

# Spring Core

Steps involved in the lifecycle of Spring bean

Instantiate – container creates an instance using Reflection API

Populate properties – simple properties & collaborating beans are resolved and set

setBeanName() – sets the bean name in the bean factory passing bean’s Id, if it implements BeanNameAware interface

setBeanFactory() – passes the beanFactory to the bean instance, if it implements BeanFactoryAware interface

Pre-Initialization – if there are any BeanPostProcessor associated with the BeanFactory, then postProcessBeforeInitialization() method will be invoked (even before properties are set)

Initialization –

If the bean implement InitializingBean interface then its overrided method afterPropertiesSet() will be invoked

If we got init-method attribute in the bean definition, then its value will be resolved to a method name in the Bean class

With JSR-250, @PostConstruct is considered the best approach to put up initialization block in place of above two approach (Since Spring 2.5)

NOTE: will see sequence for initialization, what if we got all three in the Bean class

Post-Initialization – Similar to Pre-Initialization step if there are any BeanPostProcessor associated with the BeanFactory, then postProcessAfterInitialization() method will be invoked

Ready to serve/use – in usable state

Destruction/Destroy –

If the bean implement DisposableBean interface then its overrided method destroy () will be invoked

If we got destroy-method attribute in the bean definition, then its value will be resolved to a method name in the Bean class

With JSR-250, @PreDestroy is considered the best approach to put up destruction/cleaning block in place of above two approach (Since Spring 2.5)

NOTE: We will see sequence for destroy, what if we got all three in the Bean class

The Sequence!!

Since Spring 2.5, we got three options for controlling the [lifecycle](http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/html/beans.html#beans-factory-lifecycle-combined-effects)behavior

Implementing InitializingBean & DisposableBean callback interface

Custom init() & destroy() methods; referred using bean attribute init-method and destroy-method

With JSR-250 specification, @PostContruct & @PreDestroy annotations can be used.

Now, let’s list down the sequence for Initialization & Destroy separately

Initialization

Method annotated with @PostConstruct

afterPropertiesSet() as defined by the InitializingBean callback interface

A custom configured init() method

Destroy

Methods annotated with @PreDestroy

destroy() as defined by the DisposableBean callback interface

A custom configured destroy() method

BeanPostProcessor

BeanPostPrecessor interface are responsible for the pre-initialization and post-initialization steps in the lifecycle of a Spring Bean. If you read through the API doc for BeanPostProcessor, it has two methods namely postProcessBeforeInitialization() & postProcessAfterInitialization() which are invoked if they are associated with the Spring IOC container’s BeanFactory.

These are basically to help you out to customize your logic overriding the default behavior of the IOC container like instantiation logic, dependency-resolution logic, and so forth. Generally, spring takes cares of the bean instantiation when required and manages its lifecycle and dispenses it upon completion of its work. If the requirement is to have more than one BeanPostProcessor, then developer should consider implementing ordered interface.

This BeanPostProcessor beans defined in the spring configuration xml are auto-detected and it’s applied to all the beans before they are created.

How to get ServletContext and ServletConfig object in a Spring Bean?

There are two ways to get Container specific objects in the spring bean.

Implementing Spring \*Aware interfaces, for these ServletContextAware and ServletConfigAware interfaces, for complete example of these aware interfaces, please read [Spring Aware Interfaces](https://www.journaldev.com/2637/spring-bean-life-cycle)

Using @Autowired annotation with bean variable of type ServletContext and ServletConfig. They will work only in servlet container specific environment only though.

@Autowired

ServletContext servletContext;

How you will decide when to use prototype scope and when singleton scope bean?

- You should use the prototype scope for all beans that are stateful and the singleton scope should be used for stateless beans.

There are only two states in a Stateless Session Bean life cycle .

1)Does not exist state

2)Ready for invocation from client.

The EJB container creates a pool of beans and does the necessary dependency injection. Then , if a method with **@PostConstruct** annotation is there ,it will be invoked.Now the bean is ready for client invocation.If a method annotated with**@PreDestroy** is there , then that method will be invoked at the end of client invocation. Now the bean is ready for garbage collection.

A stateless Session Bean , as the name implies , does not keep a conversational state with the calling client. In other words , if a stateless session bean has two business methods , and if a client is simultaneously accessing both the methods. Then we cannot guarantee that  the state of the bean remains the same. If another client is trying to access the bean , the container may give the existing instance to that client. So the state can be modified by the new client.

In case of a Stateful Session Bean , a conversational state is maintained between the client and the bean. In other words there will be separate bean instances for each client.So state for a client cannot be changed by a new client

**Explain inner beans in Spring.**

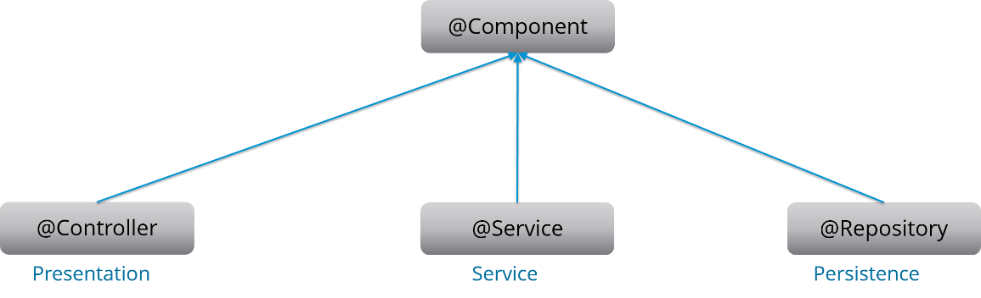
A bean can be declared as an inner bean only when it is used as a property of another bean. For defining a bean, the Spring’s XML based configuration metadata provides the use of <bean> element inside the <property> or <constructor-arg>. Inner beans are always anonymous and they are always scoped as prototypes. For example, let’s say we have one Student class having reference of Person class. Here we will be creating only one instance of Person class and use it inside Student.

Here’s a Student class followed by bean configuration file:

Student.java

studentbean.xml

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <bean id=“StudentBean" class="com.edureka.Student">  <property name="person">  <!--This is inner bean -->  <bean class="com.edureka.Person">  <property name="name" value=“Scott"></property>  <property name="address" value=“Bangalore"></property>  </bean>  </property>  </bean |

**What’s the difference between @Component, @Controller, @Repository & @Service annotations in Spring?**

**@Component:** This marks a java class as a bean. It is a generic stereotype for any Spring-managed component. The component-scanning mechanism of spring now can pick it up and pull it into the application context.

**@Controller:** This marks a class as a Spring Web MVC controller. Beans marked with it are automatically imported into the Dependency Injection container.

**@Service:** This annotation is a specialization of the component annotation. It doesn’t provide any additional behavior over the @Component annotation. You can use @Service over @Component in service-layer classes as it specifies intent in a better way.

**@Repository:** This annotation is a specialization of the @Component annotation with similar use and functionality. It provides additional benefits specifically for DAOs. It imports the DAOs into the DI container andmakes the unchecked exceptions eligible for translation into Spring DataAccessException.

**27. What do you understand by @Required annotation?**

@Required is applied to bean property setter methods. This annotation simply indicates that the affected bean property must be populated at the configuration time with the help of an explicit property value in a bean definition or with autowiring. If the affected bean property has not been populated, the container will throw BeanInitializationException.

