Guided Lab

More Spring Topics: AOP Using Spring; Spring Integration w/Hibernate; Spring Java Configuration

The Spring Framework includes many tools that help with Java development. In this lab, we will touch on how to use both AOP and Hibernate with Spring. There will also be a mini lab where you will experiment with configuring Spring using Java configuration. Since AOP and Hibernate were already covered in the last couple of labs, this will be less of a step-by-step tutorial and more of a guided lab.

Before Starting the Lab

Remove Implementation of ReportService

This lab will focus on the basics of implementing Spring AOP and Hibernate. Therefore, you will not need to worry about ClassDAO and the services that use its methods. In your Main.java, **comment out the following lines of code at the end of the main method**:

```
System.out.println("\n-----Implementation of ReportService----- ");
ReportService reportService = (ReportService) context.getBean("reportService");
reportService.printReport(System.out);
```

Also, remove the service package from the component scan in your applicationContext.xml. Change this line of code:

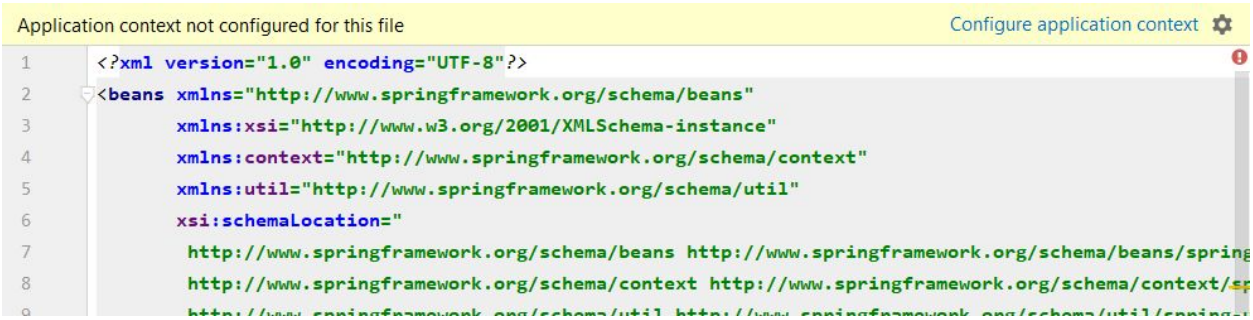
```
<context:component-scan  base-package="dao,service" />
```

To the following:

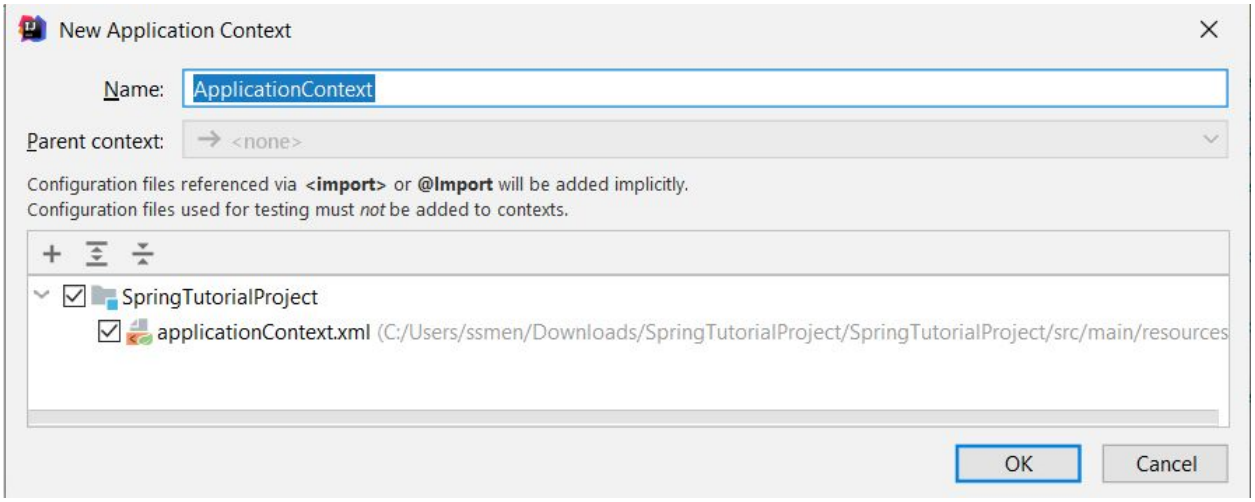
```
<context:component-scan  base-package="dao" />
```

