SAMPLE CHAPTER

# OSGi IN DEPTH

Alexandre de Castro Alves

FOREWORD BY David Bosschaert

MANNING

*OSGi in Depth*

by Alexandre de Castro Alves

**Chapter 8**

# *brief contents*

# 8 Transactions and containers

**This chapter covers**

- The concepts of atomicity, consistency, isolation, and durability (ACID)
- The use of local transactions and global transactions
- How XA resources work to achieve consensus
- Handling exceptions in the context of a transaction
- Implementing container-managed transactions

In the previous chapter, we examined how to persist data, that is, make it durable. Yet, in several cases, this may not be enough. Not only do you need to make the enterprise data durable, but you may also need to make sure that either all of the data is persisted or none of the data is persisted. This all-or-nothing attribute is called *atomicity* and is achieved through the use of transactions.

Transactions are important to the enterprise, because enterprise applications generally involve several resources, and actions executed in these resources must

be done in an atomic fashion. We'll start our discussion by investigating a classic transaction use case where we need to transfer money between banking accounts.

## 8.1 Undoing work

Consider a banking application that transfers money from one account to another. Each account is implemented as database tables in an RDBMS (relational database management system). The banking application needs to debit one account and credit the other account by executing SQL (Structured Query Language) statements.

In chapter 7, you learned that one way of doing this is to use JDBC, as shown in the following listing.

**Listing 8.1 Transferring money from one account to another**

```
double currentA = 100.0;
double currentB = 0.0;
double amount = 50.0;

DataSource ds = ...                                    ❶ Retrieve JDBC DataSource service

Connection conn =
    ds.getConnection();

Statement stmt =
    conn.createStatement();

stmt.execute("UPDATE account SET current = " + (currentA - amount)
        + " WHERE accountId = 001");                   ❷ Debit first account

stmt.execute("UPDATE account SET current = " + (currentB + amount)
        + " WHERE accountId = 002");                   ❸ Credit second account

stmt.close();
conn.close();
```

First, you retrieve the JDBC `DataSource` service ❶. Next, you debit the first account ❷, withdrawing 50.0, and credit the second account with this same amount ❸, so that the accounts start with 100.0 and 0.0 and finish with the values of 50.0 and 50.0, respectively. Note how you start with the collective sum of 100.0 and finish with this same collective sum.

> **DEFINITION** Making sure that application invariants are always met is often referred to as consistency. In the banking example, *consistency* means that at any time you always have the same overall sum.

But what happens if for whatever reason the second execute ❸ fails? If that happens, then the accounts will finish with 50.0 and 0.0, respectively; that is, 50.0 will have disappeared and you would have lost the consistency of always keeping the collective sum of 100.0. How do you cope with the fact that at any time any operation may fail? You can solve this problem by making sure you compensate for the failures adequately.

One example of this compensation, or undoing of work, is demonstrated in the next listing.

**Listing 8.2 Compensating for statement execution failure**

```
stat.execute("UPDATE account SET current = " + (currentA - amount)
        + " WHERE accountId = 001");                    <-- 1 Do work

try {
    stat.execute("UPDATE account SET current = " + (currentB + amount)
      + " WHERE accountId = 002");
} catch (Throwable e) {
    stat.execute("UPDATE account SET current = " + (currentA + amount)
            + " WHERE accountId = 001");                <-- 2 Undo work
}
```

You only need to compensate for the first statement ❶, and only if the second statement fails, by doing the opposite of what the first statement has done ❷. In this case, this would be to credit the same amount that was debited originally. Why don't you need to compensate for the second statement? The reason is that if the second statement is successful, then both statements were executed successfully and there's nothing to compensate for. Being able to either execute both statements successfully or execute neither is called *atomicity*.

**DEFINITION** *Atomicity* guarantees that all of the actions of a task are executed successfully or none at all.

This compensation approach has two problems. First, what happens if the "compensation work" itself ❷ fails also? Then you wouldn't be able to undo the work and would therefore still lose consistency. Furthermore, consider if instead of 2 statements you had 10 statements; in this case, you'd have to code 9 compensating actions, as shown in figure 8.1.

Programming compensating actions isn't trivial, it's cumbersome, and it's prone to mistakes. Fortunately, there's a better way: the use of transactions, which you'll see in the next section.

