**Project 1**

**Note:** If you have Eclipse installed on your machine correctly, you may skip to the next page.

For this project we will be using the following tools:

1. Eclipse. If you don't have this available on your machine, go here. I recommend Eclipse IDE - Java For Developers. Please **do not** get the Enterprise Java Version. <https://www.eclipse.org/downloads/packages/>
2. You will also need the latest JDK to run Eclipse. I recommend JDK 17. For that see this link: <https://www.oracle.com/java/technologies/downloads/>
3. Lastly, you will need a connection to the internet.

**If you're having problems starting Eclipse**, and it gives you an error message, this may be because your computer has two different versions of a JRE installed. Again, please make sure to install the correct version of JDK for your machine (most likely 64 bit). If you're still having problems, go to the “eclipse.ini” file located in the root Eclipse folder, and add this line: (note: the correct <version number>). This will point Eclipse to the correct JVM.

“-vm

C:/Program Files/Java/jdk<version number>/bin/javaw.exe”

It will look something like this:

“-vm

C:/Program Files/Java/jdk1.8.0\_172/bin/javaw.exe”

Go to “C:/Program Files/Java/” on your machine to find the correct jdk <version number>.

For questions, or if you still have problems, please e-mail me: [eschaeffer8@stlcc.edu](mailto:eschaeffer8@stlcc.edu)

Before we begin:

A good idea, **but is not required** is to “close” any open project that you may have open in Eclipse. This will avoid seeing other errors/warnings from other projects. You may need our in-class “Sales” project later so you can leave that one open. To close a project “right-click” on the project name (or multi-select projects by holding CTRL and clicking on each one), and select “Close Project”. This will not change or impact your other project. You can open it again later. (See below for screenshot)

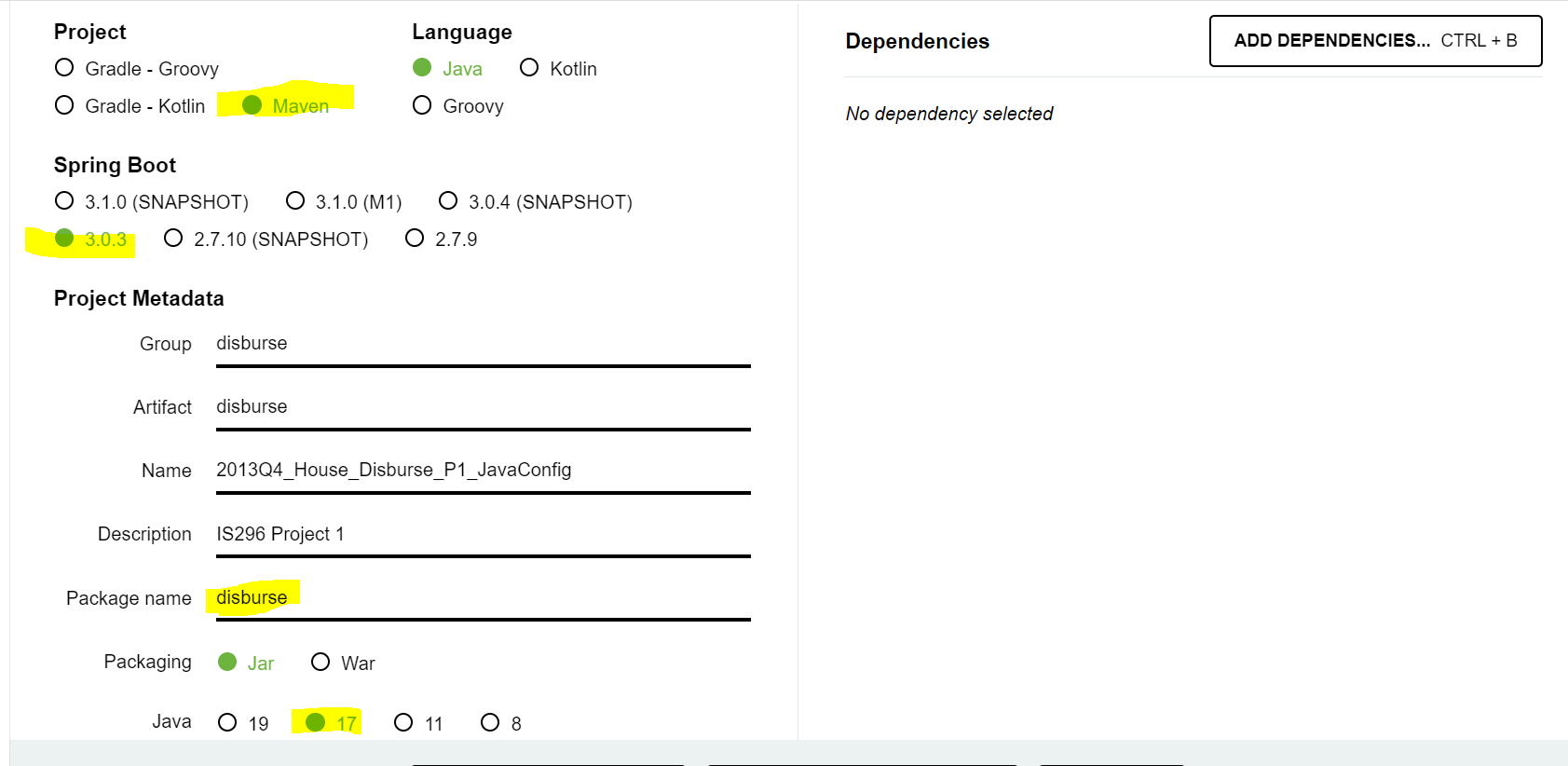


**Section I – Creating the Project in Eclipse & Using Maven Dependency Management**

a. Go to the Sprint Initializr website: <https://start.spring.io/>

b. Use the following settings.

1. Ensure the Project is marked **Maven**.
2. Ensure the **Package name** is just **disburse.** It will attempt to change as you type the Group and Artifact names.



* 1. In Eclipse in the “Package Explorer”, click on the pom.xml file.
  2. Add the same dependencies from your current in-class project. If you need to, you can go here: <https://github.com/edwinschaeffer> and select our current project. Click on the pom.xml file in Github and copy the <dependencies></dependencies> tags and (everything in between) and paste in your project and click “Save”.