Using IntelliJ Ultimate

You will use the same project for this lab that you used for the first Spring tutorial. Make sure you open your project with Intellij **Ultimate**. When opening your project using IntelliJ Ultimate, you may get the following banner when you open your applicationContext.xml file:



It is not necessary to fix this banner to run your program, but configuring your applicationContext.xml file for IntelliJ Ultimate can help with autofill. To clear this banner, click on *Configure application context*, then click on *Create new application context*. You will see this pop-up window:



Make sure you have the same configuration as seen above. Click OK to finish configuring your application context, which will dismiss the banner.

AOP Using Spring

Spring's implementation of AOP is different from AspectJ's. While AspectJ weaves the Aspects during compilation (also known as static AOP), Spring uses dynamic proxies to implement AOP (also known as dynamic AOP). Therefore, Spring AOP can only do method interception pointcuts and if more advanced AOP techniques are required, Spring can integrate with AspectJ. Everything done in the previous AOP Tutorial could be implemented by Spring AOP.

Configuring AOP using XML

Since Spring AOP uses proxies that wrap around an object to do aspects, the original way to configure AOP is by writing proxy classes. Spring 2.0 introduces schema-based configuration

using XML and annotation-based configuration using AspectJ annotations. In this section, we will do a step-by-step walkthrough on how to use XML to set up a logging aspect that will be logging our methods in the dao package.

1. Download the LoggerAspect.java file and put it in a new package named “aspect”. Although we will use this class as an aspect, you will see that it is just a simple Java class that can be instantiated and used. This file has a java.util.logging.Logger that gets initialized to store all log messages into log.txt. We will now modify our applicationContext.xml file to configure this Java class as an aspect.

2. Go to your applicationContext.xml file. We will need to add the aop namespace in the XML beans header. Add the following line right before *xsi:schemaLocation*:

   xmlns:aop="http://www.springframework.org/schema/aop"

3. Now add the following lines inside *xsi:schemaLocation=“”*:

   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop.xsd

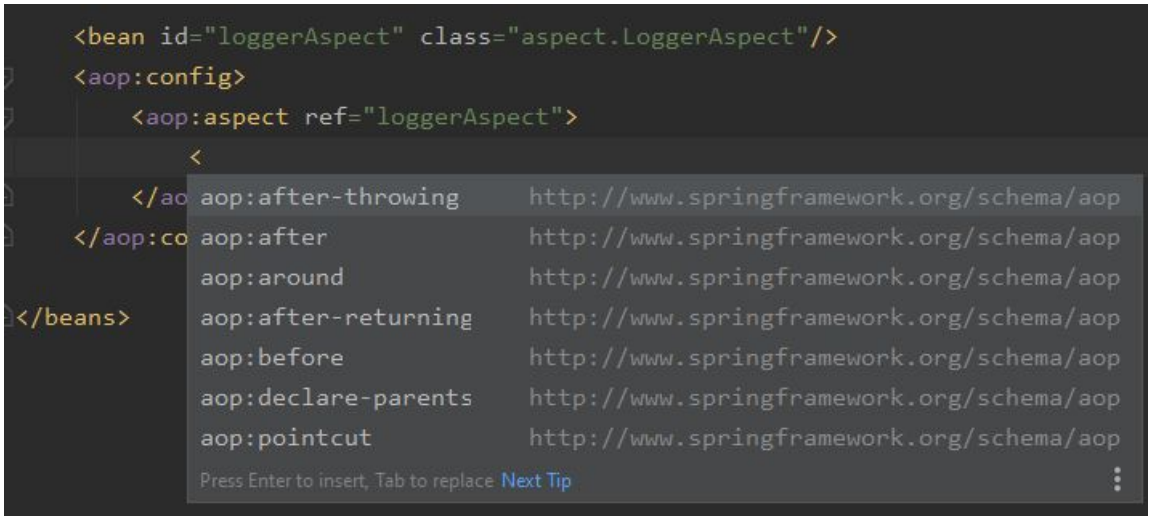
4. Next we need to instantiate our LoggerAspect class as a bean. Add the following line before the </beans> at the end of the file:

   <bean id="loggerAspect" class="aspect.LoggerAspect"/>

5. Now we can add AOP configuration to the applicationContext.xml file. Type *<aop:config>* a line above the </beans> at the end of the applicationContext.xml file. As you type, IntelliJ should suggest relevant inputs and automatically close your tag. We can add our LoggerAspect class as an aspect in these tags.