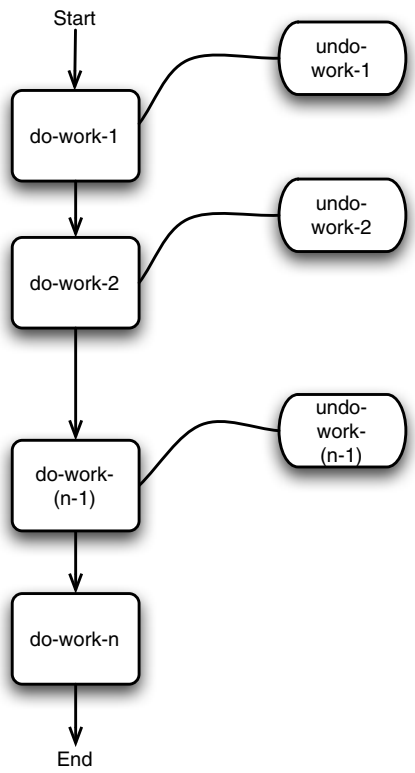


**Figure 8.1 For each do-work, you need a corresponding undo-work, with the exception of the last one.**

## 8.2 Transactions

Transactions are a programming abstraction representing a unit of work. A transaction may involve several actions, but it guarantees that either all of the actions are executed atomically or none are executed. A transaction also provides isolation, in the sense that from the outside a transaction is seen as an isolated single unit of work, even though internally it may encompass more than one activity. Finally, transactions are durable; that is, when they're terminated, their changes are guaranteed not to be lost.

> **DEFINITIONS** *Isolation* allows actions to be performed independently without impacting each other. *Durability* guarantees that the results of these actions aren't lost.

Transactions are described by three main operations:

- `begin()`
- `commit()`
- `rollback()`

Intuitively, you start a transaction by invoking the `begin()` operation. Following that, you can do all the work that's involved as part of that transaction. Finally, when you've finished, you can invoke either `commit()` to end the transaction successfully or `rollback()` to discard the work that has been done since the call to `begin()`.

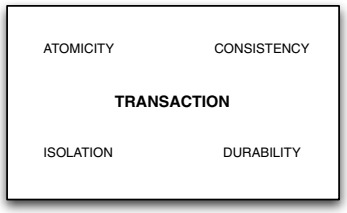


**Figure 8.2 A transaction possesses the attributes of atomicity, consistency, isolation, and durability. These are commonly referenced collectively as ACID.**

A transaction guarantees all four attributes we've discussed so far: atomicity, consistency, durability, and isolation. These are collectively known as ACID, as shown in figure 8.2.

The following listing shows how you can use transactions to solve the atomicity problem described in the previous section.

**Listing 8.3 JDBC local transactions**

```
Connection conn =
    ds.getConnection();

conn.setAutoCommit(false);

Statement stat =
    conn.createStatement();

stat.execute("UPDATE account SET current = " + (currentA - amount)
   + " WHERE accountId = 001");

stat.execute("UPDATE account SET current = " + (currentB + amount)
   + " WHERE accountId = 002");

conn.commit();
```

1 Start transaction

2 Do work

3 End transaction

By default, a JDBC connection commits for every statement executed. Setting the `AutoCommit` property to `false` (1) changes this. This causes a new transaction to be started when a statement is created. Having started a transaction, you can now safely do all the work that needs to be done. In this case, you execute both debit and credit statements (2). Finally, you commit (3). The `commit` operation serves as a demarcation; it tells the transaction that all statements used so far must be executed atomically. What happens if the second statement fails to execute? If this happens, then the infrastructure will automatically make sure that the first statement is rolled back, that is, undone. A user may also explicitly call the `rollback` method and force the undoing of the whole transaction.

In our example, a single resource was involved, an RDBMS data source. Transactions that deal with a single resource are called local transactions. In the next section, let's consider a slightly more complicated scenario involving more than one resource.

### 8.2.1 Global transactions

Again, let's consider our banking application, but now let's investigate the case where each account is kept in a different RDBMS, as shown in the following listing.

**Listing 8.4 Error-prone JDBC global transactions**

```
DataSource ds1 = ...
DataSource ds2 = ...

Connection conn1 =
    ds1.getConnection();

Connection conn2 =
    ds2.getConnection();

conn1.setAutoCommit(false);
conn2.setAutoCommit(false);

Statement stat1 =
    conn1.createStatement();

Statement stat2 =
    conn2.createStatement();

stat1.execute("UPDATE account SET current = " + (currentA - amount)
   + " WHERE accountId = 001");

stat2.execute("UPDATE account SET current = " + (currentB + amount)
   + " WHERE accountId = 002");

conn1.commit();
conn2.commit();
```

(1) Separate connections for each RDBMS

(2) Do all the work

(3) End first transaction

(4) End second transaction

When dealing with two RDBMSs, you retrieve two connections, one for each system (1). Then you perform the update operations on each account (2), one in each connection, and then commit both.

