Day 1

**Overview and Use Cases**

Reflection API

It is an API for modifying the behavior of methods and classes at runtime. It's called "reflection" because you're reflecting/introspecting into the code (so that it can modify itself).

Classes in the API:

* Class
* Method
* Modifier
* Parameter

**NOTE:** You can do things like check an if a class has a specific annotation, method or method signature, or modifier on a method. Frameworks like Spring heavily rely on Reflection to function such as detecting methods and loading classes dynamically. Even JUnit uses Reflection to detect methods with the @Test annotation on them.

**Intro to Spring**

# **Spring Framework - Intro to Spring**

The Spring Framework is an inversion of control container that provides flexible infrastructural support to create loosely coupled Java applications by utilizing dependency injection.

### References

* [Spring Framework API - Java Documentation](https://javadoc.io/doc/org.springframework/spring-core/latest/index.html)
* [Overview of Spring Framework - Spring Documentation 4.0.x](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/overview.html)

## What is Spring?

Spring an umbrella term for a family of frameworks which can be utilized to rapidly create loosely coupled Java applications. These spring frameworks provide a comprehensive and configurable model for modern enterprise Java applications, which can be easily migrated to any kind of deployment platform. This ease is due to the core feature of the Spring frameworks, which focuses on [dependency injection](https://app.revature.com/) resulting from the framework acting as an [inversion of control](https://app.revature.com/) container.

Spring enables developers to build java applications utilizing a POJO design pattern, and applying enterprises services, as needed, non-invasively to those POJOs. This loose coupling allows developers to focus on business logic for applications as Spring handles the infrastructural needs.

## Frameworks - Overview

The Spring family of frameworks consist of close to 20 modules, each focusing on a particular task or service. These are grouped into the following layers: Core Container, Data Access/Integration, Messaging, Web, AOP, Aspects, Instrumentation and Test.



### Core Container

The Core Container provide the basic framework for the IoC container and dependency injection.

* [Core & Beans](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html#beans-introduction): These modules provide the fundamental framework for springs IoC container, including dependency injection features. Beans specifically feature the BeanFactory, which is a sophisticated implementation of the factory design pattern used to create beans, which are used in dependency injection.
* [Context](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html#context-introduction): This modules builds off from the core and bean modules used for more enterprise functionality. The main feature, ApplicationContext represents the Spring IoC container and is used to instantiate, onfigure and assemble beans.
* [SpEL (Spring Expression Language)](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/expressions.html): A module which provides a powerful expression language which can be used to query and manipulate an object graph at runtime, including setting and getting property values, property assignment, method invocation, accessing array content, collections and indexer and more.

### Data Access/Integration

The Data Access/Integration layer provides support for database management or layers of abstraction for ease of use.

* [JDBC (Java Database Connectivity)](https://app.revature.com/): A module which provides a layer of abstraction for [JDBC](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/jdbc.html#jdbc-introduction)
* [ORM (Object Relational Mapping)](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/orm.html): A module which provides integration layers for ORM APIs, such as JPA, JDO and Hibernate
* [OXM (Object/XML Mapping)](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/oxm.html): A module which provides a layer of abstraction for mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream
* [JMS (Java Messaging System)](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/jms.html): A module which provides feature to produce and consume messages.
* Transaction: A module which provides programmatic and declarative support for transaction management in classes that implement special interfaces as well as POJOs

### Web

The Web layer provides basic web integration features for an application.

* [Web-Servlet](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/mvc.html): A module which provides an implementation for Spring MVC, a clean separation between model code and web forms, and also integrates with other features of the Spring framework.
* [WebSocket](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/websocket.html): A module which provides a standardized way to esablish a communication channel between a client and server with a single TCP connection.
* [Web-Portlet](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/portlet.html): A module incredibly similar to the servlet workflow, that is marked by two distinct phases, an action phase, which is executed once when any backend changes occur, and a render phase, in which information is displayed to the user.

### [AOP (Aspect Oriented Programming)](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/aop.html)

The AOP modules provides support for Aspect Oriented Programming, which is different way of thinking about program structure, as opposed to Object Oriented Programming. Aspect Oriented Programming aims to decouple code from functionality that is independent from the core functionality of that code (for example, logging information, and security are generalized functionality that is independent of other, specific business logic, and therefore would be handled by a specific aspect.

### [Aspects](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/aop.html#aop-ataspectj)

Aspects, particularly AspectJ for Spring, is the module which, when enabled, allows for the use of the AspectJ supported style of declaring and autoproxying beans based on whether the bean is being advised by one or more aspects.

### Instrumentation

A module which provides support for instrumentation (the process of planning, installing, monitoring and maintaining systems) and classloader implementation used in an application server.

### [Test](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/testing.html)

A module which provides support for integration and best practice unit testing, focusing on JUnit or TestNG, as well as mocking information.

**Spring IOC Container & Dependency Injection**

# **Spring Module - IOC Container and Dependency Injection**

The Spring Framework acts as an Inversion of Control (IoC) Container by utilizing Dependency Injection. IoC or Dependency Injection, refers to the process whereby objects define their dependencies, but creation, assembly and injection of those objects is performed by the Spring IoC container.

### References

* [Spring 4.0 - IoC Container](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html)
* [Inversion of Control and Dependency Injection - Martin Fowler](https://martinfowler.com/articles/injection.html)
* [IoC and Depedency Injection - Tutorials Teacher](https://www.tutorialsteacher.com/ioc/inversion-of-control)

## Inversion of Control

Inversion of Control is a design principle in which control over certain parts of object-oriented design is inverted to achieve loose coupling. An simple way to think of this would be to Suppose a user has a car and drives to work each day. Currently, the user is in control of the car. However, if the user schedules an uber instead, the control is inverted from the user to the uber driver, allowing the user to focus on other tasks while still allowing the car to ultimately be driven to work.

In Spring, the IoC Container is responsible for instantiating, configuring and assembling objects known as beans. It does this by getting information from the XML file and assembling the objects accordingly. In Spring there are two types of IoC Containers, the BeanFactory and Application context, which is built out of the Bean factory. More information on BeanFactory, ApplicationContext and Beans can be found in the [configuration lecture notes](https://app.revature.com/).

The ApplicationContext interface is built on top of the BeanFactory with extra functionality, such as simple integration with Spring AOP, event propagation, message resource handling, and application layer specific context (such as WebApplicationContext for web applications).

## Dependency Injection

Fundamentally, every framework implements at least some level of Inversion of Control. Dependency Injection is a more specific term for what control is inverted with regard to the Spring Framework. As the name suggests, Dependency Injection is a design patters that removes dependencies of a program by providing the configuration in an external source, such as an ML file. This loosely coupled design then makes code easier to test, and implement in a wider variety of environments.

Dependency Injection can occur through the following methods:

* Constructor Injection: Dependency Injection accomplished when the container invokes a constructor with arguments to instantiate a bean in which each argument of said constructor represents a dependency.
* Setter Injection: Dependency Injection accomplished when the container calls setter methods on a bean after invoking a no-argument constructor to instatiate a bean.

### Constructor Injection Examples

**XML Configuration**

<beans>

<bean id = "order" class = "com.revature.models.Order">

<constructor-arg ref="account"/>

<constructor-arg ref="item"/>

</bean>

<bean id = "account" class = "com.revature.models.Account">

</bean>

<bean id = "item" class = "com.revature.models.Item">

</bean>

</beans>

**Annotation Configuration**

package com.revature.models;

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

public class Order {

private Account account;

private int id;

private Item item;

@Autowired // multiple beans are distinguished by object type

public Order(Account account, int id, Item item) {

super();

this.account = type;

this.id = id;

this.item = item;

}

...

}

**Java Configuration**

package com.revature.config;

// import statements

...

@Configuration

@ComponentScan({ "com.revature.repository;com.revature.services" })

public class AppConfig {

...

@Bean(name = "orderService")

public OrderService getOrderService() {

OrderServiceImpl oserv = new OrderServiceImpl(getOrderDao()); //<--- consturctor injection

return oserv;

}

}

### Setter Injection Examples

**XML Configuration** NOTE: XML Configuration does not directly support setter injection, rather, it is accomplished through properties

<beans>

...