6. Type *<aop:aspect* into IntelliJ between *<aop:config> </aop:config>* . Add a space after *aspect* and you should see suggestions for parameters you could input like *id* and *ref*. Since we will not need to identify this aspect in other places, we only need to add the *ref* tag, which asks for the bean of the aspect we are going to configure. Type *ref="loggerAspect"*, then close the brackets by typing ‘>’. By doing so, IntelliJ should automatically close your tag with *</aop:aspect>*.

7. All the annotations you would normally do to configure your aspect via AspectJ can be configured for Spring AOP inside the *<aop:aspect>* tag. When you type *<aop* in between the *<aop:aspect>* tags, you should see tag names that correspond to AspectJ annotations you used in the AOP tutorial like the following:



To review what these tags do, go to the [previous AOP assignment] to understand these AOP concepts.

8. Type *<aop:pointcut* to add a pointcut, and add an *id="daoMethod"*. We can use this identifier to reference the pointcut. Right after the *id* parameter, type *expression="execution(* dao.*.*(..))"* . Notice that this pointcut is for all methods in the dao package. Close this tag by adding "/>" at the end of the line.

9. In the LoggerAspect class, there is a method called logReturn(..). We want this method to execute as an aspect after our DAO methods return. Try to configure it yourself by using the suggestions from IntelliJ. You will have to reference the method name, the variable name of the returning value, and the reference to the pointcut definition. Go to the next step if you are stuck.

10. Run your project to see if you were successful. You should have seen the following in your console (The ordering between the output and logging may vary):

```
"C:\Program Files\Java\jdk-13.0.2\bin\java.exe" ...
Using queryForObject method from JdbcTemplate to query studentID=12345:
Apr 02, 2020 11:46:12 AM aspect.LoggerAspect logReturn
INFO: value(s) returned from DAO was: Student{studentID=12345, name='C. Brown', address='12 Apple St.', phone='555-1234'}
Apr 02, 2020 11:46:12 AM aspect.LoggerAspect logReturn
INFO: value(s) returned from DAO was: [Student{studentID=12345, name='C. Brown', address='12 Apple St.', phone='555-1234'
Student{studentID=12345, name='C. Brown', address='12 Apple St.', phone='555-1234'}

Using the query method from JdbcTemplate 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'}
```

If you did not get the right output, make sure this is what you have in your applicationContext.xml header:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:util="http://www.springframework.org/schema/util"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/bean
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-conte
        http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd
        http://www.springframework.org/schema/aop https://www.springframework.org/schema/aop/spring-aop.xsd">
```

Also check to see if this is what you have to configure AOP:

```
    <bean id="loggerAspect" class="aspect.LoggerAspect"/>
    <aop:config>
        <aop:aspect ref="loggerAspect">
            <aop:pointcut id="daoMethod" expression="execution(* dao.*.*(..))"/>
            <aop:after-returning method="logReturn" returning="returnValue" pointcut-ref="daoMethod"/>
        </aop:aspect>
    </aop:config>
</beans>
```

Now we will configure an around advice, which is the most powerful advice in Spring AOP. You can check for exceptions, deal with return values, and do any logic before executing the method. To review what an around advice is and how to configure one, review the previous AOP assignment. The advice method is in the LoggerAspect class, named logDAO(). Notice that in the logDAO() method, the logger is called many times before and after *joinPoint.proceed().* Without AOP, these calls to the logger would be scattered throughout our dao classes.

1. Go back to the applicationContext.xml and delete the *<aop:after-returning .. />* line since it will be redundant after we implement the logDAO() method.

2. Replace the deleted line with the following:

   <aop:around method="logDAO" pointcut-ref="daoMethod" />

3. Run the program and you should see logging in the console for entering a DAO method, the time it takes for the DAO to execute, and what was being returned:

```
Apr 02, 2020 12:09:06 PM aspect.LoggerAspect logDAO
INFO: The target class is class dao.StudentDAOImpl and the method entered is getStudentById(int))
Apr 02, 2020 12:09:07 PM aspect.LoggerAspect logDAO
INFO: value(s) returned was: Student{studentID=12345, name='C. Brown', address='12 Apple St.', phone='555-1234'}
Apr 02, 2020 12:09:07 PM aspect.LoggerAspect logDAO
INFO: DAO execution time: 859 milliseconds
Apr 02, 2020 12:09:07 PM aspect.LoggerAspect logDAO
INFO: The target class is class dao.StudentDAOImpl and the method entered is getAllStudents())
Apr 02, 2020 12:09:07 PM aspect.LoggerAspect logDAO
INFO: value(s) returned was: [Student{studentID=12345, name='C. Brown', address='12 Apple St.', phone='555-1234'}, Stud
Apr 02, 2020 12:09:07 PM aspect.LoggerAspect logDAO
INFO: DAO execution time: 4 milliseconds
```

4. Now, let's test what happens when an exception occurs. In the Main.java, remove the number 5 from the following line:

   Student student = studentDAO.getStudentById(12345);

5. Now run your project. In the beginning of the console output, there should be an INFO level message that states the following:

   An exception was thrown during DataAccess: java.lang.NullPointerException

6. Since there are no students with the studentID 1234, nothing was returned, but there was an exception made. **Take a screenshot of the log message in the beginning of the console with your applicationContext.xml file opened and scrolled to where you have added the AOP configuration.**

Using AspectJ Annotations

Spring also can use AspectJ annotations to do AOP configuration, in the same way you have configured AOP in the previous AOP assignment. Refer to the previous AOP assignment if you need a refresher on using the annotations to configure aspects.

1. Delete the *<aop:config>* tag and all of it's contents, and replace it with:

   <aop:aspectj-autoproxy />

   This tells Spring to use the AspectJ annotations you have to configure an aspect.

2. Also remove the line `<bean id="loggerAspect" class="aspect.LoggerAspect"/>`. We can instead add in the <context:component-scan .. /> the aspect package as well:
   `<context:component-scan base-package="dao,aspect" />`
   Now, going back to the LoggerAspect.java file, add the annotation *@Component* to let Spring create the LoggerAspect bean by itself.

3. Annotate the LoggerAspect just like you would with AspectJ to configure logDAO() as an advice. You will need the @Aspect, @Component, @Pointcut, and @Around annotations. Refer to the previous AOP assignment as a refresher. You will need to add a pointcut to the LoggerAspect class. Refer to the AOP assignment to remember how to do this. Your pointcut should target the execution of all methods of all classes in the 'dao' package (as the one we deleted above did).

4. Add the number 5 back into the following line in Main.java:

   Student student = studentDAO.getStudentById(1234);

5. Run your project. It should run as before, showing something similar to the following as output (Note that the logs could be printed before or after the output):

```
"C:\Program Files\Java\jdk-13.0.2\bin\java.exe" ...
Apr 02, 2020 3:29:41 PM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate ORM core version 5.4.12.Final
Apr 02, 2020 3:29:42 PM org.hibernate.annotations.common.reflection.java.JavaReflectionManager <clinit>
INFO: HCANN000001: Hibernate Commons Annotations {5.1.0.Final}
Apr 02, 2020 3:29:44 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL8Dialect
Apr 02, 2020 3:29:45 PM org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformInitiator initiateService
INFO: HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
Using queryForObject method from JdbcTemplate to query studentID=12345:
Apr 02, 2020 3:29:46 PM aspect.LoggerAspect logDAO
INFO: The target class is class dao.StudentDAOHibernate and the method entered is getStudentById(int))
Apr 02, 2020 3:29:46 PM aspect.LoggerAspect logDAO
INFO: value(s) returned was: Student{studentID=12345, name='C. Brown', address='12 Apple St.', phone='555-1234'}
Apr 02, 2020 3:29:46 PM aspect.LoggerAspect logDAO
INFO: DAO execution time: 81 milliseconds
Student{studentID=12345, name='C. Brown', address='12 Apple St.', phone='555-1234'}