At first, this seems like it would work, but what happens if the application fails after the first commit ❸ but before the second commit ❹? If this unfortunate event happens, you'd be out of luck and would have decremented account 001 and never incremented adequately account 002.

When dealing with more than one resource, you need a mediator that coordinates the work across all resources. This is the role of the `TransactionManager`.

Things get a bit more involved in this case, as shown in the following listing.

**Listing 8.5 JDBC global transactions**

```
ServiceReference serviceReference =
    context.getServiceReference("javax.transaction.TransactionManager");

TransactionManager tm =
    (TransactionManager)
        context.getService(serviceReference);

XADataSource ds1 = ...
XADataSource ds2 = ...

XAConnection xaConn1 =
    ds1.getXAConnection();

XAConnection xaConn2 =
    ds2.getXAConnection();

tm.begin();

Transaction transaction = tm.getTransaction();
transaction.enlistResource(xaConn1.getXAResource());
transaction.enlistResource(xaConn2.getXAResource());

Connection conn1 = xaConn1.getConnection();
Connection conn2 = xaConn2.getConnection();

Statement stat1 =
    conn1.createStatement();

Statement stat2 =
    conn2.createStatement();

stat1.execute("UPDATE account SET current = " + (currentA - amount)
   + " WHERE accountId = 001");

stat2.execute("UPDATE account SET current = " + (currentB + amount)
   + " WHERE accountId = 002");

tm.commit();
```

❶ Retrieve TransactionManager service

❷ Retrieve XADataSource

❸ Use XAConnection

❹ Begin transaction

❺ Enlist all resources

❻ Do work

❼ Commit transaction

You start by retrieving the `TransactionManager` service ❶, which can be done by using the OSGi service registry. Next, you retrieve a special kind of data source called `XADataSource` ❷. We'll discuss `XADataSource`s later, but for the time being just assume these to be sources that are able to participate in global transactions. `XADataSource`s

can be accessed by the usual means of going through OSGi's `DataSourceFactory` or, more specifically, by calling `DataSourceFactory.createXADataSource()`. Next, you retrieve an `XAConnection` for each separate RDBMS ❸.

You're now ready to start the transaction, which you do by invoking the `TransactionManager.begin()` method ❹. Immediately, a transaction is commenced and associated with the current thread. This new transaction can be retrieved by calling `TransactionManager.getTransaction()`. Note, though, that you must call `getTransaction()` from the thread that initially started the transaction.

> **NOTE** A transaction must only be executed within the context of a single thread at a time. This makes sense because otherwise, if multiple threads were executing work for the same transaction, you wouldn't be able to determine the order of the actions.

Using the `Transaction` object, you can now enlist all resources that want to participate in this global transaction, which in this case are the RDBMSs represented by the variables `xaConn1` and `xaConn2` ❺. You're now ready to do the actual work of executing the update statements across the databases ❻. Finally, after both updates have been done successfully, you call `commit()` on the current global transaction to terminate ❼. It's noteworthy to mention that you call `commit()` only once, rather than having a commit for each separate resource, as was done in listing 8.4.

In the end, the main difference in the case of global transactions is that you have to use a `TransactionManager` to enlist and coordinate the transactional resources, which in this case were the two JDBC data sources.

What registers the `TransactionManager` service itself? That's the role of the Java Transaction API (JTA) provider, as you'll see in the next section.

### 8.2.2 Transaction providers

The JTA provider is responsible for registering the `TransactionManager` service and the `UserTransaction` service. You've seen the use of the former one already; the latter will be described in future sections. There can be at most one JTA provider in the OSGi framework, which means that there's only a single instance of the `TransactionManager` and `UserTransaction` services. Therefore, you can retrieve these services from the OSGi registry by using the service interface, such as `TransactionManager`.

Are there any available OSGi-based JTA provider implementations that you can leverage? As usual, let's check by browsing well-known bundle repositories. For example, you can check in SpringSource's repository: http://www.springsource.com/repository/app/. Type `transaction manager` in the input box, and as of this writing you'll get the following implementation: Java Open Transaction Manager 2.0.10. It's great to be able to reuse software, isn't it? You'll note that JOTM has several dependencies, such as the `javax.transaction` API itself, so don't forget to retrieve these as well.

