**Q. Inversion of Control and Dependency Injection design pattern with real world ?**

**Ans**. Inversion of Control and Dependency Injection is a **core design pattern of Spring framework** As the name suggest **Inversion of control** pattern **Inverts responsibility of managing the life cycle of the object** e.g. creating an object, setting their dependency etc from application to a framework, which makes writing Java application even more easy. The programmer often confused between **IOC and DI**, well both words used interchangeably in Java world but **Inversion of Control is a more general concept** and **Dependency Injection is a concrete design pattern.**  
  
Spring framework provides **two implementations of IOC container** in the form of **Application Context** and **BeanFactory** which manages the life-cycle of bean used by Java application.   
  
Also there are **multiple way to inject dependency in spring** e.g**. Setter Injection or Constructor Injection,** **field Injection** which uses setter method and constructor for injecting dependency,

## Inversion of Control and Dependency Injection design pattern

Look at below implementation of an AuditService whose job is to store every audit messages into database. This is one of the simplest kind of auditing Service required in Enterprise Java application.

/\*\*  
 \* Java Service class which provides auditing functionality by storing  
 \* auditing message into persistent.  
 \*/  
**public** **class** **AuditServiceImpl** **implements** AuditService{  
  
    **private** AuditDAO auditDao = **new** AuditDAO();  
       
    @**Override**  
    **public** **boolean** audit (**String** message) {  
       **return** auditDao.store(message);  
    }  
   
}

In first glance this implementation looks perfect but there are **three major problem** with this implementation:

1) Every AuditServiceImpl has its **own copy of AuditDAO** which is an **expensive object as it wraps a database connection** with in. It make no sense to create separate instances of AuditDAO, if you can share one between multiple AuditService.

2) AuditServiceImpl is **closely coupled with AuditDAO** as its creating instance of AuditDAO using new() operator. If you change the constructor of AuditDAO this code will be broken. Because of this coupling its difficult to replace AuditDAO with better implementation.

3) There is **no easy way to test audit()** method which is **dependent on auditDAO**. Since you can not mock AuditDAO you have to rely on actual implementation and if AuditDAO is an environmental dependent object which it is as it connect to different database on a different environment, your Junit test case may pass in some environment and may fail in other environments.  
**What is Dependency Injection concept?**

Dependency Injection is a design pattern on which dependency of the object (in this case AuditDAO is a dependency for AuditServiceImpl Object) is injected by the framework rather than created by Object itself. **Dependency Injection reduces coupling between multiple objects as its dynamically injected by the framework**. One of the **implementations of DI is Inversion of Control (IOC) on which framework like Spring controls object’s dependency.** There are mainly two types of Dependency Injection: Constructor Injection and Setter Injection.

In Constructor Injection, the dependency of Object is injected using constructor, while in Setter Injection, Dependency is provided by the setter method.   
Both have there pros and cons. **Constructor DI a**llows the object to be created in complete state and follows the principle of the **fully functional object** while **Setter DI** allows object to be created without its dependency. which may result in an **incomplete object** if dependency is not available.

**when do you use Setter injection and Constructor Injection in Spring".**

Another benefit of **Setter Dependency Injection is readability** since Spring is configured with **xml configuration file** and setter injection is provided with bean property which is much easier to read and understand than constructor injection which doesn't state the property.

### AuditServiceImpl using Dependency Injection

Now we will see How Dependency Injection solves all three problems we have listed with the above implementation of AuditService. here is a new implementation of AuditService with setter dependency injection.

**public** **class** AuditServiceImpl **implements** AuditService{  
  
    **private** AuditDAO auditDao;  
  
    **public void setAuditDao(AuditDAO AuditDao) {  
        this.AuditDao = AuditDao;  
    }**   
    @**Override**  
    **public** **boolean** audit (**String** message) {  
       **return** auditDao.store(message);  
    }  
   
}

**1.** Since AuditDAO is injected here its possible to **share single AuditDAO  between multiple AuditService.**

**2.** Since AuditServiceImpl is not creating an instance of **AuditDAO its no more coupled** with AuditDAO and **works with any implementation of AuditDAO,** thanks to another famous object-oriented design principle “program for interface than implementation".

3. Because **AuditDAO is injected by DI at runtime it's easy to test audit()** method by providing a mock AuditDAO class. This not only makes testing easier but also independent of environmental changes as you are not using the actual implementation of AuditService.

**1) Reduce coupling**

both constructor and setter dependency injection **reduce coupling.** like in the above example coupling between AuditService and AuditDAO is reduced by using Dependency Injection.

**2) Improves testability**

Dependency Injection **allows to replace the actual object with a mock object** which improves testability by writing simple JUnit tests which uses a mock object.

**3) Flexibility**

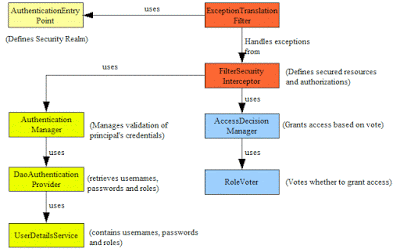
This is another advantage which comes as a side benefit of reduced coupling, because of DI you can replace non-performance implementation with a better one later.

**Q. Does Spring singleton beans are thread-safe?  
Ans**. **No**, Spring singleton beans are not thread-safe. Singleton doesn't mean bean would be thread-safe.

**Q.) Can we use more than one configuration file for our Spring project?  
Ans**. **Yes**, we can use as many as we want, all we need to is import them in the main Spring configuration file which we will load from our program.

**Q.) What is the View Resolver pattern? how it works in Spring MVC  
Ans**. View Resolver pattern is a J2EE pattern that allows a web application to dynamically choose its view technology, like HTML, JSP, Tapestry, JSF, XSLT, or any other view technology.  
In this pattern, View resolver holds a mapping of different views, controller return name of the view, which is then passed to View Resolver for selecting an appropriate view.   
  
**Q.) What is the difference between Spring MVC and Spring core?  
Ans**. The Spring MVC is part of the Spring framework, which helps we to develop Java web application using model web controller pattern, while Spring Core provides the Dependency injection and Inversion of Control. The Spring Container is part of Spring core.  
  
Both functionalities come in different JAR files. If we are developing just a core Java application using Spring, then we just need Spring Core, but if we are developing a Web application, then we need spring-mvc.jar as well.   
  
  
**Q.) What are the different implementations of the View interface we have used in Spring MVC?**  
**Ans**. UI based View like JSP, JSTLView,  
  
**Q.) How to escape HTML special characters using Spring MVC?**There are some methods in a Spring tag library,

**Q.) What is Spring Security?  
Ans**. Spring security is a project under the spring framework umbrella, which provides support for security requirements of enterprise Java projects.  
Spring Security, formerly known as aegis security, provides out of box support for creating a login screen, Remember me cookie support, securing URL, authentication provider to authenticate the user from the database, LDAP and in memory, Concurrent Active Session management support and much more.  
  
In order to use Spring security in a Spring MVC based project, we need to include spring-**security.jar and configure it in the application-Context-security.xml file,** we can name it whatever we want, but make sure to supply this to ContextLoaderListener, which is responsible for creating Spring context and initializing dispatcher servlet.



**Q. What is Bean scope in Spring MVC framework with Example**

**Ans**. Bean scope in Spring framework or Spring MVC is scope for a bean managed by Spring IOC container. We may know that Spring is a framework that is based on Dependency Injection and Inversion of Control and provides bean management facilities to Java application. In Spring-managed environment bean (Java Classes) are created and wired by the Spring framework. Spring allows we to define how those beans will be created and the scope of the bean is one of those details. 

In spring framework bean declared in ApplicationContext.xml can reside in five scopes:

**1) Singleton (default scope)**

**2) prototype**

**3) request**

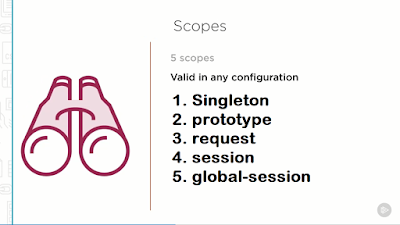
**4) session**

**5) global-session**

**Singleton** and **prototype** are two common bean scope which is available on all Spring Application Context while **request**, **session**, and **global session** bean scope are only available on Web aware application Context like WebApplicationContext.

Singleton bean scope is default scope for bean declared in Spring and applicable when we don't specify scope attribute while specifying bean details in ApplicationContext.xml or Spring configuration file. Singleton bean scope is like a Singleton pattern in Java where only one instance of the bean is created per Spring container.   
So no matter how many times we call getBean() method, the same bean instance will be returned if its bean scope is declared as Singleton. While in the case of prototype bean scope, every getBean() call creates a new instance of Spring bean.

On the other hand **request**, bean scope allows each HTTP request to have its own instance of a bean created and supplied by Spring framework, while **Session** bean scope allows a Web application to have bean instance per session basis. both of these bean scopes are available on WebApplicationContext or any web-aware application context.  
  
The Last one which **is global session** bean scope is only applicable to portlet aware bean scope and allows bean instance per global session. In short singleton vs prototype is important which clearly segregates one instance to multiple instances of bean. 