<bean id="accountServ" class="com.revature.services.AccountServiceImpl">

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

</bean>

<bean id="orderServ" class="com.revature.services.OrderServiceImpl">

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

</bean>

<bean id="itemServ" class="com.revature.services.ItemServiceImpl">

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

</bean>

</beans>

**Annotation Configuration**

package com.revature.models;

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

public class Order {

private Account account;

private int id;

private Item item;

@Autowired

public void setItem(Item item) {

this.item = item;

}

...

}

**Java Configuration**

package com.revature.config;

// import statements

...

@Configuration

@ComponentScan({ "com.revature.repository;com.revature.services" })

public class AppConfig {

...

@Bean(name="accountService")

public AccountService getAccountService() {

AccountServiceImpl aserv = new AccountServiceImpl();

as.setAccountDao(getAccountDao()); //<---setter injection

return aserv;

}

}

### Constructor vs Setter Injection

* Constructor Injection is more secure, since dependencies are required to create an object, you are guaranteed to have each dependency populated
* Consturctor Injection enables the implementation of immutable objects
* Setter Injection allows for partial dependencies since Constructor injection requires all properties to be established upon bean instantiation.
* Setter Injection occurs after constructor injection, essentially putting giving setter injection precedence over constructor injection
* Setter Injection can easily change values, and does not create new bean instances, making it more flexible than constructor injection.
* Setter Injection can resolve circular references (i.e. if Object A and Object B are dependent on each other, setter injection can be used to resolve this, whereas Constructor injection would throw a BeanCurrentlyInCreationException).

**XML vs Annotation-based configuration**

# **Spring Module - XML vs Annotation vs Java Configuration**

This page details XML-based and Annotation-based configurations for the ApplicationContext as well as Java configurations of Spring Beans.

### References

* [Spring 4.0.x Documentation - IoC Container & Image](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html)
* [Spring 4.0.x API Docs](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/overview-summary.html)

## Bean Factory & Application Context

The Spring [Bean Factory](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/beans/factory/BeanFactory.html) interface establishes the underlying basis for the Spring IoC Container functionality by providing advanced configurations capable of managing any type of object. The ApplicationContext is a sub-interface of the BeanFactory interface which adds easier integration with Spring's AOP features. This ApplicationContext represents the Spring IoC container as it is responsible for the instantiation, assembly and management of "beans". These "Beans", are the term used for objects created in Spring which form the backbone of an application. Though they are created and managed by the IoC Container, they are otherwise just a type of object in an application. When bean objects have certain dependencies, these are reflected within configuration metadata used by the container.

In other words, the BeanFactory provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality for to the creation and management of objects in the Spring framework.

With regard to the configuration metadata, there are three ways this configuration can be represented, through XML, Java annotation or Java Code.

The following image displays a high-level representation of how Spring works:



The application classes are combined with configuration data when the ApplicationContext is created and initialized, which produces a fully configured application, or system.

It is worth noting that the BeanFactory instantiates beans lazily (creating beans only when required), while ApplicationContext instantiates beans eagerly (once the bean becomes relevant, and may potentially be used).

## XML Configuration

XML configuration consists of the declaration of at least one element nested inside a top-level element. These bean definitions correspond to actual objects within your application, and typically relate to server layer, data access (DAO), infrastructure objects (such as hibernate SessionFactories), or presentation objects (such as Struts Action instances). The following code is an example of a basic bean configuration using XML:

<!-- This XML file defines service beans and is named 'services.xml' -->

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

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

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

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

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

<!-- services -->

<bean id="accountServ" class="com.revature.services.AccountServiceImpl">

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

<!-- additional collaborators and configuration for this bean go here -->

</bean>

<bean id="orderServ" class="com.revature.services.OrderServiceImpl">

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

<!-- additional collaborators and configuration for this bean go here -->

</bean>

<bean id="itemServ" class="com.revature.services.ItemServiceImpl">

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

<!-- additional collaborators and configuration for this bean go here -->

</bean>

<!-- more bean definitions for services go here -->

</beans>

<!-- This XML file defines DAO beans and is named 'daos.xml' -->

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

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

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

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

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

<bean id="accountDao" class="com.revature.dao.AccountDaoImpl">

<!-- additional collaborators and configuration for this bean go here -->

</bean>

<bean id="orderDao" class="com.revature.dao.OrderDaoImpl">

<!-- additional collaborators and configuration for this bean go here -->

</bean>

<bean id="itemDao" class="com.revature.dao.ItemDaoImpl">

<!-- additional collaborators and configuration for this bean go here -->

</bean>

<!-- more bean definitions for data access objects go here -->

</beans>

Here, the id attribute is a string used to identify specific bean definitions, while the class attribute defines the type of bean using the fully qualified class name. Note that you can substitute the id attribute with the name attribute if you would like to provide multiple aliases for a bean.

To instantiate a Spring IoC container using this XML configuration, the location path or paths should be supplied to an ApplicationContext Constructor. For XML configurations the ApplicationContext object should be derived from the [ClassPathXmlApplicationContext](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html) class:

// Single XML Path File

ApplicationContext context = new ClassPathXmlApplicationContext(new String ("services.xml"));

// Multiple XML Path Files

ApplicationContext context = new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});

## Annotation Configuration

Annotation configuration, introduced as of Spring 2.5, serves as an alternative to XML-based setups, and rely on the use of annotations and bytecode metadata to wire components as opposed to the used of angle-bracket element declarations. Utilization of annotation based configuration still requires the use of an ApplicationContext XML file; however, the bean configurations are separated from this XML file into the component class files themselves and are defined using relevant annotations on the class, method or field declaration.

To use annotation configuration the context tag will be required within the ApplicationContext configuration file. This element enables the use of annotations which is, by default, turned off. Additionally, you will need to include the xml namespace "context". For Example:

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

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

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

<!-- The context namespace (below) is required for context element -->

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

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

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

http://www.springframework.org/schema/context

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

<!-- The following enables use of annotation-based configurations to be detected within your class files -->

<context:annotation-config/>

<!-- Bean definitions -->

<bean id = "account" class = "com.revature.models.Account">

</bean>

<bean id = "order" class = "com.revature.models.Order">

</bean>

<bean id = "item" class = "com.revature.models.Item">

</bean>

</beans>

To properly configure beans, use of the @Autowired annotation should be used. This annotation injects beans based on their type, and it commonly used with singleton bean design patterns.

@Autowired [Property]

package com.revature.models;

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

public class Order {

private Account account;

private int id;

// @Autowired on properties can take the place of setter methods

@Autowired

private Item item;

public Item getItem() {

return item;

}

...

}

@Autowired [Setter]

package com.revature.models;

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

public class Order {

private Account account;

private int id;

private Item item;

@Autowired

public void setItem(Item item) {

this.item = item;

}

...

}

@Autowired [Constructor]

package com.revature.models;

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

public class Order {

private Account account;

private int id;

private Item item;

public Order() {

super();

// TODO Auto-generated constructor stub

}

@Autowired // multiple beans are distinguished by object type

public Order(Account account, int id, Item item) {

super();

this.account = account;

this.id = id;

this.item = item;

}

...

}

If multiple bean definitions exist for a particular object type the @Resource or @Qualifier annotations can disambiguate the bean. @Resource is a standard Java annotation which injects beans based on name, rather than type. The @Qualifier annotation specifies a particular @Autowired-annotated bean based on the bean's identifier (the bean name). As such, the @Qualifier annotation is a Spring annotation:

@Resource

@Resource("item\_one") // if multiple items exist...

public void setItem(Item item) {

this.item = item;

}

@Qualifier

@Autowired

public Order(@Qualifier("account")Account account, int id, Item item) {

super();

this.account = account;

this.id = id;

this.item = item;

}

@Required

package com.revature.models;

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

public class Order {

private Account account;

...

@Required

public void setAccount(Account Account) {

this.Account = account;

}

...

