CHAPTER 8

Understanding Transactions

THE EJB architecture provides for two kinds of transaction demarcation: container-managed transaction demarcation and bean-managed transaction demarcation. This chapter covers the essential aspects of transactions necessary for a typical application developer to know.

Under the *container-managed transaction demarcation* approach, the EJB Container does the bulk of the work of managing transactions for the programmer. This greatly simplifies the application developer's work when programming transactional applications. However, even though the container does the majority of the work, the Bean Provider or Application Assembler must still provide transaction-related instructions to the container. Part of this chapter describes how the example application described in Chapter 7, Entity Bean Application Example, utilizes *transaction attributes*, which are special attributes set in the deployment descriptor to instruct the container on how to manage transactions for the Benefits application.

This chapter also discusses *bean-managed transaction demarcation*. With this approach, the Bean Developer manages transaction boundaries programmatically from within the application code. The benefits enrollment application does not use bean-managed transaction demarcation. Instead, we discuss appropriate scenarios for using bean-managed transaction demarcation.

8.1 Overview of Transactions

Application programmers benefit from developing their applications on platforms that support transactions. A transaction-based system simplifies application development because it frees the developer from the complex issues of failure recovery and multi-user programming. In addition, the EJB architecture does not limit transac-

tions to single databases or single sites. The EJB architecture supports distributed transactions that can simultaneously update multiple databases across multiple sites.

How is this accomplished? The EJB architecture permits the work of an application to typically be divided into a series of units. Each unit of work is a separate transaction. While the application progresses, the underlying system ensures that each unit of work—each transaction—fully completes without interference from other processes. If not, the system rolls back the transaction, completely undoing whatever work the transaction had performed.

The EJB architecture allows enterprise beans to utilize a declarative style of transaction management that differs from the traditional transaction management style. With declarative management, the enterprise bean application declares transaction attributes in the deployment descriptor; these transaction attributes describe how to partition the work of an application into separate, discrete units of work. The transaction attributes indicate to the Container how it should apply transaction management to the execution of the bean's methods.

Under the traditional transaction management approach, the application was responsible for managing all aspects of a transaction. This entailed such operations as

- Explicitly starting the transaction
- Committing or rolling back the transaction
- Suspending and resuming the transaction association, particularly for applications that need more sophisticated transaction demarcation

A developer is required to have more programming expertise to be able to write an application that is responsible for managing a transaction from start to finish. The code for such an application is more complex, and thus more difficult to write, and it is easy for "pilot error" to occur (for example, a programmer may forget to commit a transaction). Furthermore, it is hard to re-use components that programmatically manage transaction boundaries as building blocks for applications with additional components.

With declarative transaction management, the Container manages most if not all aspects of the transaction for the application. The Container handles starting and ending the transaction, plus it maintains its context throughout the life of the transaction. The Container automatically propagates the transaction context into invoked enterprise beans and resource managers, based on the declarative instructions in the deployment descriptor. This greatly simplifies an application devel-

oper's responsibilities and tasks, especially for transactions in distributed environments. In addition, it means that the components are re-usable as building blocks for other applications which are composed of multiple components.

8.2 Declarative Transaction Demarcation

Most applications are best off using the container-managed transaction demarcation feature. (This feature is commonly referred to as *declarative transaction demarcation* or *declarative transactions*.) For this feature to work, application developers set up transaction attributes separate from their code: A transaction attribute is associated with each enterprise bean method. The EJB container uses these attributes to determine how it should handle transaction demarcation for that method. As a result, the application programmer does not need to include transaction demarcation code in the application.

8.2.1 Transaction Attributes

When a client invokes a method of an enterprise bean using the enterprise bean's home or remote interface, the Container interposes on the method invocation to inject the container services. One of the services that the Container injects is transaction demarcation.

The Bean Developer or Application Assembler uses the deployment descriptor to specify how the container should manage transaction demarcation. Essentially, the deployment descriptor allows the Bean Developer or Application Assembler to specify a transaction attribute for each method of the remote and home interface.

Keep in mind that there are some limitations on the methods to which transaction attributes may be assigned. The deployment descriptor may not assign a transaction attribute to all of the methods of the home and remote interface. For example, a session bean may define a transaction attribute only for the business methods defined in the bean's remote interface; it must not assign transaction attributes to the methods of the home interface. In addition, it may not assign transaction attributes to the methods defined in the EJBObject interface because the Container implements these methods; therefore it is meaningless to define a transaction attribute for them.

