

J2EE Without Application Server (J2EE is Easy!)

Copyright © 2006-2008, Atomikos

Introduction

J2EE-based applications have become very successful in the enterprise market, despite the notorious complexity of both the J2EE platform (the application server) and its proposed programming model (Enterprise JavaBeans or EJB). Thanks to modern notions like inversion-of-control (IoC) and aspect-oriented programming (AOP) represented in light-weight containers like the Spring framework, the programming model for J2EE can become a lot simpler and more elegant. Nevertheless, even with these tools the application server still remains an important source of complexity and cost. This article proposes a further simplification of J2EE, by showing a way to eliminate the overhead of the runtime platform: the application server. In particular, this paper shows that many applications no longer need an application server to run. As a result, J2EE applications become:

- **Easier to program:** no EJB plumbing code is required
- **Simpler:** there are no EJB classes or interfaces to inherit from
- **Easier to test:** your application and its tests can be run from within your integrated development environment (IDE)
- **Less resource-consuming:** you just need your objects, not the application server nor its objects
- **Easier to install:** there is no application-server specific installation to perform, and no extra XML files to add
- **Easier to maintain:** the overall process is much simpler, so maintenance is also easier

The unnecessary complexity of J2EE has been a show-stopper for many projects and teams. Even with the simplifications introduced in EJB 3, J2EE is still based on a complex platform (not in the least due to the application server model). Today, this complexity can often be avoided by the approach outlined in this article. In addition, it is possible to preserve or even improve the typical services like transactions and security. J2EE programming has never been more fun!

Example: The Message-Driven Bank

To illustrate our point, we will develop and install a complete sample application: a message-driven banking system. We will do this without the need for EJB or application server, while preserving the same transactional guarantees and (thanks to Spring) with an improved programming model based on "plain old Java objects" (POJO). In a later section we will generalize from message-driven to other architectures like web-based ones. Figure 1 shows the architecture of our demo application.

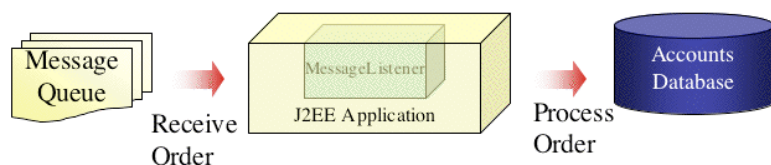


Figure 1. Architecture of the message-driven bank

In our scenario, we will process incoming bank orders from a JMS (Java Message Service) queue. The processing of an order involves updating the account database via JDBC (Java Database Connectivity). In order to avoid message loss or duplicates, we will use the Java Transaction API (JTA) and JTA/XA transactions to co-ordinate the updates: consuming a message and updating the database will happen in one atomic transaction. See the [resources](#) section for more information on why we need JTA/XA.

Coding the Application

The application will consist of two simple Java classes: `Bank` (a data access object or DAO) and `MessageDrivenBank`. This is shown in Figure 2.

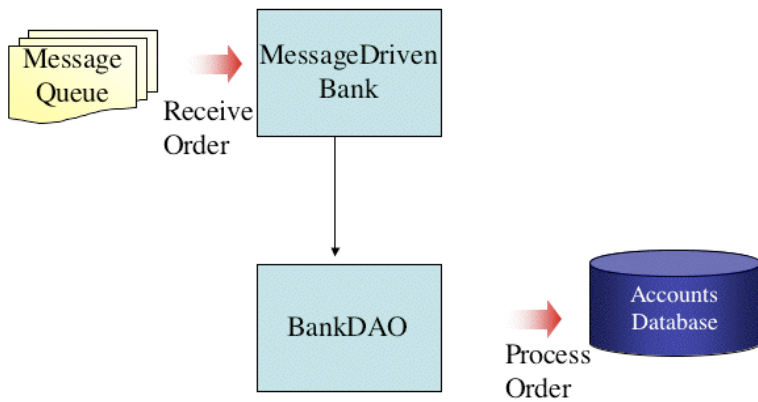


Figure 2. Classes for the message-driven bank

The Bank is a data access object that encapsulates the database access. The MessageDrivenBank is a message-driven façade and delegates to the DAO. Unlike with the classical J2EE approach, there are no EJB classes involved.

