Spring :

# IoC Container

1. [IoC Container](http://www.javatpoint.com/ioc-container)
2. [Using BeanFactory](http://www.javatpoint.com/ioc-container)
3. [Using ApplicationContext](http://www.javatpoint.com/ioc-container)

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:

* to instantiate the application class
* to configure the object
* to assemble the dependencies between the objects

There are two types of IoC containers. They are:

1. **BeanFactory**
2. **ApplicationContext**

### Difference between BeanFactory and the ApplicationContext

The org.springframework.beans.factory.**BeanFactory** and the org.springframework.context.**ApplicationContext** interfaces acts as the IoC container. The ApplicationContext interface is built on top of the BeanFactory interface. It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context (e.g. WebApplicationContext) for web application. So it is better to use ApplicationContext than BeanFactory.

#### Using BeanFactory

The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, we need to create the instance of XmlBeanFactory class as given below:

1. Resource resource=**new** ClassPathResource("applicationContext.xml");
2. BeanFactory factory=**new** XmlBeanFactory(resource);

The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object to create the object of BeanFactory.

#### Using ApplicationContext

The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

1. ApplicationContext context =
2. **new** ClassPathXmlApplicationContext("applicationContext.xml");

The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of ApplicationContext.

**Table 3.5. Feature Matrix**

| **Feature** | **BeanFactory** | **ApplicationContext** |
| --- | --- | --- |
| Bean instantiation/wiring | Yes | Yes |
| Automatic BeanPostProcessor registration | No | Yes |
| Automatic BeanFactoryPostProcessor registration | No | Yes |
| Convenient MessageSource access (for i18n) | No | Yes |
| ApplicationEvent publication | No | Yes |

# Ways to read Properties file in Spring :

* 1. In XML, new properties files can be made accessible to Spring via **the <context:property-placeholder … > namespace element**

<context:property-placeholder location="classpath:foo.properties" />

To read multiple properties files

<context:property-placeholder

location="classpath:dao.properties,

classpath:services.properties,

classpath:user.properties"

ignore-unresolvable="true"/>

or

<bean id="propertyConfigurer" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations">

<list>

<value>classpath:dao.properties</value>

<value>classpath:services.properties</value>

<value>classpath:user.properties</value>

</list>

</property>

<property name="ignoreUnresolvablePlaceholders" value="true"/>

</bean>

* 1. **Register a Properties File via Java Annotations**

Spring 3.1 also introduces **the new @PropertySource annotation**, as a convenient mechanism for adding property sources to the environment. This annotation is to be used in conjunction with Java based configuration and the @Configurationannotation:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Configuration  @PropertySource("classpath:foo.properties")  public class PropertiesWithJavaConfig {     @Bean     public static PropertySourcesPlaceholderConfigurer       propertySourcesPlaceholderConfigurer() {        return new PropertySourcesPlaceholderConfigurer();     }} |

As opposed to using XML namespace element, the Java @PropertySourceannotation **does not automatically register a PropertySourcesPlaceholderConfigurer with Spring**.

Instead, the bean must be explicitly defined in the configuration to get the property resolution mechanism working. The reasoning behind this unexpected behavior is by design and documented on [this issue](https://jira.springsource.org/browse/SPR-8539).

* 1. **Java configuration**

|  |  |
| --- | --- |
| 2  3  4  5  6  7  8  9  10 | @Bean  public static PropertyPlaceholderConfigurer properties() {    PropertyPlaceholderConfigurer ppc      = new PropertyPlaceholderConfigurer();    Resource[] resources = new ClassPathResource[]      { new ClassPathResource( "foo.properties" ) };    ppc.setLocations( resources );    ppc.setIgnoreUnresolvablePlaceholders( true );    return ppc;  } |

# [What is the difference between ApplicationContext and WebApplicationContext in Spring MVC?](http://stackoverflow.com/questions/11708967/what-is-the-difference-between-applicationcontext-and-webapplicationcontext-in-s)

Web Application context extended Application Context which is designed to work with the standard [javax.servlet.ServletContext](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletContext.html) so it's able to communicate with the container.

public interface WebApplicationContext extends ApplicationContext {

ServletContext getServletContext();

}

Beans, instantiated in WebApplicationContext will also be able to use ServletContext if they implement ServletContextAware interface

package org.springframework.web.context;

public interface ServletContextAware extends Aware {

void setServletContext(ServletContext servletContext);

}

There are many things possible to do with the ServletContext instance, for example accessing WEB-INF resources(xml configs and etc.) by calling the getResourceAsStream() method. Typically all application contexts defined in web.xml in a servlet Spring application are Web Application contexts, this goes both to the root webapp context and the servlet's app context.

Also, depending on web application context capabilities may make your application a little harder to test, and you may need to use [MockServletContext](http://static.springsource.org/spring/docs/1.1.4/api/org/springframework/mock/web/MockServletContext.html) class for testing.

**Difference between servlet and root context** Spring allows you to build multilevel application context hierarchies, so the required bean will be fetched from the parent context if it's not present in the current application context. In web apps as default there are two hierarchy levels, root and servlet contexts: .

This allows you to run some services as the singletons for the entire application (Spring Security beans and basic database access services typically reside here) and another as separated services in the corresponding servlets to avoid name clashes between beans. For example one servlet context will be serving the web pages and another will be implementing a stateless web service.

This two level separation comes out of the box when you use the spring servlet classes: to configure the root application context you should use context-param tag in your web.xml

<context-param>

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

<param-value>

/WEB-INF/root-context.xml

/WEB-INF/applicationContext-security.xml

</param-value>

</context-param>

(the root application context is created by [ContextLoaderListener](http://static.springsource.org/spring/docs/3.0.x/api/org/springframework/web/context/ContextLoaderListener.html) which is declared in web.xml

<listener>

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

</listener>

) and servlet tag for the servlet application contexts

<servlet>

<servlet-name>myservlet</servlet-name>

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

<init-param>

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

<param-value>app-servlet.xml</param-value>

</init-param>

</servlet>

Please note that if init-param will be omitted, then spring will use myservlet-servlet.xml in this example.

See also: [Difference between applicationContext.xml and spring-servlet.xml in Spring Framework](http://stackoverflow.com/questions/3652090/difference-between-applicationcontext-and-spring-servlet-xml-in-spring)

# [Difference between applicationContext.xml and spring-servlet.xml in Spring Framework](http://stackoverflow.com/questions/3652090/difference-between-applicationcontext-xml-and-spring-servlet-xml-in-spring-frame)

Spring lets you define multiple contexts in a parent-child hierarchy.

The applicationContext.xml defines the beans for the "root webapp context", i.e. the context associated with the webapp.

The spring-servlet.xml (or whatever else you call it) defines the beans for one servlet's app context. There can be many of these in a webapp, one per Spring servlet (e.g. spring1-servlet.xmlfor servlet spring1, spring2-servlet.xml for servlet spring2).

Beans in spring-servlet.xml can reference beans in applicationContext.xml, but not vice versa.

All Spring MVC controllers must go in the spring-servlet.xml context.

In most simple cases, the applicationContext.xml context is unnecessary. It is generally used to contain beans that are shared between all servlets in a webapp. If you only have one servlet, then there's not really much point, unless you have a specific use for it.

----===+++++++++++++++++++

Dependency Injection

The Dependency Injection is a design pattern that removes the dependency of the programs. In such case we provide the information from the external source such as XML file. It makes our code loosely coupled and easier for testing. In such case we write the code as:

1. **class** Employee{
2. Address address;
4. Employee(Address address){
5. **this**.address=address;
6. }
7. **public** **void** setAddress(Address address){
8. **this**.address=address;
9. }
11. }

In such case, instance of Address class is provided by external souce such as XML file either by constructor or setter method.

Two ways to perform Dependency Injection in Spring framework

Spring framework provides two ways to inject dependency

* By Constructor
* By Setter method

++++++++++++++++++++++++++++++++

10) What are the different bean scopes in spring?

There are 5 bean scopes in spring framework.

|  |  |  |
| --- | --- | --- |
| **No.** | **Scope** | **Description** |
| 1) | Singleton | The bean instance will be only once and same instance will be returned by the IOC container. It is the default scope. |
| 2) | Prototype | The bean instance will be created each time when requested. |
| 3) | Request | The bean instance will be created per HTTP request. |
| 4) | Session | The bean instance will be created per HTTP session. |
| 5) | globalsession | The bean instance will be created per HTTP global session. It can be used in portlet context only. |

++++++==============\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_--

Spring LifeCycle :



1. @PostConstruct and @PreDestroy annotations
2. The spring container finds the bean’s definition from the XML file and instantiates the bean.
3. Spring populates all of the properties as specified in the bean definition (DI).
4. If the bean implements BeanNameAware interface, spring passes the bean’s id to setBeanName() method.
5. If Bean implements BeanFactoryAware interface, spring passes the beanfactory to setBeanFactory() method.
6. If there are any bean BeanPostProcessors associated with the bean, Spring calls postProcesserBeforeInitialization()method.
7. If the bean implements IntializingBean, its afterPropertySet() method is called. If the bean has init method declaration, the specified initialization method is called.
8. If there are any BeanPostProcessors associated with the bean, their postProcessAfterInitialization() methods will be called.
9. If the bean implements DisposableBean, it will call the destroy() method.

Spring offers the following types of [collection configuration elements](http://examples.javacodegeeks.com/enterprise-java/spring/beans-spring/spring-collections-list-set-map-and-properties-example/):

* The <list> type is used for injecting a list of values, in the case that duplicates are allowed.
* The <set> type is used for wiring a set of values but without any duplicates.
* The <map> type is used to inject a collection of name-value pairs where name and value can be of any type.
* The <props> type can be used to inject a collection of name-value pairs where the name and value are both Strings.

