***Dependency Injection***

***Spring Autowiring***

***Servlet Dependency Injection***

***Spring Property Support***

***Servlet Dependency Injection***

***JSF & Spring***

***Web***

***Convert a Web Project to an Spring-MVC***

***Maven***

***Simple Spring Standalone using Maven***

***Databases***

***JDBC Database access***

***Catch JDBC Exceptions***

***Transaction***

***JDBC Access from a J2EE application***

***Property Support***

***Define a property file***

***Spring Autowiring***

***1) Activate Autowiring system***

i) Open the application context file

i.e: beans.xml, or applicationContext,xml

ii) Click on the tab Namespaces

iii) Check the option 'context'

iv) Click on the tab context (Created in step iii) )

v) Right Click on 'beans'

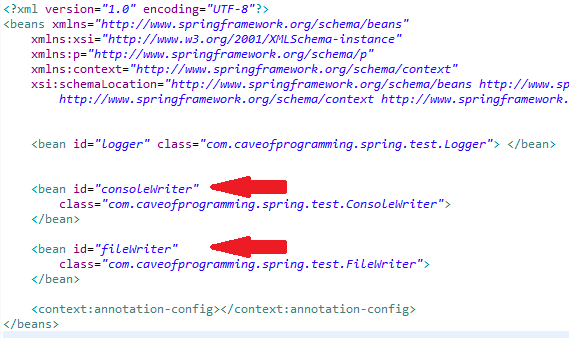
vi) Select option 'Inser <context:annotation-config> element'

***Result***: In the web.xml, the annotation context is declared and annotation can be used

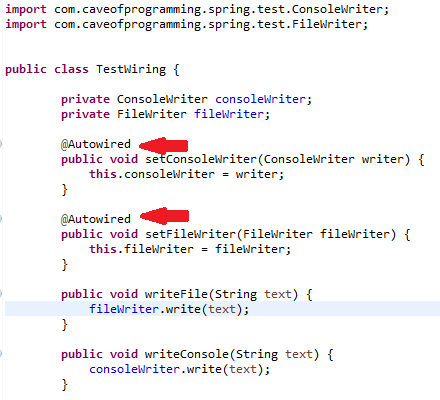
Different type of Autowiring system exists

***2A) Autowiring & application Context (XML file)***

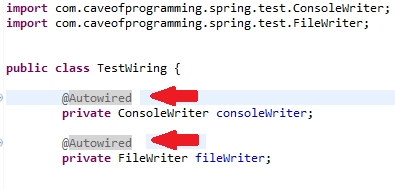
***Important***:

In the following method , you need to declare all your beans to autowire in our application context (XML file). You will get a 'No qualifying bean of type' exception otherwise ******

***Type 1) Autowired on the Set Method:***



***Type 2) Autowired on the field:***



***Important***: no setter required in the host class with this method!!!

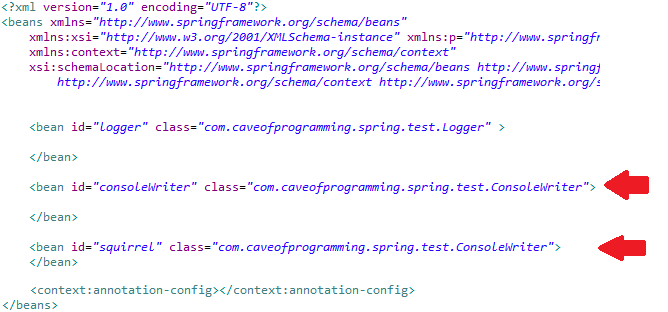
***2B) JSR-330 Way***

***Important***:

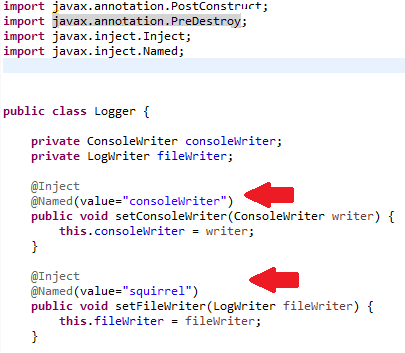
This injection is the ***Java Stack implementation of the DI***, and not the Spring way. In other word, this is the competitor version of the dependency injection.

Various Problems been reported when mixing Spring DI and the J2EE DI (java stack). It is better to choose one way of DI, and stick with it in all your web application

In the following method, you still need to declare all your beans to autowire like the previous way.



