

Introduction to the J2EE Connector Architecture

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

1. Getting started	2
2. Introduction	3
3. JCA's infrastructure	7
4. Common Client Interface	13
5. Sample resource adapter	16
6. Sample application.....	27
7. Wrapup and resources	29

Section 1. Getting started

What is this tutorial about?

This tutorial provides an overview of the Java 2 Enterprise Edition (J2EE) Connector Architecture (JCA). The tutorial starts with a high-level look at JCA, encompassing its place in the J2EE architecture, how it works to integrate enterprise-level systems, and the base elements of the architecture. In the sections that follow, you'll explore each of these elements in more detail, with step-by-step descriptions and examples. The course closes with a look at a sample application that will help you see how all the parts of a JCA-compliant and enabled system work together.

Should I take this tutorial?

To get the most from the tutorial, you should be familiar with Java programming and object-oriented programming concepts. You should also have a high-level understanding of J2EE and J2EE applications.

Tools, code samples, and installation requirements

Sample code ([helloworldra.zip](#)) is provided with this tutorial and will be explained as you go along. To execute and test the sample code, you will need a J2EE application server environment that supports the J2EE Connector Architecture. The source code is supplied in standard J2EE packaging; see your J2EE application server environment for deployment details.

About the author



Willy Farrell is an e-business Architect for IBM Developer Relations Technical

Consulting (a.k.a. [The DragonSlayers](#)), providing education, enablement, and consulting to IBM Business Partners. He has been programming computers for a living since 1981, began using Java in 1996, and joined IBM in 1998. Willy holds the following technical certifications, among others: Java 2 Programmer, WebSphere Application Server Enterprise Developer, WebSphere Studio Application Developer Solution Developer, MQSeries Solutions Expert, and IBM e-business Solution Technologist. You may contact Willy at willyf@us.ibm.com.

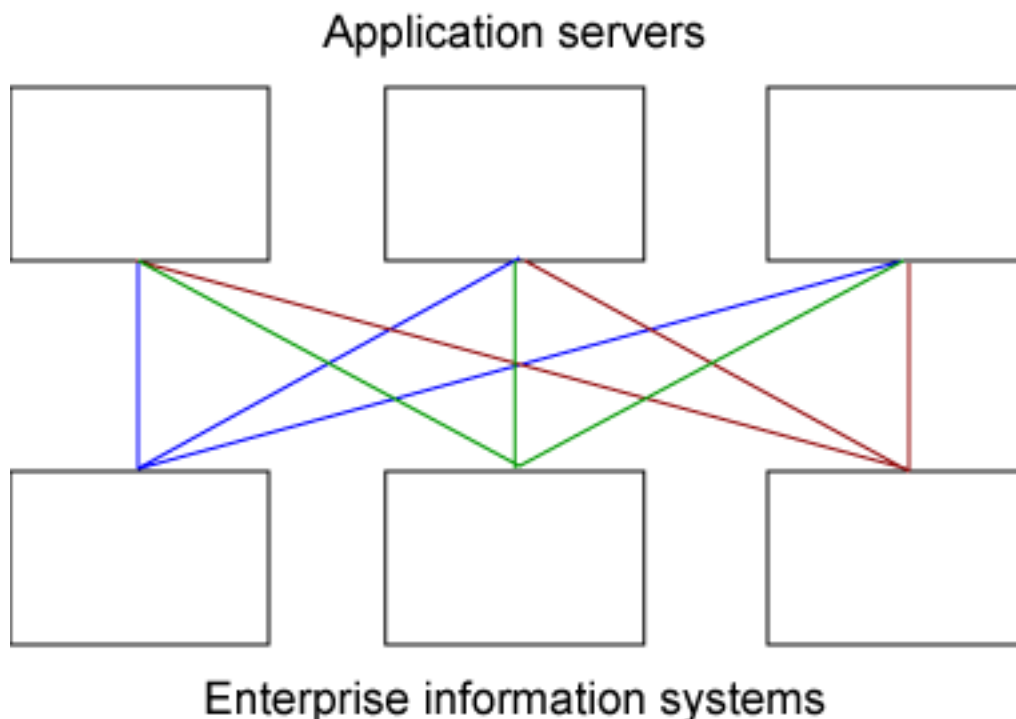
Section 2. Introduction

Integration complexity

As you develop Web-based applications, you probably find that you need to integrate those applications with the resources and data available in at least one enterprise information system (EIS), such as an enterprise resource planning (ERP) system, a supply chain management (SCM) system, or a transaction processing monitor (TPM). Such integration is the essence of e-business strategy: we leverage and transform existing infrastructure, combining it with Web and other open technologies to support new business processes, such as business-to-business (B2B) transactions.

Prior to the advent of the J2EE Connector Architecture, integrating a J2EE application and an EIS was complex and problematic, because no standard for such integration existed. Each EIS vendor supplied its own solution to the problem. An EIS vendor usually didn't support all J2EE application servers. All of this made it difficult to write truly portable applications that integrated with enterprise information systems; a custom effort was required to integrate each application server-EIS combination.

The figure below illustrates the J2EE application and EIS integration complexity.