<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-2.5.xsd">

<bean id="CustomerBean" class="com.mkyong.common.Customer">

<!-- java.util.List -->

<property name="lists">

<list>

<value>1</value>

<ref bean="PersonBean" />

<bean class="com.mkyong.common.Person">

<property name="name" value="mkyongList" />

<property name="address" value="address" />

<property name="age" value="28" />

</bean>

</list>

</property>

<!-- java.util.Set -->

<property name="sets">

<set>

<value>1</value>

<ref bean="PersonBean" />

<bean class="com.mkyong.common.Person">

<property name="name" value="mkyongSet" />

<property name="address" value="address" />

<property name="age" value="28" />

</bean>

</set>

</property>

<!-- java.util.Map -->

<property name="maps">

<map>

<entry key="Key 1" value="1" />

<entry key="Key 2" value-ref="PersonBean" />

<entry key="Key 3">

<bean class="com.mkyong.common.Person">

<property name="name" value="mkyongMap" />

<property name="address" value="address" />

<property name="age" value="28" />

</bean>

</entry>

</map>

</property>

<!-- java.util.Properties -->

<property name="pros">

<props>

<prop key="admin">admin@nospam.com</prop>

<prop key="support">support@nospam.com</prop>

</props>

</property>

</bean>

<bean id="PersonBean" class="com.mkyong.common.Person">

<property name="name" value="mkyong1" />

<property name="address" value="address 1" />

<property name="age" value="28" />

</bean>

</beans>

=====-------

The autowiring functionality has five modes which can be used to instruct Spring container to use autowiring for dependency injection:

* **no:** This is default setting. Explicit bean reference should be used for wiring.
* **byName:** When autowiring byName, the Spring container looks at the properties of the beans on which autowireattribute is set to byName in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file.
* **byType:** When autowiring by datatype, the Spring container looks at the properties of the beans on which autowireattribute is set to byType in the XML configuration file. It then tries to match and wire a property if its type matches with exactly one of the beans name in configuration file. If more than one such beans exist, a fatal exception is thrown.
* **constructor:** This mode is similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.
* **autodetect:**Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire bybyType.

Limitations of autowiring are:

* **Overriding:**You can still specify dependencies using <constructor-arg> and <property> settings which will always override autowiring.
* **Primitive data types:** You cannot autowire simple properties such as primitives, Strings, and Classes.
* **Confusing nature:** Autowiring is less exact than explicit wiring, so if possible prefer using explicit wiring.

### @Qualifier annotation

When there are more than one beans of the same type and only one is needed to be wired with a property, the@Qualifier annotation is used along with @Autowired annotation to remove the confusion by specifying which exact bean will be wired.

**@Inject**

Assuming here you're referring to the [javax.inject.Inject](http://download.oracle.com/javaee/6/api/javax/inject/Inject.html) annotations. @Inject is part of the Java CDI standard introduced in Java EE 6 (JSR-299), [read more](http://www.oracle.com/technetwork/articles/javaee/javaee6overview-141808.html). Spring has chosen to support using @Inject synonymously with their own @Autowired annotation.

So, to answer your question, @Autowired is Spring's own (legacy) annotation. @Inject is part of a new Java technology called CDI that defines a standard for dependency injection similar to Spring. In a Spring application, the two annotations works the same way as Spring has decided to support some JSR-299 annotations in addition to their own

-----\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#### 3.3.3.6. The ref element

The ref element is the final element allowed inside a property definition element. It is used to set the value of the specified property to be a reference to another bean managed by the container, a collaborator, so to speak. As mentioned in a previous section, the referred-to bean is considered to be a dependency of the bean who's property is being set, and will be initialized on demand as needed (if it is a singleton bean it may have already been initialized by the container) before the property is set. All references are ultimately just a reference to another object, but there are 3 variations on how the id/name of the other object may be specified, which determines how scoping and validation is handled.

Specifying the target bean by using the bean attribute of the ref tag is the most general form, and will allow creating a reference to any bean in the same BeanFactory/ApplicationContext (whether or not in the same XML file), or parent BeanFactory/ApplicationContext. The value of the bean attribute may be the same as either the id attribute of the target bean, or one of the values in the name attribute of the target bean.

<ref bean="someBean"/>

Specifying the target bean by using the local attribute leverages the ability of the XML parser to validate XML id references within the same file. The value of the local attribute must be the same as the idattribute of the target bean. The XML parser will issue an error if no matching element is found in the same file. As such, using the local variant is the best choice (in order to know about errors are early as possible) if the target bean is in the same XML file.

<ref local="someBean"/>

Specifying the target bean by using the parent attribute allows a reference to be created to a bean which is in a parent BeanFactory (or ApplicationContext) of the current BeanFactory (or ApplicationContext). The value of the parent attribute may be the same as either the id attribute of the target bean, or one of the values in the name attribute of the target bean, and the target bean must be in a parent BeanFactory or ApplicationContext to the current one. The main use of this bean reference variant is when there is a need to wrap an existing bean in a parent context with some sort of proxy (which may have the same name as the parent), and needs the original object so it may wrap it.

<ref parent="someBean"/>

It is injected into the second bean definition by reference using the property element's ref attribute.

<property name="message" ref="springMessage" />

**10.1.1 AOP concepts**

Let us begin by defining some central AOP concepts and terminology. These terms are not Spring-specific…​ unfortunately, AOP terminology is not particularly intuitive; however, it would be even more confusing if Spring used its own terminology.

* *Aspect*: a modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented using regular classes (the [schema-based approach](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html#aop-schema)) or regular classes annotated with the @Aspectannotation (the [@AspectJ style](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html#aop-ataspectj)).
* *Join point*: a point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point *always*represents a method execution.
* *Advice*: action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice. (Advice types are discussed below.) Many AOP frameworks, including Spring, model an advice as an *interceptor*, maintaining a chain of interceptors *around* the join point.
* *Pointcut*: a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.
* *Introduction*: declaring additional methods or fields on behalf of a type. Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any advised object. For example, you could use an introduction to make a bean implement an IsModified interface, to simplify caching. (An introduction is known as an inter-type declaration in the AspectJ community.)
* *Target object*: object being advised by one or more aspects. Also referred to as the *advised* object. Since Spring AOP is implemented using runtime proxies, this object will always be a *proxied* object.
* *AOP proxy*: an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.
* *Weaving*: linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

Types of advice:

* *Before advice*: Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
* *After returning advice*: Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.
* *After throwing advice*: Advice to be executed if a method exits by throwing an exception.
* *After (finally) advice*: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
* *Around advice*: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

Around advice is the most general kind of advice. Since Spring AOP, like AspectJ, provides a full range of advice types, we recommend that you use the least powerful advice type that can implement the required behavior. For example, if you need only to update a cache with the return value of a method, you are better off implementing an after returning advice than an around advice, although an around advice can accomplish the same thing. Using the most specific advice type provides a simpler programming model with less potential for errors. For example, you do not need to invoke the proceed() method on the JoinPoint used for around advice, and hence cannot fail to invoke it.

## Spring Model View Controller (MVC)

### WebApplicationContext

The WebApplicationContext is an extension of the plain ApplicationContext that has some extra features necessary for web applications. It differs from a normal ApplicationContext in that it is capable of resolving themes, and that it knows which servlet it is associated with.

## Difference between @Component, @Controller, @Repository & @Service annotations?

* 1. The @Component annotation marks a java class as a bean so the component-scanning mechanism of spring can pick it up and pull it into the application context. To use this annotation, apply it over class as below:

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

|  |
| --- |
| @Component  public class EmployeeDAOImpl implements EmployeeDAO {      ...  } |

2) The @Repository annotation is a specialization of the @Component annotation with similar use and functionality. In addition to importing the DAOs into the DI container, it also makes the unchecked exceptions (thrown from DAO methods) eligible for translation into Spring DataAccessException.

3) The @Service annotation is also a specialization of the component annotation. It doesn’t currently provide any additional behavior over the @Component annotation, but it’s a good idea to use @Service over @Component in service-layer classes because it specifies intent better.

4) @Controller annotation marks a class as a Spring Web MVC controller. It too is a @Component specialization, so beans marked with it are automatically imported into the DI container. When you add the @Controller annotation to a class, you can use another annotation i.e. @RequestMapping; to map URLs to instance methods of a class.

| Annotation | Meaning |

+------------+-----------------------------------------------------+

| @Component | generic stereotype for any Spring-managed component |

| @Repository| stereotype for persistence layer |

| @Service | stereotype for service layer |

| @Controller| stereotype for presentation layer (spring-mvc) |

----------------\*\*\*\*\*\*\*\*\*\*\*\*\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

@Resource vs@inject vs @autowired

It is very common confusion among the spring developers that what is the real **difference between these three (@Resource, @Autowired and @Inject) annotations used for injecting the objects**. I have come across this question from our readers so dedicated this post for explaining the main difference between these three annotations. Infact, these three work very much similar in most of the cases, there is slight differnce in few cases. I would explain that in this post. lets start from the basic details about these three annotations.

1. **@Resource** – Defined in the javax.annotation package and part of Java
2. **@Inject** – Defined in the javax.inject package and part of Java
3. **@Autowired** – Defined in the package org.springframework.bean.factory and part of Spring framework.

**@Autowired and @Inject**

1. Matches by Type
2. Restricts by Qualifiers
3. Matches by Name

**@Resource**

1. Matches by Name
2. Matches by Type
3. Restricts by Qualifiers (ignored if match is found by name)

The difference between @Inject vs. @Autowire vs. @Resource?

**@Autowired:** spring propriety annotation (as opposed to @Inject and @Resource) that inject a resource by-type, i.e. by the class of by the interface of the annotated field or contractor. In case we have few implementation of an interface or a subclass we can narrow down the selection using the @Qualifier annotation to avoid ambiguity. For a fallback match, the bean name is considered a default qualifier value. Although you can use this convention to refer to specific beans by name, @Autowired is fundamentally about type-driven injection with optional semantic qualifiers.

**@Inject:** Annotation based on JSR-330 (Dependency Injection for Java) identifies injectable constructors, methods, and fields. This annotation is an almost complete drop-in replacement for Spring’s @Autowired annotation. So, instead of using the Spring-specific @Autowired annotation, you might choose to use @Inject. One of the differences between @Autowired and @Inject is that @Inject does not have the required field so in case we fail to find a suitable object to injected it will fail while @Autowired can used required=false and allow null able field (only if required!). Advantage of @Inject annotation is that rather than inject a reference directly, you could ask @Inject to inject a Provider. The Provider interface enables, among other things, lazy injection of bean references and injection of multiple instances of a bean. In case we have few implementation of an interface or a subclass we can narrow down the selection using the @Named annotation to avoid ambiguity. @Named annotation works much like Spring’s @Qualifier

**@Resource:** annotation based on JSR-250. @Resource is quite similar to @Autowired and @Inject, but the main difference is the execution paths taken to find out the required bean to inject. @Resource will narrow down the search first by name then by type and finally by Qualifiers (ignored if match is found by name). @Autowired and @Inject will narrow down the search first by type then by qualifier and finally by the name.

## ------------------------------------\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-----------

## What is Spring MVC Interceptor and how to use it?

As you know about servlet filters that they can pre-handle and post-handle every web request they serve — before and after it’s handled by that servlet. In the similar way, you can use HandlerInterceptor interface in your spring mvc application **to pre-handle and post-handle web requests** that are handled by Spring MVC controllers. These handlers are mostly used to manipulate the model attributes returned/submitted they are passed to the views/controllers.