Like a session bean, an entity bean may also define a transaction attribute for each of the business methods in the bean's remote interface. In addition, an entity

bean may define transaction attributes for the `create` and find methods defined in the home interface, and for the `remove` methods inherited from the EJBObject and EJBHome interfaces. However, an entity bean may not define transaction attributes for all other methods defined in the EJBObject and EJBHome interfaces. Because the Container implements these other methods it would be meaningless for the entity bean to define a transaction attribute for them.

There is one implication of the above rule that is important to note. The create and remove methods of an entity bean have transaction attributes, but the `create` and `remove` methods of a session bean do not have transaction attributes. The Container treats the `create` and `remove` methods of a session bean as if they had the NotSupported transaction attribute.

8.2.2 Transaction Attribute Values

A transaction attribute may have one of the six values:

- Required
- RequiresNew
- Supports
- NotSupported
- Mandatory
- Never

The following sections describe how transaction attributes are used. Table 8.1 on page 372 summarizes these transaction attributes.

8.2.2.1 Required Transaction Attribute

The Required transaction attribute is typically used for the bean methods that update databases or other transaction-capable resource managers. The Required transaction attribute ensures that all the updates from the method are performed atomically, and that the updates done by the enterprise bean method can be included in a larger transaction. A transaction is required in such scenarios to achieve application correctness.

How does the Container interpret and apply the Required transaction attribute? If a method is assigned the Required transaction attribute, the Container executes the method with a transaction context such that

- If the method caller is already associated with a transaction context, the Container includes the execution of the method in the client's existing transaction context.
- If the method caller is not associated with a transaction context, the Container starts a transaction before the execution of the method, and it commits the transaction after the method has completed.

The EnrollmentBean in the application in Chapter 7 uses the Required transaction attribute for the `commitSelections` method. By using this attribute for this method, the application ensures that the updates to BenefitsDatabase and the Payroll System are performed as a single transaction.

8.2.2.2 RequiresNew Transaction Attribute

The RequiresNew transaction attribute, like the Required attribute, is typically used for methods that update databases. However, methods that use the RequiresNew attribute should have the database updates committed regardless of the outcome of the caller's transaction.

The Container applies the RequiresNew transaction attribute somewhat differently than the Required attribute. The Container always executes a method that is assigned the RequiresNew transaction attribute in a new transaction context. This means that the Container starts a new transaction before it executes the method and commits the transaction after the method completes. If the method caller is already associated with a transaction context at the time it calls the method, the Container suspends the association for the duration of the new transaction.

In what situations might the RequiresNew transaction attribute be useful? An application service provider might want to track, for marketing or billing purposes, the applications that each of its users has executed. Use of the RequiresNew attribute enables the application service provider to implement such tracking. The service provider uses an enterprise bean ApplicationStatistics for collecting the application usage information. Each time a user invokes an application, the application in turn invokes the `recordUsage` method on the ApplicationStatistics bean. The `recordUsage` method is assigned the RequiresNew transaction attribute to ensure that it records the usage information even if the actual applica-

tion rolls back its transaction. In contrast, if the Required attribute were used for the `recordUsage` method and the application rolled back its transaction, there would be no record that the user ran the application.

8.2.2.3 Supports Transaction Attribute

The Supports transaction attribute is used when a method does not absolutely require a transaction. When a method is assigned the Supports transaction attribute, the method's transaction context depends on the transaction context of the method caller. The Container executes the method with or without a transaction context depending on whether the method caller is associated with a transaction context.

If the caller is associated with a transaction context, the Container includes the method execution in the caller's transaction. (In this case, the Container's execution of the method is the same as if the method had been assigned the Required transaction attribute.)

If the caller is not associated with a transaction context, the Container executes the method in a manner whose transaction semantics are not defined by the EJB specification. (In this case, the Container's execution of the method is the same as if the method had been assigned the NotSupported transaction attribute.)

When would you use the Supports transaction attribute? You typically use this attribute for those methods in which atomicity of multiple updates from within the method is not an issue. These cases include methods that make no updates (directly or indirectly via other enterprise beans) to data. Or, they make only a single data update operation that is guaranteed to be atomic by other mechanisms (such as atomicity of a SQL statement). The Supports attribute allows the Container to avoid the overhead of using a transaction for executing a method when a transaction is not needed. At the same time, it tells the Container to include the work of the method into the client's transaction when this is required.

8.2.2.4 NotSupported Transaction Attribute

A method may also be assigned the NotSupported transaction attribute. When a method is assigned the NotSupported attribute, the Container invokes the method in a manner whose transaction semantics are not defined by the EJB specification. Normally, the EJB specification allows the Container to invoke a method

- With no transaction context, or
- In some container-specific local transaction context

When a caller invokes a method with the NotSupported transaction attribute, and the caller is associated to a transaction context defined by the EJB specification, the Container suspends the caller's transaction association for the duration of the method.