Step 1: Coding The Bank DAO

The source code for Bank is relatively straightforward JDBC, as shown below. I should point out that cleaning up resources and exception handling have been kept to a minimum for reasons of clarity.

```

package jdbc;

import javax.sql.*;
import java.sql.*;

public class Bank
{
    private DataSource dataSource;

    public Bank() {}

    /**
     * This method allows the DataSource to be set.
     * Spring's configuration facilities will call
     * this method for us.
     */

    public void setDataSource ( DataSource dataSource )
    {
        this.dataSource = dataSource;
    }

    //
    //Utility methods
    //

    private DataSource getDataSource()
    {
        return this.dataSource;
    }

    private Connection getConnection()
    throws SQLException
    {
        Connection ret = null;
        if ( getDataSource() != null ) {
            ret = getDataSource().getConnection();
        }
        return ret;
    }

    private void closeConnection ( Connection c )
    throws SQLException
    {
        if ( c != null ) c.close();
    }
}
  
```

```

}

public void checkTables()
    throws SQLException
{
    Connection conn = null;
    try {
        conn = getConnection();
        Statement s = conn.createStatement();
        ResultSet rs = null;

        try {
            rs = s.executeQuery ( "select * from Accounts" );
        }
        catch ( SQLException ex ) {
            //table not there => create it
            System.err.println ( "Creating Accounts table..." );
            s.executeUpdate ( "create table Accounts ( " +
                " account VARCHAR ( 20 ), owner VARCHAR(300), balance DECIMAL (19,0) )" );
            rs = s.executeQuery ( "select * from Accounts" );
        }
        if ( ! rs.next() ) {
            //empty table -> fill it
            for ( int i = 0; i < 100 ; i++ ) {
                s.executeUpdate ( "insert into Accounts values ( " +
                    "'account"+i+"', 'owner"+i+"', 10000 )" );
            }
        }
        s.close();
    }
    finally {
        closeConnection ( conn );
    }
    //That concludes setup
}

//
//Business methods are below
//
public long getBalance ( int account )
    throws SQLException
{
    long res = -1;
    Connection conn = null;
    checkTables();
    try {
        conn = getConnection();
        Statement s = conn.createStatement();
        String query = "select balance from Accounts where account='"
            +account+"account+'";
        ResultSet rs = s.executeQuery ( query );
        if ( rs == null || !rs.next() )
            throw new SQLException ( "Account not found: " + account );
        res = rs.getLong ( 1 );
        s.close();
    }
    finally {
        closeConnection ( conn );
    }
    return res;
}

public void withdraw ( int account , int amount )
    throws Exception
{
    Connection conn = null;
    try {
        conn = getConnection();
        Statement s = conn.createStatement();
        String sql = "update Accounts set balance = balance - "
            + amount + " where account ='account"+account+"'";
        s.executeUpdate ( sql );
        s.close();
    }
    finally {
        closeConnection ( conn );
    }
}
}
}

```

Note that there are no dependencies on EJB, or anything else application-server specific. Indeed, this is pure Java code that can run in any J2SE (Java Standard Edition) environment.

You should also note that we use the generic `DataSource` interface from JDBC. This means that our class is independent of the actual JDBC vendor class. You might be wondering how this is tied to your particular DBMS (database management system) vendor's JDBC implementation. This is what the Spring framework does (this technique is called **dependency injection**; the `DataSource` object is supplied by Spring when it calls the `setDataSource` method during the startup phase of our application). More information on Spring follows in the next sections. If we were using an application server then we would have to resort to JNDI (Java Naming and Directory Interface) lookups instead.