Using the query method from JdbcTemplate to query all students:
Apr 02, 2020 3:29:46 PM aspect.LoggerAspect logDAO
INFO: The target class is class dao.StudentDAOHibernate and the method entered is getAllStudents())
Apr 02, 2020 3:29:47 PM aspect.LoggerAspect logDAO
INFO: value(s) returned was: [Student{studentID=12345, name='C. Brown', address='12 Apple St.', phone='555-1234'}, Student{studentID=222
Apr 02, 2020 3:29:47 PM aspect.LoggerAspect logDAO
INFO: DAO execution time: 225 milliseconds
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
```

Spring Integration with Hibernate

Integrating Hibernate using the Spring Framework brings forth many benefits. This lab will focus on two benefits:

1. Ease of integrating Hibernate by configuring applicationContext.xml
2. Reduce boilerplate code using Spring Transactions (aka Spring AOP)

Configuring Hibernate for *model.Student*

In the Hibernate tutorial, you have used a hibernate.cfg.xml file to configure Hibernate for a project. In Spring, that file is no longer required since you can put the configuration in the applicationContext.xml. Before moving forward, make sure that a MySQL database is configured as described in the pre-class assignment.

1. Let's annotate our model.Student class using the annotations learned in the Hibernate assignment first. Open the *model.Student* class in your project, and add the @Entity and @Table annotations as follows:

```
@Entity
@Table(name = "students")
public class Student {
```

   Note: Make sure when typing the annotations, you are importing from the *javax.persistence* package. You may see red squiggly lines under "students". We will fix that later.

2. Now we need to annotate our columns in the *students* table. In front of each getter in *model.Student*, add the @Column(name = "") annotation, where the name parameter should be the column name in the *students* table. Refer to the table_script.sql file to find the column names.

3. For the getStudentID() getter, make sure to add the @Id annotation to let Hibernate know that is where the primary key for *students* is stored.

4. Go to the applicationContext.xml file. Replace your current dataSource bean (the one configured to access the SQLite database) with the following:

```
<bean id="dataSource"
      class = "org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name = "driverClassName" value = "com.mysql.cj.jdbc.Driver"/>
   <property name = "url" value = "jdbc:mysql://localhost:3306/studentdb"/>
   <property name="username" value="root"/>
   <property name="password" value="password"/>
</bean>
```

   This configures the database connection to the MySQL database. **You may have to change your url, username, and password as required**. In the previous Hibernate assignment, these settings are directly configured in the hibernate.cfg.xml file. Other

than /studentdb, your URL should be the same as how you configured it in your Hibernate lab.

5. Run the program, you should see it running the same as before. Thanks to Spring, switching to MySQL was as simple as changing the configuration.

   Note: If you have seen an error related to time zone, you can fix it by running this command in your MySQL Command Line (refer to the Hibernate pre-class assignment if you need a refresher on how to get to the mysql command line):

   ```
   SET GLOBAL time_zone = '-07:00';
   ```

   This is because by default, your database could be set up to use the timezone format from the OS, but the database driver does not like timezones stored as text such as 'MDT'.

   We can now fix the red squiggly lines you may see in the Student.java file. They are there because IntelliJ Ultimate can analyze your data source to ensure you have the right names, but it requires you to set it up. It is not required for this assignment to set up your data source for intelliJ to recognize, but it can bring you convenience.

   To set up your data source in IntelliJ, copy the database url configured in your applicationContext.xml, click on the Database button on the right sidebar, click on the Add button, then click on *Data Source from URL*. Paste your URL in the URL field if it has not been autofilled, and select MySQL as the Driver. After clicking OK, make sure to enter the username and password to log in the database, and click on Test Connection. If the connection was successful, click on OK to finish setting up your database connection for IntelliJ. Now you can right click on the red squiggly lines and wait for the red light bulb icon to appear, click on the light bulb, the click on Assign data source, select the data source you just added, then click OK for the squiggly lines to be gone.

6. After making sure your MySQL database is working, we can now configure a SessionFactory bean and use it to do object-relational mapping instead of just doing queries through JDBC. We will configure Spring's LocalSessionFactoryBean, which is an implementation of Hibernate's SessionFactory. It accepts Hibernate configuration and the data source you have just set up. Go to your applicationContext.xml and paste the below code, which configures the LocalSessionFactoryBean:

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
   <property name="dataSource" ref="dataSource"/>
   <property name="annotatedClasses">
   <list>
```

```
        <value>model.Student</value>
    </list>
    </property>
</bean>
```

The property *annotatedClasses* can be configured to include the classes you will annotate as an Entity. This is similar to the *mapping* property in the hibernate.cfg.xml file.