When might you use the NotSupported transaction attribute? There are two cases when you typically might use the NotSupported transaction attribute. You would assign the NotSupported attribute

- To those methods of an enterprise bean that use resource managers not capable of interfacing with an external transaction coordinator
- Or, when the correct application semantics does not depend on performing resource manager access in a transaction

When an enterprise bean uses resource managers incapable of interfacing with an external transaction coordinator, the Container cannot propagate the transaction context into the resource managers. Using the NotSupported transaction attribute for the bean's methods instructs the Container that the application developer has taken into consideration the dependency on the less capable transaction manager.

Our example application uses the NotSupported attribute for the methods of the ProvidencePlanBean enterprise bean. ProvidencePlanBean uses the HTTP protocol to communicate with the Providence Web site. Because the HTTP protocol does not support propagating a transaction, the Providence Assembler or Bean Developer assigned the NotSupported transaction attribute to all the methods of the ProvidencePlanBean enterprise bean. The Bean Developer designed the ProvidencePlanBean such that its communication with the Providence Web site does not require a transaction for its correctness.

8.2.2.5 Mandatory Transaction Attribute

When a method is assigned the Mandatory transaction attribute, the Container first checks that the caller is associated with a transaction context. If the caller is *not* associated with a transaction context, the Container throws the `javax.transaction.TransactionRequired` exception to the caller. If the client is associated with a transaction, the container performs the method invocation in the same way as for the Required attribute—the Container includes the execution of the method in the client's existing transaction context.

When do you use the Mandatory transaction attribute? Use this attribute for a method for which an application assembly error would occur if the method was invoked by a caller without a transaction context.

8.2.2.6 Never Transaction Attribute

When a method is assigned the Never transaction attribute, the Container first checks that the caller is associated with a transaction context. If the caller is associated with a transaction context, the Container throws the `java.rmi.RemoteException` to the client. If the caller is not associated with a transaction, the container invokes the method in the same way as for the NotSupported attribute.

You use the Never attribute when you want the Container to ensure that a transactional client does not invoke an enterprise bean method that is not capable of transaction.

8.2.2.7 Summary of Transaction Attributes

Table 8.1 provides a summary of the transaction context that the EJB Container passes to an invoked business method. The table also illustrates the transaction context that the Container passes to the resource managers called by the invoked business method. As the table illustrates, the transaction context passed by the Container is a function of the transaction attribute plus the client's transaction context.

In the table, T1 represents a transaction passed with the client request, while T2 represents a transaction initiated by the Container. Keep in mind that the enterprise bean's business method may invoke other enterprise beans, via their home and remote interfaces. When this occurs, the transaction indicated beneath the column entitled "Transaction Associated with Business Method" is passed as part of the client context to the target enterprise bean.

Table 8.1 Summary of Transaction Attributes

Transaction Attribute	Client's Transaction	Transaction Associated with Business Method	Transaction Associated with Resource Managers
Required	none	T2	T2
	T1	T1	T1

Table 8.1 Summary of Transaction Attributes

Transaction Attribute	Client's Transaction	Transaction Associated with Business Method	Transaction Associated with Resource Managers
RequiresNew	none	T2	T2
	T1	T2	T2
Supports	none	none	none
	T1	T1	T1
NotSupported	none	none	none
	T1	none	none
Mandatory	none	error	N/A
	T1	T1	T1
Never	none	none	none
	T1	error	N/A

8.2.2.8 Note on Transaction Attributes for Entity beans

As explained in "Using the ejbLoad and ejbStore Methods" on page 217, an entity bean may use the `ejbLoad` and `ejbStore` methods to perform caching of data. For this caching to work correctly, the Container must combine the `ejbLoad` method, the business method(s), and the `ejbStore` method into a single transaction.

This means that, if the bean depends on the `ejbLoad` and `ejbStore` methods to manage caching, it should not be using the NotSupported, Supports, and Never transaction attributes.

8.2.3 Transaction Attributes for Sample Application

Table 8.2 shows the transaction attributes used for methods of the entity bean sample application, the application in Chapter 7. Following the table is an explanation of why these attributes were assigned in this manner.

