Tutorial

Dependency Injection, an Introduction to Spring

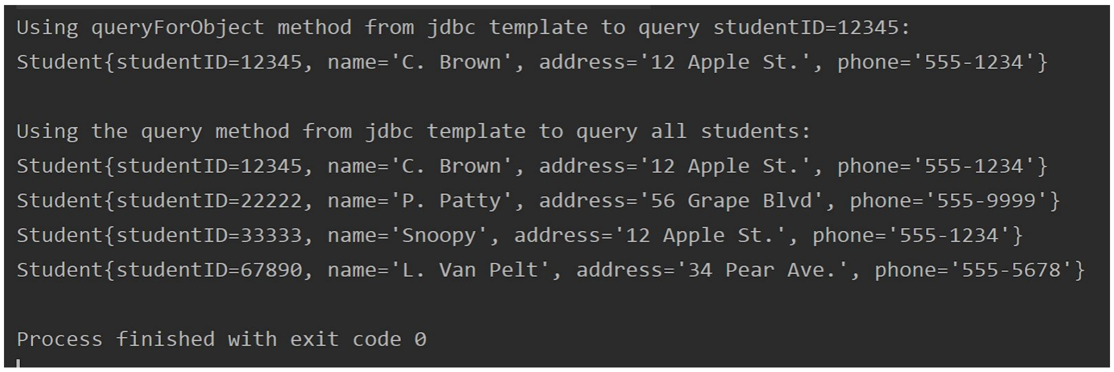
Objects often create instances of their own dependencies in traditional object-oriented programming. This can create brittle, inflexible code that is difficult to change and difficult to test. Dependency injection allows loose coupling between objects, which makes your code more flexible and more testable. Spring is a framework for Java development that uses dependency injection, allowing dependencies to be managed by the framework.

The project you have imported for the pre-class assignment is a very simple server project that accesses a local database. As you look through the source files, you will notice that there are three packages: model, dao, and service. Notice under the dao and service packages, there are interfaces and classes implementing the interface. You will see that using the Spring Framework can help make the code more maintainable and flexible. For example, in the future, you may be asked to use a different database than SQLite, or store your database in the cloud. The Spring Framework and interfaces help to layer your code. By doing this through Spring, changing how the project accesses the database is as easy as implementing different DAO classes or wiring your dataSource bean differently. Doing so still allows you to do testing with your original database and deploy your project using the new database.

The purpose of this lab is to teach you how to implement dependency injection using the Spring Framework by configuring beans via XML, annotations, and auto-wiring. You also will learn how Spring can simplify database management and learn how to use Mockito with Spring.

Part 1: Using Constructor Injection

1. There are TODO comments placed in this project. To receive full credit, you must delete these comments when finished. Run the project after following the steps for the pre-class assignment to make sure it works. You should see this as the output:



```
Using queryForObject method from jdbc template to query studentID=12345:
Student{studentID=12345, name='C. Brown', address='12 Apple St.', phone='555-1234'}

Using the query method from jdbc template to query all students:
Student{studentID=12345, name='C. Brown', address='12 Apple St.', phone='555-1234'}
Student{studentID=22222, name='P. Patty', address='56 Grape Blvd', phone='555-9999'}
Student{studentID=33333, name='Snoopy', address='12 Apple St.', phone='555-1234'}
Student{studentID=67890, name='L. Van Pelt', address='34 Pear Ave.', phone='555-5678'}

Process finished with exit code 0
```

In Main.java, note the two lines of code that call jdbcTemplate.queryForObject() and jdbcTemplate.query(). Notice that a new instance of BeanPropertyRowMapper<> is created. It automatically maps the individual fields of a row in a database to the setters in a Model object. If the names and data types of the setters in the model object match the fields in your database, BeanPropertyRowMapper will automatically store the information correctly in the object. Custom row mapping with a lot more power and efficiency could be created but is beyond the scope of this lab.

2. Now under src/main/resources, there is an XML file named applicationContext.xml. This is where you will be injecting and wiring beans. Open up the XML file and note the two beans already created to allow JdbcTemplate to work. Delete the comments after you have read and understood what is going on in the XML file.
3. Create a new class named "StudentDAOImpl" in the dao package. Make it implement the StudentDAO interface and the two methods of the interface. Create a private variable *"private JdbcTemplate jdbcTemplate"* and create a constructor that will set this variable. Use the ClassDAOImpl class as an example of what to do.
4. Implement the two methods in StudentDAOImpl by using the jdbcTemplate.queryForObject(..) and jdbcTemplate.query(..) lines of code in Main.java into the relevant method.

