CHAPTER 6

Integrating with the Enterprise Information System Tier

by Beth Stearns and Rahul Sharma

THIS chapter is about integrating Web and J2EE enterprise applications with existing enterprise information systems (EISs). This integration process is referred to as enterprise application integration (EAI). EISs provide the information infrastructure for an enterprise, and virtually no enterprise can exist without them. Examples of enterprise information systems include enterprise resource planning (ERP) systems, mainframe transaction processing (TP) systems, relational database management systems, and other legacy information systems.

EAI entails integrating applications and enterprise data sources so that they can easily share business processes and data. This integration must be accomplished without requiring significant changes to the enterprise's data. In addition, an enterprise must be assured that none of its integrated applications cause data inconsistency or otherwise compromise the integrity of the stored data. Thus, properly implemented EAI requires that there be transactional access to EISs from various applications.

The emergence of the e-business model has added another dimension to information system access: enterprises want their information to be accessible over the Web to their partners, suppliers, customers, and employees. Typically enterprises develop Web-enabled applications that access and manage information stored in their information systems. These enterprises can use J2EE applications to extend the reach of their existing information systems and make them accessible over the

Web. They can also use the J2EE technologies to develop new e-business applications. The sample application described in this book is one example of an e-business application.

Security has always been a high priority requirement in an enterprise environment. The need for security increases significantly as enterprises open their applications and data to access through the Web. An enterprise must ensure secure access to its EISs because any break in security can compromise critical information.

EISs must also support scalable client access. An increase in the number of relationships that an enterprise has to establish with its suppliers, buyers, and partners means that J2EE applications accessing EISs must be scalable and support a large number of clients.

This chapter describes the application programming model for accessing EIS resources from enterprise applications in a secure, transactional, and scalable manner. It focuses on the J2EE Connector architecture, the architecture's Common Client Interface (CCI), and illustrates how to use the J2EE Connector architecture for enterprise application integration.

6.1 Challenges in EIS Integration

Corporations typically have built their information processing systems and capabilities piecemeal. When their business grows, enterprises often find they have to combine applications that may run on different hardware platforms and that have no protocols for communicating with other software packages outside their own narrowly defined domain. Before EAI, integrating applications and data within a corporate environment was an expensive and risky proposition. In a sense, companies had "islands" of business functions and data, and each island existed in its own, separate domain. This lack of integration lead to information and process inconsistencies.

Fixing such problems meant bringing in teams of consultants and embarking on a long, expensive process of feasibility studies, systems design, and implementation. Integration projects often took years to complete with no certainty of success. Enterprises frequently abandoned projects because of cost overruns or the belated recognition of significant difficulties. Even many successful integration projects were fraught with problems.

Some of the challenges in EIS integration are:

- EISs are heterogeneous.
- Different EISs have specific client models. Legacy systems or applications that have been in existence for a long time may impose specific technology and administrative restrictions. For example, it may be difficult to create new user accounts in a legacy system or to extend this system to support development of new applications.
- Transactions and security are managed differently for different EISs. Some enterprise information systems provide advanced support for transaction and security. For example, some systems control access to their resources through transactions. Other systems offer limited or almost no support for global transactional access. For example, a system may only support transactions that are coordinated internally.
- Data consistency is an issue. EISs use different formats for storing data types.

When developing an application that integrates enterprise information systems, a component provider has to be aware of the EIS's functional and system capabilities and has to design the application components to take into account the limitations of the system. For example, an application component provider must be careful when designing components whose transactions must span multiple EISs. While a J2EE application server may support transactions that span multiple resource managers, the participating EIS resource managers may not support the two-phase commit protocol. In other cases, application components may need to limit their security requirements due to constraints of the underlying system.

The J2EE platform services coupled with the Connector architecture technologies (and other J2EE technologies such as JDBC) together provide an environment that fosters application integration with the EIS tier. The platform and technologies provide the services for integrating all types of applications, including e-commerce applications, intranet applications, and distributed applications.

6.2 Enterprise Application Integration Approaches

There are different approaches to achieving enterprise application integration. These different approaches can be used in either a two-tier or multi-tier architecture.

- A synchronous communication approach—This approach is based on synchronous adapters, which provide a synchronous request-reply communication model between an application and an EIS. When an application wants to interact with the EIS, it invokes a remote function on the EIS and then waits until the function completes and returns its reply. The reply contains the results of the function's execution on the EIS. An example of this approach is an adapter from SAP.
- An asynchronous communication approach—With this approach, an asynchronous resource adapter enables an application to call a remote function on the EIS and continue its own processing. The adapter sends the remote function to the EIS, which handles the function and returns some reply information to the application as a separate asynchronous invocation. The application does not suspend its own processing while the remote function executes on the EIS. Rather, the application continues its work and receives notification at some later point of the results of its earlier remote function invocation.
- A Java Message Server (JMS) approach—In this form of asynchronous integration, a JMS provider enables an application to access a queue-based or publish-subscribe messaging system. With queue-based communication, a queue that is independent from both the sender and receiver applications acts as a message buffer between two communicating applications. One application sends a message to this queue, while another receiver application receives its messages from the same queue. With publish-subscribe messaging, message publishers produce messages and message subscribers receive messages in which they have expressed interest. A separate publish-subscribe facility acts as the integration point—publishers publish messages to this facility and the facility delivers messages to multiple subscribers.

6.2.1 Relational Database Management System Access

Relational database management systems (RDBMS) are the most prevalent form of enterprise data store. Many application component providers use the JDBC 2.0 or 3.0 API for accessing relational databases to manage persistent data for their applications.

The JDBC API has two parts: a client API for direct use by developers to access relational databases and a standard, system-level contract between J2EE servers and JDBC drivers for supporting connection pooling and transactions. Developers do not directly use the contract between J2EE servers and JDBC drivers. Rather, J2EE server vendors use this contract to automatically provide pooling and transaction services to J2EE components. Note that, according to the JDBC 3.0 specification, the JDBC system-level contracts can be the same as the Connector architecture system contracts. Conceptually, you can consider JDBC drivers as Connector architecture pluggable resource adapters.

