.DataBean.hello(DataBean.java:43)
   at helloworld.sample.HelloWorldServlet.performTask(HelloWorldServlet.java:128)
   at helloworld.sample.HelloWorldServlet.doPost(HelloWorldServlet.java:54)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:566)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:639)
```

This shows a Java NameNotFoundException which is caused by not publishing the HelloWorld DJAR. This type of exception also occurs if WebSphere is reinstalled or if the JNDI prefix is specified incorrectly, either in the CORBASERVER resource definition, or on the HelloWorld sample Web page form.

Similar symptoms occur if the DJAR has been successfully published in the past, but now does not exist in CICS. This could occur, for example, after a CICS cold start, if the group containing the DJAR has not been installed. The exception is different in this situation, a NoSuchObjectException which is shown in Example 5-14.

Example 5-14 HelloWorld naming exception due

```
java.rmi.NoSuchObjectException: CORBA OBJECT_NOT_EXIST 1 No; nested exception is:
   org.omg.CORBA.OBJECT_NOT_EXIST:   minor code: 1  completed: No
org.omg.CORBA.OBJECT_NOT_EXIST:   minor code: 1  completed: No
   at java.lang.RuntimeException.<init>(RuntimeException.java:49)
   at org.omg.CORBA.SystemException.<init>(SystemException.java:51)
   at org.omg.CORBA.OBJECT_NOT_EXIST.<init>(OBJECT_NOT_EXIST.java:72)
   at org.omg.CORBA.OBJECT_NOT_EXIST.<init>(OBJECT_NOT_EXIST.java:61)
   at com.ibm.CORBA.iiop.IIOPConnection.locate(Unknown Source)
   at com.ibm.CORBA.iiop.GIOPImpl.locate(Unknown Source)
   at com.ibm.CORBA.iiop.ClientDelegate.createRequest(Unknown Source)
   at com.ibm.CORBA.iiop.ClientDelegate.request(Unknown Source)
   at org.omg.CORBA.portable.ObjectImpl._request(ObjectImpl.java:237)
   at helloworld.sample._HelloWorldHome_Stub.create(_HelloWorldHome_Stub.java:164)
   at helloworld.sample.DataBean.hello(DataBean.java:45)
   at helloworld.sample.HelloWorldServlet.performTask(HelloWorldServlet.java:128)
   at helloworld.sample.HelloWorldServlet.doPost(HelloWorldServlet.java:54)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:566)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:639)
...
```

CICS development deployment tool application problems

In this section we describe the symptoms of problems that can happen with the CICS development deployment tool.

Figure 5-9 shows the error you get when the CICS development deployment tool cannot run its servlet. Likely causes of this problem are that the WebSphere Application Server Admin Server is not running, or there is a configuration problem with the servlet name.

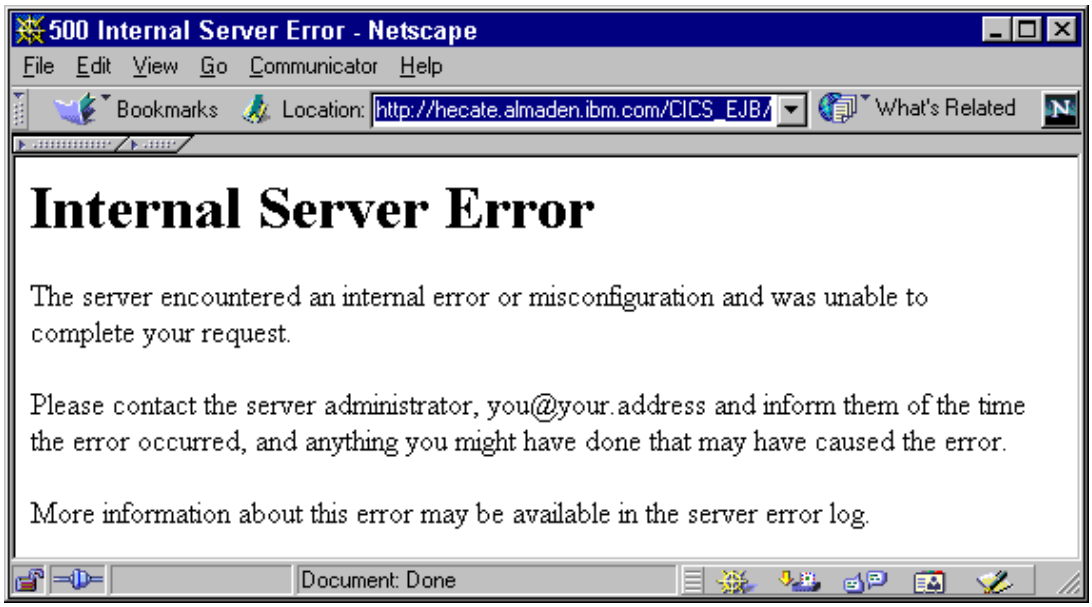


Figure 5-9 IBM HTTP Server unable to connect to WebSphere Application server

Next, Figure 5-10 shows the *Deployment tool unavailable* page. This is a generic error page that is displayed for nearly all problems with the CICS development deployment tool. It does verify that the servlet is running but the problem could be anything from a problem in WebSphere to a problem with the enterprise bean in CICS. To determine the nature of the problem, the steps outlined in 5.4.1, “Overview of debugging a Web application” on page 119 should be followed.

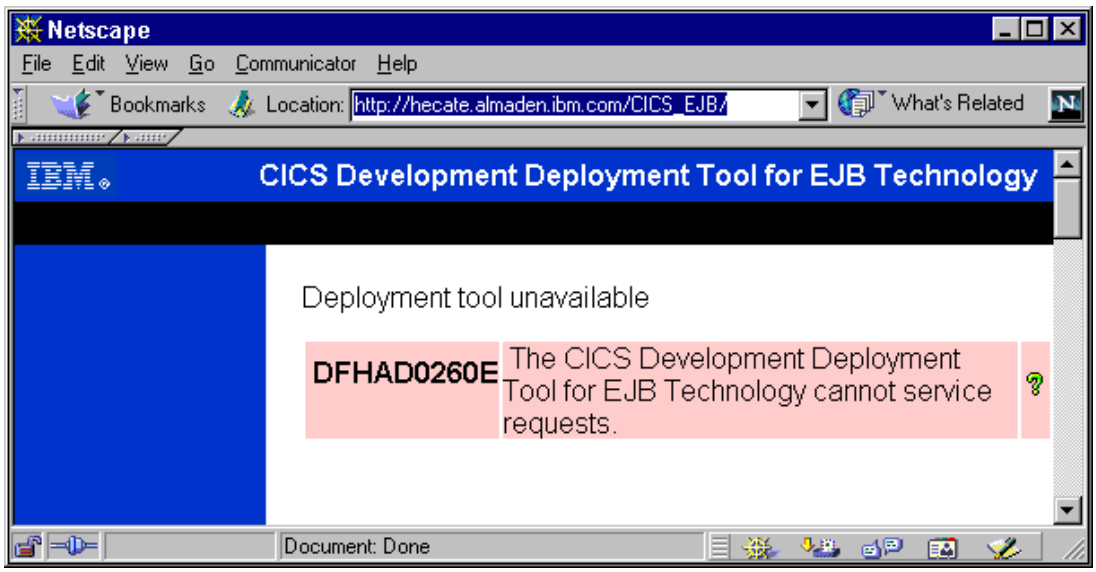


Figure 5-10 CICS development deployment tool servlet problems

Any errors in the CICS development deployment tool XML configuration file will prevent the tool's Web application from starting. These type of errors cause messages to be written to the WebSphere Application Server `stdout.log`. An extract of the WebSphere log for this type of error is shown in Example 5-15.

Example 5-15 CICS development deployment DCF file errors

```
2001.04.04 19:13:02.910 com.ibm.cics.addeploy.servlet.CicsEjbAdServlet init
[SCSCPJA5]DFHAD0501I CICS Development Deployment Tool for EJB Technology is starting.
2001.04.04 19:29:15.378 com.ibm.cics.addeploy.servlet.CicsEjbAdServlet
exceptionToMessageLog [SCSCPJA5]DFHAD0325E XML parsing error ("/>" or '>' expected.) at
line number 30.
2001.04.04 19:13:04.002 com.ibm.cics.addeploy.servlet.CicsEjbAdServlet
exceptionToMessageLog [SCSCPJA5]DFHAD0260E The CICS Development Deployment Tool for EJB
Technology cannot service requests.
```

Some problems cause messages to be written to the CICS development deployment tool's own log file not to the WebSphere logs. The location of this log file is specified in the DCF configuration file and by default is in the tools Web application root directory, `C:\WebSphere\AppServer\hosts\default_host\CICS_EJB`. An example of an error here is shown in Example 5-16. This problem was when the CICS development deployment tool CICS components had not been installed in CICS.

Example 5-16 CICS development deployment tool error in logfile example

```
2001.04.10 18:38:56.099 com.ibm.cics.addeploy.shared.CICSEJBDeployerReference
getEJBRef:NoUserID00000100
javax.naming.NameNotFoundException: DFHD/CICSDDTbean
at javax.naming.NamingException.<init>(NamingException.java:104)
at javax.naming.NameNotFoundException.<init>(NameNotFoundException.java:40)
at com.ibm.ejs.ns.jndi.CNContextImpl.doLookup(CNContextImpl.java:729)
at com.ibm.ejs.ns.jndi.CNContextImpl.lookup(CNContextImpl.java:584)
at javax.naming.InitialContext.lookup(InitialContext.java:349)
...
```

Problems with the Trader application

This section gives examples of many of the configuration problems possible with setting up the Trader application described in the other chapters of the book.

JAR files missing from the trusted middleware classpath

The Trader Web application catches most exceptions and displays the Java stacktrace on an error Web page. Figure 5-11 shows an example of this, where the enterprise bean in CICS received a NoClassDefFound exception. This occurred because the `CCF.jar` file was missing from the trusted middleware classpath, defined using TMSUFFIX in the JVM profile. This requirement is described in 7.3.6, "Adding the supporting JAR files to the trusted middleware classpath" on page 194.

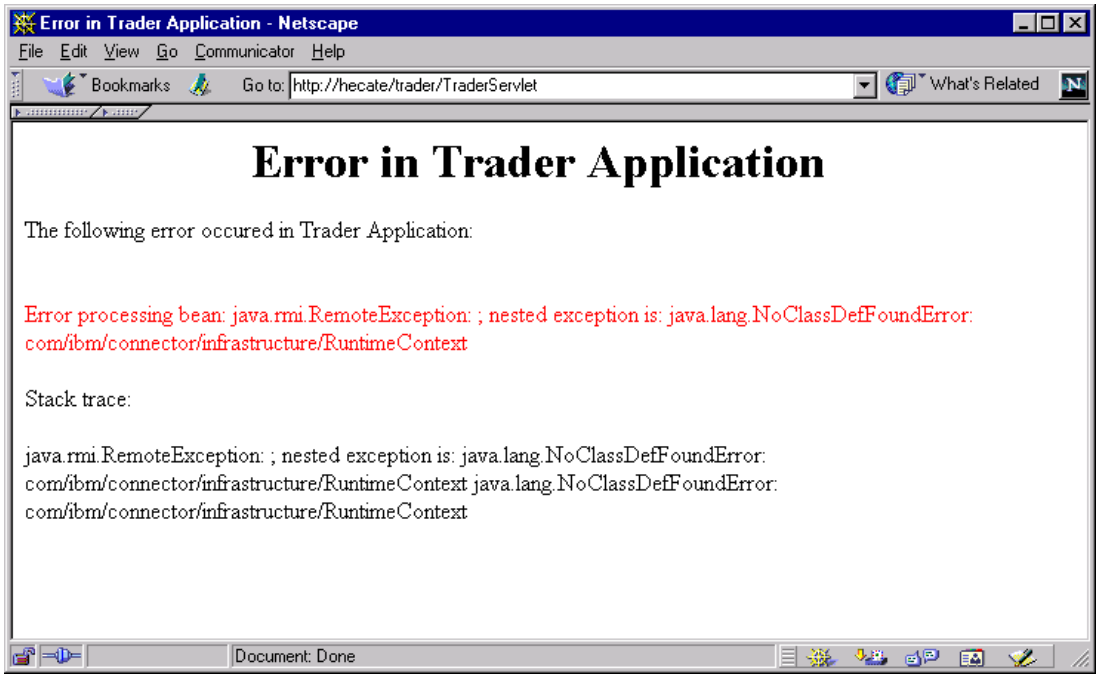


Figure 5-11 Trader error — CCF.jar missing from classpath

Trader unable to contact the CTG

Another of these types of errors is shown in Figure 5-12. In this example, the Trader application was connecting to CICS using the CTG but the CTG was not active. Again the Trader application catches this exception displaying the Java stacktrace on the Trader error page.

Figure 5-12 CCF sample unable to connect to CTG

Sometimes when a problem occurs in the Trader application, a follow-on error occurs, and the Trader error page displays information about the second error. Often the first error is what is really of interest, and in these cases, the CICS Java `stderr` files will have the information about the first problem. An example of this is shown in Figure 5-13. Here the Trader error page is showing a NullPointerException.

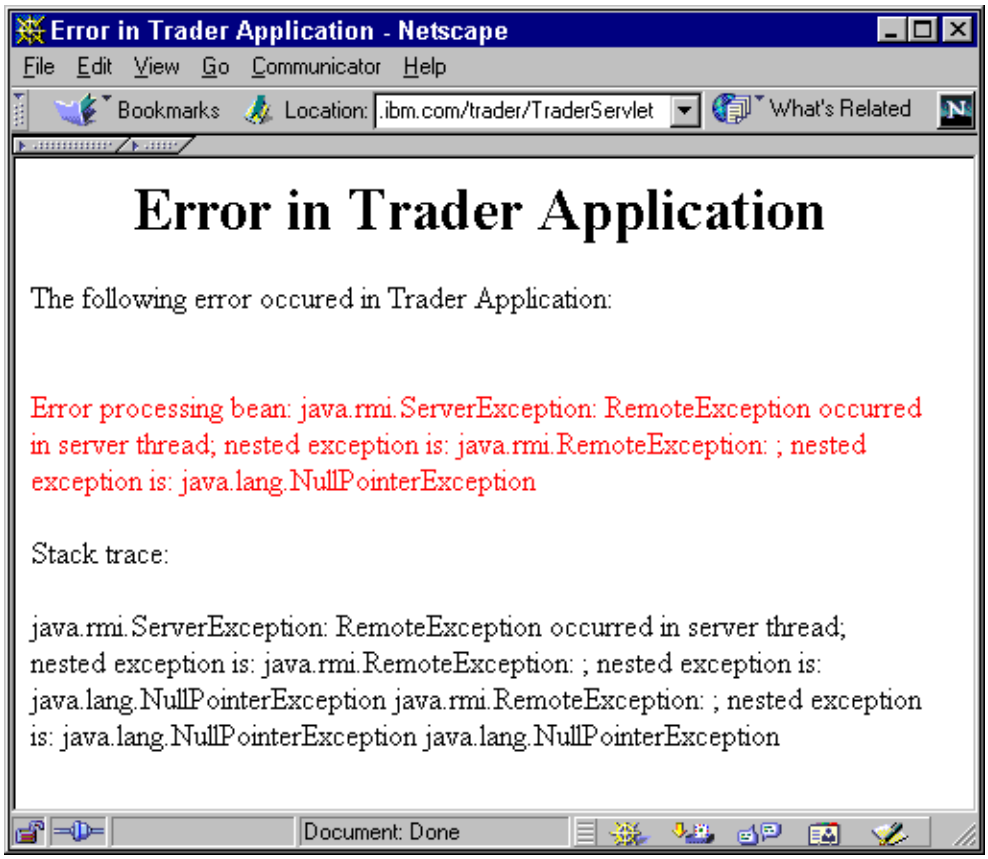


Figure 5-13 Trader showing a null pointer secondary error

Looking in the Java `stderr` files in the OS/390 HFS provides information about both the first and second problem of this example. An extract of the `stderr` file is shown in Example 5-17. This shows the first problem is an SQL exception saying *No suitable driver*. The problem here was because the `db2sqljruntime.zip` file had not been defined in the JVM profile member. Setting up the JVM profile for using DB2 with enterprise beans is discussed in “Setting CICS parameters” on page 279.

Example 5-17 Trader application SQL exception in OS/390 stderr file

```
java.sql.SQLException: No suitable driver
.at java.sql.DriverManager.getConnection(DriverManager.java:557)
.at java.sql.DriverManager.getConnection(DriverManager.java:210)
.at itso.ejb390.trader.TraderBackendDB2JDBC.openConnection(TraderBackendDB2J
.at itso.ejb390.trader.TraderBackendDB2JDBC.ejbCreate(TraderBackendDB2JDBC
.at itso.ejb390.trader.TraderBean.ejbCreate(TraderBean.java:40)
.at itso.ejb390.trader.EJSTraderHomeBean.create(EJSTraderHomeBean.java:40)
.at itso.ejb390.trader.EJSRemoteTraderHome.create(EJSRemoteTraderHome.java
.at itso.ejb390.trader._EJSRemoteTraderHome_Tie._invoke(_EJSRemoteTraderHome
...
```

Another similar problem, which also shows as a null pointer exception on the Trader error page, but has a more meaningful message in the CICS `stderr` file, is shown in Example 5-18. The messages in this example are about DB2 libraries missing from the path. The problem here is that the JVM profile member does not have the DB2 HFS directories concatenated in the LIBPATH statement. Setting up the LIBPATH for DB2 is described in “Setting CICS parameters” on page 279.

Example 5-18 Trader application missing DB2 library in OS/390 stderr file

```
Can't find library db2os390j2comp
Make sure that the library is in your path
java.lang.UnsatisfiedLinkError: no db2os390j2comp  (libdb2os390j2comp.so) in
    java.library.path
SQLException loading DLL/registering JDBC Driver
SQLSTATE is FFFFF
SQLCODE  is -1
java.sql.SQLException: Error: DB2 JDBC Driver was unable to load the DLL     db2os390j2comp
...
```

JDBC profile not available in classpath

Example 5-19 shows the Java exception received when the JDBC profile has not been copied to the classpath being used by the JVM profile. Setting up the JDBC profile is described in “Create a JDBC profile and make it accessible” on page 277.

Example 5-19 JDBC profile not available in CLASSPATH

```
java.sql.SQLException: --> JDBC is not allowed without a valid JDBC serialized profile
-> The following error messages were received while trying to read the JDBC profile:
-> Unable to find the JDBC Serialized Profile: DSNJDBC_JDBCProfile.ser.
-> Not found as System resource.
-> Also failed extended file search of all directories in classpath: /u/cicsts21/lib
.at
  COM.ibm.db2os390.sqlj.jdbc.DB2JDBCSQLCompiler.compileSQL(DB2JDBCSQLCompiler.java:166)
.at COM.ibm.db2os390.sqlj.jdbc.DB2SQLJStatement.executeQuery(DB2SQLJStatement.java:481)
.at itso.ejb390.trader.TraderBackendDB2JDBC.logon(TraderBackendDB2JDBC.java:253)
.at itso.ejb390.trader.TraderBean.logon(TraderBean.java:117)
.at itso.ejb390.trader.EJSRemoteTrader.logon(EJSRemoteTrader.java:153)
.at itso.ejb390.trader._EJSRemoteTrader_Tie._invoke(_EJSRemoteTrader_Tie.java:115)
.at com.ibm.rmi.corba.ServerDelegate.dispatch(ServerDelegate.java:284)
.at com.ibm.rmi.iiop.ORB.process(ORB.java:263)
.at com.ibm.rmi.iiop.IIOPConnection.doWork(IIOPConnection.java:1341)
.at com.ibm.cics.iiop.orb.CICSConnection.processRequest(CICSConnection.java:329)
.at com.ibm.cics.iiop.RequestProcessor.processNormalMode(RequestProcessor.java:388)
.at com.ibm.cics.iiop.RequestProcessor.main(RequestProcessor.java:170)
.at java.lang.reflect.Method.invoke(Native Method)
.at com.ibm.cics.server.Wrapper.call_main(Wrapper.java:415)
.at com.ibm.cics.server.Wrapper.callUserClass(Wrapper.java:551)
.at com.ibm.cics.server.Wrapper.main(Wrapper.java:832)
```

JDBC plan does not exist

Example 5-20 shows an SQL exception with a DB2 error code of *00F30034*. This error code can be found in the manual *DB2 Messages and Codes*, GC26-9011. For this error code, the manual says “The authorization ID associated with this connection is not authorized to use the specified plan name or the specified plan name does not exist”.

In this case, the error was caused because the plan had not been defined to DB2. Creating the JDBC profile and binding the DBRMs to DB2 is described in “Create a JDBC profile and make it accessible” on page 277.

Example 5-20 JDBC plan does not exist

```
java.sql.SQLException: DB2SQLJConnection error in native method: constructor:
    PLAN ACCESS 00F30034.........................SQLSTATE=42505 and SQLCODE=-922
.at COM.ibm.db2os390.sqlj.jdbc.DB2SQLJConnection.setError(DB2SQLJConnection.java:1551)
.at COM.ibm.db2os390.sqlj.jdbc.DB2SQLJConnection.<init>(DB2SQLJConnection.java:335)
.at
COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver.createCicsImsConnection(DB2SQLJDriver.java:1230)
.at COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver.connect(DB2SQLJDriver.java:1152)
.at java.sql.DriverManager.getConnection(DriverManager.java:537)
.at java.sql.DriverManager.getConnection(DriverManager.java:210)
.atitso.ejb390.trader.TraderBackendDB2JDBC.openConnection(TraderBackendDB2JDBC.java:290)
.at itso.ejb390.trader.TraderBackendDB2JDBC.ejbCreate(TraderBackendDB2JDBC.java:104)
.at itso.ejb390.trader.TraderBean.ejbCreate(TraderBean.java:40)
.at itso.ejb390.trader.EJSTraderHomeBean.create(EJSTraderHomeBean.java:40)
.at itso.ejb390.trader.EJSRemoteTraderHome.create(EJSRemoteTraderHome.java:35)
.atitso.ejb390.trader._EJSRemoteTraderHome_Tie._invoke(_EJSRemoteTraderHome_Tie.java:81)
.at com.ibm.rmi.corba.ServerDelegate.dispatch(ServerDelegate.java:284)
.at com.ibm.rmi.iiop.ORB.process(ORB.java:263)
.at com.ibm.rmi.iiop.IIOPConnection.doWork(IIOPConnection.java:1341)
.at com.ibm.cics.iiop.orb.CICSConnection.processRequest(CICSConnection.java:329)
.at com.ibm.cics.iiop.RequestProcessor.processNormalMode(RequestProcessor.java:388)
.at com.ibm.cics.iiop.RequestProcessor.main(RequestProcessor.java:170)
.at java.lang.reflect.Method.invoke(Native Method)
.at com.ibm.cics.server.Wrapper.call_main(Wrapper.java:415)
.at com.ibm.cics.server.Wrapper.callUserClass(Wrapper.java:551)
.at  com.ibm.cics.server.Wrapper.main(Wrapper.java:832)
```

These types of errors also produce associated CICS messages in the CICS joblog. The CICS messages for this example problem are shown in Example 5-21. This shows a CICS abend code of AD2U, which also provides hints about what has happened. The text for the AD2U abend code is “An attempt to create a DB2 thread by the subtask servicing the DB2 request failed”.

Example 5-21 CICS abend messages associated with a DB2 exception

```
DFHDU0203I 04/11/2001 14:28:01 SCSCPJA5 A transaction dump was taken for dumpcode: AD2U,
Dumpid: 1/0001.
DFHAC2248 04/11/2001 14:28:01 SCSCPJA5 Transaction CIRP running program DFJIIRP term
???? has failed with abend ASP7 following the failure of a local resource owner in the
prepare phase of syncpoint. Updates will be backed out
```

No select privilege to access table TRADER_COMPANY

Example 5-22 on page 130 shows an exception caused by a DB2 authorization problem. From the exception the SQL error code can be seen, in this example, **-551**. The DB2 manual, *DB2 Messages and Codes*, GC26-9011, contains information on SQL error codes:

-551 *auth-id* DOES NOT HAVE THE PRIVILEGE TO PERFORM OPERATION *operation* ON OBJECT *object-name*

The Java exception provides the values for *auth-id*, *operation*, and *object-name,* which in this example are CICSRS1, SELECT, and ITSOEJB.TRADER_COMPANY respectively. From this, we can determine that the error is due to the user CICSRS1 not having DB2 SELECT authorization for the TRADER_COMPANY table. These authorizations are described in 10.3.5, “Granting privileges to the CICS user ID” on page 298.

Example 5-22 No select privilege to access table TRADER_COMPANY

```
java.sql.SQLException: DB2JDBCCursor Received Error in Method prepare:SQLCODE==> -551
SQLSTATE ==> 42501 Error Tokens ==> <<DB2 6.1 ANSI SQLJ-O/JDBC 1.0>> CICSRS1
SELECT ITSOEJB.TRADER_COMPANY
.at COM.ibm.db2os390.sqlj.jdbc.DB2SQLJJDBCCursor.setError(DB2SQLJJDBCCursor.java:938)
.at COM.ibm.db2os390.sqlj.jdbc.DB2SQLJJDBCSection.prepare(DB2SQLJJDBCSection.java:576)
.at COM.ibm.db2os390.sqlj.jdbc.DB2SQLJStatement.executeQuery(DB2SQLJStatement.java:497)
.at itso.ejb390.trader.TraderBackendDB2JDBC.logon(TraderBackendDB2JDBC.java:253)
.at itso.ejb390.trader.TraderBean.logon(TraderBean.java:117)
.at itso.ejb390.trader.EJSRemoteTrader.logon(EJSRemoteTrader.java:153)
.at itso.ejb390.trader._EJSRemoteTrader_Tie._invoke(_EJSRemoteTrader_Tie.java:115)
.at com.ibm.rmi.corba.ServerDelegate.dispatch(ServerDelegate.java:284)
.at com.ibm.rmi.iiop.ORB.process(ORB.java:263)
.at com.ibm.rmi.iiop.IIOPConnection.doWork(IIOPConnection.java:1341)
.at com.ibm.cics.iiop.orb.CICSConnection.processRequest(CICSConnection.java:329)
.at com.ibm.cics.iiop.RequestProcessor.processNormalMode(RequestProcessor.java:388)
.at com.ibm.cics.iiop.RequestProcessor.main(RequestProcessor.java:170)
.at java.lang.reflect.Method.invoke(Native Method)
.at com.ibm.cics.server.Wrapper.call_main(Wrapper.java:415)
.at com.ibm.cics.server.Wrapper.callUserClass(Wrapper.java:551)
.at  com.ibm.cics.server.Wrapper.main(Wrapper.java:832)
```

Plan TRADERP is not specified in CICS DB2CONN definition

Our final problem example is shown in Example 5-23. In this example, the SQL exception shows an SQL error code of -805. Again the DB2 manual, *DB2 Messages and Codes*, GC26-9011, contains information about this SQL error, which indicates that the DBRM or package name is not found in the DB2 plan.

In this example, the reason code, 02, which is the two digit number after ‘DSNJDBC’ in the exception text, indicates the problem:

02 The DBRM name 'dbrm-name' did not match an entry in the member list or the package list.

This problem was caused by the CICS DB2CONN resource definition not specifying the correct DB2 plan name. Setting up the CICS DB2 definitions is discussed in 10.2.5, “Defining a CICS DB2 connection” on page 280.

Example 5-23 No select privilege to access table TRADER_COMPANY

```
java.sql.SQLException: DB2JDBCCursor Received Error in Method staticDescribe:SQLCODE==>
-805 SQLSTATE ==> 51002 Error Tokens ==> <<DB2 6.1 ANSI SQLJ-0/JDBC 1.0>>
DBZ1..TRADER2.000000E59D0552F7 DSNJDBC 02
.at COM.ibm.db2os390.sqlj.jdbc.DB2SQLJJDBCCursor.setError(DB2SQLJJDBCCursor.java:938)
.at COM.ibm.db2os390.sqlj.jdbc.DB2SQLJJDBCCursor.staticDescribe(DB2SQLJJDBCCursor.java:448)
.at
COM.ibm.db2os390.sqlj.runtime.DB2SQLJRTStatement.executeRTQuery(DB2SQLJRTStatement.java:111
2)
.at sqlj.runtime.ExecutionContext$StatementFrame.executeQuery(ExecutionContext.java:734)
.at sqlj.runtime.ExecutionContext.executeQuery(ExecutionContext.java:362)
.at itso.ejb390.trader.TraderBackendDB2SQLJ.logon(TraderBackendDB2SQLJ.java:558)
.at itso.ejb390.trader.TraderBean.logon(TraderBean.java:117)
.at itso.ejb390.trader.EJSRemoteTrader.logon(EJSRemoteTrader.java:153)
.at itso.ejb390.trader._EJSRemoteTrader_Tie._invoke(_EJSRemoteTrader_Tie.java:115)
.at com.ibm.rmi.corba.ServerDelegate.dispatch(ServerDelegate.java:284)
.at com.ibm.rmi.iiop.ORB.process(ORB.java:263)
.at com.ibm.rmi.iiop.IIOPConnection.doWork(IIOPConnection.java:1341)
.at com.ibm.cics.iiop.orb.CICSConnection.processRequest(CICSConnection.java:329)
.at com.ibm.cics.iiop.RequestProcessor.processNormalMode(RequestProcessor.java:388)
.at com.ibm.cics.iiop.RequestProcessor.main(RequestProcessor.java:170)
.at java.lang.reflect.Method.invoke(Native Method)
.at com.ibm.cics.server.Wrapper.call_main(Wrapper.java:415)
.at com.ibm.cics.server.Wrapper.callUserClass(Wrapper.java:551)
.at  com.ibm.cics.server.Wrapper.main(Wrapper.java:832)
```

Part 3

CICS TS V2.1: Enterprise bean scenarios

In this part we document five different scenarios where we developed and deployed enterprise beans in CICS. We start with the initial step of creating a simple HelloWorld application in the VisualAge for Java Development environment, and then move on to creating a stateful session bean called TraderBean that wraps the existing pseudo-conversational COBOL Trader application. Following this we provide details on how to develop new Java versions of COBOL applications using either the JCICS classes, or the SQLJ and JDBC interfaces. We also provide details on how we developed both a stand-alone Java test client and a sample JSP/servlet application to invoke the TraderBean, as well as information on how to deploy this into WebSphere Application Server for Windows NT.

6

Developing a HelloWorld session bean for CICS

In this chapter we explain how to develop, deploy, and test a HelloWorld stateless session bean under CICS TS V2.1. We used a Java client to test the enterprise bean from within VisualAge for Java (VAJ), as well as a standalone application from the Windows NT and the OS/390 UNIX System Services (USS) environment. Our scenario is illustrated in Figure 6-1.

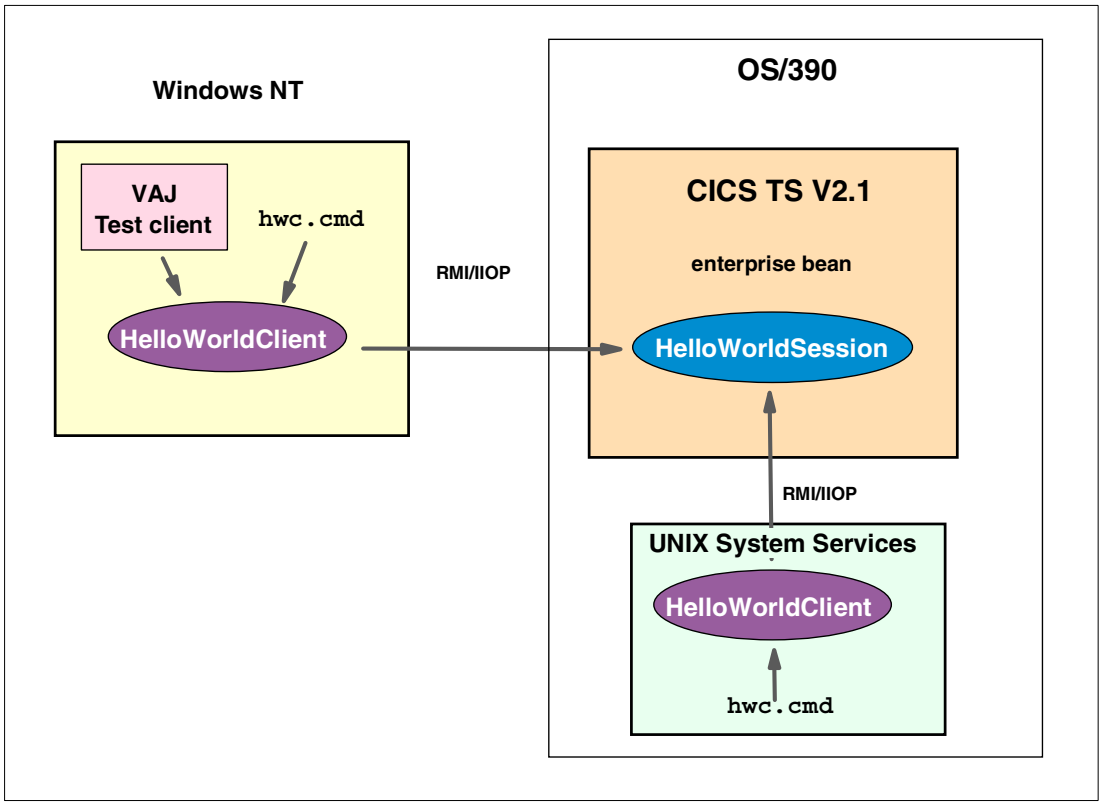


Figure 6-1 CICS Helloworld session bean

6.1 Quick start — Invoking HelloWorldBean

If you want to run our sample HelloWorld enterprise bean without following all the details specified in the following sections of this chapter you can follow the steps below. All the source code and examples used in this book are available for download from the redbooks Web site ftp://www.redbooks.ibm.com/redbooks/. For full details of the available files, refer to Appendix C, “Using the additional material” on page 315.

1. Create a CICS TCPIPSERVICE, CORBASERVER, and DJAR definition in your CICS TS V2.1 region if you have not already done so. For more details refer to 6.3.3, “Deploying to CICS” on page 150.
2. Deploy the HelloWorld deployed JAR file `hws_GEN.jar` to your CICS region. For more information refer to 6.3.3, “Deploying to CICS” on page 150.
3. On your workstation create a directory (for example, `c:\itsohelloworld`) and copy the following files, supplied with this redbook, to this directory:

   ```
   hws_CLI.jar
   hwc.jar
   hwc.cmd
   hwc_nt.properties
   ```

4. Ensure that you have a Java 2 runtime environment version 1.3.0 (or higher) installed on your workstation. You can verify your version with the command:

   ```
   java -version
   ```

5. Ensure that you have file `j2ee.jar` accessible on your workstation. If not, you can either obtain it by installing the CICS development deployment tool or by installing the Java 2 SDK Enterprise Edition available from http://java.sun.com.
6. Edit file `hwc.cmd` and make the following changes:
   a. In the line that defines the JAVA_HOME variable, change the path to match your IBM SDK 1.3 installation directory. The sample provided assumes the directory to be `C:\PROGRA~1\IBM\Java13`.
   b. In the line invoking defining `JAVA_J2EE`, change the directory to the location of the file `j2ee.jar` on your worksation. The sample provided assumes the file to be in the directory `C:\Program Files\IBM\CICS TS 2.1 Tools\Common`.
7. Edit file `hwc_nt.properties` and change the properties for the provider URL and the JNDI name to match your requirements. For more information refer to “Dependencies of the client code” on page 162.
8. Invoke the `hwc.cmd` file from a Windows command prompt, the output should be as shown in Example 6-7 on page 168.

6.2 Developing a HelloWorld session bean with VAJ

VAJ Version 3.5 enables you to easily develop enterprise beans. It includes a WebSphere Test Environment (WTE), which provides server run-time support for testing and debugging enterprise beans.

For developing the enterprise beans described in this book, we used IBM VisualAge for Java, Enterprise Edition V3.5. Note that you need, at a minimum, the features shown in the list below to be able to use VAJ as a development environment for enterprise beans.

- IBM EJB Development Environment 3.5
- IBM Enterprise Extension Libraries 3.5
- IBM WebSphere Test Environment 3.5.0.2

If not already installed on your version of VAJ, you should install these features in VAJ by selecting **File -> Quickstart -> Features -> Add Feature**, choosing the features from the list, and clicking **OK**.

6.2.1 Developing in VAJ

The following list gives an overview of the basic steps to perform in VAJ in order to develop an enterprise bean:

1. **Add a project:** Projects are used in VAJ to contain and organize Java packages in the workspace and in the repository.
2. **Add a package:** Packages are used in Java to organize the Java code. Java code that is part of a particular package has access to all classes in that package, and to all non-private methods and fields in all those classes.
3. **Add an EJB group:** EJB groups are logical groups in VAJ that allows you to organize your enterprise beans.
4. **Add an enterprise bean to your EJB group:** Enterprise beans are server-side components that implement a business entity or business task.
5. **Add a business method to your enterprise bean:** Business methods are specific to the business concept of the enterprise bean. They represent the task that the bean performs.
6. **Add a business method to the remote interface:** Enterprise beans are accessed by a client application over the network through their remote and home interfaces.

Add a project

After starting VAJ the first step is to add a new project, which is a logical program element only available within VAJ. It contains all the packages used for a particular work unit, such as an entire application. To add a project from the VAJ Workbench (Figure 6-2), choose **Selected -> Add -> Project**.

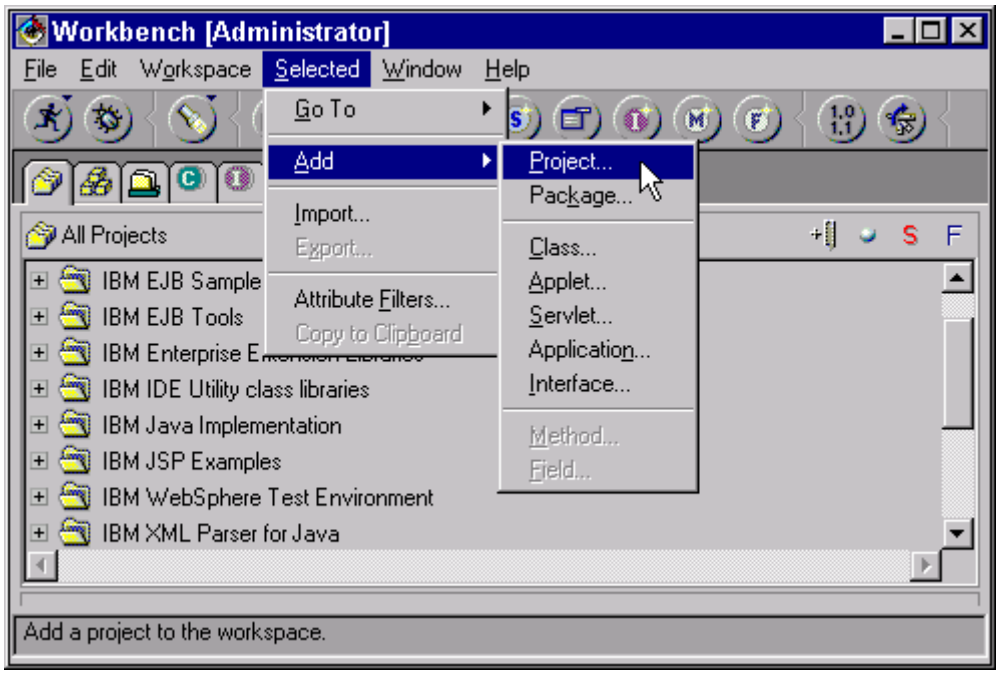


Figure 6-2 VAJ add a project

The *Add Project SmartGuide* appears, which is then used to add a project. To create a new project, select the **Create a new project named** radio button, type in the project name, `ITSO EJB 390 Redbook`, and click **Finish**.

Add a package

To add a new Java package to the project, select the project, ITSO EJB 390 Redbook, and choose **Selected -> Add -> Package**. The *Add Package SmartGuide* appears, which is illustrated in Figure 6-3.

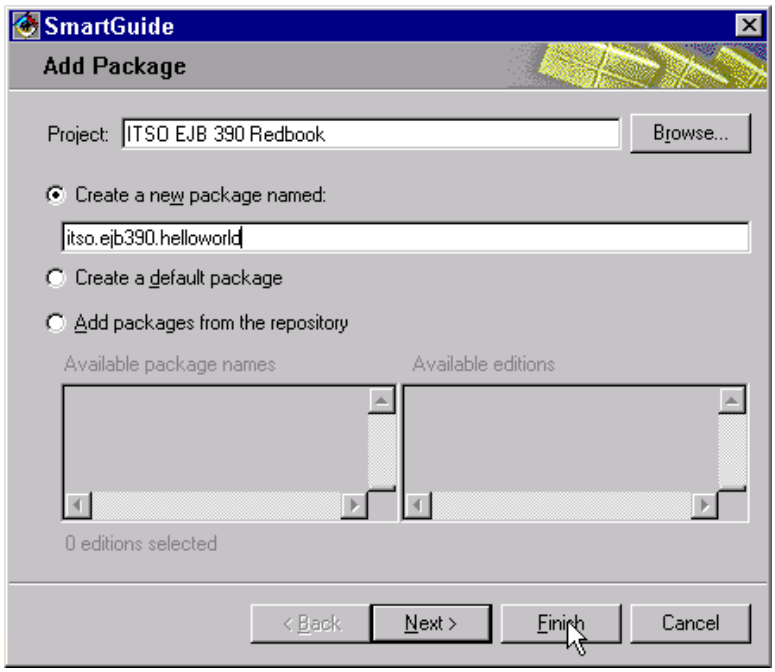


Figure 6-3 VAJ Add Package SmartGuide

Within this page you can create a new package, add a package from the repository, or create a default package. To create a new package, select the **Create a new package named:** radio button, enter the package name, *itso.ejb390.helloworld*, and click **Finish**.

Add an EJB group

Enterprise beans are logically organized in EJB groups within VAJ. You can perform global operations on an EJB group that will iterate on all of the enterprise beans that reside in the group. To add an EJB group we selected the EJB tab in the Workbench window, and clicked on **EJB -> Add -> EJB Group**. The *Add EJB Group SmartGuide* appears, which allows you to create a new EJB group or add an EJB group from the repository. To add a new EJB group, select the project, ITSO EJB 390 Redbook, and select the **Create a new EJB group named:** radio button. Now type in the EJB group name, `ITSOEJB390`, in the corresponding input field and click **Finish**.

Add an enterprise bean to your EJB group

An enterprise bean consists of a number of Java classes and interfaces. VAJ helps you to keep these classes synchronized as a single entity. To add an enterprise bean, select the EJB group, ITSOEJB390, and select **EJB -> Add -> Enterprise Bean**. The *Create Enterprise Bean SmartGuide* appears, as illustrated in Figure 6-4, where you can either create a new enterprise bean, or add enterprise beans from the repository. To create a new enterprise bean, select the **Create a new enterprise bean** radio button as well as the proper project, ITSO EJB 390 Redbook, and package, itso.ejb390.helloworld. Now type in the name of the enterprise bean, `HelloWorldSession`, and select session bean in the **Bean type** field (since CICS TS V2.1 supports only session beans). Finally, click **Finish**.

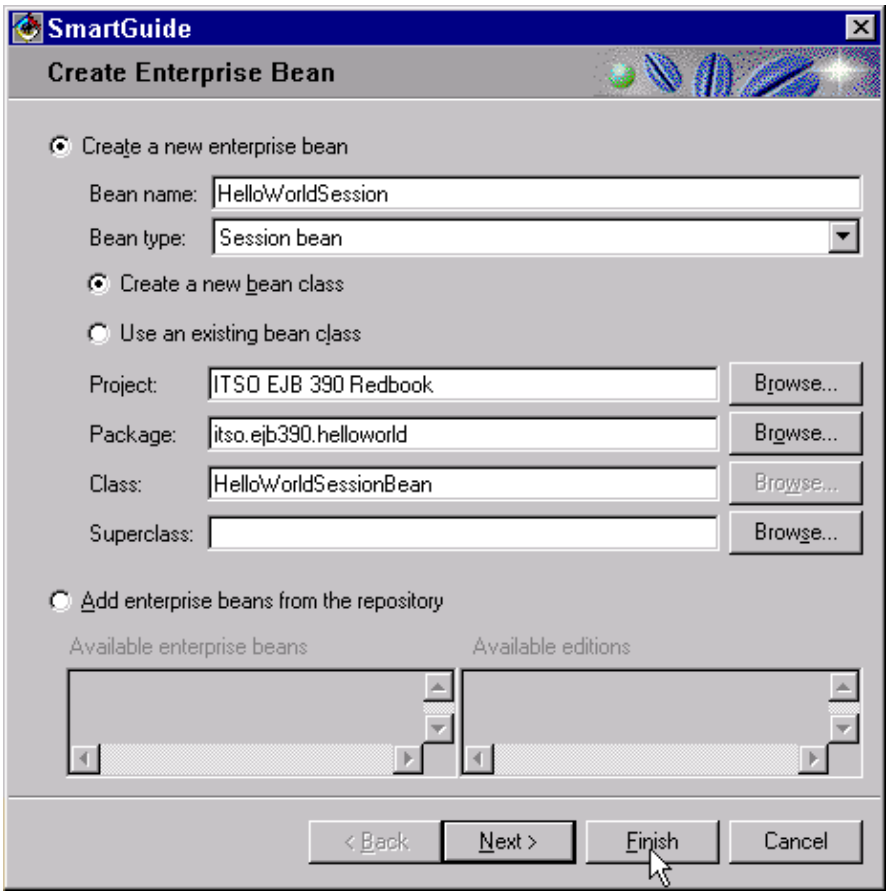


Figure 6-4 VAJ Create Enterprise Bean SmartGuide

VAJ creates now all the classes and interfaces required for the enterprise bean, such as the enterprise bean class and the remote and home interfaces. At this point in time the remote interface contains no methods, since VAJ does not know what business method should be exposed to the client.

Add a business method to your enterprise bean

The remote interface defines the bean's business purpose. To add a business method to the remote interface of the session bean, right-click the *HelloWorldSession* bean implementation in the Types panel and select **Add -> Method**, as shown in Figure 6-5.

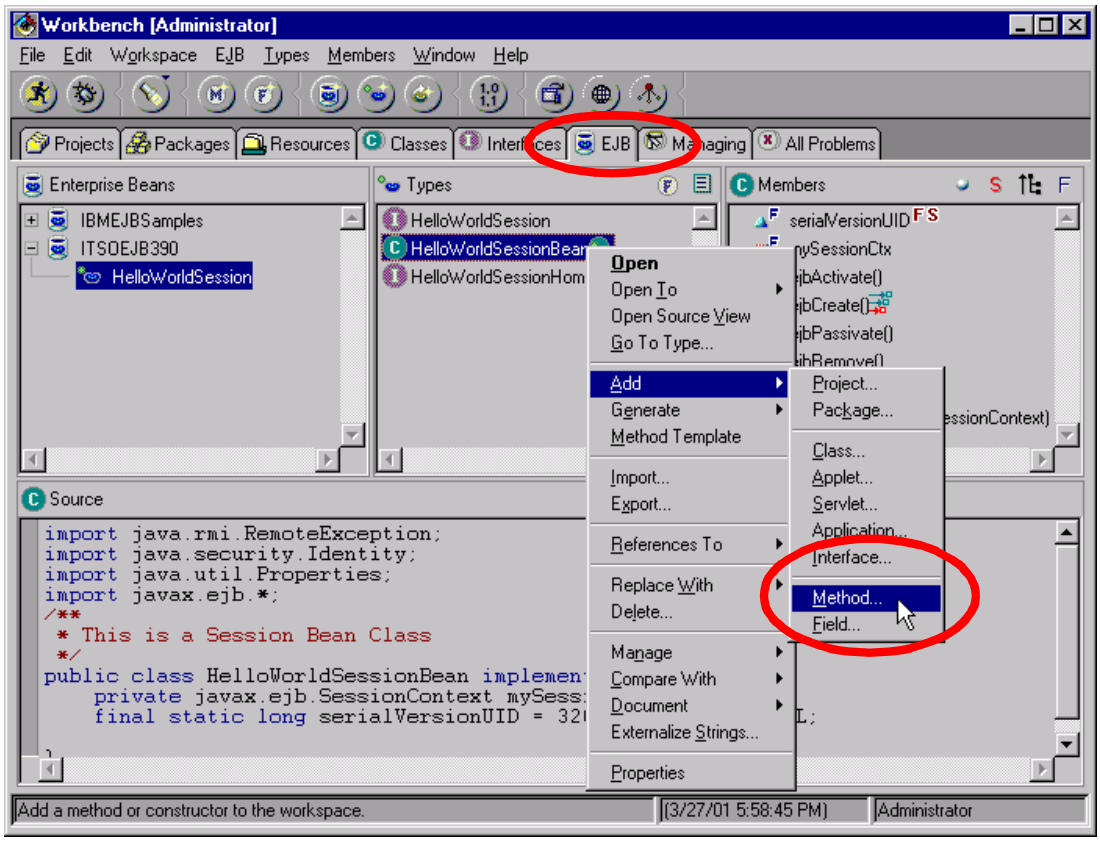


Figure 6-5 VAJ add a business method

The *Create Method SmartGuide* appears (Figure 6-6), which lets you create a new method, constructor, or main method, or add an existing method from the repository.

To create a new method, select the **Create a new method** radio button and type in the signature of the method, `public String sayHello(String msg)`. Now click the **Next** button. The *Attributes SmartGuide* appears, which allows you to fine tune or change the characteristics of your method signature. Click the **Next** button. The *Attributes SmartGuide* appears again, and allows you to specify the exceptions your method might throw. Click **Finish**.

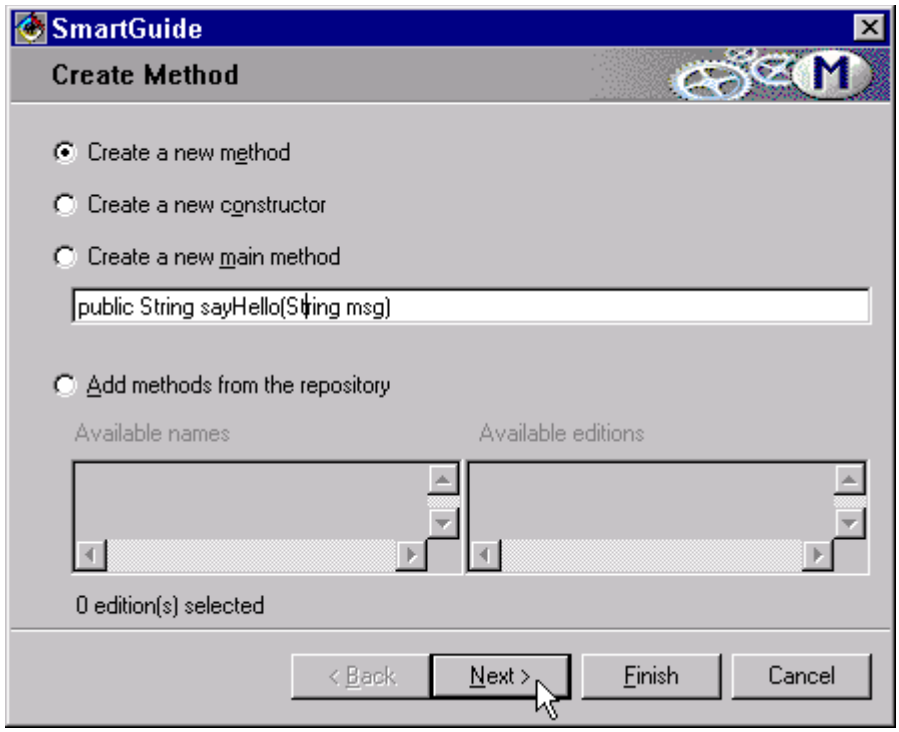


Figure 6-6 VAJ Create Method SmartGuide

The message `sayHello()` appears in the Members panel. Modify the method code, as illustrated in Figure 6-7. The Java print statement will be used to echo the input back to the user. The return statement returns the argument that was passed to the method. Press Ctrl-s to save the code.

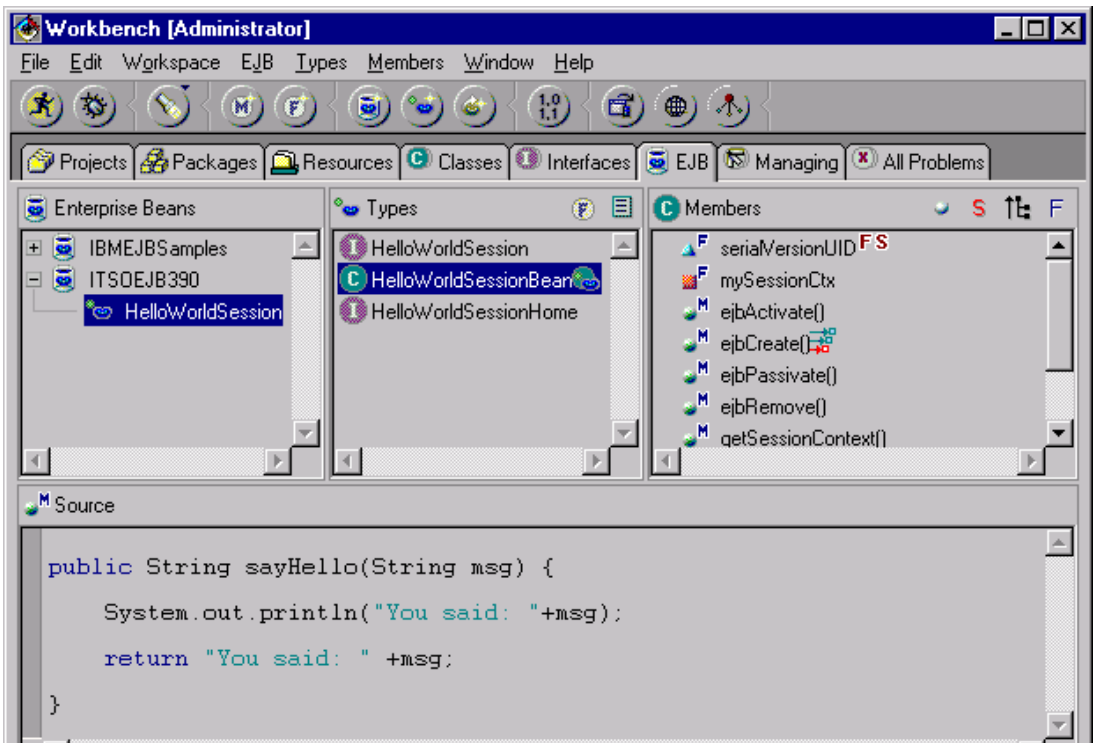


Figure 6-7 VAJ Change implementation of business method in source panel

Add a business method to the remote interface

The remote and home interface expose the capabilities of the bean and provide all the methods needed to create, update, interact and delete with the bean. So in order to have the business method available to a client, you must add it to the remote interface. To do this, right-click the method `sayHello()` in the Members panel, and select **Add To -> EJB Remote Interface**. After this step a new symbol (showing three arrows, where the top one is red) appears to the right of the method name, indicating a remote method.

6.2.2 Testing in VAJ

To test the HelloWorldSession bean, you can use the Visual Age for Java WebSphere Test Environment, which is a subset of the WebSphere Application Server, Advanced Edition. Testing the bean within VAJ on your Workstation requires the following steps:

1. **Generate deployed code:** This step generates the code (such as stubs and ties) required to tailor a generic enterprise bean definition into one that can run in an EJB server.
2. **Start the WebSphere Test Environment (WTE):** The WTE offers the JSP file, servlet, and EJB runtime and unit testing environment.
3. **Start the Persistent name server (J:NDI server).** The Persistent Name Server provides the JNDI services in the WTE, just as the COS Naming Server is used by WebSphere Application Server, Advanced Edition.
4. **Add the enterprise bean to an EJB Server Configuration:** An EJB server configuration consists of one or more EJB groups containing enterprise beans which you want to run on the EJB server.
5. **Start the EJB server:** The EJB server provides a runtime environment for one or more EJB containers. The EJB container provides a playground where your enterprise beans can run.
6. **Use the WebSphere Test Client to test your enterprise bean:** This test client provides its own user interface and allows you to test the individual methods in the home and remote interface of each enterprise bean.

Generate deployed code

The code-generation tool of VAJ generates the home and EJBObject (remote) implementations and implementation classes for the home and remote interfaces, as well as the JDBC persister and finder classes for CMP beans. It also generates the Java ORB, stubs, and tie classes required for RMI access over IIOP, as well as stubs for the home and remote interfaces. To generate the code for the enterprise bean, select the **EJB** tab in the VAJ Workbench, right-click the EJB group, *ITSOEJB390*, and select **Generate Deployed Code**.

Start the WebSphere Test Environment

In order to start the WebSphere Test Environment select **Workspace -> Tools -> WebSphere Test Environment**. This will bring up the WebSphere Test Environment Control Center (Figure 6-8) which provides a central location to start, stop, and configure the WebSphere Test Environment services, such as the Persistent Name Server.

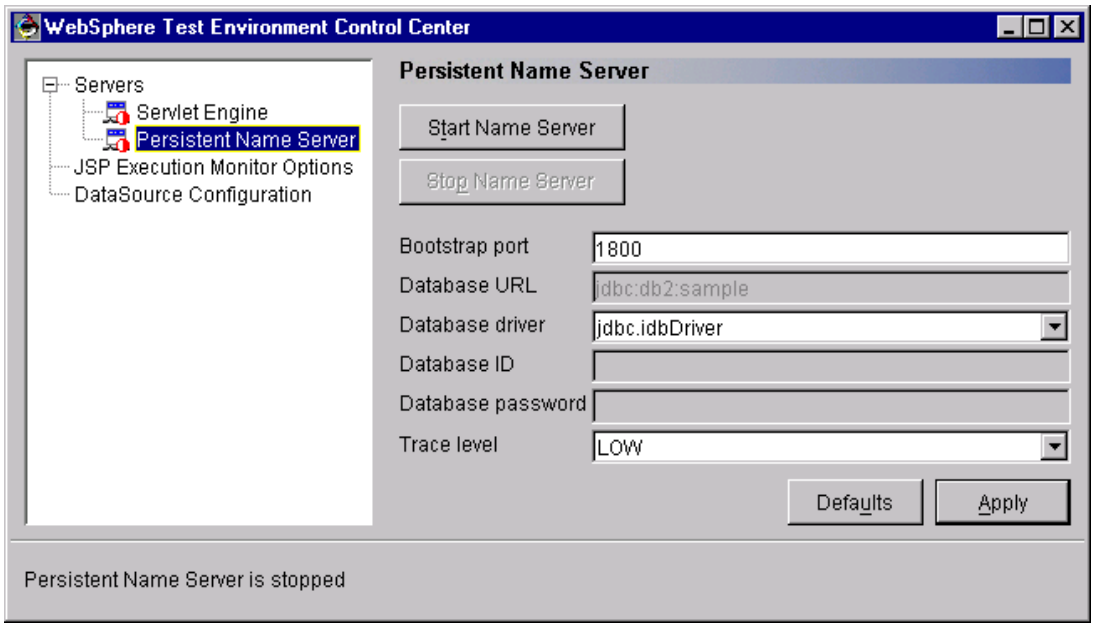


Figure 6-8 VAJ WebSphere Test Environment Control Center

Start the Persistent Name Server (JNDI server)

JNDI is a standard Java extension for accessing naming systems such as LDAP, NetWare, the WebSphere COS Naming Server, and other Naming systems. EJB servers support JNDI by organizing beans into a directory structure and providing a JNDI driver for accessing that directory structure. In the WebSphere Test Environment the JNDI services are provided by the Persistent Name Server. To start that server, select the Persistent Name Server within the WebSphere Test Environment Control Center and click the **Start Name Server** button.

If you have installed the WebSphere Application Server on you NT machine (either standard or advanced edition), the port 900 will already be in use by the WebSphere Application Server COS Naming Server. In this case, if you use the default port 900 to start the Persistent Name Server, a window will appear showing the following message:

```
Specified port is already in use. The server cannot be started.
```

Instead, you will have to specify a different port in the **Bootstrap port** input field. We used the port 1800. Ensure that you apply your changes by clicking the **Apply** button in the WebSphere Test Environment Control Center.

Attention: The JNDI Server provided by the VAJ WebSphere Test Environment is completely separate from the JNDI Server implemented in the WebSphere Application Server COS Naming Server (which is used by CICS).

Messages written during the startup process of the Persistent Name Server are directed to the VAJ Console. If not started before, a new console window will automatically appear, as shown in Figure 6-9.

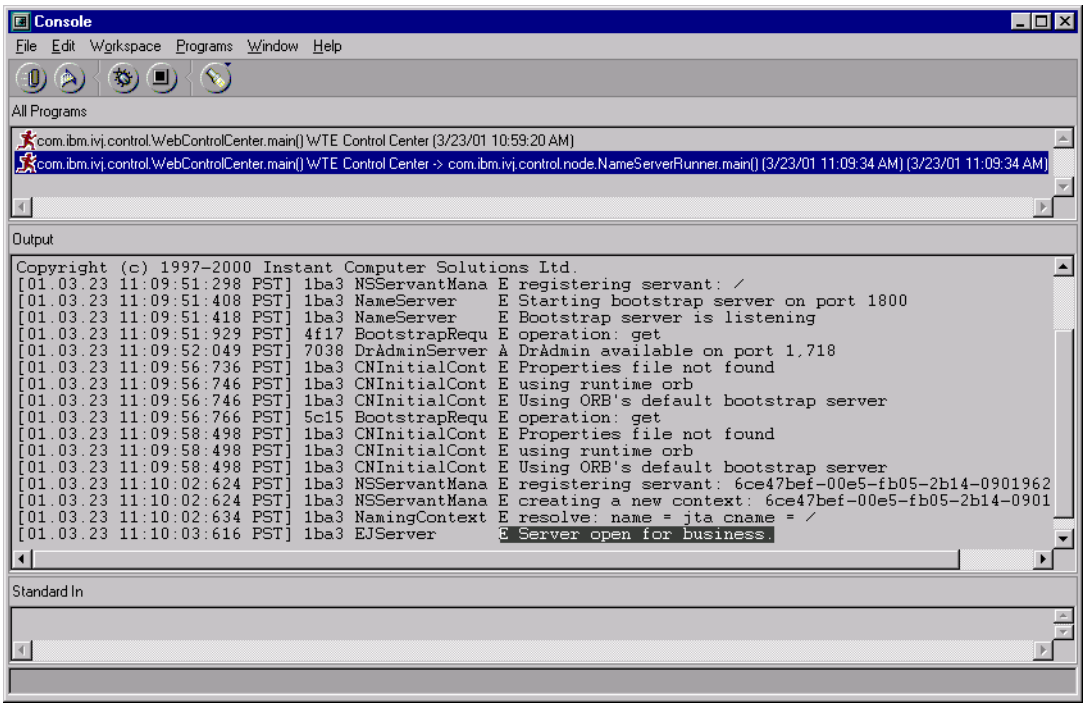


Figure 6-9 VAJ Console, after starting the Persistent Name Server

Add the enterprise bean to an EJB Server Configuration

In order to add our enterprise bean, HelloWorldSession, to an EJB Server Configuration, select the **EJB** tab in the VAJ Workbench, right-click the EJB Group, *ITSOEJB390,* and select **Add To -> Server Configuration**. Then the EJB Server Configuration window appears.

Start the EJB Server

During the start process of the enterprise server the enterprise beans which belong to the EJB Server configuration are loaded. The enterprise server will then publish the location of the enterprise beans into the JNDI server (Persistent Name Server). To start the EJB Server, right-click the *EJB Server* in the left panel of the EJB Server Configuration window, and select **Start Server**. The message in the VAJ Console, `Server open for business`, indicates that the EJB Server startup is complete.

Use the EJB Test Client to test the enterprise bean

The easiest way to test an enterprise bean is to use the EJB Test Client supplied with VAJ. This test client features its own user interface and allows you to test the individual methods in the home and remote interface of each enterprise bean. To start the EJB Test Client, select the **EJB** tab in the VAJ Workbench, right- click on the EJB Group, ITSOEJB390, and select **Run Test Client**. Then the EJB Test Client window appears together with the EJB Lookup window, as illustrated in Figure 6-10.

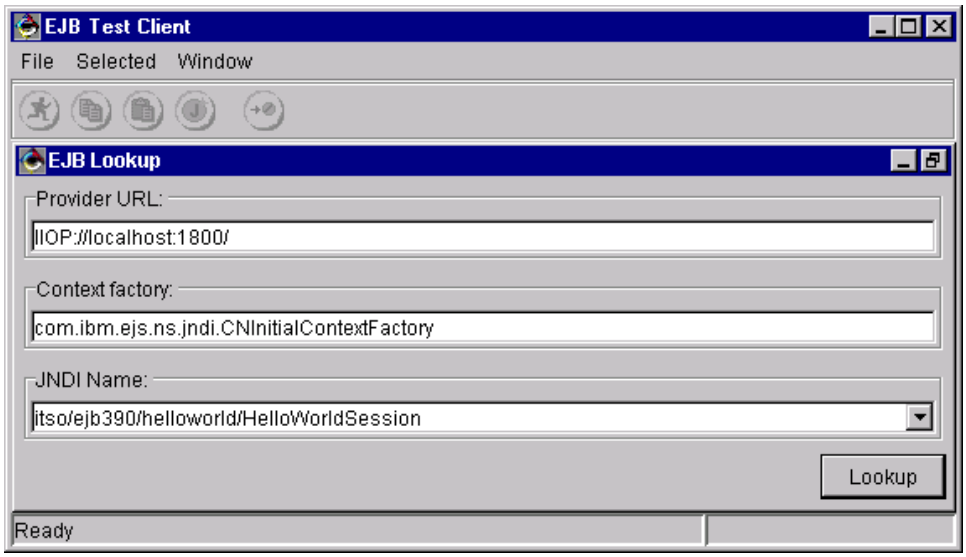


Figure 6-10 VAJ Lookup window of the EJB Test Client

In the EJB lookup window you will see the entries used for the Provider URL, Context factory, and JNDI name.

- **Provider URL:** The provider URL consist of the name of the local host and the port(1800). This URL defines the name server that will be used to lookup enterprise beans. The port number must match the port that was specified in the WebSphere Test Environment Control Center in the **Bootstrap port** field, as illustrated in Figure 6-8 on page 143.
- **Context factory:** The initial context is the starting point for any JNDI lookup, similar in concept to the root of a file system. To acquire an initial context, you use an initial context factory. For the VAJ WebSphere Test Environment, the context factory is always *com.ibm.ejs.ns.jndi.CNInitialContextFactory*.
- **JNDI name:** The JNDI name is used when publishing the location of the enterprise into the JNDI server. A client uses the JNDI name to retrieve the home object of the enterprise bean. The default JNDI name for enterprise beans created in VAJ is the fully-qualified name of the remote interface, with periods replaced by forward slashes. For our sample it is the compound name of the package name, *itso/ejb390/helloworld*, and the name of the remote interface, *HelloWorldSession*.

Click **Lookup**. The EJB Test Client now creates a reference to the JNDI server, performs a `lookup()` operation on the specified JNDI name and does a `narrow()` to get a reference to the home interface of the enterprise bean. An additional window appears, where you can test the individual methods in the home and remote interface of each enterprise bean. At this point in time, the window shows the methods of the home interface of the enterprise bean. To create an instance of the enterprise bean, right-click the `create()` method and select **Invoke**.

Now the window shows the methods of the remote interface of the enterprise bean. Select the business method, `sayHello()`, in the left panel, and specify the value for the argument in the input field on the right panel. To invoke the business method, right-click the method, and select **Invoke**, as shown in Figure 6-11.

Tip: Note that if you wish to use the EJB test client with a session bean deployed in CICS, you will need to add the class *com.ibm.cics.portable.CICSEJBMetaData* to your VAJ workspace. This class can be found in the deployed JAR file generated by the CICS JAR development tool.

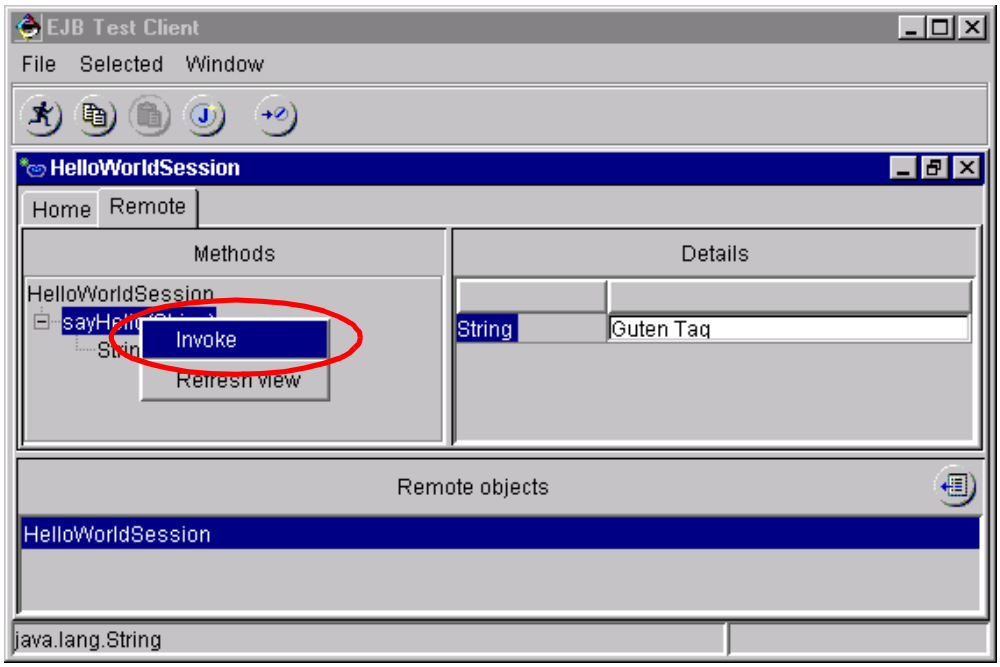


Figure 6-11 Invoke a remote method from the EJB Test Client

Now the value returned to the EJB Test Client appears in the right panel. In our case it is `You said: Guten Tag`. (See Figure 6-12.)

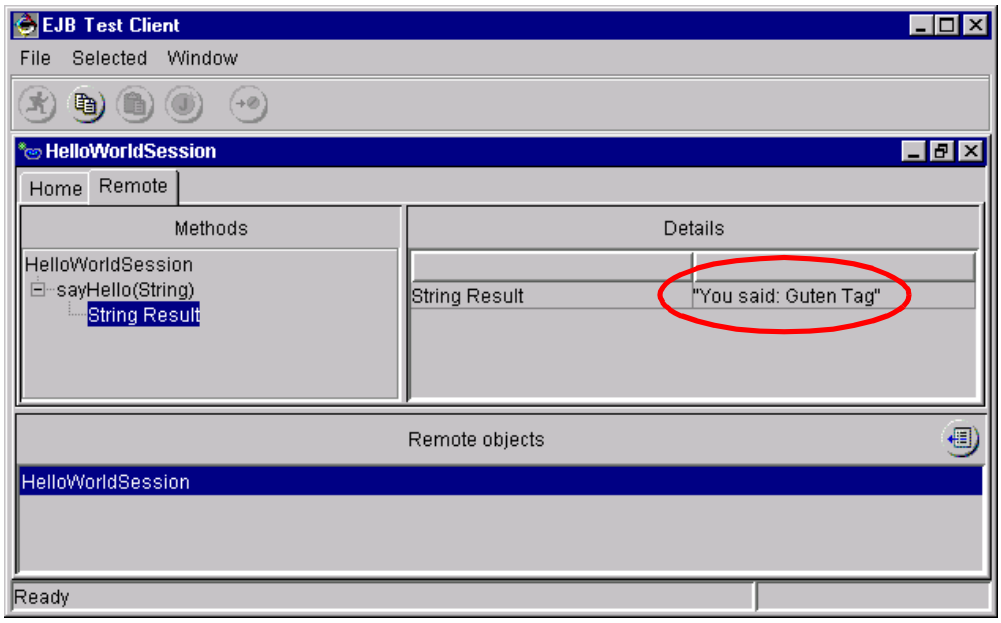


Figure 6-12 VAJ Invoke a remote method from the EJB Test Client — results

To stop the client, select **File -> Exit**.

6.3 Deploying the HelloWorld session bean to CICS

Deployment is the process to prepare an enterprise bean for the runtime environment and install it into the EJB server. This step will vary depending on the tools provided by the enterprise bean vendor. Here we describe the steps to deploy an enterprise bean developed with VAJ to a CICS EJB server.

6.3.1 Packaging an undeployed JAR file

Once you have generated the files that define the bean (the home and remote interfaces, the bean class, the bean's properties and the deployment descriptor) they have to be packaged up into one entity. This entity is called a JAR file. A JAR file is a compressed file that follows the ZIP compression format. JAR files serve as convenient, compact modules for shipping Java software.

To generate and export an EJB-JAR file containing the enterprise bean HelloWorldSession from within VAJ, select the EJB tab in the VAJ Workbench, expand the EJB group ITSOEJB390, right-click the enterprise bean *HelloWorldSession* and select **Export -> EJB JAR**. The *Export to an EJB JAR File SmartGuide* appears, which is illustrated in Figure 6-13. Type in the name of the EJB-JAR file in the **JAR file** input field and click the **Finish** button. We used `D:\itso\swl870\hws.jar` as our EJB-JAR file name.

Figure 6-13 VAJ Export to an EJB JAR File SmartGuide

Tip: If you receive a message from VAJ stating that *the file is not a zip file, or it is corrupted*, you should close the CICS Java development tool, or delete the output JAR file.

6.3.2 Generating a CICS deployed JAR file

The EJB-JAR file that was generated and exported from VAJ is an undeployed, EJB 1.0 specification level file. In order to install that file into an CICS EJB server, it must be transformed to a JAR file suitable for deployment. For CICS TS V2.1, an EJB 1.1 specification level JAR file is required. The transformed file must also contain additional code required to tailor a generic enterprise bean definition into one that can run in an EJB server.

CICS provides the *CICS JAR development tool for EJB technology* to facilitate this step. It provides a GUI interface that enables you to create or edit the EJB-JAR file's deployment descriptor together with some optional CICS specific customizations. It also converts EJB 1.0 serialized deployment descriptors into the EJB 1.1 XML equivalents. The CICS JAR development tool incorporates the *CICS code generation utility for EJB technology*. This tool automatically generates the code required to tailor a generic enterprise bean definition into one that can run in an EJB server. This code includes the CORBA stubs and ties needed for the RMI/IIOP communication. The code generation utility can also be run separately from a Windows command prompt window and can be used in a batch process. The result of this process is one or more EJB 1.1 specification level files. Refer to *Java applications in CICS*, SC34-5881, for more information on using the CICS deployment tools.

To run the CICS JAR development tool on your workstation, click the Windows **Start** button and select **Programs -> IBM CICS TS 2.1 Tools -> CICS JAR Development Tool for EJB Technology**. The CICS JAR development tool appears, which is illustrated in Figure 6-14. Select **File -> Load** and choose the name of the EJB-JAR file within the **Load From File** window. We chose our EJB-JAR file, `D:\itso\swl870\hws.jar`. Click the **Open** button. After loading the EJB-JAR file we saw our enterprise bean, HelloWorldSession, listed in the center panel and the following message in the status field near the bottom of the panel:

```
Finished reading the input file D:\itso\swl870\hws.jar
```

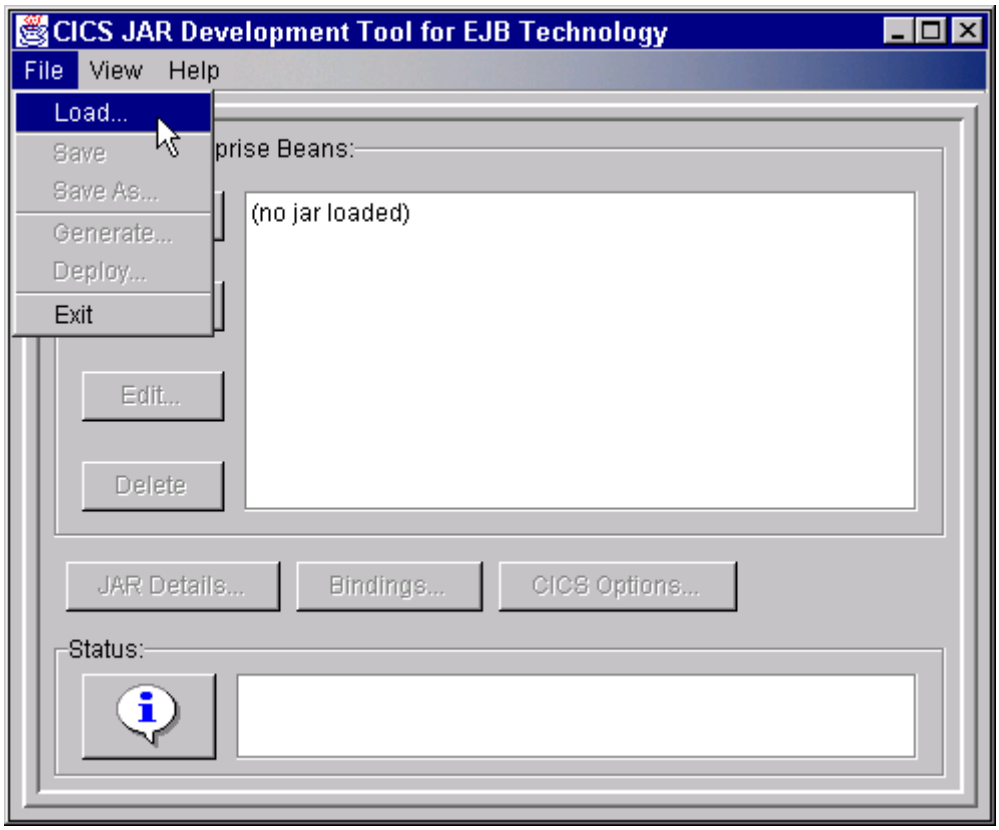


Figure 6-14 CICS JAR development tool

To look at the deployment descriptor information, select the enterprise bean, HelloWorldSession, and click the **Edit** button. This will display the window shown in Figure 6-15, where you can modify the deployment descriptor for the enterprise bean. For details on how we used this to set environment properties for JDBC in the deployment descriptor, refer to“Convert the exported file to a DJAR file” on page 294.

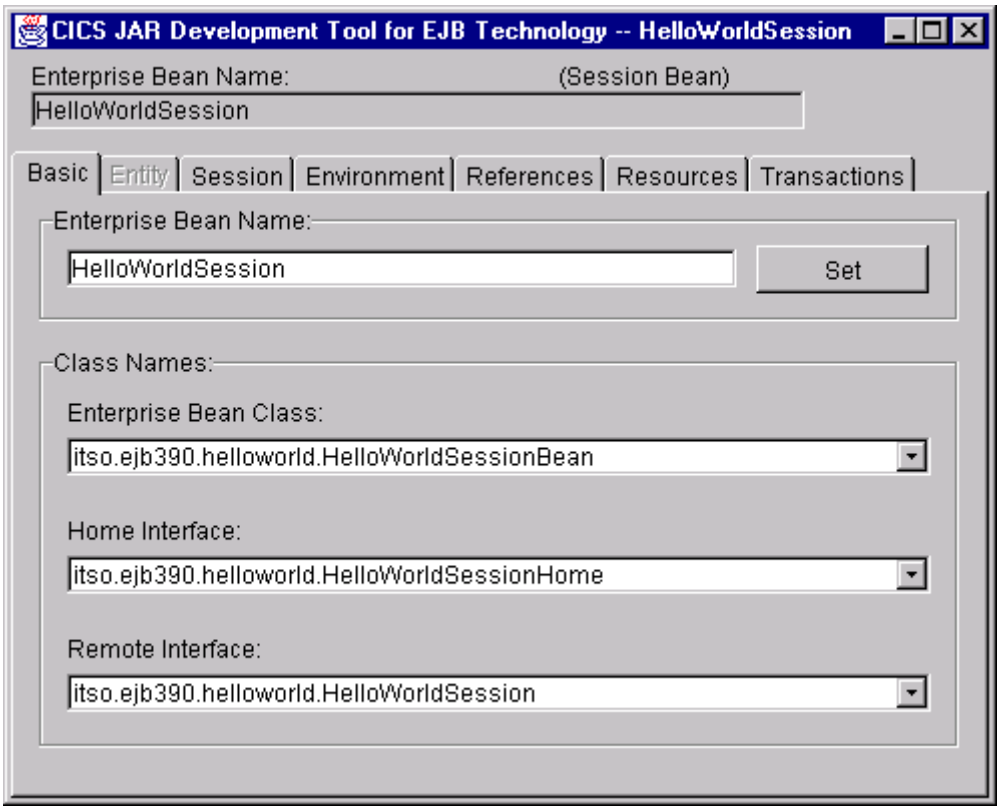


Figure 6-15 CICS JAR development tool — deployment descriptor

Close that window to return to the main CICS JAR development tool window.

To generate the container specific code for the EJB-JAR file perform the following steps:

1. Select **File -> Generate** and click the **Save** button in the window that ask you if you want to save the changes to a JAR file. This window always appears if you have used an EJB 1.0 specification level JAR file, or if you have changed the deployment descriptor.

   The tool then generates an XML version of the deployment descriptor.

2. Click the **Remove** button in the window that gives you the option of retaining the EJB 1.0 specific data.

   The Generate deployed code window appears as illustrated in Figure 6-16.

3. Select the **Verbose Output** and the **Create an EJB Client JAR File** radio buttons and click the **Generate** button.

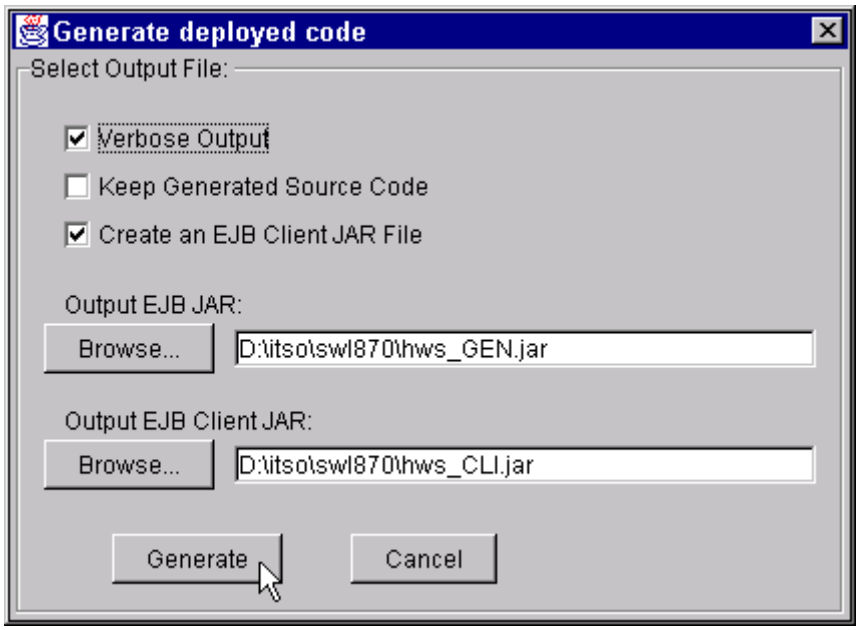


Figure 6-16 CICS JAR development tool — generate deployed code window

During the generate process, a window will pop up with the messages from the generate process. The message `Code generation completed` in the status panel indicates the deployed JAR file has been successfully generated. The generate process creates two new files, a deployed JAR file and a client JAR file. A client JAR file contains all classes and interfaces used to access enterprise beans.

6.3.3 Deploying to CICS

Once you have generated the deployed JAR file, it must be installed into a CICS EJB server. This step involves the creation of CICS resource definitions, publishing bean references to an external namespace, and making the EJB-JAR file accessible to CICS. CICS provides two alternative tools, the *CICS development deployment tool for EJB technology* and the *CICS production deployment tool for EJB technology*, to perform these operations. It is also possible to perform these operations manually. We shall describe each of these options in the following sections.

Deploying manually

We start by describing the manual way to deploy a EJB-JAR file to CICS. You may find this easier to understand than using the tools especially if you are familiar with CICS systems programming and using CEDA to define RDO resources.

First, before using CEDA, we enabled our 3270 terminal to handle mixed case input by using the following command:

```
CEOT TRANIDONLY
```

This is important, as it allows you to enter mixed case path names for HFS files.

Define CICS resource definitions

The following list gives an overview of the CICS resources you have to define as part of the final deployment step of an EJB-JAR file.

CORBASERVER A CICS resource definition that defines the attributes of an execution environment for enterprise beans and stateless CORBA objects. CORBASERVER definitions are installed in AORs.

DJAR A CICS resource definition that defines a CICS-deployed JAR file, which is a deployed JAR file, produced specifically for the CICS EJB server.

REQUESTMODEL A CICS resource definition that enables the request receiver to match incoming request to a CICS TRANSID, to define execution parameters that are used if a new request processor instance is created to handle that request.

TCPIPSERVICE A CICS resource definition that configures the CICS TCP/IP Listener to receive IIOP requests and to call the IIOP request receiver. It can also specify load-balancing and security options.

This sample assumes that you have already defined CORBASERVER, REQUESTMODEL and TCPIPSERVICE definitions as described in 4.1.5, “Installing CICS resource definitions” on page 77. It is important to note that the CORBASERVER defines the JNDIprefix, which is used as the prefix in the JNDI name space when the enterprise bean is published to the Naming Server. We used ITSO/PJA5 as our JNDIprefix attribute value.

Make the deployed JAR file accessible to CICS

Before defining a DJAR definition in CICS you should transfer the actual deployed JAR file to an HFS on your OS/390 system. We used FTP from a Windows command prompt to transfer the JAR file in binary to OS/390. We stored it in the HFS under the following path:

```
/u/cicsts21/djars/hws_GEN.jar
```

We then defined the deployed JAR file to CICS by creating a DJAR resource using the following command:

```
CEDA DEFINE DJAR(HWS) GROUP(ITSOEJB)
```

On the CEDA screen, which appears after entering this command, type in the values for the **Corbaserver** and the **Hfsfile**, as shown in Figure 6-17.

```
DEFINE DJAR(HWS) GROUP(ITSOEJB)
OVERTYPE TO MODIFY                                        CICS RELEASE = 0610
 CEDA  DEFine DJar( HWS      )
  DJar         ==> HWS
  Group        ==> ITSOEJB
  Description  ==> DJar CICS resource definitin for HelloWorld EJB
  Corbaserver  ==> PJA5
  Hfsfile      ==> /u/cicsts21/djars/hws_GEN.jar
               ==>
               ==>
               ==>
               ==>

 DEFINE SUCCESSFUL
                                                  SYSID=PJA5 APPLID=SCSCPJA5
```

Figure 6-17 CEDA define DJAR HWS

After defining the DJAR, it should be installed into the CICS region using the command:

```
CEDA INS DJAR(HWS) GROUP(ITSOEJB)
```

Installing a DJAR makes the deployed JAR file available in CICS, by copying it from the specified HFS file to the CORBASERVER subdirectory in the CICS shelf.

The enterprise bean is now available to be published to the COS Naming Server.

Publish names to the COS Naming Server

Publishing is the process which writes the IOR of the enterprise beans into the JNDI namespace. You can think of the JNDI namespace as the DNS server of the IIOP world, since this operation needs only to be performed once, even if the actual JAR file is modified. You perform this operation in CICS by using the command **CEMT PERFORM DJAR() PUBLISH**. For each bean installed from the named DJAR, an object reference (the IOR) is published to the COS Naming Server. We published our HelloWorld enterprise bean with the following command:

```
CEMT PERFORM DJAR(HWS) PUBLISH
```

Once an enterprise bean has been registered in the COS Naming Server command, a client application can use the JNDI interface to locate its home interface. Hence the enterprise bean is now available to be used. In order to unbind a home of an enterprise bean from the namespace, you can use the RETRACT option of the **CEMT PERFORM DJAR** command.

Tip: To query the contents of the JNDI Namespace in the WebSphere COS Naming Server you can use our sample Java utility class JNDIList. For details of how to use this refer to “Querying the COS Naming Server” on page 116

Deploying with the CICS development deployment tool

The CICS development deployment tool provides a route that simplifies the creation of the CICS resource definitions for application programmers with a minimum of CICS expertise. The tool transfers the EJB-JAR file in the HFS on OS/390 and creates a set of generic CICS resource definitions.

The interface provided to the Java developer is a standard HTML servlet interface, and so no knowledge of CICS RDO or the CEDA transaction is required. It is only necessary for a user to signon and then enter the name and location of the EJB-JAR file to be deployed, and the CICS CorbaServer into which the enterprise beans are deployed. However, before you can use this tool, the necessary components must be set up and the deployment configuration file must be correctly configured. This is described further in 4.2.4, “CICS development deployment tool” on page 86.

To start the CICS development deployment tool, first configure your Web browser so that it does not cache HTML pages, then enter the URL of the Web application. We entered:

```
http://hecate.almaden.ibm.com/CICS_EJB
```

The *User Login* page appears, where you enter your OS/390 user ID and password (Figure 6-18).

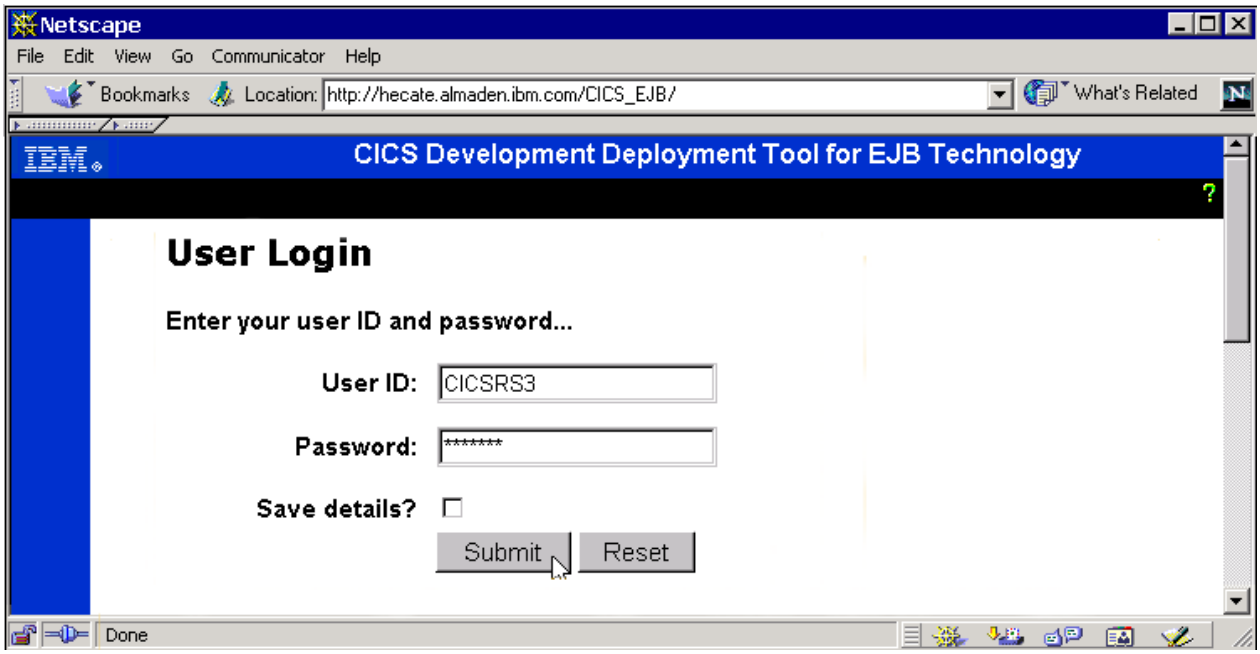


Figure 6-18 User Login page of CICS development deployment tool

Click **Submit** to proceed, which opens the *Deployment Information* page (Figure 6-19), this allows a JAR file to be deployed or undeployed in a CICS CorbaServer.

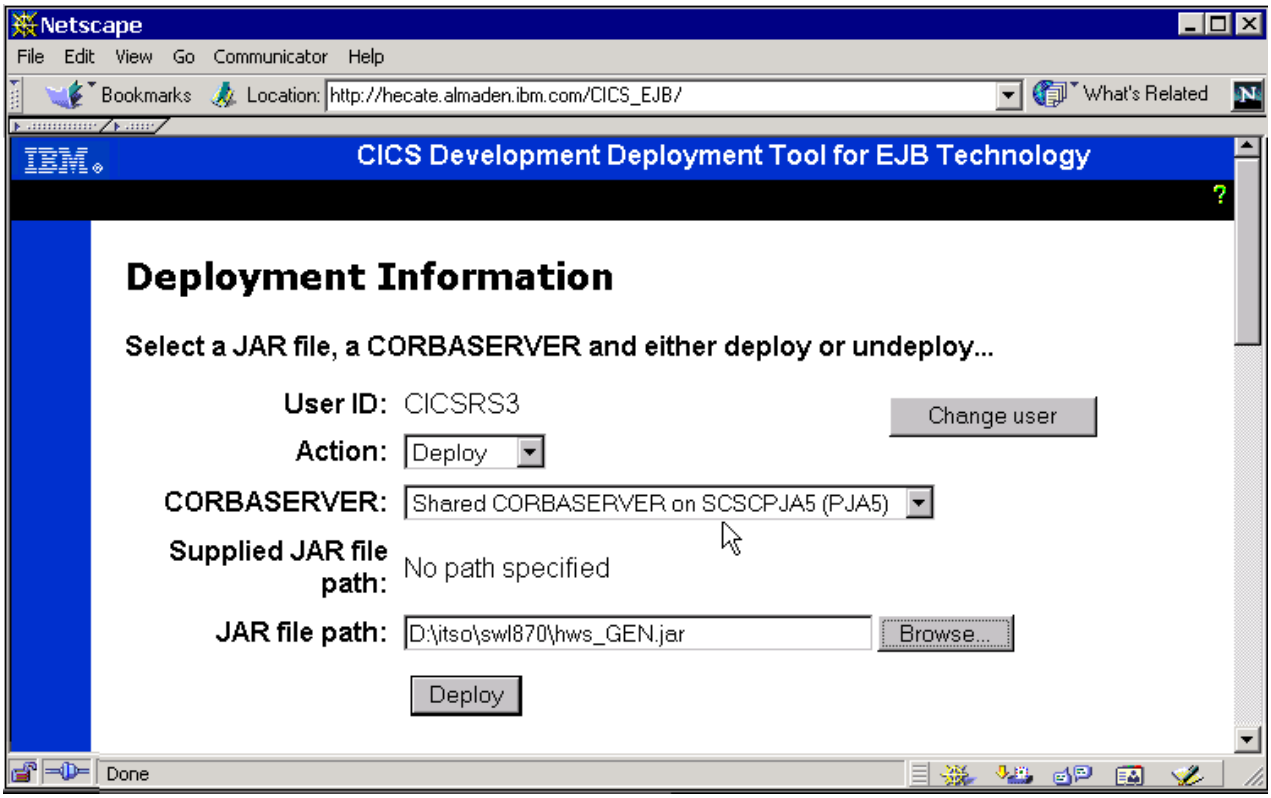


Figure 6-19 Deployment Information page of CICS development deployment tool

The selected CORBASERVER specifies the run-time environment within the CICS EJB server in which the enterprise beans in your selected JAR are to be run. Select a CORBASERVER and enter the path of the CICS deployed JAR file into the **JAR file path** field. Then click **Deploy** button to start the operation. Now the CICS development deployment tool performs the following operation using the CICSDDTbean within CICS.

1. Creates the necessary CICS resource definitions in the CICS runtime database.
2. Stores the resource definitions in the EJB-JAR file (in the file `cics-ejb-jar-ext.xmi`) for later use by the CICS production deployment tool.
3. Uploads the EJB-JAR file from the client to WebSphere, where it is validated, then transferred by FTP to the HFS on OS/390.
4. Publishes the name of the EJB-JAR file to the COS Naming Server.

The results of the deploy operation are displayed in the *Deploy Results* page (Figure 6-20).

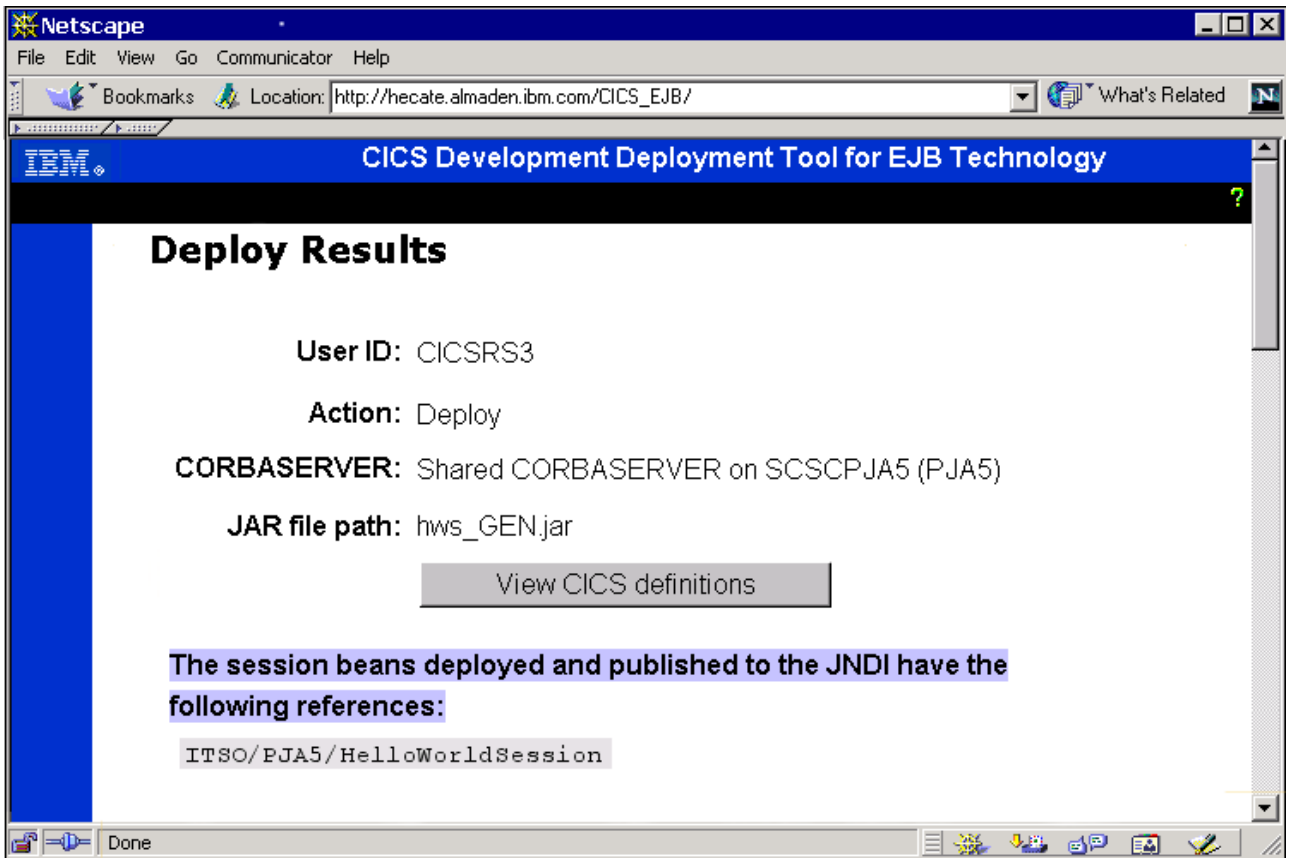


Figure 6-20 Deployment Results page of CICS development deployment tool

If a problem occurs during the deployment, a message is issued. Messages are color coded blue for warning, and pink for error. Select the **?** icon to the right of the message to display further information about the problem.

Attention: Note that the *Undeploy* operation removes the CICS resource definitions associated with a previously deployed EJB-JAR file, but it does not remove the EJB-JAR file itself from HFS.

Deploying with the CICS production deployment tool

The *CICS production deployment tool* is a workstation based tool designed for production deployment. It takes as input a CICS-deployed JAR file, either on the local workstation or on a remote OS/390 accessed by FTP, and allows you perform the following actions:

- Create a CICS DFHCSDUP input streamt o define the necessary CICS resource definitions in the CICS CSD.
- Create a CICSPlex SM data repository to define the necessary CICS resource definitions

The tool can be run either via a GUI on your workstation, or as an offline utility.

Store the deployed JAR file on the HFS

We used the GUI of the CICS production deployment tool in FTP mode, so we first of all transferred the CICS deployed JAR file, output from the CICS JAR development tool to the following location on the HFS on our OS/390 system:

```
/u/cicsts21/djars/hws_GEN.jar
```

To start the tool on your workstation, select **Programs -> IBM CICS TS 2.1 Tools -> CICS Production Deployment Tool for EJB Technology**. The GUI windows appears, which shows general information about the tool.

1. Click the **CICS EJB Production Deployment** button in the *upper left* part of this window.
2. Then click on the words **JAR file task** to reveal the full *navigation tree*.
3. Click the **Work with JAR files via FTP** radio button in order to work remotely with JAR files on OS/390, then click the **OK** button at the *bottom* of the window (Figure 6-21).

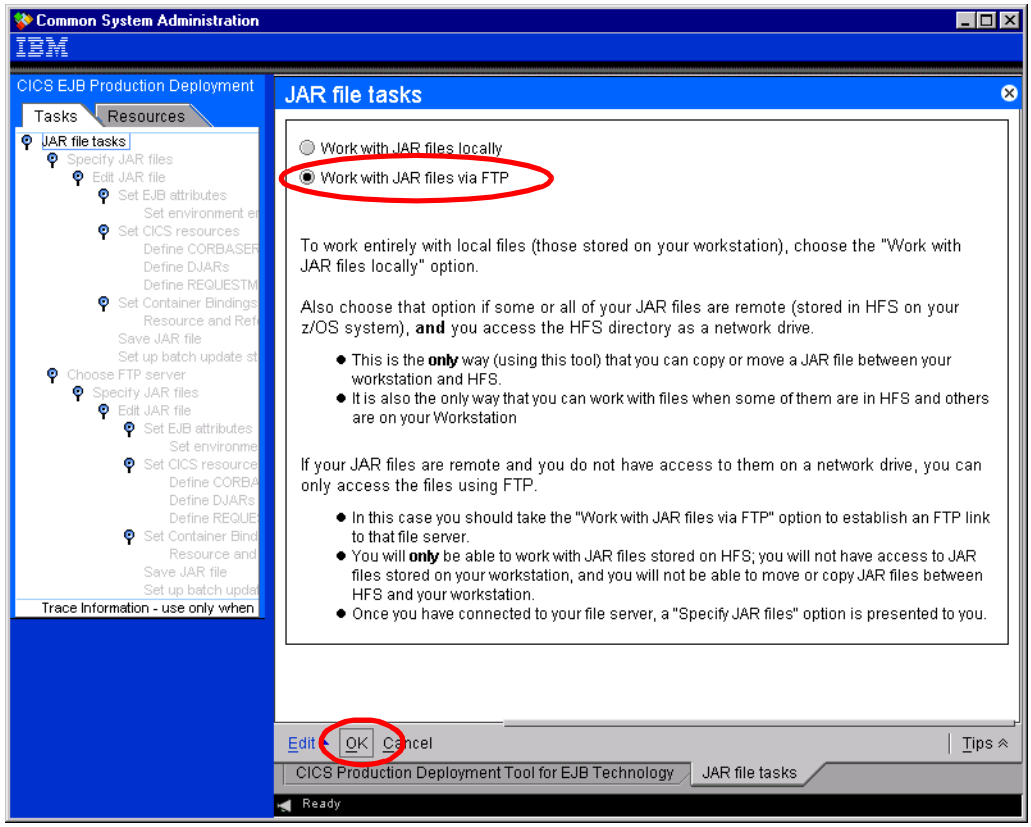


Figure 6-21 CICS production deployment tool

4. Click the **Choose FTP server** task line in the navigation tree.
5. Click the **Yes** button in the pop-up box to start a new JAR file editing session.
6. Type in your FTP server host name, userid, and password in the corresponding fields and click the **OK** button, as illustrated in Figure 6-22.

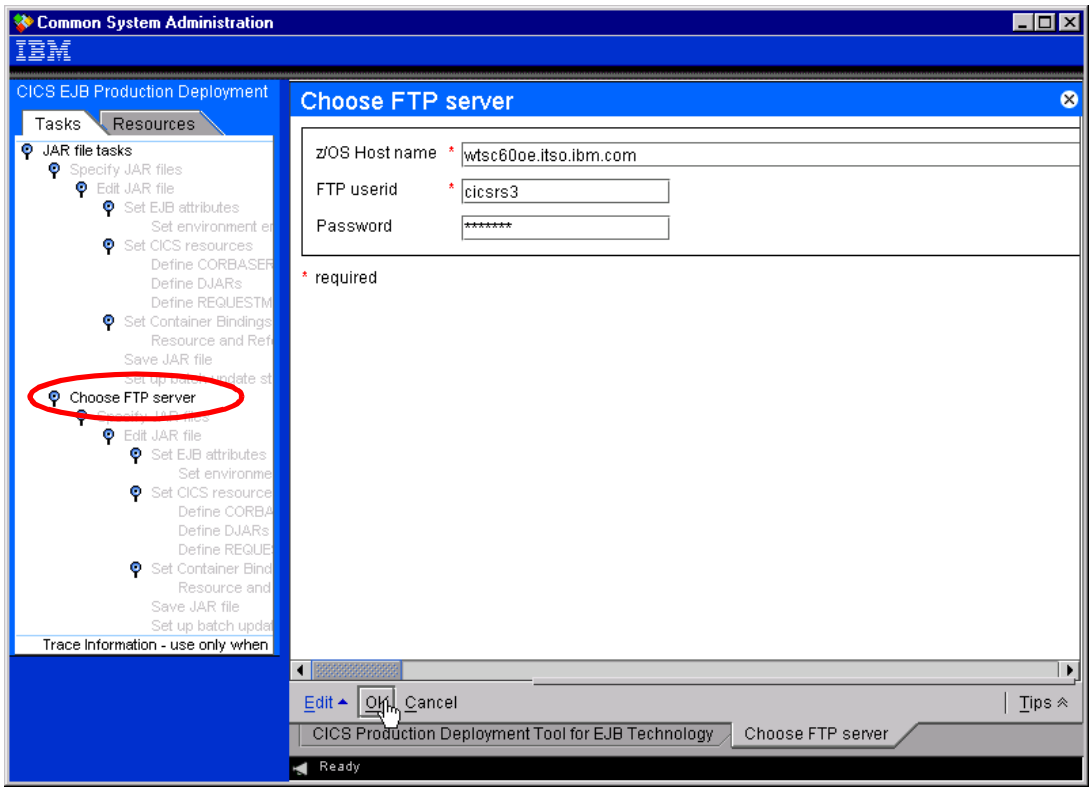


Figure 6-22 CICS production deployment tool, choose FTP server

7. Click the **Specify JAR files** task line in the navigation tree.
8. Click the **Yes** button in the pop-up box to start a new JAR file editing session.
9. Type in the source JAR file name and the destination JAR file name in the corresponding fields and click the **OK** button. We entered the following for our HelloWorld enterprise bean.

   Source JAR file name `/u/cicsts21/djars/hws_GEN.jar`

   Destination JAR file name `/u/cicsts21/djars/hws_GEN1.jar`

10. Now click the **Define DJARs** task line under the **Set CICS resources** task line in the navigation tree.
11. Click the **Create** button at the bottom of the window to define a new DJAR resource. We entered the following information for our HelloWorld session bean, as shown in Figure 6-23.

    DJAR name `HWS`

    CORBASERVER `PJA5`

    HFS file `/u/cicsts21/djars/hws_GEN1.jar`

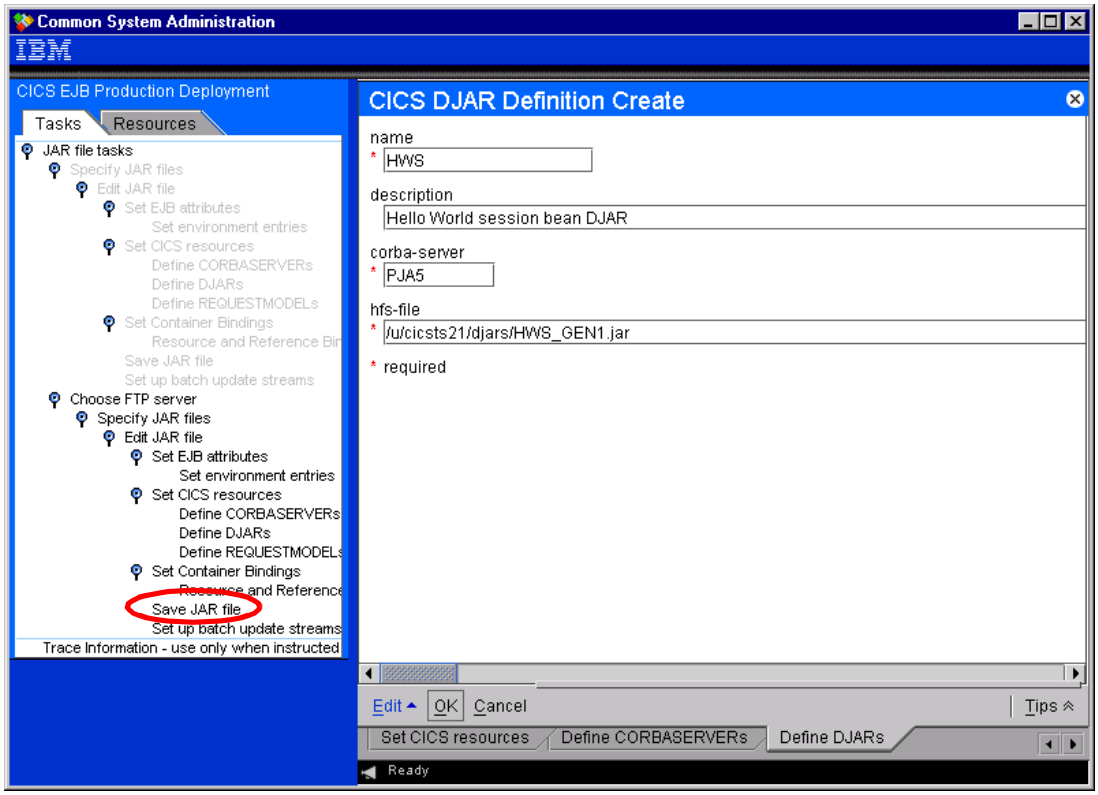


Figure 6-23 CICS production deployment tool, CICS DJAR Definition Create window

12. Click the **Save JAR file** task line in the navigation tree.
13. Click the **Yes** button to save the updated JAR file.
14. Click the **OK** button to confirm that the FTP transfer was successfull.

    At this point you have successfully created a new deployed JAR file on OS/390, containing the necessary definitions to build a DJAR definition.

15. Now click the **Set up batch update streams** task line in the navigation tree.
16. Then select the **CICS DFHCSDUP stream** radio button and click the **OK** button.
17. In the CICS parameters window, specify the name of the CICS list, the name of the CSD group and a valid HFS file name where the generated DFHCSDUP output stream is to be stored.

    We entered the following information, before clicking the **OK** button.

    CSD List `PJA5LIST`

    CSD Group `ITSOEJB`

    Output HFS File name `/u/cicsts21/djars/cicscsd.def`

    This creates a DFHSCUP output stream and stores it in the specified file on OS/390. Once the FTP transfer is complete, a window will appear; click the **OK** button to confirm that the FTP transfer was successful.

In our case, a new file, named /u/cicsts21/djars/cicscsd.def, was created. The DFHCSDUP input stream definitions contained in that file are shown in Example 6-1.

Example 6-1 DFHCSUP input stream generated by CICS production deployment tool

```
DELETE DJAR(HWS)
       GROUP(ITSOEJB)

DEFINE DJAR(HWS)
       GROUP(ITSOEJB )
       DESCRIPTION(DJAR CICS resource definition for HelloWorld EJB)
       CORBASERVER(PJA5)
       HFSFILE(/u/cicsts21/djars/hws_GEN1.jar)

ADD GROUP(ITSOEJB) LIST(PJA5LIST)
```

Applying generated CICS resource definitions

To utilize the DFHCSDUP input stream created by the CICS development deployment tool, it is necessary to copy the resource definition statements from the HFS to an MVS data set before running DFHCSDUP. The JCL we used to perform the copy step and to apply the statements is shown in Example 6-2.

Example 6-2 JCL to apply resource definition statements

```
//CRS3DUP JOB AX4328,AXP4328,
//         CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1),
//         NOTIFY=&SYSUID,
//         REGION=0M
//*
//PARMS SET HFSPATH='''/u/cicsts21/djars/cicscsd.def''',
//          CSDNAME=CICSSYSF.CICSTS21.DFHCSD
//*
//OCOPY EXEC PGM=IKJEFT01,REGION=0M
//CSDIN DD DSN=&&CSDIN,DISP=(NEW,PASS),UNIT=SYSDA,
//      SPACE=(CYL,(8,64)),DCB=(LRECL=80,BLKSIZE=6080,
//      RECFM=FB)
//HFSIN DD PATH=&HFSPATH
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
OCOPY INDD(HFSIN)OUTDD(CSDIN)
/*
//CSDUP EXEC PGM=DFHCSDUP,REGION=0M,COND=(0,NE)
//STEPLIB DD DSN=CICSTS21.CICS.SDFHLOAD,DISP=SHR
//DFHCSD DD DSN=&CSDNAME,DISP=SHR
//SYSUT1 DD UNIT=SYSDA,SPACE=(1024,(100,10))
//SYSPRINT DD SYSOUT=A
//SYSUDUMP DD SYSOUT=A
//SYSABEND DD SYSOUT=A
//CBDOUT DD SYSOUT=A
//SYSIN DD DSN=*.OCOPY.CSDIN,DISP=SHR
```

Running this JCL for the first time, we got a warning (RC=4) indicating that the CICS group did not exist while processing the delete command. Subsequently, runs also returned a warning for the reason that the CICS group ITSOEJB was already a member of the CICS list PJA5LIST while processing the add command.

Publish names to JNDI

After creating the HelloWorld DJAR resource definition, the only steps remaining were to install the resource definitions and to publish the deployed JAR file to the COS Naming Server. We used the following two commands to perform these steps:

```
CEDA INSTALL GROUP(ITSOEJB)
CEMT PERFORM DJAR(HWS) PUBLISH
```

Due to the fact that the group ITSOEJB containing the DJAR resource definition was added to the CSD list PJA5LIST, which was specified in our CICS regions GRPLIST, the DJAR HWS was always accessible after subsequent restarts of the CICS system.

6.4 Testing with a Java client application

Using the EJB Test Client supplied with VAJ is limited to the VAJ WebSphere Test Environment. If you want to use a client outside of VAJ, for example, as a standalone application in the USS environment, you have to write your own client code.

6.4.1 Writing the client within VAJ

Clients use a set of interfaces that provide access to enterprise beans and their business logic. These operations are the same for the majority of client programs. In this section we describe the Java client program, *HelloWorldClient*, which we used to test the enterprise bean, HelloWorldSession, developed in 6.2.1, “Developing in VAJ” on page 137.

The important sections of our HelloWorldClient are shown in Figure 6-24, and are discussed further in the following sections.

```
public class HelloWorldClient {
public static void main(String[] args) {
1 HelloWorldSessionHome helloWorldSessionHome =
    (HelloWorldSessionHome)EJBHelper.jndi_lookup(HelloWorldSessionHome.class);
13HelloWorldSession helloWorldSession = helloWorldSessionHome.create();
14String result = helloWorldSession.sayHello
    (System.getProperties().getProperty(METHOD_ARGUMENT, "Hello world"));
15System.out.println ("Result from business method is: " +result);
16helloWorldSession.remove();
}}
public class EJBHelper {
public static Object jndi_lookup(Class resultClass) {
2 Properties sp = System.getProperties();
3 String propFileName = sp.getProperty(PROPERTY_FILE);
4 Properties p = EJBHelper.jndi_properties(propFileName);
9 String helloWorldHomeNameString = p.getProperty(JNDI_HELLOWORLD_SESSION_NAME);
10Context ctx = new InitialContext(p);
11Object tempObject = ctx.lookup(helloWorldHomeNameString);
12return javax.rmi.PortableRemoteObject.narrow(tempObject, resultClass);
}
public static Properties jndi_properties(String propFileName) throws Exception {
5 FileInputStream propFile = new FileInputStream(propFileName);
6 Properties p = new Properties(System.getProperties());
7 p.load(propFile);
8 return p;
}}
```

Figure 6-24 HelloWorldClient and EJBHelper

The Java client needs to do the following.

- **1 Call the jndi_lookup() method of EJBHelper class.**
  The EJBHelper class contains two methods, `jndi_lookup()` and `jndi_properties()`. The `jndi_lookup()` method performs all necessary operations to get a home object reference.
- **2 Get system properties and create a property object.**
  A system property is a key/value pair that the Java runtime defines to describe the user, system environment, and Java system. The runtime defines and uses a set of default system properties. Other properties can be made available to a Java program via the `-D` command line option to the Java interpreter. Running the Java interpreter as shown below adds the values from the file defined by CLIENT_PROPERY_FILE to the list of system properties.