   Note: To implement *getStudentById()* with *queryForObject*, you will get an exception when you query for something that does not exist in the database. To handle this properly, surround the method call *jdbcTemplate.queryForObject(..)* with a try/catch block. If an *EmptyResultDataAccessException* was caught, your getStudentById() should simply return null.
5. Now we can try a dependency injection using the constructor you created in StudentDAOImpl. Open the applicationContext.xml file and add the following bean definition under the jdbcTemplate bean definition:

```
<bean id="studentDAO" class ="dao.StudentDAOImpl">
    <constructor-arg ref="jdbcTemplate" />
 </bean>
```

   The bean identifier (id) is one way that we can use to reference this instantiated class. The argument "class=" is asking for what class is to be instantiated. We added a <constructor-arg> which can inject another bean or value into the instantiated class. In this case, we added a reference to a bean that was already defined in the XML file, "jdbcTemplate".
6. Now in the class Main, change the second line of code in the main method to instantiate a StudentDAO variable instead:

```
StudentDAO studentDAO = (StudentDAO) context.getBean("studentDAO");
```

7. Replace the lines using the variable jdbcTemplate to query from the database to use the functions you implemented in StudentDAOImpl instead. Run the project. You should get the same result as before.

8. Wire up the ClassDAOImpl in the XML file just like you have done for StudentDAO in step 4, naming the bean “classDAO”.
9. Now take a look at the service classes in the service.report package. There are two classes that implement ReportService. Use constructor injection to initialize the GradesReportService class, with *id="reportService".* Refer to step 4 or section 2.4.3 in the reading to learn how to do so. (**Hint:** You will have to reference the “classDAO” bean you just instantiated in step 7).
10. Add this code after “Implementation of ReportService” in the main method:

```
ReportService reportService = (ReportService) context.getBean("reportService");
reportService.printReport(System.out);
```

11. You should see a report of all the students’ grades when you run the project. **Take a screenshot of the whole output. Name the screenshot SpringPart1.**

Part 2: Multiple Implementations and Setter Injection

Take a look at the printReport() method in ScheduleReportService and note the differences with GradesReportService. In this part of the tutorial, we will initialize ScheduleReportService instead of GradesReportService.

1. As mentioned in the reading, you can instantiate a List of objects in your XML configuration. Now go to where you have instantiated the reportService bean in applicationContext.xml and replace **“service.report.GradesReportService**” with **“service.report.ScheduleReportService”**. When you run the project, you will see that the ScheduleReportService was called instead of GradesReportService with this output:

    ScheduleReportService invoked.
    No new classes were injected. Use setter injection to inject new classes for the semester.

2. ScheduleReportService was unable to run as intended because we have not done setter injection to initiate the private member “List<Class> classes” yet. To do so, we need to add a setter injection into the bean definition. As discussed in the reading, you can inject List objects into beans as well. Paste this inside the reportService bean in your configuration file:

```
<property name="classes">
        <list>
           <ref bean="CS120"/>
           <ref bean="CS205"/>
        </list>
</property>
```

3. Since the list we are injecting are Class objects, let's instantiate these Class objects in applicationContext.xml:

```
<bean id="CS120" class = "model.Class">
    <!--use constructor injection here-->
</bean>

<bean id="CS205" class = "model.Class">
    <!--use setter injection here-->
</bean>
```

   To get full credit, you must use constructor injection to instantiate the CS120 bean and setter injection to instantiate the CS205 bean. You need to instantiate these variables:

```
int classID;
String className;
String room;
String professor;
```

   Remember for constructor injection, if you don't specify the name of the variable you are instantiating, the order of instantiating each variable matters. Also, other than className, you can use whatever value you want to instantiate these variables. Make sure that the className variable matches the name of the bean id. For example, to instantiate the className variable for the CS205 bean, insert this statement into the CS205 bean: <property name="className" value="CS205"/>
4. After you instantiated the beans, run the project and you should see that the students who have passed CS101 (C. Brown, L. Van Pelt, and Snoopy) were suggested to take CS120 and CS205.
5. **Take a screenshot of your output after "Implementation of ReportService". Name the screenshot SpringPart2.**

Part 3: Auto-Wiring

There are many different ways in which Spring can be configured. We will start by seeing how a purely XML configuration can do auto-wiring.

1. Since it takes an extra <constructor-arg/> or <property/> definition to define and wire up the beans in the XML file, the Spring framework allows a shortcut called auto-wiring. Let's try it with the second bean in our applicationContext.xml, the *jdbcTemplate* bean. Replace the whole bean definition with this line:

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate" autowire="byType"/>

Try running the project now. As you can see, the project still ran as expected. This type of auto-wiring is for setter injection, and Spring checks to see if an instantiated bean matches the type which the setter is looking for. You can also auto-wire using "byName" if the name of your setter matches the name of the bean to be injected. Go ahead and change "byType" into "byName", then run the project again.