BeanFactory v/s ApplicationContext in Spring

|  |  |
| --- | --- |
| **BeanFactory** **(org.springframework.beans)** | **ApplicationContext** **(org.springframework.context)** |
| Instantiates bean lazily i.e.; when ctx.getBean(“beanName”) is invoked from application Eg; BeanFactory bf = new XmlBeanFactory(“SpringXml”); | Instantiates bean eagerly i.e.; upon ApplicationContext startup/loaded. eg; ApplicationContext app = new ClassPathXmlApplicationContext(“SpringXml”); |
| No Supports for I18N. | Supports for Internalization (I18N) |
| No Supports for Annotation | Supports Annotation based Dependency Injection (DI) |
| Recommended to use in small lightweight application like Mobile, Applet, etc | AppplicationContext best suited for enterprise application |
| Very limited access i.e.; low level resources | It is very convenient to load resources from varios resources like ClassPath, FileSystem, etc. |
| No such support | Provides generic way to load resources such as Image file |
| Loading multiple configuration files is not possible | In large enterprise project, you have to load multiple configuration files. In that case, ApplicationContext is very good option Eg.; new ClassPathXmlApplicationContext(new String[]{“confg-1”, “confg-2”, “confg-3”}); |

Bean Scopes in Spring

|  |  |
| --- | --- |
| **Bean Scope** | **Explanation** |
| singleton | Default Returns single bean instance per IOC container |
| prototype | Returns new bean instance each time requested. |
| request | Returns single bean instance per HTTP request. Every new HTTP request will get new bean instance assigned and on completion bean will be out of scope and available for GC. |
| session | Returns single bean instance per HTTP session. |
| global- session | Returns single bean instance per HTTP global session. |

1. What Is a Circular Dependency?

It happens when a bean A depends on another bean B, and the bean B depends on the bean A as well:

Bean A → Bean B → Bean A

Of course, we could have more beans implied:

Bean A → Bean B → Bean C → Bean D → Bean E → Bean A

**What Happens in Spring**

When Spring context is loading all the beans, it tries to create beans in the order needed for them to work completely. For instance, if we didn’t have a circular dependency, like the following case:

Bean A → Bean B → Bean C

Spring will create bean C, then create bean B (and inject bean C into it), then create bean A (and inject bean B into it).

But, when having a circular dependency, Spring cannot decide which of the beans should be created first, since they depend on one another. In these cases, Spring will raise a BeanCurrentlyInCreationException while loading context.

It can happen in Spring when using **constructor injection**; if you use other types of injections you should not find this problem since the dependencies will be injected when they are needed and not on the context loading.

Solutions:

**1)Use**@Lazy

A simple way to break the cycle is saying Spring to initialize one of the beans lazily. That is: instead of fully initializing the bean, it will create a proxy to inject it into the other bean. The injected bean will only be fully created when it’s first needed.

**2)Use Setter/Field Injection**

One of the most popular workarounds, and also what [Spring documentation proposes](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html), is using setter injection.

Simply put if you change the ways your beans are wired to use setter injection (or field injection) instead of constructor injection – that does address the problem. This way Spring creates the beans, but the dependencies are not injected until they are needed.

Let’s do that – let’s change our classes to use setter injections and will add another field (message) to CircularDependencyB so we can make a proper unit test:

3) **Use**@PostConstruct

Another way to break the cycle is injecting a dependency using @Autowired on one of the beans, and then use a method annotated with @PostConstruct to set the other dependency.

**4)Implement**ApplicationContextAware**and**InitializingBean

If one of the beans implements ApplicationContextAware, the bean has access to Spring context and can extract the other bean from there. Implementing InitializingBean we indicate that this bean has to do some actions after all its properties have been set; in this case, we want to manually set our dependency.

@Override

    public void afterPropertiesSet() throws Exception {

        circB = context.getBean(CircularDependencyB.class);

    }

**In Conclusion**

There are many ways to deal with circular dependencies in Spring. The first thing to consider is to redesign your beans so there is no need for circular dependencies: they are usually a symptom of a design that can be improved.

But if you absolutely need to have circular dependencies in your project, you can follow some of the workarounds suggested here.

The preferred method is using setter injections. But there are other alternatives, generally based on stopping Spring from managing the initialization and injection of the beans, and doing that yourself using one strategy or another.

|  |  |  |
| --- | --- | --- |
| Collection Types | Element | Description |
| List | <list></list> | Helps in injecting list of values, allowing duplicates |
| Set | <set></set> | Helps in injecting set of values, but no duplicates are allowed here |
| Map | <map></map> | Helps in injecting name-value pairs of any type |
| Properties | <props></props> | Helps in injecting name-value pairs of only String type |
|  |  |  |

   private List<Object> listAddress;

    private Set<Object> setAddress;

    private Map<Object, Object> mapAddress;

    private Properties propsAddress;

Employee bean with four collection viz., <list>, <set>, <map>, <props>

Except <props> collection types, all other types got three values in the following ways

declaring values directly using ‘value’ attribute

referencing the ‘address’ bean using ‘ref’ attribute

inner bean concept for third value

For Properties, both name/value pairs are String, where we got location as keys and its OHIO cities are their values

<bean id="employee" class="com.spring.series.collections.Employee">

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

<property name="listAddress">

<list>

<value>ListAddress1</value>

<ref bean="address" />

<bean class="com.spring.series.collections.Address">

<property name="city" value="Kingsfordweg" />

<property name="state" value="Amsterdam" />

<property name="zipcode" value="1043" />

</bean>

</list>

</property>

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

<property name="setAddress">

<set>

<value>SetAddress1</value>

<ref bean="address" />

<bean class="com.spring.series.collections.Address">

<property name="city" value="Kingsfordweg" />

<property name="state" value="Amsterdam" />

<property name="zipcode" value="1043" />

</bean>

</set>

</property>

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

<property name="mapAddress">

<map>

<entry key="Map\_Key\_1" value="MapAddress1" />

<entry key="Map\_Key\_2" value-ref="address" />

<entry key="Map\_Key\_3">

<bean class="com.spring.series.collections.Address">

<property name="city" value="Kingsfordweg" />

<property name="state" value="Amsterdam" />

<property name="zipcode" value="1043" />

</bean>

</entry>

</map>

</property>

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

<property name="propsAddress">

<props>

<prop key="location1">Columbus</prop>

<prop key="location2">Delaware</prop>

<prop key="location3">Dublin</prop>

</props>

</property>

</bean>

<!-- address bean definition goes here -->

<bean id="address" class="com.spring.series.collections.Address">

<property name="city" value="Columbus" />

<property name="state" value="OHIO" />

<property name="zipcode" value="43211" />

</bean>

Ways to achieve Autowiring in Spring

XML based configuration

* Annotation

Autowiring modes

|  |  |
| --- | --- |
| **Mode** | **Explanation** |
| no | autowire=”no”Default behavior, this is what we have implemented in the earlier articles <link1> <link2>. Sometimes, it is called as explicit bean wiring. Use ‘ref’ attribute to wire the collaborating beans |
| byName | autowire=”byName”This option injects the collaborating beans using bean names defined in the Spring configuration file. If the beanName is found in the configuration file, then its get injected otherwise fatal error is raised. |
| byType | autowire=”byType”This option searches the collaborating beans using the bean types in the Spring configuration file. If the matching bean type is found in the configuration file, then its get injected otherwise a fatal error is raised.**Note:** if there are two similar bean types available in the configuration file, then it throws “NoUniqueBeanDefinitionException” excpetion. |
| constructor | autowire=”constructor”Similar to ‘byType’ but this applies to constructor arguments. If there isn’t one matching constructor, then a fatal error is raised. |
| auto-detect | autowire=”constructor”It first searches collaborating beans using ‘constructor’, if there isn’t one matching constructor available then it autowires using ‘byType’.Deprecated as of Spring 3.x [**see here**](http://docs.spring.io/spring/docs/3.0.x/javadoc-api/org/springframework/beans/factory/config/AutowireCapableBeanFactory.html#AUTOWIRE_AUTODETECT) |

Advantages of using Spring Autowiring

* Greatly reduces amount of code to be configured in the configuration XML file
* Use of <property> & <constructor-arg> tags can be eliminated.

Disadvantages

* Readability decreases, although it reduces amount of code in the configuration file
* Maintenance becomes difficult
* <property> & <constructor-arg> tags always overrides autowiring
* If there are two or more similar type beans defined in the spring configuration file, then while autowiring byType causes ambiguities and throws “NoUniqueBeanDefinitionException” exception
* Autowiring of simple types i.e.; primitives (int, float, double, etc) aren’t possible

**Conclusion:**Although, Spring autowiring concept greatly reduces the amount of code it causes ambiguity and readability. So, one should go for explicit bean wiring instead of autowiring in XML based configuration. But with Annotation, autowiring got simple improving the readability of the code.