An application component provider uses the JDBC client-level API for such operations as: obtaining a database connection, retrieving database records, executing queries and stored procedures, and performing other database functions.

6.2.2 Other Enterprise Information System Access

An enterprise environment usually includes enterprise information systems other than relational database systems, such as:

- Enterprise resource planning systems
- Mainframe transaction processing systems
- Legacy applications
- Non-relational database systems

The J2EE Connector architecture (described in “J2EE Connector Architecture” on page 154) is the standard architecture for integrating J2EE products and applications with these heterogeneous enterprise information systems and relational databases. Prior to the J2EE Connector architecture, some EIS vendors and J2EE server providers had their own vendor-specific architectures to support EIS integration. However, now most such EIS vendors provide resource adapters that adhere to the Connector architecture system contracts and thus can be plugged into any J2EE application server. The EIS vendor develops one implementation of its resource adapter and is assured that the adapter will work with any J2EE-compliant application server.

6.2.3 Simplified Application Development

An application component provider writes the business logic for an application. The component provider can concentrate on the application's business logic when using an application programming model based on the J2EE platform with the Connector architecture. The J2EE platform and the Connector architecture, along with the Web and EJB containers, handle transactions, security, and scalability issues related to EIS access. The component provider does not need to write code to handle these system-level services. In addition, the Connector architecture ensures that there is a simple, client-oriented API for accessing an EIS. Application development tools may make it even easier for the component provider to access an EIS.

Without the J2EE platform and Connector architecture support, the component provider must not only write the application's business logic, but must also include complex code to handle security and transactions. The component provider must also understand the different EIS-specific application programming models and their impact on security and transaction handling. For example, a component provider has to manage transactions using a transaction demarcation API specific to an EIS. This might be the transaction demarcation API defined in the `java.sql.Connection` interface in the JDBC API. The component provider must also explicitly code security checks so that only valid users can access an EIS.

6.3 J2EE Connector Architecture

The J2EE Connector architecture defines a standard architecture for connecting the J2EE platform to heterogeneous EISs, including enterprise resource planning, database, and mainframe transaction processing systems. It addresses the key issues and requirements of EIS integration by defining a set of scalable, secure, and transactional mechanisms that enable the integration of EISs with application servers and enterprise applications.

The Connector architecture enables an EIS vendor to provide a standard resource adapter for its enterprise information system. Because a resource adapter conforms to the Connector architecture specification, it can be plugged into any J2EE-compliant application server to provide the underlying infrastructure for integrating with that vendor's EIS. The J2EE application server, because of its support for the Connector architecture, is assured of connectivity to multiple EISs.

The J2EE application server and EIS resource adapter collaborate to keep all system-level mechanisms—transactions, security, connection management—

transparent to the application components. This enables an application component developer to focus on a component's business and presentation logic. The developer does not have to get involved in the system-level issues related to EIS integration.

6.3.1 Connector Architecture Contracts

Through its contracts, the J2EE Connector architecture establishes a set of programming design guidelines for EIS access. These guidelines enable a component provider to develop an application that integrates with EISs. As already noted, the J2EE Connector architecture defines two types of contracts: system and application. The system-level contracts exist between a J2EE application server a resource adapter. An application-level contract exists between an application component and a resource adapter. See Figure 6.1.

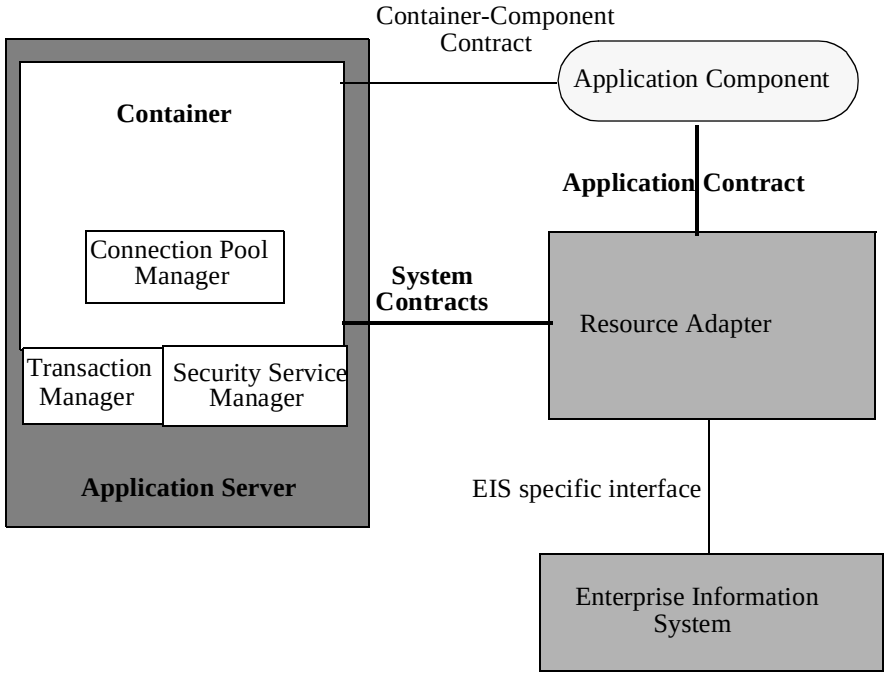

Figure 6.1 Connector Architecture System and Application Contracts

The application-level contract defines the client API that an application component uses for EIS access. The client API may be the Common Client Interface (CCI), which is an API for accessing multiple heterogeneous EISs, or it may be an API specific to the particular type of resource adapter and its underlying EIS. JDBC is an example of a client API specific to one type of resource adapter, in this case, a relational database. The CCI is the recommended client API. Tools vendors may also build their tools on top of the CCI API. See "Common Client Interface" on page 157 for more information on the CCI.