  ```
  java -DCLIENT_PROPERTY_FILE=D:\itso\swl870\client_nt.properties
  ```

  The java.lang.System class contains static methods for reading and updating system properties. Environment properties set as system properties affect the context of all applications. We created an empty property list object with the default specified in the system properties.
- **3 Get name of the property file.**
  The list of key/value pairs contained in this property file is used to determine the type of naming service and its network location. The file must also contain the JNDI name to which the home object of the enterprise bean is bound. The `jndi_lookup()` method uses this name for the JNDI `lookup()` operation on the naming directory.
- **4 Call the jndi_properties() method of EJBHelper class.**
  The `jndi_properties()` method is used to initialize a Properties object from a file containing a list of key/value pairs. The name of the property file is passed as an argument to the method.
- **5 Open a connection to the property file.**
  We use the FileInputStream constructor to open a connection to the property file, specified by the path name in the file system. A FileInputStream obtains input bytes from a file in a file system. The constructor method creates a new FileDescriptor object to represent the file connection.
- **6 Get system properties and create a property object.**
- **7 Load properties from property file.**
  We use the `load()` method of the class Properties to read the property list (key and value pairs) from the input stream. Every property occupies one line of the input stream.
- **8 Return the property object.**
- **9 Get JNDI home name of the enterprise bean.**
  This is the name to which the home object of the enterprise bean is bound. The `jndi_lookup()` method uses this name for the JNDI `lookup()` operation on the naming directory.
- **10 Create an initial context object.**
  An initial context object is a local starting point for any JNDI lookup. To create an initial context, you first create a properties table of the type *Properties*. The Properties class represents a persistent set of properties (key/value pairs). The values you add to that table determine the kind of initial context you want to use. These values are mainly the type of naming service you want to use and its network location. We have loaded the properties from the property file as described in step 4 to 8.
- **11 Obtain a remote object reference.**
  A home object is a factory responsible for instantiating and destroying EJB objects. To obtain a home object reference, you must perform a JNDI `lookup()` operation on the

naming directory. This operation returns an RMI remote object, which must you cast to a home object.

- **12 Narrow to the home object reference.**
  The `javax.naming.Context.lookup()` method returns an RMI remote object. The EJB 1.1 specification requires that you must narrow (cast) that object to a home object. Finally, we return the home object reference to the main program.
- **13 Obtain an object reference.**
  An EJB object acts as glue between the client and the bean. It delegates all client requests to the bean. To obtain an EJB object reference, you must use one of the `create()` methods on the home object reference. For stateless session beans, no parameters are passed to the `create()` method.
- **14 Invoke a business method.**
  To invoke a business method, you must use the enterprise bean object reference. In our program, we call the method `sayHello()` of the enterprise bean HelloWorldSession and save the return value in the string variable result.
- **15 Echo the return value the user.**
- **16 Remove an EJB object.**
  To destroy an EJB object, you must use the `remove()` method on the home object reference. The impact of that operation depends on the type of the bean.

Defining EJBHelper class in VAJ

To add the EJBHelper class to the package itso.ejb390.helloworld, select the **Projects** tab in the VAJ Workbench, click the project ITSO EJB 390 Redbook, and select **Add -> Class**. The *Create Class SmartGuide* appears. Select the **Create a new class** radio button and the proper project and package, type in the name of the class, `EJBHelper`, in the **Class name** field and click the **Finish** button.

To add the `jndi_properties()` method to the class EJBHelper, perform the following steps.

- Expand the project ITSO EJB 390 Redbook, right-click the class `EJBHelper`, and select **Add -> Method**.
- In the *Create Method SmartGuide*, select the **Create a new method:** radio button and click the **Next** button.
- In the *Attributes SmartGuide*, type in the method name, `jndi_properties()`, in the **Method Name** field, and the return type, Properties, in the **Return type** field.
- Select the **public** and the **static** radio button and click **Add**.
- In the Parameters window, type in the name of the argument, `propFileName`, in the **Name** field, select the **Reference Type** radio button and type in the argument type, `String`.
- Click **Add**, **Close**, and finally **Finish**.
- Edit the method's source code in the source panel.

Repeat these steps accordingly for the `jndi_lookup()` method.

Defining HelloWorldClient class in VAJ

To add the HelloWorldClient class to the package itso.ejb390.helloworld, expand the project ITSO EJB 390 Redbook, right-click the class `HelloWorldClient`, and select **Add -> Method** to open the *Create Method SmartGuide*. Select **Create a new main method:** and click **Finish**. Edit the main method's source code in the source panel.

Dependencies of the client code

An EJB server must provide JNDI access to their naming service. The naming service provides object binding and a lookup API. A Java client uses the JNDI to instantiate a connection to an EJB server and to locate a specific enterprise bean home. To do this, the client tells the JNDI API where it can find the EJB server and which kind of JNDI service provider (driver) it should have. These properties will change depending on how an enterprise bean vendor has implemented JNDI. The following extract shows the valid properties for the VAJ Test Environment:

```
Properties p = new Properties);
p.setProperty(“java.naming.factory.initial”,“com.ibm.ejs.ns.jndi.CNInitialContextFactory”);
p.setProperty(“java.naming.provider.url”, ”iiop://localhost:1800/”);
Context ctx = new Context(p);
```

If you want to use the JNDI API provided by CICS TS V2.1, the following properties would be used:

```
Properties p = new Properties);
p.setProperty(java.naming.factory.initial,“com.sun.jndi.cosnaming.CNCtxFactory”);
p.setProperty(“java.naming.provider.url”,”iiop://hecate.almaden.ibm.com:900/”);
Context ctx = new Context(p);
```

Note that `javax.naming.Context.INITIAL_CONTEXT_FACTORY` and `javax.naming.Context.PROVIDER_URL` are Java string constants supplied in the `javax.naming.Context Interface` provided by VAJ. These constants are defined as “`java.naming.factory.initial`” and “`java.naming.provider.url`”, respectively.

You can specify environment properties to the JNDI by using the environment parameter to the InitialContext constructor (as shown in the code fragments above) and application resource files. Several JNDI environment properties could also be specified by using system properties.

- **Application resource file.** This is an optional property file named `jndi.properties`, which contains a list of key/value pairs presented in the property file format. The key is the name of the property, for example, *java.naming.factory.initial*, and the value is a string in the format defined for that property. The JNDI automatically reads the application resource files from `JAVA_HOME/lib/jndi.properties`, where `JAVA_HOME` is the file directory that contains your Java runtime environment. The JNDI adds the properties from that file into the environment of the initial context.
- **System properties.** See step 2 of 6.4.1, “Writing the client within VAJ” on page 159, for an explanation of system properties.

6.4.2 Running the client within VAJ

Once we had written our own client, we could test it within VAJ or as a standalone application from the Windows NT or USS environment. To call an enterprise bean,the client must tell the JNDI API where it can find the CICS EJB server. The client allows us to specify this property, together with the properties for the initial context factory and the JNDI name of the enterprise bean home object, in a property file. Example 6-3 shows the properties file (`hwc_vaj.properties`) to call an enterprise bean locally deployed in the VAJ EJB server.

Example 6-3 Property file to call an EJB locally deployed in VAJ

```
java.naming.factory.initial=com.ibm.ejs.ns.jndi.CNInitialContextFactory
java.naming.provider.url=iiop://localhost:1800/
JNDI_HELLOWORLD_SESSION_NAME=itso/ejb390/helloworld/HelloWorldSession
```

To run the client, we performed the following steps.

- We expanded the project ITSO EJB 390 Redbook, right-clicked the `HelloWorldClient` class and selected **Run -> Run main with**.
- In the properties window for the client, we selected the **Program** tab and typed in the key/value pair defining the property file in the **Properties** field. We used PROPERTY_FILE=`D:\itso\swl870\hwc_vaj.properties` (Figure 6-25).

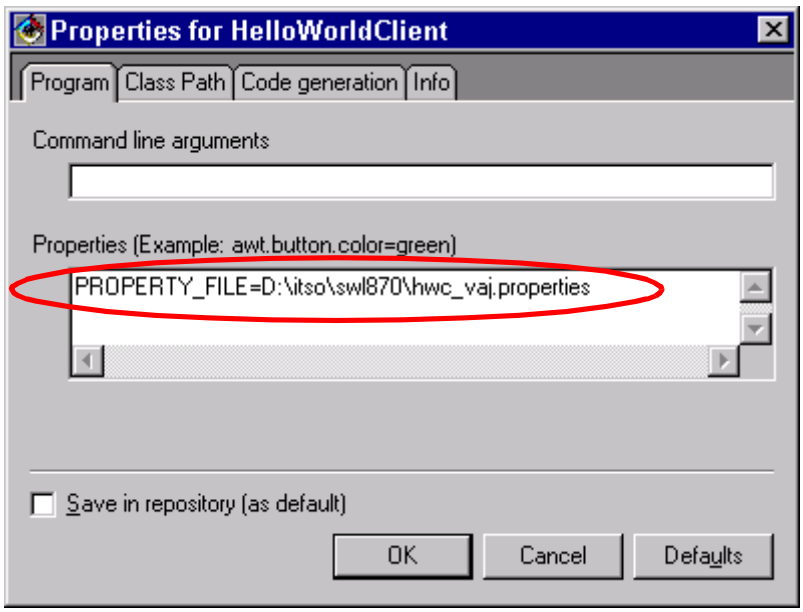


Figure 6-25 VAJ Program window for HelloWorldClient

- We selected the **Class Path** tab, selected the **Project path** check box and clicked **Edit**.
- We selected the **IBM Enterprise Extension Libraries** and **IBM WebSphere Test Environment** check boxes and clicked **OK** (Figure 6-26).

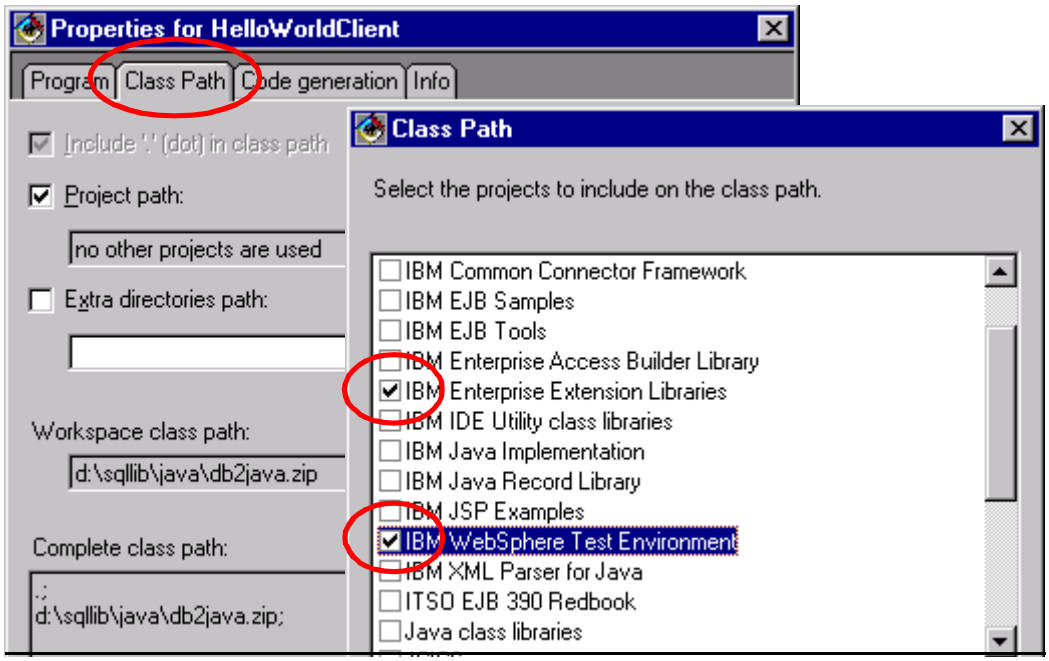


Figure 6-26 VAJ Class path window

- We clicked **OK** to start the client. We checked the output in the Console window for the client process `itso.ejb390.helloworld.HelloWorldClient.main()` to see whether the client had run successfully (Figure 6-27).

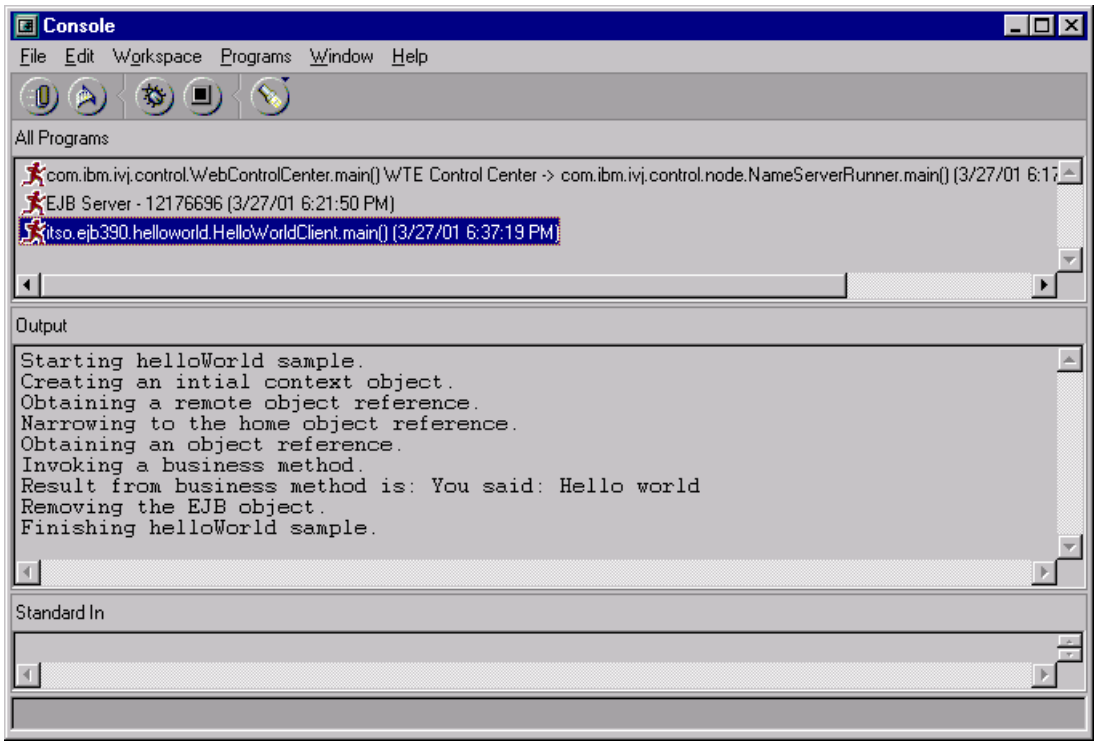


Figure 6-27 VAJ Console window showing the output of HelloWorldClient

To call an enterprise bean deployed in the CICS EJB server, we changed the properties for the EJB server and the JNDI name of the enterprise bean home object. Figure 6-4 shows the properties we used to call an enterprise bean deployed in the CICS EJB server.

Example 6-4 Property file to call an enterprise bean deployed in CICS

```
java.naming.factory.initial=com.ibm.ejs.ns.jndi.CNInitialContextFactory
java.naming.provider.url=iiop://hecate.almaden.ibm.com:900/
JNDI_HELLOWORLD_SESSION_NAME=PJA5/HelloWorldSession
```

Note: In order to check whether the enterprise bean was really executed into the CICS EJB server, you can look for the message `You said: Hello world` in one of the `stdout` files in the `cicsts21/work/<APPLID>` directory, where <APPLID> is the CICS APPLID of your CICS region. This `work` directory has a subdirectory for each CICS APPLID which is used for the java programs `stdin`, `stdout`, and `stderr` files. For more information related to the work directory, please read “Defining the CICS HFS directories” on page 69 and “The Java stdin/stdout/stderr files” on page 104.

6.4.3 Running the client from the Windows NT environment

In order to run the client as standalone application from the Windows NT environment, we created the directory (`d:\itso\swl870\client`) and copied the following files to this directory.

Client JAR file `hws_CLI.jar`

The client JAR file contains all classes and interfaces used to access enterprise beans. We generated the client JAR file with the CICS JAR development tool as described in 6.3.2, “Generating a CICS deployed JAR file” on page 148.

Java client program `hwc.jar`

The Java client program accesses the business logic of an enterprise bean. We used the file as Java client. How we generated our JAR file `hwc.jar` is described in “Packaging a Java client with VAJ” on page 165.

We also created the following two files which we will describe later in this section:

Property file `hwc_nt.properties`

Java client command script `hwc.cmd`

Note: All the files we used are available as samples with this redbook. For further details, refer to Appendix C, “Using the additional material” on page 315.

Packaging a Java client with VAJ

To generate and export a JAR file containing the Java client classes from within VAJ, we performed the following steps:

- We selected the **Projects** tab in the VAJ Workbench, expanded the project *ITSO EJB 390 Redbook*, and the *itso.ejb390.helloworld* package, right-clicked the *HelloWorldClient* class, and selected **Export** (Figure 6-28).

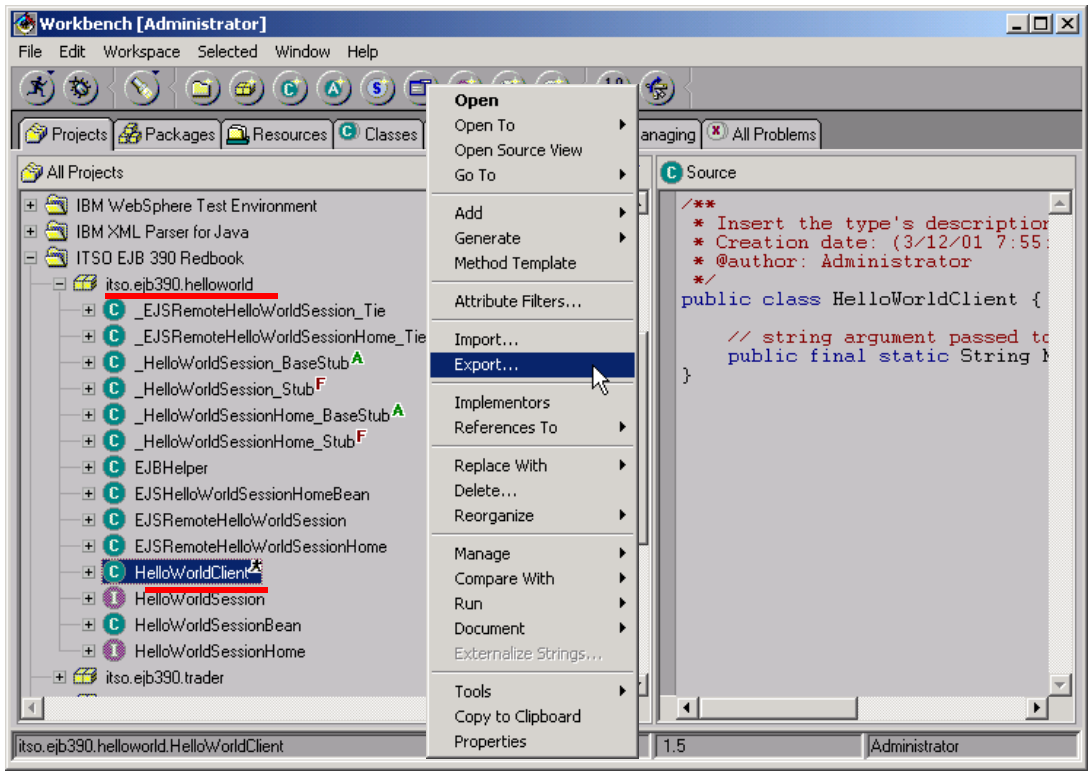


Figure 6-28 VAJ Export EJB JAR file

- We selected the **Jar file** radio button on the *Export SmartGuide* and clicked **Next**.
- In the *Export to a JAR file SmartGuide*, we typed in the name and location of the JAR file `hwc.jar` in the **Jar file** input field, selected the **class** check box, and clicked the **Details** button to select the classes we wanted to export Figure 6-29.

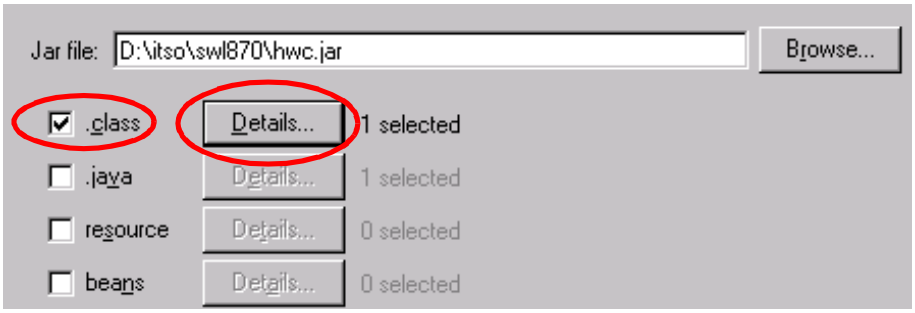


Figure 6-29 VAJ Export to a JAR file SmartGuide

- In the *class export* window we selected the *EJBHelper* class and the *HelloWorldClient* class and clicked **OK**.

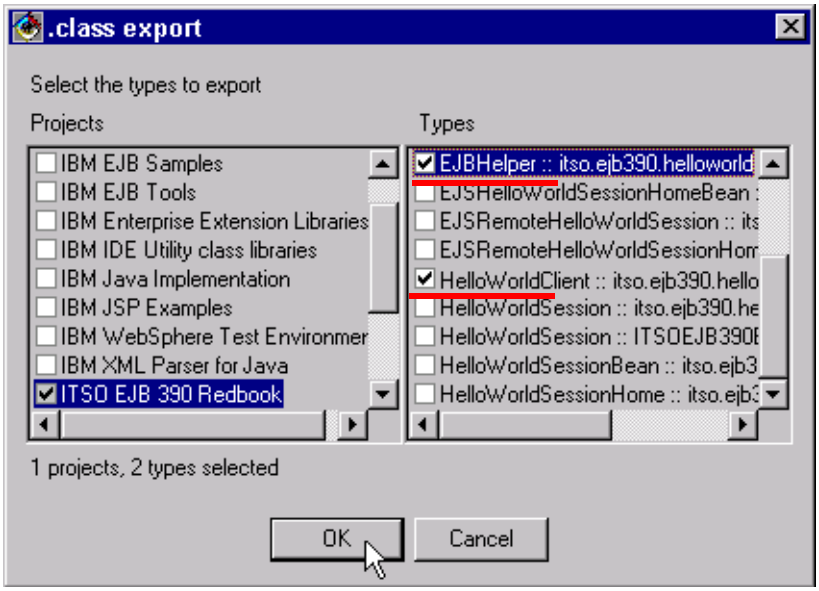


Figure 6-30 VAJ Export to a JAR file SmartGuide, class export

- Finally, we clicked the **Finish** button to export the client classes to a JAR file.

Property file for Windows NT environment

Example 6-5 shows the property file (`hwc_nt.properties`) we used to call an enterprise bean deployed in the CICS EJB server from Windows NT.

Example 6-5 Property file to call an EJB deployed in CICS from Windows NT

```
java.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
java.naming.provider.url=iiop://hecate.almaden.ibm.com:900/
JNDI_HELLOWORLD_SESSION_NAME=ITSO/PJA5/HelloWorldSession
```

Java command script for Windows NT

Example 6-6 shows the command script (`hwc.cmd`) we used to run the Java client program on Windows NT.

The command script provided assumes the file `j2ee.jar` is located in the directory `C:\PROGRA~1\IBM\j2ee`, and a Java 2 runtime in installed in `C:\PROGRA~1\IBM\Java13`. If you have this file available in another directory, you will have to edit the file `hwc.cmd` and change the `JAVA_HOME` and `JAVA_J2EE` variables accordingly.

Tip: If you have the CICS JAR development tool installed on your workstation, you will already have a copy of the j2ee.jar, the default location being `C:\Program Files\IBM\CICS TS 2.1 Tools\Common\j2ee.jar`. Or, you can obtain a copy with the Java 2 SDK, Enterprise Edition, which is available from `http://java.sun.com`.as an alternate possibility

Example 6-6 Java command script to run the Java client on WinNT

```
echo off
rem ---------------------------------------------------------------------
rem CICS EJB HelloWorld run command script
rem Use this file to call HelloWorldSession from WinNT.
rem
rem Modify the following to match your IBM SDK 1.3 installation directory:
set JAVA_HOME=C:\PROGRA~1\IBM\Java13
rem
rem Modify the following to match your directory containing j2ee.jar:
set JAVA_J2EE="C:\Program Files\IBM\CICS TS 2.1 Tools\Common"
rem ---------------------------------------------------------------------
setlocal
rem ---------------------------------------------------------------------
set CLIENTCLASSPATH=.;hwc.jar;hws_CLI.jar;%JAVA_J2EE%\j2ee.jar
rem ---------------------------------------------------------------------
echo CICS EJB HelloWorld: Querying the Java SDK level.
if exist %JAVA_HOME%\bin\java.exe goto callejb
echo CICS EJB HelloWorld: Failed, possible cause:
echo     Java support not found at $JAVA_HOME.
echo     Check the JAVA_HOME setting in the hwc.cmd command script.
goto endcmd
rem ---------------------------------------------------------------------
:callejb
echo CICS EJB HelloWorld: Starting the EJB client program.
%JAVA_HOME%\bin\java -classpath %CLIENTCLASSPATH% -DPROPERTY_FILE=.\hwc_nt.properties
itso.ejb390.helloworld.HelloWorldClient
if errorlevel 0 goto success
echo CICS EJB HelloWorld: Failed
echo     Check the JAVA_HOME and CLASSPATH settings in the hwc.sh
echo     shell script, and the CICS server installation steps.
goto endcmd
rem ---------------------------------------------------------------------
:success
echo CICS EJB HelloWorld: Completed successfully.
:endcmd
endlocal
```

Run the client on Windows NT

In order to test our HelloWorld session bean, we invoked our client using our `hwc.cmd` file from a Windows command prompt. We saw the output shown in Example 6-7, indicating that the bean had been successfully invoked in our CICS region.

Example 6-7 Output of Java client run on WinNT

```
D:\itso\swl870\client>hwc.cmd
CICS EJB HelloWorld: Querying the Java SDK level.
CICS EJB HelloWorld: Starting the EJB client program.
Starting helloWorld sample.
Creating an intial context object.
Obtaining a remote object reference.
Narrowing to the home object reference.
Obtaining an object reference.
Invoking a business method.
Result from business method is: You said: Hello world
Removing the EJB object.
Finishing helloWorld sample.
CICS EJB HelloWorld: Completed successfully.
```

6.4.4 Running the client from the USS environment

We now wished to test our HelloWorld session bean from OS/390 in the UNIX System Services (USS) envrionment. First of all we transferred the following files to the HFS of the OS/390 system in which our CICS region was running.

Client JAR file `hws_CLI.jar`

Java client `hwc.jar`

We also created the following two files which we will describe later in this section:

Property file `hwc_uss.properties`

Java client command script `hwc.sh`

Note: All the files we used are available as samples with this redbook, for further details refer to the Appendix C, “Using the additional material” on page 315.

Tip: If transferring the files using FTP, ensure that the JAR files are transferred in *binary* mode from Windows to OS/390.

Property file for USS environment

Figure 6-31 shows the properties file (`hwc_uss.properties`) we created to test our HelloWorld session bean from USS.

```
java.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
java.naming.provider.url=iiop://hecate.almaden.ibm.com:900/
JNDI_HELLOWORLD_SESSION_NAME=PJA5/HelloWorldSession
```

Figure 6-31 Property file to call an enterprise bean from USS

Java shell script

Example 6-8 shows the shell script (`hwc.sh`) we used to run the Java client program on USS.

Example 6-8 Java shell script to run the Java client on USS

```
# ------------------------------------------------------------------------
# CICS EJB HelloWorld run shell script
#
# Modify the following to match your IBM SDK 1.3 installation directory:
JAVA_HOME=/usr/lpp/java213d/J1.3
#
#------------------------------------------------------------------------
#
CLASSPATH=./hwc.jar:./hws_CLI.jar:$JAVA_HOME/standard/ejb/1_1/ejb11.jar
#
echo "CICS EJB IVP: Querying the Java SDK level"
if $JAVA_HOME/bin/java -version
then
else echo "CICS EJB HelloWorld: Failed, possible cause:"
     echo "    Java support not found at $JAVA_HOME"
     echo "    Check the JAVA_HOME setting in the hwc.sh shell script"
     exit
fi
#
echo ""
echo "CICS EJB HelloWorld: Starting the EJB client program"
if $JAVA_HOME/bin/java -classpath $CLASSPATH -DPROPERTY_FILE=./hwc_uss.properties
itso.ejb390.helloworld.HelloWorldClient
then echo "CICS EJB HelloWorld: Completed successfully"
else echo "CICS EJB HelloWorld: Failed"
     echo "    Check the JAVA_HOME and CLASSPATH settings in the hwc.sh"
     echo "    shell script, and the CICS server installation steps."
fi
```

Note: The `hwc.sh` shell script provided assumes the IBM SDK 1.3 installation directory is `/usr/lpp/java213d/J1.3`. If you use a different installation directory, you have to change the `JAVA_HOME` variable accordingly.

Run the client on USS

Before testing our client on USS we added execute permissions to the shell script using the command **chmod 775 hwc.sh.** Then, after starting the shell script, we saw the output shown in Example 6-9.

Example 6-9 Output of enterprise bean client run on USS

```
CICSRS3 @ SC69:/u/cicsrs3/helloWorld>hwc.sh
CICS EJB IVP: Querying the Java SDK level
java version "1.3.0"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.0)
Classic VM (build 1.3.0, J2RE 1.3.0 IBM OS/390 Persistent Reusable VM build hm1)
CICS EJB HelloWorld: Starting the EJB client program
Starting helloWorld sample.
Creating an intial context object.
Obtaining a remote object reference.
Narrowing to the home object reference.
Obtaining an object reference.
Invoking a business method.
Result from business method is: You said: Hello world
Removing the EJB object.
Finishing helloWorld sample.
CICS EJB HelloWorld: Completed successfully
CICSRS3 @
SC69:/u/cicsrs3/helloWorld>
```

6.5 Summary

This chapter has shown how to develop, deploy and test a simple HelloWorld enterprise bean under CICS TS V2.1. We have developed a universal client which is able to invoke the enterprise bean from within VisualAge for Java, from the Windows NT environment, and from the OS/390 UNIX System Services (USS) environment.

We have also shown how you can install your enterprise bean into a CICS EJB server manually, or use the CICS development deployment tool or CICS production deployment tool to assist with this process.

In the next chapters, we go on to describe how we wrapped an existing CICS COBOL application with a stateful session bean, and then modified this to use new business logic written in Java.

7

Wrapping the Trader application: JCICS link

In this chapter we describe how to invoke an existing CICS COBOL application from a session bean running in the CICS TS V2.1 EJB container. The application we use is the COBOL *Trader* application, which is a typical pseudo-conversational CICS application. We show how to use the CICS Java class library (JCICS) to link to the existing business logic in the Trader application, passing a COMMAREA as a byte array. Our scenario is illustrated in Figure 7-1.

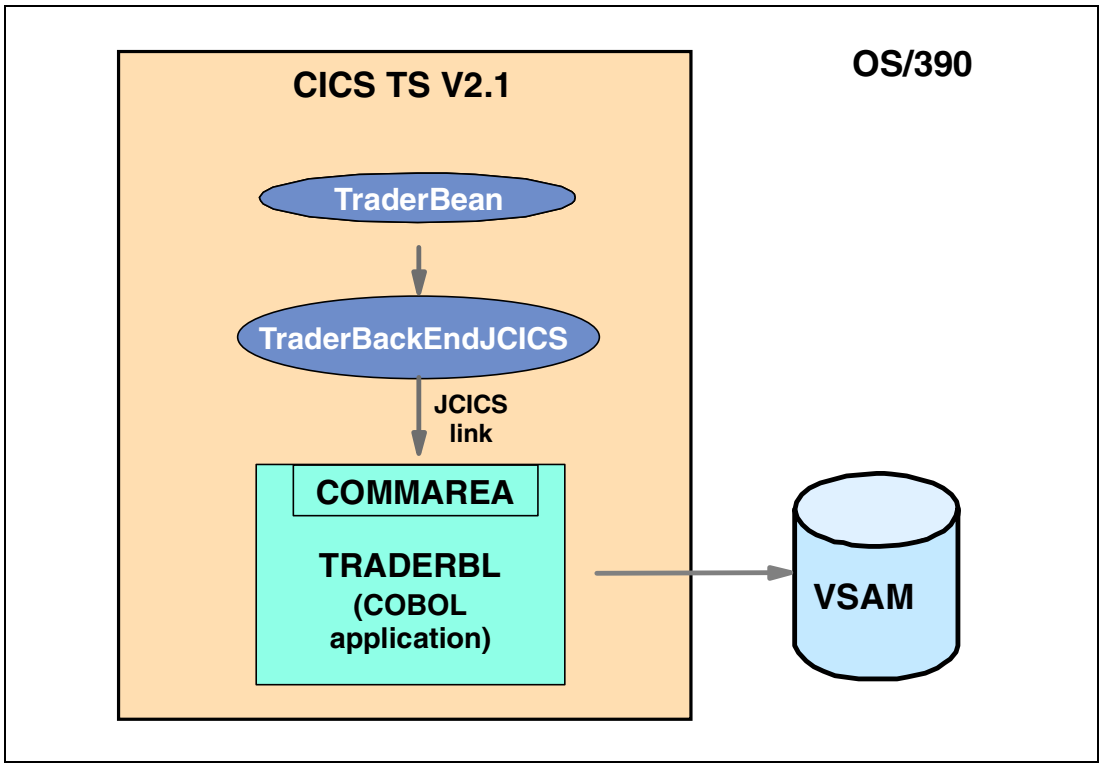


Figure 7-1 Calling a COBOL program with a JCICS link()

Introduction

The goal of this scenario and those in the following chapters is to write a re-usable enterprise bean such that a variety of different CICS application techniques can be used to provide the same business application. In our example, this means the TraderBean is designed in such a way that the actual technique of how the Trader application is invoked can be easily modified or extended to invoke the application in a different way, or to even to invoke a different application.

Before we describe how to wrap the Trader application, we give a short description of what the COBOL Trader program does and how it works. However, if you want to get some hands-on experience and run our sample enterprise bean immediately, jump straight to 7.1, “Quick start — Invoking TraderBean” on page 173.

The COBOL Trader application

The COBOL Trader sample is a traditional 3270 green-screen application that allows authenticated users to trade shares, that is, to buy and sell shares in a given group of companies, as well as obtaining real-time quotes on the value of their current holdings. Trader has been developed as a sample as part of an IBM CICS Web-enablement service offering and has been used extensively in the previous redbooks *A Performance Study of Web Access to CICS,* SG24-5480, and *Workload Management for Web Access to CICS,* SG24-6118.

The original 3270 version of the Trader application uses the CICS 3270 BMS interface and stores customer information in two VSAM files. It is a pseudo-conversational application, meaning that a chain of related non-conversational CICS transactions is used to convey the impression of a "conversation" to the users as they go through a sequence of screens that constitute a business transaction. However, key to Web-enablement, Trader also offers a business logic interface whereby the business logic program (TRADERBL) can be invoked using a COMMAREA interface from other clients with non-3270 interfaces. We use this interface in our later scenarios.

For further details on installing and using the Trader COBOL application, refer to Appendix B, “The COBOL Trader application” on page 309. To obtain the sample COBOL Trader application, refer to Appendix C, “Using the additional material” on page 315.

Wrapping a COBOL application using JCICS

JCICS is the Java class library provided with CICS TS V1.3 and V2.1. It provides a Java API for many of the EXEC CICS API commands. In this chapter we use only the `link()` method of the *Program* object in order to link from our enterprise bean into the existing business logic of the Trader application.

Developing the presentation interface

To test our enterprise bean we developed two different scenarios. First of all, in VisualAge for Java (VAJ), we developed a simple stand-alone Java application, *TraderTest*. This can be run from the command line to quickly and easily test the Trader enterprise bean. In addition, we also developed a servlet with a JSP HTML presentation interface to drive the enterprise bean. We deployed the servlet on WebSphere Application Server for Windows NT, which then made RMI/IIOP calls to our enterprise bean deployed in CICS TS V2.1. Figure 7-2 shows the topology of the test scenarios.

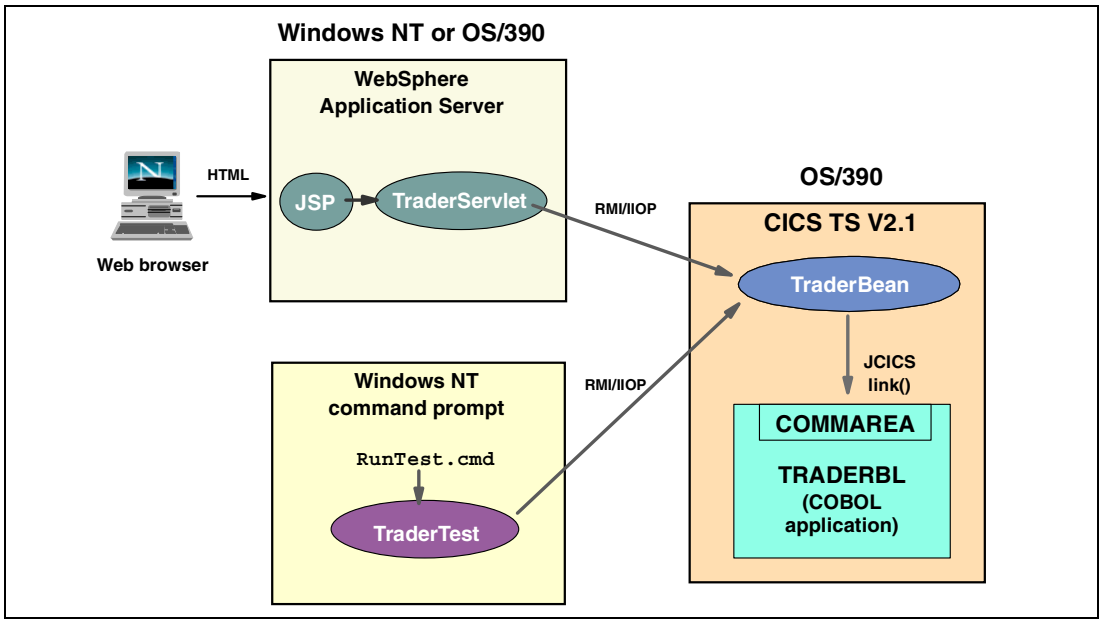


Figure 7-2 Topology of Trader test scenarios using JCICS link()

7.1 Quick start — Invoking TraderBean

If you want to run our sample Trader enterprise bean without following all the details specified in this chapter, use the steps below. All the source code and examples used in this book are available for download from the redbooks Web site `ftp://www.redbooks.ibm.com/redbooks/` and for full details of the available files and how to obtain them, refer to Appendix C, "Using the additional material" on page 315.

1. Install the COBOL Trader application in your CICS system. For more details, refer to Appendix B, "The COBOL Trader application" on page 309.
2. Create a CICS TCPIPSERVICE, CORBASERVER, REQUESTMODEL and DJAR definition if you have not already done so. For more details, refer to 6.3.3, "Deploying to CICS" on page 150.
3. Deploy the TraderBean to your CICS TS V2.1 region. For more information on how to do this, refer to Section 7.3, "Deploying the TraderBean to CICS" on page 191.
4. Test the application; this can be achieved in one of the following two ways:
   a. Use our supplied *TraderServlet* to create a Web application with an HTML front-end to TraderBean. For further details on the expected output, refer to Figure 7-50, "Quote results using TraderServlet" on page 214.
   b. Use the supplied runTest.cmd file to invoke our stand-alone Java test application *TraderTest*. To set up TraderTest, simply do the following:
      - On your workstation, create a directory (for example `C:\itsotrader`) and copy the following sample files to this directory:

        ```
        traderCLI.jar
        traderTest.jar
        runTest.cmd
        ```

      - Ensure that you have a Java 2 runtime environment at version 1.3 or greater on your workstation installed. You can verify your version with the command:

        ```
        java -version
        ```

- Ensure that you have the file `j2ee.jar` accessible on your workstation. If not, you can obtain it if you install the CICS development deployment tool, or by installing the Java 2 SDK, Enterprise Edition, available from `http://java.sun.com` as an alternate possibility.
- Invoke TraderTest using the `runtest.cmd` file. You will need to alter the input parameters as documented in the file. For further details and for an example of expected output, refer to Example 7-2 on page 198.

7.2 TraderBean development with VisualAge for Java

The next sections of this chapter describe the steps necessary to design and code our Trader the enterprise bean. If you are not familiar with enterprise bean development in VisualAge for Java (VAJ), you should first look at Chapter 6, “Developing a HelloWorld session bean for CICS” on page 135, which gives basic guidance on developing enterprise beans with VAJ.

The following steps are necessary to implement our enterprise bean:

1. Define the business methods of the enterprise bean.
2. Design the enterprise bean structure.
3. Implement the interface TraderBackend.
4. Implement CompaniesBean.
5. Implement QuotesBean.
6. Implement TraderBean.
7. Implement TraderBackendJcics.

7.2.1 Define the business methods of the enterprise bean

To design our enterprise bean, we have to define the business methods the bean provides. These correspond to the following business functions provided by the COBOL application:

GET-COMPANY	Query the list of companies
SHARE-VALUE	Retrieve current stock quote from file
BUY-SELL	Trader shares in a given company

To improve the usability of the enterprise bean, we decided that, instead of providing one business method for trading, we would define both a buy and a sell method.

Trader also provides a 3270 panel to logon to the application. The userid and password supplied are not verified with CICS, but the user ID is required when generating the stock count during the buy/sell operation. Rather than supply the user ID with each call, we decided to store the supplied user ID in an instance variable.

These are the methods we decided to implement in our TraderBean.

`logon()`	To logon to the Trader application.
`logoff()`	To logoff from the application.
`getCompanies()`	To query the companies to trade with.
`getQuotes()`	To retrieve quotes for a specific company.
`buy()`	To buy shares of a given company.
`sell()`	To sell shares of a given company.

7.2.2 Design the enterprise bean structure

After we had defined which business methods the enterprise bean must implement, we designed the enterprise bean structure. Because we wanted to be able to call the COBOL Trader application using both the JCICS `link()` method and the CICS Connector, we decided to separate the implementation details of calling the COBOL program from the enterprise bean and to implement the access to the COBOL program in its own classes, which must follow a specific interface. This approach has the advantage that, for each different technique of calling Trader, only minor changes are necessary to the enterprise bean, and the implementation details of calling Trader are hidden in the interface class.

We named the interface class *TraderBackend*. All classes which hide the implementation details of accessing Trader have to implement TraderBackend. The interface class defines the following methods:

- `logon()`
- `logoff()`
- `getCcompanies()`
- `getQuotes()`
- `buy()`
- `sell()`
- `ejbBackendBackend()`
- `ejbBackendRemove()`
- `ejbBackendActivate()`
- `ejbBackendPassivate()`

As you can see, these are very similar to the methods we defined for the enterprise bean. You will later see that the arguments of the methods differ slightly from those of the enterprise bean.

You can also see that we defined a `logon()` method even though this functionality is not directly supported by Trader. As we will explain later, we need this method to provide some data for the class implementing the access to Trader with CICS Connector.

The reason why we have added the methods `ejbBackendCreate()`, `ejbBackendRemove()`, `ejbBackendActivate()`, and `ejbBackendPassivate()` is that the back-end classes will need to be informed about these events, when using the DB2 versions of the Traderbean. However, in this version they are only present for compatibility reasons.

We named the class which implements the TraderBackend interface *TraderBackendJcics*. Figure 7-3 shows the relation between TraderBean and TraderBackendJcics.

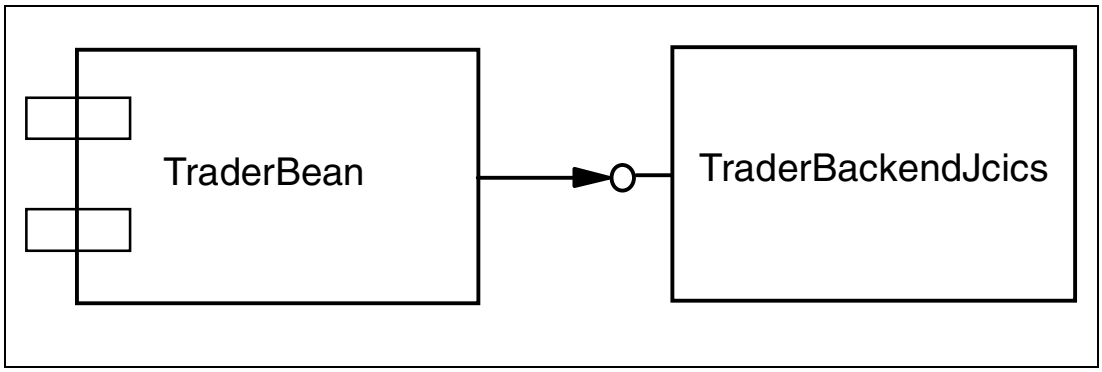


Figure 7-3 Relation between TraderBean and TraderBackendJcics

7.2.3 Implement the interface TraderBackend

The first class we need to create is the interface class TraderBackend. As we will later see, TraderBean has an instance variable referring to an instance of a class implementing this class. Such an instance will be TraderBackendJcics.

All the classes for this example will be stored in the VAJ group *ITSO EJB 390,* that we created in our HelloWorld session bean in Chapter 6. The package name of all classes related to the enterprise bean will be *itso.ejb390.trader*.

TraderBackend is declared as:

```
public interface TraderBackend extends java.io.Serializable {}
```

As you can see, TraderBackend extends java.io.Serializable. This is necessary because TraderBean is a stateful session bean, as is it required to store the conversational state of the COBOL Trader applications across different method calls. All non-transient instance variables of a stateful session bean can be passivated by the container, thus the need for session beans to implement java.io.Serializable.

TraderBackend contains the following methods:

```
public void logon(String userID, String password,
                  String connectURL, String cicsServer)
```

userID	The user ID of the person using the Trader application
password	The password of the person
connectURL	Used only for the CICS Connector in order to specify the URL of the CICS Transaction Gateway -- can be ignored for our JCICS implementation
cicsServer	Used only for the CICS Connector in order to specify the name of the CICS server

Note: We have invented the parameters *connectURL* and *cicsServer* `in order` to have the possibility to specify appropriate values from an user interface. In a real production environment you would either have these parameters coded directly in the class accessing the back-end, or you would read the parameters from a properties file. In this case you would skip the two parameters from logon and just provide `userID` and `password`.

- `public void logoff()`

  This method is used to logoff from the application.

- `public CompaniesBean getCompanies()`

  This method takes no arguments, but returns an instance of class CompaniesBean. This class holds all companies' Trader returns. The class is explained in more detail in 7.2.4, "Implement CompaniesBean" on page 177.

- `public QuotesBean getQuotes( String company, String userID )`

company	The company for which the quotes are obtained
userID	The user ID of the person using the Trader application

The method returns an instance of class QuotesBean. This bean holds quote information of the specified company. The class is explained in more detail in 7.2.5, "Implement QuotesBean" on page 177.

- `public void buy( String company, String userID, int numberOfShares )`

company	The company for which shares are bought
userID	The user ID of the person using the Trader application
numberOfShares	The number of shares to buy

- `public void sell( String company, String userID, int numberOfShares )`

company	The company for which shares are sold
userID	The user ID of the person using the Trader application
numberOfShares	The number of shares to sell

Besides our business methods, we also need to define the control methods:

- `public void ejbBackendCreate()`
- `public void ejbBackendRemove()`
- `public void ejbBackendActivate()`
- `public void ejbBackendPassivate()`

7.2.4 Implement CompaniesBean

CompaniesBean is a class which holds all companies returned by Trader. It is declared as illustrated in Figure 7-4. It implements methods to add a company, to get a company, and to return the number of companies actually held by the bean.

```
public class CompaniesBean implements java.io.Serializable {
  private java.util.Vector companies = new java.util.Vector();
}
```

Figure 7-4 Declaration of class CompaniesBean

7.2.5 Implement QuotesBean

QuotesBean is a class which holds all quote specific information returned by Trader. It is declared as illustrated in Figure 7-5. For each instance variable, the bean provides a get and set method.

```
public class QuotesBean implements java.io.Serializable {
  private java.lang.String unitSharePrice;
  private java.lang.String totalShareValue;
  private java.lang.String unitValue1Days;
  private java.lang.String unitValue2Days;
  private java.lang.String unitValue3Days;
  private java.lang.String unitValue4Days;
  private java.lang.String unitValue5Days;
  private java.lang.String unitValue6Days;
  private java.lang.String unitValue7Days;
  private java.lang.String commissionCostSell;
  private java.lang.String commissionCostBuy;
  private java.lang.String numberOfShares;
}
```

Figure 7-5 Declaration of QuotesBean

Similar to *CompaniesBean*, *QuotesBean* is returned by method `getQuotes()` of TraderBean's remote interface class Trader. Therefore, it is also necessary to implement `java.io.Serializable`, as previously described for CompaniesBean.

7.2.6 Implement TraderBean

Now that we have implemented TraderBackend we are able to write TraderBean. Most of the hard work is implemented in the back-end classes so the implementation of TraderBean is actually quite simple. To create TraderBean follow the steps below:

1. In VisualAge for Java, click the tab **EJB**.
2. Click **EJB -> Add -> EJB Group** to open the *Add EJB Group SmartGuide*.
3. As the project, select *ITSO EJB 390 Redbook*.
4. For the EJB group name specify *ITSOEJB390*.
5. Finally, click **Finish** to create the EJB group.

Now that we have defined EJB group ITSOEJB390, create the enterprise bean TraderBean.

1. Click the right mouse button on EJB group ITSOEJB390 and select **Add -> Enterprise Bean** to open the *Create EnterpriseBean SmartGuide*.
2. Enter Trader as the bean name.

   Leave all other fields as to default, which should be as follows:

BeanType	Session bean
Create a new bean class	selected
Project	ITSO EJB 390 Redbook
Package	itso.ejb390.trader
Class	TraderBean

3. Click **Next** to verify the bean's interfaces.

   The *SmartGuide* proposes TraderHome for the home interface and Trader for the remote interface. Keep the defaults and click **Finish** to create the enterprise bean and its interface classes.

As explained earlier, TraderBean must be a stateful session bean in order to keep the user ID and the reference to TraderBackend. To change the properties of the enterprise bean, click right mouse button on TraderBean and select **Properties**. The Properties window for ITSOEJB390 opens. Change the State Management Attribute to #STATEFUL and click **OK** to apply your changes and to close the window.

Now we need to define the instance variables for TraderBean. We need one variable referencing a TraderBackend interface and one to hold the user ID. Therefore, we need to define the following instance variables:

```
private TraderBackend ivTraderBackend = null;
private java.lang.String ivUserID = "";
```

The ivUserID instance variables also requires a getter and setter method since a client application should be able to retrieve the user ID. Right-click on the `ivUserID` variable and select **Generate -> Accessors** to create the `getUserID()` and `setUserID()` methods. Once these have been created, right-click the method `getUserID()` and select **Add To -> EJB Remote Interface**.

If you now select interface class *Trader*, you can see that VAJ has added the method `getUserID()` to this class.

Modify TraderBean.ejbCreate()

The `ejbCreate()` method needs to instantiate the back-end class TraderBackend and to keep a reference to it. Because the method has to know which kind of back-end needs to be instantiated, we have to pass the back-end type as a parameter. Therefore, we need to modify the existing `ejbCreate()` method by adding the parameter `type` as an argument.

Note: By default, VisualAge for Java creates a new method when the parameter list of an existing method is changed. Therefore, after you have saved your changes, you will have two `ejbCreate()` methods. One is the old one with no arguments, and one the new one with `type` as an input parameter. To prevent this behavior, you can do a *Save replace*, using Ctrl-Shift-S.

The modified `ejbCreate()` method is illustrated in Figure 7-6.

```
public void ejbCreate(String type) throws javax.ejb.CreateException,
                                          java.rmi.RemoteException {

try {
  // load backend class to handle requests
  loadClass(type);
  ivTraderBackend.ejbBackendCreate();
} catch( Exception e ) {
  throw new CreateException( e.getMessage() );
}

}
```

Figure 7-6 TraderBean.ejbCreate()

Method `loadClass()` is a private method of TraderBean which gets as input a type which specifies which type of back-end access technique should be used. Type is a String which, in this example, represents *TraderBackendJcics*, but will be used in later scenarios to specify other back-end classes such as *TraderBackendCICSConnector*. Figure 7-7 shows the implementation of the `loadClass()` method.

```
private void loadClass(String type) throws Exception {

Class loadClass=null;

if( type.equalsIgnoreCase("JCICS-COBOL") == true ) {
  loadClass =
    Class.forName("itso.ejb390.trader.TraderBackendJcics");
}
else {
  throw new TraderException( "You specified unknown type " + type );
}

ivTraderBackend = (TraderBackend)loadClass.newInstance();

}
```

Figure 7-7 Method TraderBean.loadClass()

Within this class, the method `TraderException()` has been defined. `TraderException()` extends class *Exception* and is used to throw exceptions which are related to the Trader application. Its implementation is illustrated in Figure 7-8.

```
public class TraderException extends Exception {
  public TraderException() {
    super();
  }
  public TraderException(String s) {
    super(s);
  }
}
```

Figure 7-8 TraderException

The next step is to modify the remaining control methods `ejbRemove()`, `ejbActivate()`, and `ejbPassivate()`.

Modify the remaining control methods

The remaining control methods just need to invoke the corresponding method of the back-end interface, as shown in Figure 7-9.

```
public void ejbRemove() throws java.rmi.RemoteException {
  ivTraderBackend.ejbBackendRemove();
}

public void ejbActivate() throws java.rmi.RemoteException {
  ivTraderBackend.ejbBackendActivate();
}

public void ejbPassivate() throws java.rmi.RemoteException {
  ivTraderBackend.ejbBackendPassivate();
}
```

Figure 7-9 Control methods of TraderBean

Finally, we will define the business methods `logon()`, `getCcompanies()`, `getQuotes()`, `sell()`, and `buy()`. We begin with `logon()`.

TraderBean.logon()

Method `logon()` needs to do two things:

1. Store the provided user ID in a class instance variable.
2. Invoke the `logon() method` of the back-end instance.

How this is done is illustrated in Figure 7-9.

```
public void logon(String userID, String password,
              String connectURL, String cicsServer) throws Exception {

// save userID
ivUserID = userID;

// now logon to system
ivTraderBackend.logon(userID, password, connectURL, cicsServer);

}
```

Figure 7-10 Method TraderBean.logon()

Because `logon()` is a method which a client application needs to call, we need to add it to the enterprise bean remote interface in the same way as we have done for `getUserID()`.

TraderBean.logoff()

This method just needs to invoke the `logoff()` method of the back-end instance, as shown in Figure 7-11.

```
public void logoff() throws Exception {

ivTraderBackend.logoff();

}
```

Figure 7-11 Method TraderBean.logoff()

Now that we have implemented method `logoff()`, we will implement the remaining business methods.

The remaining business methods of TraderBean

The methods `getCompanies()`, `getQuotes()`, `buy()`, and `sell()` are quite simple to implement. They do nothing else other than to call the appropriate method of TraderBackend. The methods and their implementations are shown in Figure 7-12. After defining the methods, it is then necessary to add them to Trader's remote interface.

```
public CompaniesBean getCompanies() throws Exception {
  return ivTraderBackend.getCcompanies();
}

public QuotesBean getQuotes( String company ) throws Exception {
  return ivTraderBackend.getQuotes(company,ivUserID);
}

public void buy( String company, int numberOfShares )
                                                    throws Exception{
  ivTraderBackend.buy(company,ivUserID,numberOfShares);
}

public void sell( String company, int numberOfShares )
                                                    throws Exception{
  vTraderBackend.sell(company,ivUserID,numberOfShares);
}
```

Figure 7-12 Business methods of TraderBean

7.2.7 Implement TraderBackendJcics

At this point we have implemented most of the classes necessary for the enterprise bean. What remains is to write the class which actually invokes the COBOL Trader program using the `link()` method of the JCICS Program object. We shall call this class *TraderBackendJcics* which implements interface class *TraderBackend*. The declaration of TraderBackendJcics is shown in Figure 7-13.

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

public class TraderBackendJcics implements TraderBackend {
  private final static int NUM_OF_COMPANIES = 4;
  private final static String TransactionID = "TRB2";
  private final static String ProgramName = "TRADERBL";
}
```

Figure 7-13 Declaration of TraderBackendJcics

The class defines the following three constants:

NUM_OF_COMPANIES	The number of companies returned by the COBOL program. Trader always returns four companies.
TransactionID	The CICS mirror transaction id used for the link to the COBOL Trader program.
ProgramName	The CICS program name of the Trader application.

As you can see, TraderBackendJcics imports class com.ibm.cics.server.Program. This class is defined in the JCICS API, which does not come with VisualAge for Java. Therefore, it is necessary to import the JCICS package (`com.ibm.cics.server`) into VisualAge. The JCICS classes are archived in the file `dfjcics.jar`, which is supplied with the CICS TS install in the HFS directory `$CICS_HOME/lib`. We downloaded this file to our workstation, created a new VAJ project called *JCICS*, and imported the JAR file into this project.

Now that we have declared our instance variables, we are ready to implement the methods invoking Trader. This is explained in the next sections.

Control methods for TraderBackendJcics

As we already described, `ejbBackendCreate()`, `ejbBackendRemove()`, `ejbBackendActivate()`, and `ejbBackendPassivate()` need to do nothing special in order to wrap Trader. Therefore, their implementation is quite simple, as shown in Figure 7-14.

```
public void ejbBackendCreate() throws javax.ejb.CreateException,
                                java.rmi.RemoteException {}

public void ejbBackendRemove() throws java.rmi.RemoteException {}

public void ejbBackendActivate() throws java.rmi.RemoteException {}

public void ejbBackendPassivate() throws java.rmi.RemoteException {}
```

Figure 7-14 Control methods of TraderBackendJcics

TraderBackendJcics.logon()

As we discussed earlier, method `logon()` is only necessary for the class implementing access to Trader through the CICS Connector. Since on our system, the enterprise bean and the COBOL Trader application are installed on the same CICS region, it is not necessary to specify the system ID. If you plan to link to a COBOL program running in another CICS region, you could store the system ID specified by the `cicsServer` parameter of the `logon()` method and use this value later for invocations of your COBOL program.

The implementation of method `logon()` is illustrated in Figure 7-15. As you can see, the method has no logic implemented.

```
public void logon(String userID, String password,
                  String connectURL, String cicsServer) {
}
```

Figure 7-15 TraderBackendJcics.logon()

TraderBackendJcics.logoff()

For this back-end implementation, no business logic is necessary to logoff. Therefore the method is quite simple, as illustrated in Figure 7-16.

```
public void logoff() {
}
```

Figure 7-16 TraderBackendJcics.logoff()

TraderBackendJcics.getCompanies()

Trader always expects one specific COMMAREA as input and returns the same COMMAREA structure as response. Therefore we need a call which allows us to pass a COMMAREA as input and returns a COMMAREA as output.

Calling a CICS program when using the JCICS class library is done with the method `link()` of the Program object. There are several implementations of `link()`. One is without parameters which is used when the called program exchanges no data through the COMMAREA.

Another variation of `link()` is with two parameters, taking `com.ibm.record.IByteBuffer` as an argument. This is the link method we use for our sample because this method takes the first parameter as input and returns in the second parameter the response of the called program. Therefore we need a Java class representing the COMMAREA which Trader expects as input, and that implements the `IByteBuffer` interface.

VisualAge for Java Enterprise Edition provides the Enterprise Access Builder (EAB), which enables you to convert a COBOL data structure to a Java class. Such a Java class extends com.ibm.record.CustomRecord, which implements indirectly through other classes IByteBuffer. Therefore the EAB generates a class we can use as a parameter for method `Program.link()`. We named the Java class representing the COMMAREA *TraderRecord*. The COMMAREA which Trader uses for input and output is listed in Figure 7-17, and is also supplied as the file `commarea.txt` in the source code accompanying this book.

```
       01 COMMAREA-BUFFER.
          03 REQUEST-TYPE              PIC X(15).
          03 RETURN-VALUE              PIC X(02).
          03 USERID                    PIC X(60).
          03 USER-PASSWORD             PIC X(10).
          03 COMPANY-NAME              PIC X(20).
          03 CORRELID                  PIC X(32).
          03 UNIT-SHARE-VALUES.
             05 UNIT-SHARE-PRICE       PIC X(08).
             05 UNIT-VALUE-7-DAYS      PIC X(08).
             05 UNIT-VALUE-6-DAYS      PIC X(08).
             05 UNIT-VALUE-5-DAYS      PIC X(08).
             05 UNIT-VALUE-4-DAYS      PIC X(08).
             05 UNIT-VALUE-3-DAYS      PIC X(08).
             05 UNIT-VALUE-2-DAYS      PIC X(08).
             05 UNIT-VALUE-1-DAYS      PIC X(08).
          03 COMMISSION-COST-SELL      PIC X(03).
          03 COMMISSION-COST-BUY       PIC X(03).
          03 SHARES.
             05 NO-OF-SHARES           PIC X(04).
          03 SHARES-CONVERT REDEFINES SHARES.
             05 NO-OF-SHARES-DEC       PIC 9(04).
          03 TOTAL-SHARE-VALUE         PIC X(12).
          03 BUY-SELL1                 PIC X(04).
          03 BUY-SELL-PRICE1           PIC X(08).
          03 BUY-SELL2                 PIC X(04).
          03 BUY-SELL-PRICE2           PIC X(08).
          03 BUY-SELL3                 PIC X(04).
          03 BUY-SELL-PRICE3           PIC X(08).
          03 BUY-SELL4                 PIC X(04).
          03 BUY-SELL-PRICE4           PIC X(08).
          03 ALARM-CHANGE              PIC X(03).
          03 UPDATE-BUY-SELL           PIC X(01).
          03 FILLER                    PIC X(15).
          03 COMPANY-NAME-BUFFER.
             05  COMPANY-NAME-TAB OCCURS 4 TIMES
                 INDEXED BY COMPANY-NAME-IDX PIC X(20).
```

Figure 7-17 COMMAREA of COBOL program TRADERBL

Note that before you can use the EAB, you need to add following features to your VisualAge workspace:

- IBM Enterprise Access Builder Library
- IBM Java Record Library

The following instructions explain how to create TraderRecord.

