**Spring IoC**

In Spring, the Inversion of Control (IoC) principle is implemented using the Dependency Injection (DI) design pattern.

The Spring framework provides prowerful container to manage the components. The container is based on the Inversion of Control (IoC) principle and can be implemented by using the Dependency Injection (DI) design pattern. Here the component only needs to choose a way to accept the resources and the container will deliver the resource to the components.

QuizMasterService.java

02.-----------------------

03.package com.vaannila;

04.

05.public class QuizMasterService {

06.

07.QuizMaster quizMaster;

08.

09.public void setQuizMaster(QuizMaster quizMaster) {

10.this.quizMaster = quizMaster;

11.}

12.

13.public void askQuestion()

14.{

15.System.out.println(quizMaster.popQuestion());

16.}

17.}

The value for the QuizMaster will be set using the setQuizMaster() method. The QuizMaster object is never instantiated in the QuizMasterService class, but still we access it. Usually this will throw a NullPointerException, but here the container will instantiate the object for us, so it works fine.

**SPRING MVC**

**Understanding IoC the Spring way**

Eachbean element describes an object that will be created and given an id. Each propertyelement describes a setter method on the object and the value that should be given to it. These setters are called for you by the Spring application container.

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

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

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

**xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">**

**<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource" id="ds">**

**<property name="driverClassName" value="org.h2.Driver"/>**

**<property name="url" value="jdbc:h2:~/cs"/>**

**<property name="username" value="sa"/>**

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

**</bean>**

**<bean class="org.springframework.jdbc.core.JdbcTemplate" id="jdbcTemplate">**

**<constructor-arg ref="ds"/>**

**</bean>**

**<bean class="org.springframework.samples.DatabaseCustomerService" id="databaseCustomerService">**

**<property name="jdbcTemplate" ref="jdbcTemplate"/>**

**</bean>**

**<bean class="org.springframework.samples.CustomerClient" id="client">**

**<property name="customerService" ref="databaseCustomerService"/>**

**</bean>**

**</beans>**

ref tells the Spring framework that you want to pass a reference to another bean configured in the same container.

This simple setup affords us a lot of indirection and flexibility. Now that the object creation and construction's been moved to the Spring configuration, we can hide really complex setup in the Spring configuration and keep our code none the wiser. One common way of hiding complex construction logic is through the factory pattern. The factory pattern is particularly useful if you are constructing a lot of objects that need to work together, or if you want to take lots of different factors into consideration when creating an object. Essentially, what you're doing is providing a more powerful way to describe object creation than any one class' constructor can naturally do. Spring supports this pattern explicitly. If a bean is configured that implements the org.springframework.beans.factory.FactoryBeaninterface, the getObject() method on the interface will be called and the result will be the object that's available in the Spring context. This practice is used extensively in the Spring framework itself to provide convenient ways to construct complex object graphs in a reusable way.

**<bean id="ds" class="org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseFactoryBean">**

**<property name="databaseType" value="H2"/>**

**<property name="databasePopulator">**

**<bean class="org.springframework.jdbc.datasource.init.ResourceDatabasePopulator">**

**<property name="scripts" value="setup.sql"/>**

**</bean>**

**</property>**

**</bean>**

Component scanning scans for beans with certain annotations on them and automatically registers them. Similarly, annotations discovered on the class itself will be processed. Here's the revised XML file with the appropriate XML namespace included.

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

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

xsi:schemaLocation="... http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">

<context:component-scan base-package="org.springframework.samples"/>

<!-- ... same as before ... -->

</beans>

All beans that we annotate in the org.springframework.samples package with the Spring framework's @Component annotation will be picked up and registered as beans with the context, exactly as if we had configured them using bean elements in XML. We've annotated the DatabaseCustomerService and CustomerClient classes with@Component, which lets us remove the equivalent bean elements for those beans from the XML configuration. Component scanning is very convenient because Spring does much of the heavy lifting, though it decentralizes configuration.

We know that this bean has a dependency on a JdbcTemplate. The JdbcTemplateis configured in the context already. As there is only one configured, we can simply annotate the setter on the class with @Autowired, which tells Spring to resolve the dependency by type and inject it. If there are multiple instances configured in the context then an error would be thrown in this case.

package org.springframework.samples;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jdbc.core.RowMapper;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.stereotype.Component;

import java.sql.ResultSet;

import java.sql.SQLException;

@Component

public class DatabaseCustomerService implements CustomerService {

private JdbcTemplate jdbcTemplate;

private RowMapper<Customer> customerRowMapper = new CustomerRowMapper();

public Customer getCustomerById(long id) {

return jdbcTemplate.queryForObject(

"select \* from CUSTOMERS where ID = ?", this.customerRowMapper, id);