As you've seen, one of the tasks of the `TransactionManager` is to coordinate resources, but what exactly are these, and how do they coordinate in a manner that

we're able to accomplish atomicity across distributed resources? The answers to these questions are covered in the next section.

### 8.2.3 The two-phase commit protocol

To be able to reach an agreement across distributed resources, the `TransactionManager` uses what's known as a *consensus protocol*. As you've seen, the `TransactionManager` starts by having a list of all resources that wish to participate in the transaction. It then follows a two-step process:

1. The `TransactionManager` finds out if all resources are ready to commit. It does so by invoking the method `prepare()` on each resource.
2. If all resources acknowledge the `prepare()` method successfully, then the next step is to invoke the method `commit()` on each resource, informing the resource that it can commit its local part of the transaction. If at any point there's a failure, then the `rollback()` method is invoked on each resource, informing it to discard the changes.

Because of this two-phase approach, this protocol is also known as the *two-phase commit (2PC) protocol*. The Open Group organization specifies the 2PC protocol in the XA specification, so resources and connections that participate in the 2PC protocol are often called XA resources and XA connections. An example of this interaction between a `TransactionManager` and two `XAResource`s is illustrated in figure 8.3.

As has become evident, dealing with global transactions can be daunting. We'll look into ways of simplifying this in the next section.

## 8.3 Containers

Dealing with all the details of resource managers, transaction managers, and XA protocols can be quite overwhelming. Consider again our simple bank application; at the

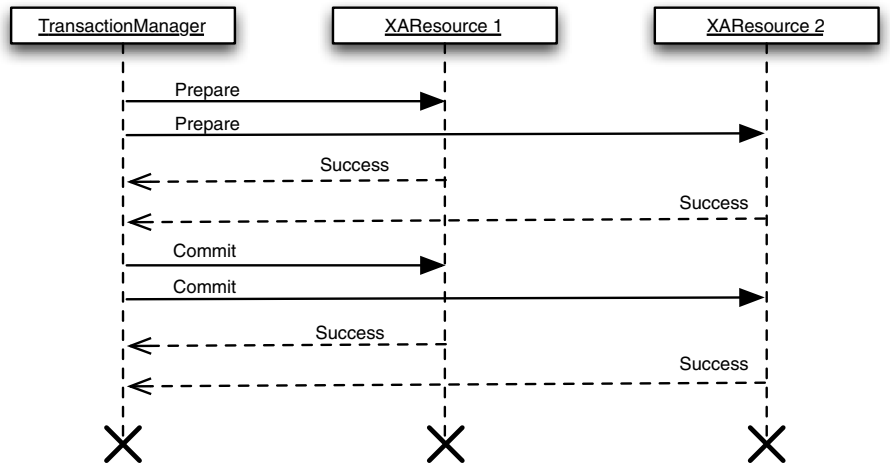


**Figure 8.3 `TransactionManager` informs resources first to prepare and then to commit.**

heart of it, we just want to debit one account and credit another one. In other words, our application logic shouldn't have to worry about enlisting resources and beginning and ending transactions. These activities are all potentially infrastructure work.

To simplify the development of transacted applications, we should refactor the transaction-handling code into some library or, more precisely, into the application container. Let's look at trivial example to illustrate the point. Suppose we define two Java annotations, `@Resource` and `@Transaction`, which respectively can be used to annotate methods that return XA resources and methods that perform work on these resources within the context of a transaction. These annotations can be implemented as follows:

```
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Transaction {
}

@Retention(value = RetentionPolicy.RUNTIME)
public @interface Resource {
}
```

The following listing uses these annotations to implement our banking example.

**Listing 8.6 Container-managed JDBC global transactions**

```
public void start(BundleContext bundleContext) throws Exception {
    ServiceReference [] serviceReferences =
      bundleContext.getServiceReferences(
          DataSourceFactory.class.toString(),
          "(" + DataSourceFactory.OSGI_JDBC_DRIVER_CLASS +
          "=org.apache.derby.jdbc.EmbeddedDriver)");

  if (serviceReferences != null) {
      DataSourceFactory dsf =
          (DataSourceFactory)
         bundleContext.getService(serviceReferences[0]);

      Properties props = new Properties();
      props.put(DataSourceFactory.JDBC_URL,
    "jdbc:derby:derbyDB1;create=true");

      ds1 =
          dsf.createXADataSource(props);

      props.put(DataSourceFactory.JDBC_URL,
         "jdbc:derby:derbyDB2;create=true");

      ds2 =
          dsf.createXADataSource(props);
  }
}

@Resource
public XAConnection getAccountToBeDebited() throws SQLException {
```