1. Select **Workspace -> Tools -> Enterprise Access Builder -> Import COBOL to Record Type** to open the *Import COBOL to Record Type SmartGuide*.
2. Click **Browse** to open the file dialog.
   a. Navigate through your directory structure to select the file which contains the COBOL definition of the COMMAREA.
   b. Click **Open** to select the file and close the file dialog window.
3. Make sure that you import the COBOL definition for a CICS Transaction.
4. Click **Next**.
5. Now you can select one or more 01 COMMAREA levels to be imported by the *SmartGuide*. All available 01 levels are listed on the left side of the window. Because our COBOL data structure has only one 01 level, just COMMAREA-BUFFER is shown.
6. Select *COMMAREA-BUFFER* and click the **>** arrow in the middle of the window as illustrated in Figure 7-18.

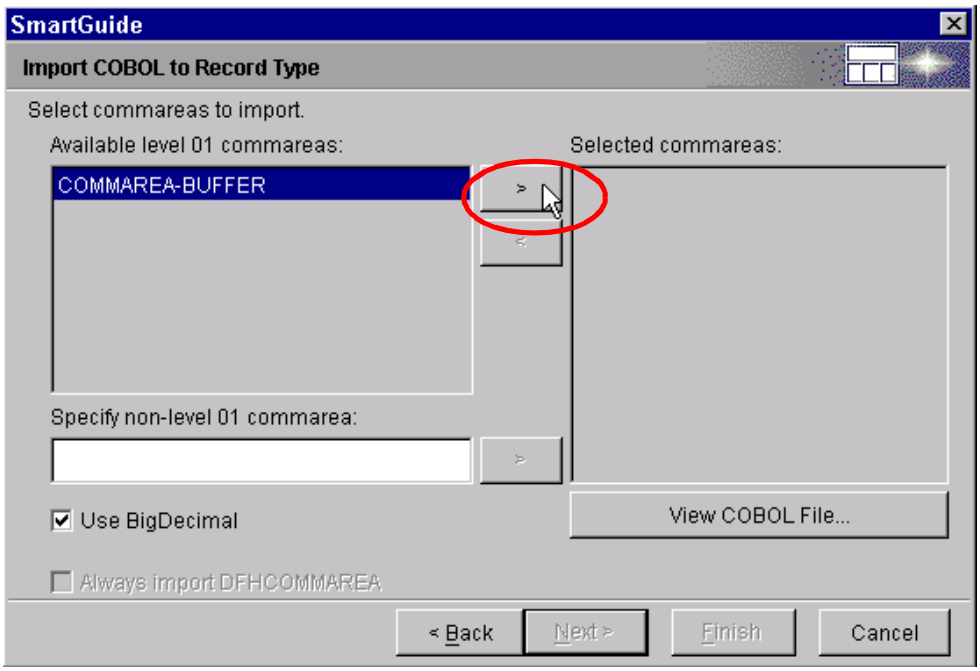


Figure 7-18 Selecting the 01 level of COMMAREA

7. Now *COMMAREA-BUFFER* switches to the right side which shows the selected COMMAREAS. Click **Next**.
8. As the project name type *ITSO EJB 390 Redbook* and for the package name enter *itso.ejb390.trader*. You can also use the **Browse** buttons to find the project and packages.
9. For the class name enter *TraderRecordType*. Your window should now look similar to Figure 7-19.

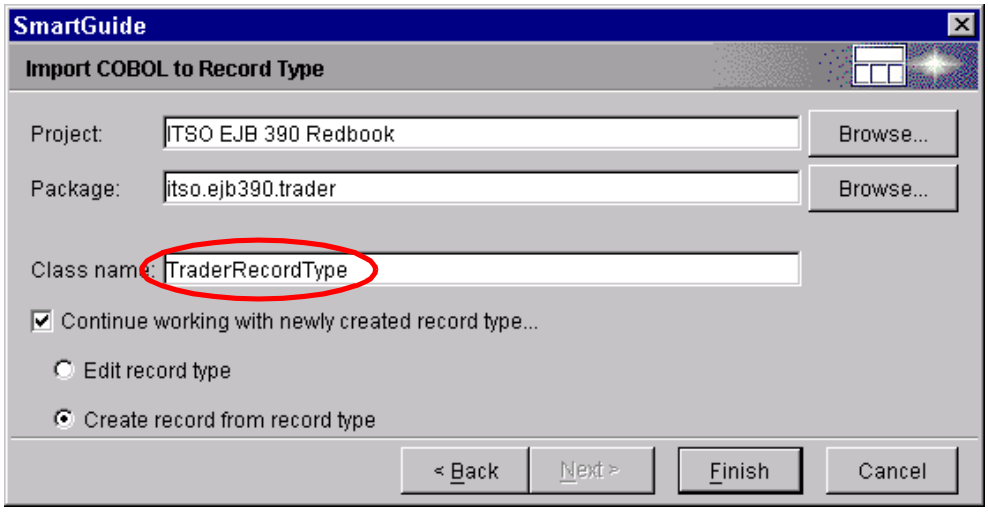


Figure 7-19 Specifying the class name for COBOL Record Type

10. Click **Finish** to create the record type and to create a record from the record type.

Now a *SmartGuide* opens which allows you to create a record from a record type. This *SmartGuide* will create the class *TraderRecord*, which is a Java class representing the COBOL COMMAREA. This is shown with the next steps.

1. As class name, type *TraderRecord*. Leave all other options as they are (Figure 7-20).

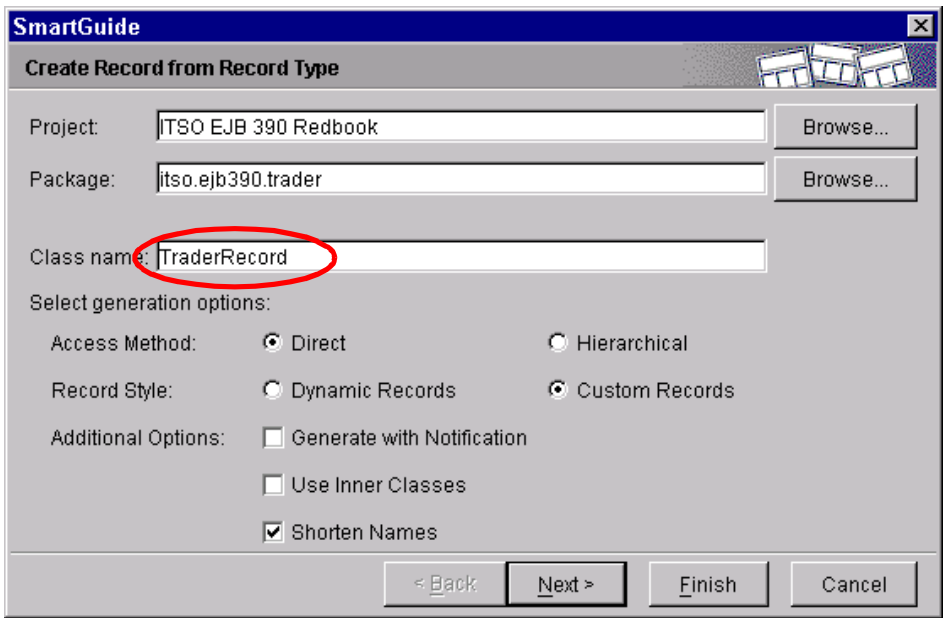


Figure 7-20 Specifying class name TraderRecord representing the COMMAREA

2. Click **Next** to modify the properties of the record.
3. Make the following changes:

Floating Point Format	IBM
Remote Integer Endian	Big Endian
Endian	Big Endian
Code Page	037
Machine Type	MVS

The window should now look as shown in Figure 7-21.

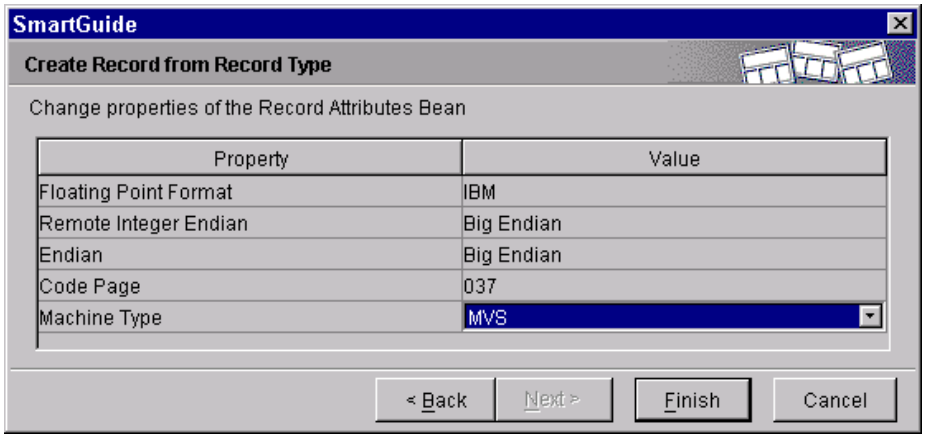


Figure 7-21 Changing the properties of TraderRecord

4. Click **Finish** to create TraderRecord.

> **Note:** If you specify an EBCDIC code page such as 037 for conversion of character data, you should ensure that you do not also use the CICS DFHCNV templates to convert the COMMAREA data from ASCII to EBCDIC, otherwise you will experience corruption of data due to double conversion. For more details on data conversion with Java in CICS refer to the redbook *Revealed! Architecting Web Access to CICS*, SG24-5466.

If you now view the classes of package itso.ejb390.trader, you will see that VisualAge has added the classes TraderRecordType, TraderRecord, and TraderRecordBeanInfo. TraderRecord is the class which we can use as representation of our COBOL COMMAREA.

Now that we have created TraderRecord, we can implement the method `getCompanies()`. The code needs to do the following:

- Instantiate a CompaniesBean.
- Instantiate a TraderRecord for input and for output.
- Set the request type to `Get_Company` in the input COMMAREA.
- Instantiate a `com.ibm.cics.server.Program`.
- Invoke method `Program.link()` by providing the TraderRecords as parameters for input and output.
- Iterate over company name array of the output COMMAREA and copy the companies to CompaniesBean.
- Return the CompaniesBean instance.

Figure 7-22 shows the implementation of method `getCompanies()`.

```
public CompaniesBean getCompanies() throws Exception {

  CompaniesBean companies = new CompaniesBean();

  // allocate commarea
  TraderRecord traderRecordIn = new TraderRecord();
  TraderRecord traderRecordOut = new TraderRecord();

  // set request type
  traderRecordIn.setRequest__Type("Get_Company");

  // allocate program and set TxID and program name
  Program program = createProgram();

  // call program
  program.link(traderRecordIn, traderRecordOut );

  // copy companies to CompaniesBean
  for (int i = 0; i < NUM_OF_COMPANIES; i++) {
    companies.addCompany(traderRecordOut.getCompany__Name__Tab(i));
  }
return companies;
}
```

Figure 7-22 TraderBackendJcics.getCompanies()

The hard work has now been done. As you will see, implementing the other methods is relatively straightforward and they all look fairly similar.

TraderBackendJcics.getQuotes()

This method is implemented in a very similar way as `getCompanies()` and does the following:

- Instantiate a QuotesBean.
- Instantiate a TraderRecord for input and for output.
- Set the request type to `Share_Value` in the input COMMAREA.
- Instantiate a `com.ibm.cics.server.Program`.
- Invoke method `Program.link()` by providing the TraderRecords as input and output.
- Copy the results from the output COMMAREA to QuotesBean.
- Return the QuotesBean instance.

Figure 7-23 shows the implementation of method `getQuotes()`.

```
public QuotesBean getQuotes( String company, String userID )
                                    throws Exception {

  QuotesBean quotes = new QuotesBean();
  // allocate commarea
  TraderRecord traderRecordIn = new TraderRecord();
  TraderRecord traderRecordOut = new TraderRecord();
  // set request type
  traderRecordIn.setRequest__Type("Share_Value");
  // set additional parameters
  traderRecordIn.setCompany__Name( company );
  traderRecordIn.setUserid( userID );
  // allocate program and set TxID and program name
  Program program = createProgram();
  // call Transaction
  program.link(traderRecordIn, traderRecordOut );

  // copy results
  quotes.setUnitSharePrice( traderRecordOut.getUnit__Share__Price() );
  quotes.setUnitValue1Days( traderRecordOut.getUnit__Value__1__Days() );
  quotes.setUnitValue2Days( traderRecordOut.getUnit__Value__2__Days() );
  quotes.setUnitValue3Days( traderRecordOut.getUnit__Value__3__Days() );
  quotes.setUnitValue4Days( traderRecordOut.getUnit__Value__4__Days() );
  quotes.setUnitValue5Days( traderRecordOut.getUnit__Value__5__Days() );
  quotes.setUnitValue6Days( traderRecordOut.getUnit__Value__6__Days() );
  quotes.setUnitValue7Days( traderRecordOut.getUnit__Value__7__Days() );
  quotes.setCommissionCostSell(traderRecordOut.getCommission__Cost__Sell());
  quotes.setCommissionCostBuy( traderRecordOut.getCommission__Cost__Buy() );
  quotes.setNumberOfShares( traderRecordOut.getNo__Of__Shares() );
  quotes.setTotalShareValue( traderRecordOut.getTotal__Share__Value() );

return quotes;
}
```

Figure 7-23 TraderBackendJcics.getQuotes()

TraderBackendJcics.buy()

This method needs to do the following:

- Instantiate a TraderRecord for input and for output.
- Set the request type to Buy_Sell in the input COMMAREA.
- Set company, userID, and number of shares in the input COMMAREA.
- Set a flag if for buying or to selling shares in the input COMMAREA.
- Instantiate a com.ibm.cics.server.Program.
- Invoke the `Program.link()` method by providing the TraderRecords as parameters for input and output.

The difference to the other methods is, that this request has additional arguments in the input COMMAREA. Further this method does not return any results. Therefore it is declared as void.

Because method `buy()` and `sell()` are very similar, we created the private method `trade()` which invokes Trader. Methods `buy()` and `sell()` call this private method and provide a flag to be able to distinguish between buy and sell.

Figure 7-24 shows the implementation of method `trade()`.

```
private void trade( String company, String userID,
  int numberOfShares, boolean buy ) throws Exception {

  // allocate commarea
  TraderRecord traderRecordIn = new TraderRecord();
  TraderRecord traderRecordOut = new TraderRecord();

  // set request type
  traderRecordIn.setRequest__Type("Buy_Sell");

  // set additional parameters
  traderRecordIn.setCompany__Name(company);
  traderRecordIn.setUserid(userID);
  traderRecordIn.setNo__Of__Shares__Dec( (short)numberOfShares );

  if( buy == true ) // if buy
    traderRecordIn.setUpdate__Buy__Sell("1");
  else // if sell
    traderRecordIn.setUpdate__Buy__Sell("2");

  // allocate program and set TxID and program name
  Program program = createProgram();

  // call Transaction
  program.link(traderRecordIn, traderRecordOut );

}
```

Figure 7-24 TraderBackendJcics.trade()

The implementation of method `buy()` is now quite simple, as illustrated in Figure 7-25.

```
public void buy( String company, String userID, int numberOfShares )
                                      throws Exception {

  trade( company, userID, numberOfShares, true );

}
```

Figure 7-25 TraderBackendJcics.buy()

TraderBackendJcics.sell()

Method `sell()` is very similar to method `buy()`. It calls internal method `trade()` as shown in Figure 7-26.

```
public void sell( String company, String userID, int numberOfShares )
                                      throws Exception {

  trade( company, userID, numberOfShares, false );

}
```

Figure 7-26 TraderBackendJcics.sell()

At this point we have completed the JCICS implementation of TraderBean. The next sections show how to deploy the enterprise bean to our CICS TS V2.1 region.

7.3 Deploying the TraderBean to CICS

This section describes how to deploy Trader enterprise bean and its related classes to a CICS TS V2.1 region. This task involves the following steps:

1. Exporting the enterprise bean and its related classes.
2. Converting the exported file to a CICS-deployed JAR file.
3. Sending the deployed JAR file to OS/390
4. Defining a DJAR in the CICS system.
5. Sending the supporting JAR files to OS/390.
6. Adding the supporting JAR files to CICS region's trusted middleware classpath.
7. Restarting the CICS JVM environment.
8. Publishing the Trader enterprise bean to the COS Naming server.

For details of how we tested the enterprise bean, refer to Section 7.4, "Testing the enterprise bean" on page 196.

7.3.1 Exporting the enterprise bean and its related classes

This section shows how to export the enterprise bean and its related classes.

1. Within VisualAge for Java, select the **EJB** tab to view the EJB groups. Select group *ITSOEJB390* and click the right mouse button.
2. Select **Export -> EJB JAR** to open the *Export to an EJB JAR File SmartGuide*.
3. If you now click **Details** besides .class you can see that only the three classes, *Trader*, *TraderBean*, and *TraderHome* are selected. Close the window and click **Select referenced types and resources**. VisualAge for Java has now also selected classes which are referenced by the enterprise bean.
4. Click again **Details** besides .class to see which classes these are. You will see that in addition to the previous three classes, the four classes, *CompaniesBean*, *QuotesBean*, *TraderBackend*, and *TraderException* are selected.

   But VAJ has not selected the classes *TraderBackendJcics*, *TraderRecord*, *TraderRecordBeanInfo*, and *TraderRecordType*. The reason for this is, that TraderBackend does not directly instantiate *TraderBackendJcics*, but loads the class dynamically. VAJ does not know this and therefore does not select these classes. The reason why we decided to load the classes dynamically rather than directly instantiating them, was that we will later run the CICS Connector version of this bean within WebSphere for Windows NT. In an Windows NT environment we cannot use the JCICS API and therefore do not want to have any reference to TraderBackendJcics.
5. Select the following additional classes so that they will also be exported.
   - TraderBackendJcics
   - TraderRecord
   - TraderRecordBeanInfo
   - TraderRecordType

   Click **OK** to close the window.
6. In the export window, you should now see that one bean and eleven classes are selected.

7. Specify path and file name for the JAR file. We created the sub-directory `C:\itsotrader` and exported to the file `C:\itsotrader\traderForCICS.jar`. Now your window should look as shown in Figure 7-27.
8. Click **Finish** to export the classes to file `traderForCICS.jar`.

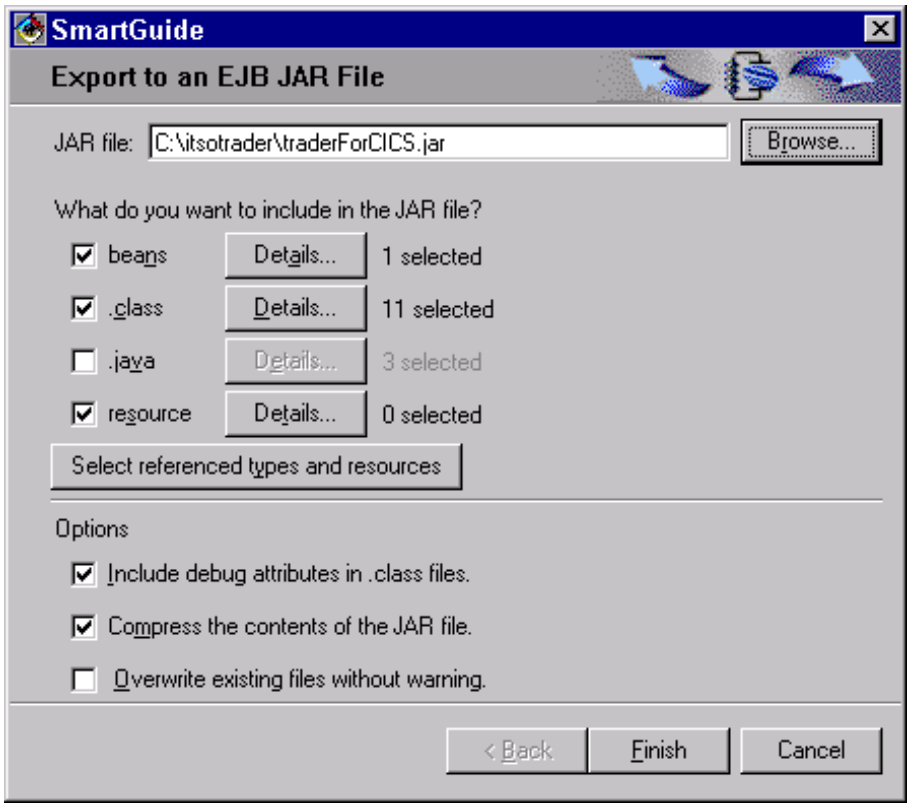


Figure 7-27 Exporting TraderBean for CICS

Note: When creating our EJB JAR file, we packaged all the required classes into one JAR file. This is convenient for testing, but has the disadvantage that as our application grows it becomes increasingly harder to package all the necessary classes into the JAR. A more manageable approach would be to break the application into a series of smaller JAR files that could be updated as required.

7.3.2 Converting the exported file to a deployed JAR file

To convert the exported JAR file to a deployed JAR file, it is necessary to use the CICS JAR development tool for EJB Technology. This generates a deployed JAR file specifically for CICS, and converts from the deployment descriptor from an EJB V1.0 specification that VAJ creates, to an EJB V1.1 specification that CICS requires. For further details on installing and using the CICS JAR development tool, refer to Section 6.3.2, “Generating a CICS deployed JAR file” on page 148.

Note that, before we started the tool, we added the EAB (`eablib.jar`) and Java Record Framework classes (`recjava.jar`) to the classpath for the CICS JAR development tool. We made these modifications to the batch file `cicsjdt.bat`, which is used to invoke the CICS JAR development tool. Since the `cicsjdt.bat` file is supplied by CICS, we also created a backup of the original file before we made these changes.

```
rem Add the user's CLASSPATH
set CLASSPATH=%CLASSPATH%;%CURRENTCP%

SET VAJINSTALL=C:\PROGRA~1\IBM\VISUAL~1

set CLASSPATH=%VAJINSTALL%\eab\runtime30\eablib.jar;%CLASSPATH%
set CLASSPATH=%VAJINSTALL%\eab\runtime30\recjava.jar;%CLASSPATH%

rem ---------------------------------------------------------------
rem Invoke the CICS JAR Development Tool.
```

Figure 7-28 Modification of cicsjdt.bat

Now that we have changed the batch file, we can start the tool to generate the CICS-deployed JAR file:

1. Start the CICS JAR development tool using **Start -> Programs -> IBM CICS TS 2.1 Tools -> CICS JAR Development Tool for EJB Technology**.
2. Click **File -> Load** and enter the path and file name of the JAR file you have exported from VisualAge, which in our case is `c:\itsotrader\traderForCICS.jar`.
3. Click **Open** to load the JAR file to the tool.
4. Now you should see *Trader* below Current Enterprise Beans. Select *Trader*.
5. Click **File -> Generate** to generate the CICS-deployed JAR file.
6. You are asked to save your changes to a JAR file. Click **Save**.
7. You are asked to remove old EJB1.0 information. Click **Remove**.
8. Now you can specify an output EJB JAR file. Use the tool's default name, which in our case is `c:\itsotrader\traderForCICS_GEN.jar`.
9. Click **Generate**.

After a short time, the tool will generate `traderForCICS_GEN.jar`. Now we need to send the file to our OS/390 system.

Tip: If you receive a message from VAJ stating that *the file is not a zip file, or it is corrupted*, you should close the CICS Java development tool, or delete the output JAR file.

7.3.3 Sending the deployed JAR file to OS/390

On our OS/390 system we created the HFS directory `/u/cicsts21/djar`. Using FTP we transferred the deployed JAR file `traderForCICS_GEN.jar` in binary mode to this directory.

7.3.4 Defining the DJAR in the CICS system

This sample assumes that you have already defined to CICS a TCPIPSERVICE, CORBASERVER and REQUESTMODEL as detailed in Section 4.1.5, "Installing CICS resource definitions" on page 77. The only new definition that is required is a DJAR definition. This can be defined as using the following command:

```
CEDA DEFINE DJAR(TRADER) GROUP(ITSOEJB)
```

If you have an existing DJAR definition, this should be discarded beforehand using the command:

```
CEMT SET DJAR(TRADER) DISCARD
```

In the CEDA define panel enter the values for the **Corbaserver** and the **Hfsfile**, as shown in Figure 7-29.

```
DEFINE DJAR(TRADER) GROUP(ITSOEJB)
OVERTYPE TO MODIFY                                      CICS RELEASE = 0610
 CEDA  DEFine DJar( TRADER )
  DJar         ==> HWS
  Group        ==> ITSOEJB
  Description  ==> DJar CICS resource definitin for Trader EJB
  Corbaserver  ==> PJA5
  Hfsfile      ==> /u/cicsts21/djars/traderForCICS_GEN.jar
               ==>
               ==>
               ==>
               ==>

 DEFINE SUCCESSFUL
                                                SYSID=PJA5 APPLID=SCSCPJA5
```

Figure 7-29 3270 CICS screen to define DJAR attributes

Note: You need to define mixed case mode for your CICS terminal in order to enter the Hfsfile parameter. You can do this by entering the CICS transaction `CEOT TRANIDONLY`.

7.3.5 Sending supporting JAR files to OS/390

Since we used the EAB and the Java Record Framework in building the Trader enterprise bean, the JAR files `eablib.jar` and `recjava.jar` must be made available to the CICS region in order for the Trader enterprise bean to run properly. The `recjava.jar` file is included in the OS/390 JVM installation in the `lib/ext` sub-irectory, however the `eablib.jar` file is not included and therefore you need to transfer it from your workstation to a OS/390 HFS directory to make it accessible to the CICS JVM.

We transferred `eablib.jar` to the CICS directory `/u/cicsts21/lib`. If you have a default installation of VisualAge for Java, you will find the orginal file in the directory:

```
C:\Program Files\IBM\VisualAge for Java\eab\runtime30
```

7.3.6 Adding the supporting JAR files to the trusted middleware classpath

Now we have to add `eablib.jar` to the CICS JVMs trusted middleware classpath. To do this, we added the JAR file to the TMSUFFIX path defined in the CICS JVM profile. In our region, SCSCPJA5, this was defined in CICSSYSF.CICS610.DFHJVM(DFHJVMPR) and was as follows:

```
TMSUFFIX=/u/cicsts21/lib/eablib.jar:
```

7.3.7 Restarting the CICS JVM environment

When you make any changes to the JVM profile you need to ensure that the CICS JVM environment is restarted in order to pick up these changes. The easiest way to do this is to issue the command:

```
CEMT SET JVM PHASEOUT
```

Note: If you modify the CICS DJAR definition you will have to perform the following commands to refresh the CICS runtime definitions:

- Discard JDAR with `CEMT DISCARD DJAR(TRADER).`
- Install DJAR with `CEDA INSTALL DJAR(TRADER) GROUP(ITSOEJB).`

This command will also cause the CICS JVMs to be reinitialized, so you do not need to issue the `CEMT SET JVM PHASEOUT` command in this case.

7.3.8 Publishing the Trader enterprise bean

Now that DJAR TRADER is defined in CICS it needs to be published to the COS Naming Server. If you refresh the DJAR it does not need to be published again, unless the new DJAR contains different enterprise beans.

To publish the DJAR enter the following CICS command:

```
CEMT PERFORM DJAR(TRADER) PUBLISH
```

There is no easy way to list which beans have been published to the WebSphere COS Naming Server. However, the JNDI API is a standard API for accessing Naming Servers, and so we wrote our own Java class JNDIList, which takes as input the Naming Server URL and a JNDI Prefix. This can be run from any Java client, and allows you to either verify that the Naming Server is responding to requests, or that your enterprise bean has indeed been published to the Name Space.

Example 7-1 shows the output of the JNDIList utility querying all beans registered with the *ITSO* prefix on the COS Naming Server listening on port 900 on our WebSphere machine, *hecate*. The output shows the Home interfaces of our TraderBean and HelloWorldSession beans used in this redbook.

Example 7-1 Output of JNDIList utility

```
C:\itsotrader>Java JNDIList iiop://hecate:900/ITSO
/PJA5 Trader.itso\.ejb390\.trader\.EJSRemoteTraderHome
/PJA5 HelloWorldSession.itso\.ejb390\.helloworld\.EJSRemoteHelloWorldSessionHome
```

For more details on how to obtain the JNDIList utility, refer to Appendix C, “Using the additional material” on page 315.

7.4 Testing the enterprise bean

Now that we have created and deployed the Trader enterprise bean, it is now time to test it. Within VAJ we developed a simple stand-alone test client *TraderTest*, this is the same client that is used in “Quick start — Invoking TraderBean” on page 173. Following this we will then show you how we developed a servlet to allow the servlet to be invoked from a Web browser.

7.4.1 Developing a stand-alone test client: TraderTest

We decided to place the test program in the VAJ project *ITSO EJB 390 Redbook* and in package *itso.ejb390.trader.test.* We called the class *TraderTest*, and it is designed to perform the following steps:

1. Create an initial context.
2. Look up Trader bean's home interface.
3. Narrow to TraderHome.
4. Create Trader session bean.
5. Logon to the Trader application.
6. Get the companies list and display them.
7. Get quotes of last found company and display them.
8. Buy and sell shares.
9. Remove the bean.

In method `main()` we defined the name service factory, the URL of the COS Naming Server, and the name under which the bean is registered in the COS Naming Server. Further we defined the type of back-end, the connection URL, and the CICS server (Figure 7-30).

```
nameService = "com.ibm.ejs.ns.jndi.CNInitialContextFactory";
providerURL = "iiop://hecate:900/";
jndiName = "ITSO/PJA5/Trader";
type = "JCICS-COBOL";
connectURL = "local:";
cicsServer = "";
```

Figure 7-30 Name service definitions of Trader test program

Method `main()` creates an instance of TraderTest and passes the constructor `nameService`, `providerURL`, `jndiName`, `type`, `connectURL`, and `cicsServer`. The implementation of the constructor is shown in Figure 7-31.

```
// create initial context
java.util.Hashtable properties = new java.util.Hashtable(2);
properties.put("java.naming.factory.initial", nameService);
properties.put("java.naming.provider.url", providerURL);
javax.naming.InitialContext ctx =
                        new javax.naming.InitialContext(properties);

// lookup trader bean's home interface
Object obj = ctx.lookup(jndiName);

// narrow to traderHome
TraderHome traderHome = (TraderHome)javax.rmi.PortableRemoteObject.narrow(
                        (org.omg.CORBA.Object)obj, TraderHome.class);

// create Trader session bean
Trader trader = traderHome.create(type);

// logon to Trader
trader.logon( "Georg", "Password", connectURL, cicsServer );

// get companies and show them
CompaniesBean companies = trader.getCompanies();
java.util.Enumeration comps = companies.getCompanies();
String company = null;
while( comps.hasMoreElements() == true ) {
  company = (String)comps.nextElement();
  show( company );
}

// get quotes of last found company and show them
QuotesBean quotes = trader.getQuotes(company);
show( "CommissionCostBuy " + quotes.getCommissionCostBuy() );
show( "CommissionCostSell " + quotes.getCommissionCostSell() );
show( "NumberOfShares " + quotes.getNumberOfShares() );
show( "TotalShareValue " + quotes.getTotalShareValue() );
show( "UnitSharePrice " + quotes.getUnitSharePrice() );
show( "UnitValue1Days " + quotes.getUnitValue1Days() );
show( "UnitValue2Days " + quotes.getUnitValue2Days() );
show( "UnitValue3Days " + quotes.getUnitValue3Days() );
show( "UnitValue4Days " + quotes.getUnitValue4Days() );
show( "UnitValue5Days " + quotes.getUnitValue5Days() );
show( "UnitValue6Days " + quotes.getUnitValue6Days() );
show( "UnitValue7Days " + quotes.getUnitValue7Days() );

// buy and sell shares
show( "Now we buy 5 shares ... " );
trader.buy(company, 5 );
show( "... and sell 2 of them" );
trader.sell(company, 2 );

// remove bean
trader.remove();
```

Figure 7-31 Implementation of Trader test program

Now we are almost ready to test the Trader enterprise bean. The only thing missing is the client stubs. These are the classes a client application needs in order to invoke the enterprise bean. VisualAge for Java provides us with the ability to generate these classes in the following manner:

1. Click the **EJB** tab.
2. Right-click the EJB group **ITSOEJB390** and select **Generate Deployed Code**. VAJ will now generate all necessary stubs and ties.

Now, when you view the package *itso.ejb390.trader*, you will see the classes VisualAge for Java has created.

Running TraderTest within VAJ

Before you run the test program you have to set the correct classpath. Normally VAJ is able to compute the classpath for you, however, because the initial context factory is dynamically loaded, VAJ is not able to find the classpath for this class. However, it is possible to make VAJ set the classpath correctly. In your `main()` method, add following statement:

```
com.ibm.ejs.ns.jndi.CNInitialContextFactory dummy = new
                                  com.ibm.ejs.ns.jndi.CNInitialContextFactory();
```

1. Click class TraderTest with the right mouse button and select **Run -> Check Class Path**.
2. Click **Compute Now** and VisualAge will compute all necessary classpaths you need to run the test program.

You need to do this only once for the class. After VisualAge has computed the classpath, comment out the statement above.

To test the Trader enterprise bean, start the TraderTest program from within VisualAge for Java as follows:

1. Expand the project ITSO EJB 390 Redbook
2. Right-click the TraderTest class and select **Run**.

After it has finished, your VAJ console should show output similar to Example 7-2.

Example 7-2 Output of TraderTest program for JCICS-COBOL

```
Starting TraderTest application with following input:
  Name service: com.ibm.ejs.ns.jndi.CNInitialContextFactory
  Naming Server: iiop://hecate:900/
  JNDI name: ITSO/PJA5/Trader
  Call type: JCICS-COBOL
  CTG: tcp://wtsc61oe.itso.ibm.com:2006
  CICS region: SCSCPJA5

Casey_Import_Export
Glass_and_Luget_Plc
Headworth_Electrical
IBM
CommissionCostBuy 010
CommissionCostSell 015
NumberOfShares 0270
TotalShareValue 000044010.00
UnitSharePrice 00163.00
UnitValue1Days 00163.00
UnitValue2Days 00162.00
UnitValue3Days 00160.00
UnitValue4Days 00161.00
UnitValue5Days 00159.00
```

```
UnitValue6Days 00156.00
UnitValue7Days 00157.00
Now we buy 5 shares ...
... and sell 2 of them
```

If you have got this far, you have successfully written and deployed an enterprise bean in CICS. The next section will show you how to design and build an HTML presentation interface for the Trader enterprise bean using a JSP/servlet front end.

7.4.2 Servlet development with VisualAge for Java

This section shows you how we developed a Web based HTML presentation interface to our Trader enterprise bean using a servlet and JSP front-end.

The original COBOL Trader application has a 3270-based user interface supporting the following panels:

- Logon to Trader
- Select a company
- Select option to:
  - Show quotes
  - Buy shares
  - Sell shares

We decided to modify the presentation interface to allow more flexibility, we used the following JSP structure to provide the following functions, shown in Figure 7-32.

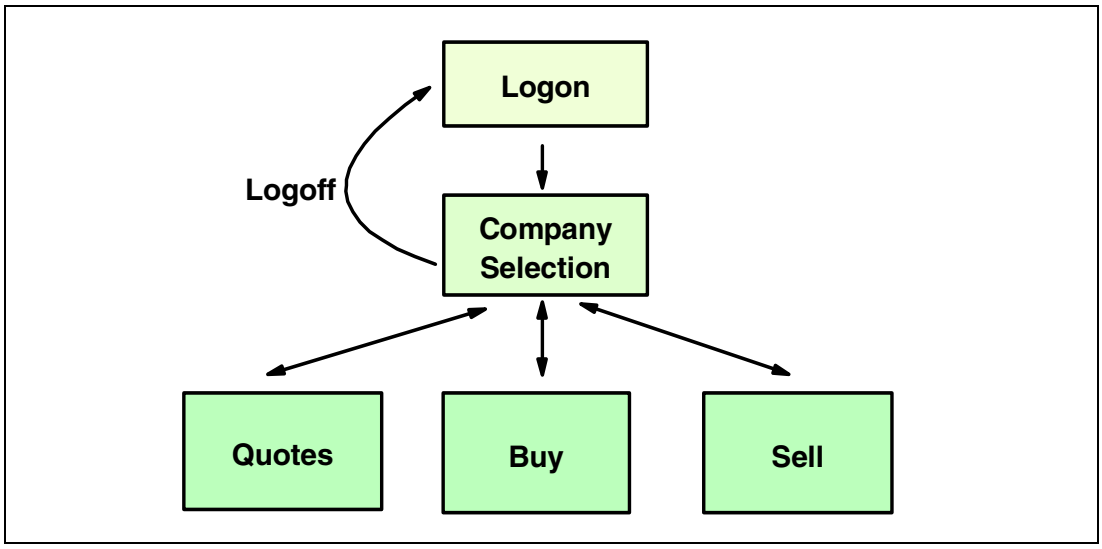


Figure 7-32 Relation between Trader windows

To create the client application, it is necessary to perform the following steps:

1. Develop the Trader servlet.
2. Develop the JSPs.
3. Configure WebSphere Application Server.
4. Test the Trader servlet.

Developing the Trader servlet

We developed all the Trader servlet related classes in the VAJ group *ITSO EJB 390 Redbook* and package *itso.ejb390.trader.servlet*. The name of the class was *TraderServlet*.

We decided to use the *Create Servlet SmartGuide* of VAJ as follows:

1. Click package itso.ejb390.trader.servlet with the right mouse button and select **Add -> Servlet** to open the *Create Servlet SmartGuide*. Project and package should already have the correct names.
2. As class name, specify *TraderServlet*.
3. Click **Finish** to create the servlet.

A servlet class created by VAJ has a method `performTask()` which handles all incoming requests. We distinguish between the following two types of requests.

Show request	This request is used to show a JSP.
Perform request	This request is used to perform an action. The request invokes a show request to display the resultant JSP.

The following requests are implemented by TraderServlet:

`handlePerformLogon()`	Logs on user to Trader application and returns JSP with list of companies.
`handlePerformBuy()`	Buys a number of shares and returns JSP with list of companies.
`handlePerformSell()`	Sells a number of shares and returns JSP with list of companies.
`handleShowCompanies()`	Returns JSP with list of companies.
`handleShowQuotes()`	Returns JSP containing quote information.
`handleShowBuy()`	Returns a JSP to buy shares.
`handleShowSell()`	Returns a JSP to sell shares.
`handleShowLogoff()`	Logs of user from Trader application and returns JSP with logoff message.
`handleShowError()`	Returns a JSP containing error information.

These methods are either called by method `performTask()` or by another handle method.

Let us now take a closer look at the various implementations of these methods.

performTask()

When a user logs on to Trader, a user session is established. As we will later see, `handlePerformLogon()` looks up the TraderBean's home interface and creates a Trader instance. Because TraderBean is a stateful session bean, we have to keep the reference to Trader on the client side.

To pass information from one servlet request to another, we use the servlet session support provided by Application Server. This service allows you to store information associated to a unique key in a *session* object and to retrieve this value at a later time. We called our session object *HttpSession*.

To start with, `PerformTask()` first obtains the session ID from the request object. If this request is the initial logon request, it creates a new Trader session bean instance using our create Trader method, and then stores the reference to this object in the session ID. If this request is not a logon request, then the Trader object is retrieved from the session ID.

Subsequently, we then check what type of request was specified in the JSP (logon, quote, buy, sell or logoff) and the appropriate method is called.

Figure 7-33 shows the most important code snippets of method `performTask()`. We have highlighted the parts which obtain the reference to the Trader object from HttpSession.

```
public void performTask(HttpServletRequest request, HttpServletResponse response) {

Trader trader = null;
// obtain HttpSession
HttpSession httpSesion = request.getSession();

// if logon request
if ( request.getParameter(fieldDoPerformLogon) != null ) {
    // create new trader session bean and keep it in session
    trader = createTrader(request);
    httpSesion.putValue(traderID, trader);
}
else {// for all other request trader must already exist
    // retrieve trader from session
    trader = (Trader)httpSesion.getValue(traderID);
}

// check which request we got and dispatch to appropriate method
String nextJsp = null;
if ( request.getParameter(fieldDoPerformLogon) != null )
nextJsp = handlePerformLogon(request, trader);
else if ( request.getParameter(fieldDoShowQuotes) != null )
nextJsp = handleShowQuotes(request, trader);
else if ( request.getParameter(fieldDoShowCompanies) != null )
nextJsp = handleShowCompanies(request, trader);
else if ( request.getParameter(fieldDoShowBuy) != null )
nextJsp = handleShowBuy(request);
else if ( request.getParameter(fieldDoPerformBuy) != null )
nextJsp = handlePerformBuy(request, trader);
else if ( request.getParameter(fieldDoShowSell) != null )
nextJsp = handleShowSell(request);
else if ( request.getParameter(fieldDoPerformSell) != null )
nextJsp = handlePerformSell(request, trader);
else if ( request.getParameter(fieldDoShowLogoff) != null )
nextJsp = handleShowLogoff(request, trader);
else
nextJsp = handleShowError(request, "Got unknown request to process, check the JSPs.");
// now process JSP
ServletContext sc = getServletContext();
RequestDispatcher rd = sc.getRequestDispatcher("/" + nextJsp);
rd.forward(request, response);
```

Figure 7-33 TraderServlet.performTask()

handlePerformLogon()

This method is used to logon to the Trader application. It performs the following steps:

- Retrieves fields from the request object, which are:

userID	The user ID
password	The password of the user
jndiPrefix	The JNDI prefix of TraderBean
nameService	Determines which name service to use
providerURL	The URL of COS Naming Server
communicationType	Determines if a JCICS `link()` or the CICS Connector is to be used
connectURL	Specifies the URL of the CTG when using the CICS Connector
cicsServer	Specifies the CICS server when using the CICS Connector

- Invokes the Trader bean's `logon()` method
- Creates UserInfo and stores the user ID

  UserInfo is a bean which holds the user ID and the last selected company. These fields are displayed in some HTML windows. UserInfo is stored in the HttpSession object and retrieved by other methods on demand. This is a shortcut to avoid retrieving the user ID from the enterprise bean for each request.
- Invokes `handleShowCompanies()` to return a list of companies

Figure 7-34 shows the complete implementation of the `handlePerformLogon()` method.

```
public String handlePerformLogon(HttpServletRequest request, Trader trader ) throws
Exception {

   // retrieve fields
   String userID = request.getParameter(fieldUserID);
   String password = request.getParameter(fieldPassword);
   String connectURL = request.getParameter(fieldConnectURL);
   String cicsServer = request.getParameter(fieldCicsServer);

   // check if userID and password are provided
   if( userID.equals("") || password.equals("") )
      return handleShowError(request, "You have to specify userID AND password" );

   // now logon to the application
   trader.logon( userID, password, connectURL, cicsServer );

   // store user info, we need it later
   UserInfoBean userInfo = new UserInfoBean();
   userInfo.setUserID(userID);
   request.getSession().putValue(userInfoID, userInfo );

   // show companies now
   return handleShowCompanies(request, trader);

}
```

Figure 7-34 TraderServlet.handlePerformLogon()

handleShowCompanies()

This method returns a list of companies. Companies are returned with bean CompaniesBean. This is the same class that TraderBean also uses to return the companies. The method performs the following steps:

- Retrieve CompaniesBean from HttpSession.
  - The method tries to find the bean in HttpSession. This is also a shortcut to avoid retrieving the companies more than once from the enterprise bean for one client session. Because CompaniesBean implements java.io.Serializable it is allowed to store it in HttpSession.
  - If CompaniesBean is not found (which is always the case for the first invocation of a new client session), it is obtained from the enterprise bean.

    CompaniesBean is stored in HttpSession.
- CompaniesBean is made available for the JSP.

Figure 7-35 shows the full implementation of this method.

```
public String handleShowCompanies(HttpServletRequest request, Trader trader) throws
Exception {

   // check for local copy
   HttpSession httpSesion = request.getSession();
   CompaniesBean companies = (CompaniesBean)httpSesion.getValue(companiesID);

   if( companies == null ) {// if not stored locally
      companies = trader.getCompanies();

      // to improve response time, store it locally
      request.getSession().putValue(companiesID, companies );
   }

   request.setAttribute(beanCompanies, companies);

   return jspCompanySelection;
}
```

Figure 7-35 TraderServlet.handleShowCompanies()

handleShowBuy()

This method is used to display the *buy* dialog. It performs the following steps:

- Retrieve the input parameter, which is:

  `company` The company in which to buy shares.
- Update company in UserInfo.

  Besides the current user ID, UserInfo also holds the last selected company. Therefore it is necessary to retrieve UserInfo from HttpSession, update it with the selected company, and store it again in HttpSession.

  Because this step is also used by other methods, we have moved it to a private method which we have named `updateCompany()`.
- Make UserInfo available for the JSP.

  This piece of code is also used by other methods. We have moved it to the private method `returnUserInfo()`.
- Return the name of the next JSP.

Figure 7-36 shows the implementation of `updateCompany()`, Figure 7-37 of `returnUserInfo()`, and Figure 7-38 of `handleShowBuy()`.

```
private void updateCompany(HttpServletRequest request) {

   String company = request.getParameter(fieldCompany);
   HttpSession httpSesion = request.getSession();
   UserInfoBean userInfo = (UserInfoBean)httpSesion.getValue(userInfoID);
   userInfo.setCompany(company);
   request.getSession().putValue(userInfoID, userInfo );

}
```

Figure 7-36 TraderServlet.updateCompany()

```
private void returnUserInfo(HttpServletRequest request) throws Exception {

   // return user info
   HttpSession httpSesion = request.getSession();
   UserInfoBean userInfo = (UserInfoBean)httpSesion.getValue(userInfoID);
   request.setAttribute(beanUserInfo, userInfo);

}
```

Figure 7-37 TraderServlet.returnUserInfo()

```
ppublic String handleShowBuy(HttpServletRequest request) throws Exception {

   // query company and update it
   updateCompany(request);

   // return user info
   returnUserInfo(request);

   // show buy panel
   return jspBuy;

}
```

Figure 7-38 TraderServlet.handleShowBuy()

handleShowSell()

This method is used to display the *sell* dialog. The only difference between this and `handleShowBuy()` is, that it returns `jspSell`.

handleShowQuotes()

This method is used to display quotes. It performs the following steps:

- Update company in UserInfo.
- Retrieve the input parameter, which is:

  `fieldcompany` The company to display quotes for.
- Invoke `Trader.getQuotes()` to retrieve the latest quotes from TraderBean.
- Return QuotesBean available for the JSP.

- Make UserInfo available for JSP.
- Return name of JSP to process next.

The implementation of `handleShowQuotes()` is illustrated in Figure 7-390.

```
public String handleShowQuotes(HttpServletRequest request, Trader trader) throws
Exception {

   // query company and update it
   updateCompany(request);

   // get the quotes
   String company = request.getParameter(fieldCompany);
   QuotesBean quotes = trader.getQuotes(company);

   // return quotes
   request.setAttribute(beanQuotes, quotes);

   // return user info
   returnUserInfo(request);

   // show the quotes
   return jspQuotes;

}
```

Figure 7-39 TraderServlet.handleShowQuotes()

handlePerformBuy()

This method is used to buy shares. It performs the following steps:

- Retrieve parameter, which is:

  `numberOfShares` The number of shares to buy.
- Get current company from UserInfo stored in HttpSession.

  Because this functionality is also used by `handlePerfromSell()`, we have moved this piece of code to the private method `getCurrentCompany()`.
- Convert number of shares to an integer.
- Invoke `Trader.buy()` to buy shares.
- Make UserInfo available for JSP.
- Invoke `handleShowCompanies()` to show again all companies.

The implementation of `getCurrentCompany()` is shown in Figure 7-41, and the implementation of `handlePerformBuy()` in Figure 7-41.

```
private String getCurrentCompany(HttpServletRequest request) throws Exception {

   HttpSession httpSesion = request.getSession();
   UserInfoBean userInfo = (UserInfoBean)httpSesion.getValue(userInfoID);
   return userInfo.getCompany();

}
```

Figure 7-40 TraderServlet.getCurrentCompany()

```
public String handlePerformBuy(HttpServletRequest request, Trader trader) throws
Exception {

   // get the number of shares
   String numberOfShares = request.getParameter(fieldNumberOfShares);
   String company = getCurrentCompany(request);

   // convert to integer
   int shares = 0;
   try {
      shares = Integer.parseInt(numberOfShares);
   } catch( Exception e ) {
   }

   // buy shares
   trader.buy( company, shares );
   // return user info
   returnUserInfo(request);
   // go back to companies selction
   return handleShowCompanies(request, trader);
}
```

Figure 7-41 TraderServlet.handlePerformBuy()

handlePerformSell()

This method is used to sell shares. Its implementation is exactly the same as `handlePerformBuy()`, except that `Trader.sell()` is called instead of `Trader.buy()`.

handleShowLogoff()

This method is used to logoff from Trader. It performs the following steps:

- Removes the session bean instance in the EJB server.
- Removes Trader from HttpSession.
- Removes CompaniesBean from HttpSession.
- Removes UserInfo from HttpSession.
- Returns the name of JSP to process next.

Figure 7-42 shows the implementation of `handleShowLogoff()`.

```
public String handleShowLogoff(HttpServletRequest request, Trader trader) throws
Exception {

   // logoff and then remove trader object in EJB server
   trader.logoff();
   trader.remove();

   // remove session variables
   request.getSession().removeValue(traderID);
   request.getSession().removeValue(companiesID);
   request.getSession().removeValue(userInfoID);

   // return to logon panel
   return jspLogon;
}
```

Figure 7-42 TraderServlet.handleShowLogoff()

TraderServlet.handleShowError()

This method is used to show an error message. It performs the following steps:

- Instantiates an ErrorMessageBean holding the error message.
- Makes ErrorMessageBean available for JSP.
- Returns the name of JSP to process next.

Figure 7-43 shows the implementation of `handleShowError()`.

```
public String handleShowError(String errorText ) throws Exception {

request.setAttribute(beanErrorMessage, new ErrorMessageBean(errorText));

return jspTraderError;

}
```

Figure 7-43 TraderServlet.handleShowError()

Now that we have implemented TraderServlet, we need to write the corresponding JavaServer Pages.

Developing the JavaServer Pages

We have created the following JavaServer Pages with WebSphere Studio:

- Logon.jsp
- CompanySelection.jsp
- Quotes.jsp
- Buy.jsp
- Sell.jsp
- Logoff.jsp
- TraderError.jsp

These JavaServer Pages obtain their information from UserInfoBean, CompaniesBean, QuotesBean, and ErrorMessageBean.

7.4.3 Configuring WebSphere Application Server for Windows NT

This section explains how we configured WebSphere Application Server Advanced Edition V3.5 for Windows NT to run our TraderServlet in order to test the Trader enterprise bean.

The following steps need to be performed when configuring WebSphere Application Server for Windows NT:

1. Exporting the JAR files
2. Copying files to the WebSphere environment
3. Creating a Web application
4. Defining the TraderServlet

Export the JAR files

For the WebSphere environment, we need the client classes for the TraderBean and the servlet classes. Then we exported the bean's client classes to the file `traderCLI.jar` and the servlet related classes to `traderServlet.jar`.

First, we create `traderCLI.jar`, as shown in the following steps:

1. In VAJ select the **EJB** tab.
2. Click group ITSOEJB390 with the right mouse button and select **Export -> Client JAR** to open the *Export to an EJB Client JAR File SmartGuide*.
3. You can see that initially one bean and six classes are selected. Click **Select referenced types and resources**. Now eight classes are selected.
4. Click **Details** besides .class to verify which classes are selected. These should be:
   - _Trader_BaseStub.class
   - _Trader_Stub.class
   - _TraderHome_BaseStub.class
   - _TraderHome_Stub.class
   - CompaniesBean.class
   - QuotesBean.class
   - Trader.class
   - TraderHome.class

   Click **OK** to close the window.
5. As the JAR file name, type `traderCLI.jar`.
6. Click **Finish** to export the class files to `traderCLI.jar`.

Now we create `traderServlet.jar`:

1. In VAJ click tab **Projects**.
2. Click with the right mouse button package *itso.ejb390.trader.servlet* and select **Export** to open the *Export SmartGuide*.
3. Select Jar file, then click **Next.**
4. Make sure that *.class* is selected.
5. Click **Details** besides *.class* to see which classes are selected. These should be:
   - ErrorMessageBean
   - TraderServlet
   - UserInfoBean

   Click **OK** to close the window again.
6. As the JAR file name specify `traderServlet.jar`.
7. Click **Finish** to export the class files to `traderServlet.jar`

Copying files to the WebSphere environment

As you will see later, we shall call the Web application *trader*, and it will run under the application server default server. Therefore, we created the following directories on the workstation running WebSphere.

- `C:\WebSphere\AppServer\host\default_host\trader\web`

  This directory contains the JSP files.
- `C:\WebSphere\AppServer\host\default_host\trader\servlets`

  This directory contains the servlets.

Having created these directories, we then did as follows:

1. Copied `traderCLI.jar` and `traderServlet.jar` to `C:\WebSphere\AppServer\host\default_host\trader\servlets.`
2. Copied the JSP files we have created as described in , “Developing the JavaServer Pages” on page 207 to `C:\WebSphere\AppServer\host\default_host\trader\web.`

Creating a Web application

Now we will show how we created a Web application for TraderServlet and how we configured it.

1. On your workstation running WebSphere Application Server start WebSphere Advanced Administrative Console. From your Windows desktop select **Start -> Programs -> IBM WebSphere -> Application Server V3.5 -> Administrator’s Console**.

Note: To improve performance we suggest that you always stop the application server before you do any changes. One way to do this is to click the application server (for example, default server) with the right mouse button and select **Stop**. After you have made your changes, start the application server. Click the application server with the right mouse button and select **Start**.

2. Select **Console -> Task -> Create a Web Application**.
3. As Web Application Name, type *trader*.
4. Deselect **Enable File Servlet**.
5. Select E**nable JSP 1.0.**
6. Click **Next**.
7. Open the Nodes subtree until you see Default Servlet Engine of your Default Server.
8. Select **Default Servlet Engine**.
9. Click **Next**.
10. Type your description.
11. For Web Application Web Path specify `/trader` instead of `/webapp/trader`.
12. Click **Next**.
13. Leave the Document Root as suggested by WebSphere: `C:\WebSphere\AppServer\host\default_host\trader\Web`
14. In addition to classpath `C:\WebSphere\AppServer\host\default_host\trader\servlets` add the following JAR files:
    - `C:\WebSphere\AppServer\host\default_host\trader\servlets\traderServlet.jar`
    - `C:\WebSphere\AppServer\host\default_host\trader\servlets\traderCLI.jar`

    Now your windows should look as illustrated in Figure 7-44.
15. Click **Finish** to create the Web application.

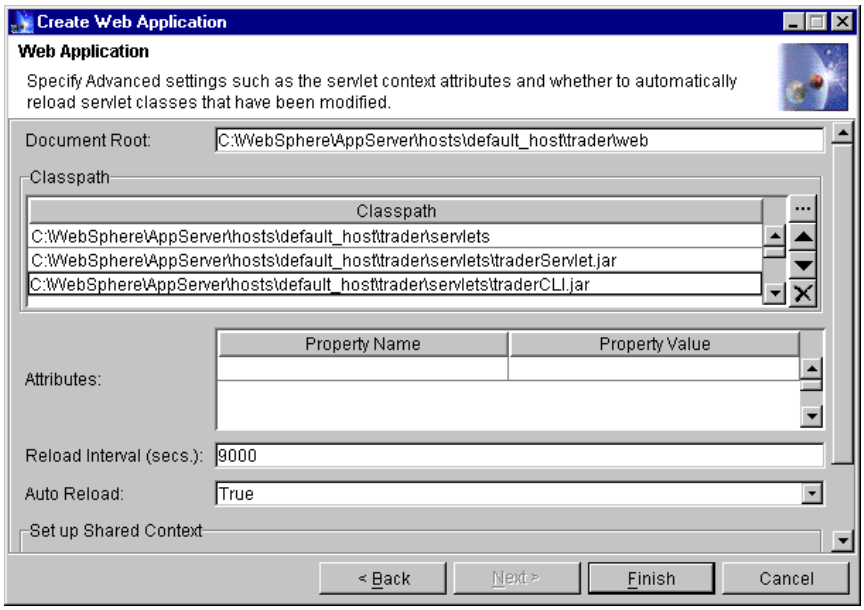


Figure 7-44 Set the classpath for TraderServlet

Defining the Trader servlet

After creating the Trader Web application, we defined the Trader servlet as follows:

1. Right-click on the Web application *trader* and select **Create -> Servlet**.
2. As servlet name, specify TraderServlet.
3. Type the description you like.
4. For Servlet Class Name enter itso.ejb390.trader.servlet.TraderServlet.
5. Click **Add** to add default_host/trader/ to the Servlet Web Path List.
6. Click **OK** to close the *Add Web Path to Servlet* window. Your *Create Servlet* window should now look as shown in Figure 7-45.
7. Click **OK** to create the servlet.

Figure 7-45 Create TraderServlet in WebSphere

7.4.4 Configuring WebSphere Application Server for OS/390

This section explains how we configured WebSphere Application Server V3.5 for OS/390 to run our TraderServlet in order to test the Trader enterprise bean deployed in CICS. The deployed JAR file did not need any modification from the one used in WebSphere Application Server for Windows NT, therefore the creation of the JAR file does not differ from that documented in “Export the JAR files” on page 208. The JSP and HTML files were also the same ones as those used for Windows NT.

The following steps need to be performed when configuring WebSphere Application Server for OS/390:

1. Copy the files to the HFS on OS/390.

   Our Web application server used the following HFS directories on OS/390:

   /usr/lpp/was35 Application server root

   /web/cics4 Location of configuration files

   /web/cics4/servlets Location of servlets and JAR files

   We transferred the files `traderCli.jar`, `traderServlet.jar`, `j2ee.jar` to this directory

   /web/cics4/webapp/trader Location of HTML documents

   We transferred the files `Buy.jsp`, `CompanySelection.jsp`, `Logon.jsp`, `Quotes.jsp`, `Sell.jsp`, `TraderError.jsp` to this directory.