The system-level contracts define a "pluggability" standard between application servers and EISs. By developing components that adhere to these contracts, an application server and an EIS know that connecting is a straight-forward operation of plugging in the resource adapter. The EIS vendor or resource adapter provider implements its side of the system-level contracts in a resource adapter, is a system library specific to the EIS. The resource adapter is the component that plugs into an application server.

You can think of an adapter as analogous to a JDBC driver. In fact, one example of a resource adapter is a JDBC driver that connects to a relational database (as specified in the JDBC specification). Other examples of resource adapters include an adapter that connects to an ERP system and one that connects to a mainframe transaction processing system.

There is also an interface between a resource adapter and its particular EIS. This interface is specific to the type of the EIS, and it may be a native interface or some other type of interface. The Connector architecture does not define this interface.

The Connector architecture defines the services that the J2EE-compliant application server must provide. These services—transaction management, security, and connection pooling—are delineated in the three Connector system-level contracts. The application server may implement these services in its own specific way. The three system contracts, which together form a Service Provider Interface (SPI), are as follows:

- Connection management contract—This contract enables an application server to pool connections to an underlying EIS, while at the same time it enables application components to connect to an EIS. Pooling connections is important to create a scalable application environment, particularly when large numbers of clients require access to the underlying EIS.
- Transaction management contract—This contract is between the transaction

manager that is provided with the application server and an EIS that supports transactions. It gives an application server's transaction manager the ability to manage transactions across multiple EIS resource managers. (A resource manager provides access to a set of shared resources.) The contract also supports transactions that do not involve an external transaction manager; that is, local transactions that an EIS resource manager handles internally.

- Security contract—The security contract enables secure access to an EIS. It provides support for a secure application environment and protects the EIS-managed resources.

6.3.2 Common Client Interface

The Connector architecture CCI defines a set of interfaces and classes whose methods allow a client component to connect to an EIS and execute remote functions and access data on that EIS. These interfaces and classes are found in the package `javax.resource.cci`, and they divide into four functional categories:

- Connection operations—Clients use the CCI's connection interfaces to represent connection factories and application-level connections to an EIS.
- Interactions with EISs—The CCI's interaction interfaces enable a client component to execute a method or function on an EIS instance.
- Data representation and retrieval—These interfaces represent data structures. Components use them to pass and retrieve data when interacting with an EIS instance.
- Metadata information access—These interfaces, which provide meta information about a resource adapter implementation and an EIS connection, are of most use to tools vendors.

In this chapter we give you an overview of the CCI API and, using code snippets, illustrate how a client might use its methods. You can find complete documentation on the interfaces and classes, and their individual methods, in the book *J2EE Connector Architecture and Enterprise Application Integration* and via Sun's Java Web site. Refer to the Preface for complete reference information.

Briefly, here is what a client component must do to invoke a method or function on an EIS instance. A client or application component must first establish a connection to the EIS's resource manager to perform any functions on the EIS.

The component establishes a connection using the CCI's ConnectionFactory methods.

The client uses the Connection instance, which represents a connection handle to the EIS, for its subsequent interactions with the EIS instance. Examples of interactions are executing a stored procedure or a remote function on the EIS instance. The client can use the Connection instance for any number of such interactions with the EIS. These client-EIS interactions are represented by Interaction instances.

The client component performs its interactions with the EIS using an Interaction object. An InteractionSpec object is typically associated with an Interaction object. The client uses an InteractionSpec object to specify properties related to the target interaction on the EIS, such as the name of the EIS function to invoke.

Most interactions involve retrieving data from the EIS or sending data to the EIS. When the application component reads data from the EIS (such as from database tables) or writes to those tables, it does so using a particular type of Record, either a MappedRecord, IndexedRecord, or ResultSet. These record types represent the retrieved or sent data. Similar to using the ConnectionFactory to create Connection instances, the client component uses a RecordFactory to create Record instances.

In addition to the above interfaces, there are two interfaces that pertain to meta information. The ConnectionMetaData interface provides meta information about an EIS connection. A second interface, ResourceAdapterMetaData, provides information about a resource adapter implementation. Tools vendors use these interfaces to obtain information about a connection or resource adapter. See "Tools for Application Development" on page 172 for more information on using meta information.

In the sections that follow, we use the CCI API to illustrate how to perform the common EIS-integration operations. It is recommended, whenever possible, that components use the methods of the Connector architecture's CCI. However, some EIS vendors may supply their own specific API for establishing a connection., requiring you to use their API rather than the CCI API. Because another alternative is to use the JDBC API, we try to also show how these operations may be performed using the JDBC.

6.3.2.1 Establishing a Connection

Virtually all enterprise information systems are accessed via objects called connections. There are techniques for getting and managing connections in an efficient manner.

Generally, a component is responsible for obtaining a connection to an EIS. Once the component has established the connection, the component uses the connection to access EIS resources. After the component is finished, it closes the connection.

The specific steps in establishing a connection to an EIS are:

1. The deployer configures a connection factory instance in the JNDI namespace. This connection factory instance is tied to the connection factory requirements specified in the deployment descriptor by the application component provider.
2. A component looks up a connection factory from the JNDI namespace. After a successful lookup, the component calls a connection factory method to create a connection to the enterprise information system.
3. The connection factory returns a connection instance. The component uses the connection instance to access EIS resources.
4. Having established a connection to the EIS, the component manages this connection and its life cycle.
5. Once the component is finished using the connection, it closes the connection instance.