**If an inner bean is defined with an id, can you use this id to fetch the bean from the container ?**  
No, An bean defined inner bean can't be accessed even if the id attribute has value. so getBean("theInnerId") will fail with NoSuchBeanDefinitionException.

 Annotation using @Qualifier

In this previous article, we have seen an example demonstrating how annotation can be used to inject dependencies. @Autowired annotation injects dependencies using byType and this could be problem when we got two beans with same data type & throws ‘NoUniqueBeanDefinitionException’ exception as we seen in the byType<link> example.

**How do you use a particular implementation of collection in your bean definition ?**   
You can use the <util:set> <util:list> and <util:map> with set-class to the implementation you want to use.For example <util:list set-class="java.util.LinkedList"> to use linkedList as implementation, and don't forget to include the schema details in the beans tag. Also util tag can let you create id of the collection , thus this can be refered or shared with any other beans by using the regular way , ie the ref tag.

The solution to above problem is, in addition to the @Autowired annotation we must include one more annotation called @Qualifier(“beanName”) to make aware the container as to which one of the two beans needs to be injected at the runtime.

**Note:** If there are two beans of same data type named ‘address’ & ‘addressB’ defined in the configuration file, then in this case spring container injects the first bean ‘address’ bean by default. This is because, container lowercase the first character of the collaborating bean.

For example,

if the dependent property is Address then it lowercase the first character and make it ‘address’ and injects it.

// property address autowired using annotation

    @Autowired

    @Qualifier("addressB")

    private Address address;

Spring Annotation using @Resource

In the previous 2 articles, we have explored about Spring annotation that how it helps to reduce the complex XML configuration to simply annotating component classes to take it effects. With @Autowired<link> example we have seen it works similar to ‘byType’ autowiring mode and with @Qualifier example we solved the ambiguity problems when we got two beans registered with same data type

So question arises here, does Spring support ‘byName’ autowiring mode similar to byType with @Autowired annotation? Yes, with @Resource you can do it

Let’s see detailed example based on this annotation @Resource

 // property address autowired using annotation

    @Resource(name="addressB")

    private Address address;

Spring Annotation using @Required

In this article, we will learn to enforce the spring container to inject certain properties into the bean otherwise asking to throw an error. With @Required annotation, container makes sure that certain property(properties) are injected into the bean.

**Conclusion:** With @Required annotation set, its the developer’s responsibility to make sure those dependencies are supplied. Otherwise ending up reading those dirty error trace in the console log

2. What are the common implementations of the Application Context ?

   The three commonly used implementation of 'Application Context' are

ClassPathXmlApplicationContext : It Loads context definition from an XML file located in the classpath, treating context definitions as classpath resources. The application context is loaded from the application's classpath by using the code .

ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");

FileSystemXmlApplicationContext : It loads context definition from an XML file in the filesystem. The application context is loaded from the file system by using the code .

ApplicationContext context = new FileSystemXmlApplicationContext("bean.xml");

XmlWebApplicationContext : It loads context definition from an XML file contained within a web application.

6. Which Dependency Inject would you suggest Constructor-based or setter-based?

It depends on the requirement. It is recommended to use constructor based DI for mandatory dependencies and setter based DI for optional dependencies. You can also use both Constructor-based and Setter-based Dependency Injection.

What does @Qualifier annotation mean?

[@Qualifier](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Qualifier.html) annotation is used to identify which bean to wire if there are more than one bean (configuration) of the same type when autowiring. To understand it, consider the following bean configuration,

|  |  |
| --- | --- |
| 1  2 | <bean id="student1" class="com.class.Student" />  <bean id="student2" class="com.class.Student" /> |

Here there are two beans of type Student.

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class Department {    @Autowired    private student student;    //...} |

When the above class is run, it will thrown an exception because it can’t find an unique bean of type Student. Hence in this case you can use @Qualifier to mention which bean definition to consider for autowire.

Here is the above example with @Qualifier used.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Department {  @Autowired  @Qualifier("student1")  private student student;  //...  } |

21. How is event handling done in Spring and what are the different types of events in spring framework?

Spring’s **ApplicationContext** provides the functionality to support events. We can create beans that listen for events which are published through the ApplicationContext. Event handling in the ApplicationContext is provided through the **ApplicationEvent** class and **ApplicationListener** interface. So if a bean implements the ApplicationListener interface, then every time an ApplicationEvent gets published to the ApplicationContext, the bean is notified. Spring provides the following events:

1. **ContextRefreshedEvent** : This event is published when the ApplicationContext is either initialized or refreshed. This can also be triggered using the refresh() method on the ConfigurableApplicationContext interface.
2. **ContextStartedEvent** : This event is published when the ApplicationContext is started using the start() method on the ConfigurableApplicationContext interface. You can poll your database or you can start/restart any stopped application after receiving this event.
3. **ContextStoppedEvent** : This event is published when the ApplicationContext is stopped using the stop() method on the ConfigurableApplicationContext interface. You can do any resource clean-up activity after receiving this event.
4. **ContextClosedEvent** : This event is published when the ApplicationContext is closed using the close() method on the ConfigurableApplicationContext interface. A closed context reaches its end of life; it cannot be refreshed or restarted.
5. **RequestHandledEvent** : This is a web-specific event telling all beans that an HTTP request has been serviced.

To listen a context event, a bean should implement the *ApplicationListener* interface which has just one method **onApplicationEvent()**.

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.

35. Can you inject null and empty string values in Spring?

Yes, you can.

# Spring MVC

@RequestMapping(value="/welcome/country/{username}")

**public** ModelAndView helloWorld( @PathVariable String username){

@RequestMapping(value="/welcome/{country}/{username}")

**public** ModelAndView helloWorld( @PathVariable Map<String,String> pathVars){

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

**public** ModelAndView helloWorld( @RequestParam(value="UserName",defaultValue="defaultvalue") String UserName){

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

**public** ModelAndView helloWorld(@Valid@ModelAttribute("student") Student student){}

@InitBinder

**public** **void** initBinder(WebDataBinder binder){

binder.setDisallowedFields(**new** String[]{"studentMobile"});//studentMobile will not bind with student class

//allow specific date format

SimpleDateFormat dateFormat=**new** SimpleDateFormat("yyyy\*\*\*dd\*\*\*MM");

binder.registerCustomEditor(Date.**class**, "studentDOB",**new** CustomDateEditor(dateFormat, **false**));

}

@ControlerAdvice//apply on top of the class exceptions.

@ResponseStatus(value=HttpStatus.***INTERNAL\_SERVER\_ERROR***)

@ExceptionHandler(value=Exception.**class**)

**public** String handleException(Exception e){

Through XML

<bean id=*"simpleMappingExceptionResolver"* class=*"org.sf.web.servlet.handler.SimpleMappingExceptionResolver"*>

<property name=*"exceptionMappings"*>

<map>

<entry key=*"NullPointerException"* value=*"ViewName"*></entry>

<entry key=*"IOEException"* value=*"ViewName"*></entry>

</map>

</property>

<property name=*"defaultErrorView"* value=*"ViewName"*></property>

</bean>

1) Benefits of Spring MVC over Struts?

1. We have clear separation between models, views and controllers in Spring.

2. It is very versatile and flexible based on interfaces but Struts forces Actions and Form object into concrete inheritance.

3. It provides both interceptors and controllers, thus helps to factor out common behavior to the handling of many requests.

4. It can be configured with different view technologies like Freemarker, JSP, Tiles, Velocity, XLST etc. and also you can create your own custom view mechanism by implementing Spring View interface.

5. In Spring MVC Controllers can be configured using DI (IOC) that makes its testing and integration easy.

6. Web tier of Spring MVC is easy to test than Struts web tier, because of the avoidance of forced concrete inheritance and explicit dependence of controllers on the dispatcher servlet.

7. Spring web layer built on top of a business object layer, which is considered a good practice. In Struts framework you need to implement your business objects.

8. Struts force your Controllers to extend a Struts class but Spring doesn’t, there are many convenience Controller implementations that you can choose to extend. 

# Spring AOP

AOP refers to Aspect Oriented Programming which behaves like OOPs as both provides the concept of modularity. But the difference is it uses aspect rather than class for the unit of modularity.  
Aspect Oriented Programming breaks down program logic into distinct parts called concerns. A cross-cutting concerns are aspects of a program that affect other concerns like transaction management, authentication, logging etc.