2. Edit the configuration files.

   We edited the `was.conf` and `httpd.conf` files to configure the application server with our new *Trader* Web application. A summary of the statements we added is shown in Figure 7-46 and Figure 7-47.

```
Service     /trader/*          /usr/lpp/was35/AppServer/bin/was350plugin.so:service_exit
Service     /webapp/*          /usr/lpp/was35/AppServer/bin/was350plugin.so:service_exit
Service     /*.jsp             /usr/lpp/was35/AppServer/bin/was350plugin.so:service_exit
```

Figure 7-46 WebSphere Application Server for OS/390 — httpd.conf

```
#Trader EJB servlet
deployedwebapp.TraderEJB.host=default_host
deployedwebapp.TraderEJB.rooturi=/trader
deployedwebapp.TraderEJB.classpath=/web/cics4/servlets
deployedwebapp.TraderEJB.documentroot=/web/cics4/webapp/trader
deployedwebapp.TraderEJB.autoreloadinterval=3000
webapp.TraderEJB.jspmapping=*.jsp
webapp.TraderEJB.filemapping=/
webapp.TraderEJB.jsplevel=1.0
webapp.TraderEJB.servlet.EJBServlet.code=itso.ejb390.trader.servlet.TraderServlet
webapp.TraderEJB.servlet.EJBServlet.servletmapping=/TraderServlet
```

Figure 7-47 WebSphere Application Server for OS/390 — was.conf

For further details on configuring WebSphere Application Server for OS/390, refer to *HTTP Server Planning, Installing, and Using,* SC31-8690, and *WebSphere Application Server Standard Edition Planning, Installing, and Using*, GC34-4835, available from the following Web site:

`http://www-4.ibm.com/software/webservers/appserv/library_390.html`

Testing the Trader servlet

After starting the application server, open your Web browser and enter the following URL:

```
http://<hostname>/trader/Logon.jsp
```

In this URL, `<hostname>` specifies the TCP/IP address of your Web server. You should now receive a logon panel as shown in Figure 7-48. The panels shown are the same for either WebSphere Application Server on Windows NT or on OS/390.

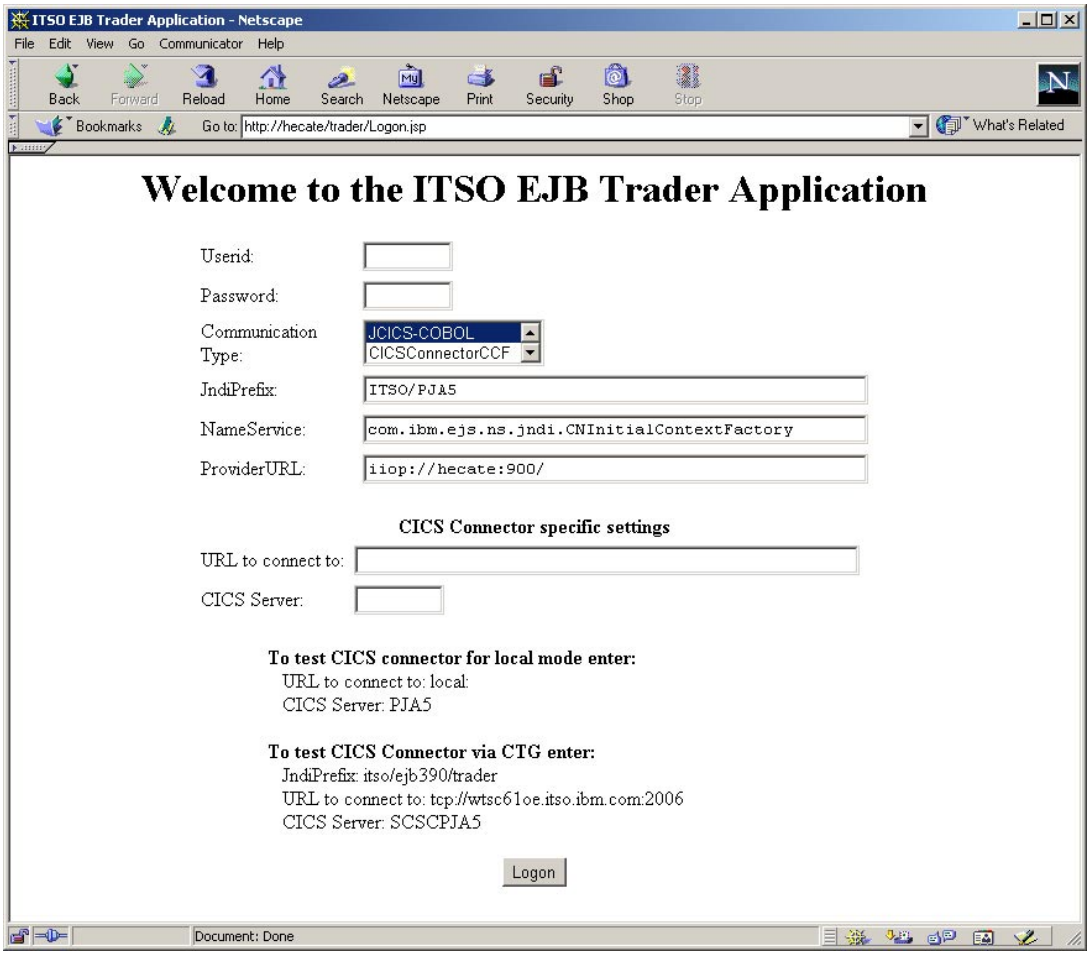


Figure 7-48 Logon to Trader application using TraderServlet

To logon, follow the steps below:

1. Enter any user ID and any password.
2. Leave JCICS-COBOL selected, to use a JCICS `link()` to invoke the COBOL Trader application.
3. If your have defined a JNDI prefix with your CORBASERVER, use this prefix for JndiPrefix. If you have no JNDI prefix defined, this field must be empty. Our JndiPrefix was ITSO/PJA5.
4. If you use the connection factory provided by WebSphere Application Server Advanced Edition for Windows NT, you can use the default value of `com.ibm.ejs.ns.jndi.CNInitialContextFactory` as the NameService. Otherwise if you used the connection factory provided by `j2ee.jar`, such as when using WebSphere Application Server V3.5 for OS/390 change it to `com.sun.jndi.cosnaming.CNCtxFactory`

5. Specify the provider URL of your COS Naming Server. In our case the value is `iiop://hecate:900/` because our COS Naming Server and Web server both ran on the workstation called *hecate*.
6. The fields **URL to connect to** and **CICS Server** are ignored for JCICS-COBOL and so can be left to default.
7. Click **Logon** to connect to trader.

The next window you should see is the company selection as illustrated in Figure 7-50.

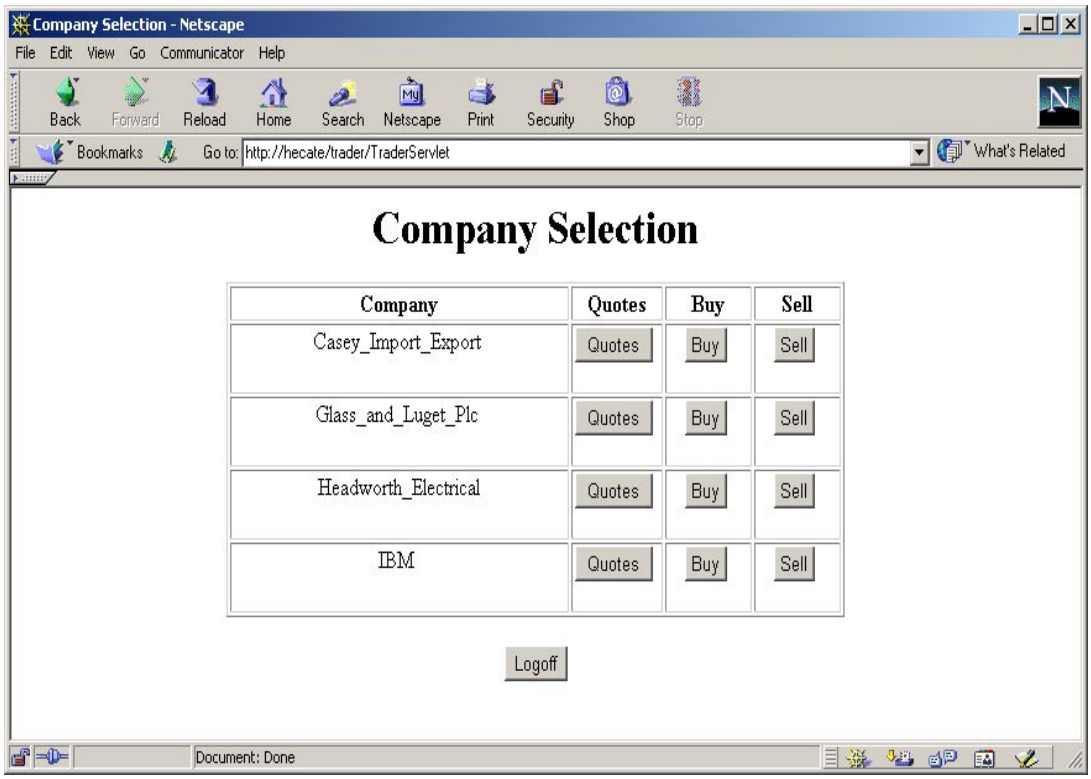


Figure 7-49 Company selection using TraderServlet

In this window you can see the same four companies as we have seen using the stand-alone TraderTest program. Choose from the following menus to obtain a quote, or trade shares.

Quotes	To view the quotes of a company
Buy	To buy shares of a company
Sell	To sell shares of a company
Logoff	To logoff from Trader and return to the logon window

If you chose Quotes, you will be presented with the HTML form shown in Example 7-51.

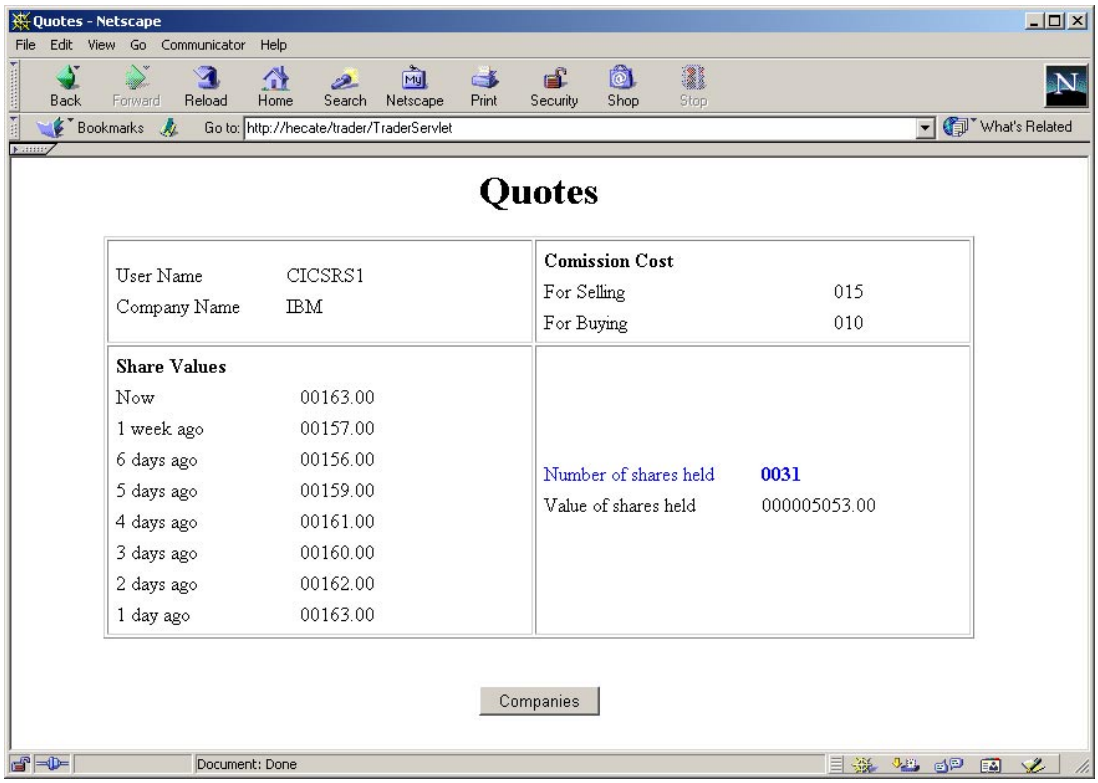


Figure 7-50 Quote results using TraderServlet

If you now click on *Companies*, this will return you to the *Company Selection* window, and you can select *Buy* or *Sell*. If you select *Buy* you will be presented with a window as shown in Example 7-51, enabling you to buy shares in this particular company.

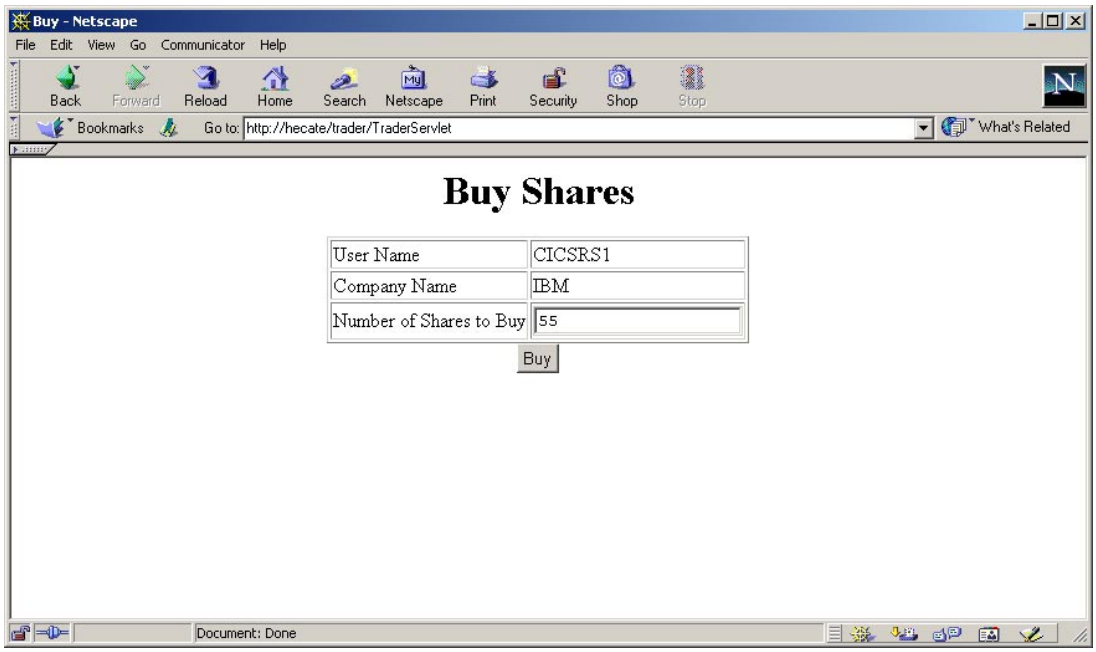


Figure 7-51 Buy Shares form using TraderServlet

7.5 Summary

This chapter has shown you how you can use an enterprise bean in CICS TS V2.1 to wrap the business logic in an existing application. The CICS program in question does not have to be a COBOL application, but could be any kind of CICS program, written in Assembler, PL/I, C, C++, or even Java.

As you will see in the following chapters, we go on to extend the Trader enterprise bean using a series of new back-end classes in place of the existing TraderBackendJcics class. These classes have the same functionality as the original COBOL Trader application, but are written in Java using either JCICS, JDBC, or SQLJ, and directly access either the existing VSAM files or DB2.

The reasons why we structured our development like this are two-fold. First, we wanted show you how to access VSAM files and relational data directly from an enterprise bean in CICS. Second, we wanted to illustrate a possible migration path in a real world environment, whereby an enterprise bean is used first to wrap a traditional COBOL application and then the enterprise bean is further developed to invoke new Java based business logic in CICS.

This does mean that our enterprise bean is structured in a somewhat different way than might otherwise be the case. All the business logic is implemented in a set of back-end classes that implement a single interface TraderBackend. The invocation of the correct back-end class is controlled using an input parameter in order to ensure that the correct back-end class is invoked. A real-life application could be structured somewhat differently with all the business logic implemented within the actual session beans (or within multiple session beans).

8

Wrapping the Trader application: CICS Connector

This aim of this chapter is to describe how to write an enterprise bean that invokes an existing CICS program (written in COBOL or any other language), using the CICS connector. To this end, we show you how we modified the TraderBean developed in the previous chapter, to use the CICS connector as opposed to the JCICS `link()` method. We then show you how we deployed this modified TraderBean into WebSphere Application Server on Windows NT, and then how we moved the same code into our CICS TS V2.1 EJB Server. This scenario is illustrated in Figure 8-1.

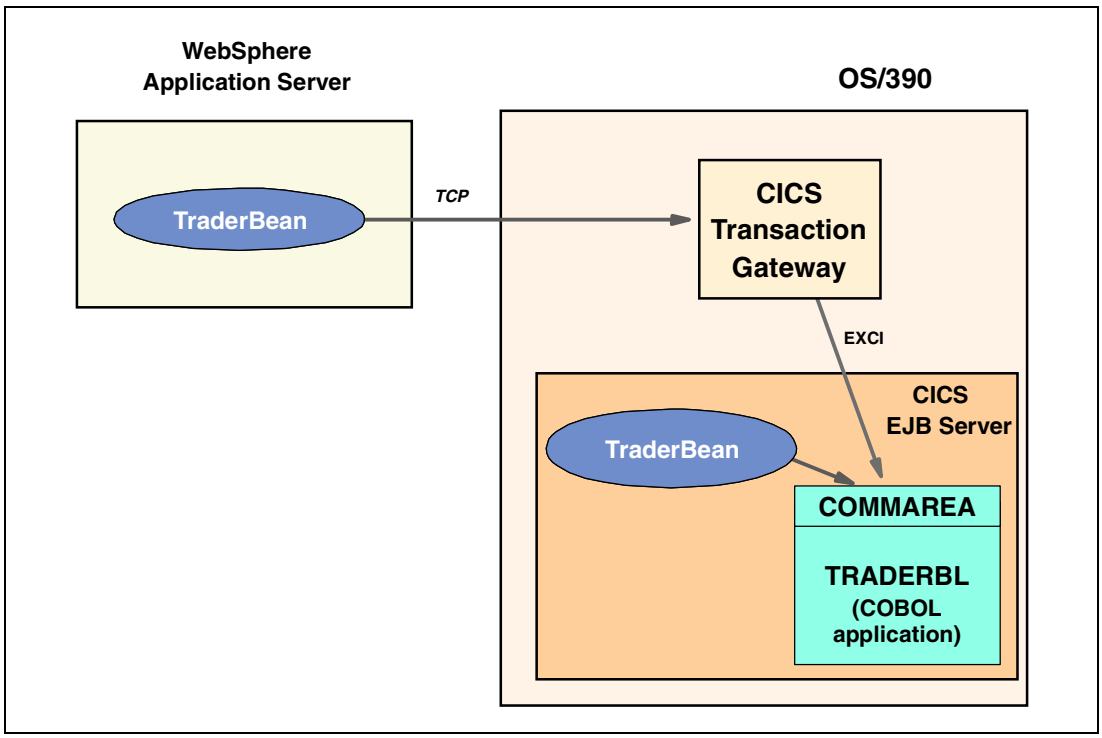


Figure 8-1 Calling a COBOL program with the CICS Connector

Common Connector Framework

The CICS connector is part of IBM's Common Connector Framework (CCF) and provides a standard infrastructure for developing client applications using JavaBean connectors. The CCF is based upon the following objects:

- **ConnectionSpec** is the object that defines all connection-relevant attributes of a CCF connector, such as hostname, port number.
- **InteractionSpec** retains all attributes of the interaction itself, such as the name of the CICS application.
- **Communication** is the object that will drive a particular interaction, specifying only an instance of the *InteractionSpec*, and an input and output record to carry the exchanged data.

The CICS Transaction Gateway provides the objects *CICSConnectionSpec*, *ECIInteractionSpec*, and *EPIInteractionSpec* classes that implement the CCF connector for CICS. These classes are provided in the CTG class library ctgclient.jar, and are also provided by VisualAge for Java and by CICS Transaction Server V2.1. For further information on the CCF, refer to the redbook *CCF Connectors and Databases Connections using WebSphere Advanced Edition*, SG24-5514.

CICS TS V2.1 connector

In CICS TS V2.1, a Java program or enterprise bean running within CICS can use the new *CICS Connector for CICS TS* to link to a suitable CICS server program. This connector runs within the CICS region and provides the same CCF connector interface as previously provided by the CTG. However, when using the CICS Connector for CICS TS, the CTG is not required, as the connector runs within the CICS TS V2.1 JVM. This function provided is equivalent to the JCICS `link()` method that we used in our previous chapter, since it allows a Java program to invoke the business logic in an existing CICS application.

The advantage of using the CICS connector for CICS TS over the JCICS `link()` method is that it allows you to take an enterprise bean that was previously deployed to an external EJB Server (and used the CTG to invoke CICS applications) and deploy it directly in the CICS TS V2.1 EJB Server without modification. To this end, we illustrate how we did just this by deploying a CCF version of our TraderBean into both WebSphere Application Server and directly into our CICS TS V2.1 EJB Server, with no modification of the code.

Note: The CICS connector for CICS TS is based on the technology in the CTG class library ctgclient.jar. Because of this, it is also possible to develop and deploy Java applications into CICS TS V2.1 that use the CTG *JavaGateway* and *ECIRequest* objects, instead of using the *ECIInteractionSpec* and *EPIInteractionSpec* objects. However, you should note that the JCICS `link()` method is a lower level interface than either of these and is therefore likely to provide the best performance, but will not be portable if your application is moved outside of a CICS region.

8.1 Quick start — Invoking TraderBean

If you want to run our sample Trader enterprise bean without following all the details specified in this chapter, use the steps below. All the source code and examples used in this book are available for download from the redbooks Web site `ftp://www.redbooks.ibm.com/redbooks/` and for full details of the available files and how to obtain them, you can refer to Appendix C, “Using the additional material” on page 315.

1. Install the COBOL Trader application in your CICS system. For more details, refer to Appendix B, “The COBOL Trader application” on page 309.
2. *Either* deploy the TraderBean to either WebSphere Application Server (see 8.3, “Deploying the enterprise bean to WebSphere” on page 227)...
3. *Or* deploy the TraderBean to your CICS TS V2.1 region (see 8.4, “Deploying the enterprise bean to CICS” on page 230). You will need to create a CICS TCPIPSERVICE, CORBASERVER, REQUESTMODEL and DJAR definition if you have not already done so; refer to 6.3.3, “Deploying to CICS” on page 150 for further details.
4. Test the application; this can be achieved in one of the following two ways:
   a. Use our supplied *TraderServlet* to create a Web application with an HTML front-end to TraderBean. For further details on the expected output refer to Figure 7-50, “Quote results using TraderServlet” on page 214.
   b. Use the supplied runTest.cmd file to invoke our stand-alone Java test application *TraderTest.* To set up TraderTest, simply do the following:
      - On your workstation, create a directory (for example, `C:\itsotrader`) and copy the following sample files to this directory:

        ```
        traderCLI.jar
        traderTest.jar
        runTest.cmd
        ```

      - Ensure that you have a Java 2 runtime environment at version 1.3 or greater installed on your workstation. You can verify your version with the command:

        ```
        java -version
        ```

      - Ensure that you have file `j2ee.jar` accessible on your workstation. If not, you can either obtain it if you install the CICS development deployment tool or by installing Java 2 SDK, Enterprise Edition available from:

        `http://java.sun.com`

      - Invoke TraderTest using the runtest.cmd file. You will need to alter the input parameters as documented in the file. For further details and for an example of expected output, refer to 8.3.1, “Testing the enterprise bean running in WebSphere” on page 229.

8.2 Adapting TraderBean for use of the CICS Connector

Modification of the TraderBean to use the CICS connector requires only minimal changes to the bean itself. This section details these changes in the method `loadClass()`.

Modify TraderBean.loadClass()

Our initial implementation of `loadClass()` only took into account access using the JCICS `link()` method to invoke the Trader COBOL application. However, because we implemented all the CICS access logic in back-end classes we need to create a new class to invoke the Trader COBOL application using the CCF, called *TraderBackendCICSConnectorCCF*. The modifications to `loadClass()` to invoke this new class are shown in Figure 8-2 on page 220.

```
private void loadClass(String type) throws Exception {

Class loadClass=null;

if( type.equalsIgnoreCase("JCICS-COBOL") == true ) {
  loadClass = Class.forName("itso.ejb390.trader.TraderBackendJcics");
}
else if( type.equalsIgnoreCase("CICSConnectorCCF") == true ) {
  loadClass =
    Class.forName("itso.ejb390.trader.TraderBackendCICSConnectorCCF");
}
else {
  throw new TraderException( "You specified unknown type " + type );
}

ivTraderBackend = (TraderBackend)loadClass.newInstance();

}
```

Figure 8-2 TraderBean.loadClass() loading TraderBackendCICSConnectorCCF

8.2.1 Implementing TraderBackendCICSConnectorCCF

This section demonstrates how to implement class TraderBackendCICSConnectorCCF which invokes the COBOL Trader application using the CICS Connector.

Because the class must implement TraderBackend, it is necessary to implement the following methods:

- logon()
- logoff()
- getCompanies()
- getQuotes()
- buy()
- sell()
- ejbBackendCreate()
- ejbBackendRemove()
- ejbBackendActivate()
- ejbBackendPassivate()

There are two ways of using the CICS Connector for CICS TS:

1. Program to the connector's Common Connector Framework (CCF) Client Interface, using VisualAge for Java Enterprise Access Builder or a similar product. This is the recommended method.
2. Program to the connector's CTG API (using the JavaGateway and ECIRequest objects). Normally, you would use this method only if you do not have access to VAJ EAB or a similar product.

For our sample we will show how to use CICS Connector using the CCF interface. For more information about CCF, refer to chapter 13 of the *e-business Enablement Cookbook for OS/390 Volume III: Java Development, or the online help of VisualAge for Java*, SG24-5980.

The following steps are necessary to implement the new back-end class TraderBackendCICSConnectorCCF, and are described in more detail below:

1. Create class TraderCommand.
2. Implement class TraderBackendCICSConnectorCCF.
3. Implement control methods for TraderBackendCICSConnectorCCF.
4. Implement TraderBackendCICSConnectorCCF.logon().
5. Implement TraderBackendCICSConnectorCCF.getCcompanies().
6. Implement TraderBackendCICSConnectorCCF.getQuotes().
7. Implement TraderBackendCICSConnectorCCF.buy().
8. Implement TraderBackendCICSConnectorCCF.sell().

Create class TraderCommand

- Before you start to use CCF with VAJ, you have to install the IBM Common Connector Framework feature. We assume that you have already also installed the IBM EJB Development Environment, Enterprise Access Builder, IBM Java Record Library, and CICS Connector features as described in the previous chapters:

Now follow these steps:

1. In VAJ select **Workspace -> Tools -> Enterprise Access Builder -> Create Command** to open the Create Command SmartGuide.
2. Select the project *ITSO EJB 390 Redbook*.
3. Select the package itso.ejb390.trader.
4. As the class name specify TraderCommand.
5. Ensure that **edit when finished** is selected.
6. Click **Browse** for ConnectionSpec.

   Select **CICSConnectionSpec** and click **OK**.
7. Click **Edit** for the ConnectionSpec to view its properties.
   a. As you can see, URL is set to *local:*. This will work with the CICS connector for CICS TS, but not with a remote CTG, so we will need to modify this later.
   b. Click **Close** to close the properties window.
8. Click **Browse** for InteractionSpec.

   Select ECIInteractionSpec and click **OK**.
9. Click **Edit** for InteractionSpec to view its properties.

a. Click the rectangle which represents the value of property programName and type `TRADERBL`. This is the name of the COBOL program we want to invoke.

b. Click **OK** to close the properties window.

10. Click **Next**.

11. Ensure that **implements IByteBuffer** is selected.

12. As class name for the input record bean, specify `itso.ejb390.trader.TraderRecord`. You can use the **Browse** button to find the class easily.

13. For output record beans select *Use input bean type as output bean type*.

14. Click **Finish** to create TraderRecord.

15. Because we specified to edit the command when we created TraderCommand, the Command Editor opens. Verify the settings you specified and make changes if necessary.

16. Close the Command Editor.

As you can see, the CICSConnectionSpec controls the way that a CICS system is accessed, including the URL of the connector. Since we let the URL default to the use of the *local* protocol, we need to modify the code in order for it to work with the CTG which will require a valid URL of our remote CTG daemon. Theoretically we would need a new Command which defines these other properties. However, we decided to reuse TraderCommand, and therefore it is necessary to change TraderCommand slightly, as explained next.

Select TraderCommand to view its instance variables. There you can see a section which allows you to add your own user defined code. Define the following two instance variables as shown in Figure 8-3.

```
// user code begin {__declarations_1}
  private String connectURL="local:";
  private String cicsServer="";
// user code end {__declarations_1}
```

Figure 8-3 Define instance variables for TraderCommand

Edit the method `getceConnectionSpec()` and add the code as illustrated in Figure 8-4.

```
// user code begin {__getceConnectionSpec()_1}
ceConnectionSpec.setURL(connectURL);
ceConnectionSpec.setCICSServer(cicsServer);
// user code end {__getceConnectionSpec()_1}
```

Figure 8-4 Change TraderCommand.getceConnectionSpec()

Make a copy of the default constructor `TraderCommand()` and change it as shown (Figure 8-5).

```
public TraderCommand(String connectURLArg, String cicsServerArg){

  super();
  try{
  // user code begin {__TraderCommand()_1}
    connectURL = connectURLArg;
    cicsServer = cicsServerArg;
  // user code end {__TraderCommand()_1}
```

Figure 8-5 New constructor for TraderCommand

Optionally, you can now delete the default constructor `TraderCommand()` and method `main()`.

Now that we have created *TraderCommand* we can create class *TraderBackendCICSConnectorCCF* which will use TraderCommand to invoke the COBOL Trader application.

Implement class TraderBackendCICSConnectorCCF

This section shows how to implement TraderBackendCICSConnectorCCF to invoke Trader using the CICS Connector. It makes use of the CCF by using TraderCommand class, which we created in the previous step.

The declaration of TraderBackendCICSConnectorCCF is shown in Figure 8-6. It is similar to TraderBackendJcics, but declares the two additional instance variables `connectURL` and `cicsServer`.

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

public class TraderBackendCICSConnectorCCF implements TraderBackend {

  private final static int NUM_OF_COMPANIES = 4;

  private String connectURL=null;
  private String cicsServer=null;
}
```

Figure 8-6 Declaration of TraderBackendCICSConnectorCCF

Implement control methods for TraderBackendCICSConnectorCCF

Similar to our previous back-end class TraderBackendJcics, the callbacks from the EJB control methods to the methods `ejbBackendCreate()`, `ejbBackendRemove()`, `ejbBAckendActivate()`, and `ejbBackendPassivate()` do not need any actual code and are only retained for compatibility with later versions of the bean. See Figure 8-7.

```
public void ejbBackendCreate() throws javax.ejb.CreateException,
                              java.rmi.RemoteException {}

public void ejbBackendRemove() throws java.rmi.RemoteException {}

public void ejbBackendActivate() throws java.rmi.RemoteException {}

public void ejbBackendPassivate() throws java.rmi.RemoteException {}
```

Figure 8-7 Control methods of TraderBackendCICSConnectorCCF

Implement TraderBackendCICSConnectorCCF.logon()

In contrast to method `TraderBackendJcics.logon()` we now have to keep `connectURL` and `cicsServer` for use later as illustrated in Figure 8-8.

```
public void logon(String userIDArg, String passwordArg, String connectURLArg, String
cicsServerArg) {

  connectURL = connectURLArg;
  cicsServer = cicsServerArg;
}
```

Figure 8-8 TraderBackendCICSConnectorCCF.logon()

Implement TraderBackendCICSConnectorCCF.logoff()

For this back-end implementation, no business logic is necessary to logoff. Therefore, the method is quite simple, as illustrated in Figure 8-9.

```
public void logoff() {
}
```

Figure 8-9 TraderBackendCICSConnectorCCF.logoff()

Implement TraderBackendCICSConnectorCCF.getCompanies()

The method `getCompanies()` does the following:

- Instantiate a CompaniesBean.
- Instantiate a TraderCommand.
- Set the request type to Get_Company in the input COMMAREA.
- Execute TraderCommand to invoke Trader.
- Iterate over company name array of the output COMMAREA and copy the companies to CompaniesBean.
- Return the CompaniesBean instance.

The implementation of this method is very similar to `TraderBackendJcics.getCompanies()`, but simpler, as you can see. Figure 8-10 shows the implementation of the method `getCompanies()`.

```
public CompaniesBean getCompanies() throws Exception {

CompaniesBean companies = new CompaniesBean();

// create trader command
TraderCommand traderCommand = new TraderCommand(connectURL, cicsServer);

// set request type
traderCommand.getCeInput().setRequest__Type("Get_Company");

// execute command
traderCommand.execute();

// read companies and copy them to CompaniesBean
TraderRecord traderCommareaOut = traderCommand.getCeOutput0();

for (int i = 0; i < NUM_OF_COMPANIES; i++) {
  companies.addCompany(traderCommareaOut.getCompany__Name__Tab(i));
}

return companies;
}
```

Figure 8-10 TraderBackendCICSConnectorCCF.getCompanies()

Implement TraderBackendCICSConnectorCCF.getQuotes()

Method `getQuotes()` performs the following steps:

- Instantiate a QuotesBean.
- Instantiate a TraderCommand.
- Set the request type to Share_Value in the input COMMAREA.
- Execute TraderCommand to invoke COBOL Trader program.
- Copy the results from the output COMMAREA.
- Return the QuotesBean instance.

Figure 8-11 shows the implementation of method `getQuotes()`.

```
public QuotesBean getQuotes( String company, String userID )
                                                    throws Exception {

QuotesBean quotes = new QuotesBean();

// create trader command
TraderCommand traderCommand = new TraderCommand(connectURL, cicsServer);
TraderRecord traderCommarea = traderCommand.getCeInput();

// set request type
traderCommand.getCeInput().setRequest__Type("Share_Value");

// set additional parameters
traderCommarea.setCompany__Name(company);
traderCommarea.setUserid(userID);

// execute command
traderCommand.execute();

// return results
TraderRecord traderCommareaOut = traderCommand.getCeOutput0();

quotes.setUnitSharePrice( traderCommareaOut.getUnit__Share__Price() );
quotes.setUnitValue1Days( traderCommareaOut.getUnit__Value__1__Days() );
quotes.setUnitValue2Days( traderCommareaOut.getUnit__Value__2__Days() );
quotes.setUnitValue3Days( traderCommareaOut.getUnit__Value__3__Days() );
quotes.setUnitValue4Days( traderCommareaOut.getUnit__Value__4__Days() );
quotes.setUnitValue5Days( traderCommareaOut.getUnit__Value__5__Days() );
quotes.setUnitValue6Days( traderCommareaOut.getUnit__Value__6__Days() );
quotes.setUnitValue7Days( traderCommareaOut.getUnit__Value__7__Days() );
quotes.setCommissionCostSell(
  traderCommareaOut.getCommission__Cost__Sell());
quotes.setCommissionCostBuy( traderCommareaOut.getCommission__Cost__Buy() );
quotes.setNumberOfShares( traderCommareaOut.getNo__Of__Shares() );
quotes.setTotalShareValue( traderCommareaOut.getTotal__Share__Value() );

return quotes;
}
```

Figure 8-11 TraderBackendCICSConnectorCCF.getQuotes()

Implement TraderBackendCICSConnectorCCF.buy()

This method needs to do the following:

- Instantiate a TraderCommand.
- Set the request type to Buy_Sell in input COMMAREA.
- Set company, userID, and number of shares in input COMMAREA.
- Set a flag indicating whether to buy or to sell shares in input COMMAREA.
- Execute TraderCommand to invoke Trader.

Just as we did in the class TraderBackendJcics, we have moved this functionality into a private method `trade()`. We then specify with an additional flag, whether or not to buy or sell shares. The implementation of method `buy()` is shown in Figure 8-12, and method `trade()` is shown in Figure 8-13.

```
public void buy( String company, String userID, int numberOfShares )
                                                        throws Exception {
trade( company, userID, numberOfShares, true );
}
```

Figure 8-12 TraderBackendCICSConnectorCCF.buy()

```
private void trade( String company, String userID, int numberOfShares,
                    boolean buy ) throws Exception {

TraderCommand traderCommand = new TraderCommand(connectURL, cicsServer);
TraderRecord traderCommarea = traderCommand.getCeInput();

traderCommand.getCeInput().setRequest__Type("Buy_Sell");

// set additional parameters
traderCommarea.setCompany__Name(company);
traderCommarea.setUserid(userID);
traderCommarea.setNo__Of__Shares__Dec( (short)numberOfShares );
if( buy == true )    // if buy
  traderCommarea.setUpdate__Buy__Sell("1");
else    // if sell
  traderCommarea.setUpdate__Buy__Sell("2");

traderCommand.execute();
}
```

Figure 8-13 Implementation of TraderBackendCICSConnectorCCF.trade()

Implement TraderBackendCICSConnectorCCF.sell()

The method `sell()` is very similar to the method `buy()`. It calls the internal method `trade()` as shown in Figure 8-14.

```
public void sell( String company, String userID, int numberOfShares )
                                                        throws Exception {
trade( company, userID, numberOfShares, false );
}
```

Figure 8-14 TraderBackendCICSConnectorCCF.sell()

8.3 Deploying the enterprise bean to WebSphere

In this section we describe how to deploy the TraderBean and its related classes to WebSphere Application Server Advanced Edition for Windows NT; this scenario is shown in Figure 8-15. The TraderBean uses our new TraderBackendCICSConnectorCCF class to link to the Trader COBOL application (TRADERBL). The TraderBackendCCF class uses the CCF to send a request to our OS/390 CTG, which forwards the request into our CICS region using the EXCI.

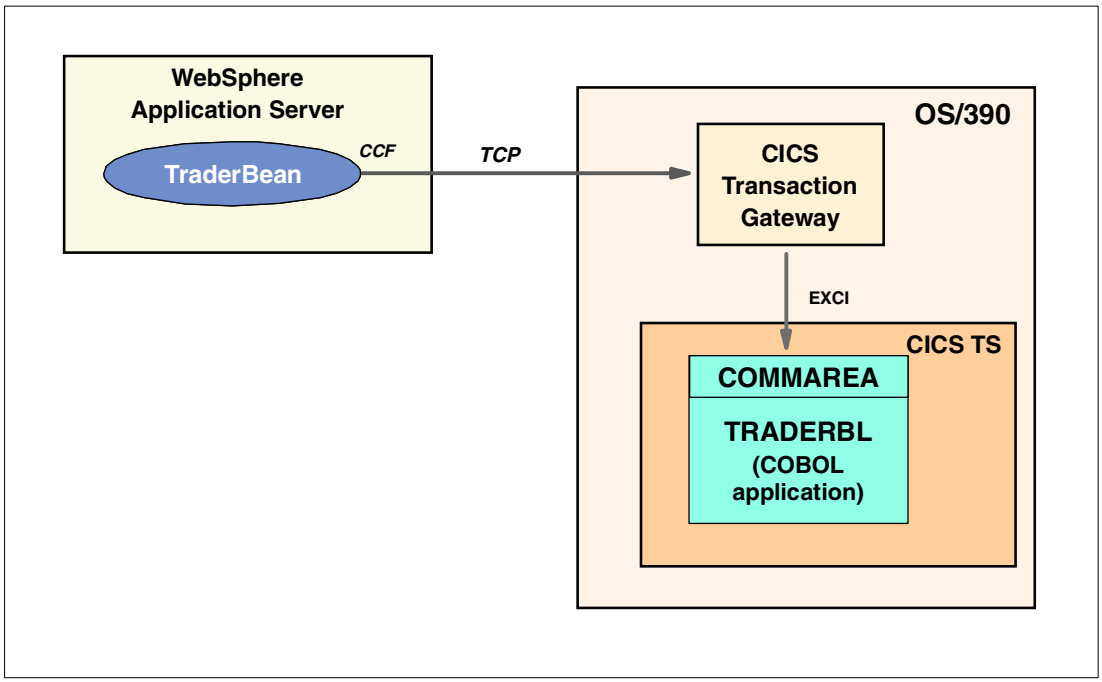


Figure 8-15 Deploying Traderbean to WebSphere

After deploying the TraderBean, we also describe how to:

1. Export the enterprise bean and its related classes.
2. Copy the JAR files to WebSphere.
3. Create an EJB container and an enterprise bean definition.

We do not describe how to configure or implement the OS/390 CTG. For further details on this subject refer to the IBM redbook *CICS Transaction Gateway V3.1, The WebSphere Connector for CICS*, SG24-6133.

Export the enterprise bean and its related classes

Export the enterprise bean from VAJ in the same way as described in“Export the enterprise bean and its related classes” on page 231. The only difference is that you do not need to include class TraderBackendJcics, because we cannot use the JCICS classes outside of the CICS environment. Therefore, you should now have thirteen classes and one bean that has been selected to export. We named our JAR file `traderForWAS.jar`.

Copy the JAR files to WebSphere

On the workstation running WebSphere, we need several JAR files to run TraderBean. The easiest way is to create a directory on this workstation for the supporting classes which we named `c:\ejbjar`. We copied the following files, all originating from our development workstation, to this directory:

traderForWAS.jar	This is the JAR file created in the previous step.
CTGCLIENT.JAR	This file can be found in `C:\IBM Connectors\CICS\classes` or in the `\classes` sub-directory for a CTG installation.
CCF.jar	This file is located in `C:\IBM Connectors\classes`.
eablib.jar	This file is located in `C:\Program Files\IBM\VisualAge for Java\eab\runtime30`.

Create an EJB container and an enterprise bean definition

On the Windows NT workstation define, using the WebSphere Advanced Administrative Console, a new EJB Container and an enterprise bean definition, as follows:

1. Stop the Default Server application server.
2. Right-click application server Default Server and select **Create -> EJBContainer**.
   a. Enter Trader as the EJBContainerName.
   b. Click **OK** to create the container and close the window.
3. Right-click on the new EJBContainer **Trader** and select **Create -> EnterpriseBean**.
   a. Specify Trader as the name.
   b. Click **Browse** to open the file browser window.
      i. Navigate to the directory containing `traderForWAS.jar`, which was in our case `C:\ejbjar`, and select it.
      ii. Double-click on `traderForWAS.jar`
      iii. The dialog shows you one bean, which is `itso.ejb390.trader.Trader/Trader.ser`.
      iv. Select this bean.
      v. Click **Select**.
      vi. A confirm dialog opens which asks you to *Deploy and Enable WLM* or to *Deploy Only*. Click **Deploy Only**.
      vii. A window opens which tells you that the bean is deploying, and then a message reports that *Command Deploy Jar file completed successfully.* Click **OK**.
   c. Back in the Create EnterpriseBean window you can see that WebSphere has inserted a JAR file name and a deployment descriptor for you. You can click **Edit** to view the bean properties, but you do not need to make any changes.
   d. Click **OK** to close the Create EnterpriseBean window.
4. Select your node.
5. To the dependent classpath, add the following JAR files we previously copied to `c:\ejbjar`

   `C:\ejbjar\eablib.jar;C:\ejbjar\recjava.jar;C:\ejbjar\CCF.jar;C:\ejbjar\CTGCLIENT.jar`
6. Click **Apply**.
7. Start the WebSphere application server *Default Server.*

8.3.1 Testing the enterprise bean running in WebSphere

We tested our enterprise bean in two ways, with our TraderTest standalone application and with our TraderServlet application. However, for both environments, because our Trader enterprise bean is now running in another container, we had to use a different JNDI name *itso/ejb390/trader/Trader*. The new name is the package name *itso.ejb390.trader* combined with the bean name and is the default WebSphere chose when we deployed the enterprise.

Testing the enterprise bean with TraderTest

TraderTest is a standalone Java application that can be run from either a command line or within VAJ. Further instructions on how to use this from the command line are given in 8.1, "Quick start — Invoking TraderBean" on page 219. We were using a CTG on our OS/390 system, the host name of which was `wtsc61oe.itso.ibm.com`, and that was configured to listen on port 2006. Thus the URL `tcp://wtsc61oe.itso.ibm.com:2006` was required for the enterprise bean running in WebSphere to connect to the CTG on OS/390.

We edited the following line in the `runtest.cmd` script so as to invoke the TraderTest application with the CICSConnectorCCF option.

```
java -classpath ".;traderTest.jar;traderCLI.jar;C:\Program Files\IBM\CICS TS 2.1
Tools\Common\j2ee.jar" itso.ejb390.trader.test.TraderTest
com.sun.jndi.cosnaming.CNCtxFactory iiop://hecate:900/ itso/ejb390/trader/Trader
CICSConnectorCCF tcp://wtsc61oe.itso.ibm.com:2006 SCSCPJA5
```

The successful output of `runtest.cmd` is shown in Example 8-1.

Example 8-1 Successful output of runtest.cmd for DB2JDBC

```
Starting TraderTest application with following input:
  Name service: com.sun.jndi.cosnaming.CNCtxFactory
  Naming Server: iiop://hecate:900/
  JNDI name: itso/ejb390/trader/Trader
  Call type: CICSConnectorCCF
  CTG: tcp://wtsc61oe.itso.ibm.com:2006
  CICS region: SCSCPJA5

Casey_Import_Export
Glass_and_Luget_Plc
Headworth_Electrical
IBM
CommissionCostBuy 010
CommissionCostSell 015
NumberOfShares 0228
TotalShareValue 000037164.00
UnitSharePrice 00163.00
UnitValue1Days 00163.00
UnitValue2Days 00162.00
UnitValue3Days 00160.00
UnitValue4Days 00161.00
UnitValue5Days 00159.00
UnitValue6Days 00156.00
UnitValue7Days 00157.00
Now we buy 5 shares ...
```

Testing the enterprise bean with TraderServlet

In addition, it is also possible to test the Trader enterprise bean using our TraderServlet that we developed in 7.4.2, “Servlet development with VisualAge for Java” on page 199. In this case, we provided the following input parameters to the initial servlet HTML:

Communication type	`CICSConnectorCCF`
JndiPrefix	`itso/ejb390/trader`
NameService	`com.ibm.ejs.ns.jndi.CNInitialContextFactory`
ProviderURL	`iiop://hecate:900/`
URL to connect to	`tcp://wtsc61oe.itso.ibm.com:2006`
CICS Server	`SCSCPJA5`

The output of the TraderServlet was the same as shown in “Testing the Trader servlet” on page 212.

8.4 Deploying the enterprise bean to CICS

In order to demonstrate the platform neutrality of using the CICS Connector, we took TraderBean and our new TraderBackendConnectorCCF class and deployed them directly into our CICS region instead of into WebSphere Application Server. No changes were necessary, as the TraderBean can use the CICS connector for CICS TS to invoke the required program. The only difference is that a URL of *local:* must be used and the SYSID of the region must be used instead of the APPLID, since the CICS Transaction Gateway is no longer used to provide connectivity to CICS.

This scenario is illustrated in Figure 8-16.

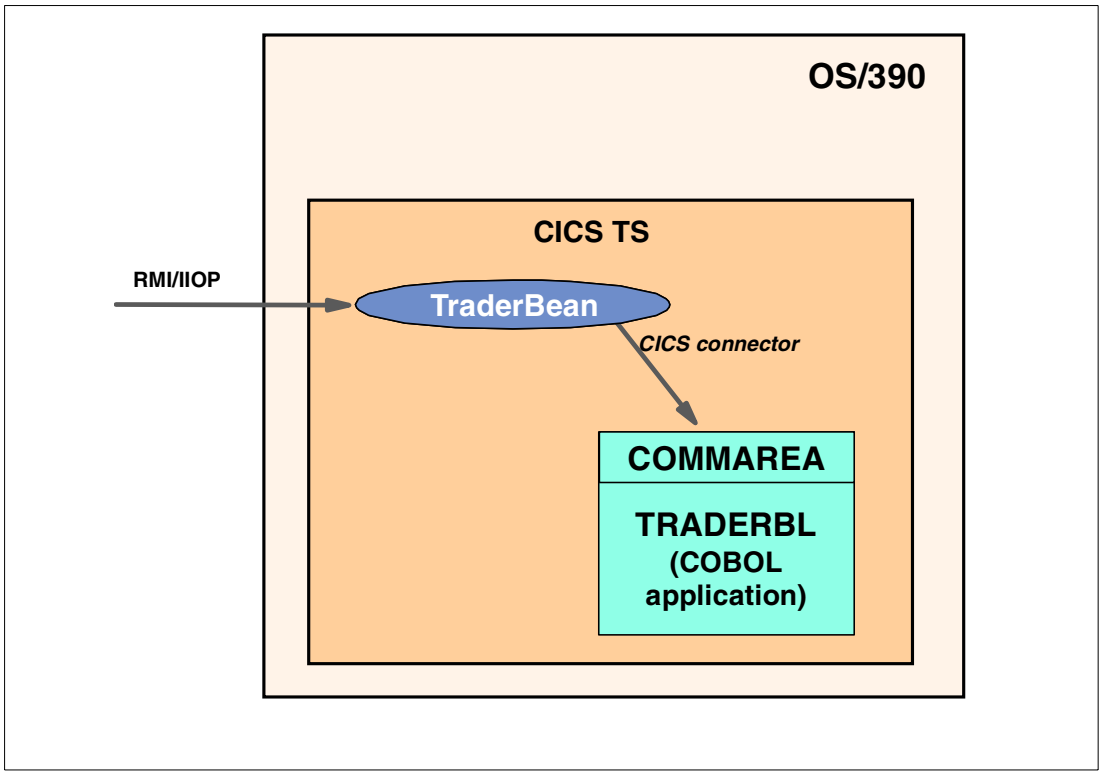


Figure 8-16 Deploying TraderBean to CICS

We do not need to modify the code we produced; the only steps we need to take are in the deployment of the TraderBean and its associated classes.

We need to do the following steps:

1. Export the enterprise bean and its related classes.
2. Convert the exported file to a deployed JAR file.
3. Send the JAR file to the OS/390 system.
4. Send the supporting JAR files to the OS/390 system.
5. Add the supporting JAR files to the trusted middleware classpath.
6. Refresh the DJAR in the CICS shelf.

Export the enterprise bean and its related classes

Export the enterprise bean the same way as described in 7.3.1, “Exporting the enterprise bean and its related classes” on page 191. The difference is, that you have to include the three new classes *TraderCommand*, *TraderCommandBeanInfo*, and *TraderBackendCICSConnectorCCF*. Therefore, you should now have fourteen classes and one bean selected.

Convert the exported file to a deployed JAR file

You should generate the deployed JAR file using the CICS JAR development tool as described in 7.3.2, “Converting the exported file to a deployed JAR file” on page 192. If the file `CCF.jar` file is not in your system CLASSPATH, you will need to add this to the CLASSPATH in `cicsjdt.bat as` described 7.3.2, “Converting the exported file to a deployed JAR file” on page 192. On our workstation the `CCF.jar` was located in `C:\IBM Connectors\classes`.

Send the JAR file to the OS/390 system

Send the deployed JAR file `traderForCICS_GEN.jar` to your HFS on OS/390 as described in 7.3.3, “Sending the deployed JAR file to OS/390” on page 193 to the OS/390 system.

Send the supporting JAR files to the OS/390 system

Because TraderBackendCICSConnectorCCF uses the CCF, the CCF classes in `CCF.jar` need to be made accessible to the CICS JVM. We FTPed the file `CCF.jar` in directory `C:\IBM Connectors\classes` to `/u/cicsts21/lib` on our OS/390 system.

Add the supporting JAR files to the trusted middleware classpath

As described in 7.3.6, “Adding the supporting JAR files to the trusted middleware classpath” on page 194, we need to change the trusted middleware classpath for our CICS region to provide the CCF classes. After the changes, DFHJVMPR in CICSSYSF.CICS610.DFHJVM looks as illustrated in Figure 8-17.

```
TMSUFFIX=/u/cicsts21/lib/eablib.jar:\
         /u/cicsts21/lib/CCF.jar
```

Figure 8-17 Modifying trusted middleware classpath for use of CCF

Refresh the DJAR in the CICS shelf

The DJAR now has to be deployed to the CICS system. In this sample we assume that you have already defined the DJAR to CICS as described in Section 7.3.4, “Defining the DJAR in the CICS system” on page 193. Therefore it is only necessary to discard the existing DJAR and re-install as follows:

```
CEMT DISCARD DJAR(TRADER
CEDA INSTALL GR(ITSOEJB) DJAR(TRADER)
```

8.4.1 Testing the enterprise bean running in CICS

We tested our enterprise bean in two ways, with our TraderTest standalone application and with our TraderServlet application. For both environments because our Trader enterprise bean is now running in another container, we had to use the original JNDI name as specified on the Jndiprefix of ITSO/PJA5 in the CICS CORBASERVER definition.

TraderTest is a standalone Java application that can be run from either a command line or within VAJ. Further instructions on how to use this from the command line are given in 8.1, “Quick start — Invoking TraderBean” on page 219. Note that since we were using the CICS connector for CICS TS, the URL we had to use was *local:* and the CICS sysid of *PJA5* had to be specified as the CICS region.

We edited the following line in the `runtest.cmd` script so as to invoke the TraderTest application with the CICSConnectorCCF option.

```
java -classpath ".;traderTest.jar;traderCLI.jar;C:\Program Files\IBM\CICS TS 2.1
Tools\Common\j2ee.jar" itso.ejb390.trader.test.TraderTest
com.sun.jndi.cosnaming.CNCtxFactory iiop://hecate:900/ ITSO/PJA5 CICSConnectorCCF local:
PJA5
```

The successful output of runtest.cmd is shown in Example 8-2.

Example 8-2 Successful output of runtest.cmd for CICSConnectorCCF

```
Starting TraderTest application with following input:
  Name service: com.sun.jndi.cosnaming.CNCtxFactory
  Naming Server: iiop://hecate:900/
  JNDI name: ITSO/PJA5
  Call type: CICSConnectorCCF
  CTG: local:
  CICS region: PJA5

Casey_Import_Export
Glass_and_Luget_Plc
Headworth_Electrical
IBM
CommissionCostBuy 010
CommissionCostSell 015
NumberOfShares 0228
TotalShareValue 000037164.00
UnitSharePrice 00163.00
UnitValue1Days 00163.00
UnitValue2Days 00162.00
UnitValue3Days 00160.00
UnitValue4Days 00161.00
UnitValue5Days 00159.00
UnitValue6Days 00156.00
UnitValue7Days 00157.00
Now we buy 5 shares ...
```

Testing the enterprise bean with TraderServlet

In addition it is also possible to test the Trader enterprise bean using our TraderServlet that we developed in 7.4.2, “Servlet development with VisualAge for Java” on page 199. In this case we provided the following input parameters to the initial servlet HTML form.

Communication type	`CICSConnectorCCF`
JndiPrefix	`ITSO/PJA5`
NameService	`com.ibm.ejs.ns.jndi.CNInitialContextFactory`
ProviderURL	`iiop://hecate:900/`
URL to connect to	`local:`
CICS Server	`PJA5`

The output of the TraderServlet was the same as shown in , “Testing the Trader servlet” on page 212.

8.5 Summary

This chapter has shown how to wrap a CICS COBOL program with Enterprise JavaBean technology. We have developed a universal enterprise bean which is able to invoke a COBOL program by either using JCICS or the CICS Connector. Of course this way of doing it is not bound to COBOL programs. You can basically invoke any kind of CICS program, no matter if it is written in Assembler, COBOL, PL/I, C, C++, or a CICS Java program using the High Performance Compiler for Java.

As you will see in the next chapters, the enterprise bean will be extended in such a way, that instead of calling the existing COBOL program, new back-end classes are developed. These classes have the same functionality as the COBOL Trader program, but are written in Java and access either the existing VSAM files or a DB2 database.

The reasons why we have done this are twofold. First, we wanted show how to access VSAM files and relational data directly from an enterprise bean in CICS. Second, we wanted to illustrate, how a possible migration path in a real -world environment might look.

Wrapping existing CICS programs with enterprise beans can be seen as tactical solution to enable very fast enterprise technology on OS/390 systems. Re-writing existing applications in pure Java can be seen as the strategic way to invent enterprise technology on OS/390 systems. Therefore, one scenario could be to wrap first existing applications very quickly to have them accessible as enterprise beans, while in the meantime the applications are re-written in pure Java technology and modern software development tools.

This scenario also shows that you can easily change the implementation behind the scene, without changing any interface of the enterprise bean. Therefore, no client code needs to be changed and distributed to the workstations, which could save a lot of time and money.

9

Rewriting the COBOL Trader application with JCICS

In this chapter we describe how to re-implement the COBOL Trader application in Java using enterprise bean technology and the JCICS classes. The goal was to rewrite the COBOL business logic in Java, but still to access the underlying VSAM files in the same manner. This is illustrated in Figure 9-1.

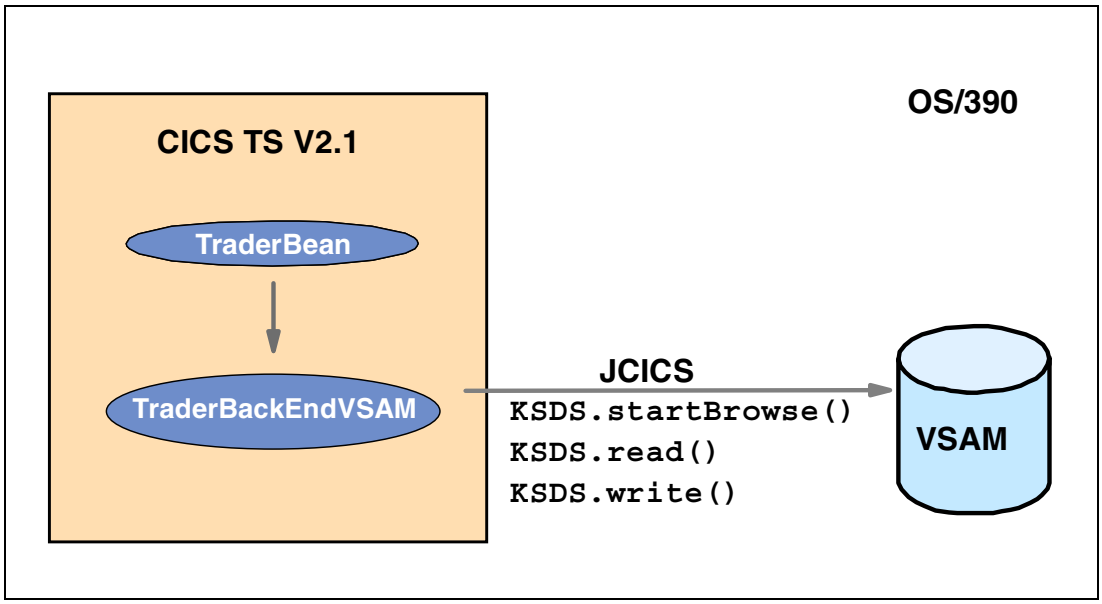


Figure 9-1 Rewriting Trader in Java

9.1 Quick start — Invoking TraderBean

If you want to run our sample Trader enterprise bean without following all the details specified in this chapter, use the steps below. All the source code and examples used in this book are available for download from the redbooks Web site `ftp://www.redbooks.ibm.com/redbooks/` and for full details of the available files and how to obtain them, you can refer to Appendix C, “Using the additional material” on page 315.

1. Install the COBOL Trader application in your CICS system. For details refer to Appendix B, “The COBOL Trader application” on page 309.
2. Create a CICS TCPIPSERVICE, CORBASERVER, REQUESTMODEL and DJAR definition if you have not already done so, for more details refer to 6.3.3, “Deploying to CICS” on page 150.
3. Deploy the TraderBean and the associated back-end classes to your CICS TS V2.1 region. (see 9.3, “Deploying the enterprise bean to CICS” on page 249).
4. Test the application, this can be achieved in one of the following two ways:
   a. Use our supplied *TraderServlet* to create a Web application with a HTML front-end to TraderBean. For further details on the expected output refer to Figure 7-50 on page 214.
   b. Use the supplied runTest.cmd file to invoke our stand-alone Java test application *TraderTest.* To set up TraderTest, simply do the following:
      - On your workstation, create a directory (for example `C:\itsotrader`) and copy the following sample files to this directory:

        ```
        traderCLI.jar
        traderTest.jar
        runTest.cmd
        ```

      - Ensure that you have a Java 2 runtime environment at version 1.3 or greater on your workstation installed. You can verify your version with the command:

        ```
        java -version
        ```

      - Ensure that you have file `j2ee.jar` accessible on your workstation. If not, you can either obtain it if you install the CICS development deployment tool or by installing Java 2 SDK, Enterprise Edition available from:

        `http://java.sun.com`

      - Invoke TraderTest using the `runtest.cmd` file. You will need to alter the input parameters as documented in the file. For further details and for an example of expected output, refer to 9.3.1, “Testing the enterprise bean” on page 250.

9.2 Adapting TraderBean to use JCICS

In Chapter 7, “Wrapping the Trader application: JCICS link” on page 171 we described how we designed, implemented, and deployed an enterprise bean using the JCICS `link()` method to invoke a COBOL program. We designed the bean in such a way that to change the way the business logic is invoked or implemented requires only minimal changes to the bean itself.

In order to rewrite the business logic with Java using JCICS to access the VSAM files, we had to perform the following steps:.

1. Implement a new class TraderBackendVsam
2. Build a Java record using the Java Record Framework.

TraderBean.loadClass()

Before we wrote the TraderBackendVsam class we modified the `loadClass()` method in TraderBean to be able to load this back-end class (Figure 9-2); this was the only change necessary to this class.