A handler interceptor can be registered for particular URL mappings, so it only intercepts requests mapped to certain URLs. Each handler interceptor must implement the HandlerInterceptor interface, which contains three callback methods for you to implement: preHandle(), postHandle() and afterCompletion().

Problem with HandlerInterceptor interface is that your new class will have to implement all three methods irrespective of whether it is needed or not. To avoid overriding, you can use HandlerInterceptorAdapter class. This class implementsHandlerInterceptor and provide default blank implementations.

Can you explain the concept of Interceptors in Spring MVC?

Handler interceptors are used when you want to apply specific functionality to certain requests. Handler Interceptors should implement the interface HandlerInterceptor.

Three methods are defined:

* preHandle(..) is called before the actual handler is executed;
* postHandle(..) is called after the handler is executed;
* afterCompletion(..) is called after the complete request has finished.

Interceptors can be configured using the interceptors property.

<bean id="handlerMapping" class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping">

<property name="interceptors">

<list>

<ref bean="yourCustomHandlerInterceptor"/>

</list>

</property>

</bean>

Spring Default Handler Mappings :

org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping

## BeanNameUrlHandlerMapping

This is the default implementation used by the [DispatcherServlet](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html), along with [DefaultAnnotationHandlerMapping](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/mvc/annotation/DefaultAnnotationHandlerMapping.html) (on Java 5 and higher). Alternatively, [SimpleUrlHandlerMapping](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/handler/SimpleUrlHandlerMapping.html) allows for customizing a handler mapping declaratively.

<bean

class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

<bean name="/welcome.htm"

class="com.mkyong.common.controller.WelcomeController" />

In above example, If URI pattern

1. **/welcome.htm** is requested, DispatcherServlet will forward the request to the “WelcomeController“.

The mapping is from URL to bean name. Thus an incoming URL "/foo" would map to a handler named "/foo", or to "/foo /foo2" in case of multiple mappings to a single handler. Note: In XML definitions, you'll need to use an alias name="/foo" in the bean definition, as the XML id may not contain slashes.

Supports direct matches (given "/test" -> registered "/test") and "\*" matches (given "/test" -> registered "/t\*"). Note that the default is to map within the current servlet mapping if applicable; see the["alwaysUseFullPath"](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/handler/AbstractHandlerMapping.html#setAlwaysUseFullPath-boolean-) property for details. For details on the pattern options, see the [AntPathMatcher](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/util/AntPathMatcher.html) javadoc.

DefaultAnnotationHandlerMapping

Registered by default in [DispatcherServlet](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html) on Java 5+. **NOTE:** If you define custom HandlerMapping beans in your DispatcherServlet context, you need to add a DefaultAnnotationHandlerMapping bean explicitly, since custom HandlerMapping beans replace the default mapping strategies. Defining a DefaultAnnotationHandlerMapping also allows for registering custom interceptors:

<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">

<property name="interceptors">

...

</property>

</bean>

Annotated controllers are usually marked with the [Controller](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/Controller.html) stereotype at the type level. This is not strictly necessary when [RequestMapping](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html) is applied at the type level (since such a handler usually implements the [Controller](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/mvc/Controller.html) interface). However, [Controller](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/Controller.html) is required for detecting [RequestMapping](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html) annotations at the method level if [RequestMapping](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html) is not present at the type level.

**NOTE:** Method-level mappings are only allowed to narrow the mapping expressed at the class level (if any). HTTP paths need to uniquely map onto specific handler beans, with any given HTTP path only allowed to be mapped onto one specific handler bean (not spread across multiple handler beans). It is strongly recommended to co-locate related handler methods into the same bean.

The [AnnotationMethodHandlerAdapter](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/mvc/annotation/AnnotationMethodHandlerAdapter.html) is responsible for processing annotated handler methods, as mapped by this HandlerMapping. For [RequestMapping](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html) at the type level, specific HandlerAdapters such as[SimpleControllerHandlerAdapter](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/mvc/SimpleControllerHandlerAdapter.html) apply.

--------------^##################$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

### [What does mvc:annotation-driven do?](http://thespringthing.blogspot.in/2010/10/what-does-mvcannotation-driven-do.html)

The annotations based MVC was introduced to the framework in Spring 2.5. This model enables the developer to reuse any POJO as a controller and is very flexible with the handler signatures. The old controller hierarchy is deprecated as of Spring 3.0. It would be removed completely from the distribution in one of the future releases.  
  
  
<**mvc:annotation-driven**> tag should be added to your Web Application context XML. This tag defaults the basic components required for delegating the requests to your Controllers.

[?](http://thespringthing.blogspot.in/2010/10/what-does-mvcannotation-driven-do.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"      xmlns:mvc="http://www.springframework.org/schema/mvc"      xsi:schemaLocation="http://www.springframework.org/schema/beans          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd                  http://www.springframework.org/schema/mvc          http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">       <mvc:annotation-driven  />   </beans> |

If this tag is not added to the XML, then you will have to manually define the beans for components like HandlerAdapter, HandlerMapping, Binding Initializer, Request Message converters, etc. This tag helps registering the following components.

1. **DefaultAnnotationHandlerMapping** - This is a HandlerMapping implementation which maps the HTTP requests to the handler methods defined using the @RequestMapping annotation.
2. **AnnotationMethodHandlerAdapter** - It is responsible for scanning the controllers to identify methods (and parameters) annotated with @MVC annotations. It scans and caches handler methods annotated with @RequestMapping. Also handles the @RequestParam, @ModelAttribute, @SessionAttributes and @InitBinder annotations.
3. **ConfigurableWebBindingInitializer** - The initializer for the Web Data Binder. Helps in declaratively configuring the Web Binder with validators, conversion services, property editors, etc.
4. **LocalValidatorFactoryBean** - Implements the validator interface and enables JSR303 validation. This is injected into ConfigurableWebBindingInitializer.
5. **FormattingConversionServiceFactoryBean** - A conversion factory that returns conversion services for basic objects like date and numbers. This factory is again injected into ConfigurableWebBindingInitializer.
6. Message Converters
   * **ByteArrayHttpMessageConverter** - A HTTP request message converter that reads a HTTP message body and returns a byte stream. It can also read a byte stream and construct a response body. Used for receiving and sending documents like PDF, XLS, etc.
   * **StringHttpMessageConverter** - A HTTP request message converter that reads a plain text request body and binds it to a String object. And vice-versa with response.
   * **FormHttpMessageConverter** - A HTTP request message converter that reads a form encoded request body and binds it to a form Binding object.
   * **SourceHttpMessageConverter** - A HTTP request converter that converts a XML message body to/from Binding Object.

If these beans are defined in the XML instead of using <**mvc:annotation-driven**>, it would look something like this.

[?](http://thespringthing.blogspot.in/2010/10/what-does-mvcannotation-driven-do.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | <beans xmlns:context="http://www.springframework.org/schema/context"  xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://www.springframework.org/schema/beans" xsi:schemalocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd          http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd                  http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">       <bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">    <property name="order" value="0">   </property></bean>     <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">    <property name="webBindingInitializer">     <bean class="org.springframework.web.bind.support.ConfigurableWebBindingInitializer">      <property name="validator" ref="validator">     </property></bean>    </property>    <property name="messageConverters">     <list>      <bean class="org.springframework.http.converter.ByteArrayHttpMessageConverter">      <bean class="org.springframework.http.converter.StringHttpMessageConverter">      <bean class="org.springframework.http.converter.FormHttpMessageConverter">      <bean class="org.springframework.http.converter.xml.SourceHttpMessageConverter">     </bean></bean></bean></bean></list>    </property>   </bean>   <bean class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean" id="validator">   <bean class="org.springframework.format.support.FormattingConversionServiceFactoryBean" id="conversion-service"  </bean></bean></beans> |

##################@@@@@@@@@@@@@@@@@@%%%%%%%%%%%%%%%%%%

# Content Negotiation using Spring MVC

 ENGINEERING

http://gravatar.com/avatar/6f858094f3a4f964dfe04598f038bcf1?s=20&d=mm [PAUL CHAPMAN](http://spring.io/team/pchapman)

 MAY 11, 2013

[18 COMMENTS](http://spring.io/blog/2013/05/11/content-negotiation-using-spring-mvc#disqus_thread)

There are two ways to generate output using Spring MVC:

* You can use the RESTful @ResponseBody approach and HTTP message converters, typically to return data-formats like JSON or XML. Programmatic clients, mobile apps and AJAX enabled browsers are the usual clients.
* Alternatively you may use view resolution. Although views are perfectly capable of generating JSON and XML if you wish (more on that in my next post), views are normally used to generate presentation formats like HTML for a traditional web-application.
* Actually there is a third possibility - some applications require both, and Spring MVC supports such combinations easily. We will come back to that right at the end.

In either case you’ll need to deal with multiple representations (or views) of the same data returned by the controller. Working out which data format to return is called Content Negotiation.

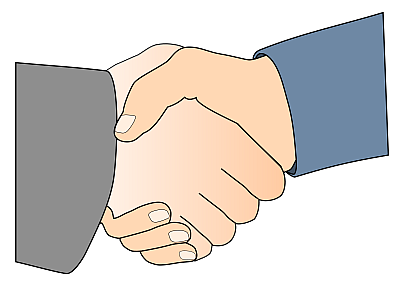
There are three situations where we need to know what type of data-format to send in the HTTP response:

* **HttpMessageConverters:** Determine the right converter to use.
* **Request Mappings:** Map an incoming HTTP request to different methods that return different formats.
* **View Resolution:** Pick the right view to use.

Determining what format the user has requested relies on a ContentNegotationStrategy. There are default implementations available out of the box, but you can also implement your own if you wish.

In this post I want to discuss how to configure and use content negotiation with Spring, mostly in terms of RESTful Controllers using HTTP message converters. In a later [post](http://blog.springsource.org/2013/06/03/content-negotiation-using-views/) I will show how to setup content negotiation specifically for use with views using Spring’sContentNegotiatingViewResolver.

## How does Content Negotiation Work?

**[caption id=“attachment\_13288” align=“alignleft” width=“200” caption=“Getting the Right Content”][/caption]**

When making a request via HTTP it is possible to specify what type of response you would like by setting the Accept header property. Web browsers have this preset to request HTML (among other things). In fact, if you look, you will see that browsers actually send very confusing Accept headers, which makes relying on them impractical. See<http://www.gethifi.com/blog/browser-rest-http-accept-headers> for a nice discussion of this problem. Bottom-line: Accept headers are messed up and you can’t normally change them either (unless you use JavaScript and AJAX).

