Wiring in Spring-workshop

A more sophisticated example of project to use containers and configuration file

In this chapter, we want to use dependency injection and the spring container.
We will have a little more realistic project and we will write a mock (prototype or fake) implementation of a service.

This is just a start. Later we will write a production standard implementation of the service.

We will work with:

- An interface
- the implementation of the interface
- application.xml file
- the client code

This is the project that we want to use for this workshop: *BookStore*.
Download the BookStore project and add it to your IDE.

Component BookService

The main component in this project will be a *BookService* component, a book management class.

So we will have an interface with the name *BookService*. The methods in this interface will be:

- A method to query all the books by a specific author.
- A method to get all the recommended books for a specific user.
- A method to search for a book by its ISBN (primary key)
- A method to get the entire catalog.
- A method to register new books.

Then we want to have a class that implements this interface.

We will configure the *BookService* using Spring and we will look up the service by using the container. So our client whether a web tier or a standalone client will be able to call the service.

The design that we want here is not to involve the business logic in our implementation class but rather the BookService to call the domain classes *Author, Catalogue, Book,* etc.

And then the domain classes can implement the business logic.

The reason for this design is that we are working with an object orientation language, and the behavior should spread across different objects and it makes it easier to unit test.

BookStore project

Look at the *Book class and BookService* interface and see what properties and methods are in these classes.

Implement the interface BookService (A mock implementation)

In the beginning maybe we do not know how to do something sophisticated for example calling the database, or maybe we just want to prototype the front end of the application and we have not built the backend yet.

So It is a good reason to use the theory that we learned from the Spring: We can begin with mocking the service and then later make another implementation for the interface that is more realistic and has all the connection to the database etc.

So for now we have a mock implementation with the name: *BookServiceMockImpl*
This implementation will have all the methods in the interface but methods can be hard coded just for the reason that everything works. Like simulating the backend to see if the client works. And when we have written our production implementation, we simply change the implementation class in the xml file.

We already have the class *BookServiceMockImpl* in our project.

We edit the class a little bit by adding some books in the constructor and implementing the methods.
This is the result after editing:

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

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import se.yrgo.spring.domain.Book;

public class BookServiceMockImpl implements BookService {
```

```
    private Map<String, Book> testBooks = new HashMap<String, Book>();

    public BookServiceMockImpl() {
        Book book1 = new Book("ISBN1", "Forecasting Rain", "Phil Don",
10.99);
        Book book2 = new Book("ISBN2", "The Tulip", "Anna Pavord", 14.99);
        Book book3 = new Book("ISBN3", "Enough", "Bill McKibben", 16.99);
        testBooks.put("ISBN1", book1);
        testBooks.put("ISBN2", book2);
        testBooks.put("ISBN3", book3);
    }

      public Book getBookByIsbn(String isbn) {
            return testBooks.get(isbn);
      }

      public List<Book> getEntireCatalogue() {
            return new ArrayList<Book>(testBooks.values());
      }

      public void registerNewBook(Book newBook) {
            testBooks.put(newBook.getIsbn(), newBook);
      }

      public List<Book> getAllBooksByAuthor(String author) {
            return null;
      }

      public List<Book> getAllRecommendedBooks(String userId) {
            return null;
      }
}
```

In this class we have a Map collection of books. Map is a good choice to simulate a database because we can do things like finding the object by the key.

Add the bean in the application.xml

In the *application.xml*, we add the following:

```
<beans>
      <bean id="bookService"
class="se.yrgo.spring.services.BookServiceMockImpl">
      </bean>
</beans>
```

Write the client class

In the client class, we do exactly the same thing that we have done before:

- we open the spring container,
- we get the required object and
- start to call its methods.

This is the whole client class:

```
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;

public class Client {
    public static void main(String[] args) {
      ClassPathXmlApplicationContext container = new
         ClassPathXmlApplicationContext("application.xml");
      BookService service = container.getBean(BookService.class);
      List<Book>allBooks = service.getEntireCatalogue();
      for(Book book:allBooks) {
           System.out.println(book);
      }
    }
}
```

Run the code. We should get three books as a result.

Later we will write a production standard implementation of the service.