1 **Create XADataSource from factory**

2 **Enlist XAConnection**

```
    return ds1.getXAConnection();
}

@Resource
public XAConnection getAccountToBeCredited() throws SQLException {
    return ds2.getXAConnection();
}

@Transaction
public void doWork() throws SQLException {
    Connection conn1 = getAccountToBeDebited().getConnection();
    Connection conn2 = getAccountToBeCredited().getConnection();

    Statement stat1 =
        conn1.createStatement();

    Statement stat2 =
        conn2.createStatement();

    stat1.execute("UPDATE account SET current = " + (currentA - amount)
       + " WHERE accountId = 001");

    stat2.execute("UPDATE account SET current = " + (currentB + amount)
       + " WHERE accountId = 002");
}
```

❸ Execute in context of a transaction

You start off by retrieving the `DataSourceFactory` from the OSGi service factory, which you use to create two `XADataSource`s, one for each bank account ❶. You store these as field instances. Next, you annotate the methods `getAccountToBeDebited()` and `getAccountToBeCredited()` with the `Resource` annotation ❷, indicating that these methods are returning XA resources (or XA connections) that should be enlisted in the transactional work to be performed. As expected, these methods are returning the `XAConnections` created from the `XADataSource` retrieved previously. Finally, the method `doWork()` ❸ implements the actual application logic that's dealing with the bank account update. Note how the method includes no details related to transaction handling; not even a commit at the end is necessary.

> **NOTE** Application containers are powerful and useful. In this section, we're looking at a barebones example of how to implement a container that manages user transactions. We could have gone further and integrated the transaction handling with the persistence handling and avoided the user code having to deal with JDBC at all.

Next, you need to come up with a mechanism to inform the container that a particular bundle has transactional Java classes. To do this, follow a similar approach to that of JPA, which is the definition of a new manifest header entry, as follows:

```
Meta-Transaction: manning.osgi.TransactionalBankAcountTransfer
```

As expected, you can use the extender pattern to find the transactional Java classes and process them adequately. This is illustrated in the following listing.

**Listing 8.7 Transactional container extender**

```
public class TransactionalContainerActivator implements BundleActivator,
    BundleListener {

    TransactionManager tm;

    public void start(BundleContext context) throws Exception {

        tm = getTransactionManagerService(context);

        context.addBundleListener(this);
    }

    public void bundleChanged(BundleEvent event) {
        if (event.getType() == BundleEvent.STARTED) {
            try {
                String transactionalClassName =
                    (String) event.getBundle().
                    getHeaders().get("Meta-Transaction");

                Class<?> transactionalClass =
                    event.getBundle().
                        loadClass(transactionalClassName);

                Object transactionalObject =
                    transactionalClass.newInstance();

                List<XAResource> resourcesToEnlist =
                    new LinkedList<XAResource>();

                Method [] methods =
                    transactionalClass.getMethods();

                Method doWorkMethod = null;

                for (Method method : methods) {
                    if (method.getAnnotation(Resource.class)
                      != null) {
                        Object obj =
                            method.invoke(transactionalObject, new Object[]{});

                        if (obj instanceof XAConnection) {
                            resourcesToEnlist.add(
                            ((XAConnection) obj).getXAResource());
                        } else if (obj instanceof XAResource) {
                            resourcesToEnlist.add((XAResource) obj);
                        } else {
                            throw new IllegalStateException("Missing Resource
                            ➥ annotation");
                        }
                    }

                    if (method.getAnnotation(Transaction.class)
                      != null) {
                        doWorkMethod = method;
                    }
                }
```

1. Retrieve TransactionManager
2. Register BundleListener
3. Check when bundle is started
4. Retrieve Meta-Transaction header entry
5. Load specified class
6. Process Resource annotation
7. Process Transaction annotation

```
            doTransactionalWork(transactionalObject, resourcesToEnlisted,
                    doWorkMethod);                                    // 8 Execute transactional work

        } catch (Exception e) {
            // Log
        }
    }

    // ...
}
```

You start by retrieving the `TransactionManager` service (1) and registering a bundle listener into the OSGi framework (2). The `TransactionManager` service is kept as a field instance, because you'll use it later when processing the user's transactional class.