So, for those situations where the Accept header property is not desirable, Spring offers some conventions to use instead. (This was one of the nice changes in Spring 3.2 making a flexible content selection strategy available across all of Spring MVC not just when using views). You can configure a content negotiation strategy centrally once and it will apply wherever different formats (media types) need to be determined.

## Enabling Content Negotiation in Spring MVC

Spring supports a couple of conventions for selecting the format required: URL suffixes and/or a URL parameter. These work alongside the use of Accept headers. As a result, the content-type can be requested in any of three ways. By default they are checked in this order:

* Add a path extension (suffix) in the URL. So, if the incoming URL is something likehttp://myserver/myapp/accounts/list.html then HTML is required. For a spreadsheet the URL should be http://myserver/myapp/accounts/list.xls. The suffix to media-type mapping is automatically defined via the JavaBeans Activation Framework or JAF (soactivation.jar must be on the class path).
* A URL parameter like this: http://myserver/myapp/accounts/list?format=xls. The name of the parameter is format by default, but this may be changed. Using a parameter is disabled by default, but when enabled, it is checked second.
* Finally the Accept HTTP header property is checked. This is how HTTP is actually defined to work, but, as previously mentioned, it can be problematic to use.

The Java Configuration to set this up, looks like this. Simply customize the predefined content negotiation manager via its configurer. Note the MediaType helper class has predefined constants for most well-known media-types.

@Configuration

@EnableWebMvc

public class WebConfig extends WebMvcConfigurerAdapter {

/\*\*

\* Setup a simple strategy: use all the defaults and return XML by default when not sure.

\*/

@Override

public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {

configurer.defaultContentType(MediaType.APPLICATION\_XML);

}

}

When using XML configuration, the content negotiation strategy is most easily setup via theContentNegotiationManagerFactoryBean:

<!--

Setup a simple strategy:

1. Take all the defaults.

2. Return XML by default when not sure.

-->

<bean id="contentNegotiationManager"

class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">

<property name="defaultContentType" value="application/xml" />

</bean>

<!-- Make this available across all of Spring MVC -->

<mvc:annotation-driven content-negotiation-manager="contentNegotiationManager" />

The ContentNegotiationManager created by either setup is an implementation ofContentNegotationStrategy that implements the PPA Strategy (path extension, then parameter, then Accept header) described above.

## Additional Configuration Options

In Java configuration, the strategy can be fully customized using methods on the configurer:

@Configuration

@EnableWebMvc

public class WebConfig extends WebMvcConfigurerAdapter {

/\*\*

\* Total customization - see below for explanation.

\*/

@Override

public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {

configurer.favorPathExtension(false).

favorParameter(true).

parameterName("mediaType").

ignoreAcceptHeader(true).

useJaf(false).

defaultContentType(MediaType.APPLICATION\_JSON).

mediaType("xml", MediaType.APPLICATION\_XML).

mediaType("json", MediaType.APPLICATION\_JSON);

}

}

In XML, the strategy can be configured using methods on the factory bean:

<!-- Total customization - see below for explanation. -->

<bean id="contentNegotiationManager"

class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">

<property name="favorPathExtension" value="false" />

<property name="favorParameter" value="true" />

<property name="parameterName" value="mediaType" />

<property name="ignoreAcceptHeader" value="true"/>

<property name="useJaf" value="false"/>

<property name="defaultContentType" value="application/json" />

<property name="mediaTypes">

<map>

<entry key="json" value="application/json" />

<entry key="xml" value="application/xml" />

</map>

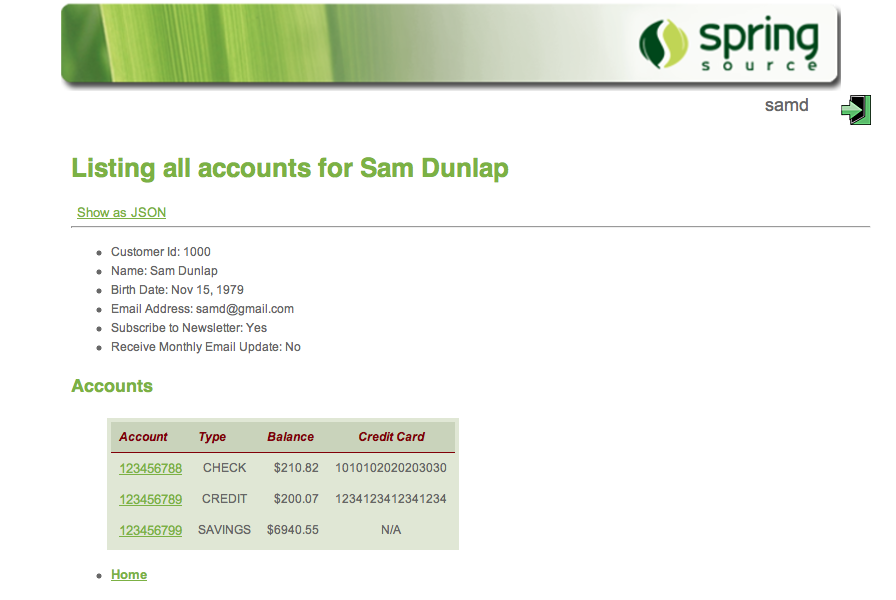
</property>

</bean>

What we did, in both cases:

* Disabled path extension. Note that favor does not mean use one approach in preference to another, it just enables or disables it. The order of checking is always path extension, parameter, Accept header.
* Enable the use of the URL parameter but instead of using the default parameter, format, we will use mediaType instead.
* Ignore the Accept header completely. This is often the best approach if most of your clients are actually web-browsers (typically making REST calls via AJAX).
* Don't use the JAF, instead specify the media type mappings manually - we only wish to support JSON and XML.

## Listing User Accounts Example

[](http://blog.springsource.org/wp-content/uploads/2013/05/account-list-1.png)

To demonstrate, I have put together a simple account listing application as our worked example - the screenshot shows a typical list of accounts in HTML. The complete code can be found at Github: <https://github.com/paulc4/mvc-content-neg>.

To return a list of accounts in JSON or XML, I need a Controller like this. We will ignore the HTML generating methods for now.

@Controller

class AccountController {

@RequestMapping(value="/accounts", method=RequestMethod.GET)

@ResponseStatus(HttpStatus.OK)

public @ResponseBody List<Account> list(Model model, Principal principal) {

return accountManager.getAccounts(principal) );

}

// Other methods ...

}

Here is the content-negotiation strategy setup:

<!-- Simple strategy: only path extension is taken into account -->

<bean id="cnManager"

class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">

<property name="favorPathExtension" value="true"/>

<property name="ignoreAcceptHeader" value="true" />

<property name="defaultContentType" value="text/html" />

<property name="useJaf" value="false"/>

<property name="mediaTypes">

<map>

<entry key="html" value="text/html" />

<entry key="json" value="application/json" />

<entry key="xml" value="application/xml" />

</map>

</property>

</bean>

Or, using Java Configuration, the code looks like this:

@Override

public void configureContentNegotiation(

ContentNegotiationConfigurer configurer) {

// Simple strategy: only path extension is taken into account

configurer.favorPathExtension(true).

ignoreAcceptHeader(true).

useJaf(false).

defaultContentType(MediaType.TEXT\_HTML).

mediaType("html", MediaType.TEXT\_HTML).

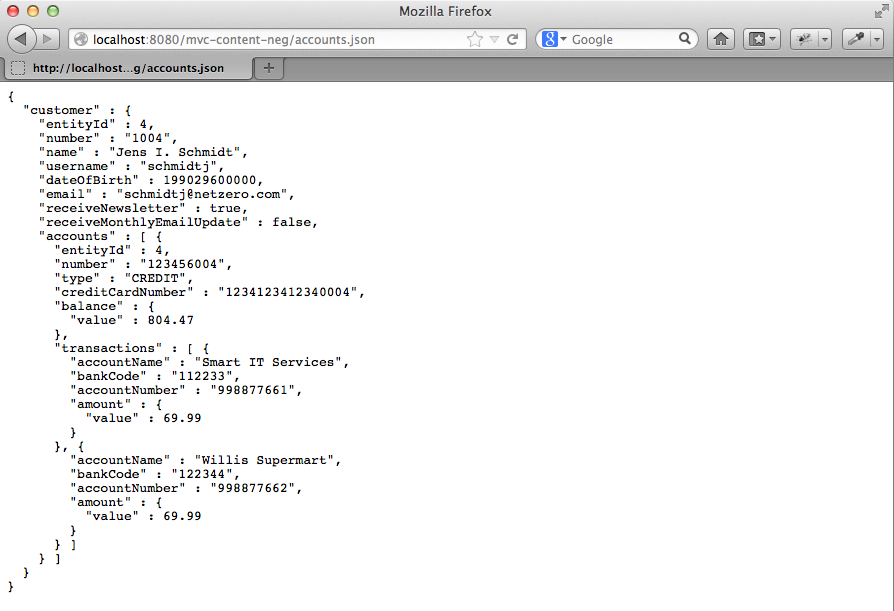
mediaType("xml", MediaType.APPLICATION\_XML).

mediaType("json", MediaType.APPLICATION\_JSON);

}

Provided I have JAXB2 and Jackson on my classpath, Spring MVC will automatically setup the necessary HttpMessageConverters. My domain classes must also be marked up with JAXB2 and Jackson annotations to enable conversion (otherwise the message converters don’t know what to do). In response to comments (below), the annotated Account class is shown [below](http://spring.io/blog/2013/05/11/content-negotiation-using-spring-mvc/#annotated-account-class).

Here is the JSON output from our Accounts application (note path-extension in URL).

[](http://blog.springsource.org/wp-content/uploads/2013/04/account-list-json.png)

How does the system know whether to convert to XML or JSON? Because of content negotiation - any one of the three (PPA Strategy) options discussed above will be used depending on how the ContentNegotiationManager is configured. In this case the URL ends in accounts.json because the path-extension is the only strategy enabled.

In the sample code you can switch between XML or Java Configuration of MVC by setting an active profile in the web.xml. The profiles are “xml” and “javaconfig” respectively.

## Combining Data and Presentation Formats

Spring MVC’s REST support builds on the existing MVC Controller framework. So it is possible to have the same web-applications return information both as raw data (like JSON) and using a presentation format (like HTML).