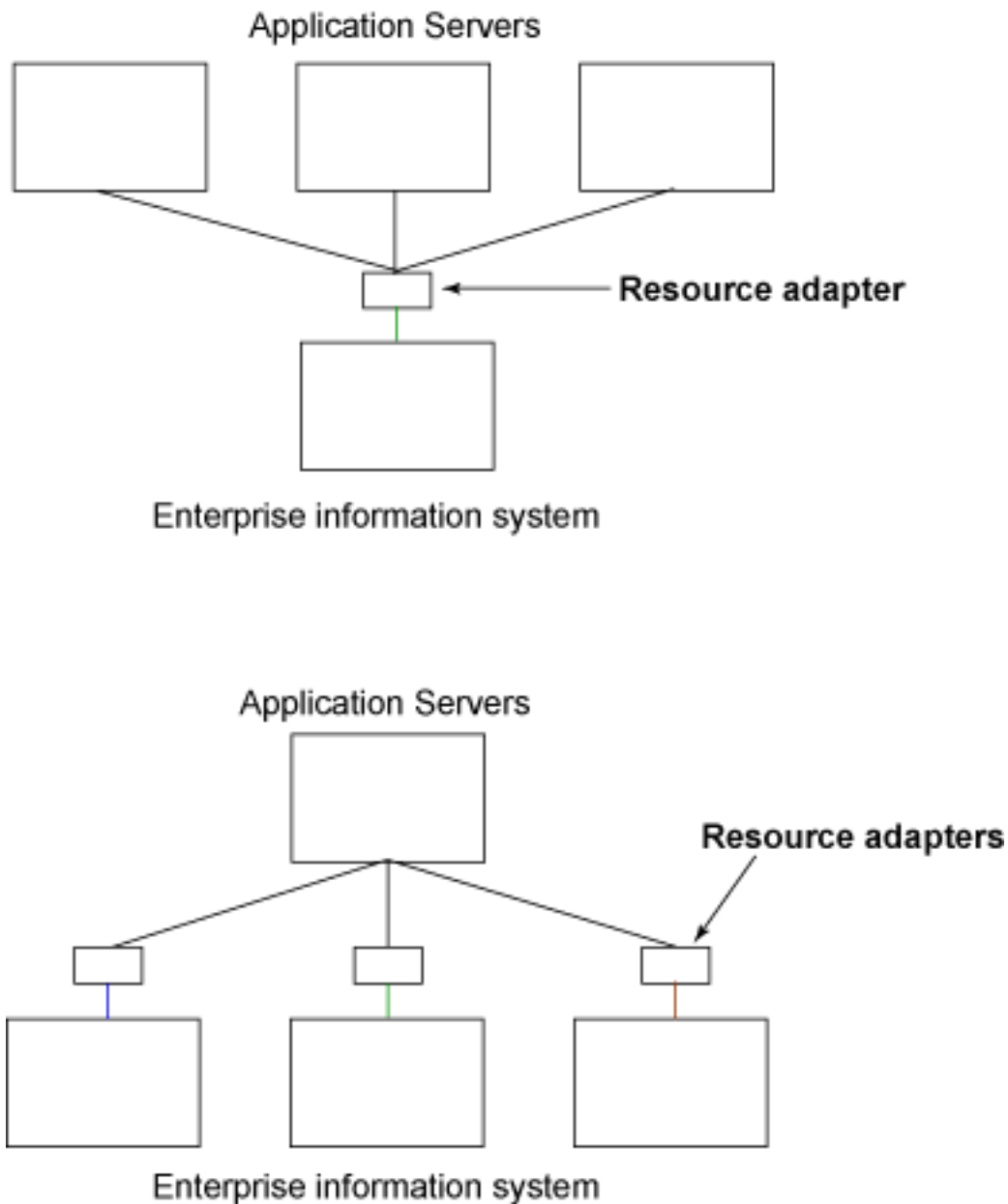


JCA simplifies integration

The J2EE Connector Architecture (JCA) resolves the problem of connecting a J2EE

application server to an EIS. By complying with the JCA standard, an EIS vendor ensures that its EIS will integrate easily with any Java-based application server. Likewise, the application-server vendor needs only to ensure its product is enabled for JCA connectivity, rather than customizing its product for every EIS on the market. Any JCA-enabled application server can integrate with any JCA-compliant EIS.

The figure below shows how JCA can simplify connecting several application servers to a single EIS, or a single application server to several EIS.



Elements of JCA

JCA is composed of three primary elements:

- System contracts
- Client API
- Resource adapter module

Each of these elements plays a specific role in JCA. We'll take a high-level look at each element separately, then move on to the more complex discussion in the next section.

System contracts

System contracts define the connection between the application server and the EIS. The EIS side of the system contract is implemented by a *resource adapter* -- a system-level software driver specific to the EIS. The application server and the resource adapter collaborate by means of the system contract to provide secure, robust, scalable access to the EIS.

Three types of system contracts are defined:

- The *connection management* contract enables physical connections to the EIS and provides a mechanism for the application server to pool those connections.
- The *transaction management* contract supports access to an EIS in a transactional context. Transactions can be managed by the application server, providing transactions that incorporate other resources besides the EIS, or they can be internal to the EIS resource manager, in which case no transaction manager is required.
- The *security* contract supports secure access to the EIS.

How system contracts are implemented on each side (application server and resource adapter) is not specified by JCA; they can be implemented as each vendor sees fit.

Client API

The second element of JCA is the *client API*. The API can be specific to the resource adapter or it can be the standard *Common Client Interface* (CCI) as defined by JCA. The CCI is meant to be used by vendors to provide integration tools and frameworks, making it easier for developers to access enterprise systems. It is recommended (but not mandated) that the resource adapter make use of the CCI.

Resource adapter module

The resource adapter module contains all of the elements necessary to provide EIS

connectivity to applications. Specifically, the resource adapter module includes the following components:

- The Java classes and interfaces that implement the resource adapter
- Any utility Java classes required by the resource adapter
- Any EIS-specific platform-dependent native libraries
- The deployment descriptor

Application servers make use of the *deployment descriptor* supplied with a resource adapter to configure it to a specific operational environment.

Resource adapter module packaging

All of the resource adapter module's files are packaged into a resource adapter archive (RAR) file using the Java archive (JAR) file format. All Java classes and interfaces are packaged in a JAR file, which is then contained by the RAR file. The native files are packaged in the RAR file, also. The deployment descriptor is named `ra.xml`, and is located in the `META-INF` folder of the RAR file.

Section 3. JCA's infrastructure

Section overview

With a basic understanding of JCA under your belt, you're ready to begin thinking more about the elements working beneath the architecture. In this section you'll learn about the classes, interfaces, and libraries that work together to create and manage connections, fulfill complex transactions, and maintain security for JCA-compliant and -enabled systems.

Obtaining and closing a connection

An application component accesses the resource adapter through `ConnectionFactory` and `Connection` interfaces, which are provided by the resource adapter implementer. These interfaces can be CCI interfaces (`javax.resource.cci.ConnectionFactory` and `javax.resource.cci.Connection`) or they can be specific to the resource adapter. The classes that implement these interfaces are also provided with the resource adapter.

The connection factory is obtained through the Java Naming and Directory Interface (JNDI) so that the application need have no knowledge of the underlying implementation class. Once the connection factory is retrieved, a connection is obtained from the connection factory through the `getConnection()` method. At least one `getConnection()` method must be provided by the connection factory, though more may be provided. It is important to note that a `Connection` is an application-level handle to an underlying physical connection to the EIS, represented by a `ManagedConnection`, which we'll discuss later.

Once the application component is finished with the connection, it calls the `close()` method on the connection. A resource adapter is required to provide a method on the connection to close the connection. This method must delegate the `close` to the `ManagedConnection` that created the connection handle. Closing a connection handle should not close the physical connection to the EIS.

Connection management

The `getConnection()` method in the connection factory does not actually create a connection; it calls the `allocateConnection()` method on its associated `ConnectionManager`. The `ConnectionManager` interface is implemented by the application server. It is associated with a connection factory in an implementation-specific manner when the connection factory is instantiated. The call to the `ConnectionManager` allows the application server to "hook in" to the resource adapter functionality to provide pooling, transaction, and security services. A `ConnectionRequestInfo` object may be passed to `allocateConnection()` to pass connection request-specific information.

The `ConnectionManager`, in turn, calls on a `ManagedConnection` to obtain the connection handle. The connection handle is passed back through the `ConnectionManager` to the connection factory and on to the application component.

Connection pooling

To support connection pooling, the resource adapter provides a class that implements the `ManagedConnectionFactory` interface. The `ManagedConnectionFactory` class acts as a factory for both connection factory instances and `ManagedConnection` instances.

When the application server needs to create a connection factory, it calls the `createConnectionFactory()` method, passing in an instance of `ConnectionManager`; this is the instance that is called when the application component calls `getConnection()` on the connection factory, as discussed previously.