Here we provide an example that illustrates how to use the principal methods of the CCI API. Our client component in this example is a session bean that wants to interact with an EIS that is a relational database. The session bean, to establish a connection to the EIS, first uses the JNDI `Context.lookup` method to look up the relational database's resource adapter and its ConnectionFactory. The `Context.lookup` method returns three pieces of information: the user's name, password, and a reference to the resource adapter's ConnectionFactory. (See Code Example 6.1.)

```
...
    Context ic = new InitialContext();
    user = (String) ic.lookup("java:comp/env/user");
    password = (String) ic.lookup("java:comp/env/password");
```

```
    cf = (ConnectionFactory) ic.lookup("java:comp/env/CCIEIS");
...
```

Code Example 6.1 Using the CCI to Look up a Resource Adapter

The client component uses the ConnectionFactory to obtain a connection to the relational database. (Note that the client retains its reference to the ConnectionFactory to later obtain a reference to the resource adapter's RecordFactory.) For security purposes, a client must identify itself before it can obtain a connection. In our example, the session bean identifies itself using the user name and password values retrieved in the JNDI look up operation. The client does not pass these values directly to the ConnectionFactory instance, but instead the bean creates an instance of a ConnectionSpec object to hold these values. When the client invokes the ConnectionFactory's `getConnection` method to request a connection, it passes the ConnectionSpec object as a parameter to the method. (See Code Example 6.2.)

```
Connection con = null;
try {
    ConnectionSpec cSpec = new CciConnectionSpec(user, password);
    con = cf.getConnection(cSpec);
    ...
```

Code Example 6.2 Using the CCI to Obtain a Connection

Code Example 6.3 illustrates how a component gets a connection to a relational database using the JDBC 2.0 API.

```
public void getConnection(...) {
       // obtain the initial JNDI context
       Context initctx = new InitialContext();

       // Perform JNDI lookup to obtain factory
       javax.sql.DataSource ds =
           (javax.sql.DataSource)initctx.lookup(
              "java:comp/env/jdbc/MyDatabase");
```

```
        // Invoke factory to get a connection
        java.sql.Connection cx = ds.getConnection();

        // Use the Connection to access the resource manager
        ...
}
```

Code Example 6.3 Establishing a Database Connection with JDBC

6.3.2.2 Interacting with an EIS

Once it has obtained the connection to the database, the session bean can invoke functions that access and update the database. An enterprise bean uses the Interaction interface's `execute` method to invoke a function that accesses an EIS. For example, the bean might want to invoke an EIS database stored procedure called `AddRecord` to add a record or row to a database table. However, before it can use the `execute` method, the bean must do the necessary set up work: create an Iteration object, define an InteractionSpec object, and create any needed Record objects.

The bean first invokes the Connection object's `createInteraction` method to create an Interaction object. The bean also instantiates an InteractionSpec object so that it can pass properties, such as the schema, catalog, and function names, to the `execute` method. (The schema and catalog pertain to the database, while the function name is the name of the procedure to be invoked.) The bean sets up these property values as shown in Code Example 6.4:

```
...
    Interaction ix = con.createInteraction();
    CciInteractionSpec iSpec = new CciInteractionSpec();
    iSpec.setSchema(user);
    iSpec.setCatalog(null);
    iSpec.setFunctionName("AddRecord");
...
```

Code Example 6.4 Setting Up InteractionSpec Values

The session bean might also need to create Record objects to handle input and output data. The bean creates an input record to pass parameters to the stored pro-

cedure `AddRecord`. If `AddRecord` returned results, then the bean would create an output record to hold those results. Our example bean gets a reference to a RecordFactory from the ConnectionFactory object, and then uses the RecordFactory's `createIndexedRecord` method to create an IndexedRecord object which it designates as the input record. (See Code Example 6.5.)

An input record maps the input parameters passed to a stored procedure. Keep in mind that a stored procedure's input parameters may be solely for input (called IN parameters) or they may hold output or returned values as well (in which case they are referred to as INOUT parameters). An output record maps equivalent output parameters of the stored procedure, referred to as the OUT and INOUT parameters.

The bean uses the Record object's `add` method to insert input values, `name` and `qty`, into the record. These two values are parameters that will be passed to the EIS function. The bean can now use the Interaction object's `execute` method to invoke the `AddRecord` stored procedure, which inserts the new record or row into the table. (See Code Example 6.5.)

```
...
    RecordFactory rf = cf.getRecordFactory();
    IndexedRecord iRec = rf.createIndexedRecord("InputRecord");
    boolean flag = iRec.add(name);
    flag = iRec.add(new Integer(qty));
    ix.execute(iSpec, iRec);
...
```

Code Example 6.5 Defining an Record and Executing an EIS Function

Similarly, the session bean might invoke a database stored procedure called `ReadRecords` to read all the records from a particular database table. Even though `ReadRecords` does not take any parameters, the bean must still create an input record and pass it as a parameter to the `execute` method. The bean merely creates the input record but does not set any values into the record. When the bean executes the stored procedure, the results are returned to an output record. (See Code Example 6.6.)

```
...
    iSpec.setFunctionName("ReadRecords");
    RecordFactory rf = cf.getRecordFactory();
```

```
    IndexedRecord iRec = rf.createIndexedRecord("InputRecord");
    Record oRec = ix.execute(iSpec, iRec);
    Iterator iterator = ((IndexedRecord)oRec).iterator();
    while (iterator.hasNext()) {
        //read in data from each entry
    }
...
```

Code Example 6.6 Returning Data from an EIS Function

Notice in Code Example 6.6 that the `execute` method returns an output record object, `oRec`. In our example, the bean casts `oRec` to an IndexedRecord type. An IndexedRecord holds its elements in an ordered list based on java.util.List. As a result, the bean uses a Java Iterator to access the elements of the list.

6.3.2.3 Transactions and the CCI

The session bean in this example did not include any transaction code because the bean used container-managed transaction demarcation rather than bean-managed transaction demarcation. That is, the bean allowed the EJB container to handle the transaction demarcation, rather than including code to handle the transaction itself.

The CCI defines a LocalTransaction interface that may be used by an enterprise bean to explicitly manage local transactions on an underlying resource manager. An enterprise bean manages a local transaction when the bean uses bean-managed transaction demarcation. It is recommended that enterprise beans use container-managed transaction demarcation.