}

@Autowired

public void setJdbcTemplate( JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

class CustomerRowMapper implements RowMapper<Customer> {

public Customer mapRow(ResultSet resultSet, int i) throws SQLException {

String fn = resultSet.getString("FIRST\_NAME");

String ln = resultSet.getString("LAST\_NAME");

String email = resultSet.getString("EMAIL");

long id = resultSet.getInt("ID");

return new Customer(id, fn, ln, email);

}

}

}

The last class is the client that makes use of the CustomerService instance. We register it just as before, using the @Component annotation. It needs a reference to theCustomerService instance just as the DatabaseCustomerService instance needed a reference to the JdbcTemplate. So, we use our old friend @Autowired

**package org.springframework.samples;**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.context.support.GenericXmlApplicationContext;**

**import org.springframework.stereotype.Component;**

**@Component**

**public class CustomerClient {**

**private CustomerService customerService ;**

**@Autowired**

**public void setCustomerService(CustomerService customerService) {**

**this.customerService = customerService;**

**}**

**public void printCustomerInformation ( long customerId ) {**

**Customer customer = this.customerService.getCustomerById( customerId );**

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

**}**

**}**

Our revised XML file is much svelter for our troubles:

 It would be nice if we could use component scanning for the JdbcTemplate instance as well. However, component scanning only works for beans annotated with @Component. As we can't add the @Component annotation to third party classes for which we may not have the source code, component scanning is not an option for the JdbcTemplate instance.

Spring offers the Java configuration support to let you describe and configure beans using Java directly. The Java configuration option offers the best of both worlds: it lets you configure any class, even classes you don't have the source code for (as with the XML Configuration option) and it's still Java-centric, so benefits from all type safety of the Java language (and the refactoring tools available in the Java IDEs).

Java Configuration processes a bean registered with the context and it looks for methods annotated with @Bean and invokes them. The result of method invocation is registered with the application context as a bean, exactly as if you had configured the object using XML. The type of the bean is the type of the returned object, and the id is taken from the method name. Because the configuration is provided by the Java code in the method, you can do any manner of setup, much like a FactoryBean lets you do. People often choose the Java configuration option because it lets you keep your bean configuration in one or two well known, central classes. XML configuration and Java configuration both provide a way to centrally describe your application.

A configuration class is a Spring bean, just like any other. All the rules that apply to regular Spring beans apply to a configuration bean, with the exception of methods annotated with@Bean. Spring will pick up your configuration class with component scanning. If you want to use other beans in your configuration class (for example, the javax.sql.DataSourceinstance that we previously configured using the XML namespace) then you have all the normal options available to you to get them, including @Autowired.. Let's look at a configuration class for our example.

**package org.springframework.samples;**

**import org.springframework.beans.factory.annotation.Autowired;**

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

**import org.springframework.stereotype.Component;**

**import org.springframework.jdbc.core.JdbcTemplate;**

**import javax.sql.DataSource;**

**@Configuration**

**public class CustomerConfiguration {**

**@Autowired private DataSource dataSource;**

**@Bean**

**public JdbcTemplate jdbcTemplate() {**

**return new JdbcTemplate(this.dataSource);**

**}**

**}**

This class uses @Autowired to obtain a reference to the embedded datasource. This is similar to the way it's used in previous examples, except that here we're using the annotation on a private field variable, not a setter method. The Spring framework will work with annotations on the constructors of a class, on field variables, or on setter methods. Finally, we have a method annotated with @Bean. This method provides the definition of theJdbcTemplate instance and means we can remove the XML configuration from our file. We aren't doing so here, but you can define multiple @Bean definition methods in a class and they can reference each other by simply invoking each other. If one method annotated with @Bean invokes another, the return value will either be a newly created object or – if the bean's been created already – the bean that's already registered with the context. In the class above, we also have an @Configuration annotation on the class. This annotation tells Spring to treat this class as a special type of component specifically for configuration. In essence, this bean benefits from all the same services as any bean registered in a Spring context, and it has extra services applied to it to enable Java configuration. To use Java Configuration, ensure that you have the CGLIB library on your classpath.