}

Note that <context-annotation-config> only looks for annotations on beans within the same application context in which it is defined. This means if you put <context:annotation-config/> within a WebApplicationContext for a DispatcherServlet, it will only check for @Autowired beans within these controllers, but not beans in your services. You can otherwise specify packages using the <context:component-scan/> element with a list of packages:

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

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

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

<!-- The context namespace, below, is required for context element -->

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

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

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

http://www.springframework.org/schema/context

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

<context:annotation-config/>

<!-- Component Scan informs Spring where to look for bean wiring configurations -->

<context:component-scan base-package="com.revature.services;com.revature.dao;com.revature.models">

<!-- Bean definitions -->

...

</beans>

The method to instantiate the Spring IoC container using annotation configuration is similar to the XML-based configuration. The location path should be specified in an ApplicationContext Constructor using the [ClassPathXmlApplicationContext](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html) class:

ApplicationContext context = new ClassPathXmlApplicationContext(new String ("applicationContext.xml"));

### @Inject and @Named

Java also provides standard annotations which can be used to inject object dependencies. These annotations, @Inject and @Named require the use of the javax.inject package. Similarly to @Autowired, @Inject can occur at the class, field and constructor argument levels, based on the bean type. The @Named annotation can serve the same purpose as the @Qualifier annotation to disambiguate between bean definitions, and additionally be used as a bean definition equivalent to the @Component annotation.

@Inject

package com.revature.models;

import javax.inject.Inject;

import javax.inject.Named;

public class Order {

private Account account;

private int id;

private Item item;

@Inject

public void setAccount(Account account) {

this.Account = account;

}

}

@Named

package com.revature.models;

import javax.inject.Inject;

import javax.inject.Named;

@Named("order")

public class Order {

private Account account;

private int id;

private Item item;

@Inject // Similarly to @Autowired, type is used to distinguish beans when a qualifier (@Named) is not used

public void setAccount(Account account) {

this.Account = account;

}

...

}

**NOTE**: Annotation injection is performed prior to XML injection. As such, the use of both will result in XML configurations overriding Annotation configurations.

## Java Configuration

Java-based configuration also uses annotations, but do not require the use of an application context file. Instead, configuration management is handled by @Configuration-annotated class files which define @Bean-annotated methods used to describe the beans.

Note that you can use @Bean-annotated methods with any class which is @Component-annotated; however, they are most commonly used with @Configuration beans since the primary purpose for a class which is @Configuration-annotated is the definition of beans. In order to autowire or inject primitives, the @ComponentScan annotation specifies where to search for defined beans. Additionally all of these beans must be imported from the org.springframework.context.annotation package.

The following example shows an illustration of the Java Configuration method:

package com.revature.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Scope;

import com.revature.dao.AccountDao;

import com.revature.dao.AccountDaoImpl;

import com.revature.dao.OrderDao;

import com.revature.dao.OrderDaoImpl;

import com.revature.dao.ItemDao;

import com.revature.dao.ItemDaoImpl;

import com.revature.services.AccountService;

import com.revature.services.AccountServiceImpl;

import com.revature.services.OrderService;

import com.revature.services.OrderServiceImpl;

import com.revature.services.ItemService;

import com.revature.services.ItemServiceImpl;

@Configuration

@ComponentScan({ "com.revature.repository;com.revature.services" })

public class AppConfig {

// =====================Setter Injection======================

@Bean(name="accountDao")

public AccountDao getAccountDao() {

return new AccountDaoImpl();

}

@Bean(name="accountService")

public AccountService getAccountService() {

AccountServiceImpl aserv = new AccountServiceImpl();

as.setAccountDao(getAccountDao()); //<---setter injection

return aserv;

}

//=================Constructor Injection=================

@Bean(name = "orderDao")

public OrderDao getOrderDao() {

return new OrderDaoImpl();

}

@Bean(name = "orderService")

public OrderService getOrderService() {

OrderServiceImpl oserv = new OrderServiceImpl(getOrderDao()); //<--- consturctor injection

return oserv;

}

@Bean(name = "itemDao")

public ItemDao getItemDao() {

return new ItemDaoImpl();

}

@Bean(name = "itemService")

public ItemService getItemService() {

ItemServiceImpl iserv = new ItemServiceImpl(getItemDao());

return iserv;

}

}

Instantiation for the Spring IoC container using Java configuration uses the [AnnotationConfigApplicationContext](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/context/annotation/AnnotationConfigApplicationContext.html) class with the @Configuration-annotated class as an argument in the constructor call:

ApplicationContext appContext = new AnnotationConfigApplicationContext(AppConfig.class);

### Summary of Annotations [Alphabetical]:

* @Autowired - Spring Annotation used to inject bean dependencies as needed, based on bean type.
* @Bean - Spring annotation which plays the same role as the element in an XML-based configuration, and you can specify an identifier for these beans with the use of the id (or name) property.
* @Component - Generic [stereotype annotation](https://app.revature.com/) used to declare an object as a bean.
* @ComponentScan - Spring Annotation which specifies path locations for defined beans to be used for potential injection.
* @Configuration - Spring annotation indicates a class file used to manage bean configurations using Java configuration similar to the Application Context file for an XML or annotation based configuration.
* @Inject - Standard Java annotation used to inject bean dependencies as needed. Equivalent to Spring's @Autowired annotation.
* @Named - Standard Java annotation for disambiguating beans based on bean name. Equivalent to Spring's @Qualifier annotation. Additionally can be used as an equivalent to Spring's @Component annotation to define beans as well.
* @Qualifier - Spring Annotation which can be used in conjunction with @Autowired to disambiguate multiple beans of a defined type.
* @Required - Standard Java annotation which indicates that an affected bean property must be populated at configuration time through an explicit property value in the bean's definition.
* @Resource - Standard Java Annotation used to inject bean dependencies based on bean name, rather than type.

**Bean lifecycle**

# **Spring Module - Bean Lifecycle**

This file details information regarding the lifecycle stages of Spring Beans as of version 4.

### Resources

* [Spring 4.0.x Documentation - IoC Container and Beans](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html)
* [Bean Factory - Interface Documentation](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/BeanFactory.html)
* [Spring Bean Life Cycle - Tutorialspoint](https://www.tutorialspoint.com/spring/spring_bean_life_cycle.htm)
* [Lifecycle Image Reference](https://howtodoinjava.com/spring-core/spring-bean-life-cycle/)

## Bean Lifecycle

The management of Beans, conducted by the BeanFactory or Application Context, includes instantiation, configuration and the eventual removal (or destruction) of beans. As a high-level overview:

1. Beans are first instantiated.
2. Their properties are set.
3. Any associated interfaces or objects are made aware of their existence.
4. The bean is made aware of any associated interfaces as well.
5. Any other methods, particularly custom created methods, are invoked.
6. Then the bean is ready for use.
7. Once the bean is no longer used, it is marked for removal and a destroy method is invoked for the bean
8. Custom destroy methods are invoked, if any.
9. Bean is the destroyed.

The following is a visualization of this lifecycle:



Specifically the lifecycle of a Spring Bean in an application context, pursuant to the associated interface methods is as follows: Instantiate Bean - Bean is instantiated Populate bean properties - Bean properties are established. Related Interfaces are made aware. This includes:

* [BeanNameAware's setBeanName](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/BeanNameAware.html) - Informs Bean Factory of Bean Name
* [BeanClassLoaderAware's setBeanClassLoader](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/BeanClassLoaderAware.html) - Callback to inform bean [class loader](https://docs.oracle.com/javase/8/docs/api/java/lang/ClassLoader.html?is-external=true) of bean. A Class loader is an object responsible for loading class.
* [BeanFactoryAware's setBeanFactory](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/BeanFactoryAware.html) - Informs bean of their owning bean factory
* [EnvironmentAware's setEnvironment](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/EnvironmentAware.html) - Informs the bean of the environment in which it runs.
* [EmbeddedValueResolverAware's setEmbeddedValueResolver](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/EmbeddedValueResolverAware.html) - Informs bean of notifications from StringValueResolver, when embedded definition values are resolved.

