JDBC in Spring Workshop

Overview

In this part first we create a pure Jdbc implementation of the interface BookDao and then we use JdbcTemplate to create a Jdbc connection to the database.
So we can call the methods in this class to retrieve, create and delete data from the database.
We still work with the BookStore project.

Implementing JDBC in our project

- There are two files *BookDao.java* and *BookDaoJdbcImpl.java* that are needed for this part of the workshop. Download and put them in a new package: *se.yrgo.spring.data*

- If you do not have the following methods in the *Book* class you should create them: *getTitle(), getAuthor() and getPrice().*

- The database we are using is *HSQLDB*.
  Add the following to your pom file:

```
<dependency>
   <groupId>hsqldb</groupId>
   <artifactId>hsqldb</artifactId>
   <version>1.8.0.10</version>
</dependency>
```

- In the *Client* class, we keep the line with *ClassPathXmlApplicationContext* and the closing container and we remove everything else.

- We create a *BookService* object with the help of container's *getBean* method and then we create a new book, and we print out the book.

```
package se.yrgo.spring.client;

import java.util.List;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import se.yrgo.spring.domain.Book;
import se.yrgo.spring.services.BookService;
import se.yrgo.spring.services.PurchasingService;
```

```
public class Client {
    public static void main(String[] args) {
       ClassPathXmlApplicationContext container = new
                        ClassPathXmlApplicationContext("application.xml");

         BookService bookService = container.getBean(BookService.class);

         bookService.registerNewBook(new Book("1234596896812", "Birds",
                                    "Bird Lover", 100.00));
         List<Book> allBooks = bookService.getEntireCatalogue();
            for(Book book:allBooks) {
                  System.out.println(book);
            }
            container.close();
      }
}
```

- We run the code and as the result, we get three books from the mocking implementation (*BookServiceMockImpl* class) and a new book which we have created here in the main method.

- Each time we run the application, we get the same data. It is because we are using the mock implementation and it is not persistent and at the end of each run, the data is lost. So now we get rid of the mock implementation and we create a real implementation which is a production standard.

Create a class called BookServiceProductionImpl

Create a new class in the service package with the name *BookServiceProductionImpl* which implements the *BookService* interface.
In this class we implement all the methods from the interface.

This is the result of the created class:

```
package se.yrgo.spring.services;

import java.util.List;
import se.yrgo.spring.data.BookDao;
import se.yrgo.spring.domain.Book;
```

```
public class BookServiceProductionImpl implements BookService {
	private BookDao dao;
	public BookServiceProductionImpl(BookDao dao) {
		this.dao = dao;
	}

	@Override
	public List<Book> getAllBooksByAuthor(String author) {
		return dao.findBooksByAuthor(author);
	}

	@Override
	public List<Book> getAllRecommendedBooks(String userId) {
	     throw new UnsupportedOperationException();
	}

	@Override
	public Book getBookByIsbn(String isbn) {
		return dao.findByIsbn(isbn);
	}

	@Override
	public List<Book> getEntireCatalogue() {
		return dao.allBooks();
	}

	@Override
	public void registerNewBook(Book newBook) {
		dao.create(newBook);

	}
}
```

In application.xml

In our *application.xml* file, we have all the wirings for our service beans.
But now we should modify the xml file a little bit and change the implementation of *bookService* to the new implementation.

We know that the *bookService* is going to delegate to a DAO, so we should tell Spring that we want to inject the DAO object into the *bookService*. We used the constructor approach for injection with ref to *bookDao* so we can create a *bookDao* in the xml file too.

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
            "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
      <!-- Dao beans -->
      <bean id="bookDao" class="se.yrgo.spring.data.BookDaoJdbcImpl" />

      <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>
```

- Now we run the code and we get the new book that we have created in the client code. If we run again, we get the same book one more time.

- If we refresh our project, we see that there are two database files under the project. *database.dat.properties and database.data.script*.

- We can open the script, with the text editor and see the insert statement in it. We can delete the files and run the application again, they will be created again.

- Everything seems to work well but there is a little problem, JDBC is a difficult API to work with. So if we want to implement JDBC in Spring, we can use the helper class which exists in the Spring framework.

JDBCTemplate

Now we write an implementation of the Jdbc with help from Spring to reduce the lines of codes.

Add the following to your pom file:

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-jdbc</artifactId>
   <version>5.3.14</version>
</dependency>
```

Create the *BookDaoSpringJdbcImpl* class which implements the *BookDao* interface.

In this class we need an instance of this *JdbcTemplate* as a private attribute. And we inject this object through wiring. We do it through constructor injection.

```
private JdbcTemplate jdbcTemplate;
```

This class comes from this library: *org.springframework.jdbc.core*

We make the template as a parameter for the constructor:

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

With using this class, all the methods, such as create, delete, findBooksByAuthor, etc will have much less code compared to the previous jdbc implementation.

For example create method will be just one line of code:

```
public void create(Book newBook) {
     jdbcTemplate.update(INSERT_BOOK_SQL, newBook.getIsbn(),
         newBook.getTitle(), newBook.getAuthor(), newBook.getPrice());
}
```

We use the method *update* of this template to create a new book. The main use of the method *update* gets two parameters, the first is the sql statement which is an insert into the *book* table

and the second parameter is an object array, which will hold all of the arguments that we want to pass to the sql.

Instead of creating the object array, we can use a variable-length arguments list and pass the four arguments in as regular parameters.