Both techniques can easily be used side by side in the same controller, like this:

@Controller

class AccountController {

// RESTful method

@RequestMapping(value="/accounts", produces={"application/xml", "application/json"})

@ResponseStatus(HttpStatus.OK)

public @ResponseBody List<Account> listWithMarshalling(Principal principal) {

return accountManager.getAccounts(principal);

}

// View-based method

@RequestMapping("/accounts")

public String listWithView(Model model, Principal principal) {

// Call RESTful method to avoid repeating account lookup logic

model.addAttribute( listWithMarshalling(principal) );

// Return the view to use for rendering the response

return ¨accounts/list¨;

}

}

There is a simple Pattern here: the @ResponseBody method handles all data access and integration with the underlying service layer (the AccountManager). The second method calls the first and sets up the response in the Model for use by a View. This avoids duplicated logic.

To determine which of the two @RequestMapping methods to pick, we are again using our PPA content negotiation strategy. It allows the produces option to work. URLs ending withaccounts.xml or accounts.json map to the first method, any other URLs ending inaccounts.anything map to the second.

## Another Approach

Alternatively we could do the whole thing with just one method if we used views to generate all possible content-types. This is where the ContentNegotiatingViewResolver comes in and that will be the subject of my next [post](http://blog.springsource.org/2013/06/03/content-negotiation-using-views/).

## Acknoweldgements

I would like to thank [Rossen Stoyanchev](http://spring.io/author/rstoyanchev/) for his help in writing this post. Any errors are my own.

## Addendum: The Annotated Account Class

**Added 2 June 2013**.

Since there were some questions on how to annotate a class for JAXB, here is part of the Account class. For brevity I have omitted the data-members, and all methods except the annotated getters. I could annotate the data-members directly if preferred (just like JPA annotations in fact). Remember that Jackson can marshal objects to JSON using these same annotations.

/\*\*

\* Represents an account for a member of a financial institution. An account has

\* zero or more {@link Transaction}s and belongs to a {@link Customer}. An aggregate entity.

\*/

@Entity

@Table(name = "T\_ACCOUNT")

@XmlRootElement

public class Account {

// data-members omitted ...

public Account(Customer owner, String number, String type) {

this.owner = owner;

this.number = number;

this.type = type;

}

/\*\*

\* Returns the number used to uniquely identify this account.

\*/

@XmlAttribute

public String getNumber() {

return number;

}

/\*\*

\* Get the account type.

\*

\* @return One of "CREDIT", "SAVINGS", "CHECK".

\*/

@XmlAttribute

public String getType() {

return type;

}

/\*\*

\* Get the credit-card, if any, associated with this account.

\*

\* @return The credit-card number or null if there isn't one.

\*/

@XmlAttribute

public String getCreditCardNumber() {

return StringUtils.hasText(creditCardNumber) ? creditCardNumber : null;

}

/\*\*

\* Get the balance of this account in local currency.

\*

\* @return Current account balance.

\*/

@XmlAttribute

public MonetaryAmount getBalance() {

return balance;

}

/\*\*

\* Returns a single account transaction. Callers should not attempt to hold

\* on or modify the returned object. This method should only be used

\* transitively; for example, called to facilitate reporting or testing.

\*

\* @param name

\* the name of the transaction account e.g "Fred Smith"

\* @return the beneficiary object

\*/

@XmlElement // Make these a nested <transactions> element

public Set<Transaction> getTransactions() {

return transactions;

}

// Setters and other methods ...

}

# Content Negotiation using Views

 ENGINEERING

http://gravatar.com/avatar/6f858094f3a4f964dfe04598f038bcf1?s=20&d=mm [PAUL CHAPMAN](http://spring.io/team/pchapman)

 JUNE 03, 2013

[3 COMMENTS](http://spring.io/blog/2013/06/03/content-negotiation-using-views#disqus_thread)

In my previous [post](http://blog.springsource.org/2013/05/11/content-negotiation-using-spring-mvc/) I introduced the concept of content negotiation and the three strategies Spring MVC uses to determine the content requested. In this post I want to extend the concept to supporting multiple views for different content-types using theContentNegotiatingViewResolver (or CNVR).

## Quick Overview

Since we already know how to setup content-negotiation from the previous [post](http://blog.springsource.org/2013/05/11/content-negotiation-using-spring-mvc/), using it to select between multiple views is very straightforward. Simply define a CNVR like this:

<!--

View resolver that delegates to other view resolvers based on the content type

-->

<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">

<!-- All configuration is now done by the manager - since Spring V3.2 -->

<property name="contentNegotiationManager" ref="cnManager"/>

</bean>

<!--

Setup a simple strategy:

1. Only path extension is taken into account, Accept headers are ignored.

2. Return HTML by default when not sure.

-->

<bean id="cnManager" class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">

<property name="ignoreAcceptHeader" value="true"/>

<property name="defaultContentType" value="text/html" />

</bean>

For every request, a @Controller would typically return a logical view name (or Spring MVC will determine one, by convention from the incoming URL). The CNVR will consult all the other view-resolvers defined in the configuration to see 1) if it has a view with the right name and 2) if it has a view that it also generates the right content - all Views ‘know’ what content-type they return. The desired content-type is determined in the exact same way discussed in the previous post.

For the equivalent Java configuration see [here](http://spring.io/blog/2013/06/03/content-negotiation-using-views/#javaconfig1). And for an extended configuration see [here](http://spring.io/blog/2013/06/03/content-negotiation-using-views/#config2). There is a demo application at Github: <https://github.com/paulc4/mvc-content-neg-views>.

For those of you in a hurry, that’s it in a nutshell.

For the rest of you, this post shows how we got to it. It discusses the concept of multiple-views in Spring MVC and builds upon that idea to define what the CNVR is, how to use it and how it works. It takes the same Accounts application from the previous [post](http://blog.springsource.org/2013/05/11/content-negotiation-using-spring-mvc/) and builds it up to return account information in HTML, as a Spreadsheet, as JSON and in XML. All using just views.

## Why Multiple Views?

One of the strengths of the MVC pattern is the ability to have multiple views for the same data.  In Spring MVC we achieve this using ’’Content Negotiation“”. My previous [post](http://blog.springsource.org/2013/05/11/content-negotiation-using-spring-mvc/)discussed content-negotiation in general and showed examples of RESTful controllers using HTTP Message Converters. But content-negotiation can also be used with Views as well.

For example, suppose I wish to display account information not just as a web-page, but also make it available as a spreadsheet too.  I could use a different URL for each, put two methods on my Spring controller and have each return the correct View type.  (BTW, if you aren´t sure how Spring can create a spreadsheet, I´ll show you that later).

@Controller

class AccountController {

@RequestMapping("/accounts.htm")

public String listAsHtml(Model model, Principal principal) {

model.addAttribute( accountManager.getAccounts(principal) ); // Duplicated logic

return ¨accounts/list¨; // View determined by view-resolution

}

@RequestMapping("/accounts.xls")

public AccountsExcelView listAsXls(Model model, Principal principal) {

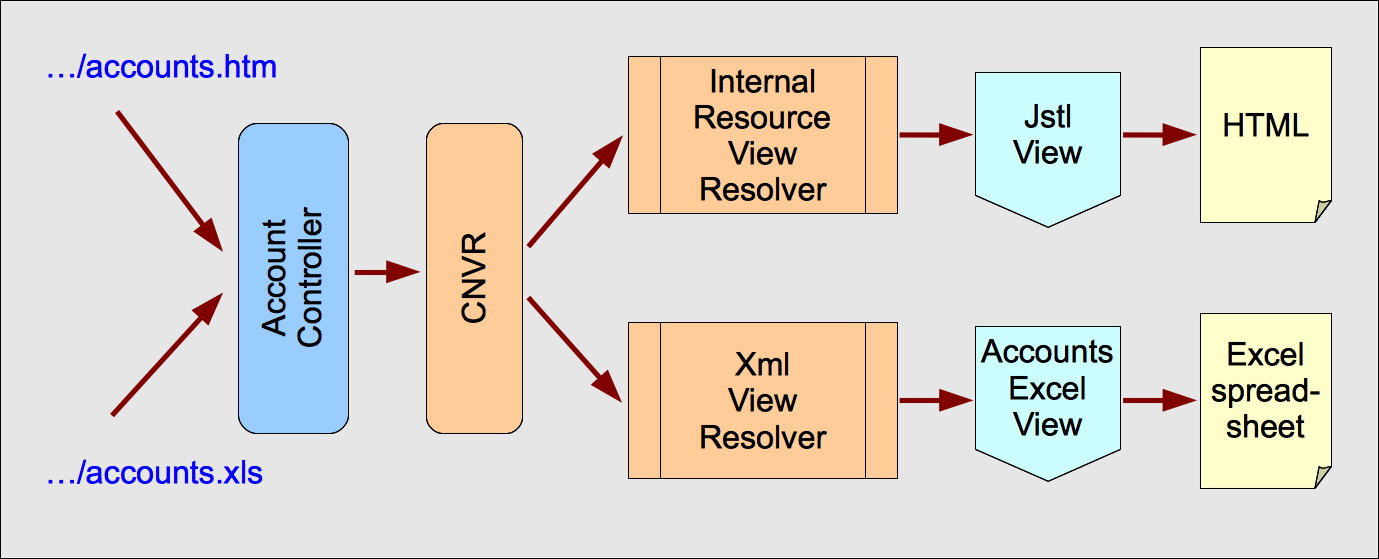
model.addAttribute( accountManager.getAccounts(principal) ); // Duplicated logic

return new AccountsExcelView(); // Return view explicitly

}

}

Using multiple methods is inelegant, defeats the MVC pattern and gets even uglier if I want to support other data formats too - such as PDF, CSV …  If you recall in the previous [post](http://blog.springsource.org/2013/05/11/content-negotiation-using-spring-mvc/) we had a similar problem wanting a single method to return JSON or XML (which we solved by returning a single @RequestBody object and picking the right HTTP Message Converter).

[caption id=“attachment\_13458” align=“alignleft” width=“380” caption=“Picking the right view via Content-Negotiation.”][](http://blog.springsource.org/wp-content/uploads/2013/06/cnvr-flow.png)[/caption]

Now we need a “smart” view resolver that picks the right View from multiple possible views.

Spring MVC has long supported multiple view resolvers, and goes to each in turn to find a view. Although the order that view resolvers are consulted can be specified, Spring MVC always picks the first view offered.  The ’‘Content Negotiating View Resolver’’ (CNVR) negotiates between all the view resolvers to find the best match for the format desired - this isour “smart” view resolver.

## Listing User Accounts Example

[](http://blog.springsource.org/wp-content/uploads/2013/05/acounts-html.png)

Here is a simple account listing application which we will use as our worked example to list accounts in HTML, in a spreadsheet and (later) in JSON and XML formats - just using views.