If using an application context, the following are also made aware:

* [ResourceLoaderAware's setResourceLoader](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/core/io/ResourceLoader.html)
* [ApplicationEventPublisherAware's setApplicationEventPublisher](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/ApplicationEventPublisher.html)
* [MessageSourceAware's setMessageSource](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/MessageSource.html)
* [ApplicationContextAware's setApplicationContext](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/ApplicationContextAware.html)
* [ServletContextAware's setServletContext (Specifically only applicable when running in a **web application context**)](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/context/ServletContextAware.html)

After Awareness:

* [postProcessBeforeInitialization methods of BeanPostProcessors](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/config/BeanPostProcessor.html#postProcessBeforeInitialization-java.lang.Object-java.lang.String-) - Method called before bean initialization callback methods are invoked, but after bean has been populated with property values.
* [InitializingBean's afterPropertiesSet](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/InitializingBean.html#afterPropertiesSet--) - Method invoked by the owner bean factory after all bean properties have been set and satisfied Aware methods (BeanFactoryAware, ApplicationContextAware, etc...)
* [Custom init-method definition] - Custom Defined init method. Signature is public void init(). This method is detected by the [getInitMethodName Method](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/support/AbstractBeanDefinition.html#getInitMethodName--)
* [postProcessAfterInitialization methods of BeanPostProcessors](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/config/BeanPostProcessor.html#postProcessBeforeInitialization-java.lang.Object-java.lang.String-) - Callback which allows post processor to decide whether to apply either the FactoryBean, created object or both through corresponding instanceof FactoryBean checks.
* Bean is ready for use.

On shutdown of a bean factory (or destruction of a bean), the following lifecycle methods apply:

* [postProcessBeforeDestruction methods of DestructionAwareBeanPostProcessors](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/annotation/InitDestroyAnnotationBeanPostProcessor.html#postProcessBeforeDestruction-java.lang.Object-java.lang.String-) - Applies the bean's BeanPostProcessor before its destruction inclucking the invocation of custom destruction callbacks
* [DisposableBean's destroy](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/DisposableBean.html#destroy--) - Method invoked by bean factory (or ApplicationContext) when a bean is to be destroyed
* [Custom destroy-method definition] - Custom Defined destroy method. Signature is public void destroy(). This method is detected by the [getDestroyMethodName Method](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/support/AbstractBeanDefinition.html#getDestroyMethodName--)

It is worth nothing that Spring does not manage this entire lifecycle for all Beans, but rather only those which exist in particular [bean scopes](https://app.revature.com/).

**Injecting primitives (XML)**

# **Spring Module - Injecting Primitives**

This page details the value attribute and utilizing configuration files to inject primitive and String data into Spring Bean objects.

### References

* [Spring 4.0.x Documentation](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html)

## Injecting Primitives

Spring beans have two methods of injecting dependent data, Constructor Injection and Setter Injection.

Constructor injection occurs when an IoC container creates a bean, passing values as arguments within that bean's constructor. Setter injection occurs when the IoC container specifically invokes a bean's setter methods for the particular properties within the object. More information on dependency injection can be found [here](https://app.revature.com/).

Spring XML-based configuration files support the <property/> and <constructor-arg/> elements, which can be used to define various property values. The <property/> and/or <constructor-arg/> element should be nested within the bean in this instance. Take for example:

package com.revature.example;

public class FruitBasket {

public FruitBasket(Apple apple, Banana banana) {

// ...

}

}

<beans>

<bean id="fruitBasket" class="com.revature.example.FruitBasket">

<constructor-arg ref="apple"/>

<constructor-arg ref="banana"/>

</bean>

<bean id="apple" class="com.revature.example.Apple"/>

<bean id="banana" class="com.revature.example.Banana"/>

</beans>

When resolving bean values, the argument's type is used. If no potential ambiguity exsists, the order in which the constructor arguments are defined is the order in which those arguments are supplied to the constructor.

To inject primitive or hard-coded values, you may use the value attribute within the <constructor-arg/> or <property/> element. This value attribute specifies a human-readable **string representation** of the data you want to pass into the bean property.

package com.revature.example;

public class ExampleBean {

public ExampleBean(int num, String word) {

// ...

}

}

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg type="int" value="12345"/>

<constructor-arg type="java.lang.String" value="Hello World"/>

</bean>

In the event ambiguity does exist, the index property can be used to explicity define the constructor arguments based on the order they are defined in the constructor signature.

package com.revature.example;

public class ExampleBean {

public ExampleBean(int num\_1, int num\_2) {

// ...

}

}

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg index="0" value="1234"/>

<constructor-arg index="1" value="4321"/>

</bean>

For Java-based and Annotation configurations, the @Value annotation can specify the injection of data when paired with a separate java configuration file. For example:

Configuration file (jdbc.properties file):

jdbc.driverClassName=oracle.jdbc.OracleDriver

jdbc.url=jdbc:jdbc:oracle:thin:@localhost:1521/orclpdb1

jdbc.username=user

jdbc.password=password

Class File:

@Configuration

@ImportResource("classpath:/com/revature/main/resources/properties-config.xml")

public class AppConfig {

@Value("${jdbc.url}")

private String url;

@Value("${jdbc.username}")

private String username;

@Value("${jdbc.password}")

private String password;

@Bean

public DataSource dataSource() {

return new DriverManagerDataSource(url, username, password);

}

}

**Scopes of a bean**

# **Topic Name**

This page details "out-of-the-box" as well as custom Scopes for Spring beans.

### References

* [Bean Scopes - Spring 4.0.x documentation](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html#beans-factory-scopes)

## Bean Scopes

Bean configuration files act similarly to the creation of a class file, in that it is a template which can be used to instantiate objects (in this case, Spring uses this configuration to create bean objects). This configuration also extends to the defined scope of those objects. As a reminder, a scope can be seen as a sub-section of a larger application, with certain defined values, properties and objects. When defining a bean in the configuration, you have the ability to define the scope as well. For XML configurations, use the scope attribute on the bean element. Otherwise you can use the @Scope annotation to override the default (Singleton) scope for configured beans.

With regard to Spring Beans, there are six scopes, four of which can only be by access when utilizing a web aware ApplicationContext. They are:

1. Singleton Scope (default) - Each bean definition is scoped to a single object instance per Spring IoC Container.
2. Prototype Scope - A single bean definition is scoped to a number of object instances.
3. Request Scope [Only valid with a web-aware ApplicationContext] - A single bean definition is scoped to the lifecycle of a single HTTP request. This means the HTTP request has its own instance of a bean created from a single bean definition.
4. Session Scope [Only valid with a web-aware ApplicationContext] - A single bean definition is scoped to the lifecycle of an HTTP Session.
5. Gloabl Session Scope [Only valid with a web-aware ApplicationContext] - A single bean definition is scoped to the lifecycle of a global HTTP session. Generally only valid when used in a portlet context.
6. Application Scope [Only valid with a web-aware ApplicationContext] - Scopes a single bean definition to the lifecycle of a ServletContext.

### [Singleton Scope](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html#beans-factory-scopes-singleton)

For a Singleton Scoped bean, the IoC container will create exactly one instance of a Spring bean per bean definition. Each of those singleton beans are then stored in a cache of singleton beans. When a named bean is requested, the associated bean within this cache is returned.

The Single Scope for beans is the default scope, meaning that if no scope is explicitly defined, the bean will be instantiated in this singleton scope.

### [Prototype Scope](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html#beans-factory-scopes-prototype)

For non-singleton beans, the prototype scope allows for the creation of a new bean instance each time a request for a specific bean definition is made.

As a general rule, the prototype scope should be used for stateful beans, while the singleton scope should be used for stateless beans.

It is also worth noting that, unlike the other scopes, Spring does not manage the complete lifecycle of beans within the prototype scope. The container will still instantiate, configure and assemble the prototype bean object, but at that point will hand this prototype off to the client, with no other records of the prototyped instance. As such, with regard to bean lifecycles, all initialization callback methods are called on every bean, regardless of scope, the same cannot be said for the destruction lifecycle callbacks.

### [Request Scope](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html#beans-factory-scopes-request)

For the Request scope, the Spring IoC Container will create a new instance of a bean, based on the associated bean definition for every HTTP request recieved. These beans are particular to individual requests, so when the request completes, the associated bean within the request scope will be discarded.