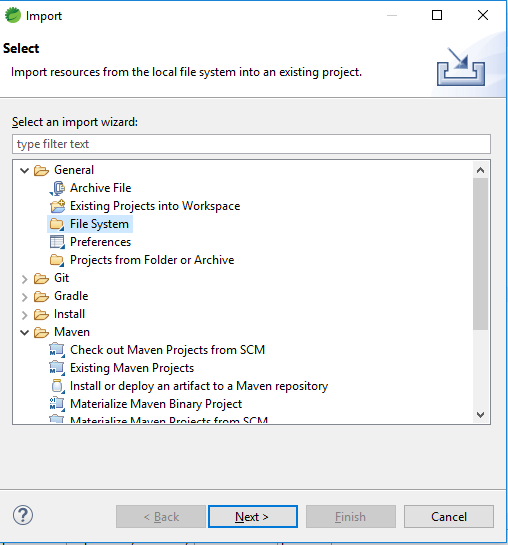
* 1. Import the SQL files included in the project directory into your project by going in Eclipse and dragging and dropping the files into your **src/main/resources** folder (easy way)

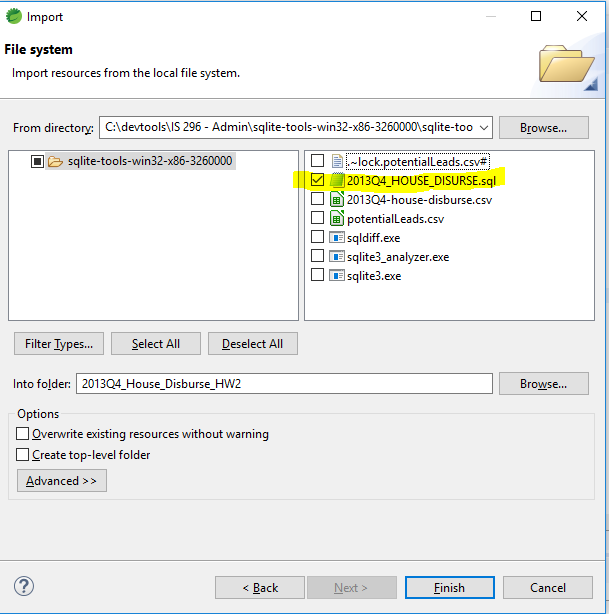
**OR** alternatively by going to:

a. File → Import... → Expand “General” → File System → Click “Next” (**See Figure 1 below**)

* 1. Select “Browse...”. Navigate to the folder where the SQL files were downloaded. Select the file on the RIGHT side of the pane and click “Finish”. (**See Figure 2 below**)

Figure 1



 Figure 2 – Select on the right side.

**Section II. Creating & Organizing Classes Using Packages**

In this assignment we are going to use a real-life data set to answer some questions about the data.

* + 1. First we need to create a package to hold “Value Objects”.

**a)** Right-click on “src/main/java” in the Package Explorer. Select “New” → “Package”. Name the package “**disburse.vo**”.

* + 1. The first class we need is a value object to hold return values. First we're going to create a class called HouseDisburseDetail.

**a)** Right-Click in the Package Explorer “src/main/java”. Select “New” → Class

**b)** For the class name call the class: HouseDisburseDetail.

**c)** When creating the class make sure the class is in the Package: **“disburse.vo”**

**d)** Click “Finish”

* + 1. Place the following fields in the Class. (“e.g. private Date BIOGUIDE\_ID”)

4. Now right click in the code area under the variable you just created. Select “Source” → Generate Getters and Setters.

5. Choose “Select All”, then click “Generate”, and save the file.

* + - * 1. Create another class called “DisburseApp”. Make sure the package is “**disburse**”. Also check the box for creating method stubs for “public static void main(String[] args)”



Create a new package called “disburse.dao”

Create a Class called JdbcHouseDisburseDetailDAO. (No extra boxes checked) Make sure the class is created inside the “disburse.dao” package.

Copy and paste the code in the accompanying file JdbcHouseDisburseDetailDAO.txt to your JdbcHouseDisburseDetailDAO class.

Notice the JdbcHouseDisburseDetailDAO class has two methods: A setter method to set the DataSource, and a method that retrieves a List of HouseDisburseDetail objects by using an SQL Select query.

Create a new package called “config”

Create a new class called “HouseDisperseConfig”. Make sure when creating the Class the class is in the “config” package.

**Section III. Creating and Wiring Spring Beans Using JavaConfig**

**Note: If you need help on this section use our in-class project. I also put a copy of the completed project on Blackboard. You can put this project into your local Eclipse by going to File → “Open Projects From File System” → Directory... → Choose the Directory where the “src” folder is located, and click “Finish”**

1. Return to the “HouseDisburseConfig” class in Eclipse. Mark the Class as a Spring configuration class by putting the @Configuration annotation on the class at the **class level.** You will then likely have to put your cursor over the error and “quick fix” by bringing in the Spring Configuration Annotation import.
2. Use the code below to create a Spring Bean that builds an “in-memory” database from the supplied sql file. (The SQL file should be in src/main/resources per earlier instructions) Make sure to “quick fix” any needed imports. Also for the DataSource select the **javax.sql.Datasource** import.

@Bean

**public** DataSource dataSource(){

**return** **new** EmbeddedDatabaseBuilder()

.addScript("classpath:2013Q4\_HOUSE\_DISBURSE.sql").build();

}

1. Return to the “**DisburseApp.java**” class in the disburse package. Copy/paste the following code inside the main method. “Quick Fix” any imports needed.

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.register(PotentialLeadsConfig.**class**);

ctx.refresh();

1. Right click on DisburseApp.java in Eclipse and select “Run As… Java Application.” This is to check to make sure you have no errors. If you get an error, check your Maven pom.xml that there is no lines that say <scope>test</scope>.
2. Return to the “HouseDisburseConfig” class in Eclipse.

* Create another Bean (“public” same as above) named “houseDisburseDAO” that returns a JdbcHouseDisburseDetailDAO object and takes a parameter of type DataSource named “**ds**” (very important to name this “ds” !!)
* In the Bean method create a variable named “hdDAO” and have it store a new instantiated JdbcHouseDisburseDetailDAO object.
* Use the setter method in the stored hdDAO variable to set the DataSource from the parameter.
* Return the variable hdDAO.