The complete code can be found at Github: <https://github.com/paulc4/mvc-content-neg-views>. It is a variation on the application I showed you last time that only uses views to generate output. **Note**: to keep the examples below simple I have used JSPs directly and anInternalResourceViewResolver. The Github project uses Tiles and JSPs because it’s easier than raw JSPs.

The screenshot of the accounts list HTML page shows all the accounts for the currently logged in user. You will see screenshots of the spreadsheet and JSON output later.

The Spring MVC controller that generated our page is below. Note that the HTML output is generated by the logical view accounts/list.

@Controller

class AccountController {

@RequestMapping("/accounts")

public String list(Model model, Principal principal) {

model.addAttribute( accountManager.getAccounts(principal) );

return ¨accounts/list¨;

}

}

To show two types of views we need two types of view resolver - one for HTML and one for the spreadsheet (to keep it simple, I will use a JSP for the HTML view). Here is the Java Configuration:

@Configuration

@EnableWebMvc

public class MvcConfiguration extends WebMvcConfigurerAdapter {

@Autowired

ServletContext servletContext;

// Will map to a bean called "accounts/list" in "spreadsheet-views.xml"

@Bean(name="excelViewResolver")

public ViewResolver getXmlViewResolver() {

XmlViewResolver resolver = new XmlViewResolver();

resolver.setLocation(new ServletContextResource(servletContext,

"/WEB-INF/spring/spreadsheet-views.xml"));

resolver.setOrder(1);

return resolver;

}

// Will map to the JSP page: "WEB-INF/views/accounts/list.jsp"

@Bean(name="jspViewResolver")

public ViewResolver getJspViewResolver() {

InternalResourceViewResolver resolver = new InternalResourceViewResolver();

resolver.setPrefix("WEB-INF/views");

resolver.setSuffix(".jsp");

resolver.setOrder(2);

return resolver;

}

}

Or in XML:

<!-- Maps to a bean called "accounts/list" in "spreadsheet-views.xml" -->

<bean class="org.springframework.web.servlet.view.XmlViewResolver">

<property name="order" value="1"/>

<property name="location" value="WEB-INF/spring/spreadsheet-views.xml"/>

</bean>

<!-- Maps to "WEB-INF/views/accounts/list.jsp" -->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="order" value="2"/>

<property name="prefix" value="WEB-INF/views"/>

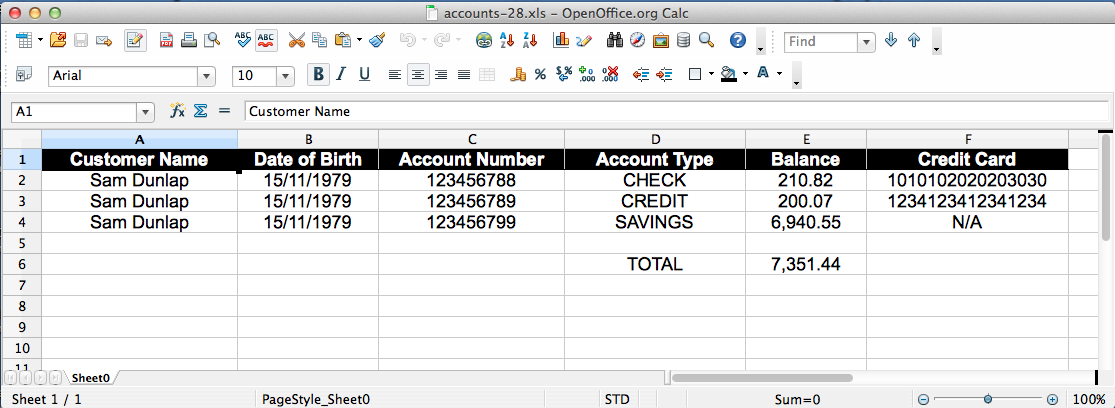
<property name="suffix" value=".jsp"/>

</bean>

And in WEB-INF/spring/spreadsheet-beans.xml you will find

<bean id="accounts/list" class="rewardsonline.accounts.AccountExcelView"/>

The generated spreadsheet looks like this:

[](http://blog.springsource.org/wp-content/uploads/2013/04/account-list-xls.png)

Here is how to create a spreadsheet using a view (this is a simplified version, the full implementation is much longer, but you get the idea):

class AccountExcelView extends AbstractExcelView {

@Override

protected void buildExcelDocument(Map<String, Object> model, HSSFWorkbook workbook,

HttpServletRequest request, HttpServletResponse response)

throws Exception {

List<Account> accounts = (List<Account>) model.get("accountList");

HSSFCellStyle dateStyle = workbook.createCellStyle();

dateStyle.setDataFormat(HSSFDataFormat.getBuiltinFormat("m/d/yy"));

HSSFSheet sheet = workbook.createSheet();

for (short i = 0; i < accounts.size(); i++) {

Account account = accounts.get(i);

HSSFRow row = sheet.createRow(i);

addStringCell(row, 0, account.getName());

addStringCell(row, 1, account.getNumber());

addDateCell(row, 2, account.getDateOfBirth(), dateStyle);

}

}

private HSSFCell addStringCell(HSSFRow row, int index, String value) {

HSSFCell cell = row.createCell((short) index);

cell.setCellValue(new HSSFRichTextString(value));

return cell;

}

private HSSFCell addDateCell(HSSFRow row, int index, Date date,

HSSFCellStyle dateStyle) {

HSSFCell cell = row.createCell((short) index);

cell.setCellValue(date);

cell.setCellStyle(dateStyle);

return cell;

}

}

## Adding Content Negotiation

As it currently stands this setup will always return the spreadsheet because theXmlViewResolver is consulted first (its order property is 1) and it always returns theAccountExcelView.  The InternalResourceViewResolver is never consulted (its order is 2 and we never get that far).

This is where the CNVR comes in. Let’s quickly review what we know about the content selection strategy discussed in the previous [post](http://blog.springsource.org/2013/05/11/content-negotiation-using-spring-mvc/). The requested content-type is determined by checking, in this order:

* A URL suffix (path extension) - for example http://...accounts.json to indicate JSON format.
* Or a URL parameter can be used. By default it is named format, for examplehttp://...accounts?format=json.
* Or the HTTP Accept header property will be used (which is actually how HTTP is defined to work, but is not always convenient to use - especially when the client is a browser).

In the first two cases the suffix or parameter value (xml, json …) must be mapped to the correct mime-type. Either the JavaBeans Activation Framework can be used or the mappings can be specified explicitly. With the Accept header property, its value is the mine-type.

## The Content Negotiating View Resolver

This is a special view resolver that has our strategy plugged into it. Here is the Java Configuration:

@Configuration

@EnableWebMvc

public class MvcConfiguration extends WebMvcConfigurerAdapter {

/\*\*

\* Setup a simple strategy:

\* 1. Only path extension is taken into account, Accept headers are ignored.

\* 2. Return HTML by default when not sure.

\*/

@Override

public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {

configurer.ignoreAcceptHeader(true)

.defaultContentType(MediaType.TEXT\_HTML);

}

/\*\*

\* Create the CNVR. Get Spring to inject the ContentNegotiationManager created by the

\* configurer (see previous method).

\*/

@Bean

public ViewResolver contentNegotiatingViewResolver(

ContentNegotiationManager manager) {

ContentNegotiatingViewResolver resolver = new ContentNegotiatingViewResolver();

resolver.setContentNegotiationManager(manager);

return resolver;

}

}

Or in XML:

<!--

View resolver that delegates to other view resolvers based on the content type

-->

<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">

<!-- All configuration is now done by the manager - since Spring V3.2 -->

<property name="contentNegotiationManager" ref="cnManager"/>

</bean>

<!--

Setup a simple strategy:

1. Only path extension is taken into account, Accept headers are ignored.

2. Return HTML by default when not sure.

-->

<bean id="cnManager" class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">

<property name="ignoreAcceptHeader" value="true"/>

<property name="defaultContentType" value="text/html" />

</bean>

The ContentNegotiationManager is exactly the same bean I discussed in the previous [post](http://blog.springsource.org/2013/05/11/content-negotiation-using-spring-mvc/).

The CNVR automatically goes to every other view resolver bean defined to Spring and asks it for a View instance corresponding to the view-name returned by the controller - in this caseaccounts/list.  Each View ‘knows’ what sort of content it can generate because there is agetContentType() method on it (inherited from the View interface).  The JSP page is rendered by a JstlView (returned by the InternalResourceViewResolver) and its content-type is text/html, whilst the AccountExcelView generates application/vnd.ms-excel.

How the CNVR is actually configured is delegated to the ContentNegotiationManager which is created in turn via the configurer (Java Configuration) or one of Spring’s many factory beans (XML).

The last piece of the puzzle is: how does the CNVR know what content-type was requested? Because the content-negotiation strategy tells it what to do: either a URL suffix is recognized, or a URL parameter or an Accept header. Exactly the same strategy setup described in the previous [post](http://blog.springsource.org/2013/05/11/content-negotiation-using-spring-mvc/), reused by the CNVR.

Note that when content-negotiation strategies were introduced by Spring 3.0 they only applied to selecting Views. Since 3.2 this facility is available across the board (as per my previous [post](http://blog.springsource.org/2013/05/11/content-negotiation-using-spring-mvc/)). The examples in this post use Spring 3.2 and may be different to older examples you have seen before. In particular most of the properties for configuring the content-negotiation strategy are now on the ContentNegotiationManagerFactoryBean and not on theContentNegotiatingViewResolver. The properties on the CNVR are now deprecated in favor of those on the manager but the CNVR itself works exactly the same way that it always did.

## Configuring the Content Negotiating View Resolver

By default the CNVR automatically detects all ViewResolvers defined to Spring and negotiates between them. If you prefer, the CNVR itself has a viewResolvers property so you can tell it explicitly which view resolvers to use. This makes it obvious that the CNVR is the master resolver and the others are subordinate to it. Note that the order property is no longer needed.

@Configuration

@EnableWebMvc

public class MvcConfiguration extends WebMvcConfigurerAdapter {

// .. Other methods/declarations

/\*\*

\* Create the CNVR. Specify the view resolvers to use explicitly. Get Spring to inject

\* the ContentNegotiationManager created by the configurer (see previous method).