### [Session Scope](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html#beans-factory-scopes-session)

For the Session scope, the Spring IoC container will create a new instance of a bean, based on the associated bean definition, each time a particular HTTP Session is created. When the session eventually ends, the associated bean will be discarded along with it.

### [Global Session Scope](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html#beans-factory-scopes-global-session)

For the Global Session scope, similarly to the previous definitions, will create a bean, however, this applies only with regard to portlet-based web applications. The portlet defines the global Session that is shared among all portlets, and the IoC Container uses this global session definition to scope Beans described at the global session level. These beans are similarly discarded when the associated global portlet session concludes.

Note if you attempt to use the Request, Session, and Global Session scope within a regular Spring IoC container (such as the ClassPathXmlApplicationContext), you will get an IllegalStateException regarding an unknown bean scope.

### [Application Scope](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html#beans-factory-scopes-application)

Within an Application Scope, a Spring IoC Container will create a new instance of the bean based on the associated bean definition once for the entire web application. This is similar to a singleton bean, but has two important differences.

1. An application scope bean is a singleton for each ServletContext not for each Spring ApplicationContext
2. Beans within the application scope are visible within the ServletContext meaning they can be used as an attribute for ServletContext

XML:

<!-- The following is redundant since the singleton scope is the implicit behavior -->

<bean id="exampleBean" class="com.revature.example.ExampleBean" scope="singleton"/>

<bean id="protoBean" class="com.revature.example.ProtoBean" scope="prototype"/>

<!-- etc... -->

Annotation:

@Configuration

public class AppConfig {

@Bean

@Scope("singleton") // This is not required since this is implicit behavior

public ExampleBean exampleBean() {

// ...

}

@Bean

@Scope("prototype")

public ProtoBean protoBean() {

// ...

}

// etc...

}

### Custom Scope

The above details the common recognized scopes, the bean scoping mechanism is extensible, allowing you to define your own custom scopes, or redefine an existing one. Note that the singleton and prototype scopes are exempt from this, and you cannot override these definitions; furthermore, it is considered bad practice to override the existing scopes at all.

To create your own custom scope you must implement the [org.springframework.beans.factory.config.Scope interface](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/config/Scope.html). This interfacce has four methods used to get and remove objects from the scope as well as register the object for later destruction. The method signatures are:

Object get(String name, ObjectFactory objectFactory)

This method returns the object from the underlying scope. For instance, a bean in the session scope would return that session-scoped bean, and if it does not already exist, will create a new instance of the bean after binding it to the session.

Object remove(String name)

This method removes an object from an underlying scope and return that object. For instance, in the previous example, this would remove the session-scoped bean from the associated session. This method should return null of the specified bean name cannot be found.

void registerDestructionCallback(String name, Runnable destructionCallback)

This method registers the destruction callback methods for the specified object when it is to be destroyed.

String getConversationId()

This method should return a particular identifier used for the object. Each scope utilized a different identification method. For instance, session scope implementations may use the session identifier.

After creation of a custom scope, you will need to make Spring aware of the implementation. To do so, you must invoke the following method from the ConfigurableBeanFactory interface, which is available on most standard ApplicationContext implmentations:

void registerScope(String scopeName, Scope scope);

Here, the first argument is a unique name associated with the custom scope. The second argument should be an actual instance of the custom scope imlpementation.

For example:

Scope revatureScope = new RevatureSpringScope();

beanFactory.registerScope("revScope", revatureScope);

After which you can configure new bean definitions with this custom scope: XML:

<bean id="revatureBean" class="com.revature.example.RevatureBean" scope="revScope"/>

Annotation:

@Configuration

public class AppConfig {

@Bean

@Scope("revScope")

public RevatureBean revatureBean() {

// ...

}

}

**Annotations**

# **Spring Module - XML vs Annotation vs Java Configuration**

This page details XML-based and Annotation-based configurations for the ApplicationContext as well as Java configurations of Spring Beans.

### References

* [Spring 4.0.x Documentation - IoC Container & Image](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html)
* [Spring 4.0.x API Docs](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/overview-summary.html)

## Bean Factory & Application Context

The Spring [Bean Factory](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/beans/factory/BeanFactory.html) interface establishes the underlying basis for the Spring IoC Container functionality by providing advanced configurations capable of managing any type of object. The ApplicationContext is a sub-interface of the BeanFactory interface which adds easier integration with Spring's AOP features. This ApplicationContext represents the Spring IoC container as it is responsible for the instantiation, assembly and management of "beans". These "Beans", are the term used for objects created in Spring which form the backbone of an application. Though they are created and managed by the IoC Container, they are otherwise just a type of object in an application. When bean objects have certain dependencies, these are reflected within configuration metadata used by the container.

In other words, the BeanFactory provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality for to the creation and management of objects in the Spring framework.

With regard to the configuration metadata, there are three ways this configuration can be represented, through XML, Java annotation or Java Code.

The following image displays a high-level representation of how Spring works:



The application classes are combined with configuration data when the ApplicationContext is created and initialized, which produces a fully configured application, or system.

It is worth noting that the BeanFactory instantiates beans lazily (creating beans only when required), while ApplicationContext instantiates beans eagerly (once the bean becomes relevant, and may potentially be used).

## XML Configuration

XML configuration consists of the declaration of at least one element nested inside a top-level element. These bean definitions correspond to actual objects within your application, and typically relate to server layer, data access (DAO), infrastructure objects (such as hibernate SessionFactories), or presentation objects (such as Struts Action instances). The following code is an example of a basic bean configuration using XML:

<!-- This XML file defines service beans and is named 'services.xml' -->

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

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

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

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

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

<!-- services -->

<bean id="accountServ" class="com.revature.services.AccountServiceImpl">

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

<!-- additional collaborators and configuration for this bean go here -->

</bean>

<bean id="orderServ" class="com.revature.services.OrderServiceImpl">

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

<!-- additional collaborators and configuration for this bean go here -->

</bean>

<bean id="itemServ" class="com.revature.services.ItemServiceImpl">

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

<!-- additional collaborators and configuration for this bean go here -->

</bean>

<!-- more bean definitions for services go here -->

</beans>

<!-- This XML file defines DAO beans and is named 'daos.xml' -->

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

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

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

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

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

<bean id="accountDao" class="com.revature.dao.AccountDaoImpl">

<!-- additional collaborators and configuration for this bean go here -->

</bean>

<bean id="orderDao" class="com.revature.dao.OrderDaoImpl">

<!-- additional collaborators and configuration for this bean go here -->

</bean>

<bean id="itemDao" class="com.revature.dao.ItemDaoImpl">

<!-- additional collaborators and configuration for this bean go here -->

</bean>

<!-- more bean definitions for data access objects go here -->

</beans>

Here, the id attribute is a string used to identify specific bean definitions, while the class attribute defines the type of bean using the fully qualified class name. Note that you can substitute the id attribute with the name attribute if you would like to provide multiple aliases for a bean.

To instantiate a Spring IoC container using this XML configuration, the location path or paths should be supplied to an ApplicationContext Constructor. For XML configurations the ApplicationContext object should be derived from the [ClassPathXmlApplicationContext](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html) class:

// Single XML Path File

ApplicationContext context = new ClassPathXmlApplicationContext(new String ("services.xml"));

// Multiple XML Path Files

ApplicationContext context = new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});

## Annotation Configuration

Annotation configuration, introduced as of Spring 2.5, serves as an alternative to XML-based setups, and rely on the use of annotations and bytecode metadata to wire components as opposed to the used of angle-bracket element declarations. Utilization of annotation based configuration still requires the use of an ApplicationContext XML file; however, the bean configurations are separated from this XML file into the component class files themselves and are defined using relevant annotations on the class, method or field declaration.