1. (Repeat Step 4.) Run the application. You should not receive an error.
2. Comment out the “dataSource” Bean below by selecting the text and pressing CTRL+SHIFT+/ . It should now look like this:

/\* @Bean

**public** DataSource dataSource(){

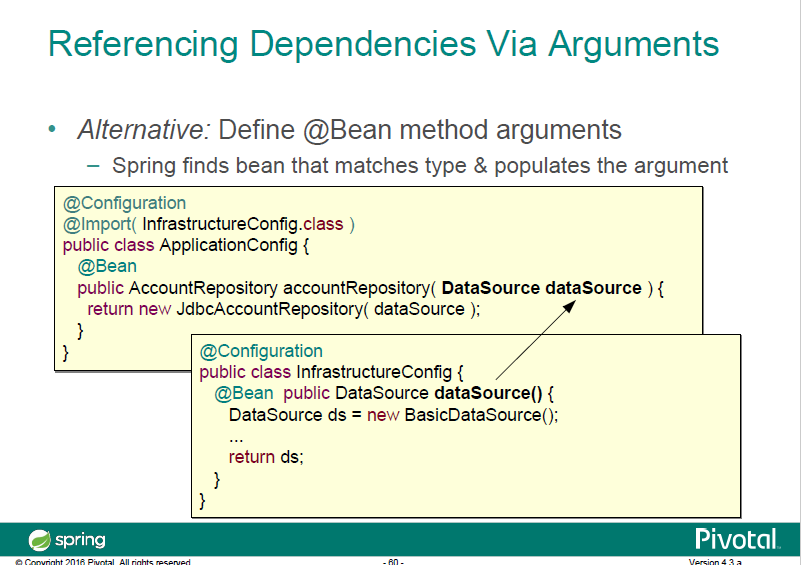
**return** **new** EmbeddedDatabaseBuilder()

.addScript("classpath:2013Q4\_HOUSE\_DISBURSE.sql").build();

} \*/

1. (Repeat Step 4.) Run the application. You **should receive** an error. Read the Exception thoroughly. Remember when your Bean method takes a parameter, you are “implying” that another Bean will be injected into it. This is called “implied injection” which is one way to perform Bean “wiring”. Spring knows to inject the DataSource Bean into the JdbcHouseDisburseDetailDAO Bean you just made because the method parameter in the JdbcHouseDisburseDetailDAO Bean matches the return **type** of the DataSource Bean.

See below for more reference:



1. Uncomment the DataSource Bean you previously commented out. (The easy way is to just use CTRL+Z.)
2. Next we’ll create another Bean with a return type of DataSource and call it “dataSource2”. The code is below. Note, we are using the other potentialLeads.sql file in dataSource2.

@Bean

**public** DataSource dataSource2(){

**return** **new** EmbeddedDatabaseBuilder()

.addScript("classpath:potentialLeads.sql").build();

}

1. (Repeat Step 4.) Run the application. You **should receive** an error. Read the Exception thoroughly. As I was explaining in Step 8 that Spring uses “implied injection” that will inject into another Bean if Spring can find a matching **type**. The JdbcHouseDisburseDetailDAO Bean has a dependency of a DataSource. However, now we have **TWO** Beans that return the same **type** of a DataSource. So should Spring inject the return value of “dataSource” or “dataSource2” into the JdbcHouseDisburseDetailDAO Bean? Spring doesn’t know. Hence the error.
2. On dataSource2 add “ds” to the Bean annotation so that it now looks like this:

@Bean(“ds”)

**public** DataSource dataSource2(){

**return** **new** EmbeddedDatabaseBuilder()

.addScript("classpath:potentialLeads.sql").build();

}

1. Repeat Step 4.) Run the application. You should **not** receive an error. The code you created from Step 5 should be the following. If you’re receiving an error, check to make sure this Bean below matches your code:

@Bean

**public** JdbcHouseDisburseDAO houseDisburseDAO(DataSource ds) {

JdbcHouseDisburseDAO hdDAO = **new** JdbcHouseDisburseDAO();

hdDAO.setDataSource(ds);

**return** hdDAO;

}

By adding “ds” to your Bean annotation in dataSource2, you are giving the “dataSource2” Bean a **name** of “ds”. In addition you are telling Spring that it is OK to inject the Bean **named** “ds” if it finds a method parameter variable that **matches both** its **name and type**. Notice the parameter above is “DataSource ds”. Spring will first try to inject by **type** only, but as in this case it found two DataSource types so then it resorts to injecting by **name and type**.

1. Remove the (“ds”) from the “dataSource2” Bean. Also, change the name of the method from “dataSource2” to “ds”. The new code should look like this:

@Bean

**public** DataSource ds(){

**return** **new** EmbeddedDatabaseBuilder()

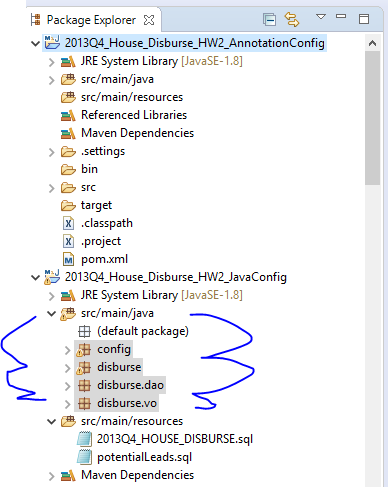
.addScript("classpath:potentialLeads.sql").build();

}

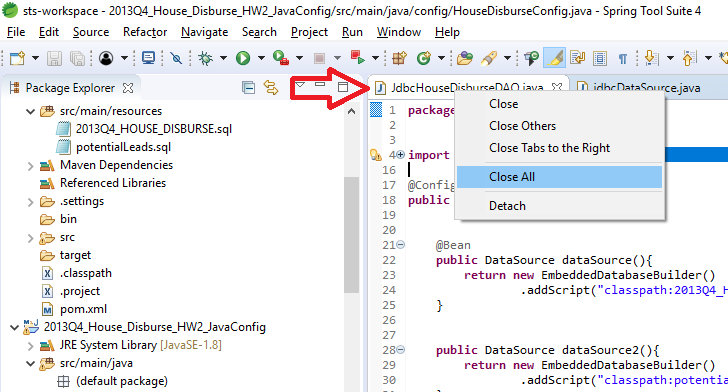
1. Repeat Step 4.) Run the application. You should **not** receive an error. Bean’s always have default names. When we don’t specify a name, the **default name is the name of the bean method.** Even though Spring gave us this capability of a default name, Pivotal (the company who maintains Spring) recommends to never rely on these auto-generated names.

