

Spring Framework12

How to use BTM as the transaction manager with Spring 2.x

These instructions have been verified against BTM 1.1.

Contents

- [Step 1: Copy the BTM jars](#)
- [Step 2: Configure connection pools beans](#)
- [Step 3: Configure BTM beans](#)
- [Step 4: Configure Spring PlatformTransactionManager](#)
- [Step 5: Configure declarative transaction management](#)

Step 1: Copy the BTM jars

Include the following jars from the BTM distribution into your classpath:

- btm-1.1.jar
- geronimo-spec-jta-1.0.1B-rc4.jar
- slf4j-jdk14.jar (or any other one available [here](#))

Step 2: Configure connection pools beans

The first things you will need to configure are the connection pools.

Here is a sample bean configuration using Embedded Derby:

```
<bean id="derbyDataSource"
class="bitronix.tm.resource.jdbc.PoolingData
Source" init-method="init"
destroy-method="close">
  <property name="className"
value="org.apache.derby.jdbc.EmbeddedXADataS
ource" />
  <property name="uniqueName" value="derby1"
/>
  <property name="minPoolSize" value="0" />
  <property name="maxPoolSize" value="5" />
  <property name="driverProperties">
    <props>
      <prop key="databaseName">users1</prop>
    </props>
  </property>
</bean>
```

API-created pools

Since the pools are created via the BTM API (ie: not with ResourceLoader) it is up to the API user to manage the lifecycle of the pools, mainly calling `init()` before usage and `close()` at shutdown.

This is why the two `init-method` and `destroy-method` attributes are set: to have Spring take care of that lifecycle.

Step 3: Configure BTM beans

The second thing you need to do is configure beans for BTM.

```
<!-- Bitronix Transaction Manager embedded
configuration -->
<bean id="btmConfig"
factory-method="getConfiguration"
class="bitronix.tm.TransactionManagerService
s">
  <property name="serverId"
value="spring-btm" />
</bean>

<!-- create BTM transaction manager -->
<bean id="BitronixTransactionManager"
factory-method="getTransactionManager"
class="bitronix.tm.TransactionManagerService
es" depends-on="btmConfig,derbyDataSource"
destroy-method="shutdown" />
```

Initialization ordering

The `TransactionManagerServices.getTransactionManager()` call will initialize BTM. Before this, all connection pools should have been created and the configuration properties should have been set.

This is why the `depends-on` attribute is set: the `BitronixTransactionManager` bean must not be created prior its configuration bean and the required connection pools.

If this is problematic for you, please let us know by voting for [BTM-4](#).

Step 4: Configure Spring PlatformTransactionManager

Next, you need to create a Spring [PlatformTransactionManager](#). There are many of them but the one we are interested in is the [JtaTransactionManager](#). This is required as Spring internally uses `PlatformTransactionManager` for all transactional work.

```
<!-- Spring JtaTransactionManager -->
<bean id="JtaTransactionManager"
class="org.springframework.transaction.jta.J
taTransactionManager">
  <property name="transactionManager"
ref="BitronixTransactionManager" />
  <property name="userTransaction"
ref="BitronixTransactionManager" />
</bean>
```

This is really all you need to get JTA support with BTM inside Spring. You could directly make use of the `JtaTransactionManager` bean in your code but there are more elegant solutions: using Spring's AOP support to get declarative transaction management.

Step 5: Configure declarative transaction management

This can easily be achieved thanks to Spring's [TransactionProxyFactoryBean](#).

The idea behind it is to wrap your bean with a Spring-generated proxy that will intercept calls and perform transaction management according to a configuration.

Here is short example:

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

This expects a `MyObject` bean to also be configured. You should then make use of the `MyObjectFacade` bean that will start a new transaction on any method call if no transaction is already running (the `<prop key="*">PROPAGATION_REQUIRED` piece), commit the transaction when the method returns or rollback the transaction if any exception is thrown (the `, -Exception</prop>` piece).

If you need more details on what can be done and how things are working refer to the [TransactionProxyFactoryBean](#) class javadoc.