Transactional J(2)EE Applications with Spring

Transactional POJO Programming

Guy Pardon (guy@atomikos.com)

Contents

- Part 1: Spring Essentials
- Part 2: Transactions in Spring
- Part 3: JTA Use Cases

Part 1

Spring Essentials

The Spring Framework

- Inversion of Control (IoC) library
  - Creation and configuration of objects
    - Keeps your application code free of class-specific configuration issues
    - Makes plain java components easier to write and use
  - Dependency injection: wiring objects together
    - Encourages programming against interfaces
  - Advantage: allows clean and focused java components
- Supports aspect-oriented programming (AOP)
  - Allows introduction of additional logic before/after a method call
  - Advantage: can replace typical appserver services
- Works both inside and outside of application server

Configuring a DataSource

```
<?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.SimpleDataSourceBean">
      <property name="uniqueResourceName">
            <value>XADBMS</value>
       </property>
      <property name="xaDataSourceClassName">
          <value>COM.FirstSQL.Dbcp.DbcpXADataSource</value>
       </property>
       ...
</bean>
</beans>
```

Ask Spring to create an object with this name and of this class, and to set the given properties

ATOMIKOS

A Client Component

```
<?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.SimpleDataSourceBean">
       ...
</bean>
<bean id="bank" class= "jdbc.Bank">
     ...
</bean>
</beans>
```

ATOMIKOS

Wiring Objects Together

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

<bean id="datasource"
      class="com.atomikos.jdbc.SimpleDataSourceBean">
      ...
</bean>

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

</beans>
```

Ask Spring to establish an association between the Objects called bank and datasource

ATOMIKOS

Bank Class: “setDataSource”

```
public class Bank
{
    private DataSource dataSource;

    public Bank() {}

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

     //the rest is pure JDBC code
     ...
}
```

Our Java code can now be independent of the actual JDBC vendor classes.

This is called by the Spring wiring runtime.

ATOMIKOS

Applications: Configuration No Longer Hard-Coded

```
public class Main
{
    public static void main ( String[] args )
    throws Exception
    {
        InputStream is = new FileInputStream("config.xml");
        XmlBeanFactory factory = new XmlBeanFactory(is);
        Bank bank = ( Bank ) factory.getBean ( "bank" );

        //the bank object is already configured and
      ready to use!!!
        bank.withdraw ( 10 , 100 );
        ...

    }
}
```

Just ask the Spring BeanFactory to use our XML configuration file.

We can ask Spring to return configured objects by their name (id)

ATOMIKOS

Spring Benefits

- Configuration is XML
  - Can be changed without recompilation
- Configuration is centralized in files
  - Easy to find and maintain
- Your Java code is pure and clean
  - Easier to write and maintain
  - More focused on what it needs to do
- Your Java code has less dependencies
  - Concrete class names are factored out
- We can control the wiring
  - We can insert proxies that add services
  - You source code does not need to know
- Enables J2EE without appserver

Transactions in Spring?

- Can we use Spring to manage transactions?
- Can we use the datasource multiple times and then still rollback in case of a failure?
- Can we rollback across multiple connectors?
- The rest of this presentation will discuss that

Part 2

Transaction Support in Spring

ATOMIKOS

Spring and Transactions

- Similar to EJB:
  - Declarative or programmatic demarcation
  - Several attributes for declarative transactions
- Different from EJB:
  - Flexibility to choose the underlying transaction strategy
  - JDBC, JTA, JDO, Hibernate, ...
  - This means that you don't necessarily need an application server to have transactions (J2EE without appserver)!

Transaction Strategies in Spring

- Strategy defines what underlying technology manages the transactions
- Single-connector strategies:
  - JDBC
  - Hibernate
  - JDO
  - Other
- Multiple-connector strategy:
  - JTA

Single-Connector Strategies

- The underlying transaction is actually a connector-specific transaction
  - JDBC: connection-level transaction
  - Hibernate: session-level transaction
  - JDO: session-level transaction

- Don't use these if you have more than one connector!
  - Only the JTA strategy is safe in that case

JDBC Strategy

- The JDBC strategy needs the actual connection to commit/rollback on
  - It must be aware of what connection your code is using
  - 1 connection per transaction, reused every time
- Consequently, your code must either:
  - Use Spring's JDBC utilities (introduces code dependency on Spring), or
  - Access the datasource via Spring's TransactionAwareDataSourceProxy (does not introduce code dependency on Spring: this is done via a proxy in the XML configuration)
- Similar implications apply to other single-connector strategies (but not to JTA strategy)

Spring XML Config?

- A datasource bean
  - Like before
- A proxy datasource
  - To wrap our datasource and return same connection within a transaction
  - Supplied to our bank
- An instance of the jdbc.Bank class
  - To do the JDBC
- The transaction strategy
- A proxy for the bank, that starts/ends a transaction for each method
  - To insert the declarative transaction code
  - Wired with the transaction strategy
- No changes in our source code!

Example: Declarative + JDBC Strategy (1)