Instead of accessing JDBC directly, we could also use Hibernate or a JDO (Java Data Objects) tool to do the persistence for us. Again, this would be free of any EJB-related code.

Step 2: Configuring The BankDAO

We will use the Spring framework to configure our application. Spring is not strictly necessary, but the advantage is that we will be able to declaratively add services like transactions and security to our Java objects. This is similar to what the application server allows for EJB, only in our case it is going to be a lot easier. Spring also allows us to decouple our classes from the actual JDBC driver implementation: it is the Spring utility that will configure the driver (based on our XML configuration data) and supply it to the `BankDAO` object (the dependency injection principle). This keeps our Java code clean and focused. The Spring configuration file for this step is shown next.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<bean id="datasource"
    class="com.atomikos.jdbc.nonxa.NonXADataSourceBean">
    <property name="user" value="sa"/>
    <property name="url" value="jdbc:hsqldb:SpringNonXADB"/>
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="poolSize" value="1"/>
    <property name="connectionTimeout" value="60"/>
</bean>

<bean id="bank" class="jdbc.Bank" init-method="checkTables">
    <property name="datasource" ref="datasource" />
</bean>

</beans>
```

This XML file contains the configuration settings for two objects: the `DataSource` needed to access the database and the `Bank` object that uses this `datasource`. The following are some fundamental tasks taken care of by Spring:

- Creation of objects ("beans") needed by the application (i.e., the bank and the `datasource`). The class names for these objects are given in the XML file, and in our case they need to have a public no-argument constructor (Spring also allows for constructor arguments but that requires a slightly different XML syntax). The objects are named (via the `id` attribute in XML) so we can later refer to them. The `id` also allows our application to retrieve the configured objects it needs.
- Initialization of these objects via properties whose values are supplied in XML. The property names in the XML file should correspond to a `setxxx` method in the referred class.
- Linking objects together: a property can be a reference to another object (such as the `datasource` in our example). References are made by `id`.

Note that in anticipation of our next step, we have chosen to configure a JTA-enabled `datasource` (supplied by Atomikos ExtremeTransactions, the enterprise-capable and J2SE-compatible JTA product we will use for our application). For simplicity of the demo, the underlying DBMS is HypersonicSQLDB (which does not require any special installation steps — it runs from within its jar file just like the JTA and Spring). However, for increased reliability it is strongly recommended that you use an XA-capable DBMS and ditto JDBC drivers. Without XA, your application will not be recoverable after a crash or restart. In the [resources](#) part of this article you can find a link to more information on transactions and XA, and in what cases you need them. As a practical exercise you could try to switch

HypersonicSQLDB for FirstSQL, an easy to install XA-compliant DBMS. Alternatively, any other enterprise-ready and XA-capable DBMS will do as well.

Step 3: Testing The BankDAO

Let's test our code then (extreme programmers would write the tests first, but for the sake of clarity we delayed this step until now). A simple unit test can be found below; this test can be run from within your standard IDE, no application server is involved. The unit test is designed as a mini-application on its own: it uses Spring to retrieve a configured Bank object to test (this is done in the `setUp` method). Note that the test uses explicit transaction demarcation: a transaction is started before each test and rollback is forced at the end of each test. This is a handy way to minimize the effects of testing on the database data.

```
package jdbc;
import com.atomikos.icatch.jta.UserTransactionImp;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import java.io.FileInputStream;
import java.io.InputStream;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class BankTest extends TestCase
{
    private UserTransactionImp utx;

    private Bank bank;

    private FileSystemXmlApplicationContext factory;

    public BankTest ( String name )
    {
        super ( name );
        utx = new UserTransactionImp();
    }

    protected void setUp()
        throws Exception
    {
        //start a new transaction
        //so we can rollback the
        //effects of each test
        //in teardown!
        utx.begin();

        //open bean XML file
        factory =
            new FileSystemXmlApplicationContext ( "jdbc/config.xml" );