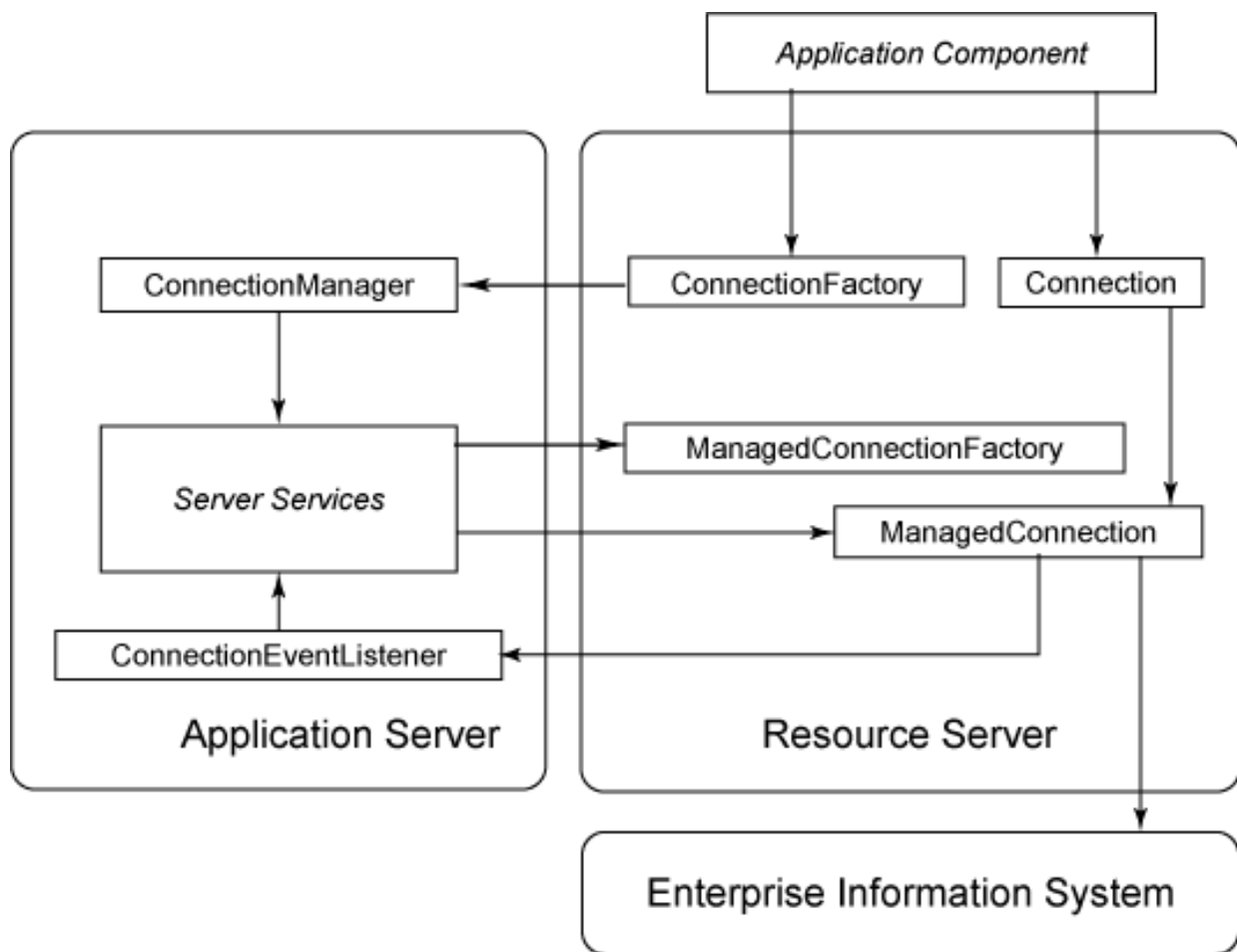
The application server uses one of two `ManagedConnectionFactory` methods to create or request managed connections. When the server requires a *new* instance of `ManagedConnection`, it calls the `createManagedConnection()` method. When the server wants to re-use an *existing* `ManagedConnection`, it calls the `matchManagedConnections()` method. To find the right match, the server passes in a set of `ManagedConnections` that could possibly meet the criteria of the requested connection, along with information about the desired connection type. Internally, the `matchManagedConnection()` method compares this information with the candidate set to determine if a match can be made. If so, it returns the matching `ManagedConnection`; if not, it returns `null`.

Additional connection pooling methods

The application server uses a number of additional `ManagedConnection` methods to facilitate connection pooling, as follows:

- The `cleanup()` method cleans up any client-specific state maintained by a given `ManagedConnection`, and invalidates all connection handles associated with that connection.
- The `destroy()` method is called when the server no longer needs the `ManagedConnection` in the pool; calling `destroy()` initiates the shutdown of the physical connection to the EIS.
- The `addConnectionEventListener()` method allows the application server to register a `ConnectionEventListener` with the `ManagedConnection`, so that the application server can be notified of close, error, and transaction occurrences. The `ManagedConnection` interface also provides a `removeConnectionEventListener()` method.

This figure shows the connection pooling process:



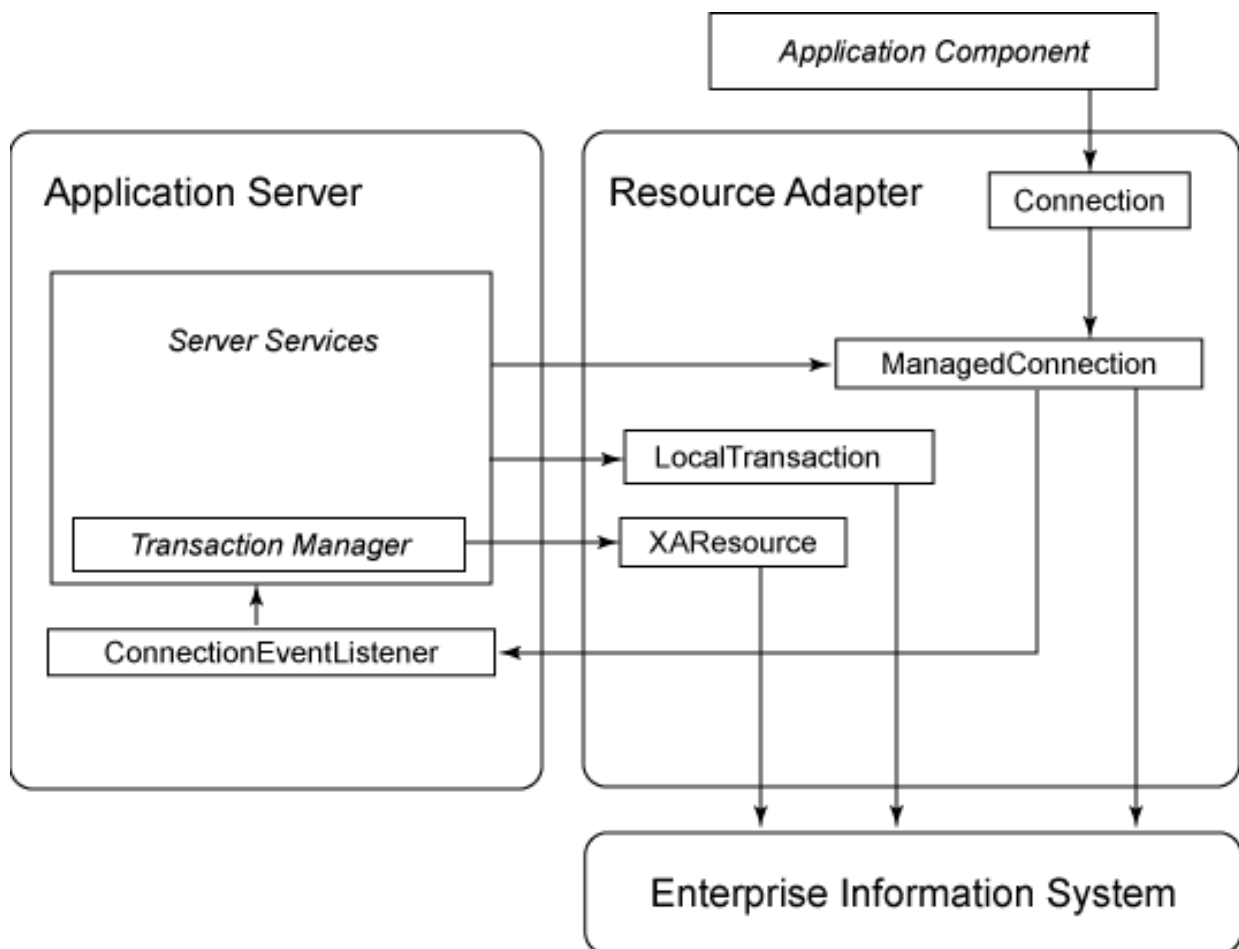
Transaction management

Two types of transaction management are available in JCA. The first type involves a transaction manager coordinating the activity of multiple resource managers across a single transaction. The second type involves a transaction with only a single resource manager, called a *local transaction*. A resource adapter can also indicate, through its deployment descriptor, that it does not support transactions.

Multiple resource managers participating in the same transaction are supported by the Java Transaction API (JTA) `XAResource` interface. This interface allows a transaction manager to manage transactions among multiple resources that support the interface. The `ManagedConnection` interface contains a method, `getXAResource()`, which returns an `XAResource` object. The application server's transaction manager uses this object to manage the transaction. For more information on `XAResource`, see the JTA specification.

Supporting local transactions

To support local transactions, the `ManagedConnection` interface provides a `getLocalTransaction()` method, which returns a `LocalTransaction` object. The `LocalTransaction` interface has methods on it to begin, commit, and roll back a transaction for the underlying EIS. In addition, the `ManagedConnection` must notify its registered `ConnectionEventListener`s when an application component begins, commits, or rolls back a transaction. The `ConnectionEventListener` interface provides `localTransactionStarted()`, `localTransactionCommitted()`, and `localTransactionRolledback()` methods for this purpose. This figure shows this process:



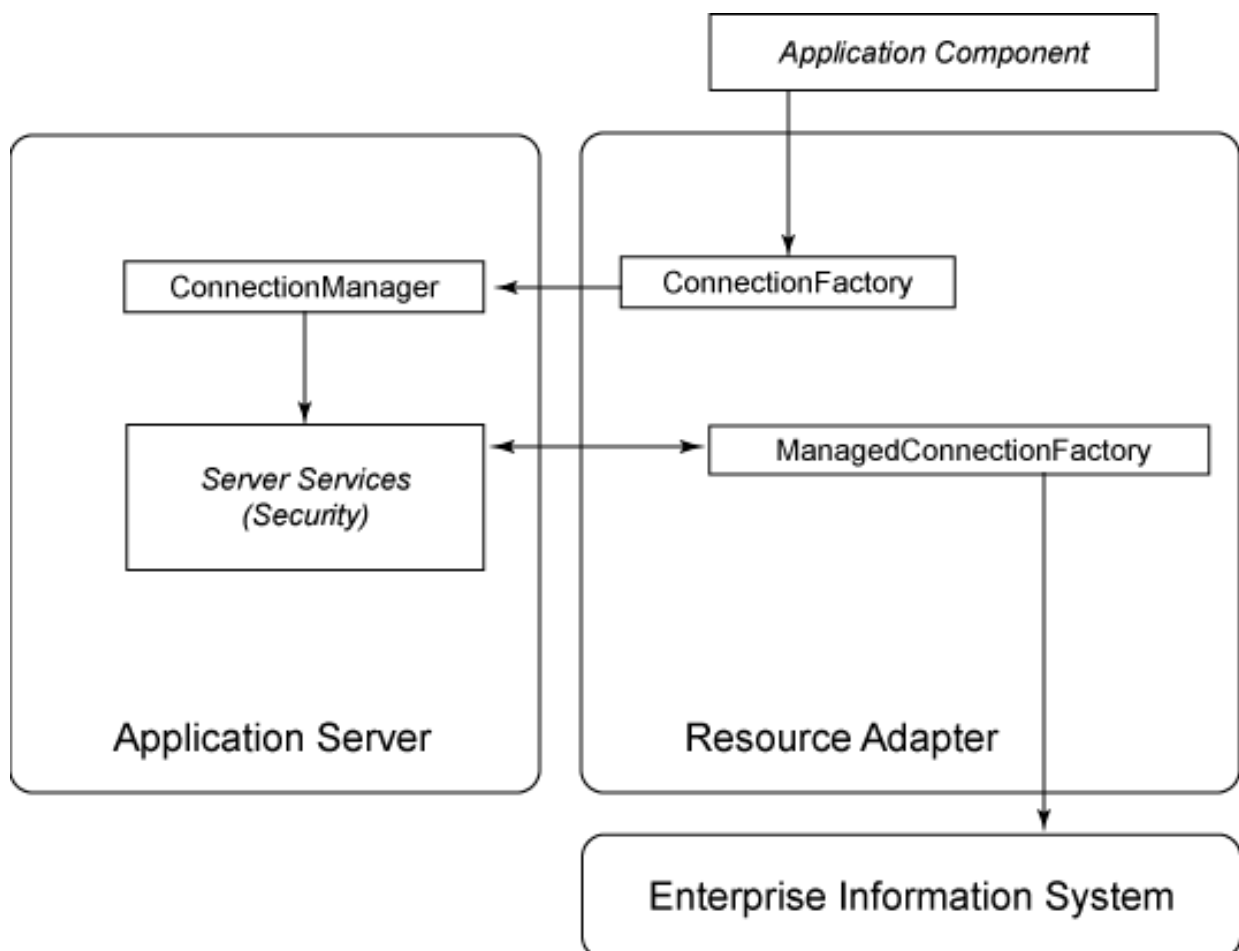
While the `getXAResource()` and `getLocalTransaction()` methods, along with the `ConnectionEventListener` notifications, are all that is required for the resource adapter to support transactions, transaction management is complex, requiring considerable collaboration between the resource adapter and the application server. A detailed understanding of JTA, the Java Transaction Service (JTS), and JCA's transaction-management contract is required to effectively provide transaction support to an EIS. See [Resources](#) on page 29 for further reading on these topics.

Security

JCA's security contract uses the Java Authentication and Authorization Service (JAAS) `Subject` class to provide security information. When a new `ManagedConnection` is created, the `createManagedConnection()` method is passed a `Subject` instance. The connection uses the `Subject` when it attempts to sign-on to the EIS. A `Subject` contains information about the `Principal`, or name, of the `Subject`, along with information about the security credentials held by the principal.

JCA defines two types of credentials. A `GenericCredential` is a Java wrapper, representing a specific security mechanism, such as a Kerberos credential. A `PasswordCredential` holds username and password information. An application server must provide implementations of both of these interfaces.