Table 8.2 Sample Application Method Transaction Attributes

Enterprise Bean Name	Method Name	Transaction Attribute
Enrollment	`getEmployeeInfo`, `getCoverageOptions`, `setCoverageOption`, `getMedicalOptions`, `setMedicalOption`, `getDentalOptions`, `setDentalOption`, `getSmokerStatus`, `setSmokerStatus`, `getSummary`	Supports
	`commitSelections`	Requires
Selection	`getCopy`, `updateFromCopy`, `remove`	Requires
SelectionHome	`findByPrimaryKey`, `findByEmployee`, `findByPlan`, `remove`	Requires
WrapperPlan	`getPlanInfo`, `getPlanType`, `getCost`, `getAllDoctors`, `getDoctorsByName`, `getDoctorsBySpecialty`, `remove`	Requires
WrapperPlanHome	`findByPlanId`, `findByPrimaryKey`, `findMedicalPlans`, `findDentalPlans`, `findByDoctor`, `create`, `remove`	Requires
Employee	`getCopy`, `updateFromCopy`, `remove`	Requires
EmployeeHome	`findByPrimaryKey`, `remove`	Requires
PremiumHealthPlan	`getPlanInfo`, `getPlanType`, `getCost`, `getAllDoctors`, `getDoctorsByName`, `getDoctorsBySpecialty`, `remove`	Requires
PremiumHealthPlanHome	`findByPlanId`, `findByPrimaryKey`, `findMedicalPlans`, `findDentalPlans`, `findByDoctor`, `create`, `remove`	Requires

Table 8.2 Sample Application Method Transaction Attributes

Enterprise Bean Name	Method Name	Transaction Attribute
ProvidencePlan	`getPlanInfo`, `getPlanType`, `getCost`, `getAllDoctors`, `getDoctorsByName`, `getDoctorsBySpecialty`, `remove`	NotSupported
ProvidencePlan-Home	`findByPlanId`, `findByPrimaryKey`, `findMedicalPlans`, `findDental-Plans`, `findByDoctor`, `remove`	NotSupported
Payroll	`setSalary`, `getSalary`, `setBenefits-Deduction`, `getBenefitsDeduction`	Supports

Wombat assigned the Required transaction attribute to the `commitSelections` method defined in the Enrollment remote interface. The `commitSelections` method updates multiple databases, and therefore Wombat uses a transaction to achieve atomicity of the multiple updates. Wombat assigned the Supports attribute to all other methods of the Enrollment remote interface because these methods do not require a transaction for their correctness (the methods only read data from the database).

Wombat assigned the Required transaction attribute to all the methods of all the entity beans. Using the Required attribute is typical for methods of an entity bean because it guarantees that the Container includes the execution of the `ejbLoad` method, the business methods, and the `ejbStore` method in a single transaction. See “Using the ejbLoad and ejbStore Methods” on page 217.

Similarly, Premium Health assigned the Required transaction attribute to all the methods of the PremiumHealthPlanEJB. These methods access the Premium-HealthDatabase.

Providence assigned the NotSupported transaction attribute to all the methods of the ProvidenceEJB enterprise bean. Providence did so because these methods invoke the ProvidenceServiceWeb application using the HTTP protocol, which does not propagate a transaction context from the client to the Web server. Providence designed the ProvidenceEJB enterprise bean and the ProvidenceService-Web application such that they work correctly without transaction propagation.

8.3 Programmatic Transaction Demarcation

The preferred way to demarcate transactions in EJB applications is to use transaction attributes. By using transaction attributes, the Bean Developer does not have to manage transaction boundaries programmatically in the enterprise bean's code. However, while declarative transaction demarcation via transaction attributes works in most cases, there are situations in which the declarative demarcation does not provide the required functionality, or in which declarative demarcation is awkward to use (such as when it forces the application developer to unnaturally partition the application into multiple enterprise beans to achieve the required transaction demarcation).

This section discusses when and how the application developer should use programmatic transaction demarcation to control transaction boundaries programmatically. The application developer may control the transaction boundaries programmatically either in the client code for the enterprise bean, or directly in the enterprise bean business methods. The following sections describe the usage of these techniques.

8.3.1 Transaction Demarcated By a Client

Typically, an enterprise bean client (such as a Web application or a stand-alone Java client application) does not manage transaction boundaries. Instead, the client invokes methods on an enterprise bean, and the EJB container in which the target enterprise bean is deployed automatically manages transactions based on the values of the transaction attributes for the invoked method. This container-provided transaction demarcation is transparent to the client. The client either sees a successful completion of the invoked method, or, if an error occurs, receives the `java.rmi.RemoteException` from the invoked method.

Certain situations require the client to demarcate transactions programmatically. Typically in these situations, the client needs to combine the invocation of multiple methods into a single transaction. The methods can be on the same enterprise bean, or they can be on multiple beans. For this to work, the client needs to demarcate transactions programmatically, which the client accomplishes using the `javax.transaction.UserTransaction` interface. (The `javax.transaction.UserTransaction` interface is part of the Java Transaction API (JTA). More information about JTA is available at `http://java.sun.com/j2ee/docs.html`.) The client obtains the `javax.transaction.UserTransaction` interface from its environment using the JNDI name `java:comp/UserTransaction`.