Spring AOP module provides the facility to add extra functionality before or after the method execution.  
Note: Aspect Oriented Programming AOP is like triggers in programming languages like java, Perl, .NET etc.

AOP Terminologies:

**Aspect:**  
An aspect represents a class that contains advices, join points etc like transaction management. An aspect can be configured through Spring XML configuration or spring AspectJ integration.  
   
**Join point:**  
Joint point represents a point in our application where we can plug-in AOP aspect. It can be method execution, exception handling, field access etc. Spring AOP only supports method execution joint type.  
   
**Advice:**  
It represents the actual action to be taken by an aspect at a particular join point. In programming point view it represents the methods to be executed at a particular join point.  
   
**Pointcut:**  
It represents the expression which is matched with join points to determine whether advice to be executed or not.  
   
**Introduction:**  
It provides the facility to add new methods or attributes to existing classes.  
   
**Target object:**  
It is an object on which advices are applied. It always be proxied object in spring because Spring AOP is implemented using runtime proxies.  
   
**Weaving:**  
Weaving is the process of linking aspects with other application types or objects to create the advised proxy objects. This can be done at compile time, load time or at runtime. Spring AOP performs weaving at the runtime.

AOP Advice Types:

**Before:**  
It executes before the method execution.   
**After:**  
It executes after the method execution. It not depends upon the method outcome.   
**after-returning:**  
It executes after the method execution when method completes successfully.  
   
**after-throwing:**  
It executes after the method execution when method exits by throwing an exception.   
**Around:**  
It executes before and after the advised method is called.  
 Declaring an aspect:

The @Aspect annotation is used to declare a class as an aspect.

|  |
| --- |
| @Aspect  **public** **class** AspectModule {    } |

Declaring a pointcut:

The @Pointcut annotation is used to declare a pointcut. The expression parameter represents the expression used for matching the join point.

|  |
| --- |
| @Pointcut("execution(expression)")  **private** **void** businessService() {  *//Block of code.*  } |

Declaring advices:

The @{ADVICE-NAME} annotations is used to declare an advice.

Advice types:

**Before:**  
It executes before the method execution.

**After:**  
It executes after the method execution. It not depends upon the method outcome.

**after-returning:**  
It executes after the method execution when method completes successfully.

**after-throwing:**  
It executes after the method execution when method exits by throwing an exception.

**Around:**  
It executes before and after the advised method is called.

Spring AOP AspectJ Annotation Configuration Advice Example:

**BusinessLogic.java-----------**

**AdviceTest.java**

|  |
| --- |
| @Aspect  public class AdviceTest {  @Pointcut("execution(\* com.javawithease.business.\*.\*(..))")  private void selectAll(){}  @Before("selectAll()")  public void beforeAdvice(){  System.out.println("Before advice executed.");  }  @After("selectAll()")  public void afterAdvice(){  System.out.println("After advice executed.");  }  @AfterReturning(pointcut = "selectAll()", returning="retVal")  public void afterReturningAdvice(Object retVal){  System.out.println("After returning advice executed.");  System.out.println("Returning value: " + retVal);  }  @AfterThrowing(pointcut = "selectAll()", throwing = "ex")  public void afterThrowingAdvice(ArithmeticException ex){  System.out.println("Throwing advice executed.");  System.out.println("Exception: " + ex.getMessage());  }} |

**applicationContext.java**

|  |
| --- |
| <aop:config>  <aop:aspect id="log" ref="adviceTest">  <aop:pointcut id="selectAll"  expression="execution(\* com.javawithease.business.\*.\*(..))"/>  <aop:before pointcut-ref="selectAll" method="beforeAdvice"/>  <aop:after pointcut-ref="selectAll" method="afterAdvice"/>  <aop:after-returning pointcut-ref="selectAll"  returning="retVal"  method="afterReturningAdvice"/>  <aop:after-throwing pointcut-ref="selectAll"  throwing="ex"  method="afterThrowingAdvice"/>  </aop:aspect>  </aop:config>    <bean id="businessLogic"  **class**="com.javawithease.business.BusinessLogic"/>  <bean id="adviceTest"  **class**="com.javawithease.business.AdviceTest"/> |

Name three typical cross cutting concerns?

Logging

Security

Transaction

. Limitations of Spring AOP?

Can only advise non-private methods

Can only apply aspects to Spring Beans

Limitations of weaving with proxies

When using proxies, suppose method a() calls method b() on the same class/interface

advice will never be executed for method b()

Spring AOP JoinPoint and Advice Arguments

We can use JoinPoint as parameter in the advice methods and using it get the method signature or the target object.

We can use args() expression in the pointcut to be applied to any method that matches the argument pattern. If we use this, then we need to use the same name in the advice method from where argument type is determined. We can use [Generic objects](https://www.journaldev.com/1663/java-generics-example-method-class-interface) also in the advice arguments.

@Aspect

public class EmployeeAspectJoinPoint {

@Before("execution(public void com.journaldev.spring.model..set\*(\*))")

public void loggingAdvice(JoinPoint joinPoint){

System.out.println("Before running loggingAdvice on method="+joinPoint.toString());

System.out.println("Agruments Passed=" + Arrays.toString(joinPoint.getArgs()));

}

//Advice arguments, will be applied to bean methods with single String argument

@Before("args(name)")

public void logStringArguments(String name){

System.out.println("String argument passed="+name);

}}

. Matching Method Signature Patterns

Matching all public methods in TransferService

Use public keyword in start, use \* to match any return type.

@Pointcut("execution(public \* com.doj.app.service.TransferService.\*(..))")

public void anyPublicMethodOfTrasferService();

Matching any public methods

Use public keyword in start, use \* to match any return type and use another \* to match any method name.

@Pointcut("execution(public \* \*(..))")

public void anyPublicMethod();

Matching any method defined in the service package

Use \* in start to match any return type, use second \* to match any class name and use another \* to match any method name.

@Pointcut("execution(\* com.doj.app.service.\*.\*(..))")

public void anyMethodInServicePackage();

Matching any method defined in the service package or it’s sub-packages

Use \* in start to match any return type, use two dots after service package means it’s include sub-packages as well, use second \* to match any class name and use another \* to match any method name.

@Pointcut("execution(\* com.doj.app.service..\*.\*(..))")

public void anyMethodInServicePackageAndSubPackage();

Matching all public methods in TransferService with return type Account

Use public keyword in start and use Account as a return type.

@Pointcut("execution(public Account com.doj.app.service.TransferService.\*(..))")

public void allPublicMethodOfTransferServiceReturnTypeAccount();

Matching all methods in TransferService with return type void and first parameter as Account

Use void keyword in start and use Account as a argument type for first parameter.

@Pointcut("execution(void com.doj.app.service.TransferService.\*(Account account, ..))")

public void allMethodOfTransferServiceVoidReturnTypeFirstArgumentAccount();

Matching all public methods in any class of the service package with any return type and method name should be transfer with taking two parameters of Account types

Use public keyword in start and use \* as any return type, use another \* for any class in the service package, use transfer method name with two parameters of Account Types.

@Pointcut("execution(public \* com.doj.app.service.\*.tranfer(Account account1, Account account2))")

public void allTranferMethodsInServicePackageWithTwoArgumentsOfAccountType();

2. Matching Type Signature Patterns

Method execution only in Spring AOP. It provides narrowed to matching all method executions within the certain types only.

Matching all methods defined in classes inside package com.doj.app.service

@Pointcut("within(com.doj.app.service.\*)")

public void allMethodsInServicePackage();

Matching all methods defined in classes inside package com.doj.app.service and it’s sub-packages

sub-packages use two dots.

@Pointcut("within(com.doj.app.service..\*)")

public void allMethodsInServicePackageAndSubPackages();

Matching all methods with a TransferService class

@Pointcut("within(com.doj.app.service.TransferService)")

public void allMethodsOfTransferService();

Matching all methods within all implementing classes of TransferService interface

Use + (plus) sign to match all implementations of an interface.