The resource adapter's deployment descriptor lists the type of security supported, the authentication mechanism used, the interface of the credentials supported, and whether re-authentication is supported. The `getConnection()` method of `ManagedConnection` takes a `Subject` as a parameter to support re-authentication. The following figure shows the security management process.



Because only a couple of methods on the resource adapter are concerned with security, security management seems somewhat simple. Like transaction management, however, the complexity is in the collaboration between the application server and the resource adapter. A comprehensive understanding of JAAS and the JCA security contract is required to ensure effective security. See [Resources](#) on page 29 for more information on JAAS.

ManagedConnectionMetaData interface

While not specifically mentioned in JCA's three system contracts, the `ManagedConnectionMetaData` interface is an important component of JCA. It provides methods to retrieve information about the `ManagedConnection` and the connected EIS. This information includes:

- The product name of the EIS
- The product version of the EIS
- The maximum number of connections that the EIS can support
- The user name for the connection

Section 4. Common Client Interface

Overview

The CCI is a standard client API for use by application components. It is designed to provide a base-level API for EIS access upon which higher level functionality will be built by EAI and tools vendors. The CCI is divided into five parts, each of which we'll discuss in the panels that follow. The CCI's five parts are:

- Connection-related interfaces
- Interaction-related interfaces
- Data representation-related interfaces
- Metadata-related interfaces
- Exceptions and warnings

A resource adapter is not required to provide support for the CCI. In fact, the resource adapter may define its own client API, different from the CCI.

Connection-related interfaces

The CCI uses its `ConnectionFactory` interface to get a connection handle to an EIS. The `ConnectionFactory` provides two `getConnection()` methods: one with no parameters and one that takes a `ConnectionSpec` instance as a parameter. The second `getConnection()` is used when connection request-specific information must be provided with the connection request.

The `ConnectionFactory` also provides methods to return metadata information about the resource adapter (`getResourceAdapterMetaData()`) and to return a record factory (`getRecordFactory()`).

`ConnectionSpec` is an empty interface that may be extended to add whatever properties are desired. Two standard properties, `UserName` and `Password`, are defined by JCA, but additional properties (or no properties) may be supported. The resource adapter should map the properties of `ConnectionSpec` to those of `ConnectionRequestInfo` when calling `allocateConnection()` on the `ConnectionManager`.

Connection methods

The `Connection` interface is an application-level connection handle that is used to access an EIS. It provides the following methods to manage connections between the application and the EIS:

- The `close()` method allows an application to close the connection handle, as required by the connection-management contract.

- The `getLocalTransaction()` method allows an application component to use local transactions.
- The `createInteraction()` method returns an `Interaction` object, which is used for accessing EIS functions.
- The `getResultSetInfo()` method returns information about the result set functionality supported by the EIS.
- The `getMetaData()` function returns metadata about the connection.

Interaction-related interfaces

An `Interaction` instance allows an application component to execute EIS functions. It has two `execute()` methods: one that takes an `InteractionSpec` and an input `Record` and returns an output `Record`, and one that takes an `InteractionSpec`, an input `Record`, and an output `Record`. This method executes the appropriate EIS function, as defined in `InteractionSpec`, and updates the output `Record`.

The `Interaction` must maintain its association with the `Connection` that created it, and the `getConnection()` must return that `Connection`.

The `close()` method on an `Interaction` should release all resources maintained for the `Interaction`, but should not close the `Connection`.

Standard interaction properties and values

The `InteractionSpec` interface provides properties for a specified EIS function. It defines the following standard properties:

- `FunctionName` is a string that represents the name of an EIS function.
- `InteractionVerb` is an integer that specifies the mode of interaction with the EIS. Allowable values for `InteractionVerb` are:
 - `SYNC_SEND`: The input record is sent to the EIS with no results returned.
 - `SYNC_SEND_RECEIVE`: The input record is sent to the EIS and a result is returned.
 - `SYNC_RECEIVE`: A result is synchronously retrieved from the EIS.
- `ExecutionTimeout` is the number of milliseconds to wait for the EIS to execute the

function.

Data representation-related interfaces

The `Record` interface provides for a Java representation of the data used for input or output to an EIS. `Record` has two standard properties: `RecordName` and `RecordShortDescription`. Additional properties to represent EIS record data must be defined by the implementer.

Three additional interfaces that extend `Record` are also provided. The `MappedRecord` interface provides access to the record elements in a key-value map collection. The `IndexedRecord` interface provides access to the record elements as an ordered collection. The `ResultSet` interface is based on the JDBC `ResultSet` and provides similar functionality for accessing EIS data.

The `RecordFactory` interface provides methods to create `MappedRecord` and `IndexedRecord` instances. One, both, or neither type of record may be supported by the `RecordFactory`. If neither, then the `getRecordFactory()` method of `ConnectionFactory` should throw an exception.

Metadata-related interfaces

The `ConnectionMetaData` interface provides information about the EIS name, the EIS version, and the user name, similar to the `ManagedConnectionMetaData` interface.

The `ResourceAdapterMetaData` interface provides information about the adapter name, version, vendor, description, and the version of JCA supported by the resource adapter. It also provides methods to determine the following capabilities of the resource adapter: whether `InteractionSpecs` are used; which `execute()` method variants are supported; and whether local transactions can be demarcated by application components.

Exceptions and warnings

`ResourceException` is the root of the exception hierarchy for the system contracts and for CCI. It provides a string describing the error, an error code, and a reference to another exception, which may be the lower-level problem that caused the `ResourceException`.

`ResourceWarning` provides information about warnings that have been returned by an EIS as the result of an `Interaction`. `ResourceWarnings` form a chain; a call to `getWarnings()` on `Interaction` retrieves the first warning, and the rest of the chain is accessed through that warning.

Section 5. Sample resource adapter

Section overview

In this section we'll look at a simple implementation of a JCA resource adapter. You'll find the sample resource adapter in the file `helloworldra.rar`. The Java classes are in the `helloworldra.jar` file within this RAR file. Source for all of the classes is also provided in `helloworldra.jar`.

As you can probably guess from its name, this resource adapter implements the ubiquitous "Hello World" functionality. As such, it doesn't actually connect to an enterprise system, and its functionality is limited to returning just the one `String` with the much-loved message. Although the resource adapter does not implement the transaction or security contracts, it does implement the CCI; it uses an `Interaction`, an `InteractionSpec`, a `RecordFactory`, and `IndexedRecords`.

You'll find the most important and interesting parts of the sample resource adapter extracted and explained in the discussion that follows. In addition to classes and interfaces, we'll discuss the source code for the deployment descriptor. From this simple implementation, you will be able to get some feel for the classes you will need to write and the relationships between them. In addition to studying the following two sections, you should spend some time reviewing the complete source code, which you will find in [Resources](#) on page 29 .

HelloWorldConnectionFactoryImpl class

The `HelloWorldConnectionFactoryImpl` class implements the CCI `ConnectionFactory` interface and provides the connection factory used by application components to create connections to the EIS. Following is the code to create a `HelloWorldConnectionFactoryImpl` instance, as well as the client methods that return connections, the record factory, and metadata.