**Q. How to specify Bean Scope in Spring Framework**

**Ans**. In order to specify bean scope, we can either use Annotation on Spring or we can define it on Application Context, for example in below Spring configuration file AuditService is configured as Singleton using singleton bean scope and PaymentService as prototype bean scope.

//bean configured on singleton bean scope  
<bean id="auditService" class="com.sample.service.impl.AuditServiceImpl"  scope="singleton"/>

Since singleton is also default scope in the spring framework, the following declaration is exactly the same and creates bean on singleton scope.

<bean id="auditService" class="com.sample.service.impl.AuditServiceImpl" />

Though I prefer explicit declaration to make bean scope loud and clear. Now every time we call getBean("auditService") it will return the same instance of AuditService.

AuditService auditService = ApplicationContext.getBean("auditService");

//bean configured on prototype bean scope  
<bean id="auditService" class="com.sample.service.impl.AuditServiceImpl"  scope="prototype"/>

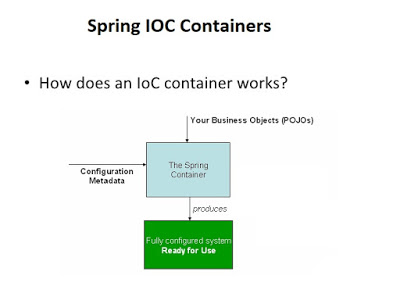
In the case of the prototype, beans cope every call to getBean("auditServie") will return different instances of AuditServiceImpl class. If we want to use Annotation to define bean scope than we can use @Scope("singleton") or @Scope("prototype") on Bean class.   
  
We will also need to enable component scanning in Order to let Spring knows about bean scope. which we can do it spring 2.5 as <context:component-scan base-package="com.sample.service.impl" />. Bean scope has not been changed from various spring version and so far two most used spring version spring 2.5 and spring 3.0 has only five bean scope.

Bean Scope in Spring 2.5 and Spring 3.0 is similar, all default scopes are still supported in spring 3.0 with the addition of few new scopes like thread scope or SimpleThreadScope  which is a scope backed by a thread. We can also register our own custom scope using CustomScopeConfigurer utility., there is no new scope for the bean is introduced on spring 3.0

**Q. Difference between Dependency Injection and Factory Design Pattern in Java Spring?**

**Ans**. The main difference between dependency injection and factory pattern is that in the case of dependency injection dependency is provided by the third party (framework or container) while in the case of factory pattern dependency is acquired by client class itself.  
 Another key difference between them is that the use of dependency injection results in loosely coupled design, but the use of factory patterns creates a tight coupling between factories and classes which are dependent on the product produced by the factory.

Factory Pattern vs. Dependency Injection

Though both Dependency Injection and Factory pattern look similar in a sense that both creates an instance of a class, and also promotes interface-driven programming rather than hard-coding implementation class,  there are some subtle differences between the Factory pattern and Dependency injection pattern, which we'll discuss next.  
 In the case **of Factory design pattern, the client class is responsible** for calling **getInstance()** of factory class to create an instance of products, it also means that client class is directly coupled with the factory and **can't be unit tested without factory class being available.**  
 On the other hand, in Dependency Injection, **the client class has no clue about how his dependencies are created and managed**. **It only knows about dependencies**.  
 Mostly dependencies are injected by a framework like a bean class exists without any hard-coded dependency, as those are injected by IOC container like Spring Framework.  
  


Dependency Injection vs. Factory Pattern in Code

To understand the difference between factory pattern and dependency injection better, let's see examples of how both DI and Factory design pattern are used :

1. Factory Pattern Code Example

**public class** CashRegister {

private PriceCalculator calculator = PriceCalculatorFactory**.getInstance();**

public void add(Transaction tx) {

int price = calcualtor.getPrice(tx);

add(price);

}}

In this case dependent class, **CashRegister is directly coupled with PriceCalculatorFactory** because its calling static gets Instance() method from PriceCalculatorFactory to satisfy its dependency. In order to test CashRegister, we must need a PriceCalculatorFactory, which is **not suitable for unit testing of this class.**On the other hand, if we use Dependency injection, then **dependencies are added by frameworks like Spring framework**or DI container like Google Guice because we reverse the responsibility of acquiring dependencies.  
  
Now it's the responsibility **of IOC container to inject dependency than the dependent class fending for himself.** In the case of dependency injection, any class just looks like a POJO.

2. Dependency Injection Code Example

public class **CashRegister** {

private PriceCalculator calculator;

public CashRegister(PriceCalculator calculator){ // constructor dependency injection.

this.calculator = calculator;

}

public void add(Transaction tx) {

int price = calcualtor.getPrice(tx);

add(price);

}

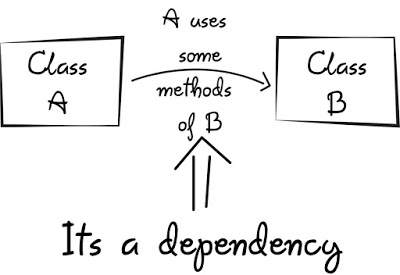
public void setCalcuator(PriceCalculator calc){ // setter injection

this.calculator = calc;

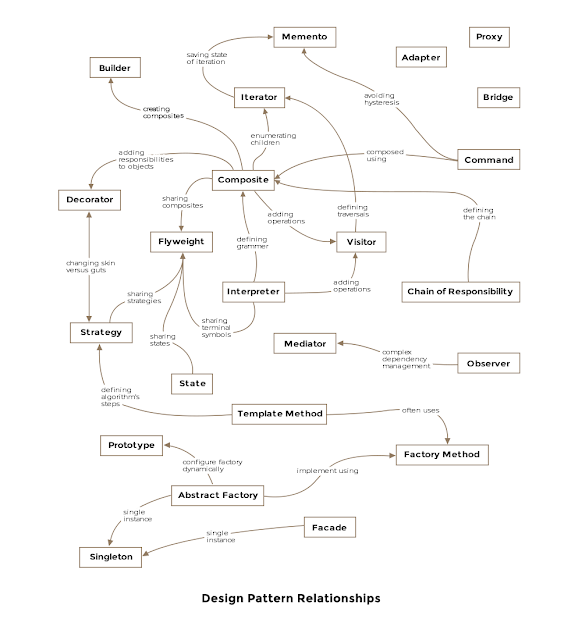
}

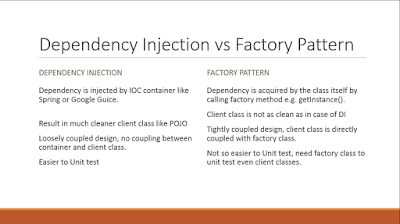
}

We can see that dependency for CashRegister, which is PriceCalculator is supplied via a constructor, this is known as constructor dependency injection.  
There is another form of DI as well, e.g. setter injection, in which dependency is provided using a setter method.  
For example, setCalcuator(PriceCalcuator) is facilitating setter injection there. We should use constructor injection to inject mandatory dependencies and setter injection for optional, good to have dependencies.



Difference between Factory Pattern vs. Dependency Injection

**1**) The factory pattern **adds coupling between objects, factories, and dependency**. **Object not only needs a dependent object to work properly but also a Factory object**. While in case of **dependency injection**, **Object just knows the dependency**, **it doesn't know anything about container or factory  
2**) As compared to the Factory pattern, **Dependency injection makes unit testing easier**. If we use the factory pattern, we need to create the object we want to test, the factory and the dependent object, of course, we factor can return a mock object, but we need all this just to start with unit testing.  On the other hand, if we use dependency injection, we just need to mock the dependency and inject it into an object we want to test, no clutter or boilerplate is required.  
**3**) **Dependency injection is more flexible than the factory pattern.** We can even switch to different DI frameworks like Spring IOC or Google Guice.  
**4**) One of the drawbacks of Dependency injection, as compared to the Factory pattern, is that **we need a container and configuration to inject the dependency, which is not required if we use a factory design pattern**.In a real sense, it's not such a bad thing because we have one place to see the dependency of our class and we can control them, but yes when we compare DI to a factory method, this is the additional step we need to do.  
**5**) Due to low coupling, **DI results in much cleaner co than factory pattern**. Our object looks like POJO, and we also come to know what is mandatory and what is an option by looking at which type of dependency injection our class is using.  
If an object is injected using Setter injection, which means it's optional and can be injected at any time, while dependencies which are injected using constructor injection means they are mandatory and must be supplied in the order they are declared.  
6) Another tricky scenario with using DI is creating an object with too many dependencies and worse if those are injected using constructor injection.  
  
That code becomes difficult to read. One solution to that problem is to use the Facade pattern and inject dependencies by encapsulating in another object.  

**7**) We should use Dependency Injection Patterns to introduce loose coupling. Use Factory Patterns if we need to delegate the creation of objects. In short, dependency injection frees our application from factory pattern boilerplate code.  


The real difference between factory and dependency injection lies in the fact that in the case of a factory, our dependent class is still reliant on a factory, which is a new form of dependency while DI takes out the dependency completely.  
 This means dependency injection provides better decoupling and unit testing of classes over the Factory design pattern.