        //the factory is Spring's entry point
        //for retrieving the configured
        //objects from the XML file

        bank = ( Bank ) factory.getBean ( "bank" );
    }

    protected void tearDown()
        throws Exception
    {
        //rollback all DBMS effects
        //of testing
        utx.rollback();
        factory.close();
    }

    public void testBank()
        throws Exception
    {
        int accNo = 10;
        long initialBalance = bank.getBalance ( accNo );
        bank.withdraw ( accNo , 100 );
        long newBalance = bank.getBalance ( accNo );
        if ( ( initialBalance - newBalance ) != 100 )
            fail ( "Wrong balance after withdraw: " +
                newBalance );
    }
}
```

```

}

public static TestSuite suite()
{
    return new TestSuite ( jdbc.BankTest.class );
}
}

```

We will need JTA transactions to make sure that both the JMS and the JDBC are performed atomically (see the resources for more information on why this is so). **In general, you should consider JTA/XA whenever two or more connectors are needed (such as JMS and JDBC in our case).** Spring doesn't offer JTA transactions by itself (it needs a JTA implementation and normally delegates to an application server to do that). However, we are using an embeddable JTA implementation that even works on any J2SE platform.

The resulting architecture is show below in figure 3. The white boxes represent our application code.

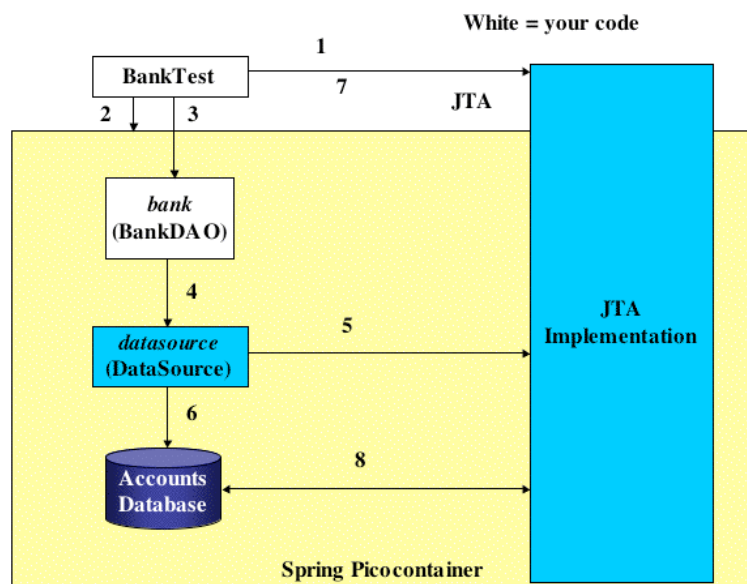


Figure 3. Architecture for the test

As you can see, the following happens when we run our test:

1. The `BankTest` starts a new transaction.
2. The test then retrieves the object named `bank` from the Spring runtime. This step triggers Spring's creation and initialization process.
3. The test calls the bank's method(s).
4. The bank invokes the object named `datasource`, which it has received from Spring via its `setDataSource` method.
5. The `datasource` is JTA-enabled and interacts with the JTA implementation to register with the current transaction.
6. JDBC statements interact with the accounts database.
7. When the method returns, the test calls `rollback` for the transaction.
8. JTA remembers the `datasource` and instructs it to rollback.

Step 4: Adding Declarative Transaction Management

Spring allows us to add declarative transaction management to any configured Java object. Suppose we want to make sure that the bank is always called with a valid transaction context. We do this by configuring a **proxy** object on top of the actual object. The proxy has the same interface as the actual object, so clients can use it just the same. The proxy can be configured to wrap each `BankDAO` method call in a transaction. The resulting configuration file is shown below. Don't be scared by the apparent volume of XML — most of the content can be reused by copy and paste into your own projects.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<bean id="datasource"
    class="com.atomikos.jdbc.nonxa.NonXADataSourceBean">