The MyServlet servlet, which transfers funds from one bank account to another, illustrates this. Figure 8.1 shows the OID for the interactions that occur during MyServlet's operation:

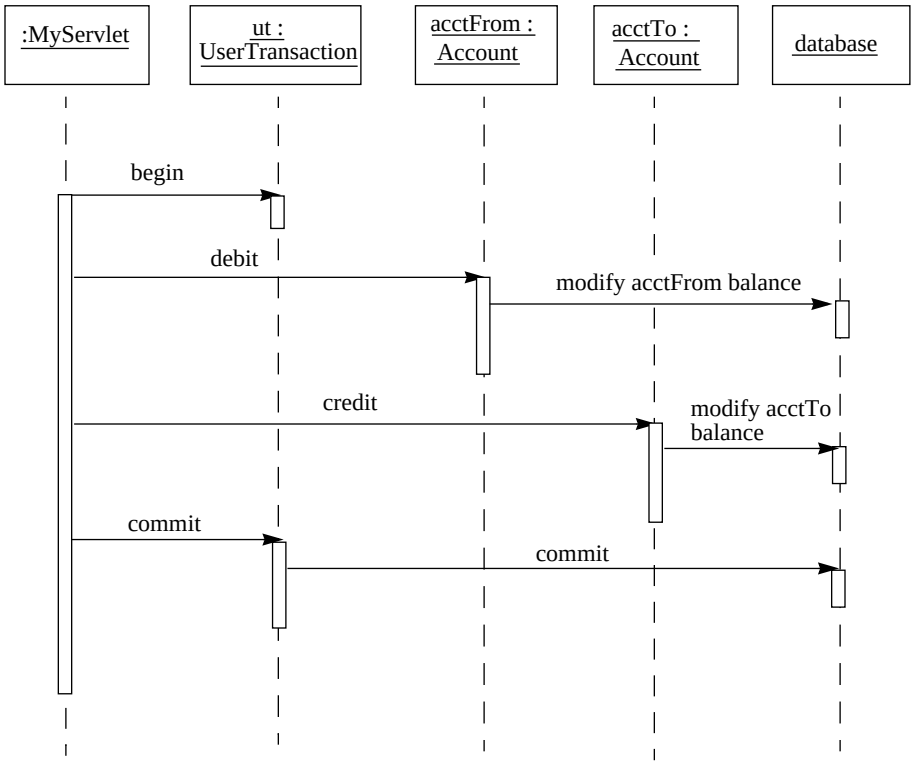

Figure 8.1 MyServlet OID

MyServlet executes the funds transfer in a single transaction by using the TransferFunds Java Bean component. (Note that TransferFunds is not an enterprise bean.) However, as far as the discussion of transactions is concerned, the TransferFunds bean is considered part of the servlet. Code Example 8.1 shows the MyServlet code:

```
public class MyServlet extends HttpServlet {
    public void service(ServletRequest req, ServletResponse resp) {
        ...
        TransferFunds transferFunds = new TransferFunds();
```

```
            transferFunds.setAccountFrom(...);
            transferFunds.setAccountTo(...);
            transferFunds.setAmount(...);
            try {
                transferFunds.execute();
            } catch (TransferException ex) {
                ...
            }
            ...
        }
    }
```

Code Example 8.1 MyServlet Class

Code Example 8.2 shows the code for the TransferFunds Java Bean:

```
import javax.transaction.*;
    ...

    public class TransferFunds {
        String accountNumberFrom;
        String accountNumberTo;
        double amount;

        public void setAccountFrom(String accountNumber) {
            accountNumberFrom = accountNumber;
        }

        public void setAccountTo(String accountNumber) {
            accountNumberTo = accountNumber;
        }

        public void setAmount(double amt) {
            amount = amt;
        }

        public void execute() throws TransferException
        {
```

```
            UserTransaction ut = null;

            try {
                ...
                AccountHome h1 = ...;
                AccountHome h2 = ...;
                Account acctFrom = h1.findByPrimaryKey(
                                            accountNumberFrom);
                Account acctTo = h2.findByPrimaryKey(
                                            accountNumberTo);

                // Obtain the UserTransaction interface.
                Context initCtx = new InitialContext();
                ut = (UserTransaction)initCtx.lookup(
                       "java:comp/UserTransaction");

                // Perform the transfer.
                ut.begin();
                acctFrom.debit(amount);
                acctTo.credit(amount);
                ut.commit();
                // Transfer was completed.
            } catch (Exception ex) {
                try {
                    if (ut != null)
                        ut.rollback();
                } catch (Exception ex) {
                }
                // Transfer was not completed.
                throw new TransferException(ex);
            }
        }
}
```

Code Example 8.2 TransferFunds Java Bean Class

Let's take a closer look at the implementation of the `execute` method of the TransferFunds Java Bean. The servlet client uses the `execute` method to accom-

plish a number of tasks. The client code works as described in the next paragraph, assuming that it does not encounter any failures.