The `BundleListener` verifies whether a bundle that's being started (3) contains the `Meta-Transaction` header entry (4), in which case it loads the specified class using the client bundle (5) to check the annotations it uses. If any of its methods declare a `Resource` annotation (6), then you invoke such a method by instantiating a new object instance and storing the return value, which must be either an `XAResource` or an `XAConnection`. Likewise, you look for a method that's annotated with the `Transaction` annotation (7). Finally, you gather all of these findings and invoke the method `doTransactionalWork()` (8), which is explained in the next listing.

**Listing 8.8 `doTransactionalWork` method**

```
private void doTransactionalWork(Object transactionalObject,
        List<XAResource> resourcesToEnlist, Method doWorkMethod)
        throws Exception {
    if (doWorkMethod != null) {
        tm.begin();                                                   // 1 Begin transaction

        javax.transaction.Transaction transaction =
tm.getTransaction();

        for (XAResource resource : resourcesToEnlist) {               // 2 Enlist all resources
            transaction.enlistResource(resource);
        }

        doWorkMethod.invoke(transactionalObject,                      // 3 Invoke client code
          new Object[]{});

        tm.commit();                                                  // 4 Commit transaction
    } else {
        throw new IllegalStateException("Missing Transaction annotation");
    }
}
```

The `doTransactionWork()` method does the bulk of the work relating to handling the user's transaction. You start by commencing the transaction using the `TransactionManager` service (1), and then you iterate through all the XA resources you retrieved from the methods that had been annotated with the `Resource` annotation and enlist each of them in the currently running `Transaction` object (2). You're now ready to

invoke the client's code ❸, which implements the application logic. But this application logic code can avoid becoming cluttered with the transaction-handling details, as in listing 8.6. Finally, after the client's code returns, you commit the transaction ❹.

In summary, note how the code in listing 8.5 is mostly refactored into an application piece, which is in listing 8.6, and a container (infrastructure) piece, which is in listings 8.7 and 8.8.

### Distilling the extender pattern

Why do we need a `Meta-Transaction` header entry to begin with? Couldn't we just have our `TransactionalExtenderActivator` class blindly go through all the resources of all started bundles looking for those classes that have been tagged with our annotations?

In addition to the obvious performance downside, another reason to avoid this approach is that the extender class becomes tightly coupled with the client bundles. Instead, it's better to define an explicit contract between the extender and the clients, which is the role of the manifest header entry.

So there you have it; we've implemented our own naïve but functional container that provides managed transactions to our client bundles. Following are some ideas for improvements:

- Similarly to JPA, change the `Meta-Transaction` manifest header entry to point to an XML file rather than the Java class name.
- Integrate JPA into the container, so that it can provide not only managed transactions but also managed persistence.
- Similar to the JEE containers, you could support other transaction attributes, such as joining to some other running transaction, rather than always starting a new transaction for your transactional Java classes.

These are just some possible ideas. Now, let's take a step back and consider the interaction between the bundles involved. There's a client bundle, which contains the application logic, the container bundle, which manages the transaction on behalf of the client bundle, and the JTA provider bundle, which provides the implementation of the `TransactionManager` itself. Does this pattern of interactions, depicted in figure 8.4, remind you of anything?

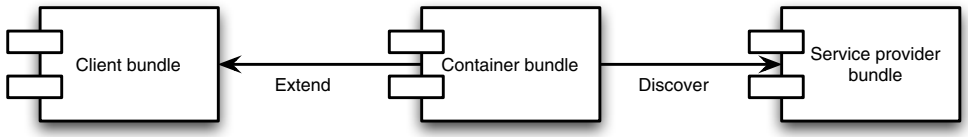


**Figure 8.4 The client bundle is extended by the container bundle. The container bundle retrieves infrastructure service from the service provider bundle.**

Yes, it's exactly how the JPA feature works. JPA similarly has a client bundle, the JPA bundle itself, and the JDBC driver bundle involved in doing the actual persistence. The JPA bundle acts as a mediator, simplifying the client's bundle usage of the JDBC driver to achieve persistence. This is the same role as our `TransactionalContainer-Activator` class; it hides the usage of the `TransactionManager` from the client bundle containing the application logic.

When well designed, application containers are just that: mediators between a client and an infrastructure service, which take care of abstracting the latter for the former. We can follow this same design pattern, which we'll denominate as the *application container design pattern*, for all types of infrastructure-related services, ranging from transactions and persistence to logging, security, and even domain-specific services, such as order processing and algorithmic trading.