**Section IV. Creating and Wiring Beans using AnnotationConfig**

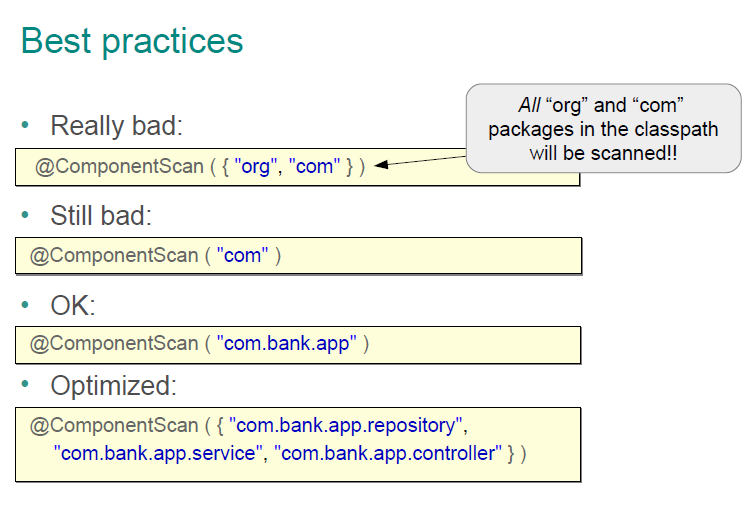
1. Create another project by repeating the steps of Section I only name this project “**2013Q4\_House\_Disburse\_P1\_AnnotationConfig”.**
2. Copy the packages from the old project to the new project by holding CTRL and clicking on each separate package.
3. In Eclipse click “Edit → Copy”, then click on the new project package select “Edit → Paste”. Yes, there will be errors, but when we fix the Maven pom.xml file, they will go away.



1. Repeat the same procedures for the two .sql files in src/main/resources. Your folder structure should look EXACTLY like mine above. I made a mistake and copied the source folder, and it created an extra sub-folder called “resources”. So, please check after the paste that there is no additional irregularities and your project tree looks like the screenshots.
2. Open the pom.xml file in the old project. Copy the <dependencies> tags and everything included in between. Paste this in the same place in the new pom.xml. Save the file and the errors should now go away.
3. Right click on one of the tabs in the code window (as shown below) and select “Close All” to clear the open files from the old project.



1. Open the HouseDisburseConfig.java file in our new project and comment out all the @Bean methods.
2. Inside the HouseDisburseConfig class at the class level underneath the current @Configuration annotation, place this annotation: @ComponentScan(“disburse”) . This means Spring will scan every file in every package that begins with “disburse”. So, “disburse”, “disburse.dao”, and “disburse.vo” will be scanned. This is generally considered bad practice because **every** class in your classpath gets scanned **INCLUDING** .jar dependency files. So, if we used “org” instead of “disburse”, this would scan EVERY class including the Maven dependencies and every Spring jar. This could result in slower startup times for large applications, or even worse a true story of mine is I was using a third party library that had it’s own Spring implementation, and things got really weird when a developer pointed our scanning at “com”. See below on what the best practices are. Note: we do not need and should not need to scan the “config” package; hence, this is the reason why we didn’t name it “disburse.config” for example.



1. Open the JdbcHouseDisburseDAO.java class. Place a @Repository annotation at the class level. Import any necessary libraries.
2. Place an @Autowired annotation at the method level of the “setDataSource” method. Your code at the top of this class at this point should look like the following:

@Repository

**public** **class** JdbcHouseDisburseDAO {

**private** DataSource dataSource;

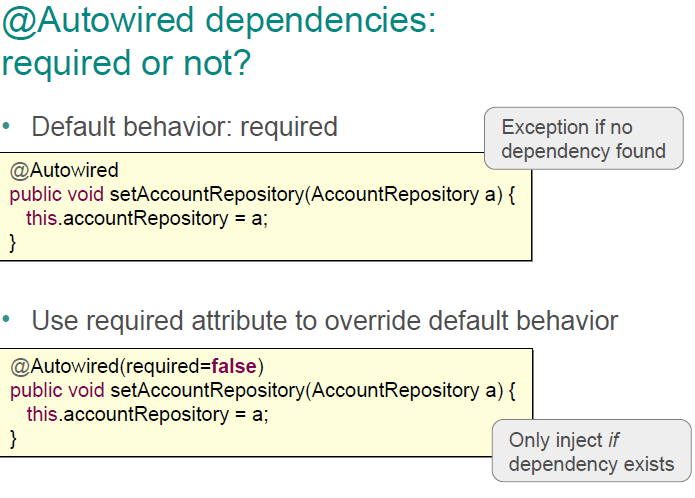
@Autowired

**public** **void** setDataSource(DataSource dataSource) {

**this**.dataSource = dataSource;

}

}

1. Expand the “disburse” root package, and right click on the “DisburseApp.java” file in our new project. Right-click DisburseApp.java → Run As… → Java Application. (It’s important to do this in our new project we just made so it will run the current project and not the old one) After running this as a Java Application you **should** receive an error.
2. Closely examine the error. The reason for the error is similar to Section 3: Step 8. However, this really is about default behavior: If the Bean exists, inject it! Therefore it’s enforcing that the Bean must exist and is “required”. However, no Bean was found. In this step on the @Autowired annotation change it to read this: @Autowired(required = false).
3.  Re-run the application as in Step 13. You **should not** receive an error. This changes the default behavior. See below as to why an error was not received.
4. Remove the “(required = false)” from the @Autowired annotation.
5. At the class level underneath the @Repository annotation add another annotation: @Lazy.
6. Re-run the application as in Step 13. You **should not** receive an error.
7. Bean names can be specified or auto-generated in AnnotationConfig as well. The default bean name is a “de-capitalized” classname. So, if the classname is TransferServiceDAO, the default bean name is “transferServiceDAO”. Therefore, you simply just lower case the first letter of the classname. In this next step we are going to retrieve an instance of a Bean from the Spring Container by calling it by name. Return to DisburseApp.java. Add this line code as the last line in the main method:

JdbcHouseDisburseDAO hdDAO = (JdbcHouseDisburseDAO)ctx.getBean("jdbcHouseDisburseDAO");

Your code should look like this:

**public** **class** DisburseApp {

**public** **static** **void** main(String[] args) {

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.register(HouseDisburseConfig.**class**);

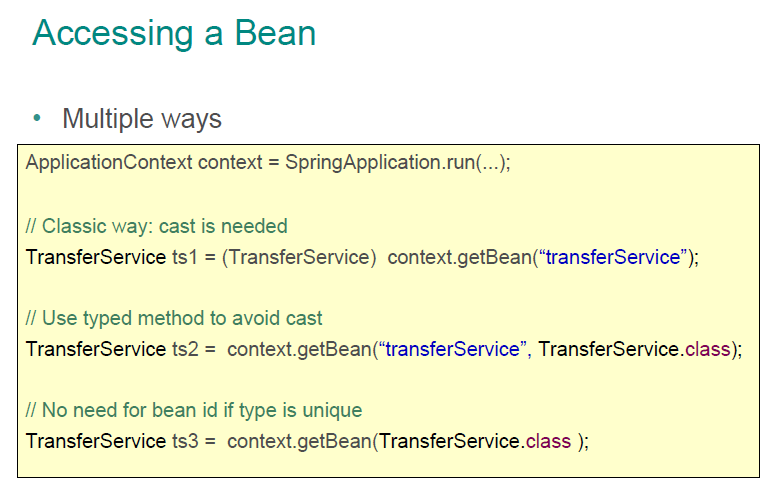
ctx.refresh();

JdbcHouseDisburseDAO hdDAO = (JdbcHouseDisburseDAO)ctx.getBean("jdbcHouseDisburseDAO");

}

}

1. Re-run the application as in Step 13. You **should** receive an error. See below as the alternate ways of accessing a Bean.