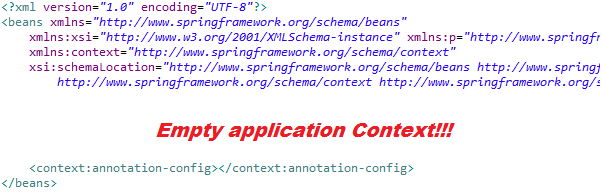
Use of the annotation @Inject, and define the bean id with the annotation @Named



***2C) Auto Discovery of Bean (Most Javaphile Way!)***

***Important***:

In the following method, ***no*** beans are declared for the autowire



i) Open the application context file

i.e: beans.xml, or applicationContext,xml

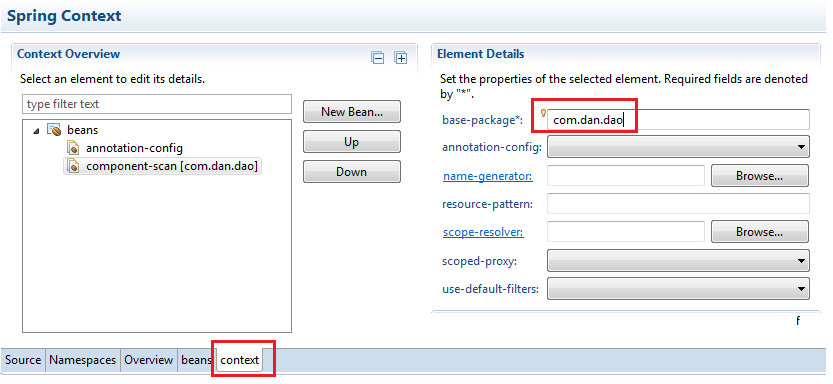
ii) Click on the tab ‘context’

iii) Right Click on 'beans'

iv) Select option '*Insert <context:component-scan> element'*

v) In the text field ***base-package***, enter the Java package were your beans are

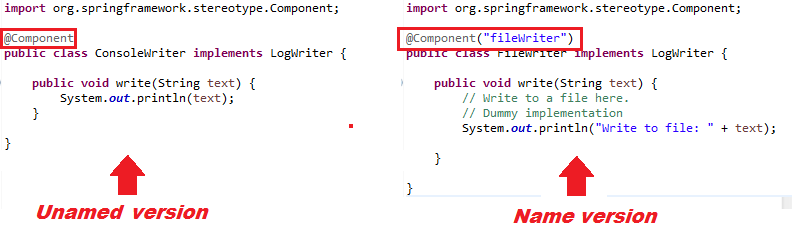
***Note***: You should group the beans to autowire under the same package



vi) Mark all the beans to be autowired with annotation @Component

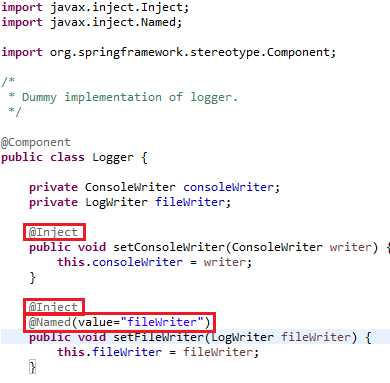
***Falculative***: You can name you Bean with a name

(the same you did in the application Context file)



vii) In the host class, annotate all fields to be autowired with @Inject

***Falculative***: If you named a bean in step vi), you can also use the ***@Named*** annotation



***Servlet Dependency Injection***

Injection of a beans into a servlet is a bit more complex than a simple Java program.

Few different ways a listed in the following section

1. Simplest Way

The simplest way relies on the listener class

org.springframework.web.context.***ContextLoaderListener***

, and also that the servlet access manually the application context with the utility class

org.springframework.web.context.support.*WebApplicationContextUtils*

*Steps*

***1 – update Web.xml***

The listener has to be registered in the web application, so add this to web.xml

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/applicationContext.xml</param-value>

</context-param>

***Note***: the applicationContext.xml file can also be in the class path

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath\*:applicationContext\*.xml</param-value>

</context-param>

***2 – Create a Spring Bean Configuration File***

And declare the beans to inject into your project

In Following example:

1. Annotation Based injection is enable (annotation-config)
2. A package scan is declared, for full Annotation injection

<?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: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/context http://www.springframework.org/schema/context/spring-context-3.2.xsd"*>

<context:annotation-config></context:annotation-config>

<context:component-scan base-package=*"com.dan.dao"*></context:component-scan>

<bean id=*"offerService"* class=*"com.dan.dao.OfferService"*></bean>

</beans>

***3 – Update Servlet’s init method***

The servlet has to do the heavily lift by locating itself the application context and access the bean manually, via the Web Application Context

**public** **class** DemoServlet **extends** HttpServlet {

**private** User danUser;

**…**

**…**

**public** **void** init(ServletConfig config) **throws** ServletException {

**super**.init(config);

WebApplicationContext context = WebApplicationContextUtils.

*getWebApplicationContext*(getServletContext());

danUser = (User)context.getBean("danUser");

}

2) Using HttpRequestHandler and Autowiring

Here, it uses a Spring equivalent of a Servlet instead of a Servlet. And this make possible the autowiring.