@Pointcut("within(com.doj.app.service.TransferService+)")

public void allMethodsOfTransferServiceImpl();

Matching all methods where the proxy implements the TransferService interface

@Pointcut("this(com.doj.app.service.TransferService)")

public void allMethodsProxyImplmentTransferService();

Matching all methods where the target object implements the TransferService interface

@Pointcut("target(com.doj.app.service.TransferService)")

public void allMethodsTargetObjectImplmentTransferService();

3. Matching Bean Name Patterns

Matching Bean Name Patterns is only supported in Spring AOP – and not in native AspectJ weaving.

Matching all methods on a Spring bean named transferService

@Pointcut("bean(transferService)")

public void allMethodsOfTransferServiceBean();

Matching all methods on Spring beans having names that match the wildcard expression \*Service

@Pointcut("bean(\*Service)")

public void allMethodsOfBeanNameAsTransferService();

4. Combining Pointcut Expressions

Pointcut expressions can be combined using ‘&&’, ‘||’ and ‘!’.

Matching any public methods within service module

@Pointcut("execution(public \* \*(..))")

private void anyPublicOperation() {}

@Pointcut("within(com.doj.app.service..\*)")

private void inService() {}

@Pointcut("anyPublicOperation() && inService()")

private void serviceOperation() {}

# Spring Rest & Security

**Mention what is the difference between SOAP and REST?**

|  |  |
| --- | --- |
| **SOAP** | **REST** |
| * SOAP is a protocol through which two computer communicates by sharing XML document * SOAP permits only XML * SOAP based reads cannot be cached * SOAP is like custom desktop application, closely connected to the server * SOAP is slower than REST * It runs on HTTP but envelopes the message | * Rest is a service architecture and design for network-based software architectures * REST supports many different data formats * REST reads can be cached * A REST client is more like a browser; it knows how to standardized methods and an application has to fit inside it * REST is faster than SOAP * It uses the HTTP headers to hold meta information |

**What are HTTP methods that can be used in Restful web services?**

GET: It requests a resource at the request URL. It should not contain a request body as it will be discarded. May be it can be cached locally or on the server.

POST: It submits information to the service for processing; it should typically return the modified or new resource

PUT: At the request URL it update the resource

DELETE: At the request URL it removes the resource

OPTIONS: It indicates which techniques are supported

HEAD: About the request URL it returns meta information

**5) Mention whether you can use GET request instead of PUT to c**

**What do you mean by Idempotent and which HTTP methods are idempotent?**

Idempotent means result of multiple successful request will not change state of resource after initial application  
For example : Delete is idempotent method because when you first time use delete, it will delete the resource (initial application) but after that, all other request will have no result because resource is already deleted. Get, put and delete are HTTP Idempotent methods.

**What is WSDL?**

WSDL (Web Services Description Language) is an XML format for describing web services and how to access them.

**What is a resource in a REST?**

Resources are identified by logical URLs; it is the key element of a RESTful design.  Unlike, SOAP web services in REST, you view the product data as a resource and this resource should contain all the required information.

      A resource is a unique URL with representation of an object which we can get contents via GET and modify via PUT, POST, DELETE.

**some key characteristics of REST includes**

REST is stateless, therefore the SERVER has no state (or session data)

With a well applied REST API, the server could be restarted between two calls as every data is passed to the server

Web service mostly uses POST method to make operations, whereas REST uses GET to access resources

@Path("/persons")

public class PersonService {

@GET

@Path("/query")

public Response getPersons(

@QueryParam("from") int from,

@QueryParam("to") int to,

@QueryParam("orderBy") List&lt;String&gt; orderBy) {

return Response

.status(200)

.entity("getPersons is called, from : " + from + ", to : " + to+ ", orderBy" + orderBy.toString()).build(); }

@Path("/persons")

public class PersonService {

@POST

@Path("/add")

public Response addPerson(

@FormParam("name") String name,

@FormParam("age") int age) {

return Response.status(200).entity("addPerson is called, name : " + name + ", age : " + age).build();

**Mention what is the difference between PUT and POST?**

“PUT”puts a file or resource at a particular URI and exactly at that URI.  If there is already a file or resource at that URI, PUT changes that file or resource.  If there is no resource or file there, PUT makes one

POST sends data to a particular URI and expects the resource at that URI to deal with the request.  The web server at this point can decide what to do with the data in the context of specified resource

[4 Ways to Secure RESTful Web Services](https://howtodoinjava.com/security/restful-web-services-security-guide/#security-methods)

[- BASIC Authentication](https://howtodoinjava.com/security/restful-web-services-security-guide/#basic-auth)

[- DIGEST Authentication](https://howtodoinjava.com/security/restful-web-services-security-guide/#digest-auth)

[- CLIENT CERT Authentication](https://howtodoinjava.com/security/restful-web-services-security-guide/#cert-auth)

[- Using API Keys](https://howtodoinjava.com/security/restful-web-services-security-guide/#api-keys)

[RESTful Web Services Security Implementations](https://howtodoinjava.com/security/restful-web-services-security-guide/#restful-security-impls)

[- Using SecurityContext](https://howtodoinjava.com/security/restful-web-services-security-guide/#securitycontext)

[- Using Annotations](https://howtodoinjava.com/security/restful-web-services-security-guide/#annotations)

**Configuration of Basic Authentication**

The main reason that form based authentication is not ideal for a RESTful Service is that Spring Security will **make use of Sessions** – this is of course state on the server, so **the statelessness constraints in REST** is practically ignored.

We’ll start by setting up Basic Authentication – first we remove the old custom entry point and filter from the main <http> security element:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <http create-session="stateless">     <intercept-url pattern="/api/admin/\*\*" access="ROLE\_ADMIN" />      <http-basic />  </http> |

Note how support for basic authentication has been added with a single configuration line – <http-basic /> – which handles the creation and wiring of both the BasicAuthenticationFilter and the BasicAuthenticationEntryPoint.

Top 5 REST API Security Guidelines

When developing REST API, one must pay attention to security aspects from the beginning. In this post I will review and explain top 5 security guidelines when [developing and testing REST APIs](http://www.restcase.com/).

REST (or REpresentational State Transfer) is a means of expressing specific entities in a system by URL path elements. REST is not an architecture but it is an architectural style to build services on top of the Web. REST allows interaction with a web-based system via simplified URLs rather than complex request body or POST parameters to request specific items from the system.

1/5 - Authorization

Protect HTTP Methods

RESTful API often use GET (read), POST (create), PUT (replace/update) and DELETE (to delete a record).

Not all of these are valid choices for every single resource collection, user, or action. Make sure the incoming HTTP method is valid for the session token/API key and associated resource collection, action, and record.

For example, if you have a RESTful API for a library, it's not okay to allow anonymous users to DELETE book catalog entries, but it's fine for them to GET a book catalog entry. On the other hand, for the librarian, both of these are valid uses.

Take a look at [CORS](https://en.wikipedia.org/wiki/Cross-origin_resource_sharing) and more if you are looking for more developers resources please check [Enable CORS](http://enable-cors.org/) website.

Whitelist Allowable Methods

It is common with RESTful services to allow multiple methods for a given URL for different operations on that entity.

For example, a GET request might read the entity while PUT would update an existing entity, POST would create a new entity, and DELETE would delete an existing entity.

It is important for the service to properly restrict the allowable verbs such that only the allowed verbs would work, while all others would return a proper response code (**for example, a 403 Forbidden**).

Protect Privileged Actions and Sensitive Resource Collections

Not every user has a right to every web service. This is vital, as you don't want administrative web services to be misused:

https://example.com/admin/exportAllData

The session token or API key should be sent along as a cookie or body parameter to ensure that privileged collections or actions are properly protected from unauthorized use.

Protect Against Cross-site Request Forgery

For resources exposed by RESTful web services, it's important to make sure any PUT, POST, and DELETE request is protected from Cross Site Request Forgery. Typically one would use a token-based approach.

CSRF is easily achieved even using random tokens if any XSS exists within your application, so please make sure you understand how to prevent XSS.