First, the client obtains the remote interfaces for the two accounts, `acctFrom` and `acctTo`, involved in the transaction. Then the client uses the JNDI API to obtain a reference to the UserTransaction interface from the servlet's environment. Once the client obtains the reference, it starts a transaction using the `begin` method of the UserTransaction interface. The client then debits the `acctFrom` account and credits the `acctTo` account. Finally, the client commits the transaction using the `commit` method of the UserTransaction interface.

However, what is even more interesting is how the `execute` method deals with failures. Notice that the `execute` method wraps all its statements into a `try` block. If the execution of the statements in the `try` block raise an exception, then this executes the block of code in the `catch` clause. The block of code in the `catch` clause attempts to roll back the in-progress transaction started by the `execute` method and throws the `TransferException` to the caller.

If the servlet container crashes before the transaction is committed, the transaction manager will automatically roll back all updates performed by the `execute` methods. For example, if the `execute` method debited `acctFrom` before the servlet container crashed, the transaction manager instructs the database that stores `acctFrom` to roll back the changes caused by the debit operation.

8.3.2 Transaction Demarcation by a Session Bean

A session bean can use the UserTransaction interface to demarcate transactions programmatically. However, an entity bean cannot use the UserTransaction interface. In this section we describe a typical scenario in which the session bean developer uses the UserTransaction interface to demarcate a transaction rather than relying on the declarative transaction demarcation via transaction attributes.

The J2EE platform does not allow a stand-alone Java application to use the UserTransaction interface. How can a stand-alone Java application perform multiple invocations to an enterprise bean within a single transaction if it cannot use the UserTransaction interface? The application can use the bean-managed transaction demarcation feature of the EJB specification to combine multiple client-invoked methods into a single transaction. It would be impossible for a stand-alone Java application to achieve this—combining multiple client-invoked methods into a single transaction—with declarative transaction demarcation.

We illustrate this using the session bean example from Chapter 4. Let's assume that a stand-alone Java client application uses the EnrollmentEJB session

bean in that example. Let's further assume, for the sake of this illustration, that the logic of the Benefits Enrollment application requires that all data access performed by the multiple steps of the entire enrollment business process be part of a single transaction. (This is not a very realistic example!)

In such a scenario, the application developer would design the EnrollmentEJB session bean as a bean with bean-managed transaction demarcation. The developer would modify the EnrollmentBean class (described in "EnrollmentBean Session Bean Class Details" on page 89) to obtain and use the UserTransaction interface in the `ejbCreate` and `commitSelections` methods, as illustrated in Code Example 8.3:

```
public class EnrollmentBean implements SessionBean
    {
        UserTransaction ut = null;
        ...
        public void ejbCreate(int emplNum) throws EnrollmentException
        {
            // Obtain the UserTransaction interface from the
            // session bean's environment.
            Context initCtx = new InitialContext();
            ut = (UserTransaction)initCtx.lookup(
                      "java:comp/UserTransaction");

            // Start a transaction.
            ut.begin();

            // The rest of the ejbCreate method
            employeeNumber = emplNum;
                ...
        }
        ...
        public void commitSelections() {

            // insert new or update existing benefits selection record
            if (recordDoesNotExist) {
                benefitsDAO.insertSelection(selection);
                recordDoesNotExist = false;
            } else {
```

```
            benefitsDAO.updateSelection(selection);
        }

        // Update information in the payroll system
        try {
            payroll.setBenefitsDeduction(employeeNumber,
                                         payrollDeduction);
        } catch (Exception ex) {
            throw new EJBException(ex);
        }

        // Commit the transaction started in ejbCreate
        try {
            ut.commit();
        } catch (Exception ex) {
            // handle exception from commit
            ...
        }
    }
    ...
}
```

Code Example 8.3 EnrollmentBean Class with Bean-Managed Transaction Demarcation

The `ejbCreate` method starts a transaction which then spans all the methods invoked by the client application. The `commitSelections` method commits the transaction after the client application has completed its work. Figure 8.2 and Figure 8.3 show the OID for these transaction operations.