\*/

@Bean

public ViewResolver contentNegotiatingViewResolver(

ContentNegotiationManager manager) {

// Define the view resolvers

List<ViewResolver> resolvers = new ArrayList<ViewResolver>();

XmlViewResolver r1 = new XmlViewResolver();

resolver.setLocation(new ServletContextResource(servletContext,

"/WEB-INF/spring/spreadsheet-views.xml"));

resolvers.add(r1);

InternalResourceViewResolver r2 = new InternalResourceViewResolver();

r2.setPrefix("WEB-INF/views");

r2.setSuffix(".jsp");

resolvers.add(r2);

// Create the CNVR plugging in the resolvers and the content-negotiation manager

ContentNegotiatingViewResolver resolver = new ContentNegotiatingViewResolver();

resolver.setViewResolvers(resolvers);

resolver.setContentNegotiationManager(manager);

return resolver;

}

}

Or in XML:

<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">

<property name="contentNegotiationManager" ref="cnManager"/>

<!-- Define the view resolvers explicitly -->

<property name="viewResolvers">

<list>

<bean class="org.springframework.web.servlet.view.XmlViewResolver">

<property name="location" value="spreadsheet-views.xml"/>

</bean>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="WEB-INF/views"/>

<property name="suffix" value=".jsp"/>

</bean>

</list>

</property>

</bean>

The Github demo project uses 2 sets of Spring profiles. In the web.xml, you can specify xmlor javaconfig for XML or Java configuration respectively. And for either of them, specify either separate or combined. The separate profile defines all view resolvers as top-level beans and lets the CNVR scan the context to find them (as discussed in the previous section). In the combined profile the view resolvers are defined explicitly, not as Spring beans and passed to the CNVR via its viewResolvers property (as shown in this section).

## JSON Support

Spring provides a MappingJacksonJsonView that supports the generation of JSON data from Java objects using the Jackson Object to JSON mapping library.  TheMappingJacksonJsonView automatically converts all attributes found in the Model to JSON.  The only exception is that it ignores BindingResult objects since these are internal to Spring MVC form-handling and not needed.

A suitable view resolver is needed and Spring doesn’t provide one.  Fortunately it is very simple to write your own:

public class JsonViewResolver implements ViewResolver {

/\*\*

\* Get the view to use.

\*

\* @return Always returns an instance of {@link MappingJacksonJsonView}.

\*/

@Override

public View resolveViewName(String viewName, Locale locale) throws Exception {

MappingJacksonJsonView view = new MappingJacksonJsonView();

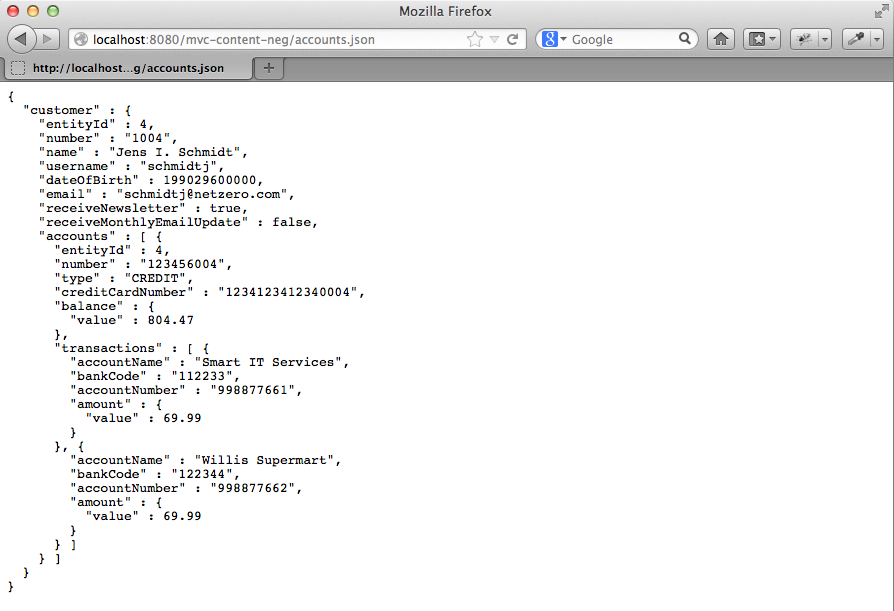
view.setPrettyPrint(true); // Lay the JSON out to be nicely readable

return view;

}

}

Simply declaring this view resolver as a Spring bean means JSON format data can be returned. The JAF already maps json to application/json so we are done. A URL like<http://myserver/myapp/accounts/list.json> can now return the account information in JSON. Here is the output from our Accounts application:

[](http://blog.springsource.org/wp-content/uploads/2013/04/account-list-json.png)

For more on this View, see the [Spring Javadoc](http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/web/servlet/view/json/MappingJacksonJsonView.html).

## XML Support

There is a similar class for generating XML output - the [MarshallingView](http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/web/servlet/view/xml/MarshallingView.html). It takes the first object in the model that can be marshalled and processes it. You can optionally configure the view by telling it which Model attribute (key) to pick - see setModelKey().

Again we need a view resolver for it. Spring supports several marshalling technologies via Spring’s [Object to XML Marshalling (OXM)](http://static.springsource.org/spring/docs/current/spring-framework-reference/html/oxm.html) abstraction. Let’s just use JAXB2 since it is built into the JDK (since JDK 6). Here is the resolver:

/\*\*

\* View resolver for returning XML in a view-based system.

\*/

public class MarshallingXmlViewResolver implements ViewResolver {

private Marshaller marshaller;

@Autowired

public MarshallingXmlViewResolver(Marshaller marshaller) {

this.marshaller = marshaller;

}

/\*\*

\* Get the view to use.

\*

\* @return Always returns an instance of {@link MappingJacksonJsonView}.

\*/

@Override

public View resolveViewName(String viewName, Locale locale)

throws Exception {

MarshallingView view = new MarshallingView();

view.setMarshaller(marshaller);

return view;

}

}

Again my classes need annotating to work with JAXB (in response to comments, I have added an example of this to the end of my previous [post](http://blog.springsource.org/2013/05/11/content-negotiation-using-spring-mvc/#annotated-account-class)).

Configure the new resolver as a Spring bean using Java Configuration:

@Bean(name = "marshallingXmlViewResolver")

public ViewResolver getMarshallingXmlViewResolver() {

Jaxb2Marshaller marshaller = new Jaxb2Marshaller();

// Define the classes to be marshalled - these must have @Xml... annotations on them

marshaller.setClassesToBeBound(Account.class, Transaction.class, Customer.class);

return new MarshallingXmlViewResolver(marshaller);

}

Or we can do the same thing in XML - note the use of the oxm namespace:

<oxm:jaxb2-marshaller id="marshaller" >

<oxm:class-to-be-bound name="rewardsonline.accounts.Account"/>

<oxm:class-to-be-bound name="rewardsonline.accounts.Customer"/>

<oxm:class-to-be-bound name="rewardsonline.accounts.Transaction"/>

</oxm:jaxb2-marshaller>

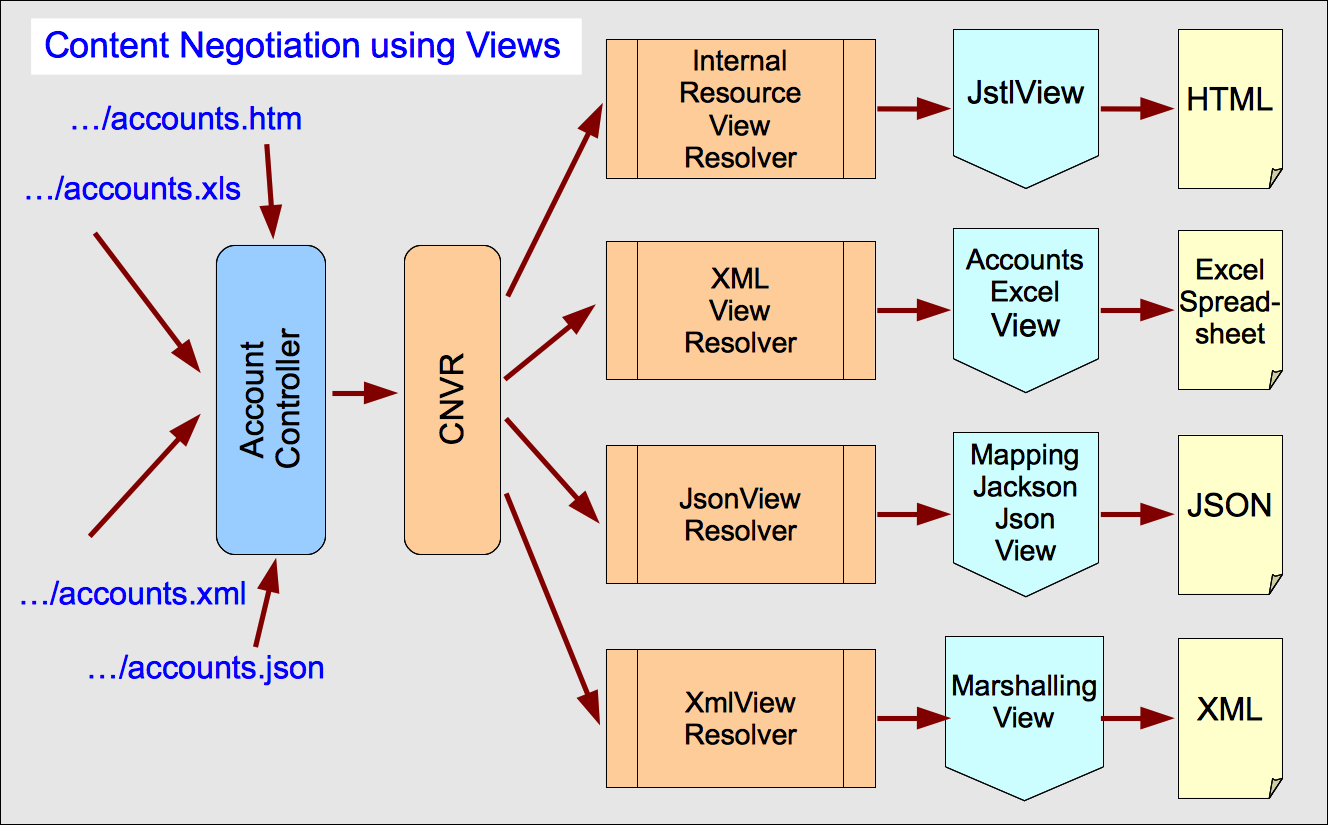
<!-- View resolver that returns an XML Marshalling view. -->

<bean class="rewardsonline.accounts.MarshallingXmlViewResolver" >

<constructor-arg ref="marshaller"/>

</bean>

This is our finished system:

[](http://blog.springsource.org/wp-content/uploads/2013/06/cnvr-all-flow.png)

## Comparing RESTful Approaches

Full support for a RESTful approach with MVC is available using @ResponseBody,@ResponseStatus and other REST related MVC annotations.  Something like this:

@RequestMapping(value="/accounts", produces={"application/json", "application/xml"})

@ResponseStatus(HttpStatus.OK)

public @ResponseBody List<Account> list(Principal principal) {

return accountManager.getAccounts(principal);

}

To enable the same content-negotiation for our @RequestMapping methods, we must reuse our content-negotiation manager (this allows the produces option to work).