**Q. What is difference between BeanFactory and ApplicationContext in Spring framework?**

**Ans.**  Both **BeanFactory** and **ApplicationContext** provides a way to get a bean from Spring IOC container by calling getBean("bean name"), but there is some difference in there working and features provided by them. One difference between the bean factory and application context is that the **bean factory only instantiate bean when we call getBean() method** while **ApplicationContext instantiates Singleton bean when the container is started**,  **It doesn't wait for getBean to be called.**

Similarity between both of BeanFactory vs ApplicationContext

Spring provides two kinds of **IOC containers, one is BeanFactory, and the other is ApplicationContext**. Syntactically **BeanFactory and ApplicationContext  both are Java interfaces and ApplicationContext extends BeanFactory.**   
 Both of them are configuration using XML configuration file. In short, **BeanFactory provides basic IOC and DI** features while **ApplicationContext provides advanced features.**

1) **BeanFactory** doesn't provide support for **internationalization** i.e. **i18n but ApplicationContext provides support** for it.

2) Another difference between BeanFactory vs ApplicationContext is the ability to publish events to beans that are registered as listeners.

3) One of the popular implementations of the **BeanFactory interface is XMLBeanFactory**while one of the popular implementations of the **ApplicationContext interface is ClassPathXmlApplicationContext**. On Java web application we use **WebApplicationContext**  which extends the **ApplicationContext** interface and adds the getServletContext method.

4) If we are using auto wiring and using **BeanFactory** than we need to **register AutoWiredBeanPostProcessor** using API which we can configure in XML if we are using  ApplicationContext.   
In summary, **BeanFactory is OK for testing and non-production**use but ApplicationContext is more feature-rich container implementation and should be favored over BeanFactory

I mostly use XML configuration file and ClassPathXmlApplicationContext to quickly run any Spring-based Java program from Eclipse  by using the following snippet of code :

public static void main(String args[]){  
    ApplicationContext ctx =new ClassPathXmlApplicationContext("beans.xml");  
    Hello hello =(Hello) ctx.getBean("hello");  
    hello.sayHello("John");  
}

here beans.xml is our spring configuration file and “hello” is a bean defined in that spring configuration file. Here we have used ClassPathXmlApplicationContext  which is an implementation of ApplicationContext interface in Spring.

Difference between Setter and Constructor Injection in Spring framework

1) The fundamental difference between setter and constructor injection, as their name implies, is How dependency is injected.  Setter injection in Spring uses setter methods like setDependency() to inject dependency on any bean managed by Spring's IOC container. On the other hand, constructor injection uses the constructor to inject dependency on any Spring-managed bean.  
 **2**) Because of using the setter method, **setter Injection in more readable** than constructor injection in Spring configuration file usually applicationContext.xml . Since the setter method has name like setReporotService() by reading Spring XML config file we know which dependency we are setting. While in constructor injection, **since it uses an index to inject the dependency, it's not as readable as setter injection** and we need to refer either Java documentation or code to find which index corresponds to which property.

**3**) Another difference between setter vs constructor injection in Spring and one of the drawbacks of **setter injection is that it does not ensures dependency Injection.** We can not guarantee that certain dependency is injected or not, which means we may have an object with incomplete dependency. On the other hand, constructor Injection does not allow we to construct an object until our dependencies are ready.  
 4) One more **drawback of setter Injection is Security**. By using setter injection, **we can override certain dependency which is not possible with constructor injection** because every time we call the constructor, a new object is gets created.  
 5) Setter and Constructor Injection in Spring, where later can help if there is a circular dependency between two object A and B.  
 If Object A and B are dependent each other i.e A is depends ob B and vice-versa. Spring throws ObjectCurrentlyInCreationException while creating objects of A and B bcz A object cannot be created until B is created and vice-versa. So **spring can resolve circular dependencies through setter-injection.** Objects constructed before setter methods invoked.

**When to use Setter Injection over Constructor Injection in Spring**

**Setter Injection has upper hand** over Constructor Injection in terms of **readability**. Since for configuring Spring we use XML files, readability is a much bigger concern. Also, a **drawback of setter Injection around ensuring mandatory dependency injected** or not can be handled by configuring Spring to check dependency using "dependency-check" attribute of tag or tag.   
 Another worth noting point to remember while comparing Setter Injection vs Constructor Injection is **that once a number of dependencies crossed a threshold like 5 or 6 it's handily manageable to passing dependency via the constructor**. **Setter Injection is the preferred choice when a number of dependencies to be injected is a lot more than normal, if some of those arguments are optional than using a Builder design pattern is also a good option.**

In Summary, both Setter Injection and Constructor Injection have there own advantages and disadvantages. The good thing about Spring is that it doesn't restrict we to use either Setter Injection or Constructor Injection and we are free to use both of them in one Spring configuration file. Use Setter injection when a number of dependencies are more or we need readability. Use Constructor Injection when Object must be created with all of its dependency.

**Q. How Spring MVC Framework works? How HTTP Request is processed?**

**Ans**. It all starts with the client, which sends a request to a specific URL. When that request hits the web container like Tomcat **it looks into web.xml**and **finds the Servlet or Filter which is mapped to that particular URL**. **It the delegate that Servlet or Filter to process the request**. Since Spring MVC is built on top of Servlet, this is also the initial flow of request in any Spring MVC based Java web application.  
 Remember, Web container like Tomcat is responsible for creating Servlet and Filter instances and invoking their various life-cycle methods like init(), service(), destroy(). **In the case of an HTTP request**, **HttpServlet handles that, and depending upon the HTTP request method various doXXX() method is invoked by container like doGet() to process GET request and doPost() to process POST request.**  
 If we remember, to enable Spring MVC, **we need to declare the DispatcherServlet from Spring MVC jar into web.xml.** This Servlet listens for a URL pattern \* as shown in below web.xml, which means all request is mapped to DispatcherServlet.  
Though it is not mandatory, we can have other servlet mapped to other URL if we want to, but if we are using Spring MVC to develop a web application or RESTful web service, it makes sense to pass through all request via DispatcherServlet.  
  
How Spring MVC process an HTTP Request

Here is the web.xml configuration for Spring MVC, we can see that DispatcherServlet is mapped to all request using URL pattern \*

<web-app>

<!-- The front controller of this Spring Web application, responsible

for handling all application requests -->

<servlet>

<servlet-name>Spring MVC Dispatcher Servlet</servlet-name>

<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>

<init-param>

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

<param-value>/WEB-INF/config/web-application-config.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>example</servlet-name>

<url-pattern>\*</url-pattern>

</servlet-mapping>

</web-app>

The URL pattern is important, if the request matches the URL pattern of DispatcherServlet then it will be processed by Spring MVC otherwise not. The DispatcherServlet passes the request to a specific controller depending on the URL requested. How does DispatcherServlet know which request needs to be passed to which controller?  
 Well, it uses the **@RequestMapping annotation or Spring MVC configuration file to find out the mapping of request URL to different controllers**. It can also use specific request processing annotations like @GetMapping or @PostMapping. Controller classes are also identified using @Controller and @RestController (in the case of RESTful Web Services) annotations. See REST with Spring course by Eugen to learn how to develop RESTful Web Service using Spring in depth.  
  
For example, the below class is a Controller that will process any request having URI "/appointments". It also has @GetMapping, which means that the method will be invoked when a GET request is received for this URL. The method annotated with @PostMapping will be invoked if the client sends a POST request to the "/appointments" URI.

@Controller

@RequestMapping("/appointments")

public class AppointmentsController {

@GetMapping

public Map get() {

return appointmentBook.getAppointmentsForToday();

}

@PostMapping

public String add(@Valid AppointmentForm appointment, BindingResult result) {

if (result.hasErrors()) {

return "appointments/new";

}

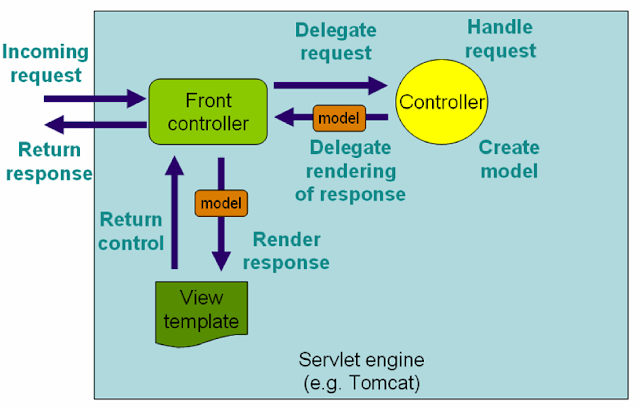
appointmentBook.addAppointment(appointment);

return "redirect:/appointments";

}

}  
After processing the request, the **Controller returns a logical view name and model to DispatcherServlet** and **it consults view resolvers** **until an actual View is determined to render the output**. DispatcherServlet then contacts the chosen view e.g. Freemarker or JSP with model data and it renders the output depending on the model data.  
This Rendered output is returned to the client as an HTTP response. On it's way back it can pass to any configured Filter as well like Spring Security filter chain or Filters configured to convert the response to JSON or XML.  
The DispatcherServlet from Spring MVC framework is an implementation of **Front Controller Pattern** (see Patterns of Enterprise Application Architecture) and it's also a single point of entry - handle all incoming requests, but again that depends upon our URL pattern mapping and our application.  
It delegates requests for further processing to additional components like **Controllers**, **Views**, **View Resolvers**, **handler** **mappers**, **exception** **handlers**, etc. It can also map directly to /, but then the exception for handling static resources needs to be configured. If we look at the web.xml configuration it also pre-loaded using the load-on-startup tag.  
**Spring MVC work Flow**

following diagram which explains the workflow of Spring MVC framework:



The flow of the RESTful Web Service request is also not very different from this. It follows the same path but in the case of REST, the Controller methods are annotated with @ResponseBody which means it doesn't return a logical view name to DispatcherServlet, instead it write the output directly to the HTTP response body.