```
<bean id="datasource"
      class="com.atomikos.jdbc.SimpleDataSourceBean">
      ...
</bean>

<bean id="dsProxy"
    class="org.springframework.jdbc.datasource.
    TransactionAwareDataSourceProxy">
    <property name="targetDataSource">
    <ref bean="datasource" /></property>
</bean>

<bean id="bankTarget" class= "jdbc.Bank">
      <property name="dataSource"><ref bean="dsProxy" />
          </property>
</bean>
```

Vendor-specific datasource setup.

Proxy required for JDBC strategy in Spring.

Our bank's JDBC is done via the proxy.

ATOMIKOS

Example: Declarative + JDBC Strategy (2)

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager"/>
    <property name="dataSource"><ref bean="datasource"></property>
</bean>

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

Clients of our bank will use a proxy that inserts transaction logic.

Transaction required, rollback on any exception.

ATOMIKOS

JTA Strategy

- In this case, the underlying transaction is a JTA transaction
- Essential if you have more than one connector to access
- This does not introduce any dependencies in your code
- The JTA strategy requires JTA-enabled DataSource instances
  - Like those available for the appserver, or
  - Like those supplied by Atomikos for standalone use

Example: Declarative + JTA Strategy (1)

```
<bean id="datasource"
      class="com.atomikos.jdbc.SimpleDataSourceBean">
      ...
</bean>

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

<bean id="atomikosTM"
      class="com.atomikos.icatch.jta.
      UserTransactionManager">
</bean>

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

Vendor-specific datasource setup (JTA enabled class).

Setup Atomikos standalone JTA.