1. Return to the HouseDisburseConfig.java file. Uncomment the first DataSource Bean such that you have a Configuration class with only Bean method like this:

@Configuration

@ComponentScan("disburse")

**public** **class** HouseDisburseConfig {

@Bean

**public** DataSource dataSource(){

**return** **new** EmbeddedDatabaseBuilder()

.addScript("classpath:2013Q4\_HOUSE\_DISBURSE.sql").build();

}

// more methods commented out

}

1. Re-run the application as in Step 13. You **should not** receive an error.
2. Return to the DisburseApp.java file Copy/Paste the following code at the end of the main method.

List<HouseDisburseDetail> hddList = hdDAO.selectQuery("Select BIOGUIDE\_ID, OFFICE, PURPOSE, PAYEE, AMOUNT from T\_2013Q4\_HOUSE\_DISBURSE LIMIT 1");

**for** (HouseDisburseDetail hdd : hddList) {

System.***out***.println(hdd.getBIOGUIDE\_ID() + "\t" + hdd.getOFFICE() + "\t" + hdd.getAMOUNT());

}

The whole code should look like this:

**package** disburse;

**import** java.util.List;

**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;

**import** config.HouseDisburseConfig;

**import** disburse.dao.JdbcHouseDisburseDAO;

**import** disburse.vo.HouseDisburseDetail;

**public** **class** DisburseApp {

**public** **static** **void** main(String[] args) {

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.register(HouseDisburseConfig.**class**);

ctx.refresh();

JdbcHouseDisburseDAO hdDAO = (JdbcHouseDisburseDAO) ctx.getBean("jdbcHouseDisburseDAO");

List<HouseDisburseDetail> hddList = hdDAO.selectQuery("Select BIOGUIDE\_ID, OFFICE, PURPOSE, PAYEE, AMOUNT from T\_2013Q4\_HOUSE\_DISBURSE LIMIT 1");

**for** (HouseDisburseDetail hdd : hddList) {

System.***out***.println(hdd.getBIOGUIDE\_ID() + "\t" + hdd.getOFFICE() + "\t" + hdd.getAMOUNT());

}

}

}

1. Return to the JdbcHouseDisburseDAO class. Remove the @Lazy annotation.
2. Delete the @Autowired annotation over the setDataSource method. Place a new @Autowired annotation over the field “private DataSource dataSource;”
3. Comment out the setDataSource method.
4. Your code at the top of the method should now look like the following:

@Repository

**public** **class** JdbcHouseDisburseDAO {

@Autowired

**private** DataSource dataSource;

/\* public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}\*/

1. Re-run the application as in Step 13. You **should not** receive an error. Also, notice the DataSource was set and the field was “private”. Spring had no trouble doing “field level” injection even with the field being “private”. However, field injection is not recommended because it’s difficult to unit test.
2. Create a Constructor in the same class that takes a DataSource argument. Place the @Autowired annotation over your constructor. Your code should now look like the following:

@Repository

**public** **class** JdbcHouseDisburseDAO {

**private** DataSource dataSource;

@Autowired

**public** JdbcHouseDisburseDAO(DataSource dataSource1) {

**this**.dataSource = dataSource1;

}

/\* public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}\*/

1. Re-run the application as in Step 13. You **should not** receive an error.
2. Now delete the @Autowired annotation over the constructor. Re-run the application as in Step 13. You **should not** receive an error. What? Why not? In this case Spring is smart enough to break its own rule. The @Repository annotation must instantiate a JdbcHouseDisburseDAO object, and since it has a matching Bean with type “DataSource”, Spring injects the DataSource. Pivotal of course does not recommend this. For consistency, it’s best to use @Autowired on Constructors as well.
3. Return to the JdbcHouseDisburseDAO.java file. Delete the constructor, and uncomment the setDataDource method. Place the @Autowired annotation at the method level over the setDataSource method
4. Your code should now look like the following:

@Repository

**public** **class** JdbcHouseDisburseDAO {

**private** DataSource dataSource;

@Autowired

**public** **void** setDataSource(DataSource dataSource) {

**this**.dataSource = dataSource;

}

1. Return to the HouseDisburseConfig.java file. Uncomment the second Bean method. Your code should look like the following.

@Configuration

@ComponentScan("disburse")

**public** **class** HouseDisburseConfig {

@Bean

**public** DataSource dataSource(){

**return** **new** EmbeddedDatabaseBuilder()

.addScript("classpath:2013Q4\_HOUSE\_DISBURSE.sql").build();

}

@Bean

**public** DataSource dataSource2(){

**return** **new** EmbeddedDatabaseBuilder()

.addScript("classpath:potentialLeads.sql").build();

}

1. Re-run the application as in Step 13. You **should not** receive an error.
2. Change the method name of the first Bean to “ds”. The code should now look like this below.

@Configuration

@ComponentScan("disburse")

**public** **class** HouseDisburseConfig {

@Bean

**public** DataSource **ds**(){

**return** **new** EmbeddedDatabaseBuilder()

.addScript("classpath:2013Q4\_HOUSE\_DISBURSE.sql").build();

}

@Bean

**public** DataSource dataSource2(){

**return** **new** EmbeddedDatabaseBuilder()

.addScript("classpath:potentialLeads.sql").build();

}

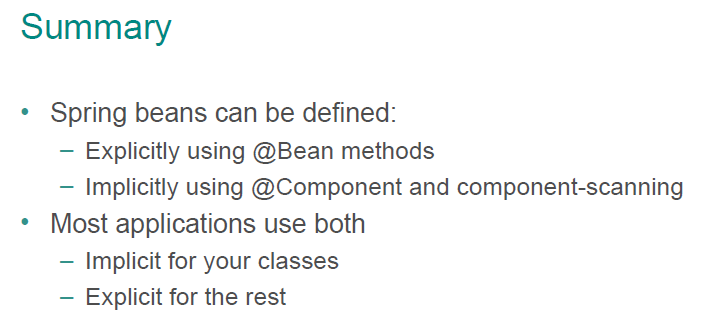
1. Re-run the application as in Step 13. You **should** receive an error.
2. Return to the JdbcHouseDisburseDAO.java file. Place an additional annotation under the @Autowired annotation called @Qualifer(“ds”). @Qualifier is another way of matching a Bean’s name at an injection point. For details see this link: <https://www.logicbig.com/tutorials/spring-framework/spring-core/inject-bean-by-name.html>