In summary, here is the flow of an HTTP request in Java application created using the Spring MVC framework:  
1) The client sends an **HTTP request to a specific URL**  
2) **DispatcherServlet of Spring MVC receives the request**  
2) It **passes the request to a specific controller depending** on the URL requested using **@Controller** and **@RequestMapping** annotations.  
3) Spring **MVC Controller then returns a logical view name and model to DispatcherServlet**.  
4) **DispatcherServlet consults view resolvers until actual View is determined** to render the output  
5**) DispatcherServlet contacts the chosen view** (like Thymeleaf, Freemarker, JSP) with model data and it renders the output depending on the model data  
6) **The rendered output is returned to the client as a response**  
That's all about what is the flow of Spring MVC or how an HTTP request is processed by Spring MVC. This is very basic but important knowledge about the Spring MVC framework and every Java and Spring developer should be familiar with this. If we know how our HTTP request is processed then we can not only understand the issues better but also troubleshoot then easily and quickly.

**Q. What is the use of DispatcherServlet in Spring MVC?**

**Ans**. Similar to other Java web framework like Struts 1.x and Struts 2.x, Spring MVC also uses a Front Controller (see Patterns of Enterprise Application Architecture)  to receive all incoming request and delegates to other components for further processing like Spring MVC controllers which are annotated using @Controller annotation and ViewResolvers like the InternalResourceViewResolver class.

A Front Controller is a common pattern in web application and used to receive requests and delegate to other components in the application for actual processing. The DispatcherServlet is a front controller like it provides a single entry point for a client request to Spring MVC web application and forwards request to Spring MVC controllers for processing.  
  
The DispatcherServlet is like any other Servlet class and it has to be declared inside the deployment descriptor or web.xml file as shown below:

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

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

<!--Defaults to WEB-INF\dispatcher-servlet.xml -->

<param-value>classpath:mvc-config.xml</param-value>

</init-param>

</servlet>

Its URL pattern is usually "\*" so that all incoming request should go through Dispatcher servlet as shown below:  
<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/\*</url-pattern>

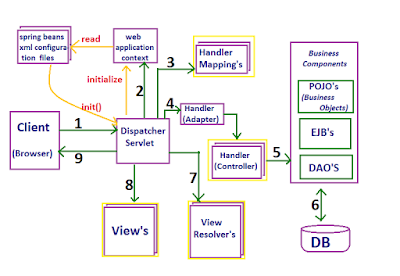
</servlet-mapping>

I haven't shown here but, DispatcherServlet is also usually preloaded using the load-on-startup tag of the deployment descriptor. We can give zero or positive value on this tag to pre-load a servlet, otherwise, the servlet will only be loaded when a request will arise.  
 If our servlet does a lot of job on initialization like DispatcherServlet which initializes all the beans declared in its web context like controllers, view resolvers, and mapping handlers then it could slow down the response time.  
 Btw, it's not the only way to declare DispatcherServlet in Spring MVC. From Spring 3.2 and Servlet 3.0 specification, we can programmatically declare DispatcherServlet using ServletContainerInitializer interface.  
 This is a Servlet 3,0 feature that allows Servlet 3.0 compliant containers like Tomcat 7 or higher to scan and load any class which implements this interface.

Spring provides an implementation of this interface as **SpringServletContainerInitializer** and a convenient class called **AbstractAnnotationConfigDispatcherServletInitialzer** in Spring 3.2 to configure DispatcherServlet without deployment descriptor.  
  
This class implements SpringServletContainerInitializer hence automatically picked by Servlet 3.0 compliant containers.

**How Dispatcher Servlet works Internally in Spring?**

It acts as a front controller and provides a single entry point for the application. It then uses handler mappings and handler adapters to map a request to the Spring MVC controllers. It uses **@Controller and @RequestMapping annotation for that purpose.**  
 Once the request is processed by the Spring MVC controller**, it returns a logical view name instead of the view**. Though, we can even configure Controler's handler methods to not return any View name by declaring return type as void.  
 We can even use @ResponseBody annotation in the case of REST to directly write the output to the HTTP response body.   
 When **DispatherServlet receives a view name, it consults the ViewResolver to find the right view.** There is a chain of ViewResolver that is maintained at the Spring MVC framework. They try to resolve the logical view name into a Physical resource like a JSP page or a FreeMaker or Velocity template.  
 The **ViewResolver is invoked in order, if first in the chain not able to resolve the view then it returns null and next ViewResolver in the chain is consults**.  
 Once the right view is found**, DispatcherServlet forwards the request along with Model data to the View for rendering like a JSP page.** By default, DispatcherServlet uses **InternalResourceViewResolver** which uses **prefix and suffix to convert a logical view name e.g. "home" to /WEB-INF/home.jsp.** The View interface also has getContentType() method, which returns content type the view produces (JstlView has text/HTML). This is usually the default content type for requests handled by the dispatcher servlet in Spring. Here is a nice diagram which explains how DispatcherServlet works internally in Spring MVC



**In short, DispatcherServlet is used for following things in Spring MVC:**

1. Receives all request as Front Controller  and provides a single entry point to the application

2. Mapping requests to correct Spring MVC controller

3. Consulting ViewResolvers to find correct View

4. Forwarding request to chosen View for rendering

Returning the response to the client

5. Creates web-context to initialize the web-specific beans like controllers, view resolvers and handler mapping

It's is one of the key components of Spring MVC which is used to receive all incoming requests and forward them to the right controllers for actual processing. It finds the right controllers by using handler mappings like SimpleUrlHandlerMapping or BeanNameUrlHandlerMapping, which check if the bean name is the same as the view name and the bean implements the View interface.  
If we are using annotations then it can also use @Controller and @RequestMapping annotations to find the right controller to process a particular request. Once the request is processed by the controller it returns a logical view name to DispatcherServlet.  
The DispatcherServlet then consults ViewResolver and LocalResolvers to find the right View to render the output. Once the correct View is chosen, it forwards the request to the View for rendering the response.

**Q. How does DispatcherServlet process request in Spring MVC**

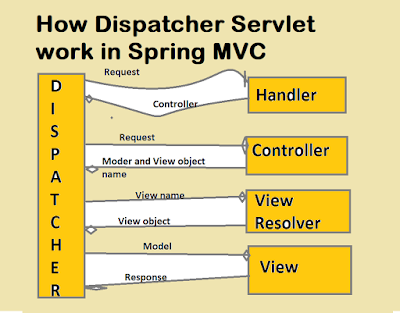
**Ans** . Dispatcher Servlet is used to handle all incoming requests s and route them through different Spring Controllers for further processing. To achieve this, it determines which controllers should handle the incoming request.  
 The **DispatcherServlet uses HandlerMapping implementations** - pre-built or provided as part of the application to route incoming requests to handler objects. By default, it uses **BeanNameUrlHandlerMapping** and **DefaultAnnotationHandlerMapping**, which is driven by @RequestMapping annotation.  
 In order to find the right methods for handling the request, it scans through all the classes declared using @Controller annotation and it also uses @RequestMapping annotation to find the types and methods responsible for handling requests.  
  
The @RequestMapping annotation can map the request

by path like @RequestMapping(“path”),

by HTTP method like @RequestMapping("path", method=RequestMethod.GET),

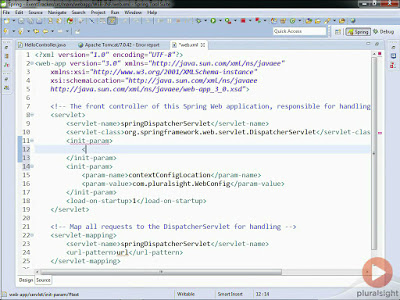
by request parameters like @RequestMapping("path"”, method=RequestMethod.POST, params="param1")

and by the presence of HTTP request header like @RequestMapping("path", header="content-type=text/\*").  
We can also apply @RequestMapping annotation at the class level to filter incoming requests.