7. Now we can configure a new implementation of StudentDAO. Create a new class in the *dao* package and name it *StudentDAOHibernate*. Have it implement StudentDAO, declare an instance variable for SessionFactory, and have a constructor to initialize SessionFactory. You should have the following so far:

```
package dao;

import model.Student;
import org.hibernate.SessionFactory;

import java.util.List;

public class StudentDAOHibernate implements StudentDAO {
    private SessionFactory sessionFactory;

    public StudentDAOHibernate(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    @Override
    public Student getStudentById(int id) {
        return null;
    }

    @Override
    public List<Student> getAllStudents() {
        return null;
    }
}
```

8. To ensure that our project will be using this implementation of StudentDAO, delete the @Component() annotation in StudentDAOImpl, and instead put the annotation in front of the class declaration of StudentDAOHibernate. By doing so, Spring will automatically inject the SessionFactory into StudentDAOHibernate.

9. Now we can implement our DAO methods just like what was done in the Hibernate assignment. Copy and paste the following inside the method *getStudentById()*:

```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
```

```
        Student student = session.get(Student.class, id);

        transaction.commit();
        session.close();
        return student;
```

This is the same way you have coded your DAOs in the Hibernate assignment. When doing the assignment, you may have noticed the boilerplate code. With Spring this code can be simplified to one line. Before we do so, run the project to make sure that the first query (querying the student with Id=12345) was successful, but the second (querying all students) failed with a NullPointerException.

10. Go back to your project's applicationContext.xml file. We will implement an important part of the Spring Framework, called Spring Transactions. This will abstract all the transactional code we just put in our *getStudentById()* method. Copy and paste the following code in your applicationContext.xml file:

```
<bean id="transactionManager"
     class="org.springframework.orm.hibernate5.HibernateTransactionManager">
   <property name="sessionFactory" ref="sessionFactory"/>
</bean>

<tx:annotation-driven/>
```

You will be asked to import another schema into your XML file. Click on `<tx` and do the command Alt + Enter to import the tx schema. The above code instantiates Spring's HibernateTransactionManager, which contains all the transactional logic, and allows annotation driven configuration for transactions. Transactions are implemented by Spring the same way as we have done with logging and Spring AOP. By annotating a method with the annotation @Transactional, Spring will enable a proxy that will run necessary transactional code before and after the method call.

11. Go back to our StudentDAOHibernate class and annotate our *getStudentById()* method with *@Transactional* (You may be asked to import a Spring package or JavaX package. Select the Spring package to import).

12. Now delete all the transactional code in the method. Instead of declaring and opening a separate session variable, you can use *sessionFactory.getCurrentSession()*, which will manage the Session instances for you. You should be left with the following:

```
@Override
@Transactional(readOnly = true)
public Student getStudentById(int id) {
    return sessionFactory.getCurrentSession().get(Student.class, id);
}
```

Note the added parameter to the @Transactional annotation. For all methods that will only be reading your database, you can add the parameter `readOnly = true` to speed up the transaction.

13. Run your project again and make sure you are getting the same result as before where the first one succeeds but the second one fails.

14. Now copy and paste the following code to replace the *getAllStudents()* method:

```
@Override
@Transactional(readOnly = true)
public List<Student> getAllStudents() {
   return sessionFactory.getCurrentSession()
           .createQuery("select a from Student a", Student.class)
           .getResultList();
}
```

The query is not a typical SQL query, but a query language for ORM. More information on it is found here.

15. Run your project again. You should see the first two queries running as normal.

Note: We did not modify the ClassDAOImpl, so the last query for the new schedule is still being done via JdbcTemplate. Because we changed the data source to MySQL, JdbcTemplate was querying against the MySQL database.

Optional: Create more methods for StudentDAO and write your own tests to see how Hibernate stores objects into the database. You can use this starter file for your JUnit tests. The Hibernate documentation will help you learn how to use it.