More resources on [CSRF](https://en.wikipedia.org/wiki/Cross-site_request_forgery):

[Cross-Site Request Forgery (CSRF) OWASP](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29)

[OWASP CSRF Prevention Cheat Sheet](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet)

Insecure Direct Object References

It may seem obvious, but if you had a bank account REST web service, you'd have to make sure there is adequate checking of primary and foreign keys:

<https://example.com/account/325365436/transfer?amount=$100.00&toAccount=473846376>  
In this case, it would be possible to transfer money from any account to any other account, which is clearly absurd. Not even a random token makes this safe.

<https://example.com/invoice/2362365>  
In this case, it would be possible to get a copy of all invoices.

This is essentially a data-contextual access control enforcement need. A URL or even a POSTed form should NEVER contain an access control "key" or similar that provides automatic verification. A data contextual check needs to be done, server side, with each request.

2/5 - Input Validation

Everything you know about input validation applies to RESTful web services, but add 10% because automated tools can easily fuzz your interfaces for hours on end at high velocity. So:

Assist the user > Reject input > Sanitize (filtering) > No input validation

Assisting the user makes the most sense, as the most common scenario is "problem exists between keyboard and chair" (PEBKAC).

Help the user input high quality data into your web services, such as ensuring a Zip code makes sense for the supplied address, or the date makes sense. If not, reject that input. If they continue on, or it's a text field or some other difficult to validate field, input sanitization is a losing proposition but still better than XSS or SQL injection.

If you're already reduced to sanitization or no input validation, make sure output encoding is very strong for your application.

Log input validation failures, particularly if you assume that client-side code you wrote is going to call your web services.

The reality is that anyone can call your web services, so assume that someone who is performing hundreds of failed input validations per second is up to no good.

Consider rate limiting the API to a certain number of requests per hour or day to prevent abuse.

URL Validations

Web applications/web services use input from HTTP requests (and occasionally files) to determine how to respond.

Attackers can tamper with any part of an HTTP request, including the URL, query string, headers, cookies, form fields, and hidden fields, to try to bypass the site’s security mechanisms.

Common names for common input tampering attacks include forced browsing, command insertion, cross-site scripting, buffer overflows, format string attacks, SQL injection, cookie poisoning, and hidden field manipulation.

Secure Parsing

Use a secure parser for parsing the incoming messages. If you are using XML, make sure to use a parser that is not vulnerable to XXE and similar attacks.

Strong Typing

It's difficult to perform most attacks if the only allowed values are true or false, or a number, or one of a small number of acceptable values. Strongly type incoming data as quickly as possible.

Validate Incoming Content-types

When POSTing or PUTting new data, the client will specify the Content-Type (e.g. application/xml or application/json) of the incoming data.

**The server should never assume the Content-Type,**it should always check that the Content-Type header and the content are the same type. A lack of Content-Type header or an unexpected Content-Type header should result in the server rejecting the content with a **406 Not Acceptable response**.

Validate Response Types

It is common for REST services to allow multiple response types (e.g. application/XML or application/JSON, and the client specifies the preferred order of response types by the Accept header in the request.

Do NOT simply copy the Accept header to the Content-type header of the response. Reject the request (ideally with a 406 Not Acceptable response) if the Accept header does not specifically contain one of the allowable types.

Because there are many MIME types for the typical response types, it's important to document for clients specifically which MIME types should be used.

XML Input Validation

XML-based services must ensure that they are protected against common XML-based attacks by using secure XML-parsing.

This typically means protecting against XML External Entity attacks, XML-signature wrapping etc.

See [http://ws-attacks.org](http://ws-attacks.org/) for examples of such attacks.

3/5 - Output Encoding

Security Headers

To make sure the content of a given resources is interpreted correctly by the browser, the server should always send the Content-Type header with the correct Content-Type, and preferably the Content-Type header should include a charset.

The server should also send an X-Content-Type-Options: nosniff to make sure the browser does not try to detect a different Content-Type than what is actually sent (can lead to XSS).

Additionally, the client should send an X-Frame-Options: deny to protect against drag'n drop clickjacking attacks in older browsers.

JSON Encoding

A key concern with JSON encoders is preventing arbitrary JavaScript remote code execution within the browser... or, if you're using node.js, on the server. It's vital that you use a proper JSON serializer to encode user-supplied data properly to prevent the execution of user-supplied input on the browser.

When inserting values into the browser DOM, strongly consider using .value/.innerText/.textContent rather than .innerHTML updates, as this protects against simple DOM XSS attacks.

XML Encoding

XML should never be built by string concatenation. It should always be constructed using an XML serializer. This ensures that the XML content sent to the browser is parseable and does not contain XML injection. For more information, please see the Web Service Security Cheat Sheet.

4/5 - Cryptography

Data in Transit

Unless the public information is completely read-only, the use of TLS should be mandated, particularly where credentials, updates, deletions, and any value transactions are performed. The overhead of TLS is negligible on modern hardware, with a minor latency increase that is more than compensated by safety for the end user.

Consider the use of mutually authenticated client-side certificates to provide additional protection for highly privileged web services.

Data in Storage

Leading practices are recommended as per any web application when it comes to correctly handling stored sensitive or regulated data. For more information, please see OWASP Top 10 2010 - A7 Insecure Cryptographic Storage.

Message Integrity

In addition to HTTPS/TLS, JSON Web Token (JWT) is an open standard ([RFC 7519](https://tools.ietf.org/html/rfc7519)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

JWT can not only be used to ensure the message integrity but also authentication of both message sender/receiver.

The JWT includes the digital signature hash value of the message body to ensure the message integrity during the transmission. For more information, you can visit <https://jwt.io/introduction/>.

5/5 - HTTP Status Codes

HTTP defines status code. When design REST API, don't just use 200 for success or 404 for error.

Here are some guideline to consider for each REST API status return code. Proper error handle may help to validate the incoming requests and better identify the potential security risks.   
You will also find more information in this [REST API Error Codes 101](http://blog.restcase.com/rest-api-error-codes-101/) blog post.

**200 OK** - Response to a successful REST API action. The HTTP method can be GET, POST, PUT, PATCH or DELETE.

**400 Bad Request** - The request is malformed, such as message body format error.

**401 Unauthorized** - Wrong or no authentication ID/password provided.

**403 Forbidden** - It's used when the authentication succeeded but authenticated user doesn't have permission to the request resource.

**404 Not Found** - When a non-existent resource is requested.

**405 Method Not Allowed** - The error checking for unexpected HTTP method. For example, the RestAPI is expecting HTTP GET, but HTTP PUT is used.

**429 Too Many Requests** - The error is used when there may be DOS attack detected or the request is rejected due to rate limiting

401 vs 403

**401 “Unauthorized”** really means Unauthenticated, “You need valid credentials for me to respond to this request”.

**403 “Forbidden”** really means Unauthorized, “I understood your credentials, but so sorry, you’re not allowed!”

Summary

In this post, I have covered the top 5 RESTful API security issues and guidelines on how to address them. Following these guidelines will result in a more secure and quality REST API service and a more developer-friendly REST API.

Some methods (for example, HEAD, GET, OPTIONS and TRACE) are defined as safe, which means they are intended only for information retrieval and should not change the state of the server.

RESTful Web Services Security Implementations

Using SecurityContext

The javax.ws.rs.core.SecurityContext interface provides access to security-related information for a request and is very similar to javax.servlet.http.HttpServletRequest.

You access the SecurityContext by injecting an instance into a class field, setter method, or method parameter using the javax.ws.rs.core.Context annotation e.g. in below code sc.isUserInRole() is used to check authorization for user.

|  |
| --- |
| @GET  @Produces("text/plain;charset=UTF-8")  @Path("/hello")  public String sayHello(@Context SecurityContext sc) {          if (sc.isUserInRole("admin"))              return "Hello World!";          throw new SecurityException("User is unauthorized.");  } |

Method level Authorization Using Annotations

This technique is widely used in enterprise application and used to verify roles and responsibities of an authenticated used – for any certain operation. JAX-RS provides below annotations for this purpose.