To use annotation configuration the context tag will be required within the ApplicationContext configuration file. This element enables the use of annotations which is, by default, turned off. Additionally, you will need to include the xml namespace "context". For Example:

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

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

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

<!-- The context namespace (below) is required for context element -->

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

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

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

http://www.springframework.org/schema/context

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

<!-- The following enables use of annotation-based configurations to be detected within your class files -->

<context:annotation-config/>

<!-- Bean definitions -->

<bean id = "account" class = "com.revature.models.Account">

</bean>

<bean id = "order" class = "com.revature.models.Order">

</bean>

<bean id = "item" class = "com.revature.models.Item">

</bean>

</beans>

To properly configure beans, use of the @Autowired annotation should be used. This annotation injects beans based on their type, and it commonly used with singleton bean design patterns.

@Autowired [Property]

package com.revature.models;

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

public class Order {

private Account account;

private int id;

// @Autowired on properties can take the place of setter methods

@Autowired

private Item item;

public Item getItem() {

return item;

}

...

}

@Autowired [Setter]

package com.revature.models;

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

public class Order {

private Account account;

private int id;

private Item item;

@Autowired

public void setItem(Item item) {

this.item = item;

}

...

}

@Autowired [Constructor]

package com.revature.models;

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

public class Order {

private Account account;

private int id;

private Item item;

public Order() {

super();

// TODO Auto-generated constructor stub

}

@Autowired // multiple beans are distinguished by object type

public Order(Account account, int id, Item item) {

super();

this.account = account;

this.id = id;

this.item = item;

}

...

}

If multiple bean definitions exist for a particular object type the @Resource or @Qualifier annotations can disambiguate the bean. @Resource is a standard Java annotation which injects beans based on name, rather than type. The @Qualifier annotation specifies a particular @Autowired-annotated bean based on the bean's identifier (the bean name). As such, the @Qualifier annotation is a Spring annotation:

@Resource

@Resource("item\_one") // if multiple items exist...

public void setItem(Item item) {

this.item = item;

}

@Qualifier

@Autowired

public Order(@Qualifier("account")Account account, int id, Item item) {

super();

this.account = account;

this.id = id;

this.item = item;

}

@Required

package com.revature.models;

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

public class Order {

private Account account;

...

@Required

public void setAccount(Account Account) {

this.Account = account;

}

...

}

Note that <context-annotation-config> only looks for annotations on beans within the same application context in which it is defined. This means if you put <context:annotation-config/> within a WebApplicationContext for a DispatcherServlet, it will only check for @Autowired beans within these controllers, but not beans in your services. You can otherwise specify packages using the <context:component-scan/> element with a list of packages:

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

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

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

<!-- The context namespace, below, is required for context element -->

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

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

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

http://www.springframework.org/schema/context

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

<context:annotation-config/>

<!-- Component Scan informs Spring where to look for bean wiring configurations -->

<context:component-scan base-package="com.revature.services;com.revature.dao;com.revature.models">

<!-- Bean definitions -->

...

</beans>

The method to instantiate the Spring IoC container using annotation configuration is similar to the XML-based configuration. The location path should be specified in an ApplicationContext Constructor using the [ClassPathXmlApplicationContext](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html) class:

ApplicationContext context = new ClassPathXmlApplicationContext(new String ("applicationContext.xml"));

### @Inject and @Named

Java also provides standard annotations which can be used to inject object dependencies. These annotations, @Inject and @Named require the use of the javax.inject package. Similarly to @Autowired, @Inject can occur at the class, field and constructor argument levels, based on the bean type. The @Named annotation can serve the same purpose as the @Qualifier annotation to disambiguate between bean definitions, and additionally be used as a bean definition equivalent to the @Component annotation.

@Inject

package com.revature.models;

import javax.inject.Inject;

import javax.inject.Named;

public class Order {

private Account account;

private int id;

private Item item;

@Inject

public void setAccount(Account account) {

this.Account = account;

}

}

@Named

package com.revature.models;

import javax.inject.Inject;

import javax.inject.Named;

@Named("order")

public class Order {

private Account account;

private int id;

private Item item;

@Inject // Similarly to @Autowired, type is used to distinguish beans when a qualifier (@Named) is not used

public void setAccount(Account account) {

this.Account = account;

}

...

}

**NOTE**: Annotation injection is performed prior to XML injection. As such, the use of both will result in XML configurations overriding Annotation configurations.

## Java Configuration

Java-based configuration also uses annotations, but do not require the use of an application context file. Instead, configuration management is handled by @Configuration-annotated class files which define @Bean-annotated methods used to describe the beans.

Note that you can use @Bean-annotated methods with any class which is @Component-annotated; however, they are most commonly used with @Configuration beans since the primary purpose for a class which is @Configuration-annotated is the definition of beans. In order to autowire or inject primitives, the @ComponentScan annotation specifies where to search for defined beans. Additionally all of these beans must be imported from the org.springframework.context.annotation package.

The following example shows an illustration of the Java Configuration method:

package com.revature.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Scope;

import com.revature.dao.AccountDao;

import com.revature.dao.AccountDaoImpl;

import com.revature.dao.OrderDao;

import com.revature.dao.OrderDaoImpl;

import com.revature.dao.ItemDao;

import com.revature.dao.ItemDaoImpl;

import com.revature.services.AccountService;

import com.revature.services.AccountServiceImpl;

import com.revature.services.OrderService;

import com.revature.services.OrderServiceImpl;

import com.revature.services.ItemService;

import com.revature.services.ItemServiceImpl;

@Configuration

@ComponentScan({ "com.revature.repository;com.revature.services" })

public class AppConfig {

// =====================Setter Injection======================

@Bean(name="accountDao")

public AccountDao getAccountDao() {

return new AccountDaoImpl();

}

@Bean(name="accountService")

public AccountService getAccountService() {

AccountServiceImpl aserv = new AccountServiceImpl();

as.setAccountDao(getAccountDao()); //<---setter injection

return aserv;

}

//=================Constructor Injection=================

@Bean(name = "orderDao")

public OrderDao getOrderDao() {

return new OrderDaoImpl();

}

@Bean(name = "orderService")

public OrderService getOrderService() {

OrderServiceImpl oserv = new OrderServiceImpl(getOrderDao()); //<--- consturctor injection

return oserv;

}

@Bean(name = "itemDao")

public ItemDao getItemDao() {

return new ItemDaoImpl();

}

@Bean(name = "itemService")

public ItemService getItemService() {

ItemServiceImpl iserv = new ItemServiceImpl(getItemDao());

return iserv;

}

}

Instantiation for the Spring IoC container using Java configuration uses the [AnnotationConfigApplicationContext](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/context/annotation/AnnotationConfigApplicationContext.html) class with the @Configuration-annotated class as an argument in the constructor call:

ApplicationContext appContext = new AnnotationConfigApplicationContext(AppConfig.class);

### Summary of Annotations [Alphabetical]:

* @Autowired - Spring Annotation used to inject bean dependencies as needed, based on bean type.
* @Bean - Spring annotation which plays the same role as the element in an XML-based configuration, and you can specify an identifier for these beans with the use of the id (or name) property.
* @Component - Generic [stereotype annotation](https://app.revature.com/) used to declare an object as a bean.
* @ComponentScan - Spring Annotation which specifies path locations for defined beans to be used for potential injection.
* @Configuration - Spring annotation indicates a class file used to manage bean configurations using Java configuration similar to the Application Context file for an XML or annotation based configuration.
* @Inject - Standard Java annotation used to inject bean dependencies as needed. Equivalent to Spring's @Autowired annotation.
* @Named - Standard Java annotation for disambiguating beans based on bean name. Equivalent to Spring's @Qualifier annotation. Additionally can be used as an equivalent to Spring's @Component annotation to define beans as well.
* @Qualifier - Spring Annotation which can be used in conjunction with @Autowired to disambiguate multiple beans of a defined type.
* @Required - Standard Java annotation which indicates that an affected bean property must be populated at configuration time through an explicit property value in the bean's definition.
* @Resource - Standard Java Annotation used to inject bean dependencies based on bean name, rather than type.

Day – 2

**Spring Boot**

# **Spring Boot Module - Spring Boot**

Spring Boot is an open source Spring Framework project used to rapidly create Java based, production-grade applications utilizing Spring Framework's IOC and module integrations. Spring Boot simplifies the process of project startup and framework integration by applying highly opinionated auto-configurations for the different Spring Boot project modules like webmvc, data, security, messaging, etc... .