<mvc:annotation-driven content-negotiation-manager="contentNegotiationManager" />

However this produces a different style of Controller method, the advantage being it is also more powerful. So which way to go: Views or @ResponseBody?

For an existing web-site already using Spring MVC and views, the MappingJacksonJsonViewand MarshallingView provide an easy way to extend the web-application to return JSON and/or XML as well.  In many cases, these are the only data-formats you need and is an easy way to support read-only mobile apps and/or AJAX enabled web-pages where RESTful requests are only used to GET data.

Full support for REST, including the ability to modify data, involves the use of annotated controller methods in conjunction with HTTP Message Converters. Using views in this case doesn’t make sense, just return a @ResponseBody object and let the converter do the work.

However, as shown <a href="[http://blog.springsource.org/2013/05/11/content-negotiation-using-spring-mvc/#combined-controller""](http://blog.springsource.org/2013/05/11/content-negotiation-using-spring-mvc/#combined-controller)here in my previous post, it is perfectly possible for a controller to use both approaches at the same time. Now the same controller can support both traditional web-applications and implement a full RESTful interface, enhancing web-applications that may have been built-up and developed over many years.

Spring has always been strong on offering developers flexibility and choice. This is no exception.

============00000000000-------------------#########################

# REST with Spring - ContentNegotiatingViewResolver vs. HttpMessageConverter+ResponseBody Annotation

[**[](https://dzone.com/users/183676/ghillert.html)**](https://dzone.com/users/183676/ghillert.html)**by**

[**Gunnar Hillert**](https://dzone.com/users/183676/ghillert.html)

**·**

**Dec. 30, 11 ·**[**Integration Zone**](https://dzone.com/enterprise-integration-training-tools-news)

**Like** [**(0)**](https://dzone.com/articles/java-forum-nord-2015-reflecting-software-architect)

**Comment (1)**

**Save**

[**Tweet**](https://dzone.com/articles/java-forum-nord-2015-reflecting-software-architect)

**15.01k Views**

*Download the*[*IDC Report: The Business Value of Red Hat Integration Products*](https://dzone.com/go?i=45227&u=http%3A%2F%2Fwww.redhat.com%2Ffiles%2Fresources%2Fen-rhjb-idc-business-value-253233.pdf%3Fsc_cid%3D70160000000x33KAAQ)*to learn more about Red hat Integration.*

**Disclaimer:**This blog posting reflects experiences while learning to implement RESTful services. Thus, certain elements of my blog posting may turn our to be not correct. Therefore, if you encounter errors, please let me know and I will post corrections as soon as possible. Please proceed with caution...   
  
As I posted in my previous blog "[Making DevNexus.com more Restful](http://hillert.blogspot.com/2011/01/making-devnexuscom-more-restful.html)", I am in the process of making more of the data from [DevNexus.com](http://www.devnexus.com/) consumable by other services, by exposing JSon and XML based endpoints. The website/application is implemented using [Spring MVC 3.0](http://www.springsource.org/documentation) in the view layer using Spring's REST support.  
  
Here are a few good reads that I came across that provide some helpful information:

1. <http://blog.springsource.com/2009/03/08/rest-in-spring-3-mvc/>
2. <http://blog.springsource.com/2009/03/16/adding-an-atom-view-to-an-application-using-springs-rest-support/>
3. <http://rwehner.wordpress.com/2010/06/09/2-ways-to-create-json-response-for-ajax-request-in-spring3/>
4. <http://www.ibm.com/developerworks/web/library/wa-restful/>
5. <http://forum.springframework.org/showthread.php?p=337206>
6. <http://static.springsource.org/spring/docs/3.0.x/reference/mvc.html>

As it turns out, there are 2 ways of implementing REST endpoints:

* use a [ContentNegotiatingViewResolver](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/servlet/view/ContentNegotiatingViewResolver.html)
* use [HttpMessageConverters](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/http/converter/HttpMessageConverter.html) in combination with the @ResponseBody annotation

**Using a ContentNegotiatingViewResolver**

When you use a ContentNegotiatingViewResolver your web controllers return ModelAndViews or view names and the ContentNegotiatingViewResolver will, based on various criteria, choose the right data representation strategy.

The highest priority hereby has the file extension which is used if available in the request. Next, the ViewResolver will look for a (definable) request parameter that identifies the view. If that does not help, the ViewResolver uses the Java Activation Framework to determine the Content-Type. If all fails, use the the HTTP Accept header. Of course the steps can be individually disabled.

A key takeaway though is, that your controllers will return a single ModelAndView/Viewname that will resolve into a specific view such as:

* org.springframework.web.servlet.view.json.MappingJacksonJsonView
* org.springframework.web.servlet.view.xml.MarshallingView
* org.springframework.web.servlet.view.documentClass.AbstractPdfView
* etc...

However, this may feel a little unnatural for certain data representations such as XML (using Jaxb annotations) or Json (using Jackson), where a dedicated view may not be necessary. Luckily, you can configure the ContentNegotiatingViewResolver to use default views which kind of solves the issue.

**Using HttpMessageConverters**

This is where HttpMessageConverters potentially could help. Whenever you use the**@ResponseBody**annotation you will be using a HttpMessageConverter (See also the Spring reference documentation, chapter "15.3.2.5 Mapping the request body with the @RequestBody annotation" and "18.3.2 HTTP Message Conversion").

What this means is, that instead of returning a [ModelAndView](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/servlet/ModelAndView.html) or view name, you will actually return data, e.g. a Collection of Objects.

If you use the Spring Context  **<mvc:annotation-driven/>**then support for XML and JSON marshalling is activated by default, provided the respective class libraries (Jaxb and/or Jackson) are present in your class-path (See the Spring documentation for details at chapter "15.12.1 mvc:annotation-driven")

Here is the interesting problem, if you use **<mvc:annotation-driven/>**you can set additional conversion services but you cannot set additional HttpMessageConverters. Consequently, if you want to do that then you have to use explicit bean declarations and remove the **<mvc:annotation-driven/>**tag from your context. This was something not too well covered in article [6]<http://www.ibm.com/developerworks/web/library/wa-restful/>

Ultimately, I am using the following bean declaration instead of the **<mvc:annotation-driven/>**  tag:

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">

<property name="webBindingInitializer">

<bean class="org.springframework.web.bind.support.ConfigurableWebBindingInitializer">

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

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

</bean>

</property>

<property name="messageConverters">

<list>

<ref bean="jsonConverter" />

<ref bean="marshallingConverter" />

<ref bean="atomConverter" />

</list>

</property>

</bean>

**My choice**

This was a bit tricky. Both approaches are somewhat overlapping and I wish the documentation would give you better guidance on which approach to use under which cicumstances. Certainly the ContentNegotiatingViewResolver option seems to be the better documented solution. On the other side, you don't need to configure explicit views when using HttpMessageConverters and the ResponseBody annotation and therefore, that setup looks a bit more streamlined.

One issue I was running into was, that for my application I can return both pure data views (Json, XML) and also Html/Jsp responses. Somehow I was not able to configure my controller easily+cleanly, to respond to the same Url with multiple controller methods (one using @ResponseBody and the other returning a JSP view)

Secondly, I wanted to also support file extensions to use the correct view or converter.  As it turns out though, HttpMessageConverters don't support that - Although there was an example somewhere for using request parameters. But that approach would require me creating additional custom classes...

Anyway, I went ahead chose the ContentNegotiatingViewResolver approach to implement my Restful services.

**Further issues**

Once I made my decision, everything seemed to go smoothly. I got my services implemented quickly and they worked perfectly in Firefox. Here is an example of the intended Url structure:

http://www.devnexus.com/s/speakers

http://www.devnexus.com/s/speakers.html

http://www.devnexus.com/s/speakers.xml

http://www.devnexus.com/s/speakers.json

With that I thought to have covered all bases: Support file extensions, but also allow clients to connect to <http://www.devnexus.com/s/speakers> and retrieve all data representations using the respective Http Accept header.

I deployed the app into production, but then the next day at the monthly Atlanta Users Group meeting - people informed me that the DevNexus site were down. That was odd, as I had accessed the site just minutes prior to the meeting.

Well, as it turned out, Google Chrome, Safari and Internet Explorer transmit a wild mixture of Http Accept headers. Consequently, when users accessed http://www.devnexus.com/s/index then the server would try to return an Xml view because Chrome and Safari requested Xml data rather than Html data.

For more details on Accept headers, please see the following fascinating:

<http://www.gethifi.com/blog/browser-rest-http-accept-headers>

What also threw me off, was that quite a few sources, incl. the Spring documentation imply that using the Http Accept header might be a actually a viable way of determining the correct view to return to clients.

To illustrate the chaos -Here is an interesting posting from the webkit mailing list:

<https://lists.webkit.org/pipermail/webkit-dev/2010-January/011188.html>

Ultimately, I got my servlet context configured in a way I think works best for me, though. The extension-less Url will always return Html now and for other data representations the file extension is mandatory. I also configured the ContentNegotiatingViewResolver to ignore Http Accept header by setting:

<property name="ignoreAcceptHeader" value="true" />

Many thanks to [this blog posting](http://www.rickherrick.com/?q=node/63) by Rick Herrick. Thus, now my servlet-context.xml file contains:

<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">

<property name="order" value="1" />

<property name="ignoreAcceptHeader" value="true" />

<property name="mediaTypes">

<map>

<entry key="xml" value="application/xml"/>

<entry key="json" value="application/json"/>

</map>

</property>

<property name="defaultViews">

<list>

<bean class="org.springframework.web.servlet.view.xml.MarshallingView">

<property name="marshaller" ref="jaxbMarshaller"/>

</bean>

<bean

class="org.springframework.web.servlet.view.json.MappingJacksonJsonView">

<property name="objectMapper" ref="jaxbJacksonObjectMapper"/>

</bean>

</list>

</property>

</bean>

<bean class="org.springframework.web.servlet.view.BeanNameViewResolver">

<property name="order" value="2"/>

</bean>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/jsp/"/>

<property name="suffix" value=".jsp"/>

<property name="order" value="3"/>

</bean>

That was a rather long day to get things working as intended but a good learning experience nonetheless. There is certainly something intriguing about having clean Urls.