Creating a connection factory instance

The `HelloWorldConnectionFactoryImpl` class's constructor requires that the `ConnectionManager` and `ManagedConnectionFactory` be passed in for use in the `getConnection()` method, as shown below:

```
...
public HelloWorldConnectionFactoryImpl(
    ManagedConnectionFactory mcf,
    ConnectionManager cm) {

    super();
    this.mcf = mcf;
    this.cm = cm;
}
...
```


Getting a connection

Neither `ConnectionRequestInfo` nor `ConnectionSpec` are supported by the `HelloWorldConnectionFactoryImpl` class, so both `getConnection()` methods do the same thing: they call the `allocateConnection()` method of the `ConnectionManager`, passing in the `ManagedConnectionFactory` and `null` for the `ConnectionRequestInfo`, as shown below:

```
...
public Connection getConnection() throws ResourceException {
    return (Connection) cm.allocateConnection(mcf, null);
}

public Connection getConnection(ConnectionSpec connectionSpec) throws ResourceException {
    return getConnection();
}
...
```

Record factory and metadata methods

The `getRecordFactory()` and `getMetaData()` methods simply return the implementation classes for the respective interfaces, as shown below:

```
...
public RecordFactory getRecordFactory() throws ResourceException {
    return new HelloWorldRecordFactoryImpl();
}

public ResourceAdapterMetaData getMetaData() throws ResourceException {
    return new HelloWorldResourceAdapterMetaDataImpl();
}
...
```

HelloWorldConnectionImpl class

The `HelloWorldConnectionImpl` class implements the CCI `Connection` interface, and provides the connection handle for application components to access the EIS. Following are the methods to create, invalidate, and close a connection, the methods to provide clients with an `Interaction` and with metadata, and the code for signalling that the class does not support local transactions and result sets.

Creating a connection

The `HelloWorldConnectionImpl` class's constructor requires that the `ManagedConnection` be passed in for use in the `close()` method. A flag is set during instantiation to indicate that this is a valid connection, that is, not closed. In other methods,

where appropriate, this flag is checked to determine if the requested action can be carried out.

```
...
public HelloWorldConnectionImpl(ManagedConnection mc) {
    super();
    this.mc = mc;
    valid = true;
}
...
```

Invalidating a connection

The `invalidate()` method sets the `ManagedConnection` reference to null and sets the flag indicating that the connection is no longer valid.

```
...
void invalidate() {
    mc = null;
    valid = false;
}
...
```

Closing a connection

The `close()` method delegates its invocation to the `ManagedConnection`, as required.

```
...
public void close() throws ResourceException {
    if (valid) {
        ((HelloWorldManagedConnectionImpl) mc).close();
    }
}
...
```

Interaction and metadata methods

The `createInteraction()` and `getMetaData()` methods simply return the implementation classes for the respective interfaces.

```
...
public Interaction createInteraction() throws ResourceException {
    if (valid) {
        return new HelloWorldInteractionImpl(this);
    } else {
        throw new ResourceException(CLOSED_ERROR);
    }
}
}
```

```
public ConnectionMetaData getMetaData() throws ResourceException {  
    if (valid) {  
        return new HelloWorldConnectionMetaDataImpl();  
    } else {  
        throw new ResourceException(CLOSED_ERROR);  
    }  
}  
...
```

Throwing a `NotSupportedException`

Because neither local transactions nor results sets are supported by this resource adapter, `getLocalTransaction()` and `getResultSetInfo()` throw `NotSupportedException`.

```
...  
public LocalTransaction getLocalTransaction() throws ResourceException {  
    throw new NotSupportedException(TRANSACTIONS_NOT_SUPPORTED);  
}  
  
public ResultSetInfo getResultSetInfo() throws ResourceException {  
    throw new NotSupportedException(RESULT_SETS_NOT_SUPPORTED);  
}  
...
```

HelloWorldManagedConnectionFactoryImpl class

The `HelloWorldManagedConnectionFactoryImpl` class implements the `ManagedConnectionFactory` interface. In the code below, you'll see how a connection factory is created, how a managed connection is created, and how the class responds to a request to match managed connections.

Creating a connection factory and a managed connection

The `createConnectionFactory()` and `createManagedConnection()` methods simply return the implementation classes for the respective interfaces. Because the security contract is not implemented by this resource adapter, the `createManagedConnection()` method does not use the `Subject` or `ConnectionRequestInfo` parameters when creating the `ManagedConnection`.

```
...  
public Object createConnectionFactory(ConnectionManager cm)  
    throws ResourceException {  
  
    return new HelloWorldConnectionFactoryImpl(this, cm);  
}  
  
public ManagedConnection createManagedConnection(  
    Subject subject,
```

```

    ConnectionRequestInfo cxRequestInfo)
    throws ResourceException {

    return new HelloWorldManagedConnectionImpl();
}
...

```

Matching managed connections

The simplicity of this resource adapter implementation makes each `ManagedConnection` indistinguishable from another. So, the `matchManagedConnections()` method simply returns the first `ManagedConnection` in the input `Set`, as shown below:

```

...
public ManagedConnection matchManagedConnections(
    Set connectionSet,
    Subject subject,
    ConnectionRequestInfo cxRequestInfo)
    throws ResourceException {

    ManagedConnection match = null;
    Iterator iterator = connectionSet.iterator();
    if (iterator.hasNext()) {
        match = (ManagedConnection) iterator.next();
    }

    return match;
}
...

```

HelloWorldManagedConnectionImpl class

The `HelloWorldManagedConnectionImpl` class implements the `ManagedConnection` interface. In the code below, you'll see how the `getConnection()`, `close()`, `cleanup()`, and `destroy()` methods work together to create, close, and clean up a connection, as well as the methods that signal that this resource adapter does not support transactions.

Creating a connection

The `getConnection()` method first invalidates an existing connection handle, if one exists, then creates a new connection handle, storing it in an instance variable. It then returns a reference to that instance variable. This maintains a one-to-one relationship between the connection handle and the `ManagedConnection`. A resource adapter may have multiple connection handles associated with a `ManagedConnection`, but is not required to do so.

```

...
public Object getConnection(
    Subject subject,
    ConnectionRequestInfo cxRequestInfo)
    throws ResourceException {

```

```
    if (connection != null) {
        connection.invalidate();
    }
    connection = new HelloWorldConnectionImpl(this);
    return connection;
}
...
```

Closing a connection

The `close()` method notifies its `ConnectionEventListeners` that the close has occurred, then invalidates the connection handle.

```
...
public void close() {

    Enumeration list = listeners.elements();
    ConnectionEvent event =
        new ConnectionEvent(this, ConnectionEvent.CONNECTION_CLOSED);
    while (list.hasMoreElements()) {
        ((ConnectionEventListener) list.nextElement()).connectionClosed(event);
    }
    connection.invalidate();
}
...
```

Cleaning up and destroying a connection

The `cleanup()` and `destroy()` methods both invalidate the connection handle. The `destroy()` method also sets the `ManagedConnection`'s instance variables to `null`, in preparation for being removed from the application server's connection pool.

```
...
public void cleanup() throws ResourceException {

    connection.invalidate();
}

public void destroy() throws ResourceException {

    connection.invalidate();
    connection = null;
    listeners = null;
}
...
```

Throwing a `NotSupportedException`

Because transactions are not supported by this resource adapter, the `getXAResource()` and `getLocalTransaction()` methods both throw `NotSupportedException`.

```
...
public XAResource getXAResource() throws ResourceException {
```

```
        throw new NotSupportedException(TRANSACTIONS_NOT_SUPPORTED_ERROR);
    }

    public LocalTransaction getLocalTransaction() throws ResourceException {

        throw new NotSupportedException(TRANSACTIONS_NOT_SUPPORTED_ERROR);
    }
    ...