[@PermitAll](https://docs.oracle.com/javaee/6/api/javax/annotation/security/PermitAll.html)

[@DenyAll](https://docs.oracle.com/javaee/6/api/javax/annotation/security/DenyAll.html)

[@RolesAllowed](https://docs.oracle.com/javaee/6/api/javax/annotation/security/RolesAllowed.html)

An example ue of annotation can be:

|  |
| --- |
| @RolesAllowed("ADMIN")  @PUT  @Path("/users/{id}")  public Response updateUserById(@PathParam("id") int id)  {      //Update the User resource      UserDatabase.updateUser(id);      return Response.status(200).build();  } |

This tutorial shows how to **Secure a REST Service using Spring and Spring Security 4** with Java based configuration. The article will focus on how to set up the Security Configuration specifically for the REST API using a Login and Cookie approach.

**2. Spring Security in the**web.xml

The architecture of Spring Security is based entirely on Servlet Filters and, as such, comes before Spring MVC in regards to the processing of HTTP requests. Keeping this in mind, to begin with, a **filter** needs to be declared in the web.xml of the application:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <filter>     <filter-name>springSecurityFilterChain</filter-name>     <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>  </filter>  <filter-mapping>     <filter-name>springSecurityFilterChain</filter-name>     <url-pattern>/\*</url-pattern>  </filter-mapping> |

The filter must necessarily be named ‘springSecurityFilterChain’  to match the default bean created by Spring Security in the container.

Note that the defined filter is not the actual class implementing the security logic but a DelegatingFilterProxy with the purpose of delegating the Filter’s methods to an internal bean. This is done so that the target bean can still benefit from the Spring context lifecycle and flexibility.

The URL pattern used to configure the Filter is **/\*** even though the entire web service is mapped to **/api/\*** so that the security configuration has the option to secure other possible mappings as well if required.

**3. The XML Security Configuration**

|  |  |
| --- | --- |
| 12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | <http entry-point-ref="restAuthenticationEntryPoint">        <intercept-url pattern="/api/admin/\*\*" access="ROLE\_ADMIN"/>          <form-login           authentication-success-handler-ref="mySuccessHandler"           authentication-failure-handler-ref="myFailureHandler"      />          <logout />     </http>       <beans:bean id="mySuccessHandler"        class="org.rest.security.MySavedRequestAwareAuthenticationSuccessHandler"/>     <beans:bean id="myFailureHandler" class=       "org.springframework.security.web.authentication.SimpleUrlAuthenticationFailureHandler"/>       <authentication-manager alias="authenticationManager">        <authentication-provider>           <user-service>              <user name="temporary" password="temporary" authorities="ROLE\_ADMIN"/>              <user name="user" password="user" authorities="ROLE\_USER"/>           </user-service>        </authentication-provider>     </authentication-manager>    </beans:beans> |

Most of the configuration is done using the **security namespace** – for this to be enabled, the schema locations must be defined and pointed to the correct 4.x XSD versions. The namespace is designed so that it expresses the common use cases of Spring Security while still providing hooks raw beans to accommodate more advanced scenarios.

**3.1. The**<http>**Element**

The <http> element is the main container element for HTTP security configuration. In the current implementation, it only secured a single mapping: /api/admin/\*\*. Note that the mapping is **relative to the root context** of the web application, not to the rest Servlet; this is because the entire security configuration lives in the root Spring context and not in the child context of the Servlet.

**3.2. The Entry Point**

In a standard web application, the authentication process may be automatically triggered when the client tries to access a secured resource without being authenticated – this is usually done by redirecting to a login page so that the user can enter credentials. However, for a **REST Web Service,**this behavior doesn’t make much sense – Authentication should only be done by a request to the correct URI and all other requests should simply fail with a 401 UNAUTHORIZEDstatus code if the user is not authenticated.

Spring Security handles this automatic triggering of the authentication process with the concept of an Entry Point – this is a required part of the configuration, and can be injected via the entry-point-ref attribute of the <http> element. Keeping in mind that this functionality doesn’t make sense in the context of the REST Service, the new custom entry point is defined to simply return 401 whenever it is triggered:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @Component( "restAuthenticationEntryPoint" )  public class RestAuthenticationEntryPoint    implements AuthenticationEntryPoint{     @Override     public void commence(       HttpServletRequest request,       HttpServletResponse response,       AuthenticationException authException) throws IOException {          response.sendError( HttpServletResponse.SC\_UNAUTHORIZED, "Unauthorized" );     }  } |

A quick sidenote here is that the 401 is sent without the WWW-Authenticate header, as required by the HTTP Spec – we can, of course, set the value manually if we need to.

**3.3. The Login Form for REST**

There are multiple ways to do Authentication for a REST API – one of the defaults Spring Security provides is **Form Login** – which uses an authentication processing filter – org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter.

The <form-login> element will create this filter and will also allow us to set our custom authentication success handler on it. This can also be done manually by using the <custom-filter> element to register a filter at the position FORM\_LOGIN\_FILTER – but the namespace support is flexible enough.

Note that for a standard web application, the auto-config attribute of the <http> element is shorthand syntax for some useful security configuration. While this may be appropriate for some very simple configurations, it doesn’t fit and should not be used for a REST API.

**3.4. Authentication should Return**200**Instead of**301

By default, form login will answer a successful authentication request with a **301 MOVED PERMANENTLY** status code; this makes sense in the context of an actual login form which needs to redirect after login. For a RESTful web service, however, the desired response for a successful authentication should be **200 OK**.

This is done by injecting a **custom authentication success handler** in the form login filter, to replace the default one. The new handler implements the exact same login as the default org.springframework.security.web.authentication.SavedRequestAwareAuthenticationSuccessHandlerwith one notable difference – the redirect logic is removed:

|  |  |
| --- | --- |
| 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  30  31  32  33  34  35 | public class MySavedRequestAwareAuthenticationSuccessHandler    extends SimpleUrlAuthenticationSuccessHandler {      private RequestCache requestCache = new HttpSessionRequestCache();      @Override      public void onAuthenticationSuccess(        HttpServletRequest request,        HttpServletResponse response,        Authentication authentication)        throws ServletException, IOException {            SavedRequest savedRequest            = requestCache.getRequest(request, response);            if (savedRequest == null) {              clearAuthenticationAttributes(request);              return;          }          String targetUrlParam = getTargetUrlParameter();          if (isAlwaysUseDefaultTargetUrl()            || (targetUrlParam != null            && StringUtils.hasText(request.getParameter(targetUrlParam)))) {              requestCache.removeRequest(request, response);              clearAuthenticationAttributes(request);              return;          }            clearAuthenticationAttributes(request);      }        public void setRequestCache(RequestCache requestCache) {          this.requestCache = requestCache;      }  } |

**3.5. Failed Authentication should return 401 instead of 302**

Similarly – we configured the authentication failure handler – the same way we did with the success handler.

Luckily – in this case, we don’t need to actually define a new class for this handler – the standard implementation – SimpleUrlAuthenticationFailureHandler – does just fine.

The only difference is that – now that we’re defining this explicitly in our XML config – it’s **not going to get a default**defaultFailureUrl**from Spring** – and so it won’t redirect.

**3.6. The Authentication Manager and Provider**

The authentication process uses an **in-memory provider** to perform authentication – this is meant to simplify the configuration as a production implementation of these artifacts is outside the scope of this post.

**3.7. Finally – Authentication against the running REST Service**

Now let’s see how we can authenticate against the REST API – the URL for login is /login – and a simple curl command performing login would be:

|  |  |
| --- | --- |
| 1  2 | curl -i -X POST -d username=user -d password=userPass  http://localhost:8080/spring-security-rest/login |

This request will return the Cookie which will then be used by any subsequent request against the REST Service.