The technique is inspired by the following article

<https://www.codeproject.com/Tips/251636/How-to-inject-Spring-beans-into-Servlets>

Steps

1. Create an HttpRequestHandler (Child class of Java Servlet)

Make sure to

1. Mark the class as @Component, with a URL to access it
2. Add the Bean to be injected as @Autowired
3. Implements the method handleRequest

@Component("myServlet")

**public** **class** MyServlet **implements** HttpRequestHandler {

@Autowired

**private** User danUser;

@Override

**public** **void** handleRequest(HttpServletRequest arg0, HttpServletResponse arg1) **throws** ServletException, IOException {

**if** (danUser != **null**)

System.***out***.println("Found a User Bean " + danUser.toString() );

**else**

System.***out***.println("Bean is NULL!!!");

}

}

1. Update web.xml

The handler of step 1) is called by the Spring class ***HttpRequestHandlerServlet***

So this class has to be registered in the web application and an URL to access it has to be defined

<servlet>

<display-name>MyServlet</display-name>

<servlet-name>myServlet</servlet-name>

<servlet-class>

org.springframework.web.context.support.HttpRequestHandlerServlet

</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>myServlet</servlet-name>

<url-pattern>/myurl</url-pattern>

</servlet-mapping>

1. Declare the Handler as a bean

You can either update the applicationContext.xml

<bean id=*"myServlet"* class=*"com.keylesson.controller.MyServlet"*>

</bean>

Or mark the handler class with an annotation

@Component("myServlet")

**public** **class** MyServlet **implements** HttpRequestHandler {

…

***JSF & Spring***

JSF is built using Java’s CDI (the Dependency injection coming from the Java Stack), and therefore is causing problem when trying to integrate Spring’s D.I.

But the following article appears to have a handle on how to fix the situation

<http://www.baeldung.com/spring-jsf>

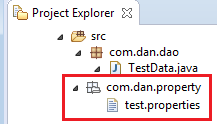
***Spring Property Support***

In an existing Spring project, having already a Spring Bean Configuration File (application Context)

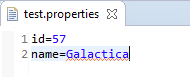
***1) Create you Property Folder and File***

i) Property file has the extension ‘.properties’

ii) A Java package must be created to contain the property file



iii) The file must contain id & Value pair like this:



***2) Activate Spring Property Support***

i) Open the application context file

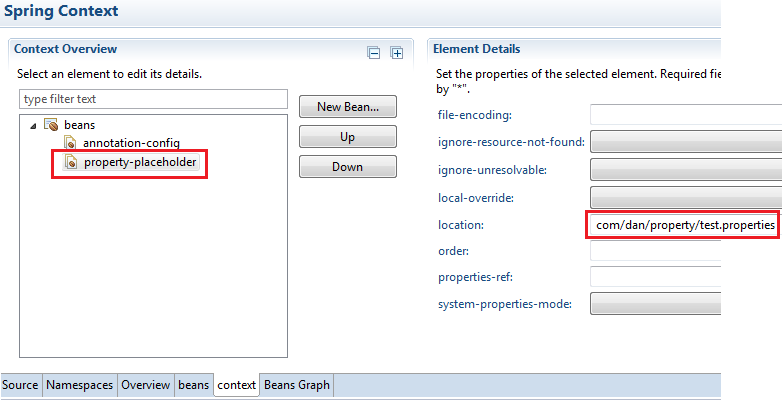
i.e: beans.xml, or applicationContext,xml

ii) Click on the tab ‘context’

iii) Right Click on 'beans'

iv) Select option '*Insert <context:property-placeholder> element'*

v) In the text field ***base-package***, enter the Java package were your beans are



Once completed, your application context file should have an extra line like the following one

(here, the property file jdbc.properties is created and used by the Spring application)

******

***3) Create Bean using the property support***

The support system appears to be use via Spring beans: the value of the propertity file are kind of injected into the Spring bean

i) Create a Bean

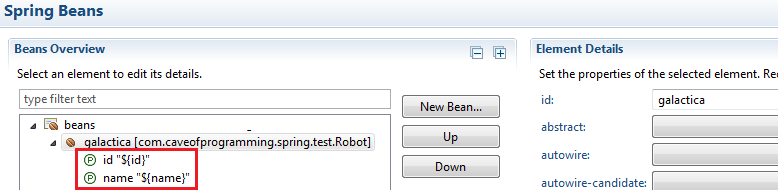
For each properties required

ii) Define properties in the beans ( insert <*property*> element)

iii) Assign an id

iv) Assign a value, using the ${ props\_fields}

🡪 props\_fields is a key defined in your property file (created in step 1))



In new window

***Convert a Web Project to an Spring-MVC***

This section starts from a Eclipse’s ***Dynamic Web Project***, and add the Spring’s ***MVC*** support.