**Application container design pattern**

*Intent*—Abstract application logic from having to know the details of infrastructure services, such as persistence and transactions.

*Participants*—Client bundle, container bundle, infrastructure-service provider bundle.

*Structure*—The container bundle mediates between the client bundle and the infrastructure-service provider bundle. This is done by having the mediator bundle reach out to the client bundle by means of the extender pattern and then using the registry design pattern to interact with the infrastructure-service provider bundle.

*Consequences*—The infrastructure-service provider bundles and client bundles are completely decoupled; the latter can use different programming models, such as annotations and declarative interfaces, without impacting the former and allowing the infrastructure services to evolve separately.

*Known uses*—JPA.

*Related patterns*—Extender design pattern, registry design pattern.

There's one final issue in the `TransactionalContainerActivator` class we've ignored so far, that of error handling. Next, we'll tackle this issue.

### 8.3.1 Error handling

Let's look at the core of the `doTransactionalWork()` method again:

```
tm.begin();

Transaction transaction = tm.getTransaction();

for (XAResource resource : resourcesToEnlist)
    transaction.enlistResource(resource);

doWorkMethod.invoke(transactionalObject, new Object[]{});

tm.commit();
```

What happens if the client code, which is invoked through the `doWorkMethod.invoke()` method, fails and raises an exception? You would have started a transaction but never invoked `commit`. The `TransactionManager` has no way of knowing about the client's exception and would still think that the transaction is valid and that an outcome is still coming. But eventually it would time out and realize something was wrong.

Instead of waiting for a timeout, the proper behavior for the container is to automatically roll back the current transaction in the case of a user error, as in the following listing:

**Listing 8.9 Rolling back a transaction**

```
tm.begin();
try {
    javax.transaction.Transaction transaction = tm.getTransaction();

    for (XAResource resource : resourcesToEnlist) {
        transaction.enlistResource(resource);
    }

    doWorkMethod.invoke(transactionalObject, new Object[]{});

    tm.commit();
} catch (Exception e) {
    tm.rollback();                                  <-- (1) Roll back if exception is raised
    // re-throw, or consume...
}
```

In this new version, you catch any exception that may have happened while the transaction is open and immediately roll back the transaction (1). This is particularly important any time you're dealing with client code, because you have no control over what the client application may be doing.

In this particular example, the client code raised an exception when things went amiss. What if the client decided to abort the transaction, not because of an exceptional situation but because of the application's logic? Would you need to raise an exception and cause it to be perhaps logged as an error, or is there a way of giving back a bit of the control related to the transaction handling to the user? We'll investigate this question in the next section.

### 8.3.2 User transactions

Using container-managed transactions, clients get shielded from the grinding details of the `TransactionManager` service, such as enlisting the XA resources. But there are cases when the client needs to have some control over the transaction handling.

For example, you may need to explicitly start and end the transaction, perhaps because you want to do some nontransactional work before getting into the transactional piece, as illustrated next:

```
nonTransactionalWork();

transactionalWork();

nonTransactionalWork();
```

Or, as you saw in the previous section, you may want to discard the transaction without having to raise an exception.

This is possible with the `UserTransaction` service. Similar to the `TransactionManager` service, the JTA provider registers the `UserTransaction` service, and you're guaranteed to have just one of it in the system. Using the `UserTransaction` service, a client can explicitly define the transaction boundaries by invoking the methods `begin()`, `commit()`, and even `rollback()`.

Let's look at an example of its usage. First, you retrieve the service as usual:

```
ServiceReference serviceReference =
      bundleContext.getServiceReference(
          UserTransaction.class.toString());

  UserTransaction ut =
      (UserTransaction) bundleContext.getService(serviceReference);
```

Next, you change the `doWork()` method of listing 8.6 to validate whether the account has enough money to be withdrawn for the transfer, as shown in the following listing.

**Listing 8.10 User transaction**

```
public void doWork() throws Exception {
    Connection conn1 = getAccountToBeDebited().getConnection();
    Connection conn2 = getAccountToBeCredited().getConnection();

    conn1.setAutoCommit(false);
    conn2.setAutoCommit(false);

    Statement stat1 =
        conn1.createStatement();

    Statement stat2 =
        conn2.createStatement();

    stat2.execute("UPDATE account SET current = " + (currentB + amount)
       + " WHERE accountId = 002");

    if (currentA - amount < 0.0)
        ut.rollback();                                  (1) Roll back current transaction

    stat1.execute("UPDATE account SET current = " + (currentA - amount)
       + " WHERE accountId = 001");

}
```

If the account doesn't have enough money to be withdrawn for the transfer, you roll back the whole transaction (1).