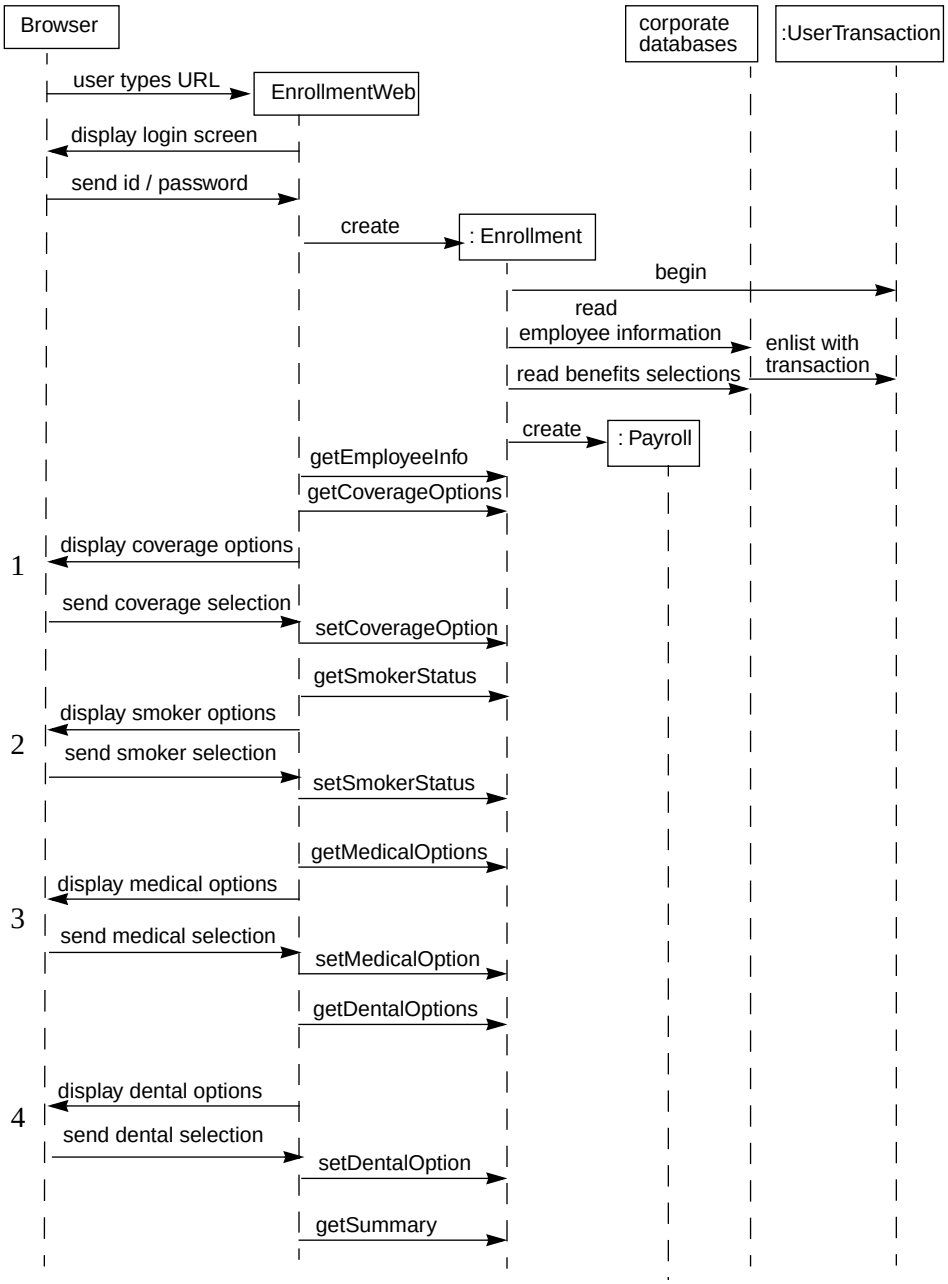

Figure 8.2 Transaction OID, Part One

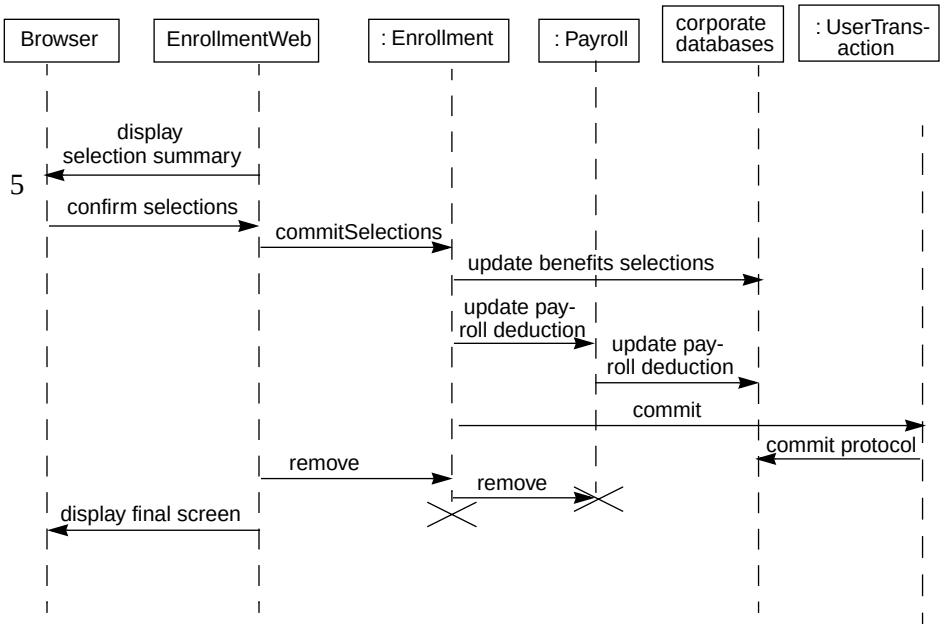

Figure 8.3 Transaction OID, Part Two

The OID diagram illustrates the interactions between the Enrollment session object and the transaction manager that take place via the UserTransaction interface. It also illustrates the interactions between the transaction manager and the corporate databases.

The Enrollment session object starts the transaction by invoking the `begin` method on the UserTransaction interface. This causes the container to include the access to the corporate databases performed by the Enrollment object as part of the transaction. As the OID diagram illustrates, the container enlists the corporate databases with the transaction.

In addition, when the Enrollment object invokes the Payroll object, the Container propagates the transaction context to the Payroll object to include the payroll deduction update as part of the transaction.

Finally, the Enrollment object commits the transaction by invoking the `commit` method on the UserTransaction interface. The transaction manager instructs the corporate databases to commit the changes made by the transaction. If the corpo-

rate database are located on multiple servers, the transaction manager performs the two-phase commit protocol.

All the accesses to the corporate databases between the `UserTransaction.begin` method invoked in the `ejbCreate` method and the `UserTransaction.commit` method invoked at the end of the `commitSelections` method are part of a single transaction.

What are the pitfalls of using bean-managed transaction demarcation? As we stated earlier, bean-managed transaction demarcation is typically used to combine multiple client-invoked methods into a single transaction. This means that a transaction is "in-progress" across a client's multiple interactions with the application. The transaction might block other transactions because a transaction causes the resource managers to hold locks on the data accessed by the transaction. If a user works slowly, or leaves the application in the middle of the in-progress transaction, the transaction may block all other users' transactions that need access to the data now locked by the slow user's transaction.