There is a need to first include ***Maven*** support before moving to Spring (that being for the download of the Spring and other libraries)  https://mail.google.com/mail/u/0/images/cleardot.gif

https://mail.google.com/mail/u/0/images/cleardot.gif

|  |
| --- |
|  |

1) Right Click on the Web Project

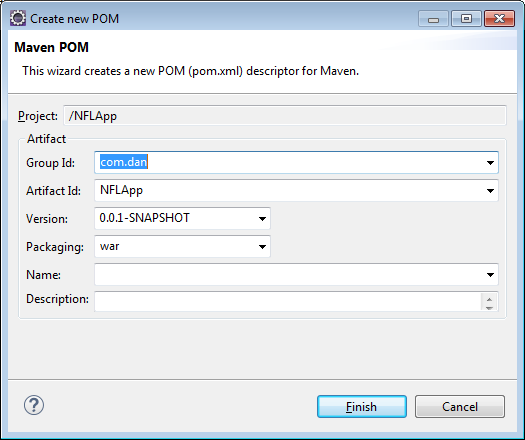
   Select Configure/Convert to Maven Project

2) From the dialog:

  i)  Enter a group ID (i.e: com.spring.dan)

  ii) Enter a Artifact ID (project name)

  iii) Click Finish



3) Locate the file pom.xml

  i) Click on tab Dependencies

  ii) add the Spring Dependencies

        Spring Core

        Spring Bean

        Spring Context

        Spring Web

        Spring Web-MVC

4) Create the Dispatcher Servlet

   i) Right Click on the Web Project

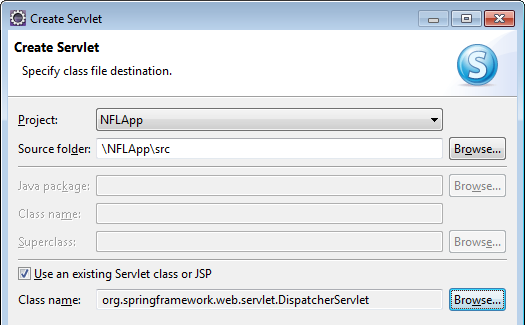
   ii) Click New/Servlet

   From the Create Servlet Dialog

   iii) Select option 'Use an existing Servlet Class or JSP'

   iv)  Click on Browse button

   v) Select DispatchServlet



Note: Eclise update the web.xml file automatically by adding the DispatchServlet configurations

5) Update the web.xml

  So that the right url can be used in the browser

  i)   Open the file web.xml

  ii)  Change the    <url-pattern>/DispatcherServlet</url-pattern> value

       to something you want to be used

       i.e <url-pattern>/MyWebService</url-pattern> value

       This will be usabe in a browser by invoking

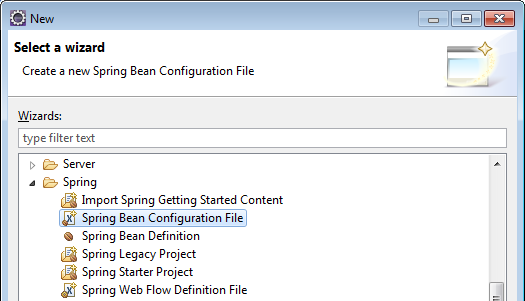
<http://localhost:8080/WebProjectName/MyWebService>

  6) Add a applicationContext file

  i) Right click on the WEB-INF folder

  ii) Select New/Others...

  iii) Select Spring / Spring Bean Configuration File

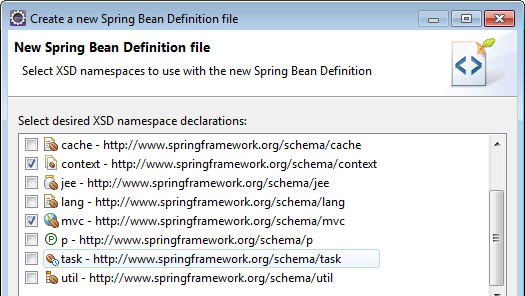


  iv) Click Next

  v)  Enter a file name : i.e applicationContext.xml

  vi) On the next tab, select the namespace

* context
* web



6) Create a Controller

i) Right click on the Web Project

ii) Create a new Java Class

   --> Name should reflect that it is a controller class

     i.e:  MeWebServiceController

iii) Add the @Controller annotation to the class

iv)  Add a service method with the annotation @RequestMapping("MyWebServiceURL")