(Note: the syntax @Bean(name = “something”) is equivalent to @Bean(“something”) )

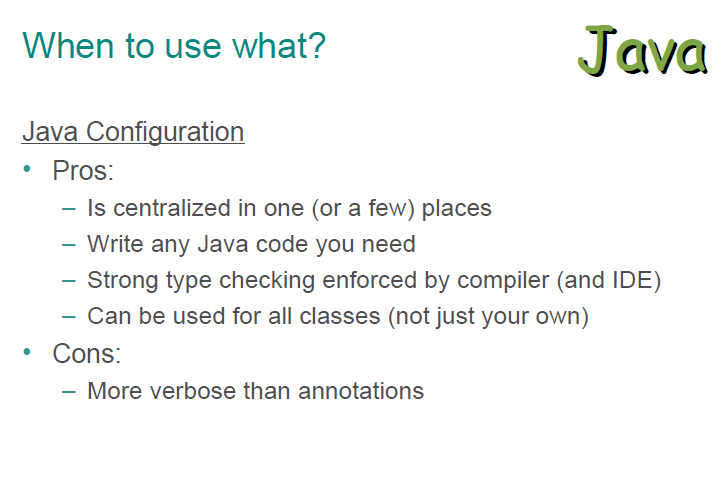
1. Re-run the application as in Step 13. You **should not** receive an error.
2. In the same JdbcHouseDisburseDAO change the @Repository annotation to @Service. Now re-run the application. You **should not** receive an error.
3. In the same JdbcHouseDisburseDAO change the @Service annotation to @Component. Now re-run the application. You **should not** receive an error.

@Service, @Component, @Repository all declare a class as a Bean using “AnnotationConfig”. Always remember when a Bean is instantiated it is held in the Spring container (otherwise known as the “application context”)  
 forever until the application stops (or unless a different lifecycle “scope” is specified for the Bean). @Component, @Service, and @Repository all are functionally the same. The reason why we use different annotations is: **1.** We are saying to the person who is maintaining the application in the future that @Service holds business logic and calculations, @Repository holds our Data Access Objects (DAOs), and @Component fits neither definition, but is declared as a bean. **2.** In addition as we get into Aspect Oriented Programming (i.e. AOP) where we want to monitor the actions of our Beans, it will be very helpful for us to separate the logic because we may want to monitor database transactions, but not calculations. Only the different annotations can tell us that.

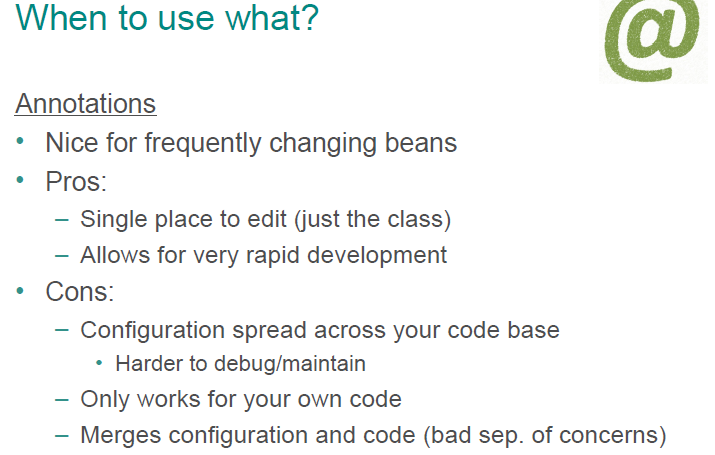
The next page is a Summary of what we learned so far. Also, the Pros and Cons of JavaConfig vs. AnnotationConfig.



**JavaConfig**



**AnnotationConfig**



**Section V. Creating and Wiring Beans using XMLConfig**

XMLConfig is the oldest method of Spring Dependency Injection. It is also now the **least** preferred way to configure Spring. Any older Spring application however you will run into, will more than likely have XML Config. Spring still supports this method of configuration, but Pivotal has pulled any type of questions about this configuration from the Spring certification exam. Generally, older developers prefer the XML way because that’s what they are used to. Newer developers prefer the JavaConfig way. AnnotationConfig really should only be used when developing an application. I don’t recommend developers using AnnotationConfig in Production because there really isn’t an easy way to figure out which classes are Beans besides manually going through each and every class. Eclipse does provide a way to figure out which classes are annotated, but this method shouldn’t really be relied upon.

1. In this section we will be creating a new project. Let’s call it “**2013Q4\_House\_Disburse\_P1\_XMLConfig**”
2. Repeat steps 1-8 in the previous section regarding copying packages.
3. **Remove** all annotations from the JdbcHouseDisburseDAO class. (if copied)
4. **Delete** the “config” package by right-clicking on it and selecting “Delete”
5. Go to the DisburseApp.java file. Delete the lines of code in the main method such that your code now matches this: (Delete the import **config.HouseDisburseConfig )**

**public** **class** DisburseApp {

**public** **static** **void** main(String[] args) {

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("HouseDisburseConfig.xml");

JdbcHouseDisburseDAO hdDAO = (JdbcHouseDisburseDAO)ctx.getBean("jdbcHouseDisburseDAO");

List<HouseDisburseDetail> hddList = hdDAO.selectQuery("Select BIOGUIDE\_ID, OFFICE, PURPOSE, PAYEE, AMOUNT from T\_2013Q4\_HOUSE\_DISBURSE LIMIT 1");

**for** (HouseDisburseDetail hdd : hddList) {

System.***out***.println(hdd.getBIOGUIDE\_ID() + "\t" + hdd.getOFFICE() + "\t" + hdd.getAMOUNT());

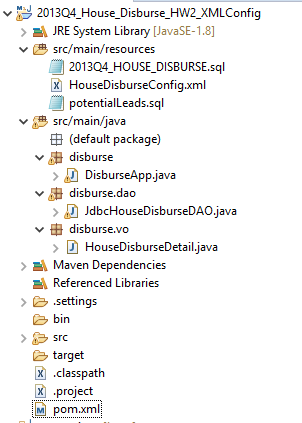
}

}

}

1. Create a new XML File in the **src/main/resources** source folder. Right-click on src/main/resources in the Package Explorer → Choose “Other…” → XML File →Click “Next” → name the file “HouseDisburseConfig.xml” → Click “Finish”

Ensure your project structure looks like this:



1. Copy and replace the entire contents of the XML file with the following code below and save it:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:jdbc=*"http://www.springframework.org/schema/jdbc"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/jdbc*

