Connection pools Workshop

Overview

In this workshop, first we configure the JDBCTemplate in the XML file.

For this we need to configure DataSource and an implementation of this DataSource in our file. We use the SimpleDriverDataSource class which is in Spring and it implements the DataSource interface.

Then we will use connection pools which are suitable for the production version and are faster than the SimpleDriverSource class which was suitable for the developer version.
We will configure this connection pool (the DBCP implementation) in our XML file.

Configuring JdbcTemplate in the xml file

In order to run the above code, we need to manage the connections.

First we should change the implementation of DAO in xml file:

```
<!-- Dao beans -->
<bean id="bookDao" class="se.yrgo.spring.data.BookDaoSpringJdbcImpl" >
</bean>
```

And we need to inject the *JdbcTemplate* into the constructor. In the xml file, we add the template bean:

```
<!-- Templates -->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
</bean>
```

To find the name of the package and what properties are needed, we go to docs.spring.io and search for JdbcTemplate:
https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/JdbcTemplate.html

In the *all methods* tab, we search for methods beginning with ‘set’. All of these methods are properties that we can configure.

Now we look at the constructors.There is one property that is required and we always need to set it. Therefore Spring has made this property to be injected through the constructor. This is *DataSource*. So we need to create an object of type DataSource and inject it into the template.

DataSource

DataSource is a part of standard Java. It is a wrapper class and has a method *getConnection*. Any class that implements this interface, is capable of returning a database connection using a kind of strategy which usually is some kind of a connection pool.
There is no class in standard Java that implements this interface. But database vendors would implement this interface.

Using SimpleDriverDataSource class in Spring

In Spring there is a class called *SimpleDriverDataSource* which implements this interface.

https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/datasource/SimpleDriverDataSource.html

There are four properties we should set for this class:

1. driverClass
2. url
3. user name
4. password

In the xml file:

```
<!-- DataSources -->
<bean id="dataSource"
        class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
         <property name= "driverClass" value="org.hsqldb.jdbcDriver" />
         <property name= "url" value
                     ="jdbc:hsqldb:file:database.dat;shutdown=true" />
         <property name = "username" value="sa"/>
         <property name="password" value ="" />
</bean>
```

You can find the driver name and database url, username and password form The *BookDaoJdbcImpl* class.

Inject the datasource into the constructor of jdbcTemplate

```
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
      <constructor-arg ref="dataSource" />
```

```
</bean>
```

Now we go to the graph in our *SpringExplorer* and see all of our beans in the graph.
We see that the *jdbcTemplate* is connected to the *dataSource* bean, but it is not connected to the DAO.

Inject the template into the constructor of the Dao bean

In Dao bean, we should inject the template into the constructor of the DAO.

```
<!-- Dao beans -->
<bean id="bookDao" class="se.yrgo.spring.data.BookDaoSpringJdbcImpl" >
      <constructor-arg ref="jdbcTemplate"/>
</bean>
```

Now if we go back to our graph, everything is connected.

This is the complete xml file:

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
            "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
    <!-- DataSources -->
    <bean id="dataSource" class=
             "org.springframework.jdbc.datasource.SimpleDriverDataSource">
      <property name= "driverClass" value="org.hsqldb.jdbcDriver" />
      <property name= "url" value
                ="jdbc:hsqldb:file:database.dat;shutdown=true" />
      <property name = "username" value="sa"/>
      <property name="password" value ="" />
       </bean>

      <!-- Templates -->
   <bean id="jdbcTemplate"
                class="org.springframework.jdbc.core.JdbcTemplate">
      <constructor-arg ref="dataSource" />
    </bean>

   <!-- Dao beans -->
   <bean id="bookDao" class="se.yrgo.spring.data.BookDaoSpringJdbcImpl" >
       <constructor-arg ref="jdbcTemplate"/>
   </bean>
```

```
   <bean id="bookService"
         class="se.yrgo.spring.services.BookServiceProductionImpl">
        <constructor-arg ref="bookDao"/>
   </bean>

   <bean id="accountsService"
            class="se.yrgo.spring.services.AccountsServiceMockImpl"/>
   <bean id="purchasingService" class=
          "se.yrgo.spring.services.PurchasingServiceImpl">
      <constructor-arg ref = "bookService"/>
      <constructor-arg ref = "accountsService" />
    </bean>
</beans>
```

Now if we run the client code, we should see the new book that we have created. We can delete the two database files and run the code again, to see if it creates the table in the beginning. And we run the code.

Using Connection Pools

The above implementation for the connection to the database is very slow. For production systems, we are going to use connection pools.
A connection pool is something that we install on our server and when we start the server, it automatically creates several database connections. The process of creating and opening the connections will be slow, but it will be only when the server starts up.

Now if the client wants to create a new book, it makes the call to the server, and since we have a connection pool and there are several connections open, the pool chooses one of the free connections and connects the client to this connection. And when the client is finished with this connection, this will not be closed and can be used for further service requests.

DBCP

We use *DBCP* implementation. Add the following to your pom file:

```
<!-- https://mvnrepository.com/artifact/commons-dbcp/commons-dbcp -->
<dependency>
     <groupId>commons-dbcp</groupId>
     <artifactId>commons-dbcp</artifactId>
```

```
      <version>1.4</version>
 </dependency>
```

Now we configure this connection pool in our xml file:

We change the class name in the *dataSource* bean. We can find the Java doc for DBCP from this link:
http://commons.apache.org/proper/commons-dbcp/apidocs/index.html

The name of the class is *BasicDataSource*.
We get the package name and in the xml file, we change the class name.

Then we need to configure the properties. As soon as we change the class name, we get an error by Spring Tool on the property line of *driveClass*. This is because this property is not called *driverClass* in this class.

To find the correct name, in the above page, we go to the *Method Summary* section and in the methods that begin with *set*, we find the method *setDriverClassName*.
So we change the *driveClass* to *driverClassName*:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" >
      <property name= "driverClassName" value="org.hsqldb.jdbcDriver" />
      <property name= "url" value
                  ="jdbc:hsqldb:file:database.dat;shutdown=true" />
      <property name = "username" value="sa"/>
      <property name="password" value ="" />
</bean>
```

The other properties were common between these two classes. so we leave them there.

Now we delete the database files and then we run the client code. It should work.

destroy-method="close"

In our xml file, for any of the beans, we can specify a *destroy-method="close"* to close the method for this bean.

We specify this for our datasource so we close the connection.
This is specifically good for databases. So it will close the database every time we run the code.

```
<!-- DataSources -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
                                destroy-method="close">
```

And when we call close on the container, Spring will call the close method for this bean.

Now we delete these four database files and we run our code two times to see the result. It should work properly.

Improvement in our code

init-method="createTables"

In BookDaoSpringJdbcImpl we cut the try catch block in the constructor and make a method *createTables()* and paste the block that we cut, under this method:

Changes in *BookDaoSpringJdbcImpl*:

```
public BookDaoSpringJdbcImpl(JdbcTemplate jdbcTemplate) {
            this.jdbcTemplate=jdbcTemplate;
      }

      private void createTables() {
            try {
                  jdbcTemplate.update(CREATE_TABLE_SQL);
            }catch(Exception e) {
                  System.err.println("Table already exists");
            }
      }
```

This method will never be called unless in the xml file in the bookDAO, we put an *init-method*

```
<!-- Dao beans -->
<bean id="bookDao" class="se.yrgo.spring.data.BookDaoSpringJdbcImpl"
                        init-method="createTables">
      <constructor-arg ref="jdbcTemplate">
      </constructor-arg>
</bean>
```

This is the complete application.xml:

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
            "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
      <!-- DataSources -->
      <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
                          destroy-method="close">
         <property name= "driverClassName" value="org.hsqldb.jdbcDriver" />
         <property name= "url" value
                   ="jdbc:hsqldb:file:database.dat;shutdown=true" />
          <property name = "username" value="sa"/>
          <property name="password" value ="" />
      </bean>

      <!-- Templates -->
      <bean id="jdbcTemplate"
                  class="org.springframework.jdbc.core.JdbcTemplate">
         <constructor-arg ref="dataSource" />
      </bean>

      <!-- Dao beans -->
      <bean id="bookDao" class="se.yrgo.spring.data.BookDaoSpringJdbcImpl"
                 init-method="createTables">
          <constructor-arg ref="jdbcTemplate">
          </constructor-arg>
      </bean>

      <bean id="bookService"
              class="se.yrgo.spring.services.BookServiceProductionImpl">
         <constructor-arg ref="bookDao">
         </constructor-arg>
      </bean>

      <bean id="accountsService"
            class="se.yrgo.spring.services.AccountsServiceMockImpl"/>

      <bean id="purchasingService" class=
            "se.yrgo.spring.services.PurchasingServiceImpl">
         <constructor-arg ref = "bookService"/>
         <constructor-arg ref = "accountsService" />
```

```
      </bean>
</beans>
```

We run the code two or more times to see if it works.