Revised XML Looks like this:

**<?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">**

**<context:component-scan base-package="org.springframework.samples"/>**

**<jdbc:embedded-database id="ds" type="H2">**

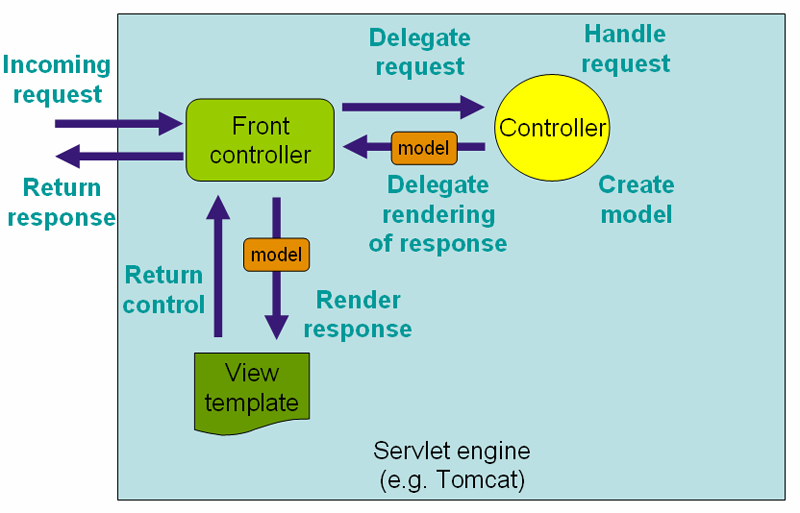
**<jdbc:script location="classpath:setup.sql"/>**

**</jdbc:embedded-database>**

**</beans>**

**Spring MVC**

Spring MVC includes most of the same basic concepts as other so-called web MVC frameworks. Incoming requests enter the framework via a Front Controller. In the case of Spring MVC, this is an actual Java Servlet called DispatcherServlet. Think ofDispatcherServlet as the gatekeeper. It doesn't perform any real web or business logic, but rather delegates to POJOs called Controllers where the real work is done (either in whole or via the back-end). When the work has been done, it's the responsibility of Views to produce the output in the proper format (whether that's a JSP page, Velocity template, or JSON response). Strategies are used to decide which Controller (and which method(s) inside that Controller) handles the request, and which View renders the response. The Spring container is used to wire together all these pieces. It all looks something like this



### Bootstrapping the DispatcherServlet and Spring Container

As mentioned, all incoming requests flow through a DispatcherServlet. Like any other Servlet in a Java EE application, we tell the Java EE container to load this Servlet at web app startup time via an in the web app's WEB-INF/web.xml. TheDispatcherServlet is also responsible for loading a SpringApplicationContext that is used to perform wiring and dependency injection of managed component. On this basis, we specify some init parameters to the Servlet which configure the Application Context. Let's look at the config in web.xml:

**See web.xml in SpringWebApp project.**

A number of things are being done here:

* We register the DispatcherServlet as as a Servlet called appServlet
* We map this Servlet to handle incoming requests (relative to the app path) starting with "/"
* We use the ContextConfigLocation init parameter to customize the location for the base configuration XML file for the Spring Application Context that is loaded by the DispatcherServlet, instead of relying on the default location of *<servletname>-context.xml*).

*Wait, What if Somebody Doesn't Want to Configure Spring via XML?*

The default type of Application Context loaded by the DispatcheServlet expects to load at least on XML file with Spring bean definitions. As you'll see, we'll also enable Spring to load Java-based config, alongside the XML

### The Controller

Now let's create a minimal controller:

NOTE: See FirstController.java in SpringWebApp Project.

Let's walk through the key aspects of this class:

* The class has been annotated with the @Controller annotation, indicating that this is a Spring MVC Controller capable of handling web requests. Because @Controller is a specialization of Spring's @Component *Stereotype* annotation, the class will automatically be detected by the Spring container as part of the container's component scanning process, creating a bean definition and allowing instances to be dependency injected like any other Spring-managed component.
* The home method has been annotated with a @RequestMapping annotation, specifying that this method should handle web requests to the path "/", i.e. the *home* path for the application.
* The home method simply logs a message to system out, and then returns WEB-INF/views/home.jsp, indicating the view which should handle the response, in this case a JSP page.

Finally, as previously mentioned, we need to create a minimal Spring Application Context definition file.

NOTE: Please See spring-context.xml in SpringWebApp Project.

Let's examine the contents of this file:

* You'll note that a few different Spring XML namespaces are being used: *context*, *mvc*, and the default *beans*
* The <context:component-scan> declaration ensures the Spring container does component scanning, so that any code annotated with @Component subtypes such as@Controller is automatically discovered. You'll note that for efficiency, we limit (toxyz.sample.baremvc in this case) what part of the package space Spring should scan in the classpath
* The <mvc:annotation-driven> declaration sets up Spring MVC's support for routing requests to @Controllers, as well as how some things like conversion, formatting and validation are handled (with some sensible defaults based on what (libraries) is present in your classpath, and the ability to override if needed)

As trivial as it is, running this application involves all the major pieces of a working Spring MVC application. Let's walk through the major sequences and component interactions:

* When the web app starts up, the DispatcherServlet is loaded and initialized because of the entry in web.xml.
* The DispatcherServlet loads an annotation-based Application Context, which has been configured to scan for annotated components via a regular expression specifying the base package(s).
* Annotated components such as the HomeController are detected by the container.
* The HTTP request to http://localhost:8080/baremvc hits the servlet engine and is routed to our (*baremvc*) webapp.
* The implicit "/" path at the end of the URL matches the regex that has been registered for the DispatcherServlet, and the request is routed to it
* The DispatcherServlet needs to decide what to do with the request. It uses a *strategy*called a HandlerAdapter to decide where to route the request. The specific HandlerAdapter type (or types, since they can be chained) to be used can be customized, but by default, an annotation-based strategy is used, which routes requests appropriately to specific methods in classes annotated as @Controller, based on matching criteria in @RequestMapping annotations found in those classes. In this case, the regex on the *home* method is matched, and it's called to handle the request.
* The *home* method does its work, in this case just printing something to system out. It then returns a string that's a hint (in this case, a very explicit one, WEB-INF/views/home.jsp) to help chose the View to render the response.
* The DispatcherServlet again relies on a strategy, called a ViewResolver to decide which View is responsible for rendering the response. This can be configured as needed for the application (in a simple or chained fashion), but by default, anInternalResourceViewResolver is used. This is a very simple view resolver that produces a JstlView which simply delegates to the Servlet engine's internalRequestDispatcher to render, and is thus suitable for use with JSP pages or HTML pages.
* The Servlet engine renders the response via the specified JSP

### Taking It to the Next Level

Essentially, the controller code ideally needs to be something like this variant, where a purely logical view name (whether simple or composite) is returned:

|  |
| --- |
| //... |
| @Controller | |

|  |  |
| --- | --- |
| public class HomeController { | |
|  |

|  |  |
| --- | --- |
| @RequestMapping(value = "/") | |
| public String home() { |

|  |  |
| --- | --- |
| System.out.println("HomeController: Passing through..."); | |
| return "home"; |

|  |  |
| --- | --- |
| } | |
| } |
| Spring MVC's ViewResolver Strategy is actually the mechanism meant to be used to achieve this looser coupling between the controller and the view. As already mentioned, in the absence of the application configuring a specific ViewResolver, Spring MVC sets up a default minimally configured InternalResourceViewResolver, a very simple view resolver that produces a JstlView. There are potentially other view resolvers we could use, but to get a better level of decoupling, all we actually need to do is set up our own instance of InternalResourceViewResolver with slightly tweaked configuration.InternalResourceViewResolver employs a very simple strategy; it simply takes the view name returned by the controller, and prepends it with an optional prefix (empty by default), and appends it with an optional suffix (empty by default), then feeds that resultant path to a JstlView it creates. The JstlView then delegates to the Servlet engine'sRequestDispatcher to do the real work, i.e. rendering the template. Therefore, to allow the controller to return logical view names like home instead of specific view template paths like WEB-INF/views/home.jsp, we simply need to configure this view resolver with the prefix WEB-INF/views and the suffix .jsp, so that it prepends and appends these, respectively, to the logical name returned by the controller.  One easy way to configure the view resolver instance is to introduce the use of Spring's Java-based container configuration, with the resolver as a bean definition:  We are already doing component scanning, therefore since @Cofiguration is itself an@Component, this new configuration definition with the (resolver) bean inside it is automatically picked up by the Spring container. Then Spring MVC scans all beans and finds the resolver.  NOTE: This can also be configured in the XML. Handling User Input NOTE: Please see SecondController.java in SpringWebApp project.  Key elements in the new code:   * We're using another @RequestMapping annotation to make requests ending in with the path /compare to the new *compare* method * We are expecting the caller to pass us the two String input parameters as part of the GET request, so we grab them via the @RequestParam annotation. Note that we are relying on the default handling for this annotation, which assumes that these params are required. The client will receive an HTTP 400 error if they are missing. Note also that this is just one way of passing in parameters to a Spring MVC application. For example, it's easy to grab parameters that are embedded as part of the request URL path itself, for a more REST-style approach * We use our Comparator instance to compare the two strings * We stuff the comparison result into the *Model* object under the key result, so that it can be accessed by the View. Think of a Model as a glorified hashmap, in simplest terms.   Finally, we need to supply the controller with a Comparator instance to use. We have annotated the comparator field in the controller with Spring's @Autowired annotation (which will be detected automatically after the controler is detected) and instructs the Spring container to inject a Comparator into that field. Therefore, we need to ensure the container has one available. For this purpose, a minimal Comparator implementation has been created, which simply does a case insensitive comparison. For simplicity's sake, this class has itself been annotated with Spring's @Service Stereotype annotation, a type of@Component and thus will automatically be detected by the Spring container as part of the container's component scanning process, and injected into the controller |