Spring with Java Configuration

Spring can also be configured using JavaConfig. Using just Java syntax, we can configure our project the same way as using XML. This has the benefit of allowing the compiler to check your configuration data types and allows the compiler to help you keep your configuration and

normal code in sync. Remember that the file we will be configuring is not normal Java code. It should not include any business logic in your projects, but only code to configure beans.

1. In your src/main/java directory, create a new class named AppConfig.

2. On top of the class declaration, add the annotation *@Configuration*. This annotation lets Spring know that this Java class is a configuration file.

3. In the Main.java, replace the line `ApplicationContext context ...` with the following:
   ```
   ApplicationContext context = new
   AnnotationConfigApplicationContext(AppConfig.class);
   ```
   This line of code tells Spring to load the configuration in the AppConfig.java file instead of applicationContext.xml. Now every time you run your main method, Spring will look at the AppConfig.java file.

4. We have configured autowiring, scanning for aspects, and annotation driven transactions in our applicationContext.xml file with the following statements:

   ```
   <context:component-scan  base-package="dao,aspect" />
   <!--Use @ComponentScan(basePackages = {"dao", "aspect"}) -->

   <aop:aspectj-autoproxy />
   <!-- Use @EnableAspectJAutoProxy -->

   <tx:annotation-driven/>
   <!-- Use @EnableTransactionManagement -->
   ```

   To enable these functions using JavaConfig, you will need to type the annotations shown in the comments above under the *@Configuration* annotation and before the class declaration of AppConfig.

5. Now in the class body, every method call with an *@Bean* annotation is a corresponding bean definition in applicationContext.xml. Below is how the bean definition for the *dataSource* bean should be configured using JavaConfig. It is an example of setter injection. As you understand the structure of the bean definition in JavaConfig, type the following (without the comments) to instantiate your data source:

```java
@Bean // Every bean definition is annotated with @Bean and is public
public DataSource dataSource() {
    // The bean's class is the type declaration. Seen here as DataSource.
    // XML's <bean id="dataSource" ../> is represented as dataSource() in JavaConfig.

    // Below is the instantiation of the dataSource bean. Each bean would be
    // initialized to the class it represents using the new command.
    DriverManagerDataSource dataSource = new DriverManagerDataSource();

    // Use setters to configure the bean to be instantiated.
    // The following is simply setter injection:
    dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost:3306/studentdb");
    dataSource.setUsername("root");
    dataSource.setPassword("password");

    // Return the instantiated bean:
    return dataSource;
}
```

Note that for the DataSource class, you should have imported *javax.sql.DataSource*.

6. The following is an example of instantiating the Session Factory. Include this in your AppConfig.java:

```java
@Bean
public LocalSessionFactoryBean sessionFactory() {
    LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();

    sessionFactory.setDataSource(dataSource()); // There are two ways to set your dataSource.
    // Instead of declaring a dataSource parameter, we directly "call" the method that
    // instantiated the dataSource() bean. Note that this is not making another instance
    // of dataSource but instead is referring to the bean created by the dataSource() method.

    sessionFactory.setAnnotatedClasses(Student.class);

    Properties properties = new Properties(); // This is how we can set Hibernate properties
    properties.setProperty("hibernate.show_sql", "true");
    sessionFactory.setHibernateProperties(properties);
    return sessionFactory;
}
```

Another example of using the dataSource bean to instantiate another bean is seen here with JdbcTemplate. Include it in your AppConfig.java:

```java
@Bean // An example of constructor injection.
// Note that dataSource is a parameter here. This is a type of autowiring.
// We let Spring take care of finding the right bean that is of a
// DataSource type, and use it to inject into this instance of JdbcTemplate
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
    return new JdbcTemplate(dataSource);
}
```

7. Now by yourself, instantiate the bean for the HibernateTransactionManager class. It is in the same format as the JdbcTemplate bean definition, except that you will be injecting a SessionFactory. After you are done, run your project again. Make sure that the AOP and Hibernate configurations are running properly.

8. Your project should now be outputting all two queries. After running your project successfully, save all of the output from the console to submit on Canvas.

Submission

Submit the following on Canvas:

1. LoggerAspect.java
2. Student.java
3. StudentDAOHibernate.java
4. applicationContext.xml
5. AppConfig.java
6. A screenshot or printout of your output console.