Anyway, after processing the request Controller returns the logical view name and model to DispatcherServlet. It then consults to view resolvers to find the actual View to render the output. The view resolution strategy can be specified using a ViewResolver implementation, by default, DispatcherServlet uses InternalResourceViewResolver to convert logical view name to actual View object like a JSP.  
 After this DispatcherServlet contacts the chosen view e.g. a JSP file with model data and it renders the output depending on the model data. This rendered output is returned to the client as a response. Sometimes we don't even need a view e.g. in case of RESTful Web services.   
 **10 Points about DispatcherServlet**

**1**) The DispatcherServlet is the main controller of the Spring MVC Application. All incoming web request passes through DispatcherServlet before processed by individual Spring controllers i.e classes annotated using @Controller annotation.  
**2**) The DispatcherServlet of Spring MVC is an Implementation of Front Controller Pattern (see Introduction to Spring MVC 4). A Front Controller is nothing but a controller that handles all requests for a website. They are often used in Web applications to implement workflows.



**3**) Like any other Servlet, DispatcherServlet of Spring MVC framework is also declared and configured in web.xml file as shown below:

<web-app>

<servlet>

<servlet-name>SpringMVC</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>SpringMVC</servlet-name>

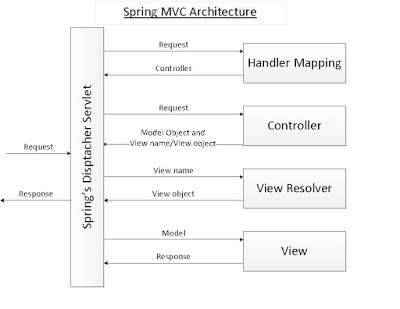
<url-pattern>\*</url-pattern>

</servlet-mapping>

</web-app>

**4**) The DispatcherServlet is an actual Servlet, it inherits from the HttpServlet base class. Servlet engine like Tomcat create an instance of this class and calls it various life-cycle methods e.g. **init(), service() and destroy().**  
5) The DispatcherServlet provides a Single point of entry for our Spring MVC web application. As I said before, it handles all incoming requests.

6) Spring's DispatcherServlet is also completely integrated with the Spring IoC container and as such allows we to use every feature of the Spring framework like dependency injection.  
7) The **dispatcher servlet is configured as load-on-startup = 1** which means this Servlet should be created by Servlet container when we deploy the application rather than creating when a request arrived for this request.  
This is done to reduce the response time of the first request because DispatcherServlet does a lot of job at the startup to scan and find all controllers and request mappings.   
8) During initialization of DispatcherServlet, the Spring MVC framework will look for a file named [**servlet-name]-servlet.xml** in the WEB-INF directory of our web application and create the beans defined there e.g. if servlet name is "**SpringMVC**" as shown in the above web.xml configuration then it will look for a file named **SpringMVC-Servlet.xml.**  
It also overrides the definitions of any beans defined with the same name in the global scope. We can change the exact location of this configuration file by using contextConfigLocation servlet initialization parameter.

[](http://javarevisited.blogspot.sg/2018/02/top-20-spring-rest-interview-questions-answers-java.html#axzz57Kv4wGXe)

**9**) In the Spring MVC framework, each DispatcherServlet has its own **WebApplicationContext** , which inherits all the beans already defined in the root WebApplicationContext. These inherited beans can be overridden in the servlet-specific scope, and new scope-specific beans can be defined locally to a given servlet instance.  
**10**) The DispatcherServlet of Spring MVC framework can also return the last-modification-date, as specified by the Servlet API. It determines the last modification date by looking for an appropriate handler mapping and test if the handler that is found implements the LastModified interface. If yes, then it calls the getLastModified(request) method of the LastModified interface, and value is returned to the client.  
That's all about the DispatcherServlet of Spring MVC framework.

**Q. What is the Role of InternalResourceViewResolver in Spring MVC**

**Ans**: One of the important parts of that processing was view resolution, which is handled by the ViewResolver interface. The **InternalResourceViewResolver is an implementation of ViewResolver** in the Spring MVC framework **which resolves logical view names** like "hello" to internal physical resources like Servlet and JSP files e.g. jsp files placed under the WEB-INF folder. It is a subclass of UrlBasedViewResolver, which uses "prefix" and "suffix" to convert a logical view name returned from the Spring controller to map to actual, physical views.  
For example, if a user tries to access /home URL and HomeController returns "home" then DispatcherServlet will consult InternalResourceViewResolver and it will use prefix and suffix to find the actual physical view which is integral to a web application.

Like, if prefix is "/WEB-INF/views/" and suffix is ".jsp" then "home" will be resolved to "/WEB-INF/**home.jsp**" by InternalResourceViewResolver.  
It's also a  best practice to put all JSP files inside the WEB-INF directory, to hide them from direct access (e.g. via a manually entered URL). If we put then inside the WEB-INF directory then only controllers will be able to access them.  
Even though it's not mandatory that View can only be JSP, it can be JSON also, for example for REST web services, but for the sake of simplicity, we'll take the example of JSP as view in this article.

**Q. How to configure InternalResorveViewResolver in Spring MVC**

**Ans**. Let's first start with the configuration of view resolvers in Spring. We can configure this ViewResolver either using Java Configuration or XML configuration as shown below:  
**1. Configuring ViewResolver using XML in Spring**

Here is some XML snippet to configure a view resolve in Spring, we can use this if we are working on a Spring project which is using XML based confirmation:  
<bean id="viewResolver"

class="org.springframework.web.servlet.view.InternalResourceViewResolver"

prefix="/WEB-INF/" suffix=".jsp" />

**2. Configuring ViewResolver using Java Configuration**

From Spring 3.0 we can also configure view resolver using Java i.e. without XML. We can use the following code to configure the internal resource view resolver in our spring project:  
@Bean

public ViewResolver viewResolver() {

InternalResourceViewResolver irv = new InternalResourceViewResolver();

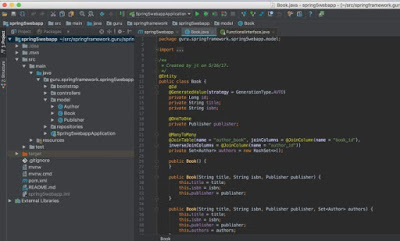
irv.setPrefix("/WEB-INF/");

irv.setSuffix(".jsp");

return irv;

}

We can see that both the XML and Java offers a simple approach to configure internal resource view resolver in Spring. Though, I suggest we use Java Configuration which is modern and now becoming the standard way to configure Spring application.



**Important points about InteralResourceViewResolver in Spring MVC**

**1**) When chaining **ViewResolvers**, an **InternalResourceViewResolver** always needs to be last, as it will attempt to resolve any view name, no matter whether the underlying resource actually exists.  
**2**) The InternalResourceViewResolver is also the **default view resolver** of DispatcherServlet class, which acts as the front controller in Spring MVC framework.  
**3**) **By default, InternalResourceViewResolver returns InternalResourceView** (i.e. Servlets and JSP) but it can be configured to return JstlView by using the viewClass attribute as shown below:

/\*\*

\* Sets the default setViewClass view class to requiredViewClass: by default

\* InternalResourceView, or JstlView if the JSTL API is present.

\*/

public InternalResourceViewResolver() {

Class viewClass = requiredViewClass();

if (viewClass.equals(InternalResourceView.class) && jstlPresent) {

viewClass = JstlView.class;

}

setViewClass(viewClass);

}

/\*\*

\* This resolver requires InternalResourceView.

\*/

@Override

protected Class requiredViewClass() {

return InternalResourceView.class;

}

The advantage of using JstlView is that JSTL tags will get the Locale and any message source configured in Spring. This is particularly important when we are using JSTL tags for formatting for displaying messages.  
JSTL's formatting tags need a Locale to properly format locale-specific values like currency and dates. It's message tags can use a Spring message source and a Locale to properly choose the message to render in HTML depending upon Locale.  
We can further see Spring in Action 5th Edition by Craig Walls for more details on JstlView class. The book is a gem and my favorite to learn Spring in detail. It is now also updated to cover Spring 5.0 changes.

4. The **InteralResourceViewResolver is one of the several built-in view resolvers provided by Spring framework**, some of the most useful ones are listed below:

**BeanNameViewResolver** - resolves views as beans in the Spring application context whose ID is the same as the view name. For example, if we have a bean with id = "home" and a controller return a logical view name as "home" then this bean will be resolved by BeanNameViewResolver.

**FreeMarkerViewResolver** - resolver views as FreeMarker templates

**JasperReportsViewResolver** - resolves views as JasperReports definitions

**XsltViewResolver** - resolves views to be rendered as the result of an XSLT transformation.

Bryan Hassen's has also explained about different types of view resolvers in Spring on his classic course  Introduction to Spring MVC 4, if we have a Pluarlsight membership then that is one of the best resources.

5. The most important benefit of using ViewResolver in Spring MVC is that it decouples request-handling logic in the controller from the view-rendering of a view. In short, the controller doesn't know anything about which view technology is used to render the view.  
It just returns a logical name which could resolve to a JSP, FreeMarker template, Apache tiles or any other view technology. It also means we can change the view layer without changing controller as long as logical view name is same.