To implement bean-managed transaction demarcation, the session bean in the previous example uses the methods of the LocalTransaction interface to do transaction management. The session bean calls the Connection object's `getLocalTransaction` method to explicitly create the transaction context. The `getLocalTransaction` method returns a LocalTransaction instance. The bean then starts the transaction using the LocalTransaction object's `begin` method. It then proceeds to do its transactional work—invoking the stored procedure to insert the new record into the database table—and finally calls the LocalTransaction object's `commit` method to commit the transaction. The code to use a local transaction to insert a database record might look as shown in Code Example 6.7.

```
...
    iSpec.setFunctionName("AddRecord");
    RecordFactory rf = cf.getRecordFactory();
    IndexedRecord iRec = rf.createIndexedRecord("InputRecord");
    boolean flag = iRec.add(name);
    flag = iRec.add(new Integer(qty));
    javax.resource.cci.LocalTransaction transaction =
           con.getLocalTransaction();
    transaction.begin();
    ix.execute(iSpec, iRec);
    transaction.commit();
...
```

Code Example 6.7 Performing a Local Transaction

6.3.3 Guidelines for Connection Management

It is important for application servers and components to manage connections efficiently. Connections are expensive to create and remove. If an application server created a new connection for each component's request, and then subsequently destroyed that connection instance when the component completed its work, it would be virtually impossible for the server and the application to support large numbers of users. To avoid this problem, J2EE application servers support connection pooling. While each application server can implement its own connection pooling mechanism, adhering to the Connector architecture ensures that the pooling mechanism is efficient, scalable, and extensible.

An application server, by providing connection pooling, enables connections to be shared among client sessions so that a larger number of concurrent sessions can access an EIS. If each component were to acquire an EIS connection and hold it until its removal, it would be difficult to scale up an application to support thousands of users. Since holding on to an EIS connection across long-lived instances or transactions is expensive, and since there is often a physical limitation to the number of connections to an EIS, components must manage connections efficiently. Application component providers need to follow sound connection management practices.

Connection management is especially important when applications migrate from a two-tier architecture to a multitier component-based architecture. For example, it may be fine for a two-tier JDBC application to share a single connec-

tion across an entire application. However, after migrating to a component-based partitioning, the same application must deal with shared connections across multiple component instances.

In the next sections we provide guidelines for addressing application programming model issues related to connections. These guidelines are applicable to using a CCI connection as well as a JDBC connection to a relational database.

6.3.3.1 Connection Life Cycle and Connection Pooling

A component can get a connection to a database from within any client- or container-invoked method. We recommend that components open and close their connections within a single method. Components should not hold connection state across methods. Components should only consider retaining connections when the design of an application requires components to share connections across component instances or method invocations.

Keep in mind that, when a component does retain a connection across multiple methods, there is the cost of additional system resources and increased programming model complexity to manage the connection. To illustrate, consider a stateful session bean instance that retains the results of queries and database access operations across methods. The session bean obtains a connection and starts a transaction through that connection. Ordinarily, the database internally handles the transaction itself with no external transaction management. However, since the session bean wants to have this transaction span multiple methods, the bean must keep the connection open across method invocations and must include code to manage the on-going transaction.

Ideally, containers take care of connection sharing. The J2EE platform, through the Connector architecture, defines a standardized way of implementing connection sharing across different containers. Components should use the connection sharing mechanisms standardized by the J2EE Connector architecture, especially if they want to maximize an application's portability across containers. This is preferable to using any vendor-specific mechanisms offered by different containers and JDBC drivers, as these mechanisms limit portability across containers.

6.3.3.2 Connection Management by Component Type

A J2EE application is typically composed of components of different types: JSP pages, servlets, and enterprise beans. These component types vary in terms of their

support for container-managed activation and passivation, execution of an instance for multiple clients, sharing of an instance across multiple clients, and other factors. Since connection management can vary by component type, an application component provider must account for such differences when deciding on a connection management model for an application. Here are a few examples that illustrate these differences.

A JSP page or servlet acquires and holds on to a connection to an EIS, whether that connection is initiated through the CCI or JDBC, in relation to the life cycle of its HTTP session. The JSP page or servlet can handle multiple HTTP requests across a single HTTP session, provided that those requests come from Web clients using the same EIS connection.

A stateful session bean can share an open connection and its client-specific query results across multiple methods. However, keep in mind that stateless session beans are designed to retain no state specific to a client. As a result, while stateless session beans can share a connection across methods, they maintain no client-specific state associated with the connection.

For entity beans, the EJB specification identifies methods that are allowed to perform EIS access through a connection. These include `ejbCreate`, `ejbPostCreate`, `ejbRemove`, `ejbFind`, `ejbActivate`, `ejbLoad`, `ejbStore`, and any business methods from the remote interface. An entity bean cannot access an EIS from within the `setEntityContext` and `unsetEntityContext` methods because a container does not have a meaningful transaction or security context when these two methods are called.

6.3.3.3 Multiple Connections

Some JDBC drivers don't support multiple concurrent connections under a single transaction. To be portable, components should avoid opening multiple concurrent connections to a single database or EIS. However, multiple component instances can access the same database using different connections.

6.3.4 Security Architecture

An enterprise is dependent on its information systems for its business activities. Few enterprises can tolerate lost or inaccurate information. Information loss or unauthorized access to an EIS can be extremely costly. Thus, enterprises require that the security of their enterprise information systems never be compromised. As a result,

applications must provide access to enterprise information systems without creating security threats to these valuable resources.

Enterprises should clearly establish the requirements and architecture for a secure EIS environment. They must also weigh the cost of implementing, administering, and running a secure system against the security needs of individual applications. It is best for applications to require only the level of protection needed by the enterprise; applications should also reduce the level of protection for less sensitive information or where the system is less vulnerable to threats. Enterprises need to establish their security requirements versus the cost before devising a security architecture for EIS integration.