The OSGi framework provides us with one advantage that may have gone unnoticed here. In other platforms, the `UserTransaction` service would have been handled for the application by means of the container. In this case, the application has the flexibility of getting this service itself, when and if needed. Table 8.1 summarizes the different flavors of transaction handling we've discussed so far.

**Table 8.1 Transaction-handling styles**

| Characteristics | Advantages | Disadvantages |
|---|---|---|
| Compensation handling | JTA provider not needed | Error prone; does not scale |
| `TransactionManager` | Most flexible | Client has to enlist resources and control transaction directly |
| `UserTransaction` | Some flexibility; user can determine demarcations | Client has to retrieve `UserTransaction` service |
| Container managed | Client is shielded from all complexity | Least flexible approach |

Transactions are a powerful concept, because they provide a simple abstraction to deal with the complex problem of achieving atomicity across diverse actions, a common enough problem for most enterprise applications. Furthermore, OSGi improves this by allowing us to work with different transaction-handling approaches, varying from container-managed transactions to letting the client use the complete service.

## 8.4 Summary

It's not always enough to make a single action durable. There are cases, such as the transfer of money from one bank account to another, where several actions need to be made durable in an atomic fashion. That is, either all of them happen or none of them happen; otherwise, the application consistency may be lost.

There are several methods for achieving atomicity, one being the definition of an undo-work for each do-work. But this approach doesn't scale. Another approach is the use of transactions, which by definition are ACID (atomic, consistent, isolated, and durable).

Transactions provide three main operations: `begin`, `commit`, and `rollback`. Transactions can be local, if they involve a single resource, or global, if they involve several resources. When they're global, a `TransactionManager` is needed to coordinate consensus across all of the resources.

The JTA provider is responsible for registering a `TransactionManager` service and a `UserTransaction` service in the OSGi registry.

The `TransactionManager` service can be unyielding to use, so it's not uncommon for a container bundle to abstract it for a client bundle. The container bundle can extend the client bundle, for example, by providing annotations, and therefore prevent the client bundle from having to deal directly with the `TransactionManager` service. But if the client bundle needs additional flexibility, such as defining its own transaction demarcations, then it can interact directly with the `UserTransaction` service, which is simpler to use.

As you've seen, to support transactions it's often necessary to coordinate across distributed resources. But how do you find these resources to begin with? Finding resources in enterprise-wide *yellow pages* is the subject of our next chapter.

JAVA

# OSGi IN DEPTH

Alexandre de Castro Alves

OSGi is a mature framework for developing modular Java applications. Because of its unique architecture, you can modify, add, remove, start, and stop parts of an application without taking down the whole system. You get a lot of benefit by mastering the basics, but OSGi really pays off when you dig in a little deeper.

**OSGi in Depth** presents practical techniques for implementing OSGi, including enterprise services such as management, configuration, event handling, and software component models. You'll learn to custom-tailor the OSGi platform, which is itself modular, and discover how to pick and choose services to create domain-specific frameworks for your business. Also, this book shows how you can use OSGi with existing JEE services, such as JNDI and JTA.

## What's Inside

- Deep dives into modularization, implementation decoupling, and class-loading
- Practical techniques for using JEE services
- Customizing OSGi for specific business domains

Written for Java developers who already know the basics, *OSGi in Depth* picks up where *OSGi in Action* leaves off.

**Alexandre Alves** is the architect for Oracle CEP, coauthor of the WS-BPEL 2.0 specification, and a member of the steering committee of EP-TS.

For access to the book's forum and a free eBook for owners of this book, go to manning.com/OSGiinDepth

Free eBook
SEE INSERT

"Gives you a deep understanding of OSGi."
—From the Foreword by David Bosschaert, Red Hat

"Crucial if you're developing OSGi-based systems."
—Steve Gutz, IBM

"A thorough book on an increasingly important topic."
—Rick Wagner, Red Hat

"Alex has loads of experience to back up his words."
—Mike Keith, Oracle

"A jumpstart resource."
—Benjamin Muschko
Webs Inc.

ISBN 13: 978-1-935182-17-7
ISBN 10: 1-935182-17-X
57500
9 781935 182177

MANNING $75.00 / Can $78.99 [INCLUDING eBOOK]