The concept of view resolution in Spring MVC is also very important from both the Spring interview as well as the Spring certification point of view. If we are preparing for Spring certification, I suggest we go through some questions shared by David Mayer's Spring Mock exams to test our knowledge of view resolution concept in Spring MVC.

That's all about what does InternalResourceViewResolver do in Spring MVC or what is the role of InternalResourceViewResolver. It's one of the useful class from Spring MVC and as a Java Spring developer, we should be familiar with it.

**Q. Difference between @Autowired and @Inject annotation in Spring?**

**Ans**. The @Autowired annotation is used for auto-wiring in Spring framework. If we don't know, autowiring is a process on which Spring framework figure out dependencies of a Spring bean, instead of we, a developer, explicitly specifying them in the application context file. We can annotate fields and constructor using @Autowired to tell Spring framework to find dependencies for we.  
The @Inject annotation also serves the same purpose, but the main difference between them is that @Inject is a standard annotation for dependency injection and @Autowired is spring specific.  
Since Spring is not the only framework which provides dependency injection, in the future if we change our container and moves to another DI framework like Google Guice, we need to reconfigure our application.

We can potentially avoid that development effort by using standard annotations specified by **JSR-330 e.g.  @Inject, @Named, @Qualifier, @Scope and @Singleton.**  
A bean declared to be auto-wired using @Inject will work in both Google Guice and Spring framework, and potentially any other DI container which supports JSR-330 annotations.

Difference between @Autowired vs @Inject Annotation

If we have worked with Hibernate and JPA in past then JSR-330 annotation is nothing but like JPA annotations which standardize the Object-Relational mapping across the framework. When we use the JPA annotations like @Entity, our code will not only work on Hibernate but also on other ORM tools and framework e.g. TopLink.  
Btw, like all similar things in the world, even though both @Autowired and @Inject serve the same purpose there are a couple of differences between them,

1) The first and most important difference between @Autowired and @Inject annotation is that the **@Inject** annotation is only available from Spring 3.0 onwards, so if we want to use annotation-driven dependency injection in Spring 2.5 then we have to use the @Autowired annotation.  
2) The second difference between these two annotations is that unlike Spring's @Autowired,  the **@Inject**does not require the 'required' attribute.  
3) The third most common difference between @Autowired and @Inject annotation is that **@Autowired**  is Spring specific while **@Inject**  is the standard for Dependency Injection, specified in JSR-330.  
In general, I recommend the use of JSR 330 annotation for DI, the @Inject annotation is as capable as Spring's @Autowired and if we want we can also mix and match this with Spring's @Value and @Lazy annotations.  
4) The @Autowired annotation was added on Spring 2.5 and used for annotation-driven dependency injection. It works in conjunction with @Component annotation and <context:component-scan /> to streamline development cycle.  
From Spring 3.0, Spring offers support for JSR-330 dependency injection annotations e.g. @Inject, @Named, and @Singleton. It also added more Spring specific annotations e.g. @Primary, @Lazy, and @DependsOn annotation.

5) The @Inject annotation is good from the portability point of view. Since @Autowired is specific to Spring framework, if we ever decided to move to Google Guice or any other dependency injection framework then we need to re-implement our dependency injection logic, even though our application remains same. All bean creation logic needs to be changed to match with Google Guice's implementation.

**Q. Differences between @RequestParam(query parameters) and @PathVariable annotations in Spring MVC?**

**Ans.** As the name suggests, @RequestParam is used to get the request parameters from **URL**, also known as **query parameters**, while @PathVariable extracts values from URI.  
  
For example, if the incoming HTTP request to retrieve a book on topic "Java" is http://localhost:8080/shop/order/1001/receipts**?date=12-05-2017**, then we can use the **@RequestParam** annotation to retrieve the query parameter date and we can use **@PathVariable** to extract the orderId i.e. "1001" as shown below:  
  
@RequestMapping(value="/order/{orderId}/receipts", method = RequestMethod.GET)  
 public List listUsersInvoices

( @PathVariable("orderId") int order, @RequestParam(value = "date", required = false) Date dateOrNull) {  
...  
}  
The required=false denotes that the query parameter can be optional, but the URL must have the same URI.

**Q. How to extract Query Parameters in Spring MVC using @RequestParam**

**Ans**. Spring MVC is a rich framework to develop both web applications and RESTful web services in Java. It provides several ways to retrieve data from the incoming HTTP requests like

Request Parameters

Path Variables

Form inputs

1. Using @RequestParam to get Query parameters

In a Spring MVC application, we can use the @RequestParam annotation to accept query parameters in Controller's handler methods.  
  
http://localhost:8080/eportal/orders?id=1001 @RequestParam  
  
@RequestMapping("/orders")  
public String showOrderDetails(@RequestParam("id") String orderId, Model model){  
   model.addAttribute("orderId", orderId);  
   return "orderDetails";  
}  
If the name of the query parameter is the same as the name of the variable in handler's @RequestParam annotated argument then we can simply use @RequestParam without specifying the name of a query parameter, Spring will automatically derive the value (see Introduction to Spring MVC).

Also, here is the code to prove the point:  
  
**URL: http://localhost:8080/eportal/trades?tradeId=2001**  
  
@RequestMapping("/trades")  
public String showTradeDetails(@RequestParam String tradeId,Model model){  
  model.addAttribute("tradeId", tradeId);  
  return "tradeDetails";  
}  
  
We can see that we have just annotated the method parameter tradeId with @RequestParam without specifying the name of the query parameter because the name of both request parameter and argument name is the same, i.e., "tradeId."  
  
2. Using @PathVariable annotation to extract values from URI

We can use Spring MVC's @PathVaraible annotation to extract any value which is embedded in the URL itself. Spring calls it a URI template, where @PathVariable is used to obtain some placeholders from the URI itself.  
  
If we have worked in RESTful Web services, we might know that the REST URIs contains values, e.g. a REST API to retrieve a book using ISBN number looks like the following:  
  
URL: http://localhost:8080/book/9783827319333  
  
Now, to extract the value of ISBN number from the URI in our Spring MVC Controller's handler method, we can use @PathVariable annotation as shown in the following code:  
  
@RequestMapping(value="/book/{ISBN}", method= RequestMethod.GET)  
public String showBookDetails(@PathVariable("ISBN") String id,  
                              Model model){  
   model.addAttribute("ISBN", id);  
   return "bookDetails";  
}  
  
Similar to @RequestParameter annotation, we also can also omit the value attribute in @PathVariable annotation, if the name of the path variable's placeholder in the @RequestMapping annotation is same to the variable name in the handler method's @PathVariable annotated parameter

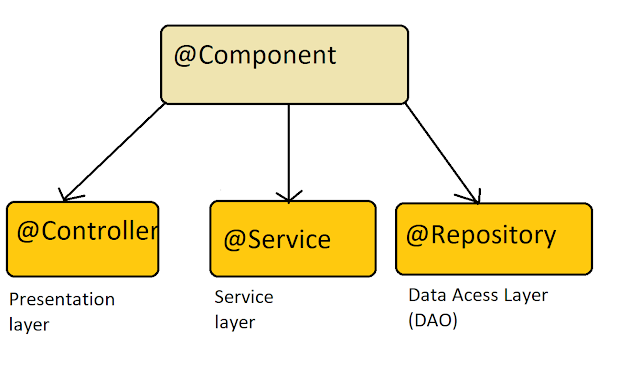
For example, we can rewrite the above code as shown below:  
  
@RequestMapping(value="/book/{ISBN}", method= RequestMethod.GET)  
public String showBookDetails(@PathVariable String ISBN, Model model){  
   model.addAttribute("ISBN", ISBN);  
   return "bookDetails";  
}  
  
Spring MVC provides several useful annotations to map and extract data from the HTTP request, and as of Spring developer, we should be aware of these, e.g. **@RequestMapping, @RequestParam, and @PathVariable.**  
Difference between @PathVariable and @RequestParam in Spring

1) The **@RequestParam** is used to extract **query parameters** while @PathVariable is used to extract data right from the URI  
2) **@RequestParam** is more useful on a traditional web application where data is mostly passed in the query abatements while @PathVariable is more suitable for RESTful web services where URL contains values, like http://localhost:8080/book/9783827319333, here data, which is ISBN number is part of URI.  
3) **@RequestParam** annotation can specify **default values** if a query parameter is not present or **empty by using a defaultValue** attribute, **provided the required attribute is false**.  
4) Spring MVC allows we to use multiple @PathVariable annotations in the same method, provided, no more than one argument has the same pattern.

**Q. Difference between @Component, @Service, @Controller, and @Repository in Spring ?**

**Ans**. During the initial release of Spring, all beans are used to be declared in an XML file. For a large project, this quickly becomes a massive task, and Spring guys recognize the problem rather quickly. In later versions, they provide annotation-based dependency injection and Java-based configuration. From Spring 2.5 annotation-based dependency injection was introduced, which automatically scans and registers classes as Spring bean which is annotated **using @Component annotation**.  
This means we don't declare that bean using the <bean> tag and inject the dependency, it will be done automatically by Spring. This functionality was enabled and disabled using <context:component-scan> tag.  
  
Now that we know what does @Component annotation does let's see what does @Service, @Controller, and @Repository annotation do.  
  