An EIS integration security architecture should fulfill a variety of requirements to ensure support for distributed applications. The security architecture should:

- Support a consistent end-to-end security architecture across Web, EJB, and EIS tiers for applications based on the J2EE platform.
- Fit with the existing security environment and infrastructure supported by an EIS.
- Support authentication and authorization of users who are accessing enterprise information systems.
- Be transparent to application components. This includes support for enabling end-users to log on only once to the enterprise environment and access multiple enterprise information systems.
- Enable applications to be portable across security environments that enforce different security policies and support different mechanisms.

Achieving these goals depends on the cost/benefit trade-offs for the security requirements. Keep in mind, though, that the more an architecture takes care of security requirements for the application, the easier the application development effort.

6.3.4.1 Security Programming Guidelines

An application component provider follows the security model defined for the particular J2EE component—EJB, JSP, or servlet. We recommend the following guidelines for all types of components:

- An application component provider should declaratively specify security requirements for an application in the deployment descriptor. The security requirements include security roles, method permissions, and the authentication approach for EIS sign on.
- Security can be managed at the application level by an application component that is security aware. The provider of such a component includes a simple programmatic interface through which the component manages security. This programmatic interface allows the application component provider to make access control decisions based on the security context—the principal and role—associated with the caller of a method and to do programmatic sign on to an EIS. (See “Application-Managed Sign on” on page 171.)
- Other development roles, such as the J2EE server provider, deployer, and system administrator, should satisfy an application’s security requirements in the operational environment. These security requirements are specified in the deployment descriptor.

6.3.4.2 EIS Sign on

From a security perspective, the mechanism for getting a connection to a resource is referred to as *EIS sign on*. A user requests a connection to be established under its security context. This security context includes various attributes, such as role, access privileges, and authorization level for the user. All application-level invocations to the database using this connection are then provided through the security context associated with the connection.

If the EIS sign on mechanism involves authentication of the user, then an application component provider can authenticate the user in one of two ways.

- The component provider allows the deployer to set up the EIS sign on information and the container manages sign on. For example, the deployer sets the user name and password for establishing the database connection. The container then takes the responsibility of managing the database sign on. This is some-

times referred to as container-managed EIS sign on.

- The component provider implements sign on to the database from the component code. The component provides explicit security information for the user requesting the connection. This is referred to as application-managed EIS sign on.

We recommend that a component let the container manage EIS sign on. This removes the burden of managing security information for the sign on from the application component provider. It also enables J2EE servers to provide additional useful security services, such as single sign on across multiple EISs and principal mapping across security domains.

Container-managed EIS sign on has other advantages. It enables the application component provider to avoid hard-coding security details in the component code. A component with hard-coded security logic is less portable because its code must be changed if deployed on containers with different security policies and mechanisms.

6.3.4.2.1 Container-Managed Sign on

Here we illustrate how the application component provider delegates the responsibility of setting up and managing EIS sign on to the container. The deployer sets up the EIS sign on so that the user account for connecting to the database is always `eStoreUser`. The deployer also configures the user identification and authentication information—user name and password—that is needed to authenticate `eStoreUser` to the database.

The component provider then uses the CCI API (see “Common Client Interface” on page 157 for more information on the CCI) to establish a connection to the database. The component obtains a reference to a ConnectionFactory, then uses the ConnectionFactory to get the connection to the database. Notice that it invokes the `getConnection` method, the method that actually connects to the database, with no security parameters. See Code Example 6.8

```
// Obtain the initial JNDI context
    Context ic = new InitialContext();
    cf = (ConnectionFactory) ic.lookup("java:comp/env/CCIEIS");
    ...
```

```
    con = cf.getConnection();
...
```

Code Example 6.8 Using the CCI for Container-Managed Sign on

It is also possible to use the JDBC API for container-managed EIS sign on. Code Example 6.9 shows the component code for invoking the connection request method on the `javax.sql.DataSource` with no security parameters. As in the previous example, the component instance relies on the container to do the sign on to the database using the security information configured by the deployer. Code Example 6.10 contains the corresponding connection factory reference deployment descriptor entry, where the `res-auth` element specifies that sign on is performed by the container.

```
// Obtain the initial JNDI context
Context initctx = new InitialContext();
// Perform JNDI lookup to obtain connection factory
javax.sql.DataSource ds = (javax.sql.DataSource)initctx.lookup(
                            "java:comp/env/jdbc/MyDatabase");

// Invoke factory to obtain a connection. The security
// information is not given, and therefore it will be
// configured by the Deployer.
java.sql.Connection cx = ds.getConnection();
```

Code Example 6.9 Container-Managed Sign on with JDBC

```
<resource-ref>
    <description>description</description>
    <res-ref-name>jdbc/MyDatabase</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

Code Example 6.10 Connection Factory Reference Element

6.3.4.2.2 Application-Managed Sign on

With application-managed sign on, the application component provider performs a programmatic sign on to the database. The component passes explicit security information (user name, password) to the connection request method. When using the CCI API, the component first sets up a ConnectionSpec object to hold the security information parameters. It then passes the ConnectionSpec object to the `getConnection` method. See Code Example 6.11.

```
...
    Context ic = new InitialContext();
    username = (String) ic.lookup("java:comp/env/user");
    password = (String) ic.lookup("java:comp/env/password");
    cf = (ConnectionFactory) ic.lookup("java:comp/env/CCIEIS");
    try {
        ConnectionSpec cSpec = new CciConnectionSpec
            (username, password);
        con = cf.getConnection(cSpec);
    ...
```

Code Example 6.11 Using the CCI for Application-Managed Sign on

Application-managed sign on can also be accomplished using the JDBC API. Again, the component passes the security information, the user's name and the password, to the connection request method of the `javax.sql.DataSource`. See Code Example 6.12.