```

```

    <property name="user" value="sa"/>
    <property name="url" value="jdbc:hsqldb:SpringNonXADB"/>
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="poolSize" value="1"/>
    <property name="connectionTimeout" value="60"/>
</bean>

<bean id="jtaTransactionManager" class="com.atomikos.icatch.jta.UserTransactionManager"
    init-method="init" destroy-method="close">
    <!-- when close is called, should we force transactions to terminate or not? -->
    <property name="forceShutdown" value="true"/>
</bean>

<bean id="jtaUserTransaction" class="com.atomikos.icatch.jta.UserTransactionImp"/>

<bean id="springTransactionManager"
    class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManager" ref="jtaTransactionManager"/>
    <property name="userTransaction" ref="jtaUserTransaction"/>
</bean>

<bean id="bankTarget" class="jdbc.Bank" init-method="checkTables">
    <property name="dataSource" ref="datasource" />
</bean>

<bean id="bank" class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="springTransactionManager" />
    <property name="target" ref="bankTarget" />
    <property name="transactionAttributes">
        <props>
            <prop key="*">PROPAGATION_REQUIRED, -Exception</prop>
        </props>
    </property>
</bean>
</beans>

```

This XML file tells Spring to configure the following objects:

1. The **datasource** needed to connect via JDBC.
2. The **jtaTransactionManager** and **jtaUserTransaction** objects have been added to prepare for Spring's configuration for JTA transactions.
3. The **springTransactionManager** object has been added to tell Spring it needs to use JTA.
4. The BankDAO has been renamed to **bankTarget** (for a reason explained below).
5. The **bank** object has been added to wrap transactions around all the methods of the bankTarget. We configured the bank object to use the springTransactionManager, meaning that all transactions will be JTA transactions. The transaction setting is PROPAGATION_REQUIRED for each method, and rollback is forced on any Exception.

Of all these objects, you can easily copy and paste jtaTransactionManager, jtaUserTransaction and springTransactionManager to other projects. The only application-specific objects are the datasource, the bankTarget and the bank. The bank object is interesting: it is in fact a **proxy** to the bankTarget; it assumes the same interface. The trick is the following: when our application asks Spring to configure and return the object called "bank", Spring will actually return the proxy (which looks exactly the same to our application) and this proxy will start/end transactions for us. This way, neither the application nor the bank class itself needs to know JTA! Figure 4 illustrates what we get at this stage.

White = your code

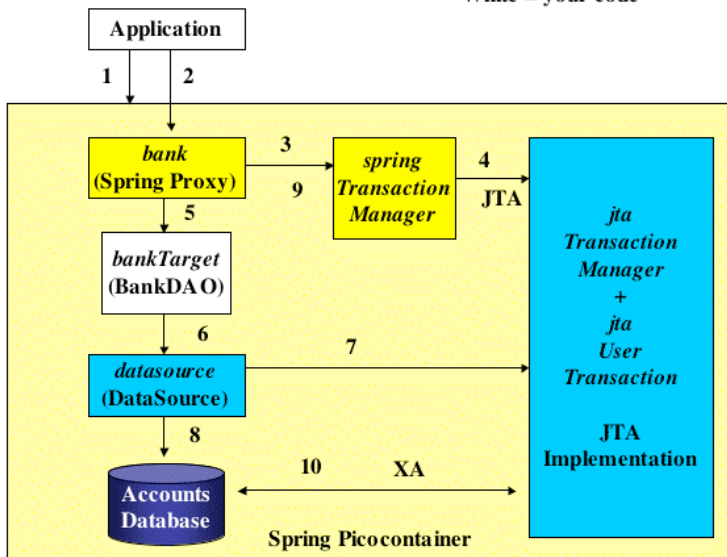


Figure 4. Architecture with declarative JTA transactions in Spring

Things now work as follows:

1. The application retrieves the object named **bank**. This triggers the Spring initialization process and the proxy is returned. To the application, this proxy looks and behaves like an instance of our **Bank** class.
2. When a method of the **bank** is called, this call goes via the proxy.
3. The proxy uses the **springTransactionManager** to create a new transaction.
4. The **springTransactionManager** was configured to use **JTA**, so it delegates to the **JTA**.
5. The call is now forwarded to the actual **Bank**, named **bankTarget**.
6. The **bankTarget** uses the **datasource** it received from Spring.
7. The **datasource** registers with the transaction.
8. The database access happens via regular **JDBC**.
9. Upon returning, the proxy terminates the transaction: if no exception happened in the previous sequence then the termination instruction is **commit**. Otherwise, it will be **rollback**.
10. The transaction manager co-ordinates **commit** (or **rollback**) with the database.