@Controller       --> Step iii)

public class MeWebServiceController{

    @RequestMapping("MyWebService")   --> Step iv)

    public void showPage(){

     return "home";    --> That is the jsp or xhtml file name to be displayed

    }

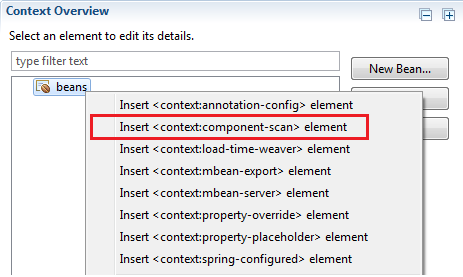
}

7) Modify DispatcherServlet-servlet.xml

i) Open the file and select tab context

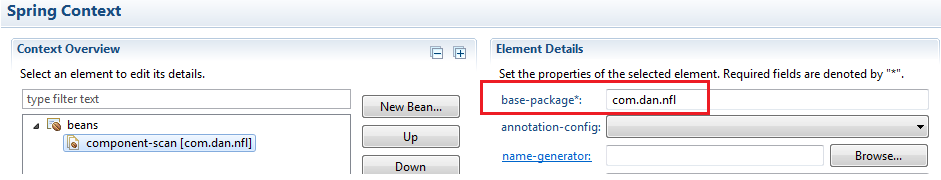
ii) Right Click on  beans

iii) Select <context:component-scan



iv) in the field base-package, enter your controller package

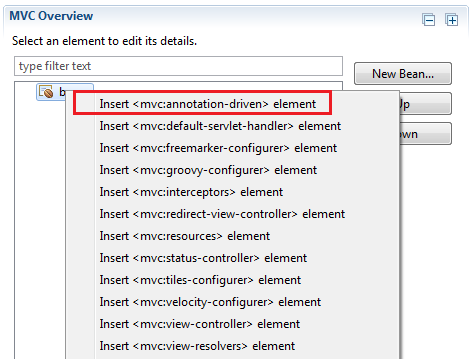
    i.e: com.spring.web



v) Select tab mvc

vi) Right Click on beans

vii) Select mvc:annotation-driven



8) Create a Resolver class, and move jsp files

 i) Create a new folder in under WEB-INF

 ii) Move your JSPs file inside

 Now create a resolver class

 iii) Open the DispatcherServlet-servlet.xml file (if not already done)

 iv) Select tab beans

 v) Click on button New Beans...

 vi) Put the name

 vii) the class type must be InternalResourceViewResolver

 vii) Click Next

 viii) Enter the 2 properties

    A) Name 'prefix', Value: /WEB-INF/jsp  --> The folder name you entered in step 8i)

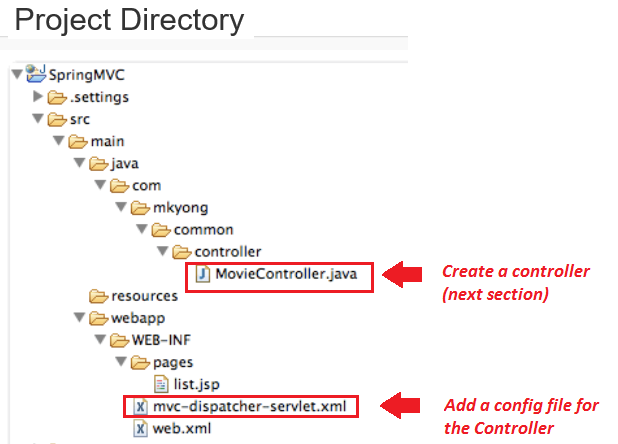
       Name 'suffix', Value: the file type of your application --> .jsp or .xhtml

        (Must have the . at the begginning)

***Simple Spring MVC REST server***

From MkYong web site: <https://www.mkyong.com/spring-mvc/spring-3-rest-hello-world-example/>

1. Your final project should have the following items



1. Create a Contoller class, with the appropriate annotation:



***Simple Spring Standalone using Maven***

1. Open a DOS command window
2. Create Maven Project

Replace the Group of Project (***groupId***) and Project Name (***artifactId***) and enter the following command

mvn archetype:generate -DgroupId=gc.ca.cbsa.logparsing -DartifactId=SPPHStats -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

1. Navigate to the folder of the Maven project (created in step 2)
2. Add the Spring Maven dependencies

<dependency>

<groupId>org.springframework</groupId>

<artifactId>***spring-core***</artifactId>

<version>4.1.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>***spring-beans***</artifactId>

<version>4.1.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>***spring-context***</artifactId>

<version>4.1.4.RELEASE</version>

</dependency>

1. Convert to Eclipse

Enter command: ***mvn eclipse:eclipse***

***JDBC Database access***

***The library commons-dbcp (from Apache) is often used with Spring JDBC.***

* ***Make sure to add the following dependency in your pom.xml***

***<dependency>***

***<groupId>commons-dbcp</groupId>***

***<artifactId>commons-dbcp</artifactId>***

***<version>1.4</version>***

***</dependency>***

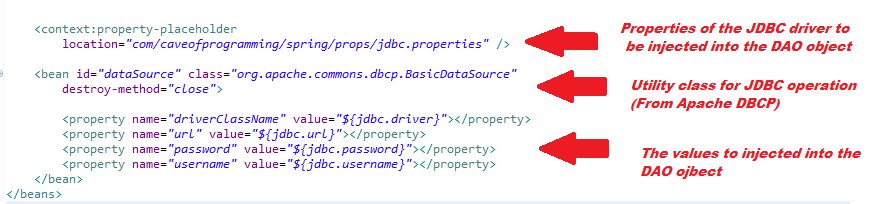
***It give access to the usefull class "org.apache.commons.dbcp.BasicDataSource"***