```
private void loadClass(String type) throws Exception {

Class loadClass=null;
if( type.equalsIgnoreCase("JCICS-COBOL") == true ) {
  loadClass = Class.forName("itso.ejb390.trader.TraderBackendJcics");
}
else if( type.equalsIgnoreCase("CICSConnectorCCF") == true ) {
  loadClass = Class.forName("itso.ejb390.trader.TraderBackendCICSConnectorCCF");
}
else if( type.equalsIgnoreCase("JCICS-Java") == true ) {
  loadClass = Class.forName("itso.ejb390.trader.TraderBackendVsam");
}
else {
  throw new TraderException( "You specified unknown type " + type );
}
ivTraderBackend = (TraderBackend)loadClass.newInstance();
}
```

Figure 9-2 TraderBean.loadClass() loading TraderBackendVsam

The declaration of class TraderBackendVsam is shown in Figure 9-3.

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

public class TraderBackendVsam implements TraderBackend {
  private final static java.lang.String CUSTFILE = "CUSTFILE";
  private final static java.lang.String COMPFILE = "COMPFILE";
  private final static java.lang.String DOT = ".";
}
```

Figure 9-3 Declaration of TraderBackendVsam

The class declares the string constants `CUSTFILE` and `COMPFILE` which represent the CICS file definitions of the VSAM files. String constant `DOT` is the separator character for the key of VSAM file `CUSTFILE`.

Because the class must implement the TraderBackend interface, it is necessary to implement the following methods:

- logon()
- logoff()
- getCompanies()
- getQuotes()
- buy()
- sell()
- ejbBackendCreate()
- ejbBackendRemove()
- ejbBackendActivate()
- ejbBackendPassivate()

How these methods and classes used by these methods are implemented is shown in the next sections.

9.2.1 Java Record Framework

VSAM files are accessed from a Java program in CICS using the JCICS API. JCICS classes define methods which usually take byte arrays as arguments. These arrays contain either data keys of VSAM records or VSAM record data. VisualAge for Java Enterprise Edition comes with the Enterprise Access Builder plug-in which allows you to create Java classes representing such keys or data records from COBOL data definitions. These generated classes also provide built-in character translation from Unicode to EBCDIC or ASCII and vice versa.

The COBOL Trader application operates on the two VSAM files *COMPANY* and *CUSTOMER*. How the data records and data keys are mapped to Java classes is described in the following sections.

COMPANY record

COMPANY represents all existing companies it is possible for the user to buy shares in. We extracted the COBOL data definition representing the COMPANY-IO-BUFFER structure and saved it in file `company.txt`, this is shown in Figure 9-4, and is supplied along with our sample code.

```
      01 COMPANY-IO-BUFFER.
          03 COMPANY                   PIC X(20).
          03 SHARE-VALUE.
             05 SHARE-VALUE-INT-PART   PIC X(05).
             05 FILLER                 PIC X(01).
             05 SHARE-VALUE-DEC-PART   PIC X(02).
          03 VALUE-1                   PIC X(08).
          03 VALUE-2                   PIC X(08).
          03 VALUE-3                   PIC X(08).
          03 VALUE-4                   PIC X(08).
          03 VALUE-5                   PIC X(08).
          03 VALUE-6                   PIC X(08).
          03 VALUE-7                   PIC X(08).
          03 COMMISSION-BUY            PIC X(03).
          03 COMMISSION-SELL           PIC X(03).
```

Figure 9-4 company.txt

One record consists of a company name, the current share value, the share values of the last seven days, and commission rates. The commission rates are for future use and are therefore not relevant for this implementation. We will now create with VAJ the Java class CompanyRecord representing this VSAM record.

1. In VisualAge for Java select **Workspace -> Tools -> Enterprise Access Builder -> Import COBOL to Record Type** to open the *Import COBOL to Record Type SmartGuide*.
2. Click **Browse** to open the file dialog.
   a. Navigate through your directory structure to select file `company.txt` which contains the COBOL definition of the COMPANY-IO-BUFFER.
   b. Click **Open** to select the file and close the file dialog window.
3. Make sure that you import the COBOL definition for generic COBOL code. You can select this option in the window.
4. Click **Next**.

5. Now you can select one or more *01 COMMAREA* levels to be imported by the *SmartGuide*. All available 01 levels are listed on the left side of the window. Because our COBOL data structure has only one 01 level, the left side shows just COMPANY-IO-BUFFER.

> **Attention:** This data structure is not a COMMAREA, it is just a normal COBOL data definition. Therefore, the description of the *Import COBOL to Record Type* window might be a little bit misleading.

6. Select COMPANY-IO-BUFFER and click the **>** arrow in the middle of the window.
7. Now COMPANY-IO-BUFFER switches to the right side which shows the selected COMMAREAS. Click **Next**.
8. As project name enter *ITSO EJB 390 Redbook* and for package name enter *itso.ejb390.trader*. You can also use the **Browse** buttons to find the project and package names.
9. As the class name specify *CompanyRecordType*.
10. Click **Finish** to create the record type, and to create a record from the record type.

Now a *SmartGuide* opens which allows you to create a record from a record type. This *SmartGuide* will create the class CompanyRecord which is a Java class representing one data record of VSAM file COMPANY. This is shown in the next steps.

1. As class name type CompanyRecord. All other options should be left as they are.
2. Click **Next** to modify the properties of the record.
3. Enter the following:

Floating Point Format	IBM
Remote Integer Endian	Big Endian
Endian	Big Endian
Code Page	037
Machine Type	MVS

> **Note:** If you specify an EBCDIC code page such as 037 for conversion of character data, you should ensure that you do not also use the CICS DFHCNV templates to convert the COMMAREA data from ASCII to EBCDIC, otherwise you will experience corruption of data due to double conversion. For more details on data conversion with Java in CICS refer to the redbook *Revealed! Architecting Web Access to CICS*, SG24-5466.

4. Click **Finish** to create CompanyRecord.

If you now view the classes of package itso.ejb390.trader, you will see that VisualAge has added the classes *CompanyRecordType*, *CompanyRecord*, and *CompanyRecordBeanInfo*. CompanyRecord is the class which we will use as the representation of our VSAM data record for file COMPANY.

COMPANY record key

The key of COMPANY is company name. Therefore we created file `companyKey.txt`, which just contains the COBOL definition for company name. The file is illustrated in Figure 9-5.

```
       01 COMPANY-IO-BUFFER.
          03 COMPANY                   PIC X(20).
```

Figure 9-5 companyKey.txt

Now use this file as input to the VisualAge *Import COBOL to Record Type SmartGuide* as described in “COMPANY record” on page 238. The output of this operation should be the classes CompanyKeyRecordType, CompanyKeyRecord, and CompanyKeyRecordBeanInfo.

CUSTOMER record

The CUSTOMER record represents all existing customer to company relations. We have extracted the COBOL data definition representing one CUSTOMER record and saved it in the file `customer.txt` which is shown in Figure 9-6.

```
       01 CUSTOMER-IO-BUFFER.
          03 KEYREC.
             05 CUSTOMER               PIC X(60).
             05 KEYREC-DOT             PIC X(01).
             05 COMPANY                PIC X(20).
          03 CONVERT1.
             05 NO-SHARES              PIC X(04).
          03 CONVERT2 REDEFINES CONVERT1.
             05 DEC-NO-SHARES          PIC 9(04).
          03 BUY-FROM                  PIC X(08).
          03 BUY-FROM-NO               PIC X(04).
          03 BUY-TO                    PIC X(08).
          03 BUY-TO-NO                 PIC X(04).
          03 SELL-FROM                 PIC X(08).
          03 SELL-FROM-NO              PIC X(04).
          03 SELL-TO                   PIC X(08).
          03 SELL-TO-NO                PIC X(04).
          03 ALARM-PERCENT             PIC X(03).
```

Figure 9-6 customer.txt

One record consists of a customer, a separator, a company, and the number of shares the customer holds for this company. Customer, separator, and company are the record key. For this application, all the other fields can be ignored, as they are not used.

Now use this file as input to the VisualAge *Import COBOL to Record Type SmartGuide* as described in “COMPANY record” on page 238. The output of this operation should be the classes CustomerRecordType, CustomerRecord, and CustomerRecordBeanInfo.

CUSTOMER record key

The key of CUSTOMER is customer, a separator, and company. Therefore we created file `customerKey.txt`, which just contains these COBOL definitions. The file is illustrated in Figure 9-7.

```
01 CUSTOMER-IO-BUFFER.
          03 KEYREC.
             05 CUSTOMER              PIC X(60).
             05 KEYREC-DOT            PIC X(01).
             05 COMPANY               PIC X(20).
```

Figure 9-7 customerKey.txt

Now use this file as input to the VisualAge *Import COBOL to Record Type SmartGuide* as described in “COMPANY record” on page 238. The output of this operation should be the classes CustomerKeyRecordType, CustomerKeyRecord, and CustomerKeyRecordBeanInfo.

9.2.2 Implementing TraderBackendVsam

The following section details how we implemented the methods in the new back-end class TraderBackendVsam.

TraderBackendVsam.getCompanies()

Now that we have created the record classes, we can implement method `getCcompanies()`. The code needs to do the following:

- Instantiate a CompaniesBean which is returned as a result later.
- Instantiate a KSDS (Keyed Sequenced Data Set) object.
- Tell the KSDS object the name of the CICS file resource name.
- Obtain a KeyedFileBrowse object from object KSDS with an empty key.
- Iterate through the file, read one record after each other, and store the result in CompaniesBean.
- End the file browse.
- Return the CompaniesBean instance.

Figure 9-8 shows the implementation of method `getCompanies()`.

```
public CompaniesBean getCompanies() throws Exception {

CompaniesBean companies = new CompaniesBean();

KSDS companiesKSDS = new KSDS();
companiesKSDS.setName(COMPFILE);

// obtain a browser
CompanyKeyRecord companyKey = new CompanyKeyRecord();
companyKey.setInitialValues();

KeyedFileBrowse kfb = companiesKSDS.startBrowse(
                              companyKey.getBytes() );

KeyHolder kh = new KeyHolder( companyKey.getBytes() );
RecordHolder rh = new RecordHolder(  );

// now iterate over companies
for( int i=0; i<4; i++ ) {
  kfb.next(rh, kh );
  CompanyRecord cr = new CompanyRecord( rh.value );
  companies.addCompany( cr.getCompany() );
}

// end browsing
kfb.end();

return companies;
}
```

Figure 9-8 TraderBackend.getCompanies()

TraderBackendVsam.getQuotes()

This method returns quote information related to a specific customer and company. The method does the following:

- Instantiates a QuotesBean which is returned as result later.
- Tries to read a customer - company record. If the record is not found, an empty record is created.
- From the customer record retrieves the actual number of shares the customer holds for this company and store it in QuotesBean.
- Reads company data.
- Stores the result in QuotesBean.

Figure 9-9 shows the implementation of method `getQuotes()`.

```
public QuotesBean getQuotes(String company, String userID)
                                            throws Exception {
QuotesBean quotes = new QuotesBean();

// read customer data
CustomerRecord cur = new CustomerRecord( );
try {
  readCustomer(company,userID, cur, false );
} catch( RecordNotFoundException rnfe ) {
  createCustomer( company, userID, cur );
}

// set number of shares
quotes.setNumberOfShares( cur.getNo__Shares() );

// read company data
CompanyRecord cor = new CompanyRecord( );
readCompany( company, cor );

// copy company data
String shareValueString = cor.getShare__Value__Int__Part() +
                      "." + cor.getShare__Value__Dec__Part();

double shareValue = Double.valueOf(shareValueString).doubleValue();
double shareValueTotal = shareValue * cur.getDec__No__Shares();

quotes.setTotalShareValue( String.valueOf(shareValueTotal) );
quotes.setUnitSharePrice( cor.getShare__Value__Int__Part() +
                  "." + cor.getShare__Value__Dec__Part() );
quotes.setUnitValue1Days( cor.getValue__1() );
quotes.setUnitValue2Days( cor.getValue__2() );
quotes.setUnitValue3Days( cor.getValue__3() );
quotes.setUnitValue4Days( cor.getValue__4() );
quotes.setUnitValue5Days( cor.getValue__5() );
quotes.setUnitValue6Days( cor.getValue__6() );
quotes.setUnitValue7Days( cor.getValue__7() );
quotes.setCommissionCostSell( cor.getCommission__Sell() );
quotes.setCommissionCostBuy( cor.getCommission__Buy() );

return quotes;
}
```

Figure 9-9 TraderBackendVsam.getQuotes()

As you can see, this method calls the three private methods `readCustomer()`, `createCustomer()`, and `readCompany()`. How these methods are implemented is shown next.

TraderBackendVsam.readCustomer()

This method reads one customer record. The key is customer name and company. The method does the following:

- Instantiates a KSDS object.
- Tells the KSDS object the name of the CICS file resource name for customer.
- Instantiates a CustomerKeyRecord and store the key values *company*, *separator*, and *customer* name.

- Reads one record providing the key and stores a result object. The method distinguishes between a read and a read for update operation which is controlled with as a *boolean* flag provided as parameter to `readCustomer()`.
- Copies the result to a result object provided as parameter to this method.

The implementation of this method is illustrated in Figure 9-10.

```
private void readCustomer(String company, String userID,
                          CustomerRecord curOut, boolean forUpdate )
                                                    throws Exception {

KSDS customerKSDS = new KSDS();
customerKSDS.setName(CUSTFILE);

// create key
CustomerKeyRecord ckr = new CustomerKeyRecord();
ckr.setInitialValues();
ckr.setCompany(company);
ckr.setKeyrec__Dot(DOT);
ckr.setCustomer(userID);

// read customer record
RecordHolder rh = new RecordHolder(  );

if( forUpdate == false )       // if for read only
  customerKSDS.read(ckr.getBytes(), rh );
else                           // if read for update
  customerKSDS.readForUpdate(ckr.getBytes(), rh );

// copy customer results
curOut.setBytes( rh.value );
}
```

Figure 9-10 TraderBackendVsam.readCustomer()

TraderBackendVsam.createCustomer()

This method creates an empty customer record. It does the following:

- Sets company, customer name, and separator which are also provided as parameters.
- Sets the number of shares to zero.
- Writes the new record.

How the method is implemented is shown in Figure 9-11.

```
private void createCustomer(String company, String userID,
                            CustomerRecord curOut ) throws Exception {

curOut.setCompany(company);
curOut.setCustomer(userID);
curOut.setKeyrec__Dot(DOT);
curOut.setDec__No__Shares((short)0);

writeCustomer(company, userID, curOut );

}
```

Figure 9-11 TraderBackendVsam.createCustomer()

The method uses the private method `writeCustomer()` which is described below.

TraderBackendVsam.writeCustomer()

This method writes one customer record. The key is the `company`, the `customer name`, and the `separator` character. The method does the following:

- Instantiates a KSDS object.
- Tells the KSDS object the name of the CICS file resource name for customer.
- Instantiates a CustomerKeyRecord object and assign it company, customer name, and separator character.
- Invokes the `write()` method of instance KSDS.

How the method is implemented is illustrated in Figure 9-12.

```
private void writeCustomer(String company, String userID,
                           CustomerRecord cur ) throws Exception {

KSDS customerKSDS = new KSDS();
customerKSDS.setName(CUSTFILE);

// create key
CustomerKeyRecord ckr = new CustomerKeyRecord();
ckr.setCompany(company);
ckr.setKeyrec__Dot(DOT);
ckr.setCustomer(userID);

// read customer record
customerKSDS.write( ckr.getBytes(), cur.getBytes() );

}
```

Figure 9-12 TraderBackendVsam.writeCustomer()

TraderBackendVsam.readCompany()

This method reads one company record. The key is the company. The method does the following:

- Instantiates a KSDS object.
- Tells the KSDS object the name of the CICS file.
- Instantiates a CompanyKeyRecord and stores the key value company.

- Reads one record providing the key and a result object.
- Copies the result to a result object provided as parameter to this method.

The implementation of this method is illustrated in Figure 9-13.

```
private void readCompany(String company, CompanyRecord corOut)
                                                throws Exception {

// read company data
KSDS companyKSDS = new KSDS();
companyKSDS.setName(COMPFILE);

CompanyKeyRecord companyKey = new CompanyKeyRecord();
companyKey.setInitialValues();
companyKey.setCompany(company);

RecordHolder rh = new RecordHolder(  );
companyKSDS.read(companyKey.getBytes(), rh );
corOut.setBytes( rh.value );

}
```

Figure 9-13 TraderBackendVsam.readCompany()

Implement TraderBackendVsam.buy()

This method is used to buy a specific amount of shares of a company. The method just invokes the private method `doBuyOrSell()` as shown in Figure 9-14.

```
public void buy(String company, String userID, int numberOfShares)
                                                   throws Exception {

doBuyOrSell(company, userID, numberOfShares, true );

}
```

Figure 9-14 TraderBackendVsam.buy()

TraderBackendVsam.doBuyOrSell()

This method is used to either buy or sell shares of a specific customer for a given company. It performs the following steps:

- Tries to read the company file. If the company does not exists, an exception is thrown.
- Tries to find a customer/company record in customer file. If no record is found, an empty record is created. If the record is found it is locked for update.
- If shares are bought, the requested number of shares are added to the actual number of shares the customer holds for the company.
- If shares are sold, it first checks if the actual held number of shares is greater or equal the number of shares to be sold. If yes, the requested number of shares are subtracted from the actual number of shares.
- The record is updated.

The implementation of the method is illustrated in Figure 9-15.

```
private void doBuyOrSell(String company, String userID, int numberOfShares, boolean
doBuy) throws Exception {

   // read company to check if it exists (if not, an exception is thrown)
   CompanyRecord cor = new CompanyRecord( );
   readCompany( company, cor );

   // try to read customer for update
   CustomerRecord cur = new CustomerRecord();
   boolean rnfeException = false;

   try {
      readCustomer( company, userID, cur, true );
   } catch( RecordNotFoundException rnfe ) {
      rnfeException = true;
   }

   // if buy
   if( doBuy == true ) {
      if (rnfeException==false) {
      cur.setDec__No__Shares( (short)
         (cur.getDec__No__Shares() + numberOfShares) );
      }
      else { // if new customer
         cur.setDec__No__Shares( (short)(numberOfShares) );
      }

   } else {// if sell
      if( cur.getDec__No__Shares() >= numberOfShares )// if enough to sell
         cur.setDec__No__Shares( (short)
            (cur.getDec__No__Shares() - numberOfShares) );
   }

   // update customer
   if ( rnfeException == false ) {
      updateCustomer( cur );
   }  else {
      createCustomer( company, userID, cur );
      }

}
```

Figure 9-15 TraderBackendVsam.doBuyOrSell()

This method calls private method `updateCustomer()`, which is described below.

TraderBackendVsam.updateCustomer()

This method updates a customer record and does the following:

- Instantiates a KSDS object.
- Tells the KSDS object the name of the CICS file resource name for customer.
- Invokes method rewrite of the KSDS instance.

How the method is implemented is shown in Figure 9-16.

```
private void updateCustomer( CustomerRecord cur ) throws Exception {

KSDS customerKSDS = new KSDS();
customerKSDS.setName(CUSTFILE);

// update customer record
customerKSDS.rewrite( cur.getBytes() );

}
```

Figure 9-16 TraderBckendVsam.updateCustomer()

Implement TraderBackendVsam.sell()

This method is used to sell a specific amount of shares of a company. The method just invokes the private method `doBuyOrSell()` as shown in Figure 9-17.

```
public void sell(String company, String userID, int numberOfShares)
                                                    throws Exception {

  doBuyOrSell(company, userID, numberOfShares, false );

}
```

Figure 9-17 TraderBackendVsam.sell()

The remaining methods of TraderBackendVsam

Because TraderBackendVsam implements TraderBackend, the following methods need to be implemented as well.

- `logon()`
- `logoff()`
- `ejbBackendCreate()`
- ejbBackendRemove()
- `ejbBackendActivate()`
- `ejbBackendPassivate()`

For this back-end class, these remaining methods have no specific implementation, and are implemented only for compatibility with the future JDBC back-end. Therefore, they just need to be defined but have no additional business logic implemented (Figure 9-18).

```
public void logon(String userID, String password,
                  String connectURL, String cicsServer) throws Exception {
}

public void logoff() throws Exception {
}

public void ejbBackendCreate() throws javax.ejb.CreateException,
                                java.rmi.RemoteException {
}

public void ejbBackendRemove() throws java.rmi.RemoteException {
}

public void ejbBackendActivate() throws java.rmi.RemoteException {
}

public void ejbBackendPassivate() throws java.rmi.RemoteException {
}
```

Figure 9-18 Remaining methods of TraderBackendVsam

Now that we have implemented TraderBackendVsam, we need to deploy the modified enterprise bean to CICS. This is described in the next section.

9.3 Deploying the enterprise bean to CICS

This section describes how to deploy TraderBean and its related classes to CICS.

We need to do the following steps:

1. Export the enterprise bean and its related classes.
2. Convert the exported file to a deployed JAR file.
3. Send the deployed JAR file to OS/390.
4. Reinstall DJAR TRADER.

Export the enterprise bean and its related classes

Export the enterprise bean the same way as described in 7.3.1, “Exporting the enterprise bean and its related classes” on page 191. The only difference is that you also need to select the following additional classes built using the Java Record Framework.

- CompanyKeyRecord
- CompanyKeyRecordBeanInfo
- CompanyKeyRecordType
- CompanyRecord
- CompanyRecordBeanInfo
- CompanyRecordType
- CustomerKeyRecord
- CustomerKeyRecordBeanInfo
- CustomerKeyRecordType
- CustomerRecord
- CustomerRecordBeanInfo
- CustomerRecordType
- TraderBackendVsam

Therefore you should have 27 classes and one bean selected.

Convert the exported file to a deployed JAR file

Generate the deployed JAR file using the CICS JAR development tool as described in 7.3.2, “Converting the exported file to a deployed JAR file” on page 192.

Send the JAR file to the OS/390 system

Send the deployed JAR file `traderForCICS_GEN.jar` to your HFS on OS/390 in the same way as described in 7.3.3, “Sending the deployed JAR file to OS/390” on page 193 to the OS/390 system.

Refresh the DJAR in the CICS shelf

The DJAR now has to be deployed to the CICS system. In this sample we assume that you have already defined the DJAR to CICS as described in 7.3.4, “Defining the DJAR in the CICS system” on page 193. Therefore it is only necessary to discard the existing DJAR and re-install as follows:

```
CEMT DISCARD DJAR(TRADER
CEDA INSTALL GR(ITSOEJB) DJAR(TRADER)
```

9.3.1 Testing the enterprise bean

We tested our enterprise bean in two ways, with our TraderTest standalone application and with our TraderServlet application.

TraderTest is a standalone Java application that can be run from either a command line or within VAJ. Further instructions on how to use this from the command line are given in 9.1, “Quick start — Invoking TraderBean” on page 236.

We edited the following line in the `runtest.cmd` script so as to invoke the traderTest application with the CICSConnectorCCF option.

```
java -classpath ".;traderTest.jar;traderCLI.jar;C:\Program Files\IBM\CICS TS 2.1
Tools\Common\j2ee.jar" itso.ejb390.trader.test.TraderTest
com.sun.jndi.cosnaming.CNCtxFactory iiop://hecate:900/ ITSO/PJA5 JCICS-Java
```

The successful output of `runtest.cmd` is shown in Example 9-1.

Example 9-1 Output of runtest.cmd for JCICS-Java

```
Starting TraderTest application with following input:
  Name service: com.sun.jndi.cosnaming.CNCtxFactory
  Naming Server: iiop://hecate:900/
  JNDI name: ITSO/PJA5
  Call type: JCICS-Java

Casey_Import_Export
Glass_and_Luget_Plc
Headworth_Electrical
IBM
CommissionCostBuy 010
CommissionCostSell 015
NumberOfShares 0228
TotalShareValue 000037164.00
UnitSharePrice 00163.00
UnitValue1Days 00163.00
UnitValue2Days 00162.00
UnitValue3Days 00160.00
UnitValue4Days 00161.00
```

```
UnitValue5Days 00159.00
UnitValue6Days 00156.00
UnitValue7Days 00157.00
Now we buy 5 shares
...
```

Testing the enterprise bean with TraderServlet

In addition, it is also possible to test the Trader enterprise bean using our TraderServlet that we developed in 7.4.2, “Servlet development with VisualAge for Java” on page 199. In this case it is necessary to provide the following input parameters to the initial servlet HTML form.

Communication type	JCICS-Java
JndiPrefix	ITSO/PJA5
NameService	com.ibm.ejs.ns.jndi.CNInitialContextFactory
ProviderURL	iiop://hecate:900/

The output of the TraderServlet was the same as shown in “Testing the Trader servlet” on page 212.

9.4 Summary

This scenario has illustrated how it is possible to modify the business logic within a traditional CICS COBOL application, without affecting the external presentation interface. It has also demonstrated the powerful nature of the Java Record Framework in mapping Java classes onto the underlying CICS VSAM files.

10

Rewriting the Trader session bean using JDBC/SQLJ

In this chapter we describe how to access DB2 data within an enterprise bean running in the CICS TS V2.1 container, using both the Java Database Connectivity (JDBC) API and the SQL Java (SQLJ) API. We show how we re-wrote the CICS Trader application to access DB2 data instead of CICS VSAM files, using JDBC or SQLJ calls. Our new configuration is illustrated in Figure 10-1.

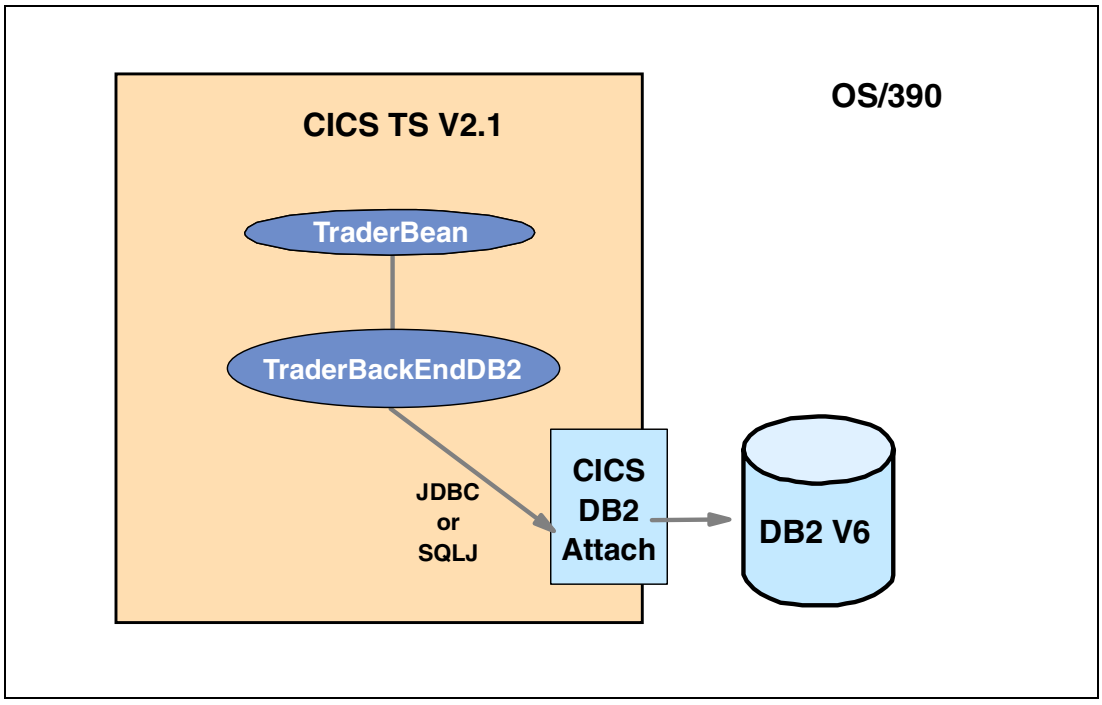


Figure 10-1 TraderBean using JDBC or SQLJ

In Chapter 7, “Wrapping the Trader application: JCICS link” on page 171 we show you how we developed an enterprise bean (TraderBean) to wrap the Trader COBOL application, allowing us to invoke the Trader application from a EJB environment without modification to the original COBOL code. The design of the TraderBean was such that modifying it to access DB2 data instead of VSAM files requires only minimal changes to the bean itself. In this chapter we describe the steps necessary to create a TraderBackEnd interface class for use with both JDBC and SQLJ as a means for accessing DB2 data.

For information on using JDBC to dynamically access DB2 data from CICS, refer to 10.2, “Accessing DB2 using JDBC” on page 256. For information on using static SQL using SQLJ calls from CICS, refer to 10.3, “Accessing DB2 using SQLJ” on page 283.

The version of DB2 used in this scenario was DB2 UDB Server for OS/390 Version 6, which we will refer to as DB2 V6.

In the following sections we now provide a discussion of the important considerations when using JDBC and SQLJ from enterprise beans running in the CICS TS V2.1 EJB container.

CICS TS V2.1 JDBC/SQLJ support

CICS Java applications can now access DB2 data via the JDBC and SQLJ APIs. This applies to both CICS JVM applications (including enterprise beans) and CICS Java program objects (that is, programs bound using VisualAge for Java, Enterprise Edition for OS/390, Version 2). The JDBC API uses the dynamic SQL model; the SQLJ uses the static SQL model.

In a CICS environment, the DB2 JDBC driver is link-edited with the CICS DB2 language interface stub DSNCLI, therefore JDBC and SQLJ requests are converted by the JDBC driver into EXEC SQL requests and then routed into the existing CICS-DB2 Attachment Facility. All existing tuning and control parameters available to CICS DB2 applications can be used with CICS Java applications using JDBC and SQLJ.

Important: DB2 required APAR. The support for CICS TS V2.1 is supplied by DB2 V6 in APAR PQ44115 and on DB2 V7 via PQ45186.

You can find general information about JDBC at the Web site:

```
http://java.sun.com/products/jdbc/index.html
```

For information how to use JDBC to access DB2, refer to the Web site:

```
http://www.software.ibm.com/data/db2/os390/jdbc.html
```

JDBC 2.0 and J2EE

The Java 2 Platform, Enterprise Edition (J2EE) defines a standard for developing multi-tier enterprise applications. J2EE simplifies enterprise applications by basing them on standardized, modular components. The J2EE specification requires application servers to support a specific set of protocols and Java enterprise extensions, including JDBC 2.0, Enterprise JavaBeans 1.1, Java Servlets 2.2, Java Server Pages 1.1.

The JDBC API has been factored into two complementary components. The first component is the API that is core to the Java platform and comprises the updated contents of the *java.sql* package. The second component, termed the JDBC 2.0 Optional Package API, consists of a new package, *javax.sql*, which is the JDBC 2.0 Standard Extension API.

The JDBC 2.0 Optional Package API provides support for DataSources, connection pooling, distributed transactions and rowsets.The DataSource interface represents a particular Database Management System (DBMS) or some other data source interface and provides an alternative to the *DriverManager* class for making a connection to a data source. It makes the code more portable and easier to maintain. However, since CICS TS V2.1 only supports use of the DB2 JDBC 1.2 API it does not currently support the use of DataSources. It is planned that a future release of CICS TS will support use of JDBC 2.0 and hence use of DataSources from Enterprise JavaBeans.

Commit and rollback

JDBC / SQLJ applications are allowed to issue commit and rollback method calls, which will be converted into a EXEC CICS SYNCPOINT or EXEC CICS SYNCPOINT ROLLBACK command. Hence a JDBC or SQLJ commit results in the whole CICS unit of work being committed, not just the updates made to DB2. Committing work done independently of the rest of the CICS unit of work is not supported by CICS.

Autocommit causes a commit after each update to DB2 and when a connection is closed. Because a commit results in the whole unit of work being committed, the usage of autocommit in a CICS environment is discouraged, so the DB2 JDBC driver sets a default of autocommit(false) when running in a CICS environment.

10.1 Quick start — Invoking TraderBean

If you want to run our sample JDBC or SQLJ enterprise bean without following all the details specified in this chapter, use the steps below. All the source code and examples used in this book are available for download from `ftp://www.redbooks.ibm.com/redbooks/`, which is the redbooks FTP server. For further details, refer to Appendix C, “Using the additional material” on page 315.

1. Create a CICS TCPIPSERVICE, CORBASERVER, REQUESTMODEL and DJAR definition, if you have not already done so. For more details, refer to 6.3.3, “Deploying to CICS” on page 150.
2. Deploy the TraderBean to your CICS TS V2.1 region. For more information on how to do this, refer to 7.3, “Deploying the TraderBean to CICS” on page 191.
3. If you want to run the sample using **JDBC**, follow the steps below. If you want to run the **SQLJ** sample, proceed with step 4.
   a. Set up the DB2 database on OS/390 as described in 10.2.3, “Setting up the database” on page 274.
   b. Customize the JDBC runtime environment as described in 10.2.4, “Customizing the JDBC runtime environment” on page 276.
   c. Define a DB2 connection to your CICS region as described in 10.2.5, “Defining a CICS DB2 connection” on page 280.
   d. Grant the DB2 privileges to your CICS user ID as described in 10.2.6, “Granting privileges to the CICS user ID” on page 282.
4. If you want to run the sample using **SQLJ**, follow the steps below:
   a. Set up the DB2 database on OS/390 as described in 10.2.3, “Setting up the database” on page 274.
   b. Prepare the SQLJ application on OS/390 as described in 10.3.3, “Preparing the SQLJ program on OS/390” on page 295. This includes customizing the SQLJ serialized profile and binding the plan for the SQLJ application.

c. Modify the CICS DB2 connection as described in 10.3.4, “Modifying the CICS DB2 connection” on page 298.

d. Grant the required privileges to your CICS user ID as described in 10.3.5, “Granting privileges to the CICS user ID” on page 298.

e. Deploy the JAR file in your CICS system as described in “Refresh the DJAR in the CICS shelf” on page 273.

5. Use the supplied `runTest.cmd` file to invoke our stand-alone Java test application *TraderTest*. To set up TraderTest, simply do the following:

a. On your workstation, create a directory (for example `C:\itsotrader`) and copy the following sample files to this directory:

- `traderCLI.jar`
- `traderTest.jar`
- `runTest.cmd`

b. Ensure that you have a Java 2 runtime environment at version 1.3 or greater on your workstation installed. You can verify your version with the command:

```
java -version
```

c. Ensure that you have file `j2ee.jar` accessible on your workstation. If not, you can either obtain it either by installing Java 2 SDK, Enterprise Edition available from `http://java.sun.com` or when you install the CICS development deployment tool.

d. Invoke TraderTest using the `runtest.cmd` file. You will need to alter the input parameters as documented in the file. For further details and for an example of expected output with JDBC, refer to Example 10-10 on page 282, and for an example with SQLJ, refer to Example 10-13 on page 299.

10.2 Accessing DB2 using JDBC

The next sections describe the following steps, which were necessary to develop and test our session bean accessing DB2. These steps were:

1. Developing the JDBC application.
2. Deploying the session bean to CICS.
3. Setting up the database.
4. Customizing the JDBC runtime environment.
5. Defining the CICS DB2 connection.
6. Granting privileges to the CICS user ID.
7. Testing the enterprise bean.

10.2.1 Developing the JDBC application

This section describes how to write a session bean accessing DB2 using JDBC. In this context, we concentrate on describing the class that implements the interface TraderBackend. All the necessary steps to design the enterprise bean and implement the interface TraderBackend are already described in 7.2, “TraderBean development with VisualAge for Java” on page 174.

To implement the new class, we have to consider the following steps:

1. Designing the database layout

2. Adapting TraderBean for use of JDBC back-end class
3. Implementing TraderBackendDB2JDBC

Tip: *Setting up JDBC in VAJ*
If you want to utilize JDBC with VAJ, you need to add the JDBC driver that you will be using to your workspace classpath. When you install VAJ, SQL classes are installed and located in the package `java.sql`. To add the IBM UDB driver, select **Window -> Options**, select **Resources**, enter the fully qualified path of `db2java.zip`, and click OK.

Designing the database layout

To design the database layout, we have to look at the information the bean provides by means of its business methods. This is basically the companies used by the COBOL Trader application, as well as how many shares are held by each user for each company.

To store this information in DB2, we need two tables, *TRADER_COMPANY* and *TRADER_USER*. The first table, TRADER_COMPANY, contains the names of the companies and the quote specific information, such as the actual share price and the commission cost for trading shares. The TRADER_COMPANY table is illustrated in Table 10-1.

Table 10-1 TRADER_COMPANY table

Column name	Column type	Length	Nulls
c_name	character	20	no
c_sv_1d	decimal	7,2	no
c_sv_2d	decimal	7,2	no
c_sv_3d	decimal	7,2	no
c_sv_4d	decimal	7,2	no
c_sv_5d	decimal	7,2	no
c_sv_6d	decimal	7,2	no
c_sv_7d	decimal	7,2	no
c_sv_now	decimal	7,2	no
c_cc_buy	character	3	no
c_cc_sell	character	3	no

The column “c_name” uniquely identifies a particular company in our application and therefore serves as the primary key. The enterprise bean uses this table only to obtain data about the companies, which means that it performs read-only access on this table. All necessary data must be inserted into the table before the enterprise bean is started.

The second table, TRADER_USER, contains the current share holdings for each user. This table is illustrated in Table 10-2.

Table 10-2 TRADER_USER table

Column name	Column type	Length	Nulls
u_name	character	8	no
u_c_name	character	20	no

Column name	Column type	Length	Nulls
u_sn_held	integer		no

The combination of the columns “u_name” and “u_c_name” serves as the primary key. The enterprise bean uses this table to obtain the current number of company shares held by the user, as well as to update this number if the user trades shares.

Adapting TraderBean for use of JDBC back-end class

It did not take much work to modify our TraderBean for use with DB2. The only change necessary was to modify the method `loadClass()`. Our previous implementation of the `loadClass()` method, described in 9.2, “Adapting TraderBean to use JCICS” on page 236, takes into account only access with JCICS-COBOL, the CICS Connector, and JCICS-Java. The new class that uses JDBC to access DB2 we will name *TraderBackendDB2JDBC*. Therefore `loadClass()` needs to be modified in such a way that it can also load this new class. Our changes are illustrated in Figure 10-2.

```
private void loadClass(String type) throws Exception {
Class loadClass=null;
if( type.equalsIgnoreCase("JCICS-COBOL") == true ) {
  loadClass = Class.forName("itso.ejb390.trader.TraderBackendJcics");
}
else if( type.equalsIgnoreCase("CICSConnectorCCF") == true ) {
  loadClass = Class.forName("itso.ejb390.trader.TraderBackendCICSConnectorCCF");
}
else if( type.equalsIgnoreCase("JCICS-Java") == true ) {
  loadClass = Class.forName("itso.ejb390.trader.TraderBackendVsam");
}
else if( type.equalsIgnoreCase("DB2JDBC") == true ) {
  loadClass =
    Class.forName("itso.ejb390.trader.TraderBackendDB2JDBC");
}
else {
  throw new TraderException( "You specified unknown type " + type );
}
ivTraderBackend = (TraderBackend)loadClass.newInstance();
}
```

Figure 10-2 TraderBean.loadClass() loading TraderBackendDB2SQLJ

Implementing TraderBackendDB2JDBC

This section demonstrates how to implement class TraderBackendDB2JDBC, which accesses DB2 on OS/390.

Because the class must implement TraderBackend, it is necessary to implement the following methods:

- `ejbBackendCreate()`
- `ejbBackendRemove()`
- `ejbBackendPassivate()`
- `ejbBackendActivate()`
- `logon()`
- `getCompanies()`
- `getQuotes()`
- `buy()`
- `sell()`
- `logoff()`

It is also necessary to implement two further methods to manage the database connection.

- `openConnection()`
- `closeConnection()`

TraderBackendDB2JDBC class

The TraderBackendDB2JDBC class is an implementation of the interface class TraderBackend. Figure 10-3 shows the declaration of TraderBackendDB2JDBC.

```
import java.text.*;
import javax.naming.*;
import java.sql.*;

public class TraderBackendDB2JDBC implements TraderBackend {

  // fields to be stored if passivated
  private java.lang.String    ivUserID = "";
  private java.lang.String    ivJDBCURL = null;
  private java.sql.Connection ivCon = null;
}
```

Figure 10-3 Declaration of class TraderBackendDB2JDBC

The class defines three instance variables, which are used by some of the methods:

ivUserID	Holds the user ID that was used for the logon. The user ID is needed to clean up the database when the user logs off from the application.
ivJDBCURL	Holds the URL that is needed to identify a data source.
ivCon	Holds the connection to the data source when the bean is in the state *Method-Ready*.

TraderBackendDB2JDBC imports three packages. We use the *javax.naming* package to read the environment entries passed to the enterprise bean, the *java.sql* package to be able to perform the JDBC API calls, and the *java.text* package to format the data returned by the SQL queries.

Now that we have declared our instance variables, we are ready to implement the methods belonging to the JDBC back-end class.

TraderBackendDB2JDBC.openConnection()

The `openConnection()` method requests a connection instance from the DriverManager by specifying the location of the database as a URL. It then sets the autocommit property of the JDBC connection to false.

The implementation of method `openConnection()` is illustrated in Figure 10-4. The declaration of this method is private, because it is only used internally by the methods, which implements the back-end interface.

```
private void openConnection() throws SQLException
{
  ivCon = DriverManager.getConnection(ivJDBCURL);
  ivCon.setAutoCommit(false);
}
```

Figure 10-4 TraderBackendDB2JDBC.openConnection()

Note: Although the DB2 JDBC driver sets a default of autocommit (false) when running in the CICS environment, we also set the autocommit property to false in our code in case the method is ever invoked from a non-CICS environment on OS/390.

TraderBackendDB2JDBC.closeConnection()

The `closeConnection()` method closes the connection and frees all resources associated with the connection. Then it sets the instance variable `ivCon` to null to avoid serialization problems. Figure 10-5 shows the implementation of method `closeConnection()`.

```
private void closeConnection() throws SQLException
{
  ivCon.close();
  ivCon = null;
}
```

Figure 10-5 TraderBackendDB2JDBC.logoff()

TraderBackendDB2JDBC.ejbBackendCreate()

When the client invokes the `create()` method on the EJB home of the session bean, its life cycle begins. The container receives the create request, and invokes the `ejbCreate()` method of the instance that matches the client create request.

In our case, the `ejbCreate()` method of the TraderBean calls the `ejbBackendCreate()` method of the back-end class that matches to the type passed as parameter from the client to the `ejbCreate()` method. Thus the `ejbBackendCreate()` method is called only once to the begin of the life cycle of the TraderBean (Figure 10-6).

```
public void ejbBackendCreate() throws java.rmi.RemoteException
{
  String jdbcDriver = null;

  try {
    InitialContext initCtx  = new InitialContext();
    jdbcDriver = (String)initCtx.lookup("java:comp/env/jdbcDriver");
    ivJDBCURL = (String) initCtx.lookup("java:comp/env/jdbcURL");
    // register the DB2 driver
    if (jdbcDriver != null && jdbcDriver.length() != 0)
      { Class.forName(jdbcDriver); }
    openConnection();
  } catch (Exception ex) {
       throw new java.rmi.RemoteException(ex.getMessage());
  }
}
```

Figure 10-6 TraderBackendDB2JDBC.ejbBackendCreate()

The `ejbBackendCreate()` method uses the following environment properties.

jdbcDriver This property specifies the name of the appropriate JDBC driver.

jdbcURL This property specifies the URL that is needed to identify a data source.

In the EJB 1.1 specification, all deployed beans have an environment naming context that can be accessed using the JNDI API. This default JNDI context provides the bean with access to environment properties, which are defined in the deployment descriptor. Figure 10-6 shows how we use the default JNDI context to access the properties `jdbcDriver` and `jdbcURL`. Notice that we look up environment properties in *java:comp/env*, which is the location for all environment entries.

Due to the fact that you can specify the DB2 JDBC driver via the `jdbc.drivers` system properties in the CICS environment file `dfjjvmpr.props`, applications running in CICS do not need to load the driver themselves using the `Class.forName()`. Instead, the DriverManager class will load the required class for the application. For that reason, we only load the driver if the jdbcDriver property contains a real value. Finally, the `ejbBackendCreate()` method calls the `openConnection()` method to establish a connection to the database.

TraderBackendDB2JDBC.ejbBackendRemove()

When the client invokes the `remove()` method on the EJB home of the session bean, its life cycle ends. The container receives the remove request, and invokes the `ejbRemove()` method on the instance. At this time, the bean instance should perform any cleanup operations, such as closing open resources, including database connections.

In our case, the `ejbRemove()` method of the TraderBean calls the `ejbBackendRemove()` method of the back-end class. Thus the `ejbBackendRemove()` method is called only once at the end of the life cycle of the TraderBean, and calls the `closeConnection()` method to close the database connection, freeing all resources associated with that connection. This is illustrated in Figure 10-7.

```
public void ejbBackendRemove() throws java.rmi.RemoteException
{
  try {
    closeConnection();
  } catch (Exception ex) {
      throw new java.rmi.RemoteException(ex.getMessage());
  }
}
```

Figure 10-7 TraderBackendDB2JDBC.ejbBackendRemove()

TraderBackendDB2JDBC.ejbBackendPassivate()

During the lifetime of a stateful session bean, the container can passivate the bean instance to conserve resources. The instance fields are saved to storage, and the bean instance is evicted from memory. To alert the bean that it is to enter the Passivated state, the container invokes the `ejbPassivate()` method. At this time, the bean instance should close any open resources and set all nonserializable fields to null. This will prevent problems from occurring when the bean is serialized.

In our case, the `ejbPassivate()` method of the TraderBean calls the `ejbBackendPassivate()` method of the TraderBackendDB2JDBC class. Thus the `ejbBackendPassivate()` method is called if the container decides to passivate the bean (Figure 10-8).

```
public void ejbBackendPassivate() throws java.rmi.RemoteException
{
  try {
    closeConnection();
  } catch (Exception ex) {
      throw new java.rmi.RemoteException(ex.getMessage());
  }
}
```

Figure 10-8 TraderBackendDB2JDBC.ejbRemove()

The `ejbBackendPassivate()` method calls the `closeConnection()` method to close the database connection, freeing all resources associated with that connection. The implementation is identical to method `ejbBackendRemove()` method.

TraderBackendDB2JDBC.ejbBackendActivate()

When the client makes a request on an EJB object whose bean is passivated, the container activates the instance and invokes the *ejbActivate()* method, this is to allow the bean to open any resources needed. In our case, the `ejbActivate()` method of the TraderBean calls the `ejbBackendActivate()` method of the back-end class, which calls the `openConnection()` method to establish a new connection to the database (Figure 10-9).

```
public void ejbBackendActivate() throws java.rmi.RemoteException
{
  try {
    // open connection
    openConnection();
  } catch (Exception e) {
    throw new java.rmi.RemoteException(ex.getMessage());
  }
}
```

Figure 10-9 TraderBackendDB2JDBC.ejbBackendActivate()

TraderBackendDB2JDBC.logon()

The `logon()` method uses the supplied userid to query the database in order to display the list of companies (Figure 10-10).

```
public void logon(String userID, String password, String connectURL, String cicsServer)
throws Exception
{
   // save userID
1  ivUserID = userID;

2  Statement stmt = null;
   PreparedStatement pstmt = null;

   try {
      // create the statement (select the company names)
3     stmt = ivCon.createStatement();
      // create the prepared statement
      // (insert user, company, number of shares held into user table)
4     pstmt = ivCon.prepareStatement(sql_insertShareNumber);

5     // execute the query
      ResultSet result = stmt.executeQuery(sql_selectCompany);

      // set parameters
6     pstmt.setString(1, userID);

      // retrieve the result
7     while (result.next()) {
         pstmt.setString(2, result.getString(1));
         // execute the insert
         try {
8           pstmt.executeUpdate();
         } catch (SQLException ex) {
            if (ex.getErrorCode() != -803) throw ex;
         }
      }
   } catch (Exception ex) {
      System.out.println("Exception detected!");
      System.out.println("********* S T A C K T R A C E *********");
      ex.printStackTrace();
      throw ex;
   } finally {
9     if (stmt != null) stmt.close();
      if (pstmt != null) pstmt.close();
   }
}
```

Figure 10-10 TraderBackendDB2JDBC.logon()

The following list summarize the logic in method `logon()`.

- 1 Store the provided user ID in the class's instance variable.

  The user ID needs to be stored, since it is required to clean up the database when the user logs off from the application.

- 2 Define the Statement objects.

  Statement objects are used to execute SQL statements.

- 3 Create the statement.

  The `createStatement()` method creates the statement object.

- 4 Create the prepared statement.

  The `prepareStatement()` method creates a PreparedStatement object for sending parameterized SQL statements to the database. The SQL statement defined by sql_insertShareNumber is as follows:

  ```
  INSERT INTO itsoejb.trader_user(u_name,u_c_name,u_sn_held) VALUES(?,?,?)
  ```

- 5 Execute a query.

  The `executeQuery()` method executes an SQL query and generates a ResultSet instance. The SQL statement that is defined by the sql_selectCompany constant looks as follows:

  ```
  SELECT c_name FROM itsoejb.trader_company
  ```

- 6 Supply the values for the user ID and number of shares hold.

  The `set...()` methods supply the values to be used in place of the question mark placeholders to the prepared statement.

- 7 Retrieve the result and copy the company names to the CompaniesBean.

  The `next()` method on the ResultSet instance advances the iterator to successive rows.

- 8 Execute the update.

  The `executeUpdate()` method executes an SQL update. In our sample, a new row is inserted into the TRADER_USER table for each company.

  Note: We have to tolerate the SQL error **-803** (non-unique value). The reason is that an insert would result in duplicate values in the index column, because the application keeps all positive (>0) share holdings when the user logs off from the application.

- 9 Close the statements.

  The `close()` method closes the statements and frees all resources associated with the statements.

Assuming that a user named Steffen logs on to the application the first time, the table TRADER_USER would contain the entries shown in Table 10-3 after calling the `logon()` method.

Table 10-3 Content of TRADER_USER table after logon from Steffen

u_name	u_c_name	u_sn_held
Casey_Import_Export	Steffen	0
Class_and_Luget_Plc	Steffen	0
Headworth_Electrical	Steffen	0
IBM	Steffen	0

TraderBackendDB2JDBC.getCcompanies()

The `getCompanies()` method generates an SQL query to return the list of available companies; see Figure 10-11.

```
public CompaniesBean getCompanies() throws Exception
{
1 Statement stmt = null;
2 CompaniesBean companies = new CompaniesBean();

  try {
3   stmt = ivCon.createStatement();
4   ResultSet result = stmt.executeQuery(sql_selectCompany);
    while (result.next()) {
5     companies.addCompany(result.getString(1));
    }
  } catch (Exception ex) {
      ex.printStackTrace(); throw ex;
  } finally {
6     stmt.close();
  }
7 return companies;
}
```

Figure 10-11 TraderBackendDB2JDBC.getCompanies()

The following list summarizes the logic in method `getCompanies()`.

- 1 Define the Statement object.
- 2 Instantiate a CompaniesBean.
- 3 Create the statement.
- 4 Execute a query and generate a ResultSet instance.

  The SQL statement that is defined by the sql_selectCompany constant looks as follows:

  ```
  SELECT c_name FROM itsoejb.trader_company
  ```

- 5 Retrieve the result and copy the company names to the CompaniesBean.
- 6 Close the statement.
- 7 Return the CompaniesBean.

TraderBackendDB2JDBC.getQuotes()

The `getQuotes()` method is used to execute an SQL query to obtain the current value of a given user's share holding in the chosen company.

Figure 10-12 shows the implementation of method `getQuotes()`.

```
public QuotesBean getQuotes(String company, String userID) throws Exception
{
1 PreparedStatement pstmt = null;
2 QuotesBean quotes = new QuotesBean();
  try {
3   pstmt = ivCon.prepareStatement(sql_selectQuotes);
4   pstmt.setString(1, company);
5   ResultSet result = pstmt.executeQuery();
    double unitSharePrice = 0;
    DecimalFormat df = new DecimalFormat("00000.00");
    FieldPosition fp = new FieldPosition(1);
6   if (result.next()) {
      quotes.setUnitValue1Days(df.format(result.getDouble(1),
                                         new StringBuffer(),fp).toString());
      quotes.setUnitValue7Days(df.format(result.getDouble(7),
                                         new StringBuffer(),fp).toString());
      unitSharePrice = result.getDouble(8);
      quotes.setUnitSharePrice(df.format(unitSharePrice,
                                         new StringBuffer(),fp).toString());
      quotes.setCommissionCostSell(result.getString(9));
      quotes.setCommissionCostBuy(result.getString(10));
    }
7   pstmt.close();
8   pstmt = ivCon.prepareStatement(sql_selectShareNumber);
9   pstmt.setString(1, userID);
    pstmt.setString(2, company);
10  result = pstmt.executeQuery();
11  if (result.next()) {
      int numberOfSharesHeld = result.getInt(1);
      quotes.setNumberOfShares( new DecimalFormat("0000").format
                (numberOfSharesHeld,new StringBuffer(),fp).toString());
      double totalShareValue = unitSharePrice * numberOfSharesHeld;
      quotes.setTotalShareValue(df.format(totalShareValue,
                                          new StringBuffer(),fp).toString());
    }
  } catch (Exception ex) { ex.printStackTrace(); throw ex; }
 finally {
    pstmt.close();
  }
12return quotes;
}
```

Figure 10-12 TraderBackendDB2JDBC.getQuotes()

The following list summarizes the logic in method `getQuotes()`.

- 1 Define the prepared Statement object.
- 2 Instantiate a QuotesBean.

  QuotesBean is a class which is used to hold all quote specific information returned by the SQL statement.
- 3 Create the prepared statement.

  The SQL statement defined by sql_selectQuotes constant is as follows:

  ```
  SELECT c_sv_1d,...,c_sv_7d,c_sv_now,c_cc_sell,c_cc_buy
    FROM itsoejb.trader_company WHERE c_name = ?
  ```
- 4 Supply the value for the company name.

- 5 Execute a query and generate a ResultSet instance.
- 6 Retrieve the result and copy the information to the QuotesBean.

  We have used the class DecimalFormat to format the decimal numbers to the same format that is used by the COBOL Trader application.

Note: It is also possible to use `getBigDecimal()` instead of `getDouble(). However`, due to the fact that CICS TS V2.1 only supports use of the DB2 JDBC 1.2 API, you cannot use `getBigDecimal(int columnIndex)`, as it is only supported by JDBC 2.0. Instead, you have to use `getBigDecimal(int columnIndex, int scale)`, which is deprecated in JDBC 2.0 and therefore leads to a warning when using it in VAJ V3.5.

- 7 Close the prepared statement.
- 8 Create the prepared statement.

  The SQL statement defined by sql_selectShareNumber constant looks as follows:

  ```
  SELECT u_sn_held FROM itsoejb.trader_user WHERE u_name=? AND u_c_name=?
  ```

- 9 Supply the values for the user ID and the company name.
- 10 Execute the SQL query and generate a ResultSet instance.
- 11 Retrieve the result and copy the information to the QuotesBean.

  The total share value is calculated from the number of shares held by the user multiplied by the current share value.
- 12 Return the QuotesBean.

TraderBackendDB2JDBC.buy()

The `buy()` method uses the company name, the user ID, and the number of shares to execute an SQL statement in order to perform a “buy shares” operation (see Figure 10-13).

```
public void buy(String company, String userID, int numberOfShares) throws Exception
{
1  if (numberOfShares > 9999) return;
2  PreparedStatement pstmt = null;
   try {
3   pstmt = ivCon.prepareStatement(sql_selectShareNumber);
4   pstmt.setString(1, userID);
    pstmt.setString(2, company);
5   ResultSet result = pstmt.executeQuery();
    int numberOfSharesHeld = 0;
6   if (result.next()) numberOfSharesHeld = result.getInt(1);
    numberOfSharesHeld += numberOfShares;
7   if (numberOfSharesHeld > 9999) return;
8   pstmt.close();
9   pstmt = ivCon.prepareStatement(sql_updateShareNumber);
10  pstmt.setInt(1, numberOfSharesHeld);
    pstmt.setString(2, userID);
    pstmt.setString(3, company);
    pstmt.executeUpdate();
  } catch (Exception ex) { ex.printStackTrace(); throw ex; }
  finally {
11    pstmt.close();
  }
}
```

Figure 10-13 TraderBackendDB2JDBC.buy()

The following list summarizes the logic in method `buy()`.

- 1 Verify that the number of shares to buy does not exceed the maximum.

  The maximum number of shares a user can buy is 9999 and is defined by the COBOL Trader application.

- 2 Define the prepared Statement object.
- 3 Create the prepared statement.

  The SQL statement defined by sql_selectShareNumber constant looks as follows:

  ```
  SELECT u_sn_held FROM itsoejb.trader_user WHERE u_name=? AND u_c_name=?
  ```

- 4 Supply the values for the user ID and the company name.
- 5 Execute the SQL query and generate a ResultSet instance.
- 6 Retrieve the result, copy the number of shares held by the user to a local variable and calculate the new number of shares held.
- 7 Verify that the number of shares does not exceed the maximum.

  The maximum number of shares a user can hold is 9999 and is defined by the COBOL Trader application.

- 8 Close the prepared statement.
- 9 Create the prepared statement.

  The SQL statement defined by sql_updateShareNumber constant is as follows.

  ```
  UPDATE itsoejb.trader_user SET u_sn_held=? WHERE u_name=? AND u_c_name=?
  ```

- 10 Supply the values for the user ID, the company name, and the number of shares, and execute the update.
- 11 Close the prepared statement.

Assuming that the user Steffen buys 250 shares of the company IBM, the table TRADER_USER would contain the entries shown in Table 10-4 after calling the `buy()` method.