How about testing at this stage? We can reuse the **BankTest** with its explicit transaction demarcation: because of the **PROPAGATION_REQUIRED** setting, the proxy will execute with the transaction context created in the **BankTest**.

Step 5: Coding The MessageDrivenBank

In this step we will add the **JMS** processing logic. In order to do this, we merely need to implement the **JMS MessageListener** interface. We also add the public **setBank** method to make Spring's dependency injection work. The source code is below.

```
package jms;

import jdbc.Bank;
import javax.jms.Message;
import javax.jms.MapMessage;
import javax.jms.MessageListener;

public class MessageDrivenBank
implements MessageListener
{
    private Bank bank;

    public void setBank ( Bank bank )
    {
        this.bank = bank;
    }

    //this method can be private
    //since it is only needed within
    //this class
    private Bank getBank()
    {
```

```

        return this.bank;
    }

    public void onMessage ( Message msg )
    {
        try {
            MapMessage m = ( MapMessage ) msg;
            int account = m.getIntProperty ( "account" );
            int amount = m.getIntProperty ( "amount" );
            bank.withdraw ( account , amount );
            System.out.println ( "Withdraw of " +
                amount + " from account " + account );
        }
        catch ( Exception e ) {
            e.printStackTrace();

            //force rollback
            throw new RuntimeException (
                e.getMessage() );
        }
    }
}

```

Step 6: Configuring The MessageDrivenBank

Here we configure our `MessageDrivenBank` to listen on a transactional (JTA-aware) `QueueReceiverSessionPool`. This gives us the same message guarantees as EJB (no message loss nor duplicate messages), but with simple POJO objects instead. When a `MessageListener` is plugged into the pool, the pool will make sure that messages are received within a JTA/XA transaction. Combined with a JTA/XA-capable JDBC datasource we get reliable messaging. The resulting Spring configuration can be found below...

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org
/dtd/spring-beans.dtd">

<!--
    NOTE: no explicit transaction manager bean
    is necessary
    because the QueueReceiverSessionPool will
    start transactions by itself.
-->

<beans>

<bean id="datasource"
    class="com.atomikos.jdbc.nonxa.NonXADataSourceBean">
    <property name="user" value="sa"/>
    <property name="url" value="jdbc:hsqldb:SpringNonXADB"/>
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="poolSize" value="1"/>
    <property name="connectionTimeout" value="60"/>
</bean>

<bean id="xaFactory" class="org.activemq.ActiveMQXAConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>

<bean id="queue" class="org.activemq.message.ActiveMQQueue">
    <property name="physicalName" value="BANK_QUEUE"/>
</bean>

<bean id="bank" class="jdbc.Bank" init-method="checkTables">
    <property name="dataSource" ref="datasource"/>
</bean>

<bean id="messageDrivenBank" class="jms.MessageDrivenBank">
    <property name="bank" ref="bank"/>
</bean>