***Steps***

1. ***Get the common-dbcp libraries in your project***
2. ***Create a Entity Class***
3. ***Create a DAO class***
4. ***Facultative: Create a property file and activate Spring property system***

***(see earlier in this document)***

1. ***The main class calls the DAO class***

******

***Text Version (with using Property Support)***

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*

destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*/>

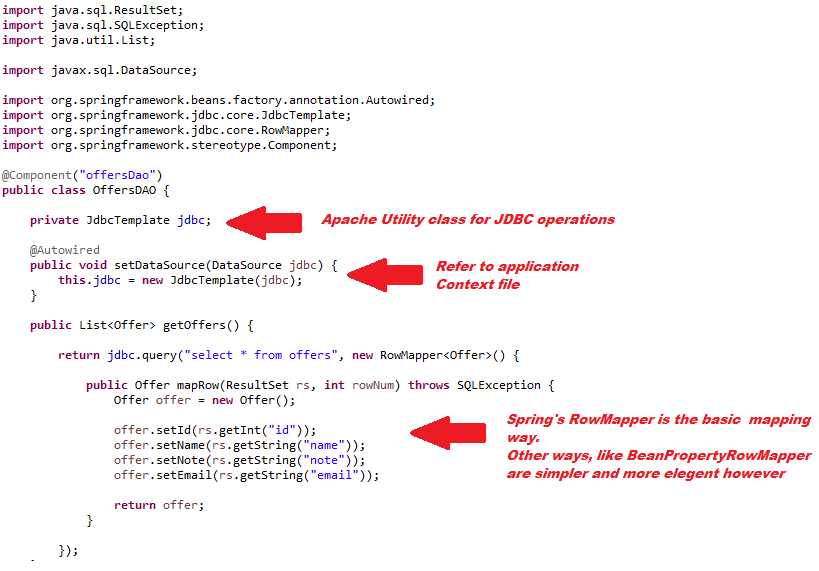
<property name=*"url"* value=*"jdbc:mysql://localhost:3306/javatest"*/>

<property name=*"password"* value=*"root"*/>

<property name=*"username"* value=*"D@njo2018"*/>

</bean>

***Example of DAO, using Apache DBCP and Spring property system (step 3 & 4)***

******

***Here another example with query and insert operation using the JdbcTemplate*** ***object***

@Component

**public** **class** UserDaoService **implements** UserDaoServiceInterface{

**private** JdbcTemplate jdbcTemplate;

@Autowired

**public** UserDaoService(JdbcTemplate jdbcTemplate) {

**this**.jdbcTemplate = jdbcTemplate;

}

**public** List<User> findAll() {

**return** jdbcTemplate.query( "SELECT \* FROM user",

**new** RowMapper() {

**public** Object mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

User user = **new** User();

user.setId(rs.getInt("id")); user.setName(rs.getString("name"));

user.setBirthDate(rs.getDate("birthDate"));

**return** user;

}

});

}

**public** User findOne(**int** id) {

**return** (User)jdbcTemplate.queryForObject( "SELECT \* FROM user WHERE id=?",

**new** Object[] { id },

**new** RowMapper() {

**public** Object mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

User user = **new** User();

user.setId(rs.getInt("id"));

user.setName(rs.getString("name"));

user.setBirthDate(rs.getDate("birthDate"));

**return** user;

}

});

}

**public** User save(User user) {

jdbcTemplate.update("INSERT INTO user (name, birthDate) VALUES (?, ?)",

**new** Object[] { user.getName(), user.getBirthDate() });

**return** user;

}

***Final Step, call the DAO from the main class (step 5)***

******

***Alternative to RowMapper 🡪 BeanPropertyRowMapper***

With this mapper, there is no need to manually correlate the class fields with its respective DB Table fields.

******

***Use NamedParameterJdbcTemplate***

***To be able to specify parameter in your query use NamedParameterJdbcTemplate instead of JdbcTemplate***

@Component("personDao")

**public** **class** PersonDAO {

**private** NamedParameterJdbcTemplate jdbc;

@Autowired

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

**this**.jdbc = **new** NamedParameterJdbcTemplate(jdbc);

}

**public** Person getPerson(**int** id) {

MapSqlParameterSource params = **new** MapSqlParameterSource();

params.addValue("***id***", id);

// Will use the syntax *:somefield* in the SQL

**return** jdbc.queryForObject(“select \* from person where id=***:id***",

params,

**new** RowMapper<Person>() {

**public** Person mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

Person person = **new** Person();

person.setId(rs.getInt("id"));

person.setFirstName(rs.getString("firstName"));

person.setLastName(rs.getString("lastName"));