Table 10-4 Content of TRADER_USER table after user Steffen has bought 250 shares

u_name	u_c_name	u_sn_held
Casey_Import_Export	Steffen	0
Class_and_Luget_Plc	Steffen	0
Headworth_Electrical	Steffen	0
IBM	Steffen	250

TraderBackendDB2JDBC.sell()

The `sell()` method is very similar to the method `buy()`. The only difference is how it calculates the new number of shares held by the customer, as shown in Figure 10-14.

```
public void sell(String company, String userID, int numberOfShares) throws Exception
{
  if (numberOfShares > 9999) return;
  PreparedStatement pstmt = null;
  try {
    pstmt = ivCon.prepareStatement(sql_selectShareNumber);
    pstmt.setString(1, userID);
    pstmt.setString(2, company);
    ResultSet result = pstmt.executeQuery();
    int numberOfSharesHeld = 0;
    if (result.next()) numberOfSharesHeld = result.getInt(1);
    if (numberOfShares > numberOfSharesHeld) return;
    numberOfSharesHeld -= numberOfShares;
    pstmt.close();
    pstmt = ivCon.prepareStatement(sql_updateShareNumber);
    pstmt.setInt(1, numberOfSharesHeld);
    pstmt.setString(2, userID);
    pstmt.setString(3, company);
    pstmt.executeUpdate();
  } catch (Exception ex) {
      ex.printStackTrace(); throw ex;
  } finally {
      pstmt.close();
  }
}
```

Figure 10-14 TraderBackendDB2JDBC.sell()

Assuming that the user Steffen sells 125 shares of the company IBM, the table TRADER_USER would contain the entries shown in Table 10-5 after calling the `sell()` method.

Table 10-5 Content of TRADER_USER table after Steffen has sold 125 shares

u_name	u_c_name	u_sn_held
Casey_Import_Export	Steffen	0
Class_and_Luget_Plc	Steffen	0
Headworth_Electrical	Steffen	0
IBM	Steffen	125

TraderBackendDB2JDBC.logoff()

This method is driven when a user logs off, and delete entries from the database for which the number of shares held is zero. Figure 10-15 shows the implementation of method `logoff()`.

```
public void logoff() throws Exception
{
1 PreparedStatement pstmt = null;
  try {
2   pstmt = ivCon.prepareStatement(sql_deleteShareNumber);
3   pstmt.setString(1, ivUserID);
4   pstmt.executeUpdate();
  } catch (Exception ex) {
      System.out.println("Exception detected!");
      System.out.println("********* S T A C K T R A C E *********");
      ex.printStackTrace();
      throw ex;
  } finally {
5     if (pstmt != null) pstmt.close();
   }
}
```

Figure 10-15 TraderBackendDB2JDBC.logoff()

The following list summarizes the logic in method `logoff()`.

- 1 Define the prepared Statement object.
- 2 Create the prepared statement.

  The SQL statement defined by sql_deleteShareNumber constant looks as follows:

  ```
  DELETE FROM itsoejb.trader_user WHERE u_name=? AND u_sn_held=?
  ```

- 3 Supply the values for the user ID and the number of shares held. The application keeps all positive share holdings of a user when he logs off, which means that only those entries are deleted for which the number of shares held is equal zero.
- 4 Execute the SQL update.
- 5 Close the prepared statement.

After the user Steffen has logged off the application, the table TRADER_USER would contain the entry shown in Table 10-6.

Table 10-6 Content of TRADER_USER table after Steffen has logged off

u_name	u_c_name	u_sn_held
IBM	Steffen	125

10.2.2 Deploying the enterprise bean to CICS

This section describes how we deployed TraderBean and its related classes to our CICS TS V2.1 region. To do this, it is necessary to perform the following steps.

1. Export the enterprise bean and its related classes.
2. Convert the exported file to a DJAR file.
3. Send the DJAR file to OS/390.
4. Refresh the DJAR in the CICS region.

Export the enterprise bean and its related classes

This section shows how we exported the enterprise bean and its related classes.

1. Within VisualAge for Java,select the **EJB** tab to view the EJB groups. Select group **ITSOEJB390** and click the right mouse button.
2. Select **Export -> EJB JAR** to open the *Export to an EJB JAR File SmartGuide*.
3. You should see that the Trader bean and three additional classes are selected by default. Click **Select referenced types and resources** to ensure that VAJ now also selects classes which are referenced by the enterprise bean. This will cause *CompaniesBean, QuotesBean, TraderBackend,* and *TraderException* to also be selected. But VAJ has not selected the classes which implement the different back-ends.
4. In addition, select the following classes which implement the back-ends:
   - CompaniesBean
   - CompanyKeyRecord
   - CompanyKeyRecordBeanInfo
   - CompanyKeyRecordType
   - CompanyRecord
   - CompanyRecordType
   - CustomerKeyRecord
   - CustomerKeyRecordBeanInfo
   - CustomerKeyRecordType
   - CustomerRecord
   - CustomerRecordBeanInfo
   - CustomerRecordType
   - TraderBackendCICSConnectorCCF
   - TraderBackendDB2JDBC
   - TraderBackendJcics
   - TraderBackendVsam
   - TraderCommand
   - TraderCommandBeanInfo
   - TraderRecord
   - TraderRecordBeanInfo
   - TraderRecordType

   Now click **OK** to close the window.
5. In the export window, you should now see that 1 bean and 28 classes are selected.
6. Specify path and file name for the JAR file. We used `C:\itsotrader\traderForCICS.jar`. Your window should look as shown in Figure 10-16.
7. Click **Finish** to export the classes to file `traderForCICS.jar`.

Tip: If you receive a message from VAJ stating that *the file is not a zip file, or it is corrupted*, you should close the CICS Java development tool, or delete the output JAR file.

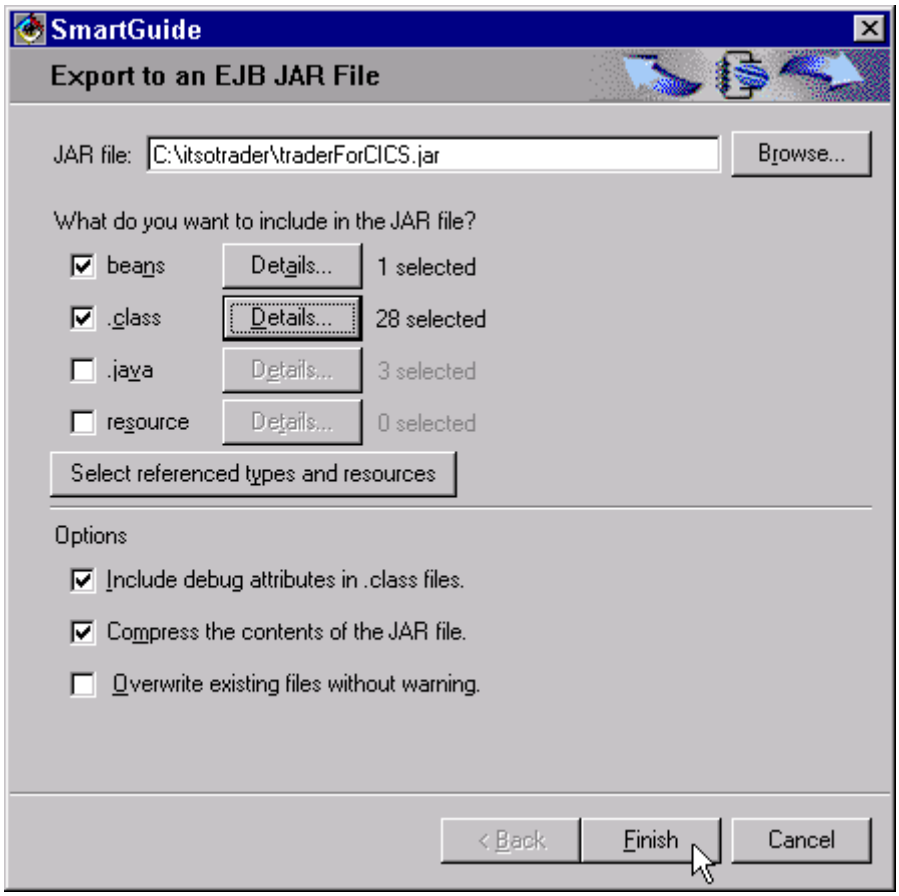


Figure 10-16 Exporting TraderBean for CICS (including TraderBackendDB2JDBC)

Convert the exported file to a DJAR file

The next step is to convert the exported JAR file to a deployed JAR file, and to edit the deployment descriptor. We used the CICS JAR development tool for this purpose, for further details on using this tool, refer to 6.3.2, “Generating a CICS deployed JAR file” on page 148.

- Start the CICS JAR development tool.
- Click **File -> Load** and enter the path and file name of the JAR file you have exported from VisualAge, which is in our case `c:\itsotrader\traderForCICS.jar`.
- Click **Open** to load the JAR file into the tool.
- Now you should see *Trader* below Current Enterprise Beans. Select Trader.
- Click **Edit** and then select the **Environment** tab. Now enter the two environment properties which are used by the `ejbCreate()` method as described in “TraderBackendDB2JDBC.ejbBackendCreate()” on page 260. To do so, perform the following steps, the final results of which are shown in Figure 10-17.
  - `Enter jdbcDriver` in the **Name:** field
  - select `java.lang.String` within the **Type:** field
  - Click the **Set** button
  - Enter `jdbcURL` in the **Name:** field
  - Enter `jdbc:db2os390sqlj:` in the **Value:** field
  - select `java.lang.String` within the **Type:** field
  - Click the **Set** button.

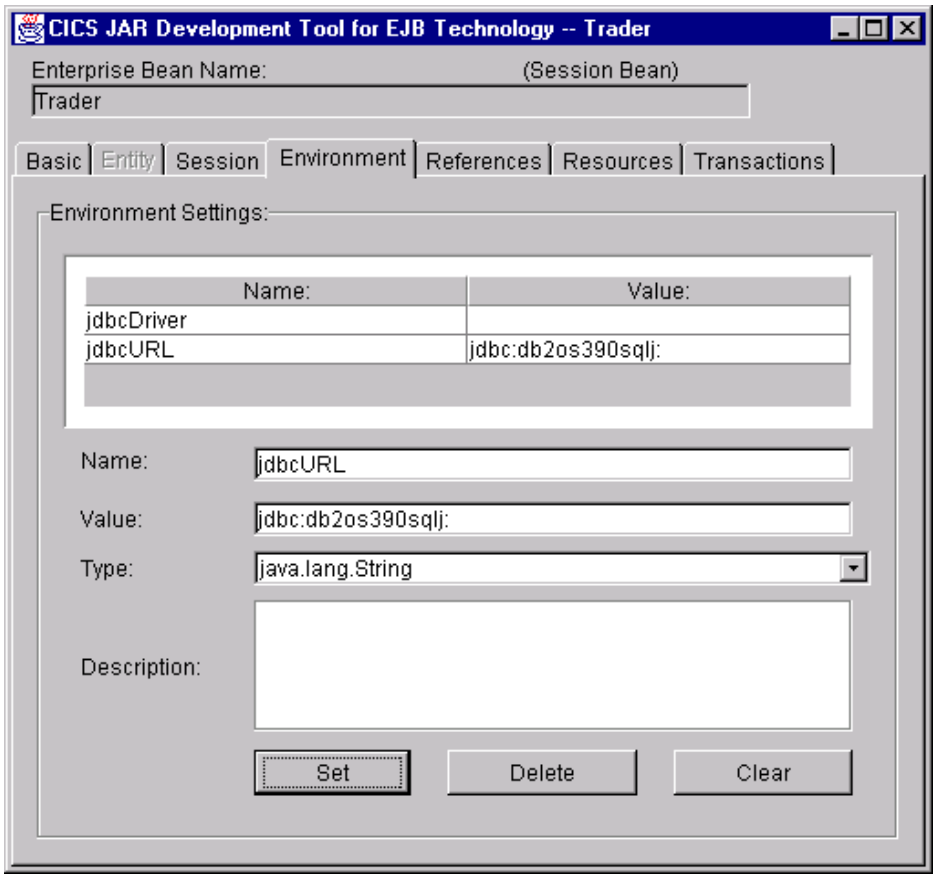


Figure 10-17 CICS JAR development tool, environment properties for DB2

- Close the window to return to the main CICS JAR development tool window.
- Click **File -> Generate** to generate the DJAR.
- You are asked to save your changes to a JAR file, click **Save**.
- You are asked to remove old EJB1.0 information, click **Remove**.
- Now you can specify an output EJB JAR file, use the tool's default name, which in our case is `c:\itsotrader\traderForCICS_GEN.jar`.
- Click **Generate**.

After a short time the tool generates the deployed JAR file `traderForCICS_GEN.jar`.

Send the deployed JAR file to OS/390

The next step is to transfer the deployed JAR file to OS/390. We created the HFS directory `/u/cicsts21/djars` on our OS/390 system and then, using FTP, we transferred the deployed JAR file `traderForCICS_GEN.jar` in binary mode to this directory.

Refresh the DJAR in the CICS shelf

The updated JAR file now has to be refreshed in the CICS region. In this sample we assume that you have already defined the DJAR to CICS as described in 7.3.4, "Defining the DJAR in the CICS system" on page 193. Therefore it is only necessary to discard the existing DJAR and re-install as follows:

```
CEMT DISCARD DJAR(TRADER)
CEDA INSTALL GR(ITSOEJB) DJAR(TRADER)
```

10.2.3 Setting up the database

In this section we explain how to create the database, the layout of which is described in "Designing the database layout" on page 257. The following objects need to be defined:

1. Define a DB2 database.
2. Define a DB2 table space.
3. Define DB2 tables.
4. Define DB2 indexes.
5. Load the data into the DB2 tables.

We will show how this is done in the next sections.

Define a DB2 database

In DB2, a database is a set of DB2 structures, such as a table and index spaces, and the data and indexes contained within them. When you define a DB2 database, you give a name to an eventual collection of tables and associated indexes, as well as to the table spaces in which they reside. In this context, a DB2 database is only an organizational entity.

Also, *database* has a different meaning for DB2 applications running on OS/390 compared to other environments. In a JDBC/SQLJ application you must be connected to a data source before you can execute SQL statements. A data source on OS/390 is a DB2 subsystem. In other environments, such as Windows NT, a data source is usually referred to as a database.

Example 10-1 shows the SQL statement we have used to define the DB2 database named TRADERDB, specifying DSN8G610 as the storage group to be used and BP0 as the default buffer pool. A DB2 storage group is a set of volumes on direct access storage devices, and a buffer pool is an area of virtual storage in which DB2 temporarily stores pages of table spaces or indexes.

Example 10-1 Create database statement for TRADERDB

```
CREATE DATABASE TRADERDB
  STOGROUP DSN8G610
  BUFFERPOOL BP0
  CCSID
EBCDIC;
```

Define a DB2 table space

A table space is one or more data sets in which one or more tables are stored. A table space can consist of a number of VSAM data sets. We used a simple table space using a primary space allocation of 20 KB, as shown in Example 10-2.

Example 10-2 Create tablespace statement for TRADERTS

```
CREATE TABLESPACE TRADERTS
   IN TRADERDB
   USING STOGROUP DSN8G610
             PRIQTY 20
             SECQTY 20
             ERASE NO
   LOCKSIZE ANY LOCKMAX SYSTEM
   BUFFERPOOL BP0
   CLOSE NO
   CCSID
EBCDIC;
```

Define DB2 tables

All data in a DB2 database is presented in tables, collections of rows all having the same columns. So when you create a table in DB2, you define an ordered set of columns.

Example 10-3 shows the SQL statements we have used to create the tables TRADER_COMPANY and TRADER_USER.

Note: After creation both tables are marked as unavailable until their primary indexes are explicitly created.

Example 10-3 Create table statements for TRADER_COMPANY and TRADER_USER

```
CREATE TABLE ITSOEJB.TRADER_COMPANY
   (  C_NAME    CHAR(20)       NOT NULL,
      C_SV_1D   DECIMAL(7,2)   NOT NULL,
      C_SV_2D   DECIMAL(7,2)   NOT NULL,
      C_SV_3D   DECIMAL(7,2)   NOT NULL,
      C_SV_4D   DECIMAL(7,2)   NOT NULL,
      C_SV_5D   DECIMAL(7,2)   NOT NULL,
      C_SV_6D   DECIMAL(7,2)   NOT NULL,
      C_SV_7D   DECIMAL(7,2)   NOT NULL,
      C_SV_NOW  DECIMAL(7,2)   NOT NULL,
      C_CC_SELL CHAR(3)        NOT NULL,
      C_CC_BUY  CHAR(3)        NOT NULL,
      PRIMARY KEY(C_NAME)
   )
IN TRADERDB.TRADERTS;

CREATE TABLE ITSOEJB.TRADER_USER
   ( U_NAME    CHAR(8)        NOT NULL,
     U_C_NAME  CHAR(20)       NOT NULL,
     U_SN_HELD INTEGER        NOT NULL,
     PRIMARY KEY(U_NAME, U_C_NAME)
   )
IN TRADERDB.TRADERTS;
```

Define DB2 indexes

An index is an ordered set of pointers to the data in DB2 table. The index is stored separately from the table. Each index is based on the values of data in one or more columns of a table. In DB2, indexes are used to ensure the uniqueness of the primary key and to improve performance. In most cases, access to data is faster with an index. A table with a unique index cannot have rows with identical keys. Example 10-4 shows the statements we have used to create the indexes.

Example 10-4 Create index statements for TRADER_COMPANY_PK and TRADER_USER_PK

```
CREATE UNIQUE INDEX ITSOEJB.TRADER_COMPANY_PK
               ON ITSOEJB.TRADER_COMPANY
                    (C_NAME   ASC)
               USING STOGROUP DSN8G610
                    PRIQTY 12
                    ERASE NO
               BUFFERPOOL BP0
               CLOSE NO;

CREATE UNIQUE INDEX ITSOEJB.TRADER_USER_PK
               ON ITSOEJB.TRADER_USER
                    (U_NAME ASC, U_C_NAME ASC)
               USING STOGROUP DSN8G610
```

```
                    PRIQTY 12
                    ERASE NO
                BUFFERPOOL BP0
                CLOSE
NO;
```

Load the data into the DB2 tables

You can load data into DB2 tables by using the DB2 LOAD utility or SQL INSERT statement. You will probably load most of your tables using the LOAD utility. We have used the SQL INSERT statement, because it was only necessary to fill the table TRADER_COMPANY with four rows. Example 10-5 shows the SQL INSERT statement to load the data for the company IBM into the TRADER_COMPANY table.

Example 10-5 SQL INSERT statement to load data into TRADER_COMPANY

```
INSERT INTO ITSOEJB.TRADER_COMPANY
  ( C_NAME, C_SV_1D, C_SV_2D, C_SV_3D, C_SV_4D, C_SV_5D,
    C_SV_6D, C_SV_7D, C_SV_NOW, C_CC_SELL, C_CC_BUY)
  VALUES ( 'IBM', 163.0, 162.0, 160.0, 161.0, 159.0,
           156.0,157.0,163.0, '015','010'
);
```

10.2.4 Customizing the JDBC runtime environment

In this section we describe the steps required to customize the JDBC runtime environment. Note that you have to perform these steps only once.

1. Customize the cursor properties file.
2. Create a JDBC profile and make it accessible.
3. Bind the DBRMs.
4. Set environment variables for CICS.

We assume that you have already installed the DB2 JDBC and SQLJ. Also, we concentrate on those steps which are important for running JDBC and SQLJ application for a CICS environment. Refer to the DB2 manual *Application Programming Guide and Reference for Java*, SC26-9018, for more information on JDBC and SQLJ administration.

Customize the cursor property file

The cursor properties file describes the DB2 cursors that the SQLJ/JDBC driver uses to process JDBC result sets. You can customize the cursor property file to modify the number of DB2 cursors available for JDBC and to control cursor names.

Choose the number of cursors for JDBC result sets

The default cursor properties file, `db2jdbc.cursors`, defines 125 cursors with hold and 125 cursors without hold. This number of cursors is too large for CICS applications, and it results in a JDBC profile size that is large enough to degrade performance. Specifying five cursors with hold and five cursors without hold should be should be adequate for most CICS applications.

Example 10-6 shows the cursor property file we have used to create a JDBC profile appropriate for CICS applications. As you can see, we have specified five cursors with hold and five cursors without hold.

Example 10-6 Cursor property file

```
cursor=DB2OS390NOHOLD001:nohold
cursor=DB2OS390NOHOLD002:nohold
cursor=DB2OS390NOHOLD003:nohold
cursor=DB2OS390NOHOLD004:nohold
cursor=DB2OS390NOHOLD005:nohold
cursor=DB2OS390HOLD001:hold
cursor=DB2OS390HOLD002:hold
cursor=DB2OS390HOLD003:hold
cursor=DB2OS390HOLD004:hold
cursor=DB2OS390HOLD005:hold
```

Create a JDBC profile and make it accessible

To create a JDBC profile, you have to execute the db2genJDBC utility. The JDBC profile contains the JDBC program properties used at runtime.

Choose parameter values for the db2genJDBC utility

The default value for the statement parameters used by the db2genJDBC utility is not appropriate for CICS applications, because it generates a large JDBC profile. For VAJ SQLJ and JDBC applications that run in a CICS environment, large JDBC profiles can degrade performance.

Choose a value for the statements parameter that is lower than the default of 150. The default value produces more sections than are necessary for typical CICS applications. A larger number of sections results in a larger JDBC profile size. A value of 10 should be adequate for most CICS applications.

Execute the db2genJDBC utility

The shell script we used to generate a JDBC profile is illustrated in Example 10-7.

Example 10-7 USS shell script to execute db2genJDBC utility

```
# ------------------------------------------------------------------
# Shell script to run the DB2 db2genJdBC utility from USS.
#
# Modify the following to match your DB2/390 installation directory:
DB2_HOME=/service_db2v6/usr/lpp/db2/db2610
#------------------------------------------------------------------
CLASSPATH=$CLASSPATH:$DB2_HOME/classes/db2sqljclasses.zip
export CLASSPATH
export LIBPATH=$LIBPATH:$DB2_HOME/lib/
export DB2SQLJPROPERTIES=./mydb2sqljjdbc.properties
$DB2_HOME/bin/db2genJDBC -statements=10 -cursors=./db2jdbc.cursors
```

The `db2genJDBC` utility creates four DBRMs and a JDBC profile.

Note: We use the environment variable `DB2SQLJPROPERTIES` to specify the runtime properties file for the DB2 for OS/390 SQLJ/JDBC driver. The runtime properties file lets you specify program preparation and runtime options that the DB2 for OS/390 SQLJ/JDBC driver uses. We used the runtime properties file only to specify the name of the partitioned data set into which DBRMs are placed. Therefore, our properties file contains one entry.

```
DB2SQLJDBRMLIB=CICSRS3.ITSO.DBRMLIB
```

Make the JDBC profile accessible

The JDBC profile name is *program name_JDBCProfile.ser*, where *program name* is the name you have specified using the `-pgmname` option of the `db2genJDBC` utility. Because we did not specify this option, the default *DSNJDBC* was used. Therefore, the name of the generated profile is *DSNJDBC_JDBCProfile.ser*.

You have to make sure the JDBC profile is accessible in a directory specified in the CLASSPATH environment variable. For application running in CICS, the CLASSPATH is specified using the CLASSPATH setting in the JVM profile. In “DFHJVM and JVM profiles” on page 73 we describe how we tailored the default profile provided by CICS to match our installation. In that profile, the CLASSPATH is defined as follows:

```
CLASSPATH=/usr/lpp/cicsts/cicsts21/lib
```

In order to make the JDBC profile available, we copied the JDBC profile to this path.

Bind the DBRMs

The db2genJDBC utility also creates four DBRMs, one for each DB2 isolation level. The name of the DBRMs are *program name*<*x*>, where <*x*> is a number between 1 and 4. In our case, the names of the generated profiles are DSNJDBC1, DSNJDBC2, DSNJDBC3 and DSNJDBC4. By default, the DBRMs are written to the dataset DBRMLIB.DATA.

You must bind the DBRMs into packages and include them into a plan that the JDBC application is able to access it at runtime.

DB2 provides a sample job in DSN610.SDSNSAMP.DSNTJJCL. Our bind job after tailoring the DB2 sample job is shown in Example 10-8.

Example 10-8 JCL to bind the JDBC DBRMs

```
//         CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1),
//         NOTIFY=&SYSUID,REGION=0M
//JOBLIB   DD  DISP=SHR,DSN=DSN610.SDSNLOAD
//BINDJDBC EXEC PGM=IKJEFT01,DYNAMNBR=20
//DBRMLIB  DD DISP=SHR,DSN=CICSRS3.ITSO.DBRMLIB
//SYSTSPRT DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
//SYSTSIN  DD  *
  DSN SYSTEM(DBZ1)

   BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC1) ISOLATION(UR)
   BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC2) ISOLATION(CS)
   BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC3) ISOLATION(RS)
   BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC4) ISOLATION(RR)
   BIND PLAN(DSNJDBC) -
             PKLIST(DSNJDBC.DSNJDBC1,    -
                    DSNJDBC.DSNJDBC2,    -
                    DSNJDBC.DSNJDBC3,    -
                    DSNJDBC.DSNJDBC4) RETAIN

END
/*
```

Note: For JDBC and SQLJ applications, and mixed JDBC and SQLJ applications, the DB2 JDBC driver uses information from the JDBC profile to set the name of the DBRM in the parameter list of the first EXEC SQL statement executed. The first SQL issued will always have the DBRM name set to the JDBC base program with the default isolation level appended, that is, DSNJDBC2 by default; or if, for example, the JDBC profile was generated using pgmname=OTHER, the DBRM name will be OTHER2. Because we do not specify the `-pgmname` option when generating the JDBC profile, the default DBRM used was DSNJDBC2, which is the DBRM for the isolation level cursor stability. In CICS, it is recommended to use cursor stability unless there is a specific need for using repeatable read. Cursor stability is recommended to allow a high level of concurrency and to reduce the risk of deadlocks.

Setting CICS parameters

It is necessary to modify several CICS parameters in order to use JDBC and SQLJ programs within CICS. These are described in the following sections.

STEPLIB

The CICS DB2 Attachment facility has to load the DB2 program request handler, DSNAPRH. To do this, the DB2 library SDSNLOAD should be placed in the MVS linklist, or added to the STEPLIB concatenation of your CICS job. Example 10-9 shows the modifications we made to our CICS startup procedure to enable DB2 support.

Example 10-9 JCL to start CICS region with DB2 support

```
//CICTSEJB PROC START='INITIAL',REG='0M',OUTC='*'
// COMMAND 'V NET,ACT,ID=APCPJA5,ALL'
//CICS61O  EXEC PGM=DFHSIP,REGION=&REG,TIME=144O,
//             PARM=('START=&START','SYSIN')
//STEPLIB  DD  DSN=CICSTS21.CICS.SDFHAUTH,DISP=SHR
//         DD  DSN=DSN61O.SDSNLOAD,DISP=SHR
//SYSABEND DD  SYSOUT=&OUTC
//SYSIN    DD  DSN=CICSSYSF.CICSTS21.SYSIN(PJA5SIT),DISP=SHR
//DFHRPL   DD  DSN=CICSSYSF.APPL61.LOADLIB,DISP=SHR
//         DD  DSN=CICSTS21.CICS.SDFHLOAD,DISP=SHR
//         DD  DSN=CEE.SCEECICS,DISP=SHR
//         DD  DSN=CEE.SCEERUN,DISP=SHR
//DFHCXRF  DD  SYSOUT=&OUTC
//DFHAUXT  DD  DISP=SHR,DCB=BUFNO=5,
//             DSN=CICSSYSF.CICS61O.PJA5.DFHAUXT
//DFHBUXT  DD  DISP=SHR,DCB=BUFNO=5,
//             DSN=CICSSYSF.CICS61O.PJA5.DFHBUXT
DFHDMPA  DD  DSN=CICSSYSF.CICS61O.PJA5.DFHDMPA,DISP=SHR
DFHDMPB  DD  DSN=CICSSYSF.CICS61O.PJA5.DFHDMPB,DISP=SHR
DFHINTRA DD  DSN=CICSSYSF.CICS61O.PJA5.DFHINTRA,DISP=SHR
DFHTEMP  DD  DSN=CICSSYSF.CICS61O.PJA5.DFHTEMP,DISP=SHR
DFHGCD   DD  DSN=CICSSYSF.CICS61O.PJA5.DFHGCD,DISP=SHR
DFHLCD   DD  DSN=CICSSYSF.CICS61O.PJA5.DFHLCD,DISP=SHR
DFHCSD   DD  DSN=CICSSYSF.CICSTS21.DFHCSD,DISP=SHR
DFHJVM   DD  DSN=CICSSYSF.CICS61O.DFHJVM,DISP=SHR
DFHEJDIR DD  DSN=CICSSYSF.CICS61O.PJA5.DFHEJDIR,DISP=SHR
DFHEJOS  DD  DSN=CICSSYSF.CICS61O.PJA5.DFHEJOS,DISP=SHR
DFHADJM  DD  DSN=CICSSYSF.CICS61O.PJA5.DFHADJM,DISP=SHR
```

Environment variables

To enable JDBC/SQLJ support, the following environment variables need to be set in the CICS JVM profile. The profile we used was CICSSYSF.CICS610.DFHJVM (DFHJVMPR). For more details on setting JVM profiles, refer to “DFHJVM and JVM profiles” on page 73.

- **LIBPATH**

  The DB2 for OS/390 SQLJ/JDBC driver contains several dynamic load libraries(DLLs). Modify the LIBPATH to include the directory that contains these DLLs. In our case, SQLJ and JDBC were installed in the HFS directory `/service_db2v6/usr/lpp/db2/db2610`, so we have added the following line to the LIBPATH concatenation of the CICS JVM profile:

  ```
  /service_db2v6/usr/lpp/db2/db2610/lib/
  ```

- **TMSUFFIX**

  TMSUFFIX is used to control the trusted middle classpath for the persistent reusable JVM. The DB2 zip operation archives `db2sqljruntime.zip`, and `db2jdbcruntime.zip` contains all of the classes necessary to run JDBC and SQLJ programs. We added the following lines to the TMSUFFIX concatenation of our JVM profile:

  ```
  /service_db2v6/usr/lpp/db2/db2610/classes/db2sqljruntime.zip:\
  /service_db2v6/usr/lpp/db2/db2610/classes/db2jdbcruntime.zip
  ```

- **DB2SQLJPROPERTIES**

  DB2SQLJPROPERTIES specifies the fully-qualified name of the runtime properties file for the DB2 for OS/390 SQLJ/JDBC driver. The runtime properties file contains various entries of the form parameter=value that specify program preparation and runtime options that the DB2 for OS/390 SQLJ/JDBC driver uses. Most properties in the runtime properties file are not used in a CICS environment, and we did not use this file in our CICS region.

10.2.5 Defining a CICS DB2 connection

A CICS attachment facility is provided with CICS that allows you to operate DB2 with CICS. The connection between CICS and DB2 can be created or terminated at any time, and CICS and DB2 can be started and stopped independently. You also have the option of CICS automatically connecting and reconnecting to DB2. The DB2 system can be shared by several CICS systems, but each CICS system can be connected to only one DB2 subsystem at any one time.

You use a DB2CONN definition to define the global attributes of the connection to be established between CICS and DB2 as well as the attributes of pool threads and command threads to be used with the connection.

Defining a DB2 connection

Several ways exists to define a DB2 connection in CICS. We used the CEDA transaction to define a DB2 connection. From a CICS terminal, we used the command `CEDA DEFINE DB2CONN(DB2CON) GROUP(ITSOEJB)`. We then specified the DB2 connection definition attributes as shown in Figure 10-18.

We specified the name of the DB2 subsystem to which the CICS DB2 attachment facility is to connect, in our case `DBZ1`, and the name of the DB2 plan to be used, which was `DSNJDBC`. Also, we specified CICSRS1 as the authorized CICS user ID using the attributes SIGNID and AUTHTYPE.

Note: DROLLBACK(YES) should not be specified on a DB2ENTRY definition or the DB2CONN pool definition used by transactions running enterprise beans as part of an OTS transaction. With this attribute, if a deadlock is detected, the CICS-DB2 Attach will issue a CICS syncpoint rollback request, which is not allowed in an OTS transaction, and an ASPD abend will result. Enterprise beans should use DROLLBACK(NO). They should test for an SQLException with an SQLCODE of **-913** and issue an OTS rollback request.

```
B2Conn        : DB2CON
 Group          : ITSOEJB
 DEscription  ==> DB2 CONNECTION FOR JDBC / SQLJ
CONNECTION ATTRIBUTES
 CONnecterror ==> Sqlcode            Sqlcode | Abend
 DB2id        ==> DBZ1
 MSGQUEUE1    ==> CDB2
 MSGQUEUE2    ==>
 MSGQUEUE3    ==>
 Nontermrel   ==> Yes                Yes | No
 PUrgecycle   ==> 00 , 30            0-59
 SIgnid       ==> CICSRS1
 STANdbymode  ==> Reconnect          Reconnect | Connect | Noconnect
 STATsqueue   ==> CDB2
 TCblimit     ==> 0012               4-2000
 THREADError  ==> N906D              N906D | N906 | Abend
POOL THREAD ATTRIBUTES
 ACcountrec   ==> None               None | TXid | TAsk | Uow
 AUTHId       ==>
 AUTHType     ==> Sign              Userid | Opid | Group | Sign | TErm
                                    | TX
 DRollback    ==> No                 Yes | No
 PLAN         ==> DSNJDBC
 PLANExitname ==>
 PRiority     ==> High               High | Equal | Low
 THREADLimit  ==> 0003               3-2000
 THREADWait   ==> Yes                Yes | No
COMMAND THREAD ATTRIBUTES
 COMAUTHId    ==>
 COMAUTHType  ==> Userid             Userid | Opid | Group | Sign | TErm
                                    | TX
 COMThreadlim ==> 0001               0-2000
```

Figure 10-18 Define panel for DB2CONN

Installing a DB2 connection

A DB2CONN must be installed before the CICS DB2 connection can be started. Because it contains information regarding pool threads and command threads, as well as global type information, a DB2CONN represents the minimum required to start the CICS DB2 connection. Only one DB2CONN can be installed in a CICS region at any one time.

To install the DB2CONN, we used the CEDA INSTALL command from a CICS terminal:

```
CEDA INSTALL DB2CONN(DB2CON) GROUP(ITSOEJB)
```

Starting the CICS DB2 attachment facility

The CICS DB2 attachment facility can be started automatically at initialization, or manually using the `CEMT SET DB2CONN` command. This command sets the attributes of the CICS DB2 connection. To start the CICS DB2 connection, we used the following command:

```
CEMT SET DB2CONN CONNECTED
```

10.2.6 Granting privileges to the CICS user ID

To enable the authorized CICS user ID CICSRS1 to access the tables TRADER_COMPANY and TRADER_USER we had to grant the appropriate privileges to this user ID. In order to allow applications to use JDBC to access DB2 data we had to grant general access for the the package DSNJDBC.* and the DB2 plan DSNJDBC.

Grant table privilege

For the table TRADER_COMPANY, the user ID CICSRS1 needs only the privilege to use the SELECT statement. We granted that privilege with the following command:

```
GRANT SELECT ON TABLE ITSOEJB.TRADER_COMPANY TO CICSRS1;
```

For the table TRADER_USER, the CICSRS1 needs the privileges to use the SELECT, INSERT, UPDATE and DELETE statement. We granted these privileges with the following command:

```
GRANT SELECT,INSERT,UPDATE,DELETE ON TABLE ITSOEJB.TRADER_USER TO CICSRS1;
```

Note: Using dynamic SQL from a generic interface such as JDBC, the DB2 uses the end user's privileges to access relational data. This is required because DB2 does not have any secure way to determine which program the end user is executing. Instead, DB2 simply detects SQL statements issued by a JDBC driver. In order to allow the end user to perform the SQL statements, you have to grant table privileges for the tables accessed by the JDBC application. In our case, the CICS user ID CICSRS1 needed the proper table privileges to be able to perform the SQL commands specified in the JDBC application.

Grant plan privileges

In "Bind the DBRMs" on page 278, we described how to bind the DBRMs into packages and include them into a plan. These packages and the plan are needed in order to run JDBC applications on OS/390. Therefore, they must be made available for all JDBC applications. We granted the necessary privileges with the following commands:

```
GRANT EXECUTE ON PLAN DSNJDBC TO PUBLIC;
GRANT EXECUTE ON PACKAGE DSNJDBC.* TO PUBLIC;
```

10.2.7 Testing the JDBC enterprise bean

We tested our JDBC version of our Trader enterprise bean in two different ways.

Test the enterprise bean with TraderTest

First we tested with our standalone test program TraderTest. This is a standalone Java application that can be run from either a command line or within VAJ. Instructions on how to use this from the command line are given in 10.1, "Quick start — Invoking TraderBean" on page 255. We edited the following line to invoke the traderTest application with the DB2SQLJ option.

```
java -classpath ".;traderTest.jar;traderCLI.jar;C:\Program Files\IBM\CICS TS 2.1
Tools\Common\j2ee.jar" itso.ejb390.trader.test.TraderTest
com.sun.jndi.cosnaming.CNCtxFactory iiop://hecate:900/ ITSO/PJA5/Trader DB2JDBC
```

The successful output of `runtest.cmd` is shown in Example 10-11.

Example 10-10 Output of runtest.cmd for DB2JDBC

```
C:\itsotrader>runtest.cmd
Starting TraderTest application with following input:
  Name service: com.sun.jndi.cosnaming.CNCtxFactory
  Naming Server: iiop://hecate:900/
```

```
  JNDI name: ITSO/PJA5
  Call type: DB2JDBC

Casey_Import_Export
Glass_and_Luget_Plc
Headworth_Electrical
IBM
CommissionCostBuy 010
CommissionCostSell 015
NumberOfShares 0198
TotalShareValue 000032274.00
UnitSharePrice 00163.00
UnitValue1Days 00163.00
UnitValue2Days 00162.00
UnitValue3Days 00160.00
UnitValue4Days 00161.00
UnitValue5Days 00159.00
UnitValue6Days 00156.00
UnitValue7Days 00157.00
Now we buy 5 shares ...
... and sell 2 of theM
```

Test the enterprise bean with TraderServlet

Next we tested the our JDBC application from our TraderServlet that we previously developed in 7.4.2, “Servlet development with VisualAge for Java” on page 199. We invoked the Trader servlet on our using the URL:

```
http://hecate/trader/Logon.jsp
```

We provided the following input parameters to the initial servlet HTML:

Communication type	DB2JDBC
JndiPrefix	ITSO/PJA5
NameService	com.ibm.ejs.ns.jndi.CNInitialContextFactory
ProviderURL	iiop://hecate:900/

The output of the TraderServlet was the same as shown in “Testing the Trader servlet” on page 212.

10.3 Accessing DB2 using SQLJ

SQLJ is a standard way to embed SQL statements in Java programs; it is a lower level API than JDBC, so it is less complex and more concise. You can use both JDBC and SQLJ statements in the same source code.

The SQLJ standard has three components: embedded SQLJ, a translator, and a runtime environment. The translator translates SQLJ files that contain embedded SQLJ to produce .java files and profiles that use the runtime environment to perform SQL operations.

You can find general SQLJ information at the following Web site:

```
http://www.sqlj.org
```

For more information on SQLJ syntax for DB2, visit the IBM DB2 SQLJ Web site at:

```
http://www.software.ibm.com/data/db2/java/sqlj
```

The next sections describe the following steps which were necessary to develop and test our session bean accessing SQLJ. These steps were:

1. Developing the SQLJ application.
2. Deploying the session bean to CICS.
3. Preparing the SQLJ program on OS/390.
4. Modifying the CICS DB2 connection.
5. Grants privileges to CICS user ID.
6. Refresh the DJAR in the CICS system.
7. Testing the enterprise bean.

10.3.1 Developing the SQLJ application

This section describes how to write an session bean accessing DB2 for OS/390 using SQLJ. We describe the following steps:

1. Adapting TraderBean for use of SQLJ.
2. Setting up SQLJ support in VAJ.
3. Creating the `TraderBackendDB2SQLJ` SQLJ file.
4. Implementing `TraderBackendDB2SQLJ`.

Adapting TraderBean for use of SQLJ

This section shows which changes are necessary for class TraderBean. As you will see, it is only necessary to modify method `loadClass()` as illustrated in the next section. No other change is necessary for TraderBean.

Modifying TraderBean.loadClass()

The method `loadClass()` needs to be modified in such a way that it can also load the new SQLJ back-end class, named TraderBackendDB2SQLJ, as illustrated in Figure 10-19.

```
private void loadClass(String type) throws Exception {

Class loadClass=null;

if( type.equalsIgnoreCase("JCICS-COBOL") == true ) {
  loadClass = Class.forName("itso.ejb390.trader.TraderBackendJcics");}
else if( type.equalsIgnoreCase("CICSConnectorCCF") == true ) {
  loadClass = Class.forName("itso.ejb390.trader.TraderBackendCICSConnectorCCF");}
else if( type.equalsIgnoreCase("JCICS-Java") == true ) {
  loadClass = Class.forName("itso.ejb390.trader.TraderBackendVsam");}
else if( type.equalsIgnoreCase("DB2JDBC") == true ) {
  loadClass = Class.forName("itso.ejb390.trader.TraderBackendDB2JDBC");}
else if( type.equalsIgnoreCase("DB2SQLJ") == true ) {
  loadClass = Class.forName("itso.ejb390.trader.TraderBackendDB2SQLJ");}
else { throw new TraderException( "You specified unknown type " + type );}

ivTraderBackend = (TraderBackend)loadClass.newInstance();

}
```

Figure 10-19 TraderBean.loadClass() loading TraderBackendDB2SQLJ

Setting up SQLJ support in VAJ

VisualAge for Java provides an SQLJ Tool that implements the SQLJ standard, enabling you to simplify database access. The translator component is integrated into the IDE, enabling you to import, translate, and edit SQLJ files. The runtime environment is an installable feature in VisualAge for Java. The runtime environment must be added to your workspace before you can successfully compile and execute translated SQLJ code.

To add the *SQLJ Runtime Library* feature to your workspace, select **File -> Quickstart -> Features -> Add feature** and click the **OK** button. Now select the *SQLJ Runtime Library V3.0* from the list, and click the **OK** button.

Creating the TraderBackendDB2SQLJ SQLJ file

You cannot add `#sqlj` statements directly to your source code in a source window in VAJ. You have to create a new `.sqlj` file in your file system with a text editor, which you can then import into your VAJ project. Figure 10-20 shows the content of the file `TraderBackendDB2SQLJ.sqlj` we used when importing it into VAJ.

```
package itso.ejb390.trader;
import java.sql.*;
public class TraderBackendDB2SQLJ implements TraderBackend {
public TraderBackendDB2SQLJ() {
  super();
}
public void buy(String company, String userID, int numberOfShares) throws
java.lang.Exception {}
private void closeConnection() throws SQLException {}
public void ejbBackendActivate() throws java.rmi.RemoteException {}
public void ejbBackendCreate() throws java.rmi.RemoteException {}
public void ejbBackendPassivate() throws java.rmi.RemoteException {}
public void ejbBackendRemove() throws java.rmi.RemoteException {}
public CompaniesBean getCompanies() throws Exception {}
public QuotesBean getQuotes(String company, String userID) throws Exception {}
public void logoff() throws Exception {}
public void logon(String userID, String password, String connectURL, String cicsServer)
throws Exception {}
private void openConnection() throws SQLException {}
public void sell(String company, String userID, int numberOfShares) throws Exception {}
}
```

Figure 10-20 SQLJ skeleton file of TraderBackendDB2SQLJ class

Once you have created an SQLJ file, you need to import it into a project and translate the file, before you can use it. To import the SQLJ file, select **Workspace -> Tools -> SQL -> Import**. In the SQLJ Import window type a project name in the **Project Name** field (in our case ITSO EJB 390 Redbook), and type an **SQLJ file** name in the SQLJ file name field.

Note: The name of the SQLJ file must be same as the class which is defined in the `.sqlj` file. In our case, we have provided the file `TraderBackendDB2SQLJ.sqlj`.

One you have imported an SQLJ file into your project, you can edit and translate it from within VAJ. Each time you edit an imported SQLJ file, you need to translate it to ensure that the SQLJ file in your project resources directory and the source code in your project are synchronized.

To edit or translate your SQLJ file, right-click your project, select **Open** and select the **Resources** tab. To edit the SQLJ file, right-click it and select **Tools -> SQL -> Edit**. To translate the SQLJ file, right-click it and select **Tools -> SQL -> Translate**.

Implementing TraderBackendDB2SQLJ

This section demonstrates how to implement class TraderBackendDB2SQLJ, which uses SQLJ to access DB2 on OS/390. As shown in “Implementing TraderBackendDB2JDBC” on page 258, this class must implement all the following methods belonging to the TraderBackend interface class, as follows:

- `ejbBackendCreate()`
- `ejbBackendRemove()`
- `ejbBackendPassivate()`
- `ejbBackendActivate()`
- `logon()`
- `getCompanies()`
- `getQuotes()`
- `buy()`
- `sell()`
- `logoff()`

It is also necessary to implement two further methods to manage the database connection:

- `openConnection()`
- `closeConnection()`

TraderBackendDB2SQLJ class

The declaration of TraderBackendDB2SQLJ is shown in Figure 10-21. It is similar to TraderBackendDB2JDBC, but declares the variable `ivConCtx` instead of `ivCon`. Also, it defines *iterators* to retrieve the rows from the result table.

```
import java.text.*;
import javax.naming.*;
import java.sql.*;
import sqlj.runtime.*;

#sql context ctx;

public class TraderBackendDB2SQLJ implements TraderBackend {

       #sql public static iterator iter_selectCompany (String c_name);
       #sql public static iterator iter_selectQuotes
          (double c_sv_1d,...,double c_sv_7d,double c_sv_now,
           String c_cc_sell,String c_cc_buy);
       //fields to be stored if passivated
       private java.lang.String ivUserID ="";
       private java.lang.String ivJDBCURL =null;
       private ctx ivConCtx =null;
}
```

Figure 10-21 Declaration of TraderBackendDB2SQLJ

Each SQLJ executable clause requires, either explicitly or implicitly, a *connection context* object that designates the database connection at which the SQL operations specified in that clause will be executed. In the declaration of TraderBackendDB2SQLJ we define a connection class named `ctx` with the SQLJ clause `#sql context ctx`. The instance variable `ivConCtx` is later used to invoke the constructor for class `ctx` using a JDBC connection as the argument.

A capability central to SQL is the ability to execute queries that retrieve a result set of rows from the database. An SQLJ result set iterator is a Java object from which data returned by an SQL query can be retrieved. SQLJ supports two mechanism for matching iterators columns to query columns: *bind by position* and *bind by name*.

We use the named bindings to columns in our sample. When you declare an iterator that binds by name, you specify names for each of the iterator columns. Those names must match the names of columns in the database. The names are case-insensitive. The iterators iter_selectCompany, iter_selectQuotes and iter_selectShareNumber are used in our sample to retrieve the data returned by the SQL queries.

Also, TraderBackendDB2SQLJ imports one additional package, `sqlj.runtime.*`, which provides the SQLJ runtime support.

TraderBackendDB2SQLJ.openConnection()

The `openConnection()` method requests a connection instance from the DriverManager by specifying the location of the database as URL. It sets the autocommit property of the JDBC connection to *false*, and invokes the constructor for the connection context class, using the JDBC connection as an argument. The implementation of method `openConnection()` is illustrated in Figure 10-22.

```
private void openConnection() throws SQLException
{
  Connection con = DriverManager.getConnection(ivJDBCURL);
  con.setAutoCommit(false);
  ivConCtx = new DefaultContext (con);
}
```

Figure 10-22 TraderBackendDB2SQLJ.logon()

TraderBackendDB2SQLJ.closeConnection()

The `closeConnection()` method closes the connection connect object, which also closes the underlying connection and frees all resources associated with the connection. Then it sets the instance variable `ivConCtx` to null to avoid serialization problems. Figure 10-23 shows the implementation of method `closeConnection()`.

```
private void closeConnection() throws SQLException
{
  ivConCtx.close();
  ivConCtx = null;
}
```

Figure 10-23 TraderBackendDB2JDBC.logoff()

Control methods for TraderBackendDB2SQLJ

The control methods `ejbBackendCreate()`, `ejbBackendRemove()`, `ejbBackendActivate()`, `ejbBackendPassivate()` and `ejbBackendRemove()` are identical to the implementation used in TraderBackendDB2JDBC, because the differences in establishing a connection using JDBC and SQLJ are hidden in the private methods `openConnection()` and `closeConnection()`. Their TraderBackendJDBC control methods are shown in Figure 10-6 on page 260 to Figure 10-9 on page 262.

TraderBackendDB2SQLJ.logon()

The `logon()` method is very similar to `TraderBackendDB2JDBC.logon()`, but simpler. Figure 10-24 shows the implementation of the method `logon()`.

```
public void logon(String userID, String password, String connectURL, String cicsServer)
throws Exception
{
1      ivUserID = userID;
2      iter_selectCompany i = null;
       try {
3        #sql [ ivConCtx ] i = { SELECT c_name FROM trader_company};
         String company = null;
4    while (i.next()) {
       company = i.c_name();
           try {
5         #sql [ ivConCtx ] { INSERT INTO trader_user
                    (u_name, u_c_name, u_sn_held)
                    VALUES (:userID, :company, 0) };
           } catch (SQLException ex) {
               if (ex.getErrorCode() != -803) throw ex;
           }
         }
       } catch (Exception ex) {
           ex.printStackTrace(); throw ex;
       } finally {
6          if (i != null) i.close();
       }
}
```

Figure 10-24 TraderBackendDB2SQLJ.logon()

The following list summarize the logic in method `logon()`:

- 1 Store the provided user ID in the class's instance variable.

  The user ID needs to be stored, since it is required to clean up the database when the user logs off from the application.

- 2 Define the iterator object.

  The iterator is used to retrieve the company names from the result set returned by the SQL query. The named iterator class iter_selectCompany was created by the following SQLJ clause:

  ```
  #sql public static iterator iter_selectCompany (String c_name);
  ```

  The iterator class has the accessor method `c_name()`, which returns the data from the result table column.

- 3 Execute the SQL query.

  This SQLJ clause executes the SELECT statement, constructs an iterator object that contains the result table for the SELECT statement, and assigns the iterator object to variable `i`. The connection context that is used for executing the SQL operation is the value of the Java variable `ivConCtx`. As you can see, we use an unqualified name for the table name (that means that we specify only the second part of the two-part name). We provide the qualifier later when processing the SQLJ file on OS/390.

- 4 Retrieve the result and copy the company name to the local variable company.

  The `next()` method, which is a method of the generated class iterator_selectCompanies, advances the iterator to successive rows of the result set. The method `next()` returns a value of *true* when a next row is available, and a value of *false* waen all rows have been fetched.

- 5 Execute the SQL update.

  This SQLJ clause performs an insert operation. In our sample, a new row is inserted into the TRADER_USER table for each company. The values for the user ID, company name and the number of shares are provided by *host variables*, which are distinguished by the colon character (:).

  Note: As for the logon method of TraderBackendDB2JDBC, we have to tolerate the SQL error -803 (insert would result in duplicate values in index columns), because the application keeps all positive (>0) share holdings when the user logs off from the application.

- 6 Close the iterator.

  The `close()` method closes the SQLJ iterator.

TraderBackendDB2SQLJ.getCcompanies()

Figure 10-25 shows the implementation of method `getCcompanies()`.

```
public CompaniesBean getCompanies() throws Exception
{
1 iter_selectCompany i = null;
2 CompaniesBean companies = new CompaniesBean();
  try {
3   #sql [ ivConCtx ] i = { SELECT c_name FROM trader_company};
4   while (i.next()) {
      companies.addCompany(i.c_name());
    }
  } catch (Exception ex) {
     ex.printStackTrace(); throw ex;
  } finally {
5 if (i != null) i.close();
  }
6 return companies;
}
```

Figure 10-25 TraderBackendDB2SQLJ.getCompanies()

The `getCompanies()` method does the following:

- 1 Define the iterator object.
- 2 Instantiate a CompaniesBean.
- 3 Execute the SQL query.
- 4 Retrieve the result and copy the company names to the CompaniesBean.

- 5 Close the iterator.
- 6 Return the CompaniesBean.

TraderBackendDB2SQLJ.getQuotes()

Figure 10-26 shows the implementation of method `getQuotes()`.

```
public QuotesBean getQuotes(String company, String userID) throws Exception
{
1  iter_selectQuotes i = null;

2  QuotesBean quotes = new QuotesBean();

   try {
      // perform the sqlj statement (select the quotes of a company)
3     #sql [ ivConCtx ] i = { SELECT c_sv_1d, ... ,c_sv_7d, ,c_sv_now,
                                          c_cc_sell,c_cc_buy
               FROM trader_company
               WHERE c_name = :company };
      double unitSharePrice = 0;
      DecimalFormat df = new DecimalFormat("00000.00");
      FieldPosition fp = new FieldPosition(1);

4     if (i.next()) {
        quotes.setUnitValue1Days(df.format(i.c_sv_1d(),
                                       new StringBuffer(),fp).toString());
                                ...
             quotes.setUnitValue7Days(df.format(i.c_sv_7d(),
                                       new StringBuffer(),fp).toString());
        unitSharePrice = i.c_sv_now();
        quotes.setUnitSharePrice(df.format(unitSharePrice,
                                       new StringBuffer(),fp).toString());
        quotes.setCommissionCostSell(i.c_cc_sell());
        quotes.setCommissionCostBuy(i.c_cc_buy());
         }
           int numberOfSharesHeld = 0;
5       #sql [ ivConCtx ] { SELECT u_sn_held
                            INTO :numberOfSharesHeld
                            FROM trader_user
                            WHERE u_name = :userID AND u_c_name = :company};

6 quotes.setNumberOfShares( new DecimalFormat("0000").format(
                  numberOfSharesHeld,new StringBuffer(),fp).toString());
         double totalShareValue = unitSharePrice * numberOfSharesHeld;
         quotes.setTotalShareValue(df.format(totalShareValue,
                                      new StringBuffer(),fp).toString());
   } catch (Exception ex) {
      ex.printStackTrace(); throw ex;
   } finally {
7     if (i != null) i.close();
   }
8  return quotes;
}
```

Figure 10-26 TraderBackendDB2SQLJ.getQuotes()

The `getQuotes()` method requires the company name and the user ID to do the following steps:

- 1 Define the iterator object.

- 2 Instantiate a QuotesBean.

  QuotesBean is a class which is used to hold all quote specific information returned by the SQL statement.

- 3 Execute the SQL query.

  The named iterator class iter_selectQuotes was created by the following SQLJ clause:

```
#sql public static iterator iter_selectQuotes
 (double c_sv_1d, ... , c_sv_7d, double c_sv_now,
  String c_cc_sell, String c_cc_buy);
```

- 4 Retrieve the result and copy the information to the QuotesBean.

  We have used the class *DecimalFormat* to format the decimal numbers to the same format that is used by the COBOL Trader application.

- 5 Execute the SQL query.

  Rather than creating an iterator, we use here SQLJ's support of SELECT ... INTO ... FROM, because the query returns only one row.

- 6 Copy the information to the QuotesBean.

  The total value is calculated from the number of shares hold by the user multiplies by the current share value.

- 7 Close the iterators.

- 8 Return the QuotesBean.

TraderBackendDB2SQLJ.buy()

The `buy()` method uses the company name, the user ID, and the number of shares, as follows (Figure 10-27).

```
public void buy(String company, String userID, int numberOfShares) throws
java.lang.Exception
{
1 if (numberOfShares > 9999) return;
  try {
    int numberOfSharesHeld = 0;
2 #sql [ ivConCtx ] { SELECT u_sn_held INTO :numberOfSharesHeld FROM trader_user
                        WHERE u_name = :userID AND u_c_name = :company};
3 numberOfSharesHeld += numberOfShares;
4 if (numberOfSharesHeld > 9999) return;
5 #sql [ ivConCtx ] { UPDATE trader_user
                        SET u_sn_held = :numberOfSharesHeld
                        WHERE u_name = :userID and u_c_name = :company};
  } catch (Exception ex) {
   ex.printStackTrace(); throw ex;
  }
}
```

Figure 10-27 TraderBackendDB2SQLJ.buy()

The following steps are performed in the `buy()` method:

- 1 Verify that the number of shares to buy does not exceed the maximum.

  The maximum number of shares a user can buy is 9999 and is defined within the COBOL Trader application.

- 2 Execute the SQL query.

- 3 Calculate the new number of shares held.
- 4 Verify that the number of shares held does not exceed the maximum (9999).
- 5 Execute the SQL update. This SQLJ clause performs an update operation.

TraderBackendDB2SQLJ.sell()

The `sell()` method is very similar to the `buy()` method. The only difference is how it calculates the new number of shares held by the customer, as shown in Figure 10-28.

```
public void sell(String company, String userID, int numberOfShares) throws Exception
{
   if (numberOfShares > 9999) return;

   try {
               // perform the sqlj statement (select number of shares held)
      int numberOfSharesHeld = 0;
      #sql [ ivConCtx ] { SELECT u_sn_held INTO :numberOfSharesHeld FROM trader_user
WHERE u_name = :userID AND u_c_name = :company};

               // calculate new number of shares held
      if (numberOfShares > numberOfSharesHeld) return;
      numberOfSharesHeld -= numberOfShares;
      // perform the sqlj statement (update number of shares held)
      #sql [ ivConCtx ] { UPDATE trader_user
               SET u_sn_held = :numberOfSharesHeld
               WHERE u_name = :userID and u_c_name = :company};
   } catch (Exception ex) {
      System.out.println("Exception detected!");
      System.out.println("********* S T A C K T R A C E *********");
      ex.printStackTrace();
      throw ex;
   }
}
```

Figure 10-28 TraderBackendDB2SQLJ.sell()

TraderBackendDB2SQLJ.logoff()

The logoff() method is shown in Figure 10-29.

```
public void logoff() throws Exception
{
  try {
1   #sql [ ivConCtx ] { DELETE FROM trader_user
                        WHERE u_name = :ivUserID AND u_sn_held = 0};
  } catch (Exception ex) {
      ex.printStackTrace(); throw ex;
  }
}
```

Figure 10-29 TraderBackendDB2SQLJ.logoff()

This `logoff()` method performs the following:

- 1 Execute the SQL update.

  This SQLJ clause performs a delete operation.

10.3.2 Deploying the enterprise bean to CICS

This section describes how to deploy TraderBean and its related classes to CICS TS V2.1. We need to do the following steps.

1. Export the enterprise bean and its related classes.
2. Convert the exported file to a DJAR file.
3. Send the DJAR file to the OS/390 system.

Export the enterprise bean and its related classes

Export the enterprise bean the same way as described in “Export the enterprise bean and its related classes” on page 271. The difference is that you have to include the new class TraderBackendDB2SQLJ and the additional classes and resources this uses.

1. Within VisualAge for Java, select the **EJB** tab to view the EJB groups. Select group **ITSOEJB390** and click the right mouse button.
2. Select **Export -> EJB JAR** to open the *Export to an EJB JAR File SmartGuide*.
3. You should see that the Trader bean and 3 additional classes are selected by default. Click **Select referenced types and resources** to ensure that VAJ now also selects classes which are referenced by the enterprise bean. This will cause *CompaniesBean, QuotesBean, TraderBackend,* and *TraderException* to also be selected. But VAJ has not selected the classes which implement the different back-ends.
4. In addition select the following classes which implement the back-ends:
   - CompaniesBean
   - CompanyKeyRecord
   - CompanyKeyRecordBeanInfo
   - CompanyKeyRecordType
   - CompanyRecord
   - CompanyRecordType
   - CustomerKeyRecord
   - CustomerKeyRecordBeanInfo
   - CustomerKeyRecordType
   - CustomerRecord
   - CustomerRecordBeanInfo
   - CustomerRecordType
   - ctx
   - TraderBackendCICSConnectorCCF
   - TraderBackendDB2JDBC
   - TraderBackendDB2SQLJ
   - TraderBackendDB2SQLJ_SJProfileKeys
   - TraderBackendJcics
   - TraderBackendVsam
   - TraderCommand
   - TraderCommandBeanInfo
   - TraderRecord
   - TraderRecordBeanInfo
   - TraderRecordType

   Now click **OK** to close the window.
5. In the export window you should now see that 1 bean and 31 classes are selected.
6. Select **resource** check box and click **Details**.
7. Select the **ITSO EJB 390 Redbook** check box in the left panel of the window so that the resources will also be exported. One of these resources is the uncustomized profile

TraderBackendDB2SQLJ_SJProfile.ser, which is addressed in 10.3.3, “Preparing the SQLJ program on OS/390” on page 295.