```
// Obtain the initial JNDI context
Context initctx = new InitialContext();

// Perform JNDI lookup to obtain factory
javax.sql.DataSource ds = (javax.sql.DataSource)initctx.lookup(
                            "java:comp/env/jdbc/MyDatabase");

// Get connection passing in the security information
java.sql.Connection cx = ds.getConnection("eStoreUser",
                                          "password");
```

Code Example 6.12 Application-Managed Sign on with the JDBC API

6.3.4.3 Handling EIS Access Authorization

An application component provider relies on both the container and EIS for authorizing access to EIS data and functions. The application component provider specifies security requirements for application components declaratively in a deployment descriptor. A set of security roles and method permissions can be used to authorize access to methods on a component. For example, an application component provider declaratively specifies the `PurchaseManager` role as the only security role that is granted permission to call the `purchase` method on a `PurchaseOrder` enterprise bean. The `purchase` method in turn drives its execution through an ERP Logistics application by issuing a purchase requisition. In effect, this application has authorized only end-users with the `PurchaseManager` role to do a purchase requisition. This is the recommended authorization model.

An application component provider can also programmatically control access to enterprise information system data and functions based on the principal or role associated with the client who initiated the operation. For example, the EJB specification allows component code to invoke `getCallerPrincipal` and `isCallerInRole` to get the caller's security context. An application component provider can use these two methods to perform security checks that cannot be expressed declaratively in the deployment descriptor.

An application can also rely on an enterprise information system to do access control based on the security context under which a connection to the enterprise information system has been established. For example, if all users of an application connect to the database as `dbUser`, then a database administrator can set explicit permissions for `dbUser` in the database security domain. The database administrator can deny `dbUser` permission to execute certain stored procedures or to access certain tables.

6.4 Tools for Application Development

Application development tools can simplify EIS integration. By providing support for end-to-end application development, tools also minimize the difficulties of working with vendor-specific client APIs. Different tools provide different functionality within the application development process. Generally, tools can be divided into these functional areas:

- Data and function mining tools, which enable application component providers to look at the scope and structure of data and functions in an existing informa-

tion system.

- Object-oriented analysis and design tools, which enable application component providers to design an application in terms of enterprise information system functionality.
- Application code generation tools, which generate higher level abstractions for accessing data and functions. A mapping tool that bridges different programming models, such as an object to relational mapping, falls into this category.
- Application composition tools, which enable application component providers to compose application components from generated abstractions (such as those described in previous bullets). These tools typically use the JavaBeans component model to enhance ease of programming and composition.
- Deployment tools, which are used by application component providers and deployers to set transaction, security, and other deployment time requirements.

Since programming access to enterprise information system data and functions is a complex application development task in itself, we recommend that application development tools be used to reduce the effort and complexity involved in EIS integration. The CCI enables application components and EAI frameworks to drive interactions across heterogeneous EISs using a common client API. EAI vendors developing such frameworks rely on the Connector architecture's CCI and vendor-specific metadata repository. In essence, the tool vendor provides an application integration framework that sits on top of the functionality provided by resource adapters from different EISs. The vendor uses the CCI as a standard way to plug in different resource adapters with the integration framework. Figure 6.2 illustrates how enterprise application development tools facilitate EIS integration within the Connector architecture.

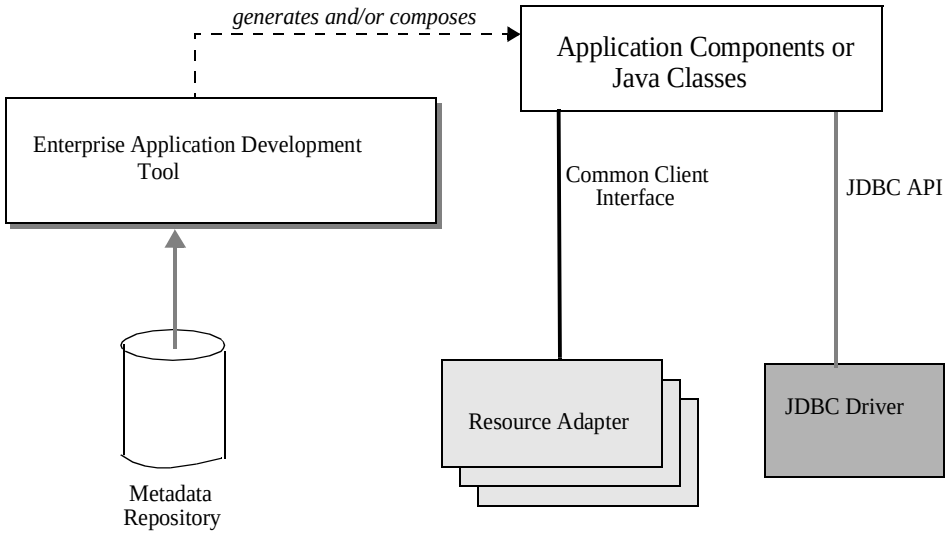

Figure 6.2 Enterprise Application Development Tool Within Connector Architecture

The vendor's development tool might also use a metadata repository, which maintains meta information about functions on an EIS system, to drive its CCI-based interactions with an EIS. Such a repository typically contains type mapping and data structure information useful for invocation parameters.

A tool that adds functionality to a resource adapter uses the CCI as a plug-in contract to the resource adapter. The tool generates Java classes based on the meta information it extracts from a metadata repository. These classes encapsulate CCI-based interactions and expose a simple application programming model (typically based on the JavaBeans framework) to application developers. An application component can use these generated Java classes to access the EIS.

6.4.1 Access Objects

A component can access data and functions in an enterprise information system in several ways, either directly by using the corresponding client API or indirectly by abstracting the complexity and low-level details of an EIS access API into higher level *access objects*. Access objects can take different forms—they can be data

access objects, command beans, or records. There are advantages to using access objects, as follows:

- An access object can adapt a low-level programming API for accessing EIS data and/or functions to an easy-to-use API that is consistent across multiple types of enterprise information systems. For example, an access object may follow a design pattern that maps EIS function parameters to setter methods and return values to getter methods. The application component provider uses an EIS function by first calling the appropriate setter methods to set up the parameters, then calling the method corresponding to the EIS function, and finally calling the getter methods to retrieve the results.
- An access object facilitates a component's ability to adapt to different EIS resources. For example, a component can use an access object to adapt its persistent state management to a different database schema or to a different type of database.
- A component can be composed from access objects that support the JavaBeans model, or, by using application development tools, the component can be linked with generated access objects. This simplifies the application development effort.

Access objects primarily provide a programming technique to simplify application development. Because of this, we recommend that they be used whenever an application component provider needs to access data or functions in an EIS. In some cases, tools may be available to generate such access objects. In other cases, the application component provider may need to hand-code an access object.

6.4.1.1 Guidelines for Access Objects

There are some general guidelines to follow in developing access objects:

- An access object should not make assumptions about the environment in which it will be deployed and used.
- An access object should be designed to be usable by different types of components. For example, if an access object follows the set-execute-get design pattern described previously, then its programming model should be consistent across both enterprise beans and JSP pages.

- An access object should not define declarative transaction or security requirements of its own. It should follow the transaction and security management model of the component that uses it.
- All programming restrictions that apply to a component apply to the set of access objects associated with it. For example, an enterprise bean isn't allowed to start new threads, to terminate a running thread, or to use any thread synchronization primitives. Access objects should conform to the same restrictions.

6.4.1.2 Types of Access Objects

As noted earlier, there are different types of access objects, and these different access objects are used for different purposes.

- Encapsulating functions

  An access object can encapsulate one or more EIS functions, such as business functions or stored procedures. This type of access object is referred to as a command bean. An application component uses a command bean to interact with a resource adapter and execute an EIS function. The command bean is associated to an EIS function and it hides the low-level programming aspects of accessing a remote EIS function. The application component, rather than having to program through the CCI or EIS client-side API, instead accesses the EIS function by programming to the command bean's interface. Figure 6.3 shows a command bean.

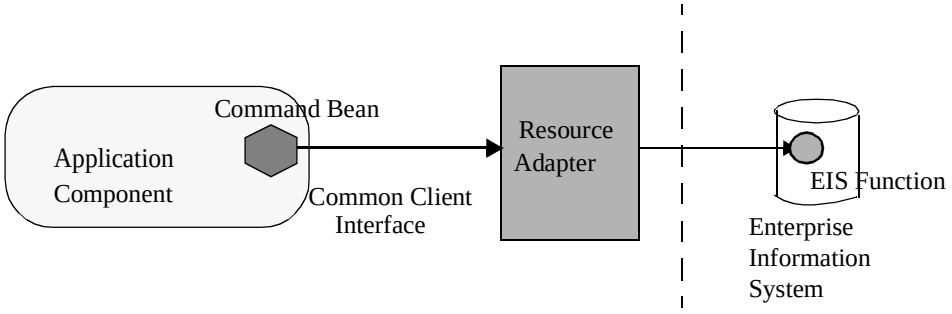

Figure 6.3 Command Bean

The following code implements a command bean that drives a purchase

requisition business process on an enterprise resource planning system by mapping purchasing functions to method calls on a purchase function object.