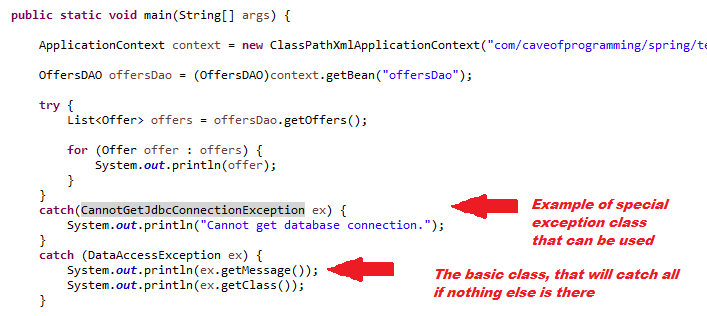
**return** person;

}

***Catch JDBC Exceptions***

Spring has a hierarchy of class to handle the JDBC exceptions. The mother class, and the most generic and easy to use is

***org.springframework.dao.DataAccessException***

******

Here a text version for copy-paste purpose…

**try** {

List<Offer> offers = offersDao.getOffers();

**for**(Offer next: offers)

System.***out***.println(next.toString());

} **catch**(CannotGetJdbcConnectionException cgjce){

cgjce.printStackTrace();

} **catch**(DataAccessException dae){

dae.printStackTrace();

}

***BeanPropertySqlParameterSource***

Passing parameters to an SQL can be easier by using the BeanPropertySqlParameterSource class.

The class maps fields of the DB table with their equivalent In the Java Class

Note: Like before the :someName syntax has to be used in the SQL statement

// Here, the values of the fields of Java Class's Offer, will be

//made accessible in the SQL, by using the :someName syntax

**public** **boolean** create(Offer offer) {

BeanPropertySqlParameterSource params =

**New** BeanPropertySqlParameterSource(offer);

**return** jdbc.update("insert into offers (name, text, email) " +

"values (:name, :text, :email)", params) == 1;

}

There exists also a multiple object, or array of object version of example above

(Seen in lesson 49)

//Class SqlParameterSourceUtils handles multi entities transaction

//Same as BeanPropertySqlParameterSource example, but here,

//multiple objects can be mapped and stored at the same time

**public** **int**[] create(List<Offer> offers) {

SqlParameterSource[] params =

SqlParameterSourceUtils.*createBatch*(offers.toArray());

**return** jdbc.batchUpdate("insert into offers (name, text, email) “

+ “values (:name, :text, :email)", params);

}

***Transaction***

You can rollback SQL operations, if many operation is required, but one of those fails (All have to succeed or All have to fail kind of situation)

The spring class that handles this is:

***org.springframework.jdbc.datasource.DataSourceTransactionManager***

***Steps:***

In your application context

1. ***Add DataSourceTransactionManager +***

***Transaction annotation***

Copypast the following section

<bean id="transactionManager"

class="org.springframework.jdbc.datasource.***DataSourceTransactionManager***">

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

</bean>

<tx:annotation-driven />

***In the header of the file, add the following context (IN RED) among the existing one (Core, Context, etc)***

***<beans xmlns=***[***http://www.springframework.org/schema/beans***](http://www.springframework.org/schema/beans)

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

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

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

***xmlns:tx=***[***http://www.springframework.org/schema/tx***](http://www.springframework.org/schema/tx)

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

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

[***http://www.springframework.org/schema/tx***](http://www.springframework.org/schema/tx) ***http://www.springframework.org/schema/tx/spring-tx-3.2.xsd***

[***http://www.springframework.org/schema/context***](http://www.springframework.org/schema/context) ***http://www.springframework.org/schema/context/spring-context-3.2.xsd">***

1. ***Add Transactional Annotation***

***Mark the method of the class where a rollback has to happen if there is an SQL operation that failed: add the annotation*** @Transactional ***To the method***

//The @Transactional annotation marks the method as being handled by

//Spring's DataSourceTransactionManager object

@Transactional

**public** **int**[] create(List<Offer> offers) {

SqlParameterSource[] params = SqlParameterSourceUtils.*createBatch*(offers.toArray());

**return** jdbc.batchUpdate("insert into offers (name, text, email) “ +

“values (:name, :text, :email)", params);

}

***JDBC Access from a J2EE application***

There are 2 ways to connect a J2EE app to a JDBC:

1. Simple J2EE Naming/Lookup Context (Contex & InitialContext())
2. Spring bean and injection

Both way have the common path however

***Common Steps***

1 – Server Context

In both way, Add the context information in the web server

For ***Tomcat***:

The file ***context.xml*** need to have this line

<Resource name**=**"jdbc/TestDB" auth**=**"Container" type**=**"javax.sql.DataSource"

maxTotal**=**"100" maxIdle**=**"30" maxWaitMillis**=**"10000"

username**=**"root" password**=**"D@njo2018"

driverClassName**=**"com.mysql.jdbc.Driver"

url**=**"jdbc:mysql://localhost:3306/javatest"/>

***Note***: replace the username, password with the correct values, and make sure the DB URL

points the correct Database name

2 – Update web.xml

The J2EE look or the Spring method need a reference of the context in ***web.xml***