8. Then click **OK** to close the window.
9. In the export window you should now see that 1 bean, 31 classes, and 2 resources are selected.
10. Specify path and file name for the JAR file. We used `C:\itsotrader\traderForCICS.jar`. Now your window should look as shown in Figure 10-30.

Click **Finish** to export the classes to file `traderForCICS.jar`.

Tip: If you receive a message from VAJ stating that *the file is not a zip file, or it is corrupted*, you should close the CICS Java development tool, or delete the output JAR file.

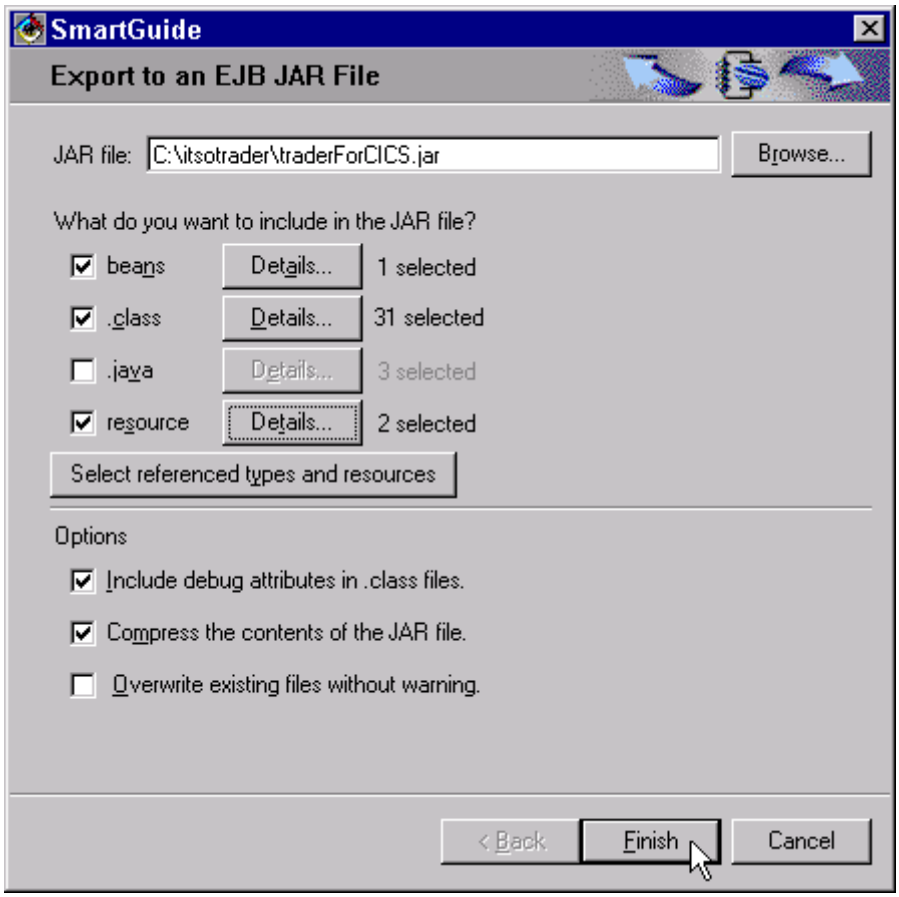


Figure 10-30 Exporting TraderBean for CICS (including TraderBackendDB2SQLJ)

Convert the exported file to a DJAR file

Convert the exported EJB-JAR file to a deployed JAR file using the CICS JAR development tool as described in “Convert the exported file to a DJAR file” on page 272. Note, however, that you need to click the **Retain** button when you are asked to remove the old EJB1.0 information, otherwise the SQLJ profile will be deleted from the DJAR file.

Send the DJAR file to the OS/390 system

Within UNIX System Services (USS) we created the directory `/u/cicsts21/djars`. With standard FTP we sent the file `traderForCICS_GEN.jar` in binary mode to this directory.

10.3.3 Preparing the SQLJ program on OS/390

After you write an SQLJ application, you must generate an executable form of the application. With SQLJ, creating an executable is different from JDBC, where you just need to compile your code to get an executable. As shown in Figure 10-31, preparing the SQLJ application for execution on OS/390 involves the following steps:

1. Translating and compiling the SQLJ application.
2. Customizing the SQLJ serialized profile.
3. Binding the plan for the SQLJ application.

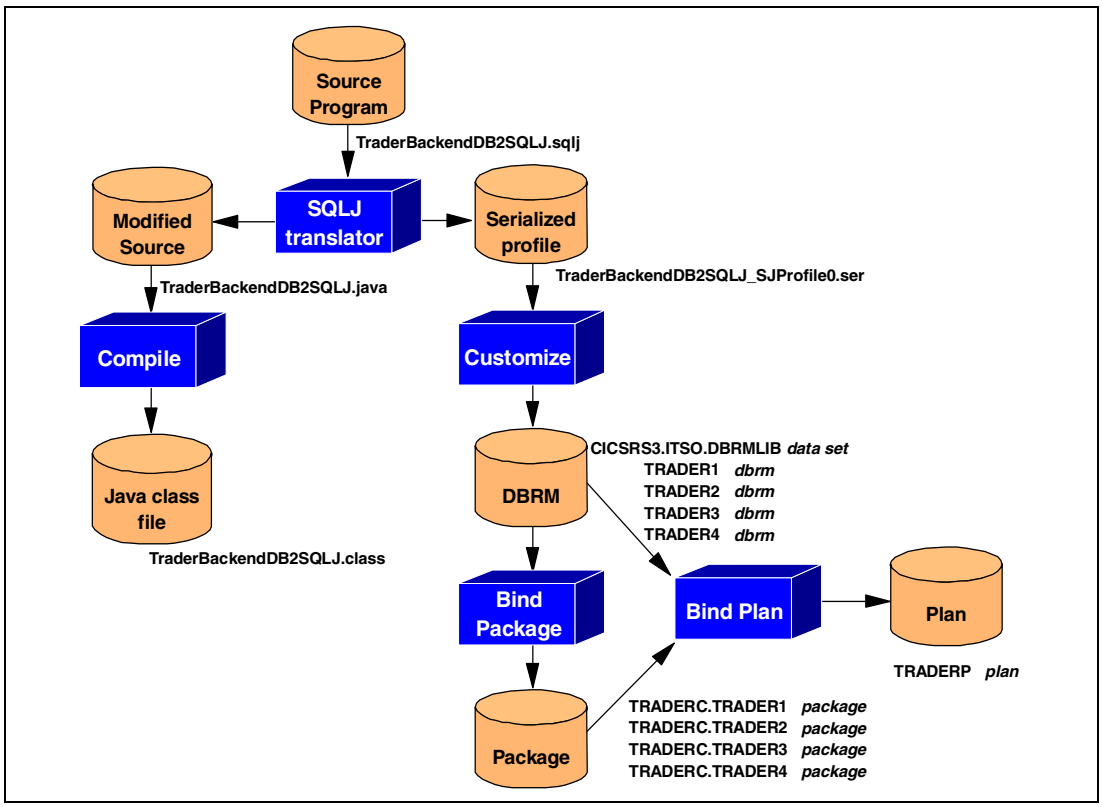


Figure 10-31 The SQLJ program preparation process

Translating and compiling the SQLJ application

This step is performed by VAJ during the import of your SQLJ file or when you start the translate step from within VAJ explicitly, as described in “Creating the TraderBackendDB2SQLJ SQLJ file” on page 285. VAJ itself invokes the DB2 SQLJ translator, which generates a Java source program and produces a default serialized profile. The serialized profile file is named *program-name*`_SJProfile0.ser`. In our case, the translation of the file `TraderBackendDB2SQLJ.sqlj` produced the following files:

```
TraderBackendDB2SQLJ.java
TraderBackendDB2SQLJ_SJProfile.ser
```

The SQLJ translator replaces embedded SQL statements in the SQLJ program with calls to a generic SQLJ runtime layer and saves information about the SQL operations found in the SQLJ program in the default serialized profile. Because all database vendors supporting SQLJ ship the same SQLJ translator, binary portability of the serialized profile is guaranteed. Using the default serialized profile at runtime, all SQLJ statements are mapped to JDBC.

Customizing the SQLJ default serialized profile

Customizing a default serialized profile is the process to replace that profile with a customized profile which can be used to exploit optimizations available within the target database. DB2 uses this capability to replace the default profile, which maps SQLJ to JDBC, with a customized profile that maps SQLJ statements to pre-bound DB2 static SQL packages and plans. This customization lets Java exploit DB2's static SQL support.

Note: By default, if you run an SQLJ program with an uncustomized profile, then dynamic SQL is used, since the SQL will be mapped to pure JDBC calls at runtime. Only if you perform the additional steps on OS/390 to customize the profile will static SQL really be used for SQLJ.

The default serialized profile generated by the SQLJ translator tool must transferred to the OS/390 system. In our case, the profile is packaged together with the enterprise bean and its related classes in the DJAR file. The DJAR file was already sent to the OS/390 system, as described in "Send the DJAR file to the OS/390 system" on page 294.

On OS/390, the DB2 tool `db2profc` is used from the USS shell to customize a serialized profile. The output from the SQLJ customizer are four DBRMs (one for each isolation level, as illustrated in Table 10-7) and a modified (customized) serialized profile.

Table 10-7 Trader's SQLJ DBRMs and their isolation level

DBRM Name	Bind with isolation level
*DBRM-name*1	Uncommitted read (**UR**)
*DBRM-name*2	Cursor stability (**CS**)
*DBRM-name*3	Read stability (**RS**)
*DBRM-name*4	Repeatable read (**RR**)

The shell script we used to generate a customized serialized profile, `db2profc.sh`, is illustrated in Figure 10-11. You have to pass the name (without the suffix) as parameter to the shell script. In our case, we used the command:

```
db2profc.sh traderForCICS_GEN
```

Tip: You have to make sure that the SQLJ customized profile is accessible within your CICS environment. The recommended way is to package it within your DJAR file. The shell script shown in Example 10-11 updates the customized profile `TraderBackendDB2SQLJ_SJProfile0.ser` in the deployed JAR file `traderForCICS_GEN.jar`

Example 10-11 USS shell to execute db2profc

```
#------------------------------------------------------------------------
# Shell script to run the DB2 utility db2profc from USS.
#
# Modify the following to match your DB2/390 installation directory:
DB2_ROOT=/service_db2v6
DB2_HOME=/service_db2v6/usr/lpp/db2/db2610
#------------------------------------------------------------------------
if ls -la $1.jar
  then
  else echo "Please specify the name of the JAR file (without the suffix)."
       echo "f.e. traderForCICS_GEN"
       exit
fi
```

```
jar -xvf $1.jar itso/ejb390/trader/TraderBackendDB2SQLJ_SJProfile0.ser
if ls -la itso/ejb390/trader/TraderBackendDB2SQLJ_SJProfile0.ser
  then rm -r itso
  else echo "The JAR file must contain the following class:"
       echo "itso/ejb390/trader/TraderBackendDB2SQLJ_SJProfile0.ser"
       exit
fi
export CLASSPATH=$CLASSPATH:$DB2_HOME/classes/db2sqljclasses.zip
export LD_LIBRARY_PATH=$DB2_HOME/lib:$LD_LIBRARY_PATH
export LIBPATH=$DB2_ROOT/usr:$DB2_ROOT/usr/lib:$DB2_HOME/lib:$LIBPATH
export PATH=$DB2_HOME/bin:$PATH
export STEPLIB=DSN610.SDSNLOAD
export DB2SQLJPROPERTIES=../mydb2sqljjdbc.properties
#
cp ./$1.jar ./$1_uncustomized.jar
mkdir temp
cp ./$1.jar temp
cd temp
jar -xvf ./$1.jar
rm ./$1.jar
$DB2_HOME/bin/db2profc -schema=ITSOEJB -pgmname=TRADER
itso/ejb390/trader/TraderBackendDB2SQLJ_SJProfile0.ser
jar -cvf ./$1.jar *
cp ./$1.jar ./..
cd ..
rm -r temp
jar -tvf ./$1.jar itso/ejb390/trader/TraderBackendDB2SQLJ_SJProfile0.ser
jar -tvf ./$1_uncustomized.jar itso/ejb390/trader/TraderBackendDB2SQLJ_SJProfile0.ser
```

Once this has completed successfully, you should ensure that the updated deployed JAR file, containing the customized serialized profile, is copied to the HFS directory which the CICS DJAR definition points to. We did this using the following command:

```
cp traderForCICS_GEN.jar /u/cicsts21/djars
```

Binding the plan for the SQLJ application

The SQLJ customizer produces four DBRMs, one for each isolation level. The name of the DBRMs are specified by the `-pgname` option of the `db2profc` tool — in our case, the names of the generated profiles are `TRADER1`, `TRADER2`, `TRADER3`, and `TRADER4`.

To communicate the SQL requests contained in these DBRMs to DB2, you must bind the DBRMs into packages and include them into a plan. The bind job we used is illustrated in Example 10-12 and is supplied with our sample code in the file `bindtrad.jcl`.

Note: With the QUALIFIER option we specified the qualifier for the unqualified table names used in our application. The option PKLIST determines what packages to include in the package list for the plan. In order to include also the packages necessary for JDBC applications in our plan TRADERP, we specified all packages of the collection DSNJDBC and all packages of the collection TRADERC.

Example 10-12 JCL to bind the SQLJ DBRMs

```
//CRS3BIT JOB AX4328,AXP4328,
//         CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1),
//         NOTIFY=&SYSUID,
//         REGION=0M
//*
//JOBLIB   DD  DISP=SHR,DSN=DSN610.SDSNLOAD
```

```
//BINDSQLJ EXEC PGM=IKJEFT01,DYNAMNBR=20
//DBRMLIB  DD DISP=SHR,DSN=CICSRS3.ITSO.DBRMLIB
//SYSTSPRT DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
//SYSTSIN  DD  *
  DSN SYSTEM(DBZ1)
   BIND PACKAGE (TRADERC) QUALIFIER(ITSOEJB) -
     MEMBER(TRADER1) ISOLATION(UR)
   BIND PACKAGE (TRADERC) QUALIFIER(ITSOEJB) -
     MEMBER(TRADER2) ISOLATION(CS)
   BIND PACKAGE (TRADERC) QUALIFIER(ITSOEJB) -
     MEMBER(TRADER3) ISOLATION(RS)
   BIND PACKAGE (TRADERC) QUALIFIER(ITSOEJB) -
     MEMBER(TRADER4) ISOLATION(RR)
   BIND -
     PLAN(TRADERP) QUALIFIER(ITSOEJB) -
     PKLIST(TRADERC.*, DSNJDBC.*) RETAIN
END
/*
//
```

Attention: In EJB 1.1, isolation levels are not controlled through declarative attributes, which means that you cannot specify transaction isolation in the EJB 1.1 deployment descriptor. Instead, an enterprise bean can use JDBC isolation facilities and any deployment time isolation control provided by enterprise bean containers. But note that EJB 1.1 does not require the container to provide isolation control.

10.3.4 Modifying the CICS DB2 connection

In 10.2.5, “Defining a CICS DB2 connection” on page 280, we described how to define, install, and start a CICS DB2 connection. We specified `DSNJDBC` as the plan to be used for the DB2 connection, which contains all packages necessary to run JDBC applications on OS/390. However, in “Binding the plan for the SQLJ application” on page 297, we have generated a new plan TRADERP which contains both the packages for the SQLJ application and the packages required to run JDBC applications. In order to use that plan from now on, we had to change the DB2 connection definition attribute PLAN for the CICS DB2 connection definition DB2CON.

We used the **`CEMT SET DB2CONN PLAN(TRADERP)`** command to change the plan name from DSNJDBC to TRADERP.

Note: You do not need to specify the name of the DB2 connection definition, because only one DB2CONN can be installed in a CICS system at one time. Also, it is not necessary to stop the connection before changing the PLAN attribute, because a new plan will be determined the next time the transaction releases the thread.

10.3.5 Granting privileges to the CICS user ID

In “Binding the plan for the SQLJ application” on page 297, we have described how to bind the plan for the SQLJ application. To authorize the user ID CICSRS1 to execute that plan, we used the following command:

```
GRANT EXECUTE ON PLAN TRADERP TO CICSRS1;
```

Note: The content of a static SQL statement is available to DB2 when the program is bound to DB2. DB2 binds the statements as packages or plans using the privileges of the person who issued the bind request. This step lets DB2 execute static SQL statements using the privileges of the owner of the package or plan. The owner of the package or plan can then grant execute privileges to individual end users, such as an authorized CICS user ID. In other words, end users do not need table privileges in order to run DB2 static SQL programs. Instead, only the owner of the package or plan must have these privileges. In our case, the privileges granted for the tables TRADER_COMPANY and TRADER_USER, as described in “Grant table privilege” on page 282, are not needed to run the SQLJ application. We only need these privileges to run the JDBC application.

10.3.6 Refreshing the DJAR in the CICS region

Refresh the DJAR in the shelf in the same way as described in “Refresh the DJAR in the CICS shelf” on page 273.

10.3.7 Testing the SQLJ enterprise bean

We tested our SQLJ version of our Trader enterprise bean with both our standalone test client TraderTest and our servlet.

Testing the enterprise bean with TraderTest

First we tested with our standalone test program TraderTest. This is a standalone Java application that can be run from either a command line or within VAJ. Instructions on how to use this from the command line are given in 10.1, “Quick start — Invoking TraderBean” on page 255. We edited the following line to invoke the TraderTest application with the DB2SQLJ option.

```
java -classpath ".;traderTest.jar;traderCLI.jar;C:\Program Files\IBM\CICS TS 2.1
Tools\Common\j2ee.jar" itso.ejb390.trader.test.TraderTest
com.sun.jndi.cosnaming.CNCtxFactory iiop://hecate:900/ ITSO/PJA5/Trader DB2SQLJ
```

The successful output of `runtest.cmd` is shown in Example 10-13.

Example 10-13 Output of runtest.cmd for DB2SQLJ

```
C:\itsotrader>runtest.cmd
Starting TraderTest application with following input:
  Name service: com.sun.jndi.cosnaming.CNCtxFactory
  Naming Server: iiop://hecate:900/
  JNDI name: ITSO/PJA5
  Call type: DB2SQLJ

Casey_Import_Export
Glass_and_Luget_Plc
Headworth_Electrical
IBM
CommissionCostBuy 010
CommissionCostSell 015
NumberOfShares 0198
TotalShareValue 000032274.00
UnitSharePrice 00163.00
UnitValue1Days 00163.00
UnitValue2Days 00162.00
UnitValue3Days 00160.00
UnitValue4Days 00161.00
UnitValue5Days 00159.00
UnitValue6Days 00156.00
```

```
UnitValue7Days 00157.00
Now we buy 5 shares ...
... and sell 2 of them
```

Testing the enterprise bean with TraderServlet

Next we tested the our JDBC application from our TraderServlet that we previously developed in 7.4.2, “Servlet development with VisualAge for Java” on page 199. We invoked the Trader servlet on our using the URL:

```
http://hecate/trader/Logon.jsp
```

We provided the following input parameters to the initial servlet HTML.

Communication type	`DB2SQLJ`
JndiPrefix	`ITSO/PJA5`
NameService	`com.ibm.ejs.ns.jndi.CNInitialContextFactory`
ProviderURL	`iiop://hecate:900/`

The output of the TraderServlet was the same as shown in “Testing the Trader servlet” on page 212.

10.4 Summary

This chapter has shown how to rewrite a CICS COBOL program as a session bean accessing DB2. We have used dynamic SQL in form of JDBC as well as static SQL in form of SQLJ to access the data in the database.

The primary advantages of SQLJ over JDBC are:

- **Better performance:** SQLJ computes the DB2 access path at compilation time.
- **Better security:** SQLJ executes SQL statements with the privileges of the person who created the database plan.
- **Simpler syntax and greater robustness:** SQLJ statements are verified when the precompilation is executed.

Besides these many advantages, there may exist other reasons why you want to use dynamic SQL. These reasons could be that you need all or part of the SQL statement to be generated during application execution, or that you want the statement to always use the most optimal access path, based on the current database statistics.

Besides JDBC and SQLJ, there is also another way to access DB2 from a session bean: using Data Access Beans. Data Access Beans are JavaBeans which wrap JDBC classes to provide more function and to make them easier to use. Using the sample JDBC code provided with the book, it should be easy to extend the enterprise bean in such a way.

Part 4

Appendixes

A

Security customization: DFHXOPUS

In this appendix we describe how the user replaceable module (URM) DFHXOPUS can be used to control the security authorization for inbound EJB request to CICS TS V2.1. We also provide a modified version of DFHXOPUS (COBXOPUS), written in COBOL that is capable of modifying the user ID under which an IIOP request runs based on a simple lookup table using the bean name.

For further details on how to obtain the sample code refer to Appendix C, “Using the additional material” on page 315.

Security functions of DFHXOPUS

The aim of the DFHXOPUS is to enable the setting of the CICS user ID for the request receiver transaction, CIRR.

The name of the URM to be used is set in the TCPIPSERVICE resource definition, and a PROGRAM resource definition must also be supplied. If you do not specify a URM name in the TCPIPSERVICE, no URM is called. If you use SSL client certificates to generate a user ID for incoming request then the RACF user ID associated with the certificate is used and the URM is not called. How DFHXOPUS fits into the flow of a IIOP request to the CICS EJB Server is illustrated in Figure A-1.

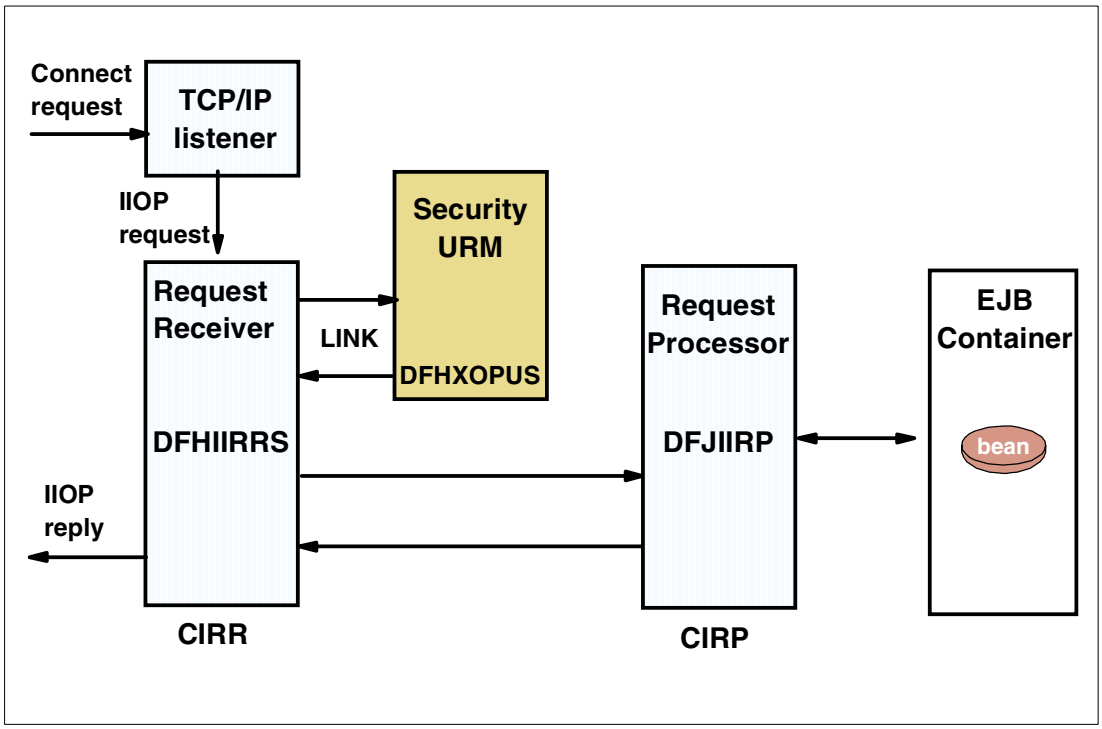


Figure A-1 DFHXOPUS function

For further details on the functions of DFHXOPUS refer to *CICS Supplied Transactions*, SC34-5724 and *Java applications in CICS*, SC34-5881.

The sample COBXOPUS

The sample URM we developed (COBXOPUS) is illustrated in Example A-1. It is written in COBOL and uses a simple lookup table to determine the CICS user ID based on the bean name. Unlike the sample C version of DFHXOPUS, it does not contain any logic to handle authentication using SSL client certificates.

In the example, if the bean name is “HelloWorld”, then the user ID CICSRS1 is used; if the bean name is “TRADER”, then CICSRS2 is used, if ithe bean name is “BEAN3”, then CICSRS3 is used. Otherwise, it defaults to using CICSUSER. In a production environment, a more sophisticated mechanism could be used — perhaps using data from a DB2 database or VSAM file to determine the user ID.

Note: You should observe that the return code from the URM should be set to 1 (one) on a successful completion and 0 (zero) on a non-successful completion. A zero return code will cause an exception to be driver in the EJB client.

Example: A-1 Sample COBXOPUS

```
************************************************************
* COBXOPUS   -  IIOP security URM
* COBOL version of DFHXOPUS C Sample
*
* Userid selected from table of bean names
*
* Provided with redbook SG246284
* For lastest version see ftp://www.redbooks.ibm.com/redbooks
************************************************************

 ID DIVISION.
 PROGRAM-ID. COBXOPUS.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
* The name of the enterprise bean padded with blanks
 01  NAMEOFBEAN                          PIC X(16).
* Message to show module has been entered.
 01  ENTRYMSG.
     05  FILLER1                         PIC X(21)
            VALUE 'COBXOPUS ENTRY; TRAN:'.
     05  ENTTRANID                       PIC X(4).
     05  FILLER2                         PIC X(6)
            VALUE ' BEAN:'.
     05  ENTBEANID                       PIC X(16).
* Message to show module is about to exit.
 01  XITMSG                              PIC X(32)
            VALUE 'COBXOPUS EXIT'.
* Message to show that a userid has been set from the table.
 01  USERIDSETMSG.
     05  FILLER                          PIC X(15)
             VALUE 'USERID SET TO: '.
     05  USETID                          PIC X(8).
     05  FILLER                          PIC X(16)
          VALUE ' FOR BEAN NAME: '.
     05  USETNAME                        PIC X(16).
* Message to show that the commarea does
* not point to a bean name.
 01  NOBEAN                              PIC X(12)
            VALUE 'NO BEAN NAME'.
************************************************************
*       table of bean names and userids                    *
************************************************************
 01  BEANNAMESANDUSERIDS.
*                                <----name-----><userid>
     05  FILLER PIC X(24) VALUE 'HelloWorld      CICSRS1 '.
     05  FILLER PIC X(24) VALUE 'TRADER          CICSRS2 '.
     05  FILLER PIC X(24) VALUE 'BEAN3           CICSRS3 '.
 01  BEANTABLE REDEFINES BEANNAMESANDUSERIDS.
     05  BEANELEMENT OCCURS 3 INDEXED BY I.
         10  NAMEINTABLE                 PIC X(16).
         10  USERIDINTABLE               PIC X(8).
 LINKAGE SECTION.
```

```
*****************************************************************
*  Commarea for COBXOPUS (standard DFHXOPUS commarea)
*****************************************************************
 01  DFHCOMMAREA.
     05  SXOPUS.
       10  STANDARDHEADER          PIC X(4).
       10  PIIOPDATA               POINTER.
       10  LIIOPDATA               PIC S9(8) COMP.
       10  PREQUESTBODY            POINTER.
       10  LREQUESTBODY            PIC S9(8) COMP.
       10  CORBASERVER             PIC XXXX.
       10  PBEANNAME               POINTER.
       10  LBEANNAME               PIC S9(8) COMP.
       10  BEANINTERFACETYPE       PIC S9(8) COMP.
       10  PMODULE                 POINTER.
       10  LMODULE                 PIC S9(8) COMP.
       10  PINTERFACE              POINTER.
       10  LINTERFACE              PIC S9(8) COMP.
       10  POPERATION              POINTER.
       10  LOPERATION              PIC S9(8) COMP.
       10  USERID                  PIC X(8).
       10  TRANSID                 PIC XXXX.
       10  FLAGBYTES               PIC XXXX.
       10  RETURNCODE              PIC S9(8) COMP.
       10  REASONCODE              PIC S9(8) COMP.
 01  PASSEDBEANNAME                      PIC X(16).
*****************************************************************
 PROCEDURE DIVISION.
         MOVE EIBTRNID TO ENTTRANID.
*****************************************************************
* put bean name into field with blank padding to right
*****************************************************************
         MOVE SPACES TO ENTBEANID.
         SET ADDRESS OF PASSEDBEANNAME TO PBEANNAME.
         MOVE PASSEDBEANNAME(1:LBEANNAME) TO ENTBEANID.
         EXEC CICS WRITEQ TD QUEUE('CSSL') FROM(ENTRYMSG)
         END-EXEC.
         IF  LBEANNAME = ZERO
         THEN EXEC CICS WRITEQ TD QUEUE('CSSL') FROM(NOBEAN)
              END-EXEC
              GO TO XIT.
*****************************************************************
* set CICSUSER as default
*****************************************************************
         MOVE 'CICSUSER' TO USERID.
*****************************************************************
* scan table and set userid if bean defined in table
*****************************************************************
         MOVE ENTBEANID TO NAMEOFBEAN.
         SET  I TO 1.
         SEARCH BEANELEMENT
             WHEN    NAMEINTABLE(I) = NAMEOFBEAN
                     MOVE USERIDINTABLE(I) TO USERID
                     MOVE USERID TO USETID
                     MOVE NAMEOFBEAN TO USETNAME
                     EXEC CICS WRITEQ TD QUEUE('CSSL')
                               FROM(USERIDSETMSG)
                     END-EXEC
         END-SEARCH.
```

```
*****************************************************************
* exit with commarea set
*****************************************************************
 XIT.
         MOVE 1    TO RETURNCODE.
         MOVE ZERO TO REASONCODE.

*        EXEC CICS WRITEQ TD QUEUE('CSSL') FROM(DFHCOMMAREA)
*        END-EXEC.

         EXEC CICS WRITEQ TD QUEUE('CSSL') FROM(XITMSG)
         END-EXEC.

         EXEC CICS  RETURN
         END-EXEC.
```

Deploying the sample COBXOPUS

Here is how to deploy this sample URM:

- Download the source code COBXOPUS.
- Translate and compile COBXOPUS and place the load module in a dataset in your CICS region DFHRPL concatenation.
- Define a program definition for COBXOPUS.
- Specify COBXOPUS in the URM parameter in the relevant TCPIPSERVICE definition.

To verify the sample is functioning correctly, the command **CEMT I TAS** will display the user ID that a suspended task is running under. Tasks can be suspended using the CEDX transaction against the relevant request processor transaction, which by default is CIRP.

Testing the sample COBXOPUS

If the sample DFHXOPUS user exit functions correctly, you should see the following messages displayed in the CICS CSMT log. The messages shown in Example A-2 were produced when testing the URM with the Hello World IVP enterprise bean documented in Section 4.3.1, “Running the IVP OS/390 USS client application” on page 94.

Example: A-2 COBXOPUS messages written to CSMT log

```
COBXOPUS ENTRY; TRAN:CIRR BEAN:HelloWorld
USERID SET TO: CICSRS1  FOR BEAN NAME: HelloWorld
COBXOPUS EXIT
CICS EJB hello world sample called with string: Hello from CICS EJB IVP client
```

B

The COBOL Trader application

This appendix describes the 3270 COBOL Trader application used as the basis of the enterprise bean examples provided in Part 3 on page 133 of this redbook.

We start by providing a description of how the original 3270 based version of the Trader application functions. We then provide a summary of what definitions are required when installing the Trader application in your CICS system.

To obtain the sample COBOL Trader application and accompanying JCL, refer to Appendix C, “Using the additional material” on page 315. Note along with the 3270 version of Trader, we also supply a Web-enabled version for use with CICS Web support, and the CICS Transaction Gateway.

The 3270 Trader COBOL application

Trader, written in COBOL, uses the VSAM access method for file access and the CICS 3270 BMS programming interface. It is a pseudo-conversational application, meaning that a chain of related non-conversational CICS transactions is used to convey the impression of a "conversation" to the users as they go through a sequence of screens that constitute a business transaction. The application consists of two modules: TRADERPL, which contains the 3270 presentation logic; and TRADERBL, which contains the business logic. TRADERPL invokes TRADERBL using an EXEC CICS LINK and passing a COMMAREA structure for input and output. TRADERBL contains logic to query and write to the persistent VSAM data, stored in two files the *company file* and the *customer file*.

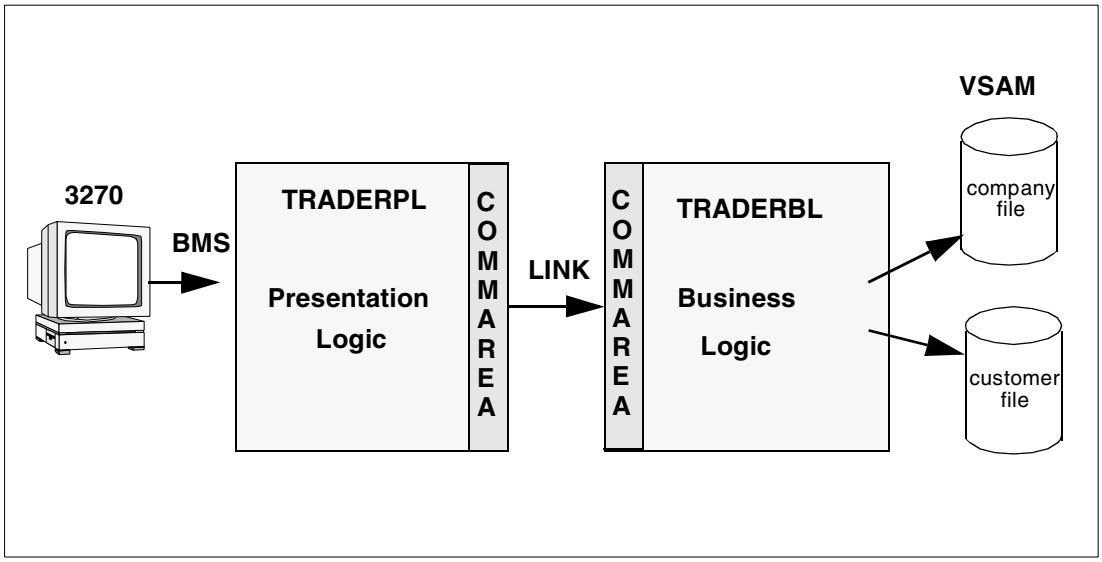


Figure B-1 Trader application structure

At each step, the application presents a set of options. The user makes a choice, then presses the required key in order to send their selections back to the application. The application performs the necessary actions based on the user's choice and presents the results together with any possible new options. The application has a strict hierarchical menu structure which allows the user to return to the previous step by using the PF3 key.

3270 application flows

In this section we describe a typical business transaction when using the 3270 Trader application:

1. The program TRADERPL is invoked on a 3270 capable terminal by entering the initial CICS transaction identifier (TRAD). TRADERPL calls TRADERBL, passing an inter-program COMMAREA of 400 bytes. TRADERBL expects the COMMAREA to contain a request type and associated data. There are three request types: *Get_Company* to return a company list, *Share_Value* to return a list of share values, or *Buy_Sell* to buy or sell shares. In this step the request type is *Get_Company*.

   When TRADERBL receives a *Get_Company* request, it browses the company file and returns the first four entries to TRADERPL. At this point the user has not entered any request, but the application assumes that a Get_Company request will be following. TRADERPL then sends the *signon display* (T001 shown in Figure B-2), which prompts for a userid and password. The list of companies is stored in the COMMAREA associated with the terminal when the TRAD transaction ends, so that it will be available at the next task in the pseudo-conversational sequence.

```
                          Share Trading Demonstration              TRADER.T001

                          Share Trading Manager: Logon


                     Enter your User Name:


                     Enter your Password:


    ------------------------------------------------------------------------------
    PF3=Exit                                                          PF12=Exit
```

Figure B-2 Trader signon display

2. The next transaction invokes TRADERPL, which receives the *signon display* (T001) and the saved COMMAREA from step 1. Using the company data acquired in step 1, TRADERPL sends the *company selection display* (T002), the format of which is shown in Figure B-3. TRADERPL then returns, specifying the next transaction to run and the associated COMMAREA.

```
                       Share Trading Demonstration              TRADER.T002

                       Share Trading Manager: Company Selection


                            1. Casey_Import_Export

                            2. Glass_and_Luget_Plc

                            3. Headworth_Electrical

                            4. IBM


                     Please select a company (1,2,3 or 4) :


    ------------------------------------------------------------------------------
    PF3=Return                                                        PF12=Exit
```

Figure B-3 Company selection display

3. The user selects the company to trade from the *Company Selection* display, and presses Enter. The program TRADERPL is invoked and sends the *Options display* (T003, shown in Figure B-4) to the terminal. The user can now decide whether to buy, sell, or get a new real-time quote. TRADERPL returns, specifying the next transaction to run and the associated COMMAREA.

```
                        Share Trading Demonstration              TRADER.TOO3

                        Share Trading Manager: Options


                         1. New Real-Time Quote

                         2. Buy Shares

                         3. Sell Shares


                     Please select an option (1,2 or 3):


    ------------------------------------------------------------------------
    PF3=Return                                                      PF12=Exit
```

Figure B-4 Options menu display

4. The user then selects option **1** and presses the Enter key. TRADERPL is invoked and determines that the user's request is a *Share_Value* request type. TRADERPL calls TRADERBL, passing the request type and the company selected earlier. TRADERBL reads the customer file to determine how many shares are held, then reads the company file to determine the price history, and returns the information to TRADERPL. TRADERPL uses this data to build a *Real-Time Quote* display (T004) as illustrated in Figure B-5. This display shows the recent history of share values for the company chosen, the number of shares held with this company, and the total value of these shares. TRADERPL returns, specifying the next transaction to run and the associated COMMAREA data.

```
                     Share Trading Demonstration              TRADER.TOO4

                     Share Trading Manager: Real-Time Quote

          User Name:     TRADER

          Company Name:  IBM

          Share Values:                   Commission Cost:
             NOW:        00163.00            for Selling:        015
             1 week ago: 00157.00            for Buying:         010
             6 days ago: 00156.00
             5 days ago: 00159.00
             4 days ago: 00161.00
             3 days ago: 00160.00
             2 days ago: 00162.00         Number of Shares Held: 0100
             1 day ago:  00163.00         Value of Shares Held:  000000000.00



    ------------------------------------------------------------------------
    PF3=Return                                                      PF12=Exit
```

Figure B-5 Real-time quote display

5. The user now presses **PF3** to go back to the *options menu*. TRADERPL is invoked and sends the *Options display* (T003) to the terminal (repeating the actions of step 3), and returns, specifying the next transaction to run and the associated COMMAREA data.
6. The user now requires to purchase shares, so selects option **2** and presses the Enter key. Program TRADERPL receives map T003 and determines that the user wants to buy shares, and sends the *Shares-Buy* display (T005) shown in Figure B-6. TRADERPL returns, specifying the next transaction to run and the associated COMMAREA.

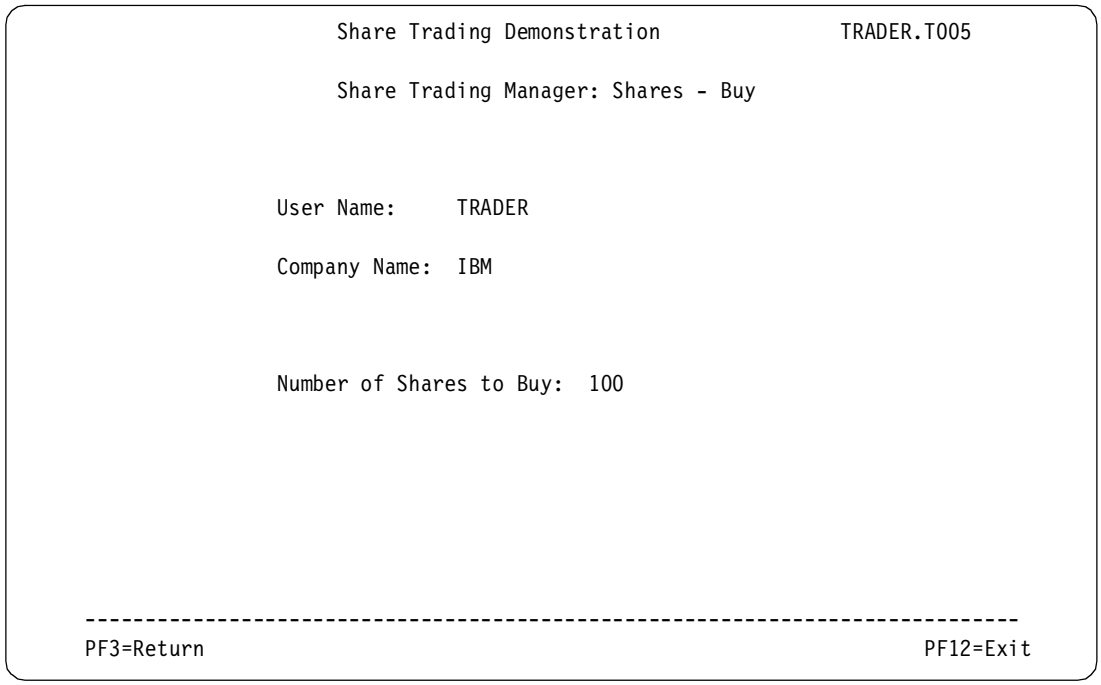


Figure B-6 Shares — Buy display

7. Program TRADERPL receives the T005 screen and builds a *Buy_Sell* request COMMAREA which is passed to program TRADERBL. TRADERBL reads the company file and then performs a READ for UPDATE and REWRITE to the customer file to update the customers share holdings. The success of the request is returned to TRADERPL in the COMMAREA, and TRADERPL sends the *Options display* (T003) reporting the successful buy to the user. TRADERPL returns, specifying the next transaction to run and the associated COMMAREA.
8. Next the user checks his share holdings by repeating step 4.
9. The user returns to the options screen by repeating step 5.
10. The business transaction is completed by the user pressing PF12, which performs an EXEC CICS SEND TEXT to write a message to the terminal reporting the session is complete. TRADERPL then executes the final EXEC CICS RETURN command. No COMMAREA is specified because the pseudo-conversation is over, and there is no conversation state data to retain.

CICS resource definitions

To install the COBOL Trader application the following CICS resources need to be created:

- Files

  Trader uses the following two VSAM files:

  - **COMPFILE**

    This file is used to store the list of companies and associated share prices. It can be created using the supplied JCL `TRADERCOCJL.TXT` which requires as input the file `TRADERCODATA.TXT`

  - **CUSTFILE**

    This file is used to store the list of users and share holdings. It can be created using the supplied JCL `TRADERCUJCL.TXT`

- Transactions

  The 3270 version of Trader requires just one transaction **TRAD**, which should specify the program TRADERPL

- Programs

  CICS program definitions are only required if program autoinstall is not active. The 3270 trader application uses two COBOL programs which will need to compiled and placed in a dataset in your CICS region DFHRPL concatenation.

  - **TRADERPL**

    This contains the 3270 presentation logic and is invoked by transaction TRAD.

  - **TRADERBL**

    This contains the business logic and is invoked by program TRADERPL

- Mapset

  Trader uses a the Mapset **NEWTRAD** which comprises the maps T001, T002, T003, T004, T005 and T006. The Mapset is supplied in the file `NEWTRADB.TXT` and will need to be assembled and placed in a dataset in your CICS region DFHRPL concatenation.

For further information on creating the resource definitions for Trader, refer to the supplied file `TRADERRDO.TXT` which contains the output of a CSD extract for the Trader application.

C

Using the additional material

This redbook contains additional material that can be downloaded from the Web.

Locating the additional material on the Internet

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG246284/SG246284src.zip

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number.

Using the Web material

The additional Web material that accompanies this redbook includes the following files in the following directories:

- **DFHXOPUS**

  This directory contains the sample DFHXOPUS URM as documented in Appendix A-1, “DFHXOPUS function” on page 304. The following files are provided:

ReadMe.txt	Readme file
COBXOPUS.txt	COBOL DFHXOPUS sample

- **HelloWorldEJB**

  This directory contains the following files for use with the HelloWorld EJB used in Chapter 6, “Developing a HelloWorld session bean for CICS” on page 135. The following files are provided in two sub-directories as follows:

  - **CSD**

    This sub-directory contains the following files:

ReadMe.txt	Readme file
dfhcsdup.jcl	JCL to run DFHCSDUP
cicscsd.def	Input to DFHCSDUP to create CICS resource definitions

– **Programs**

This sub-directory contains the following files

ReadMe.txt	Readme file
hws.jar	Contains all classes, beans, and resources used as input for CICS JAR Development Tool.
hws_GEN.jar	This file is the deployed JAR file the HelloWorld bean, it is the output of the CICS JAR Development Tool.
hws_CLI.jar	This contains all classes, beans, and resources used by clients accessing the HelloWorldSession enterprise bean
hwc.jar	This contains all the classes for the stand-alone test client for the HelloWorld bean.
hwc.cmd	This is the command script used to start the stand-alone test client from Windows NT.
hwc.sh	This is the shell script used to start the stand-alone test client from USS on OS/390.
hwc_vaj.properties	This is the properties file used to call the HelloWorldSession enterprise bean from the client running in VAJ.
hwc_nt.properties	This is the properties file used to call the HelloWorldSession enterprise bean from the stand-alone client running on Windows NT.
hwc_vaj.properties	This is the properties file used to call the HelloWorldSession enterprise bean from the stand-alone client running on USS.

- **JNDIList**

This directory contains the JNDIList application we used to query our COS Naming Server. This is described further in Chapter 5, “Troubleshooting enterprise beans in CICS TS V2.1” on page 103. The following files are provided:

ReadMe.txt	Readme file
JNDIList.java	JNDIList java source
JNDIList.class	JNDIList class

- **Repository**

Note: When importing the VAJ repository file you should import the following projects: *ITSO EJB 390 Redbook* and *JCICS*. You should also select the check box **Add most recent project addition to workspace**, before you import the projects.

After importing the supplied repository you will also need to add the following features to your VAJ workspace:

– IBM EJB Development Environment
– IBM Enterprise Access Builder Library
– IBM Java Record Library
– SQLJ Runtime Library
– CICS Connector (or import the ctgclient.jar file)

This directory contains the VAJ repository file which contains all files from the VAJ *ITSO EJB OS390 Redbook* project that we developed during the course of this project. The development and use of these files is described further in the individual chapters in Part 3, “CICS TS V2.1: Enterprise bean scenarios” on page 133. The following files are provided

Readme.txt	Readme file

ITSO_EJB_390_Redbook.dat	VAJ repository file
ITSO_EJB_390_Redbook.dat.pr	This subdirectory contains the following files which are required for the SQLJ. samples:

- **TraderBackendDB2SQLJ.sqlj**
- **TraderBackendDB2SQLJ_SJProfile0.ser**

- **TraderCobol**

  This directory contains all the sample code for installing and configuring the COBOL Trader sample, and also the Web-enabled CICS Web support and servlet version used in the previous redbooks *A Performance Study of Web Access to CICS, SG24-5748*, and *Workload Management for Web Access to CICS, SG24-6133*. The structure of the following sub-directories is documented in the **ReadMe.txt** file:

  – **Cobol**

  This sub-directory contains the following COBOL files

traderbl.txt	TRADERBL application
tradercv.txt	Trader CWS converter
traderpl.txt	TRADERPL application
tradwbsr.txt	Trader TS state management program
deltsqs.txt	TS deletion program
commarea.txt	The COMMAREA data structure of TRADERBL
company.txt	Data structure representing a record in VSAM file COMPFILE
companyKey.txt	Data structure representing the key of VSAM file COMPANY
customer.txt	Data structure representing a record in VSAM file CUSTOMER
customerKey.txt	Data structure representing the key of VSAM file CUSTOMER

  – **Html**

  This directory contains the following files

tradbann.html	Trader CWS HTML template
tradcomp.html	Trader CWS HTML template
tradend.html	Trader CWS HTML template
tradfoot.html	Trader CWS HTML template
tradhead.html	Trader CWS HTML template
tradqbs.html	Trader CWS HTML template
tradsign.html	Trader CWS HTML template

  – **Setup**

  This directory contains the following files

newtradb.txt	Mapset for trader 3270 application
tradercodata.txt	Trader company file data
tradercojcl.txt	JCL for creating company file
tradercujcl.txt	JCL for creating customer file
traderrdo.txt	RDO definitions for Trader

  – **Servlet**

  This directory contains the following files

StateData.java	Trader servlet Java source
trader.java	Trader servlet Java source
traderb2.java	Trader servlet Java source
TraderBase.java	Trader servlet Java source
TraderCommarea.java	Trader servlet Java source
StateData.class	Trader servlet Java class
trader.class	Trader servlet Java class
traderb2.class	Trader servlet Java class

TraderBase.class Trader servlet Java class
TraderCommarea.class Trader servlet Java class

- **TraderEJB**

  This directory contains all the files for the EJB implementation of Trader

  - **DB2**

    This directory contains files contain all the DB2 commands to setup the Trader database on OS/390.

    ReadMe.txt Readme file

    Create.db2 DB2 commands to create a DB2 database, table space and tables.

    LoadDB.db2 DB2 commands to insert data into the table ITSOEJB.TRADER_COMPANY.

    GrantPrivileges.db2 DB2 commands to grant privileges to PUBLIC and to user CICSRS1.

  - **JDBC**

    This directory contains all the files necessary to customize the JDBC run-time environment on OS/390.

    ReadMe.txt Readme file
    db2genJDBC.sh Shell script to run the DB2 utility db2genJDBC from USS
    db2jdbc.cursors Cursor property file
    DSNJDBC_JDBCProfile.ser JDBC profile
    dsntjjcl.jcl Bind job to bind JDBC DBRMs
    mydb2sqljjdbc.properties Run-time properties file for the DB2 for OS/390 SQLJ/JDBC driver
    dsnjdbc.dbrm DBRM for isolation level UR
    dsnjdbc.dbrm DBRM for isolation level CS
    dsnjdbc.dbrm DBRM for isolation level RS
    dsnjdbc.dbrm DBRM for isolation level RR

  - **JSP**

    This directory contains the following JSP files for the servlet to test trader.

    ReadMe.txt Readme file
    Buy.jsp JSP for buy dialog.
    CompanySelection.jsp JSP for company selection dialog.
    Logoff.jsp JSP to show logoff message.
    Logon.jsp JSP to show logon dialog.
    Quotes.jsp JSP to show quotes.
    Sell.jsp JSP to show sell dialog.
    TraderError.jsp JSP to show error message.

  - **Programs**

    This directory contains the following jar and cmd files for the TraderBean.

    ReadMe.txt Readme file
    traderAll.jar This contains all classes,Java source files, beans, and resources of the complete EJB Trader application. It is not required to run the trader enterprise bean unless the VAJ repository file (`.dat`) is not used. It contains no VAJ specific information such as project and names and EJB groups (which are contained in the `.dat` file)

traderForCICS.jar	This is the undeployed Jar file ready for deployment to any enterprise Java server. It contains all classes, beans, and resources for the trader enterprise bean.
traderForWAS.jar	Contains all classes, beans, and resources used as input for WebSphere deployment.
traderForCICS_GEN.jar	This is the deployed Jar file for use in CICS TS V2.1, it is the output of the CICS JAR development tool. If using SQLJ, you should note this contains an uncustomized serialized profile.
traderCLI.jar	This contains all classes, beans, and resources used by clients accessing EJB Trader.
traderServlet.jar	This contains all servlet related classes, beans, and resources.
traderTest.jar	This contains the TraderTest class for the stand-alone test client program (runTest.cmd).
runTest.cmd	This is the stand-alone test client program used to run the trader bean from the Windows command line.

– **SQLJ**

This directory contains the following files necessary to prepare the SQLJ application.

ReadMe.txt	Readme file
bindtrad.jcl	Bind job to bind trader DBRMs.
db2profc.sh	Shell script to run the DB2 utility db2profc from USS.
mydb2sqljjdbc.properties	Run-time properties file for the DB2 for OS/390 SQLJ/JDBC driver.
trader1.dbrm	DBRM for isolation level UR.
trader1.dbrm	DBRM for isolation level CS.
trader1.dbrm	DBRM for isolation level RS.
trader1.dbrm	DBRM for isolation level RR.
traderForCICS_GEN_uncustomized.jar	This file is used as input for shell script db2profc.sh. It is the output of the CICS JAR development tool. It contains an uncustomized serialized profile.
traderForCICS_GEN.jar	This file is used as input for CICS deployment of the SQLJ version. It is the output of the shell script db2profc.sh. It contains the customized SQLJ profile.

System requirements for downloading the Web material

The following system configuration is recommended for downloading the additional Web material.

Hard disk space: 3 MB minimum
Operating System: Windows NT, or 2000.
Processor: Intel 486 or higher
Memory: 128 MB

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material Zip file into this folder. This will create the six sub-directories as documented in this appendix, and the readme files with further instructions.

Special notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere.,The Power To Manage., Anything. Anywhere.,TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjøbenhavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

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

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

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

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

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

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 324.

- *Enterprise JavaBeans for z/OS and OS/390 WebSphere Application Server V4.0,* SG24-6283
- *EJB Development with VisualAge for Java for WebSphere Application Server,* SG24-6144
- *CICS Transaction Gateway V3.1, The WebSphere Connector for CICS,* SG24-6133

Other resources

These publications are also relevant as further information sources:

- Horswill et al., *Designing and Programming CICS Applications*, O’Reilly, ISBN 1-56592-676-5, SR23-9692
- Bainbridge et al., *CICS and Enterprise JavaBeans,* IBM Systems Journal Vol. 40, No. 1, 2001.
- Vlada Matena & Mark Hapner*, Enterprise JavaBeans Specification, V1.1*, Sun Microsystems, available from:

  http://www.javasoft.com/products/ejb

Referenced Web sites

These Web sites are also relevant as further information sources:

- http://www-4.ibm.com/software/ts/cics/library/books/zos/
  CICS TS V2.1 on-line library, where you can find all the CICS TS V2.1 manuals referenced in this redbook
- http://www-4.ibm.com/software/ts/cics/library/infocenter/
  CICS TS V2.1 Information Center
- http://www.ibmlink.ibm.com/usalets&parms=H_201-060
  CICS TS V2.1 announcement letter
- http://www.ibm.com/software/ts/cics/txppacs/
  CICS SupportPacs
- http://www-4.ibm.com/software/webservers/appserv/library_390.html
  WebSphere Application Server for OS/390 and z/OS library
- http://www.ibm.com/software/webservers/appserv/efix.html
  WebSphere Application Server FixPack downloads
- http://www.ibm.com/software/webservers/appserv/library.html
  WebSphere Application Server library

- http://java.sun.com/
  Sun Java Web site
- http://java.sun.com/products/jdbc/index.html
  Sun JDBC Web site
- http://www.software.ibm.com/data/db2/os390/jdbc.html
  DB2 for OS/390 Java Database Connectivity
- http://www.sqlj.org
  Information on SQLJ set of standards.
- http://www.software.ibm.com/data/db2/java/sqlj
  Java enablement with DB2

How to get IBM Redbooks

Search for additional Redbooks or redpieces, view, download, or order hardcopy from the Redbooks Web site:

ibm.com/redbooks

Also download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

Redpieces are Redbooks in progress; not all Redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Abbreviations and acronyms

AOR	Application-owning region
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AWT	Abstract windowing toolkit
BMP	Bean-managed persistence
CCF	Common Connector Framework
CMP	Container managed persistence
CORBA	Component Object Request Broker Architecture
CTG	CICS Transaction Gateway
CWS	CICS Web support
DBMS	Database management system
DNS	Domain Name System
DPL	Distributed program link
EAB	Enterprise Access Builder
EBCDIC	Extended Binary Coded Decimal Interchange Code
ECI	External Call Interface
EJB	Enterprise JavaBeans
EJS	Enterprise Java Server
EPI	External Presentation Interface
ESI	External Security Interface
ESM	External Security Manager
EXCI	External CICS Interface
FOR	File-owning region
FTP	File Transfer Protocol
GIOP	General Inter-ORB Protocol
GUI	Graphical user interface
HFS	Hierarchical File System
HTML	Hypertext Transfer Protocol
HTTP	Hypertext Markup Language
IDE	Integrated development environment
IDL	Interface definition language
IIOP	Internet Inter-ORB Protocol
IOR	Interoperable object reference
ISC	Inter-system communication
J2EE	Java 2 Enterprise Edition
JAR	Java archive
JDBC	Java Database Connectivity
JDK	Java Developer's Kit
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JPDA	Java Platform Debugger Architecture
JSDK	Java Servlet Development Kit
JSP	JavaServer Page
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
LPAR	Logical Partition
LUW	Logical unit of work
OMG	Object Management Group
OS/390	Operating System 390
OTS	Object transaction service
PB	Persistence Builder
RDBMS	Relational database management system
RMI	Remote Method Invocation
RPC	Remote procedure call
SDK	Software Development Kit
SQL	Structured query language
SQLJ	SQL Java
SSL	Secure socket layer
TCB	Task control block
TCP/IP	Transmission Control Protocol/Internet Protocol
UML	Unified Modeling Language
UOW	Unit of work
URL	Uniform resource locator
USS	Unix System Services
VAJ	VisualAge for Java
WTE	WebSphere Test Environment
XMI	XML metadata interchange
XML	eXtensible Markup Language

Index

F

G

H

I

J

K

L

M

N

O

X

Z

Redbooks

Enterprise JavaBeans for z/OS and OS/390 CICS Transaction Server V2.1

(0.5” spine)
0.475”<->0.873”
250 <-> 459 pages

Enterprise JavaBeans for z/OS and OS/390

CICS Transaction Server V2.1

Redbooks

Understand the CICS EJB Server and how to deploy enterprise beans

Integrate enterprise beans with your COBOL applications

Develop session beans in VisualAge for Java

In this IBM Redbook, we first provide an introduction to both EJB and the way it has been implemented within the CICS architecture. We also include a summary of the different configurations in which servlets and enterprise beans can be used to access CICS applications.

Following this, we document how to set up and configure a CICS region to support enterprise beans, how to use the various new tools and features required, and how to deploy and test the product samples. Then we provide information on how to diagnose and fix problems when deploying and testing enterprise beans in CICS.

Finally, we document five scenarios in which we developed enterprise beans and deployed them to CICS. We start with the initial step of creating a simple HelloWorld session bean using the VisualAge for Java Development environment, and then move on to creating a stateful session bean called TraderBean that wraps the existing pseudo-conversational COBOL Trader application.

Following this, we provide details on how to develop new Java versions of COBOL applications using either the JCICS classes, or the SQLJ and JDBC interfaces. We also provide details on how we developed a sample JSP/servlet application to invoke the TraderBean and information on how to deploy this in WebSphere Application Server for Windows NT and WebSphere Application Server for OS/390.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-6284-00 ISBN 0738423157