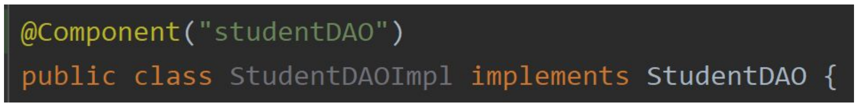
2. We can do the same thing for *studentDAO* and *classDAO*, except that since both use constructor injection, we need to use *autowire="constructor"* instead. Change the bean definition for studentDAO and classDAO the same way as we have done for the jdbcTemplate bean in step 2, but using *autowire="constructor"* instead. Run the project to make sure it works.
3. Now add another *autowire="constructor"* with the reportService bean and delete the constructor-arg reference to classDAO. In this case, you cannot delete the <property/> tag yet since it is needed to run ScheduleReportService.
4. Run the program and make sure you get the same output as before.

Part 4: Annotation-Based Configuration with Auto-Wiring

1. It may seem convenient already that in one line the Spring framework can wire a bean to be used in the project. However, through something called component scanning, wiring a bean becomes even easier. To enable this functionality, add the following line into your applicationContext.xml **before all the bean definitions**:

   <context:component-scan base-package="dao,service" />

   This tells the Spring Framework to scan the dao and service packages for components that it needs to instantiate the beans for. You will see how this reduces the code in theXML file even further.
2. Delete the *studentDAO* bean definition in the applicationContext.xml and then go to the StudentDAOImpl class. Right before the declaration of the class, add this annotation as follows:

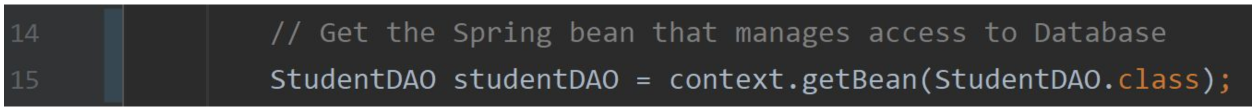


```
@Component("studentDAO")
public class StudentDAOImpl implements StudentDAO {
```

   Adding this annotation tells the Spring Framework to create a bean from this class (if component scanning is enabled in the configuration file) and name the instantiated bean studentDAO. Now run your project again. It should run normally. **Do the same thing for the classDAO bean.**
3. If you are planning to only have one implementation of the interface at a time, you would not have to include the bean identifier in the @Component annotation. Since that is what we plan to do with the DAOs, go ahead and remove ("studentDAO") and ("classDAO")

after the @Component annotation. When you run your program, it will throw an error saying that the bean “studentDAO” was not found. This is because Spring will name your bean as the class name with the first letter in lowercase by default. In this case, the bean is named “studentDAOImpl”.

4. To fix this, navigate to the main method and go to where context.getbean() is called. To make sure Spring finds the correct bean, go ahead and change getBean(“studentDAO”) to getBean(“studentDAOImpl”). The program should be running normally. To avoid casting and the problem of possible different bean names, we can also call getBean() by class, allowing the removal of the casting we created:



```
        // Get the Spring bean that manages access to Database
        StudentDAO studentDAO = context.getBean(StudentDAO.class);
```

   Replace the line instantiating StudentDAO with the code shown above. Then run your project to make sure it works.

5. Now we can remove the bean definition for reportService as well. To do so, we need to move the declaration of the the <list /> bean that is currently inside the reportService bean definition using <util:list>. Delete the reportService bean definition then paste this line of code in the XML file before the class beans:

```
<util:list id="classes">
        <ref bean="CS120"/>
        <ref bean="CS205"/>
</util:list>
```

   We just used the util namespace to declare a list of classes as a bean in XML.

6. Now we can use annotations to wire this up with ScheduleReportService. Add the @Component("reportService") annotation in ScheduleReportService. Try running the program now. You will get a message saying that there were no classes injected. This is because Spring by default did not use setter injection.

7. To fix this, we need to add the @Autowired annotation in the ScheduleReportService class so the Spring Framework will know what else needs to be wired. Place the annotation right in front of the setClasses method in ScheduleReportService. This will cause the list of classes declared in your configuration file to be injected into the ScheduleReportService bean. Your project should run now just like before.

Part 5: Using Mockito in Spring to Unit Test

Spring also allows you to wire Mockito mocks as objects to test. For example, if we did not have a complete implementation of classDAO but we wanted to make sure our ScheduleReportService class was working properly, we could mock up the classDAO object to act like it was working properly. Before starting this part of the tutorial, make sure that Mockito is a dependency in your project and that you have the test files in your project. To review how to use Mockito, review the Mocking with Mockito pre-class assignment and tutorial from CS 203.