```

HelloWorldInteractionImpl class

The `HelloWorldInteractionImpl` class implements the CCI `Interaction` interface. On this panel, you'll see how some of the more important elements of this class work together to create, execute, and close an `Interaction`.

Creating an Interaction

Because JCA requires that an `Interaction` maintain its association with the connection handle used to create it, the constructor requires that a `Connection` be passed in. A flag is set during instantiation to indicate that this is a valid `Interaction`, that is, not closed. In other methods, where appropriate, this flag is checked to determine if the requested action can be carried out.

```
...
public HelloWorldInteractionImpl(Connection connection) {

    super();
    this.connection = connection;
    valid = true;
}
...

```

Executing an Interaction

The `HelloWorldInteractionImpl` class only supports the `execute()` variant that takes an `InteractionSpec`, an input record, and an output record (the other `execute()` throws `NotSupportedException`). The method ensures that:

- The `Interaction` is valid
- The correct `InteractionSpec` implementation instance has been passed in
- The correct input record type has been provided
- The correct output record type has been provided

The `execute()` method

If all conditions are correct, the "Hello World!" message is placed in the output record. If a condition is not correct, then an exception is thrown.

```

...
public boolean execute(InteractionSpec ispec, Record input, Record output)
    throws ResourceException {

    if (valid) {
        if (((HelloWorldInteractionSpecImpl) ispec)
            .getFunctionName()
            .equals(HelloWorldInteractionSpec.SAY_HELLO_FUNCTION)) {
            if (input.getRecordName().equals(HelloWorldIndexedRecord.INPUT)) {
                if
                    (output.getRecordName().equals(HelloWorldIndexedRecord.OUTPUT)) {
                    ((HelloWorldIndexedRecord) output).clear();
                    ((HelloWorldIndexedRecord) output).add(
                        OUTPUT_RECORD_FIELD_01);
                } else {
                    throw new ResourceException(INVALID_OUTPUT_ERROR);
                }
            } else {
                throw new ResourceException(INVALID_INPUT_ERROR);
            }
        } else {
            throw new ResourceException(INVALID_FUNCTION_ERROR);
        }
    } else {
        throw new ResourceException(CLOSED_ERROR);
    }
    return true;
}
...

```

Closing an Interaction

The `close()` method clears the connection handle and marks the `Interaction` as invalid.

```

...
public void close() throws ResourceException {

    connection = null;
    valid = false;
}
...

```

HelloWorldInteractionSpec class

To hide as much implementation detail as possible from application components using this resource adapter, the `HelloWorldInteractionSpec` interface extends `InteractionSpec`. The `InteractionSpec` interface lets us provide all the information application components must have. In the next panel, we'll look at the implementation class for this interface.

```

...
public interface HelloWorldInteractionSpec extends InteractionSpec {

```

```
    public static final String SAY_HELLO_FUNCTION = "sayHello";

    public String getFunctionName();
    public void setFunctionName(String functionName);
}
```

HelloWorldInteractionSpecImpl class

The `HelloWorldInteractionSpecImpl` class implements the `HelloWorldInteractionSpec` interface. It has one property, `FunctionName`, with a getter and setter to access the property. The JCA specification says that the properties in implementations of `InteractionSpec` must be bound or constrained. This implementation provides the property as bound.

```
...
public String getFunctionName() {

    return functionName;
}

public void setFunctionName(String functionName) {

    String oldFunctionName = functionName;
    this.functionName = functionName;
    firePropertyChange("FunctionName", oldFunctionName, functionName);
}
...
```

HelloWorldIndexedRecord class

Like `HelloWorldInteractionSpec`, the `HelloWorldIndexedRecord` interface is used to hide implementation details from application components.

```
...
public interface HelloWorldIndexedRecord extends IndexedRecord {

    public static final String INPUT = "input";
    public static final String OUTPUT = "output";
    public static final int MESSAGE_FIELD = 0;
}
```

HelloWorldIndexedRecordImpl class

The `HelloWorldIndexedRecordImpl` class implements the `HelloWorldIndexedRecord` interface. It has two properties, `Name` and `ShortDescription`, with getters and setters to access the properties. Because this class must also implement the `List` interface, it maintains an `ArrayList` as an instance variable, and implements all `List` methods by calling the corresponding method on the `ArrayList`.


```
...
public class HelloWorldIndexedRecordImpl implements HelloWorldIndexedRecord {

    private ArrayList list = new ArrayList();
    private String name;
    private String description;
    ...
}
```

HelloWorldRecordFactoryImpl class

The `HelloWorldRecordFactoryImpl` class implements the `RecordFactory` interface. It does not support creating `MappedRecords`.

The `createIndexedRecord()` method ensures that the requested record name is valid, then creates the record and returns it. If the record name is not valid, an exception is thrown.

```
...
public IndexedRecord createIndexedRecord(String recordName)
    throws ResourceException {

    HelloWorldIndexedRecordImpl record = null;

    if ((recordName.equals(HelloWorldIndexedRecord.INPUT))
        || (recordName.equals(HelloWorldIndexedRecord.OUTPUT))) {
        record = new HelloWorldIndexedRecordImpl();
        record.setRecordName(recordName);
    }
    if (record == null) {
        throw new ResourceException(INVALID_RECORD_NAME);
    } else {
        return record;
    }
}
...
```

The deployment descriptor

The deployment descriptor provides the fully qualified names for the following JCA components:

- The `ManagedConnectionFactory` implementation class
- The `ConnectionFactory` interface
- The `ConnectionFactory` implementation class
- The `Connection` interface
- The `Connection` implementation class

It also indicates that neither transactions nor re-authentication are supported. If the transaction and security contracts are supported, the deployment descriptor will contain additional elements.

```
< !DOCTYPE connector PUBLIC "-//Sun Microsystems, Inc.//
  DTD Connector 1.0//EN" 'http://java.sun.com/dtd/connector_1_0.dtd'>

<connector>
  <display-name>Hello World Sample</display-name>
  <vendor-name>Willy Farrell</vendor-name>
  <spec-version>1.0</spec-version>
  <eis-type>Hello World</eis-type>
  <version>1.0</version>
  <resourceadapter>
    <managedconnectionfactory-class>
      com.ibm.ssysa.helloworldra.HelloWorldManagedConnectionFactoryImpl
    </managedconnectionfactory-class>
    <connectionfactory-interface>
      javax.resource.cci.ConnectionFactory
    </connectionfactory-interface>
    <connectionfactory-impl-class>
      com.ibm.ssysa.helloworldra.HelloWorldConnectionFactoryImpl
    </connectionfactory-impl-class>
    <connection-interface>
      javax.resource.cci.Connection
    </connection-interface>
    <connection-impl-class>
      com.ibm.ssysa.helloworldra.HelloWorldConnectionImpl
    </connection-impl-class>
    <transaction-support>
      NoTransaction
    </transaction-support>
    <reauthentication-support>
      false
    </reauthentication-support>
  </resourceadapter>
</connector>
```

Section 6. Sample application

Section overview

You should now have a fairly comprehensive understanding of how JCA's components work together to create and manage complex interactions with an enterprise system. All that's left to do is play with the code itself. You will find a sample application that uses the sample resource adapter in the file `helloworldra.ear`. It is a J2EE Web application with an input HTML form (with only a Submit button), a servlet controller, a JavaBeans component that invokes the resource adapter, and a JavaServer Pages (JSP) component to display the results of the invocation.