*http://www.springframework.org/schema/jdbc/spring-jdbc.xsd*

*http://www.springframework.org/schema/context*

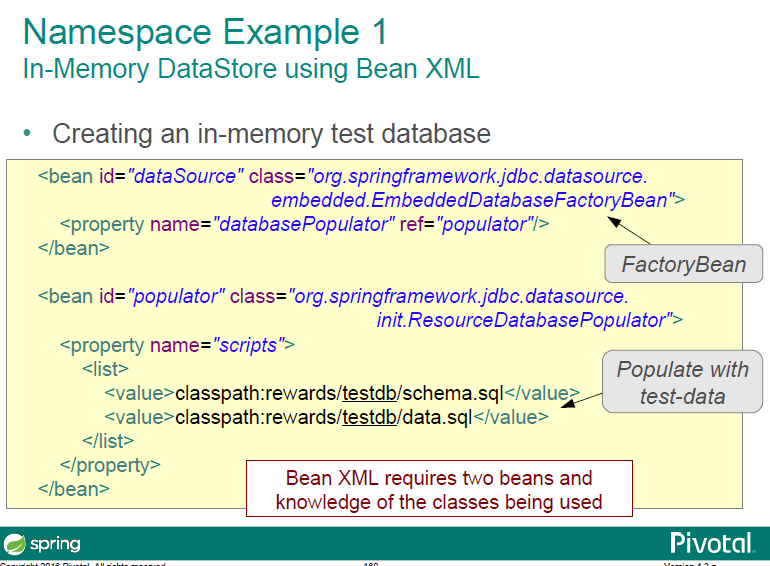
*http://www.springframework.org/schema/context/spring-context.xsd"*>

</beans>

Spring needs the namespace declarations (xmlns) and schema documents (.xsd) to define how tags are interpreted.

Throughout the project we have used JavaConfig to configure a DataSource(s). XMLConfig is limited to how it can initialize beans on start up. For example, you can not execute an init-method **and** pass an argument to it in XMLConfig. In JavaConfig you can.

To make things easier Spring evolved the namespace for DataSources. The tags we normally use in XML for instantiating beans is <bean>. However, see the slide below. The “normal” way to use Beans to instantiate the embedded database is we would have to use **two** Beans. If we use the JDBC Namespace, we can avoid this process. “Under the hood” Spring is still using two beans. **Note, we still are creating a DataSource Bean** with an embedded database, but this is just named differently, and the code is shortened.



1. By having the highlighted lines below in our namespace we can use a “jdbc” tag and **not** resort to using the two <bean> tags listed above in the alternative method. (No action needed on this step)

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:jdbc=*"http://www.springframework.org/schema/jdbc"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/jdbc*

*http://www.springframework.org/schema/jdbc/spring-jdbc.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*>

<jdbc:embedded-database id=*"ds"*>

<jdbc:script location=*"classpath:2013Q4\_HOUSE\_DISBURSE.sql"*/>

</jdbc:embedded-database>

</beans>

1. Copy/Paste (or change) the code of the XML file so that your code matches the code below. Notice the <jdbc> tags. This creates a Bean named “dataSource”. As shown from the slide on the next page, this is functionally equivalent to the way we used to configure this in JavaConfig.

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:jdbc=*"http://www.springframework.org/schema/jdbc"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/jdbc*

*http://www.springframework.org/schema/jdbc/spring-jdbc.xsd*

*http://www.springframework.org/schema/context*

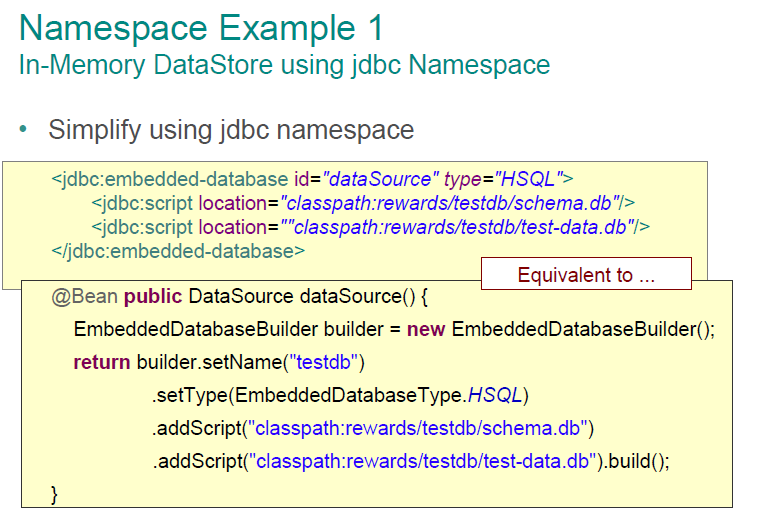
*http://www.springframework.org/schema/context/spring-context.xsd"*>

<jdbc:embedded-database id=*"ds"*>

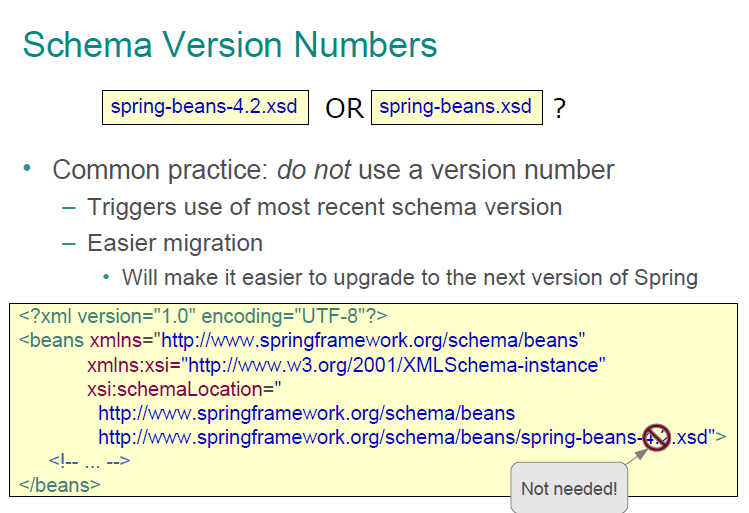
<jdbc:script location=*"classpath:2013Q4\_HOUSE\_DISBURSE.sql"*/>