We can use curl to authentication and **store the cookie it receives in a file**:

|  |  |
| --- | --- |
| 1  2 | curl -i -X POST -d username=user -d password=userPass -c /opt/cookies.txt  http://localhost:8080/spring-security-rest/login |

Then **we can use the cookie from the file** to do further authenticated requests:

|  |  |
| --- | --- |
| 1  2 | curl -i --header "Accept:application/json" -X GET -b /opt/cookies.txt  http://localhost:8080/spring-security-rest/api/foos |

This authenticated request will correctly **result in a 200 OK**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | HTTP/1.1 200 OK  Server: Apache-Coyote/1.1  Content-Type: application/json;charset=UTF-8  Transfer-Encoding: chunked  Date: Wed, 24 Jul 2013 20:31:13 GMT    [{"id":0,"name":"JbidXc"}] |

**4. The Java Security Configuration**

Here’s how the Java configuration would look like:

A quick note, the [Spring Security 4](http://docs.spring.io/spring-security/site/migrate/current/3-to-4/html5/migrate-3-to-4-xml.html#m3to4-xmlnamespace-form-login) config has **changed the old defaults** for XML configuration to be the same as Java defaults.

Old XML Configuration defaults – before Spring Security 4:

loginProcessingUrl: /j\_spring\_security\_check

usernameParameter: j\_username

passwordParameter: j\_password

Current XML Configuration defaults:

loginProcessingUrl: /login

usernameParameter: username

passwordParameter: password

# Spring Annotations

For spring to process annotations, add the following lines in your application-context.xml file.

<context:annotation-config />

<context:component-scan base-package="...specify your package name..." />

|  |  |
| --- | --- |
| Spring Annotation Tip | Spring supports both Annotation based and XML based configurations. You can even mix them together. Annotation injection is performed before XML injection, thus the latter configuration will override the former for properties wired through both approaches. |

@Service

Annotate all your service classes with @Service. All your business logic should be in Service classes.

|  |  |
| --- | --- |
| 1  2  3  4 | @Service  public class CompanyServiceImpl implements CompanyService {  ...  } |

@Repository

Annotate all your DAO classes with @Repository. All your database access logic should be in DAO classes.

|  |  |
| --- | --- |
| 1  2  3  4 | @Repository  public class CompanyDAOImpl implements CompanyDAO {  ...  } |

@Component

Annotate your other components (for example REST resource classes) with @Component.

|  |  |
| --- | --- |
| 1  2  3  4 | @Component  public class ContactResource {  ...  } |

@Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.   
@Autowired

Let Spring auto-wire other beans into your classes using @Autowired annotation.

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5 | @Service  public class CompanyServiceImpl implements CompanyService {     @Autowired    private CompanyDAO companyDAO;} | |
| Spring Annotation Tip | Spring beans can be wired by name or by type.  @Autowire by default is a type driven injection. @Qualifier spring annotation can be used to further fine-tune autowiring.  @Resource (javax.annotation.Resource) annotation can be used for wiring by name.  Beans that are themselves defined as a collection or map type cannot be injected through @Autowired, because type matching is not properly applicable to them. Use @Resource for such beans, referring to the specific collection or map bean by unique name. |

@Transactional

Configure your transactions with @Transactional spring annotation.

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7 | @Service  public class CompanyServiceImpl implements CompanyService {    @Autowired    private CompanyDAO companyDAO;    @Transactional    public Company findByName(String name) {      Company company = companyDAO.findByName(name);      return company;  }  ...} | |
| Spring Annotation Tip | To activate processing of Spring's @Transactional annotation, use the <tx:annotation-driven/> element in your spring's configuration file. |

The default @Transactional settings are as follows:

Propagation setting is PROPAGATION\_REQUIRED.

Isolation level is ISOLATION\_DEFAULT.

Transaction is read/write.

Transaction timeout defaults to the default timeout of the underlying transaction system, or to none if timeouts are not supported.

Any RuntimeException triggers rollback, and any checked Exception does not.

These default settings can be changed using various properties of the @Transactional spring annotation. 

|  |  |
| --- | --- |
| Spring Annotation Tip | Specifying the @Transactional annotation on the bean class means that it applies to all applicable business methods of the class. Specifying the annotation on a method applies it to that method only. If the annotation is applied at both the class and the method level, the method value overrides if the two disagree. |

@Scope

As with Spring-managed components in general, the default and most common scope for autodetected components is singleton. To change this default behavior, use @Scope spring annotation.

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Component  @Scope("request")  public class ContactResource {  ...  } |

Similarly, you can annotate your component with @Scope("prototype") for beans with prototype scopes.

|  |  |
| --- | --- |
| Spring Annotation Tip | Please note that the dependencies are resolved at instantiation time. For prototype scope, it does NOT create a new instance at runtime more than once. It is only during instantiation that each bean is injected with a separate instance of prototype bean. |

Spring MVC Annotations

@Controller

Annotate your controller classes with @Controller.

|  |  |
| --- | --- |
| 1  2  3  4 | @Controller  public class CompanyController {  ...  } |

@RequestMapping

You use the @RequestMapping spring annotation to map URLs onto an entire class or a particular handler method. Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping.

|  |  |
| --- | --- |
| 1 | @Controller  @RequestMapping("/company")  public class CompanyController {    @Autowired    private CompanyService companyService;...} |

@PathVariable

You can use the @PathVariable spring annotation on a method argument to bind it to the value of a URI template variable. In our example below, a request path of /company/techferry will bind companyName variable with 'techferry' value.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Controller  @RequestMapping("/company")  public class CompanyController {    @Autowired    private CompanyService companyService;    @RequestMapping("{companyName}")    public String getCompany(Map<String, Object> map,              @PathVariable String companyName) {      Company company = companyService.findByName(companyName);      map.put("company", company);      return "company";  }...} |

@RequestParam

You can bind request parameters to method variables using spring annotation @RequestParam.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Controller  @RequestMapping("/company")  public class CompanyController {    @Autowired    private CompanyService companyService;    @RequestMapping("/companyList")    public String listCompanies(Map<String, Object> map,              @RequestParam int pageNum) {      map.put("pageNum", pageNum);      map.put("companyList", companyService.listCompanies(pageNum));      return "companyList";    }...} |

Similarly, you can use spring annotation @RequestHeader to bind request headers.

@ModelAttribute

An @ModelAttribute on a method argument indicates the argument should be retrieved from the model. If not present in the model, the argument should be instantiated first and then added to the model. Once present in the model, the argument's fields should be populated from all request parameters that have matching names. This is known as data binding in Spring MVC, a very useful mechanism that saves you from having to parse each form field individually.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Controller  @RequestMapping("/company")  public class CompanyController {    @Autowired    private CompanyService companyService;    @RequestMapping("/add")    public String saveNewCompany(@ModelAttribute Company company) {      companyService.add(company);      return "redirect:" + company.getName();    }...} |

@SessionAttributes

@SessionAttributes spring annotation declares session attributes. This will typically list the names of model attributes which should be transparently stored in the session, serving as form-backing beans between subsequent requests.

|  |  |
| --- | --- |
| 18  9 | @Controller  @RequestMapping("/company")  @SessionAttributes("company")  public class CompanyController {    @Autowired    private CompanyService companyService;...} |

@SessionAttribute works as follows:

@SessionAttribute is initialized when you put the corresponding attribute into model (either explicitly or using @ModelAttribute-annotated method).

@SessionAttribute is updated by the data from HTTP parameters when controller method with the corresponding model attribute in its signature is invoked.

@SessionAttributes are cleared when you call setComplete() on SessionStatus object passed into controller method as an argument.

The following listing illustrate these concepts. It is also an example for pre-populating Model objects.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | @Controller  @RequestMapping("/owners/{ownerId}/pets/{petId}/edit")  @SessionAttributes("pet")  public class EditPetForm {      @ModelAttribute("types")           public Collection<PetType> populatePetTypes() {          return this.clinic.getPetTypes();      }      @RequestMapping(method = RequestMethod.POST)      public String processSubmit(@ModelAttribute("pet") Pet pet,              BindingResult result, SessionStatus status) {          new PetValidator().validate(pet, result);          if (result.hasErrors()) {              return "petForm";          }else {              this.clinic.storePet(pet);              status.setComplete();              return "redirect:owner.do?ownerId="                  + pet.getOwner().getId();          }    }} |

Spring Security Annotations

@PreAuthorize

Using Spring Security @PreAuthorize annotation, you can authorize or deny a functionality. In our example below, only a user with Admin role has the access to delete a contact.

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Transactional  @PreAuthorize("hasRole('ROLE\_ADMIN')")  public void removeContact(Integer id) {    contactDAO.removeContact(id);  } |