They are nothing but the specialized form of @Component annotation for certain situations. Instead of using @Component on a controller class in Spring MVC, we use @Controller, which is more readable and appropriate.  
  
By using **@Controller annotation we do two things,** first, we declare that this class is a Spring bean and should be created and maintained by Spring ApplicationContext, but also we indicate that its a controller in MVC setup. This latter property is used by web-specific tools and functionalities.

For example, DispatcherServlet will look for @RequestMapping on classes that are annotated using @Controller but not with @Component.  
  
This means @Component and @Controller are the same with respect to bean creation and dependency injection but later is a specialized form of former. Even if we replace @Controller annotation with @Compoenent, Spring can automatically detect and register the controller class but it may not work as we expect with respect to request mapping.

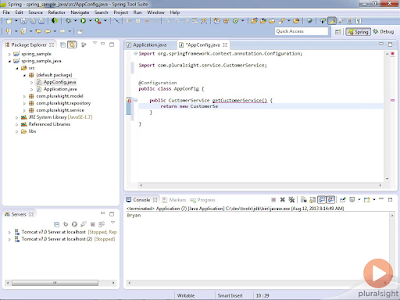


The same is true for **@Service and @Repository annotation**, **they are a specialization of @Component in service and persistence layer.** A Spring bean in the service layer should be annotated using @Service instead of @Component annotation and a spring bean in the persistence layer should be annotated with @Repository annotation.  
  
By using a specialized annotation we hit two birds with one stone. First, they are treated as Spring bean, and second, we can put special behavior required by that layer.  
  
For example, @Repository's not only helping in annotation based configure but also catch Platform-specific exceptions and re-throw them as one of Spring’s unified unchecked exception.  
  
Though for that we also need to declare org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor as Spring bean in our application context.

This bean post-processor adds an advisor to any bean that’s annotated with @Repository so that any platform-specific exceptions are caught and then rethrown as one of Spring’s unchecked data access exceptions.

**Q. How does Component Scanning work in Spring?**

**Ans**. From Spring 2.0, Spring provides <**context:component-scan**> and annotation-driven dependency injection to automatically detect and register Spring bean instead of specifying them in the XML file.  
  
But, it only scans @Component and does not look for @Controller, @Service, and @Repository in general. They are scanned because they themselves are annotated with @Component.  
  
Just take a look at @Controller, @Service, and @Repository annotation definitions:  
  
@Component  
public @interface Service {  
….  
}  
  
@Component  
public @interface Repository {  
….  
}  
  
@Component  
public @interface Controller {  
…  
}  
  
Thus, it’s not wrong to say that **@Controller, @Service, and @Repository are special types of @Component annotation.**<**context:component-scan**> picks them up and registers their following classes as beans, just as if they were annotated with @Component.  
  
They are scanned because they themselves are annotated with @Component annotation. If we define our own custom annotation and annotate it with @Component, then it will also get scanned with <context:component-scan>.

[](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fspring-mvc4-introduction)

Difference between @Component, @Service, @Controller, and @Repository in Spring

Here is a nice summary of what does @Component, @Service, @Controller, and @Repository annotation do in Spring Framework:

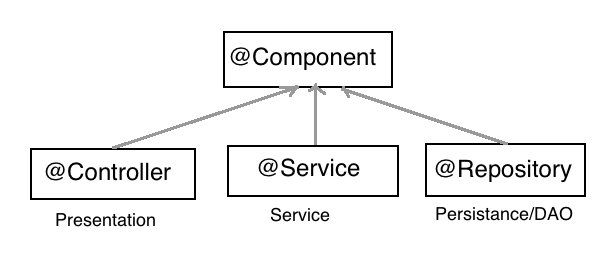
@**Component** is a generic stereotype for any Spring-managed component or bean.

**@Repository** is a stereotype for the persistence layer.

@**Service** is a stereotype for the service layer.

**@Controller** is a stereotype for the presentation layer (spring-MVC).

And here is the nice diagram to explain the hierarchy of all these annotations in Spring Framework:

  
  
Their only difference comes in their purpose i.e. @Controller is used in Spring MVC to define controller, which are first Spring bean and then the controller. Similarly, @Service is used to annotated classes that hold business logic in the Service layer and @Repository is used in the Data Access layer.  
  
**Q. Difference between @RestController and @Controller Annotation in Spring MVC and REST?**

**Ans**. The **@RestController** annotation in Spring MVC is nothing but a combination of **@Controller and @ResponseBody** annotation. It was **added into Spring 4.0 to** make the development of RESTful Web Services in Spring framework easier. If we are familiar with the REST web services we know that the fundamental difference between a web application and a REST API is that the response from a web application is generally view (HTML + CSS + JavaScript)  because they are intended for human viewers while REST API just returns data in form of JSON or XML because most of the REST clients are programs. This difference is also obvious in the @Controller and @RestController annotation.  
  
The job of @Controller is to create a Map of the model object and find a view but **@RestController simply returns the object** **and object data is directly written into HTTP response as JSON or XML**.This can also be done with traditional @Controller and use @ResponseBody annotation but since this is the default behavior of RESTful Web services, Spring introduced @RestController which combined the behavior of @Controller and @ResponseBody together.  
  
In short, the following two code snippet are equal in Spring MVC:

@Controller

@ResponseBody

public class MVCController {

.. our logic}

@RestController

public class RestFulController {

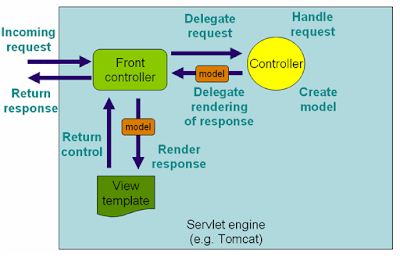
.... our logic

}

Obviously, everybody would like to declare just one annotation instead of two. Also, the @RestController is more obvious and telling than the previous two.

In the Spring framework, A **Controller** is a class that is **responsible for preparing a model Map with data to be displayed by the view as well as choosing the right view itself**. It can also directly write into the response stream by using @ResponseBody annotation and complete the request.  
  
The **behavior of writing directly into response stream is very useful for responding calls to RESTful web services** because their we just return data instead of returning a view   
  
If we have developed RESTful Web services before Spring 4 like in Spring 3 or Spring 3.1, we would have been familiar with using a combination of **@Controller and @ResponseBody to create a RESTful response**. Spring guys take cognizant of these issues and created @RestController.  
Now, we don't need to use @Controller and @RestponseBody annotation, instead we can use @RestController to provide the same functionality.   


Difference between @RestController and @Controller in Spring

1. The **@Controller is a common annotation that is used to mark a class as Spring MVC** Controller while **@RestController is a special controller used in RESTFul** web services and the equivalent of **@Controller + @ResponseBody**.  
2. The **@RestController is relatively new**, added only on Spring 4.0 but **@Controller is an old annotation**, exists since Spring started supporting annotation, officially it was added on Spring 2.5 version.   
3. The **@Controller annotation indicates that the class is a "Controller**" like a web controller while **@RestController** annotation indicates that the class is a controller **where @RequestMapping methods assume @ResponseBody semantics by default** i.e. servicing REST API.  
4. The **@Controller is a specialization of @Component annotation** while **@RestController is a specialization of @Controller annotation**. It is actually a convenience controller annotated with @Controller and @ResponseBody as shown below.  
  
@Target(value=TYPE)  
@Retention(value=RUNTIME)  
@Documented  
@Controller  
@ResponseBody  
**public @interface RestController**  
and here is how the declaration of @Controller looks like:  
  
@Target(value=TYPE)  
@Retention(value=RUNTIME)  
@Documented  
@Component  
**public @interface Controller**  
5. One of the key differences between @Controler and @RestCotroller in Spring MVC is that once we mark a class @RestController then every method is written a domain object instead of a view. [](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fspring-mvc4-introduction)

6. Another key difference between **@RestController** and @Controller is that we **don't need to use @ResponseBody**on every handler method once we annotate the class with @RestController as shown below:  
  
with @RestControler

@RestController

public class Book{

@RequestMapping(value={"/book"})

public Book getBook(){

//...

return book;

}

}

without @RestController  
@Controller

public class Book{

@RequestMapping(value={"/book"})

@ResponseBody

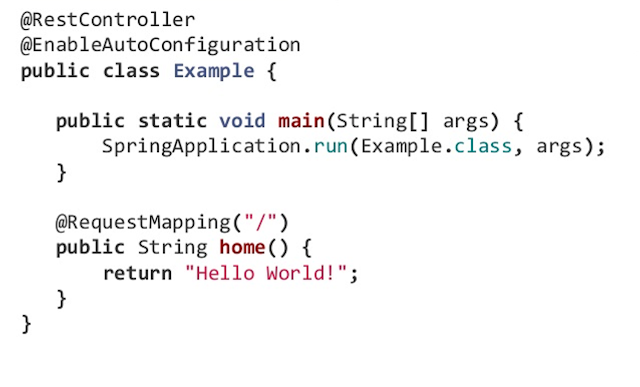
public Book getBook(){

//...

return book;

}}

We can see that if we use Spring MVC @Controller annotation to create a RESTful response we need to annotate each method with the @ResponseBody annotation, which is not required when we use @RestController. It not only makes our code more readable but also saves a couple of keystrokes for we.  
Here is a simple HelloWorld example using @RestController and SpringBoot framework:



So, if we are creating a RESTful Web Services it's better to use @RestController than combining @Controller to @ResponseBody.