Spring Boot has really become popular as more and more companies create or convert their systems using microservice patterns.

### References

* [Spring Boot docs](https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started.html#getting-started-introducing-spring-boot)
* [AutoConfiguration with Spring Boot](https://docs.spring.io/spring-boot/docs/1.3.8.RELEASE/reference/html/using-boot-auto-configuration.html)

### Spring Boot Dependency Management

Needless to say Spring and Spring Boot require an increasing number of dependencies. Spring Boot projects are built using dependency packages. The basic Spring Boot dependency packages are included in the project configuration (pom.xml or build.gradle) file and are called spring-boot-starter-\*. Including one spring-boot-starter-\* is enough to get your project started as it will include all the necessary pieces required to start developing and testing your solutions. A basic Spring Boot application configuration will container 4 parts:

* parent configuration
* spring-boot-starter
* spring-boot-starter-test
* spring-boot-maven-plugin

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.3.1.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.ex</groupId>

<artifactId>phase-1</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>phase-1</name>

<description>Spring boot demo phase 1</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

<exclusions>

<exclusion>

<groupId>org.junit.vintage</groupId>

<artifactId>junit-vintage-engine</artifactId>

</exclusion>

</exclusions>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

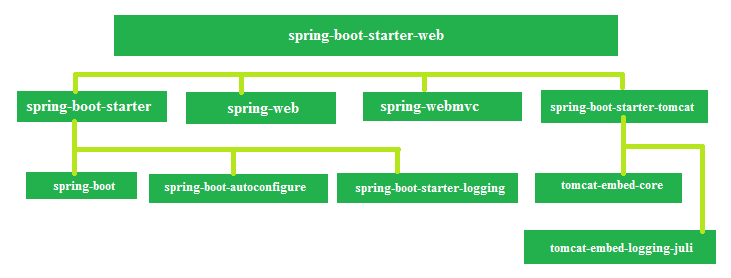
</plugins>

</build>

That will be all that you need for a basic Spring Boot project. Lets illustrate the difference between a basic project and more complicated one. A simple Spring Boot Web application will also have 4 parts:

* parent configuration
* spring-boot-starter-webmvc
* spring-boot-starter-test
* spring-boot-maven-plugin

And that is all the, spring-boot-starter-webmvc dependency replaces the spring-boot-starter dependency, but it contains all of the dependencies needed for the basic application and the web application.



In the diagram you can see how the spring-boot-starter-webmvc is an extension of the spring-boot-starter.

## AutoConfiguration

As we have covered the core Spring Framework and how to integrate different modules and wire dependencies, the biggest draw to using Spring Boot is the autoconfiguration. Spring Boot takes a highly opinionated stance on the module integrations reducing the amount of Spring Configuration needed to start developing and testing your solutions. Imagine integrating webmvc without having to start a server, or register your dispatcher servlet, or register your internal view resolver. Spring Boot automatically assumes you need all of these beans, creates them and wires them together. Better yet it automatically prescribes the solutions and its alternatives.

To further elaborate on this example if you include the Spring Boot WebMVC module you will automatically get:

* Embedded Tomcat Servlet Container (Jetty or UnderTow)
* Standard DispatcherServlet
* JSP Internal View Resolver

To enable auto configuration in Spring Boot, you need to include @EnableAutoConfiguration or @SpringBootApplication on a class. If you use @EnableAutoConfiguration then the class that the annotation is on must also be annotated with @Configuration. @SpringBootApplication doesn't require the same treatment, but it is important to note @SpringBootApplication is the combination of the @SpringBootConfiguration, @EnableAutoConfiguration, and @ComponentScan annotations.

The Spring team have created a tool call [Spring Initializr](https://start.spring.io/) to help create Spring Boot application. Any application created using Spring Initializr either through the site or IDE plugin/integrations, will be create with a main class that is already annotated with @SpringBootApplication

## Overriding Configuration/Custom Configuration

Beside the features surrounding automatic configuration, Spring Boot offers a couple of other ways to configure the application in a manual way. Manual/Custom configuration can be done through either Java code configuration or application.properties.

Supplying custom configuration in the application.properties is the easier but less dynamic way of application configuration. The application.properties file is a set of key/value pairs that supply values for module beans or your own application beans.

### Overriding some Module bean configuration (application.properties)

server.port=80

server.servlet.context-path=/myapp

spring.application.name=My Super Awesome Application

The above snippet will override the server port of the embedded server and the application context of the embedded server, also it will change the default application name.

Using Java code configuration is a bit more complicated, but it is more dynamic than application.properties. Using code configuration requires the developer to understand a lot about the associated beans in the module in order to override the default configuration.

### Configuring Hibernate beans in code

@Configuration

@EnableTransactionManagement

@EnableJpaRepositories

**public** **class** **HibernateBeanConfiguration** {

}

## Running a Spring Boot Application

Running a Spring Boot application is just a matter of either configuring your IDE to run the main method of the main class or using the included maven wrapper or gradle wrapper with the command ./mvnw spring-boot:run or ./gradlew bootRun respectively. It is important though to highlight the main method of the main class

**public** **static** **void** **main**(**String**[] args) {

**SpringApplication**.**run**(**Application**.**class**, args);

}

The main method invokes the SpringApplication.run method which is passed the class type of the main class. This starts the Spring Boot application using the main class as the root class for configuration and component scanning.

It is super important to remember to keep classes and interfaces inside of the same root package of the project. Otherwise it will be required to include an additional @ComponentScan annotation to enumerate the other packages for scanning.

### Spring Boot CLI

Another, but not required tool, of Spring Boot is the Spring Boot CLI. The Spring Boot CLI allows for even faster application development or prototyping. The Spring Boot CLI runs groovy scripts, which gives Java-like syntax, but reduces the boilerplate code. [Spring Boot CLI](https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-cli.html)

**MVC Architecture and Annotations**

# **Spring Module - MVC Architecture**

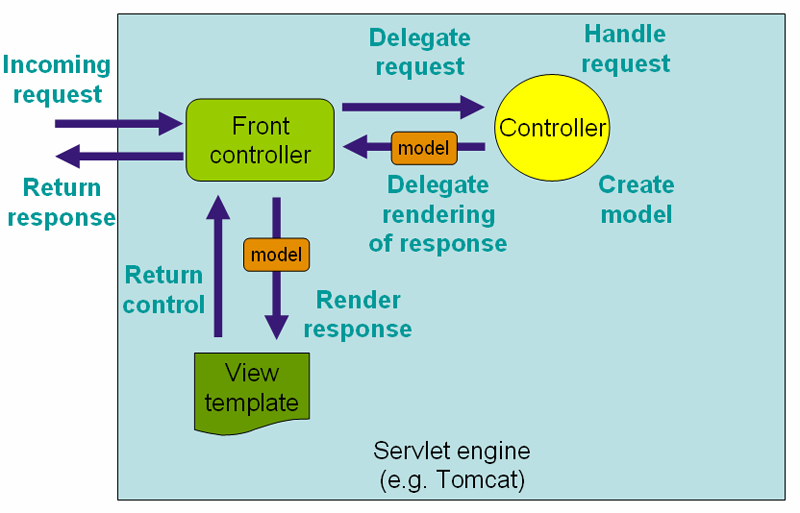
This page details an overview of the Spring MVC Framework and explains how to utilize this framework to create a Spring based Web application

### References

* [Spring 4.0.x Documentation - Web MVC Framework](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/mvc.html)
* [Spring 4.0.x API Docs](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/overview-summary.html)

## Spring Model View Controller Architecture

The Spring MVC Framework, like many other MVC Frameworks, is a request driven structure, designed around a central Servlet which dispatches requests to controllers and offers functionality to support the web application. As a breif recap, Models represent data to be stored and/or retrieved from a database or environment, the View is the user interface and displays model data and the controller handles requests between the model and responds with appropriate views. Unlike other MVC frameworks, the Spring DispatcherServlet is completely itegrated with the Spring IoC container, allowing the use of every other Spring feature in addition to this functionality. A high level overview of this request processing structure can be viewed below:



### Dispatcher Servlet and MVC

The Spring Web Model-View-Controller (MVC) Framework is based on the use of a DispatcherServlet, which uses a FrontController design pattern. This DispatcherServlet routes requests to configurable handlers, view resolutions, and also provides support for uploading files. The default controller is based on the @Controller stereotype annotation, as well as the @RequestMapping annotation, both of which are detailed in the notes below.