Example: Declarative + JTA Strategy (2)

```
<bean id="transactionManager"
    class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManager"><ref bean="atomikosTM" /></property>
      <property name="userTransaction"><ref bean="atomikosUTX" /></property>
</bean>

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

Tell Spring to use the Atomikos JTA implementation.

ATOMIKOS

Using Other JTA Implementations

- Class org.springframework.transaction.jta.JtaTransactionManager has additional properties:
  - userTransactionName: JNDI name where usertransaction can be found (requires appserver)
  - transactionManagerName: JNDI name where transactionmanager can be found (requires appserver)
- Appserver-specific subclasses exist
  - Optimized for specific application servers
- Conclusion: in principle, any JTA implementation can be used with Spring
  - Use a standalone JTA if you want out-of-container applications

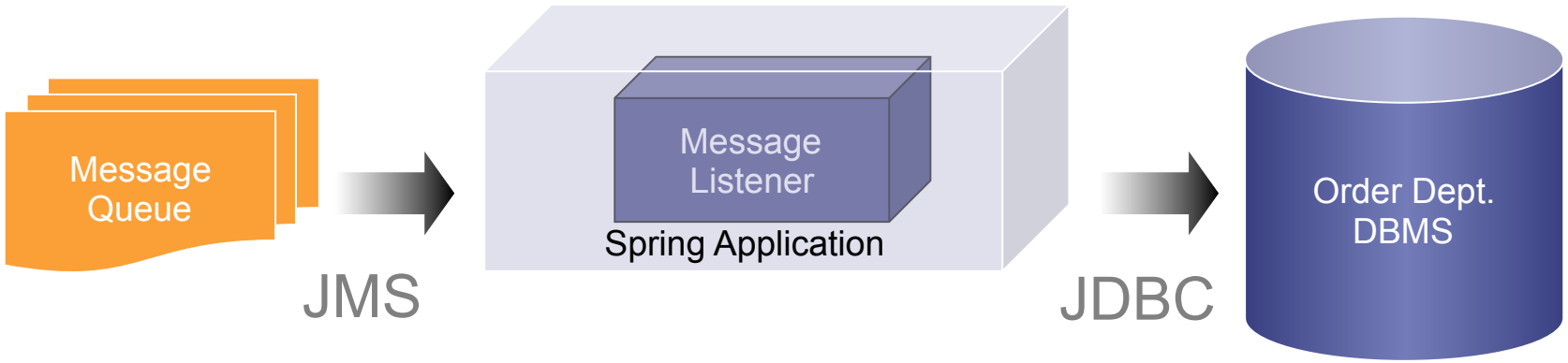
Transaction Attributes in Spring

- PROPAGATION_REQUIRED
- PROPAGATION_REQUIRES_NEW
- PROPAGATION_SUPPORTS
- PROPAGATION_NEVER
- PROPAGATION_MANDATORY
- PROPAGATION_NOT_SUPPORTED
- PROPAGATION_NESTED

Part 3

JTA Use Cases

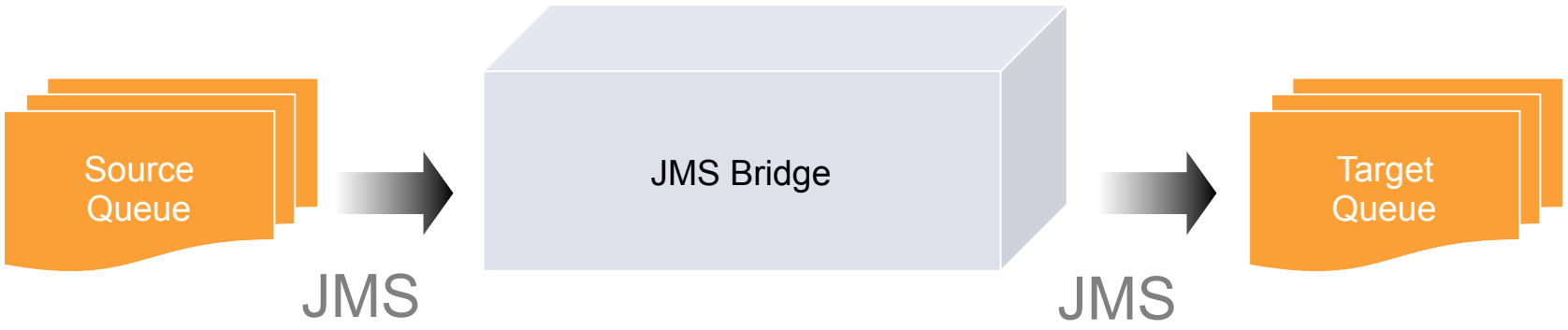
Use Case 1: From Queue to DB



Avoiding message loss and duplicates requires JTA!

ATOMIKOS

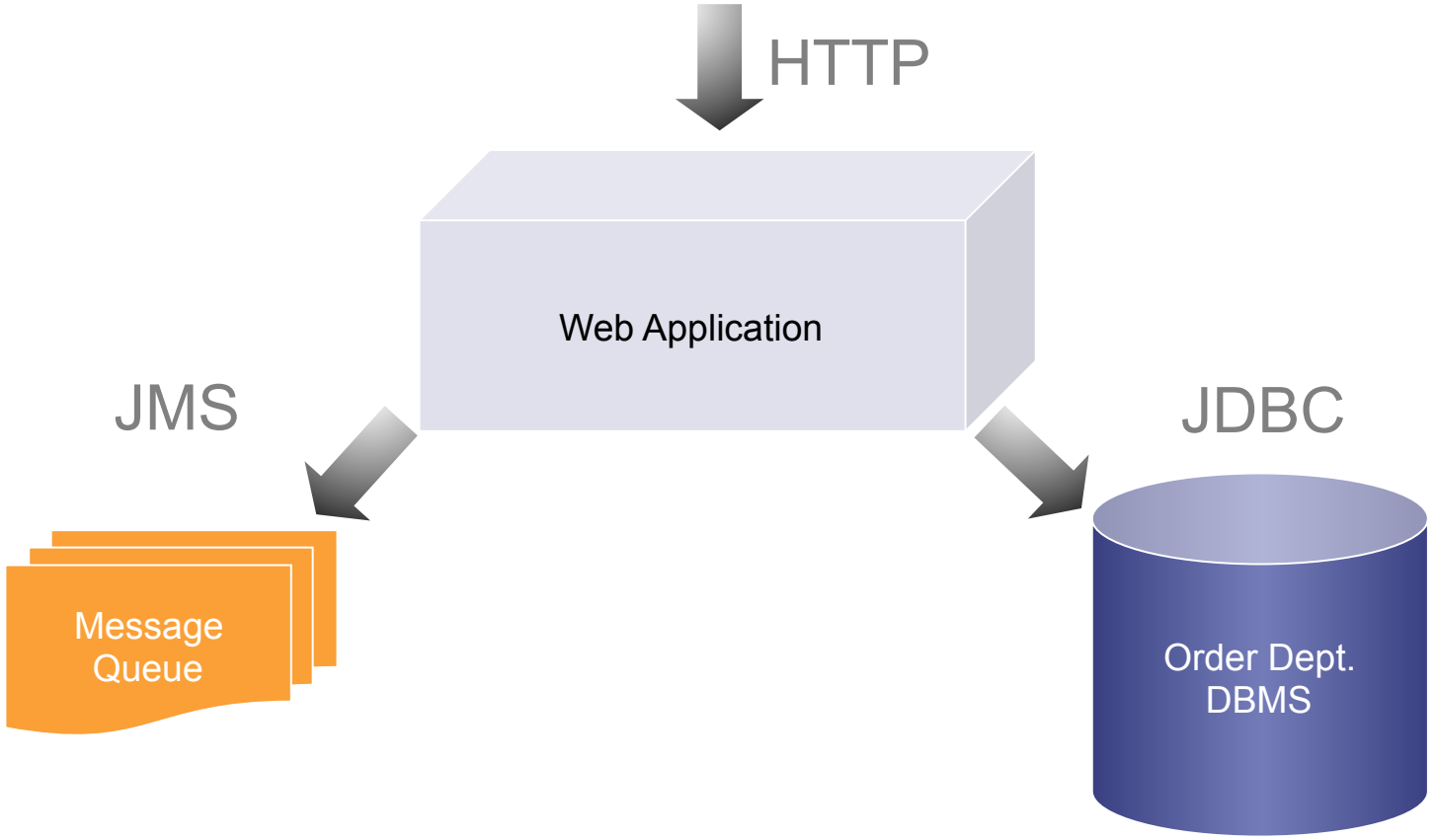
Use Case 2: From Queue to Queue



Avoiding message loss and duplicates requires JTA!

ATOMIKOS

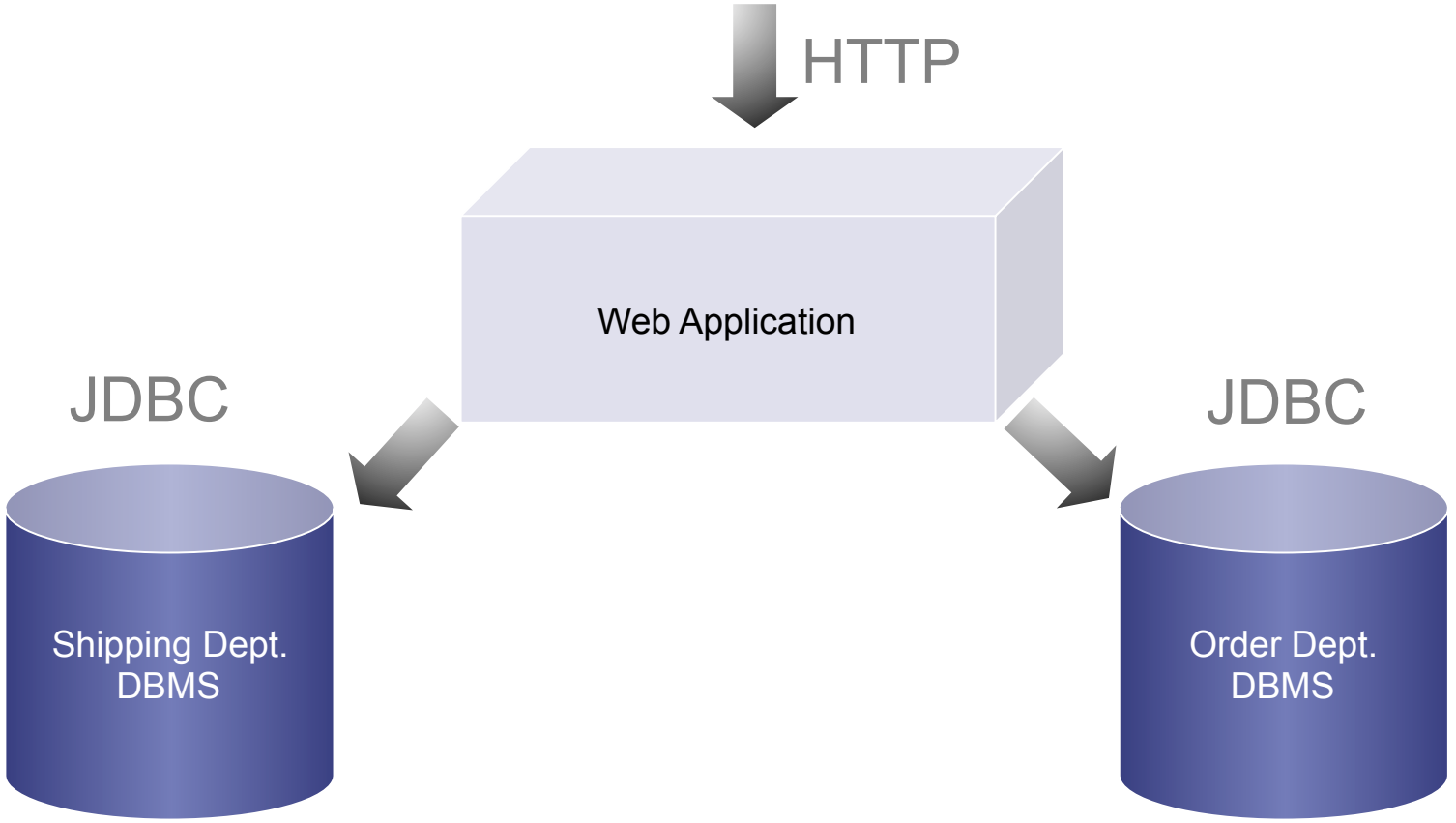
Use Case 3: From Web to DB and Queue



Consistent Processing Requires JTA!

ATOMIKOS

Use Case 4: Between 2 DBs



Consistent Processing Requires JTA!

ATOMIKOS

Conclusion: Spring Improves J2EE

- Enables J2EE without application server
  - Components are freed of the EJB harness
  - CMT is still possible
  - Managed security is also possible
  - Less hardware resources needed
  - End of the deployment descriptor nightmare
  - Easier to develop, maintain, test and install
- Overall project cost is much lower, without loss of reliability

More Information?

- Read more in this article:
  http://www.atomikos.com/Publications/J2eeWithoutApplicationServer

- Download Atomikos TransactionsEssentials
  - http://www.atomikos.com/Main/TransactionsEssentials
  - Transaction management for Spring and JSE
  - A jar file that enables JTA in all your applications
  - Includes connection pooling (JDBC/JMS)
  - Includes message-driven JMS support for Spring
  - Downloadable from http://www.atomikos.com

- Contact me at guy@atomikos.com