**Q. JDBC Database Connection Pool in Spring Framework – How to Setup Example**

**Ans**. Setting up JDBC Database Connection Pool in Spring framework is easy for any Java application, just matter of changing a few configurations in the spring configuration file. If we are writing core java application and not running on any web or application server like Tomcat or  Weblogic,

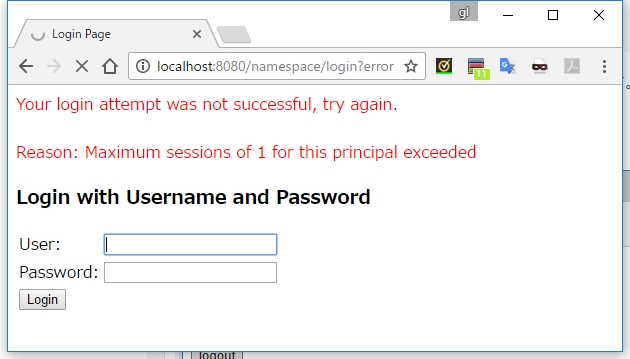
**Q. How to limit number of concurrent session in a Java web application using Spring Security?**

**Ans**. If we don't know, Spring security can limit the number of sessions a user can have in a Java web application. If we are developing a web application especially a secure web application in Java JEE then we must have come up with the requirement similar to many online banking portals have like only one session per user at a time or no concurrent session per user. If the user tries to open a new session then either an alert is shown or his previous session is closed. Even though we can also implement this functionality without using spring security but with Spring security, it's just a piece of cake with coffee :).   
  
We just need to add a couple of lines of XML in our spring security configuration file and we are done. In order to implement this functionality, we can use the <**concurrency-control**> tag.  
  
We can configure a maximum number of the session our application support and then Spring security will automatically detect if user breach that limits and direct them to invalid session url we have specified with this tag e.g. to a logout page.   
Similar to this, Spring Security provides lots of Out of Box functionality a secure enterprise or web application needed for authentication, authorization, session management, password encoding, secure access, session timeout, etc.   
  
In our spring security example, we have seen how to do LDAP Authentication in an Active directory using spring security and in this spring security example we will see how to limit the number of session users can have in Java web application or restricting concurrent user session.  
Spring Security Example: Limit Number of User Session

We will need to include the following xml snippet in our Spring Security Configuration file mostly named as applicaContext-security.xml. We can name the file whatever we want but just make sure we use the same name in all relevant places. If we are not sure how to enable Spring Security in Java web application, check that article first.   
  
Here is sample spring security Example of limiting user session in Java web application:

**<session-management invalid-session-url="/logout.html">  
    <concurrency-control max-sessions="1" error-if-maximum-exceeded="true" />  
</session-management>**

As we see we can specify how many concurrent sessions per user is allowed, a most secure system like online banking portals allows just one authenticated session per user.   
  
The Max-session specifies how many concurrent authenticated session is allowed and if error-if-maximum-exceeded set to true it will flag an error if a user tries to login into another session.  
  
For example, if we try to log in twice from our browser to this spring security application then we will receive an error saying "Maximum Sessions of 1 for this principal exceeded" as shown below:

[](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Flearn-spring-security-4-basics-hands-on%2F)

We can even specify a URL where the user will be taken if they submit an invalid session identifier that can be used to detect session timeout. The session-management element is used to capture the session related stuff.   
  
This is just an example of what Spring security can add to our Java web application. It provides many such advanced and necessary features which can be enabled using some XML tag or annotations.   
  
If we are interested to learn more about advanced Spring security features, I suggest we go through the Learn Spring Security course by Eugen Paraschiv, which the most up-to-date online course on Spring Security and covers new security features from Spring Security 5 release. 

Dependency

This code has a dependency on the spring-security framework. We need to download spring security jar like spring-security-web-3.1.0.jar and add it into application classpath.

This simple example of spring security shows the power of spring security, a small piece of xml snippet can add very useful and handy security feature in our Java web application.   
  
I strongly recommend using spring security for our new or existing Java web application created using Servlet JSP.

Managing Database connection pool using Apache Commons DBCP and Commons Pool along-with Spring framework is a nice choice but if we have the luxury of having web server and managed J2EE Container, consider using Connection pool managed by J2EE server those are a better option in terms of maintenance, flexibility and also help to prevent java.lang.OutofMemroyError: PermGen Space in tomcat by avoiding loading of JDBC driver in web-app class-loader.

Also keeping JDBC connection pool information in Server makes it easy to change or include settings for JDBC over SSL. In this article, we will see how to set up Database connection pool in spring framework using Apache Commons DBCP and commons pool.jar

This article is in continuation of my tutorials on spring framework and database like LDAP Authentication in J2EE with Spring Security and manages session using Spring security  If we haven’t read those articles then we may find them useful.

By the way, if we are new to Spring framework then I also suggest we join a comprehensive and up-to-date course to learn Spring in depth. If we need recommendations, I highly suggest we take a look at Spring Framework 5: Beginner to Guru, one of the comprehensive and hands-on course to learn modern Spring. It' also most up-to-date and covers Spring 5.  
  
 It's also very affordable and we can buy in just $10 on Udemy sales which happen every now and then.

Spring Example JDBC Database Connection Pool

Spring framework provides a convenient JdbcTemplate class for performing all Database related operations. if we are not using Hibernate than using Spring's JdbcTemplate is a good option. JdbcTemplate requires a DataSource which is javax.sql.DataSource implementation and we can get this directly using spring bean configuration or by using JNDI if we are using the J2EE web server or application server for managing Connection Pool.   
  
See How to setup JDBC connection Pool in tomcat and Spring for JNDI based connection pooling for more details. In order to the setup Data source we will require the following configuration in our applicationContext.xml (spring configuration) file:

//Datasource connection settings in Spring  
<bean id="springDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close" >  
   <property name="url" value="jdbc:oracle:thin:@localhost:1521:SPRING\_TEST" />  
   <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />  
   <property name="username" value="root" />  
   <property name="password" value="root" />  
   <property name="removeAbandoned" value="true"/>  
   <property name="initialSize" value="20" />  
   <property name="maxActive" value="30" />  
</bean>  
  
//Dao class configuration in spring  
 <bean id="EmployeeDatabaseBean" class="com.test.EmployeeDAOImpl">  
    <property name="dataSource" ref="springDataSource"/>  
 </bean>

Below configuration of DBCP connection pool will create 20 database connection as initialSize is 20 and goes up to 30 Database connection if required as maxActive is 30. we can customize our database connection pool by using different properties provided by Apache DBCP library.   
  
The above example is creating a connection pool with Oracle 11g database and we are using oracle.jdbc.driver.OracleDriver comes along with ojdbc6.jar or ojdbc6\_g.jar,  to learn more about how to connect Oracle database from Java program see the link.

Java Code for using Connection pool in Spring

Below is a complete code example of DAO class which uses Spring JdbcTemplate to execute a SELECT query against the database using database connection from the Connection Pool. If we are not initializing Database connection pool on start-up than it may take a while when we execute our first query because it needs to create a certain number of SQL connection and then it executes query but once connection pool is created subsequent queries will execute faster.

//Code for DAO Class using Spring JdbcTemplate  
package com.test  
import javax.sql.DataSource;  
import org.log4j.Logger;  
import org.log4j.LoggerFactory;  
import org.springframework.jdbc.core.JdbcTemplate;  
  
/\*\*  
 \* Java Program example to use DBCP connection pool with Spring framework  
 \* @author Javin Paul  
 \*/  
public class EmployeeDAOImpl implements EmployeeDAO {  
  
    private Logger logger = LoggerFactory.getLogger(EmployeeDAOImpl.class);  
    private JdbcTemplate jdbcTemplate;  
  
    public void setDataSource(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
  
    @Override  
    public boolean isEmployeeExists(String emp\_id) {  
        try {  
            logger.debug("Checking Employee in EMP table using Spring Jdbc Template");  
            int number = this.jdbcTemplate.queryForInt("select count(\*) from EMP where emp\_id=?", emp\_id);  
            if (number > 0) {  
                return true;  
            }  
        } catch (Exception exception) {  
            exception.printStackTrace();  
        }  
        return false;  
    }  
}

Dependency:

1. we need to include oracle driver-jar like ojdbc\_6.jar in our classpath.

2. Apache DBCP and commons-pool jar in the application classpath.

That's all on how to configure the JDBC Database connection pool in the Spring framework. As I said its pretty easy using Apache DBCP library. Just matter of few configurations in spring applicationContext.xml and we are ready. If we want to configure the JDBC Connection pool on tomcat (JNDI connection pool) and want to use in spring than see here.