1. In the "src/test" folder, there is a "java" folder and a "resources" folder. Inside the "resources" folder is the XML file which we will use to configure the JUnit test in the java folder. We will need to annotate our test to ensure that the configuration we made for the tests is the one the tests use when executed. To do so, add these two annotations on top of the declaration of "public class ReportServiceTest" and have IntelliJ import the relevant packages:

   @ContextConfiguration("/ReportServiceTestContext.xml")

   This annotation tells Spring where to find the configuration file for the test class.

   @ExtendWith(SpringExtension.class)

   This annotation tells JUnit to use Spring in our test class.

2. Now open the *ReportServiceTestContext.xml* file under "src/test/resources". There are two bean definitions you need to make. To allow Mockito to mock the ClassDAO object, we can insert the following bean definition:

   ```
    <bean id="classDAO" class="org.mockito.Mockito" factory-method="mock">
           <constructor-arg value="dao.ClassDAO"/>
   </bean>
   ```

   This is the equivalent of declaring the following in your Test class:

   ```
   ClassDAO classDAOTest = Mockito.mock(ClassDAO.class);
   ```

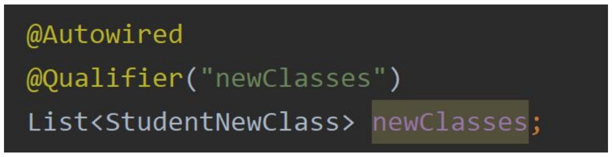
3. Now we can choose to explicitly define the ScheduleReportService bean class, or use component scanning to auto-wire the bean. Since we already set up component scanning earlier to run the ScheduleReportService, we simply need to add the following to our ReportServiceTestContext.xml file:

   ```
   <context:component-scan  base-package="service" />
   ```

   Note that unlike before, we did not add the dao package in our component scan. This is because if Spring was configured to scan the dao package, Spring would want to inject a jdbcTemplate into the classDAO even though we want that class to be mocked instead.
4. Now, look at the rest of the bean definitions in the XML file. These are the lists that we will need to use to test to see if our ScheduleReportService functions properly. You will need to reference these bean definitions to properly wire them for our test.
5. There are five tests in our test class. The first four tests just check if you have wired everything correctly. To do so, you will need to add the @Autowired annotation and the @Qualifier annotation to the relevant instance variables. To pass the first test correctInitialization(), you will need to add the @Autowired annotation in two places: in

front of the ClassDAO instance variable and in front of the ReportService instance variable. After you have done so, try running the first test only.

6. The next three tests will check if your classes lists are wired correctly. Those lists were declared as instance variables after the ReportService instance variable. Now add the appropriate annotations to wire the lists. To do so, you will need to add the @Qualifier annotation. Since these three lists are very similar, Spring needs you to specify which list bean you want to wire each instance variable to. You can add the bean id of the list you want to wire in the @Qualifier annotation. For example, if I want to wire up the *List<StudentNewClass> newClasses;* instance variable, I need to annotate it with @Autowired and @Qualifier("newClasses") as follows:

   ```
   @Autowired
   @Qualifier("newClasses")
   List<StudentNewClass> newClasses;
   ```

   Try that and you should pass the next test, *correctNewClassesList()*. Do the same thing for the next two lists.
7. After the first four tests are passing, you are ready to use Mockito to mock ClassDAO in the last test, *scheduleReportServicePrint()*. Go to the printReport() method of ScheduleReportService and see where the ClassDAO methods are called. Mock the ClassDAO appropriately using Mockito.when() to make sure the right lists are returned. You should have three Mockito.when() statements mocking these methods:
   a. One Mockito.when() statement to mock classDAO.getNewClasses(), which should return the *newClasses* bean.
   b. Two Mockito.when() statements to mock classDAO.queryClassesByName(). This call is made in a private method, *ScheduleReportService.printNewClass(..)*. *classList1* should be returned when the value queried is *"EE300"* and *classList2* should be returned when *"PH201"* is queried.
8. **After you have correctly mocked the ClassDAO function, all the tests should pass. Take the screenshot of all the passed tests with the output starting from "ScheduleReportService invoked." (You may need to scroll down in the output window to see it). Name the screenshot SpringPart5.**

Submission

Submit the following on Canvas:

1. Main.java
2. ClassDAOImpl.java
3. StudentDAOImpl.java
4. ScheduleReportService.java
5. ReportServiceTest.java
6. applicationContext.xml
7. ReportServiceTestContext.xml

8. SpringPart1 Screenshot
9. SpringPart2 Screenshot
10. SpringPart5 Screenshot

Preparation for the Next Assignment

Do not delete your Spring project since the next assignment will build upon it. In the next assignment, you will put Hibernate, AOP, and Spring together, being exposed to how these frameworks work hand in hand for larger scale projects.