We'll close this tutorial with a look at the bean that invokes the resource adapter, as well as another class (provided separately from the `.ear` file) that helps deploy an `InteractionSpec` instance into JNDI.

HelloWorldBean class

This `HelloWorldBean` class, through its `execute()` method, invokes the resource adapter, using the CCI interface. First, the `ConnectionFactory` is retrieved from JNDI and is used to create a `RecordFactory`. The `RecordFactory` is used to create an input record and an output record.

Next, an `InteractionSpec` is retrieved from JNDI. Then a `Connection` is created from the `ConnectionFactory` and an `Interaction` is created from the `Connection`. The `Interaction` is used to execute the function, and the results are stored in the bean's `Message` property.

You will observe that the two JNDI lookups are different. The lookup of the `ConnectionFactory` is done using the `java:comp/env` context, while the lookup of the `InteractionSpec` is not. This is because the environment that was used to develop and test the resource adapter and sample application did not have a deployment tool to bind an `InteractionSpec` into JNDI; it only provided for binding the `ConnectionFactory`, which could then be accessed using a resource reference in the Web application.

```
...
public void execute() throws NamingException, ResourceException {
    InitialContext context = new InitialContext();
    ConnectionFactory cxFactory =
        (ConnectionFactory) context.lookup("java:comp/env/HelloWorld");
    RecordFactory recordFactory = cxFactory.getRecordFactory();
    IndexedRecord input =
        recordFactory.createIndexedRecord(HelloWorldIndexedRecord.INPUT);
    IndexedRecord output =
        recordFactory.createIndexedRecord(HelloWorldIndexedRecord.OUTPUT);
    InteractionSpec ispec =
```

```
(InteractionSpec) context.lookup("jca/HelloWorldISpec");
Connection connection = cxfactory.getConnection();
Interaction interaction = connection.createInteraction();
interaction.execute(ispec, input, output);
message = (String) output.get(HelloWorldIndexedRecord.MESSAGE_FIELD);
interaction.close();
connection.close();
}
...
```

DeployISpec class

The `DeployISpec` class, which is provided in the `helloworldradeploy.jar` file, is used to deploy the `InteractionSpec` object into JNDI if your deployment environment does not provide a tool to accomplish that. It has a simple `main()` method that creates the `InteractionSpec` and binds it into JNDI. This class should be run after the resource adapter is deployed and the connection factory has been bound into JNDI.

```
...
public static void main(String[] args) throws NamingException {

    Properties properties = new Properties();
    properties.put(Context.INITIAL_CONTEXT_FACTORY, INITIAL_CONTEXT_FACTORY);
    InitialContext context = new InitialContext(properties);
    HelloWorldInteractionSpecImpl ispec = new HelloWorldInteractionSpecImpl();
    ispec.setFunctionName(HelloWorldInteractionSpec.SAY_HELLO_FUNCTION);
    context.bind("jca/HelloWorldISpec", ispec);
}
...
```

Running the code

After deploying the resource adapter into your application server environment, create a connection factory with the JNDI name of `jca/HelloWorld`. Then run the `DeployISpec` class to bind the `InteractionSpec` into JNDI under the name `jca/HelloWorldISpec`.

After deploying the Web application, you should pull up a browser and load the URL `http://SERVER_NAME/hello`. Once the page has loaded, click the Submit button and the results page with the "Hello World!" message should appear.

The sample resource adapter and sample application were successfully deployed and tested on IBM WebSphere Application Server Advanced Edition Version 4.03.

Section 7. Wrapup and resources

Summary

In this tutorial, you have been introduced to the J2EE Connector Architecture. We started with a high-level view of JCA and its primary elements: system contracts, the client API, and the resource adapter module. From there, we moved on to a more detailed discussion, encompassing the interfaces, classes, and methods that work beneath JCA to create and manage connections to an EIS. As part of this discussion, we looked at the source code for an actual JCA implementation. Each of the most relevant components of the implementation was pulled out, and its various functions explained in detail. We closed with an actual resource adapter implementation, which you are free to continue exploring on your own.

This tutorial has provided you a hands-on, step-by-step introduction to the J2EE Connector Architecture, the most relevant components beneath that architecture, and the functionality of each of those components. You should now have a fairly good foundation for building your own JCA resource adapter and connecting to an EIS.

Resources

- Download [helloworldra.zip](#), the sample code for this tutorial.
- You can download the most recent version of *J2EE (including JCA)* from Sun Microsystems (<http://java.sun.com/downloads/index.html>).
- The *J2EE Connector Architecture specification* (<http://java.sun.com/j2ee/download.html#connectorspec>) is the definitive source for information about JCA.
- The *Java Transaction API specification* (<http://java.sun.com/products/jta/>) and the *Java Transaction Service specification* (<http://java.sun.com/products/jts/>) help you understand how to implement the JCA transaction management contract.
- *Java theory and practice* columnist Brian Goetz offers a three-part introduction to the Java Transaction Service, starting with "[Understanding JTS -- An introduction to transactions](#)" (*developerWorks*, March 2002, <http://www-106.ibm.com/developerworks/java/library/j-jtp0305.html>).
- Because exception-handling is an important part of any component you build, you may also want to check out Srikanth Shenoy's "[Best practices in EJB exception handling](#)" (*developerWorks*, May 2002, <http://www-106.ibm.com/developerworks/java/library/j-ejbexcept.html>).
- The *Java Authentication and Authorization Specification*

(<http://java.sun.com/products/jaas/index-10.html>) will help you understand how to implement the JCA security contract.

- *developerWorks* two-part Java security tutorial serves as a more hands-on introduction to the JAAS. Start with "[Part 1: Crypto basics](http://www-106.ibm.com/developerworks/education/r-jsec1.html)" (July 2002, <http://www-106.ibm.com/developerworks/education/r-jsec1.html>), then see "[Part 2: Authentication and authorization](http://www-106.ibm.com/developerworks/education/r-jsec2.html)" (July 2002, <http://www-106.ibm.com/developerworks/education/r-jsec2.html>)
- If you haven't yet checked out the [Go-ForIT Chronicles](http://www-106.ibm.com/developerworks/java/library/i-extreme2/) (*developerWorks*, June 2001, <http://www-106.ibm.com/developerworks/java/library/i-extreme2/>), you should. Every installment is written by a different member of the DragonSlayer team -- including your host Willy Farrell -- and features live-action coverage of such essential topics as extreme programming, Enterprise JavaBeans, JavaServer Pages technology, and more.
- You'll find hundreds of articles about every aspect of Java programming in the *developerWorks Java technology zone* (<http://www-106.ibm.com/developerworks/java/>).
- Also see the [developerWorks Java technology tutorials page](#) for a complete listing of free tutorials.

Feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

Colophon

This tutorial was written entirely in XML, using the *developerWorks* Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial [Building tutorials with the Toot-O-Matic](#) demonstrates how to use the Toot-O-Matic to create your own tutorials. *developerWorks* also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.