</jdbc:embedded-database>

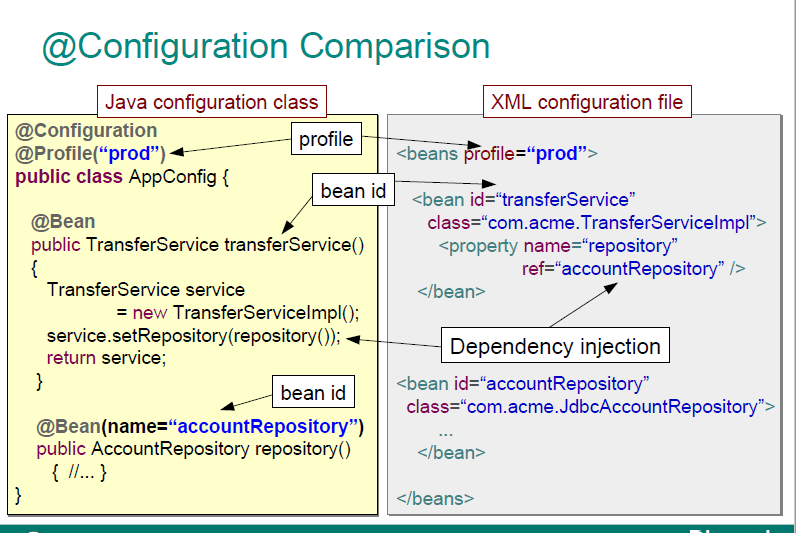
</beans>

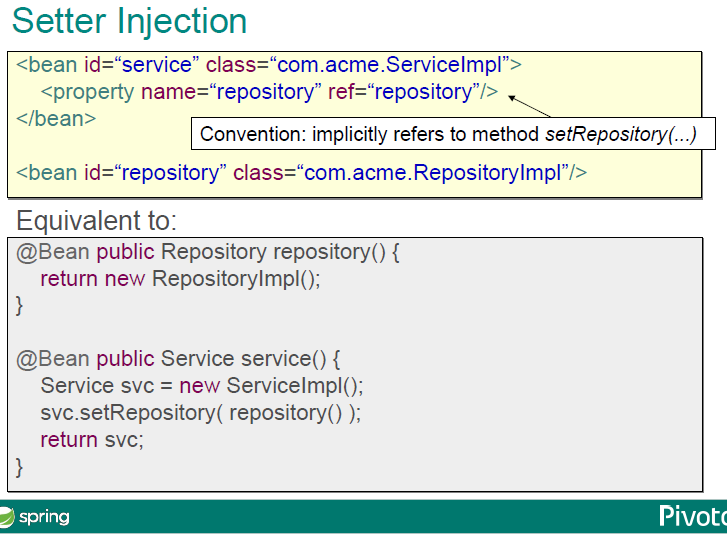


1. The XSD documents you see in the namespace refer to how the XML file is laid out. Think of them like a manifest of what tags and attributes we can use in the XML file to configure Spring. So, xsd version 3.2 might NOT let me use a <property> tag for example, but version 4.1 might introduce the tag. Pivotal recommends **not** putting a version number on the xsd. This may seem like a moot point since XML is on its way out from the Spring configuration space, but say you’re a new developer for an organization that still uses these XML configs. You look up on the internet how to configure a bean for SpringMVC, and you develop the solution, and it doesn’t work mysteriously. If the previous developer used a version number on the **xsd**, this could be the reason. See the below slide for best practices.



1. Now let’s make a Bean in XML for the **JdbcHouseDisburseDAO** class. See the slide below on how XML configuration compares to JavaConfig. In XML it is required to give a Bean an “id”. This serves as the Bean “name”. Remember in JavaConfig and AnnotationConfig Bean names are “auto-generated”. Here, we have to specify the name arbitrarily.
2. Next we have to supply a fully-qualified class name. The fully qualified class name for JdbcHouseDisburseDAO is disburse.dao. **JdbcHouseDisburseDAO**.
3. Now notice the <property> tag in the slide below. <property> tags **imply setter injection.** Repeat this in your sleep. **The “name” attribute is the shortened name of the SETTER method you wish to inject.** So what is the setter method name which you wish to inject? setDataSource, right? Here’s the tricky part. You take the “set” off and then you de-capitalize the word “DataSource” making it “dataSource”. Confused yet? The “ref” attribute is the name of the Bean you wish to inject. Examine the next two slides.





To accomplish setter injection... closely examine the code below. Copy/Paste this under the closing </jdbc> tags

<bean id=*"jdbcHouseDisburseDAO"* class=*"disburse.dao.JdbcHouseDisburseDAO"*>

<property name=*"dataSource"* ref=*"ds"*> </property>

</bean>

1. Right click in our current project on DisburseApp.java → Run As… → Java Application. You **should not** receive an error.
2. Return to the JdbcHouseDisburseDAO class. Comment out the “setDataSource” method. Create a constructor that takes in a DataSource as a parameter. The first couple lines of code should now look like the following:

**public** **class** JdbcHouseDisburseDAO {

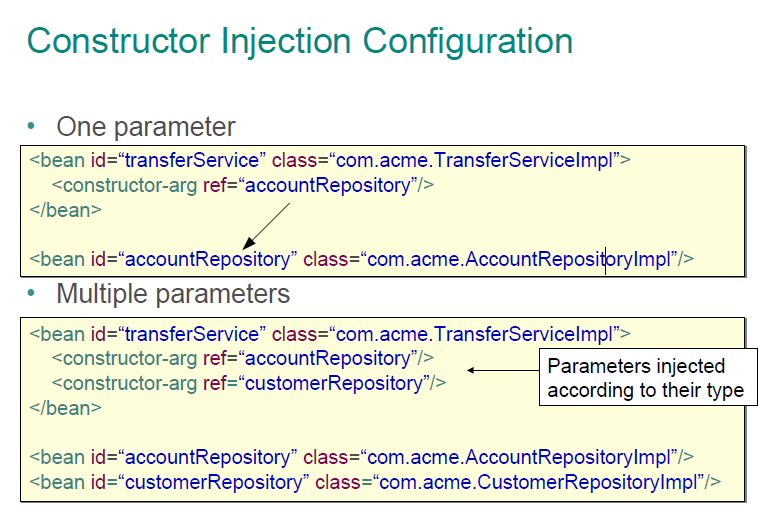
**private** DataSource dataSource;

**public** JdbcHouseDisburseDAO(DataSource dataSource) {

**this**.dataSource = dataSource;

}

1. Examine the slide below. Notice there is no longer “<property>” tags (because we’re not performing setter injection). On your own delete the <property> tags and use the self closing <contructor-arg /> tags below to inject the DataSource in the bean. Run the application. You should **not** receive an error.



1. You can actually still do AnnotationConfig in spring with XML. Only now, we’re using component scan XML tags.

In your HouseDisburseConfig.xml file comment out or delete the “jdbcHouseDisburseDAO” bean. Add the following in its place:

<context:component-scan base-package=*"disburse"* />

1. Place a @Repository annotation on the JdbcHouseDisburseDAO file and (optionally) an @Autowired annotation above the constructor.
2. Run the application. You should **not** receive an error.

**End of Project**