```
PurchaseFunction pf = // instantiate access object for PurchaseFunc-
tion
// set fields for this purchase order
pf.setCustomer("Wombat Inc");
pf.setMaterial(...);
pf.setSalesOrganization(...);
pf.execute();
// now get the result of purchase requisition using getter methods
```

- Encapsulating persistent data

  When an access object encapsulates access to persistent data, such as that stored in a database management system, it is called a data access object (DAO). Often, tools generate data access objects based on the database schema. Data access objects can provide a consistent API across different types of database management systems. The sample application uses data access objects to access order objects stored in different types of databases.

  The principal advantage of data access objects is that they decouple the user of the data access object from the programming mechanism that accesses the underlying data. The DAO exposes the same interface to its clients regardless of the API it uses to access the EIS data. Even changes to the schema or function specification of the EIS may not impact the DAO's user interface. This means that the user's programming model does not have to change when EIS access mechanisms change or there are modifications to the EIS schema.

- Representing data structure

  Another type of access object, called a record, is used as a Java representation of a data structure. As such, it can be used to hold input or output data for an EIS function. A record can be a custom implementation, in which case a tool generates the record from the meta information in a repository, or it can be a generic implementation, in which case it extracts meta information from a metadata repository at runtime. Such meta information includes type mapping and data representation.

- Aggregating behaviors

  An access object can aggregate access to other access objects, providing a higher level abstraction of application functionality. For example, a `Pur-`

`chaseOrder` aggregated access object can drive its purchase requisition business process through the `PurchaseFunction` access object and use a data access object `PurchaseData` to maintain persistent attributes of the purchase order.

6.4.1.3 Using Access Objects

A component can use access objects in different ways depending on the functionality they offer. Some common ways to use access objects are:

- Define a one-to-one association between components and access objects. That is, each access object encapsulates the EIS functionality required by a particular component. This approach enables components to have Web access to EIS resources that are encapsulated by an access object.
- Define components to aggregate the behavior of multiple access objects. This approach is often used when a component accesses multiple EIS resources or adds additional business logic to the functionality defined by multiple EIS resources.

6.5 Summary

This chapter has described designs and guidelines for integrating enterprise information systems into enterprise applications. These guidelines enable an application component provider to develop an enterprise application based on its overall functional and system requirements for EIS integration. The chapter focuses on accessing EIS resources from a component, using tools to simplify and reduce application development effort involved in accessing EISs, obtaining and managing connections to EISs, and supporting the security requirements of an application.

The current version of the J2EE platform includes the Connector architecture, which provides full support for integrating all types of enterprise information systems, including database and legacy systems, with the J2EE platform. The JDBC API is also available for accessing relational databases.

6.6 References and Resources

For our latest thinking on the enterprise information systems tier, see

http://java.sun.com/j2ee/blueprints/eis_tier/