We create final static variables for the sql statements and we use them in our methods:

- One for insert into book:

```
private static final String INSERT_BOOK_SQL = "insert into BOOK (ISBN,
TITLE, AUTHOR,PRICE) values (?, ?, ?, ?) ";
```

- One for creating table and another one to get all the books:

```
private static final String CREATE_TABLE_SQL = "create table BOOK(ISBN
VARCHAR(20), TITLE VARCHAR(50), AUTHOR VARCHAR(50), PRICE DOUBLE)";

private static final String GET_ALL_BOOKS_SQL = "select * from BOOK";
```

For the method *List<Book>allBooks()* we use the query method from the jdbcTemplate class. The first parameter is the sql query that we want to run, the second parameter is to pass in a new instance of a class that we write ourselves. This is because Spring doesn't know how to convert a resultset from a database into a book or a list of books.

This class will tell Spring how to take a resultSet from the database and convert it to a list of books.

```
public List<Book> allBooks() {
    return jdbcTemplate.query(GET_ALL_BOOKS_SQL, new BookMapper());
}
```

Since the use of BookMapper is only in this class, it can be written as an inner class in the same file. It should not be public.

```
class BookMapper implements RowMapper<Book> {
      @Override
      public Book mapRow(ResultSet rs, int rowNumber) throws SQLException {
            String isbn = rs.getString("ISBN");
            String title = rs.getString("title");
            String author = rs.getString("author");
            double price = rs.getDouble("price");
            Book book = new Book(isbn, title, author, price);
            return book;
```

```
        }
}
```

This class implements *RowMapper* in the org.springframework.jdbc.core *package.*

findBookByAuthor method

And for the implementation for the *findBookByAuthor*, we use a query method and as parameters, we will give the sql statement, the *BookMapper* object and the author for the replacement for '?'.

```
public List<Book> findBooksByAuthor(String author) {
      return jdbcTemplate.query("select * from Book where author=?", new
                                              BookMapper(), author);
}
```

findByIsbn method

The same way, we write the findByIsbn method but this time, we use queryForObject method which returns only one result back:

```
public Book findByIsbn(String isbn) {
      return jdbcTemplate.queryForObject("select * from Book where ISBN=?",
                              new BookMapper(), isbn);
}
```

delete method

And for the *delete* method, we use the *update* method. All the inserts, updates, deletes and creating and dropping tables are done through the *update* method.

```
public void delete(Book redundantBook) {
   jdbcTemplate.update("Delete from Book where ISBN=?",
                                    redundantBook.getIsbn());
}
```

Creating the table

We need to create the table, if it doesn't already exist. We do it in the constructor. We call the update method from the jdbcTemplate. This code should be written inside a try catch block.

```
public BookDaoSpringJdbcImpl(JdbcTemplate jdbcTemplate) {
      this.jdbcTemplate=jdbcTemplate;
      try {
           jdbcTemplate.update(CREATE_TABLE_SQL);
      }catch(Exception e) {
            System.err.println("Table already exists");
      }
}
```

This is the complete *BookDaoSpringJdbcImpl* class:

```
package se.yrgo.spring.data;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import se.yrgo.spring.domain.Book;

import se.yrgo.spring.services.BookService;

public class BookDaoSpringJdbcImpl implements BookDao {

      private static final String INSERT_BOOK_SQL = "insert into BOOK
(ISBN, TITLE, AUTHOR,PRICE) values (?, ?, ?, ?) ";
      private static final String CREATE_TABLE_SQL = "create table
BOOK(ISBN VARCHAR(20), TITLE VARCHAR(50), AUTHOR VARCHAR(50), PRICE
DOUBLE)";
      private static final String GET_ALL_BOOKS_SQL = "select * from BOOK";

       private JdbcTemplate jdbcTemplate;
```

```
	public BookDaoSpringJdbcImpl(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate=jdbcTemplate;
		try {
		    jdbcTemplate.update(CREATE_TABLE_SQL);
		}catch(Exception e) {
			System.err.println("Table already exists");
		}
	}
	@Override
	public List<Book> allBooks() {
		return jdbcTemplate.query(GET_ALL_BOOKS_SQL, new BookMapper());
	}

	@Override
	public Book findByIsbn(String isbn) {
		return jdbcTemplate.queryForObject("select * from Book where
					ISBN=?", new BookMapper(), isbn);
	}

	@Override
	public void create(Book newBook) {
		jdbcTemplate.update(INSERT_BOOK_SQL, newBook.getIsbn(),
		  newBook.getTitle(), newBook.getAuthor(), newBook.getPrice());
	}

	@Override
	public void delete(Book redundantBook) {
		jdbcTemplate.update("Delete from Book where ISBN=?" ,
			redundantBook.getIsbn());
	}

	@Override
	public List<Book> findBooksByAuthor(String author) {
		return jdbcTemplate.query("select * from Book where author=?",
				new BookMapper(), author);
	}
}

class BookMapper implements RowMapper<Book> {
   @Override
   public Book mapRow(ResultSet rs, int rowNumber) throws SQLException {
```

```
        String isbn = rs.getString("ISBN");
        String title = rs.getString("title");
        String author = rs.getString("author");
        double price = rs.getDouble("price");
        Book book = new Book(isbn, title, author, price);
        return book;
    }
}
```

In order to run the above code, we need to manage the connections.
We need to tell the Spring how to manage the database connection. We will do it in the next workshop.