<bean id="queueConnectionFactoryBean"
    class="com.atomikos.jms.QueueConnectionFactoryBean">

```


3. The `queueReceiverSessionPool` detects a new message on the queue.
4. The `queueReceiverSessionPool` starts a new transaction and registers with it.
5. The `queueReceiverSessionPool` calls the registered `MessageListener`, the **messageDrivenBank** in our case.
6. This triggers a call on the **bank**.
7. The bank uses the **datasource** to access the database.
8. The `datasource` registers with the transaction.
9. The database is accessed via JDBC.
10. When the processing is done, the `queueReceiverSessionPool` terminates the transaction. Unless there is a `RuntimeException`, the desired outcome is `commit`.
11. The transaction manager initiates two-phase commit with the message queue.
12. The transaction manager initiates two-phase commit with the database.

Step 7: Coding The Application

Since we don't use a container, we merely provide a Java application to bootstrap the entire banking system. Our Java application is very simple: it suffices to retrieve the configured objects (wired together by Spring during the read-in of the XML file). This application can run on any compliant JDK (Java Development Kit), and does not need an application server run.

```
package jms;

import java.io.FileInputStream;
import java.io.InputStream;
import org.springframework.context.support.FileSystemXmlApplicationContext;
import jdbc.Bank;

public class StartBank
{
    public static void main ( String[] args )
        throws Exception
    {
        FileSystemXmlApplicationContext factory =
            new FileSystemXmlApplicationContext ( args[0] );

        //retrieve the pool; this will also start the pool as specified in config.xml
        //by the init-method attribute!
        factory.getBean ( "queueReceiverSessionPool" );
        factory.registerShutdownHook();

        System.out.println ( "Bank is listening for messages..." );
    }
}
```

That's it! Isn't J2EE easy these days?

Reflections on Generalization

In this section we look at some additional concepts that are important in many J2EE applications. We will see that an application server is not really necessary for these either.

Clustering and Scalability

Robust enterprise applications typically need cluster capabilities in order to scale up and down. In the case of message-driven applications this is easy: we automatically get the inherent scalability of JMS applications. If we need more processing power, then we just need to add more processes that connect to the same JMS server. A useful measurement of server performance is the number of outstanding messages in the queues. In other situations like web-based architectures (see below) we can easily use the clustering ability of the web environment.

Method-Level Security

One of the classical arguments in favor of EJB is the ability to add method-level security. Although not shown in this article, it is possible to configure method-level security in Spring (similar to how we added method-level transaction demarcation).

Generalization to Non-Message-Driven Applications

The platform we have used can easily be integrated into any J2EE web application server, without the need for changing any of the source code (except the main application class). Alternatively, the back-end processing can be invoked via JMS; this keeps the web server responsive and independent of the latency caused by back-end processing. In any case, an EJB container is not needed in order to do container-managed transactions or to have container-managed security.

What about Container-Managed Persistence?

Existing and proven technologies such as JDO (Java Data Objects) or Hibernate don't necessarily need an application server either. In addition, these tools are dominating the managed persistence market already.

Conclusion

J2EE is easy without an application server, and possible today. That being said, there are certainly applications that can't be implemented without an application server (yet): for instance, if you need general JCA (Java Connectivity API) functionality then the platform we propose is not enough. However, this is likely to change since the benefits of not having to use an application server to develop, test and deploy are simply too great. More and more people are convinced that the future of J2EE is in a modularized "pick-what-you-need" architecture, as opposed to the monolithic application server-based approach of today. In such a scenario, J2EE developers will finally be freed from the restrictive harness that the application server and EJB impose.

Resources

- [The complete source code](#) for this article
- Guy Pardon's [presentation on Transactions in Spring](#) published at TheServerSide
- More information on [Atomikos ExtremeTransactions](#) and message-driven functionality without EJB
- The homepage of [Spring](#)
- More information on [JUnit](#)
- [FirstSQL](#) is an easy to install XA-compliant DBMS
- More information on [HSQLDB](#)
- More information on [ActiveMQ](#)

About Atomikos

Atomikos (<http://www.atomikos.com> - "Reliability Through Atomicity") is the industry leader in Java-based eXtreme Transaction Processing (XTP) with its next generation TP monitor software. For demanding service-oriented or event-driven architectures where data quality is mission-critical, Atomikos ExtremeTransactions guarantees data consistency everywhere, even in the presence of failures or crashes and beyond the limitations of J2EE.