<resource-ref>

<description>DB Connection</description>

<res-ref-name>jdbc/TestDB</res-ref-name>

<res-type>javax.sql.DataSource</res-type>

<res-auth>Container</res-auth>

</resource-ref>

*J2EE Naming/Lookup Context Steps*

3 – Lookup for Datasource

In this method, the servlet of code has to lookup to the DB Datasource itselve

**try** {

//First action is to lookup for the JDBC datasource defined in the

//web.xml and the web server context (context.xml for Tomcat server)

Context initContext = **new** InitialContext();

Context envContext = (Context) initContext.lookup("java:comp/env");

DataSource ds = (DataSource) envContext.lookup("jdbc/TestDB");

Connection conn = ds.getConnection();

Statement statement = conn.createStatement();

String sql = "select id, foo,bar from testdata";

ResultSet rs = statement.executeQuery(sql);

**int** count = 1;

**while** (rs.next()) {

String message = String.*format*("id #%d: %-15s %s",

rs.getInt("id"),

rs.getString("foo"), rs.getInt("bar"));

System.***out***.println(message);

}

} **catch** (NamingException e) {

e.printStackTrace();

}**catch** (SQLException e) {

e.printStackTrace();

}

***Spring bean and injection***

Similar to the J2EE technique above, but here will define the DataSource as a Spring Bean

Note: Very same principle as the first Spring JDBC example of this section of the document

3 – Make sure you have a Spring Bean Definition

4 – Define a Data Source Bean

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*

destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/javatest"*/>

<property name=*"password"* value=*"root"*/>

<property name=*"username"* value=*"D@njo2018"*/>

</bean>

5 – Add the Bean in your code (Servlet or Handler)

The code requiring JDBC access has to add a bean: Here @Resource been used

@Controller

@RequestMapping("/kfc")

**public** **class** HelloController {

//Here is the Data Source Bean defined in the Context (previous step)

@Resource(name = "dataSource")

**private** DataSource dataSource;

But @Autowired could have been used

//Make sure that the bean id of the context file matches

//the name variable of the field to be wired by Spring

@Autowired

**private** DataSource dataSource;

6 – Access the Data Source (JDBC coding)

**try** {

Connection conn = dataSource.getConnection();

Statement statement = conn.createStatement();

String sql = "select id, foo,bar from testdata";

ResultSet rs = statement.executeQuery(sql);

**while** (rs.next()) {

rs.getInt("id"),rs.getString("foo"), rs.getInt("bar"));

}

} **catch** (SQLException e) {

}

***Property Support***

Spring has a property file system which can inject value into Beans

***1 - Define a property file***

A file with the extension .properties has to existing among

1. Your Classes (like below)
2. WEB-INF folder

******

Here an example of a JDBC properties file

jdbc.username = root

jdbc.password = D@njo2018

jdbc.driver = com.mysql.jdbc.Driver

jdbc.url = jdbc:mysql://localhost:3306/springtutorial

***Note: A good practice is to put a prefix in front of every value defined in a the property file. Use the file name as prefix (i.e: jdbc.XYZ..)***

***2 – Declare the Property File***

The file has to be declared into Spring’s application Context file, using the syntax:

<context:property-placeholder location=*"classpath:com/caveofprogramming/spring/web/config/jdbc.properties,*

*WEB-INF/mq.properties"* />

***Note 1*** : We have a double property file example here. You can have 1 or more than 2.

**Note 2**: ***jdbc.properties*** is placed among the Java Classes (see picture above), so relies on classpath key

***mq.properties*** is placed in the WEB-INF folder, so no classpath key required here.

***3A – Inject value into Beans***

Next step is to inject the value defined in property file into Beans, using the Spring’s Expression language.

Spring’s @Autowired annotation on the Setter method

@Component("offersDao")

**public** **class** OffersDAO {

String username;

String password;

@Autowired

**public** **void** setUsername(@Value("${jdbc.username}") String username) {

**this**.username = username;

}

@Autowired

**public** **void** setPassword(@Value("${jdbc.password}") String password) {

**this**.password = password;

}

3B – Nested Injection in Application Context file

Another option is to use the property’s values directly into the Application Context file. Spring’s Expression language is used inside the Bean’s definition themselves.

Here an example of a JDBC Data source Bean definition

<context:property-placeholder

location=*"classpath:com/caveofprogramming/spring/web/config/jdbc.properties"* />

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*

destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"${jdbc.driver}"* />

<property name=*"url"* value=*"${jdbc.url}"* />

<property name=*"username"* value=*"${jdbc.username}"* />

<property name=*"password"* value=*"${jdbc.password}"* />

</bean>

***Note*:** This example depends on the ***jdbc.properties*** file show in section 1