Therefore, transactions that span multiple user interactions with an application should be used only in environments with a small population of well-behaved users. And, just as importantly, they should always be used with a great deal of care.

For example, it would be very unusual if a Web-based application—with its multitude of unregulated users—used transactions that span user interactions. For this reason, the benefits applications described in Chapter 4 and Chapter 7 do not use transactions that span interactions with the user.

8.3.3 Pitfalls of Using Programmatic Transaction Demarcation

The developer using programmatic transaction demarcation needs to be very careful in the placement of the `begin`, `commit`, and `rollback` calls in the application code. For example:

- The programmer must ensure that `begin` is not called when the application is already associated with a transaction. J2EE does not support nested transactions.
- The programmer must ensure that the application will eventually `commit` or `rollback` the transaction. This may be non-trivial if the application code has many execution paths and Java exceptions are thrown. If the application does not `commit` or `rollback` a transaction, the transaction manager will eventually timeout the transaction and roll it back. But before the timeout expires, the

locks held by the transaction may block other transactions from making progress.

Therefore, an application developer should use declarative transaction demarcation whereever possible, and apply programmatic transaction demarcation only for those cases for which declarative transaction demarcation does not work. The Container implements declarative transaction demarcation and properly handles all the application execution paths.

8.4 Conclusion

This chapter explained the different transaction attributes defined by the EJB architecture. It showed how to apply these attributes with declarative transaction demarcation. The chapter also explained and demonstrated how to do transaction demarcation by clients and by session beans.

From here, we move to the issues surrounding security. Chapter 9, Managing Security, describes how to handle security from the point of view of application developers and Deployers.