The DispatcherServlet, as you may guess, is an actual Servlet. As such, it must inherit it's functionality from the HttpServlet base class, and is declared in the web.xml file of your application. Any requests you want the DispatcherServlet to handle must also be mapped in the same web.xml file. As a reminder, Servlets require a top-level Servlet element in which servlet-name, servlet-class and other optional elements (such as load-on-startup) are nested. Request Mappings utilize the top-level servlet-mapping element with nested servlet-name and url-pattern elements

For Example:

<web-app>

<servlet>

<description>This is an optional description of this servlet. An example servlet</description>

<servlet-name>exampleServlet</servlet-name>

<servlet-class>com.revature.web.Dispatcher</servlet-class>

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

</servlet>

<servlet-mapping>

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

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

</servlet-mapping>

</web-app>

In this example, all requests which start with /example will be handled by the Dispatcher, which is named exampleServlet.

### ViewResolvers and InternalResourceView

Before Discussing annotations, it is worth briefly discussing how views are resolved. All Spring Web MVC controllers must resolve to a logical view name, either explicitly (for example, returning a String, View or ModelAndView) or implicitly (i.e. based on conventions). In Spring, these views are resolved by a view resolver using a logical view name. Spring has a number of view resolvers, though we will only breifly discuss a few (more information can be found in the [Spring Documentation](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/mvc.html#mvc-viewresolver)):

* AbstractCachingViewResolver: Abstract view resolver which, when extended, can cache views.
* XmlViewResolver: Implementation of ViewResolver that accepts an XML configuration file with the same DTD as Spring’s XML bean factories. The default configuration file is /WEB-INF/views.xml.
* UrlBasedViewResolver: Implementation of the ViewResolver which directly resolves view names to URLs, without an explicit mapping definition. Useful if if your logical names match the names of your views.
* InternalResourceViewResolver: Subclass of UrlBasedViewResolver that supports InternalResourceView (such as Servlets and JSPs) and subclasses (such as JstlView and TilesView). View classes for all views generated by this resolver can be set using setViewClass(..) Method. See the [UrlBasedViewResolver javadocs](https://docs.spring.io/spring-framework/docs/4.0.x/javadoc-api/org/springframework/web/servlet/view/UrlBasedViewResolver.html) for more details.

The InternalResourceView is a Wrapper class for JSP or other resources within the same web application. It can be used to expose model objects as request attributes and forward requests to a specified resource URL using the RequestDispatcher. In particular the InteralResourceViewResolver supports definitions for InternalResourceViews for a simple and straightforward resolution of @Controller class methods.

For example:

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

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

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

</bean>

### Spring Web Annotations

As discussed in the [stereotype annotation lecture notes](https://app.revature.com/), the @Controller annotation specifies a class as a controller, a special type of bean used particularly to handle web application requests. The introduction of the @Controller annotation, as of Spring 3.0, allows for the creation of RESTful Web services and applications through the use of the @PathVariable annotation as well as other flexible features. The dispatcher will scan these @Controller-annotated and @RequestMapping-annotated classes and detect any @RequestMapping-annotated methods. Depending on the context, this @RequestMapping annotation can be configured with url mapping, http request method types, url values, and other parameters.

For example (Example from Spring Documentation):

@Controller

@RequestMapping("/appointments")

public class AppointmentsController {

private final AppointmentBook appointmentBook;

@Autowired

public AppointmentsController(AppointmentBook appointmentBook) {

this.appointmentBook = appointmentBook;

}

@RequestMapping(method = RequestMethod.GET)

public Map<String, Appointment> get() {

return appointmentBook.getAppointmentsForToday();

}

@RequestMapping(value="/{day}", method = RequestMethod.GET)

public Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO.DATE) Date day, Model model) {

return appointmentBook.getAppointmentsForDay(day);

}

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

public AppointmentForm getNewForm() {

return new AppointmentForm();

}

@RequestMapping(method = RequestMethod.POST)

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

if (result.hasErrors()) {

return "appointments/new";

}

appointmentBook.addAppointment(appointment);

return "redirect:/appointments";

}

}

In this example, requests to the AppointmentsController are indicated by the "/appointments" url, and may handle a combination of GET as well as POST HTTP methods depending on the information provided. If a GET request is made simply to /appointments, then the controller will return a Map using the getAppointmentsForToday() method. If the GET request is made to the url "/appointments/new", then an AppointmentForm will be returned using the AppointmentForm() method.

Additionally, the special redirect prefix used here allows for a UrlBasedViewResolver (and subclasses) to recognize that a redirect is needed, and may resolve the view using the logical name. When using this redirect prefix supports both relative and absolute URLs: relative\*:

return: "redirect:appointments/new";

\*We used a relative path above

absolute

return: "https://revature.com"";

### @RequestBody

After defining a Request Mapping, there are a number of ways to detect and utilize variables sent with the request. For information within the body of the request, you should use @RequestBody, which allows Spring to map data from the body of a request to a defined java object.

REQUEST:

<form action="loginExample" method="POST">

<input type="text" name="username" value="Revature">

<input type="password" name="password" value="password">

<input type="submit" value="Post Data">

</form>

LOGIN OBJECT:

public class LoginForm {

private String username;

private String password;

// constructors, getters, setters, etc...

}

CONTROLLER:

@Controller("examplePost")

public class ExampleController {

...

@RequestMapping(value="/loginExample", method=RequestMethod.POST)

@ResponseStatus(value = HttpStatus.OK)

public void postData(@RequestBody LoginForm loginForm) {

System.out.println("Login values [username]: " + loginForm.getUsername()

+ ", [password]: " + loginForm.getPassword());

}

}

### @ResponseStatus

When a specific view is not returned, the logical name used is based on the value of the @RequestMapping, however you should still provide some response. The @ResponseStatus annotation marks a method or exception class with an HTTP response status code, as well as the reason that should be returned. These examples use this annotation primarily for debugging purposes, but this annotation can be useful when you do not expect to return a particular view from a controller.

### @RequestParam

Alternatively the @RequestParam annotation should be used to bind request parameters to specific values in the mapped method's signature. You may specify values when using @RequestParam either using the name attribute, or simply with the name (or value) of the parameter. REQUEST:

<form action="loginExample" method="POST">

<input type="text" name="username" value="Revature">

<input type="password" name="password" value="password">

<input type="submit" value="Post Data">

</form>

CONTROLLER:

@Controller("examplePost")

public class ExampleController {

...

@RequestMapping(value="/loginExample", method=RequestMethod.POST)

@ResponseStatus(value = HttpStatus.OK)

public void postData(@RequestParam(name="username") String username, @RequestParam("password") String password) {

System.out.println("Login values [username]: " + username + ", [password]: " + password);

}

}

The @RequestParam annotation also has a required attribute which dictates whether the specified parameter must be present for the request to complete. By default, this parameter is set to true, but can be optionally set to false: @RequestParam(name="id", required="false"). Note that all of these examples could be expressed with the value attribute instead of the name attribute, which is functionally the same: @RequestParam(value="id", required="false").

### @PathVariable

Finally, the @PathVariable annotation can identify properties directly within the URL of a request for use within a method. To map the url parameter, the parameter should be enclosed in curly braces within the RequestMapping value and the name should match the method parameter:

CONTROLLER:

@Controller

public class LibraryController {

...

@RequestMapping(value="/library/{bookISBN}", method=RequestMethod.GET)

public String getBookName(@PathVariable int bookISBN) {

switch (bookISBN) {

case 12345:

return "harry-potter";

case 54321:

return "huckleyberry-finn";

default:

return "error";

}

}

}

### @RequestHeader

In addition to request parameters and body content, header data can be viewed and mapped to a method in a similar syntax to the @RequestParam annotation. This mapping uses, as you may expect, the @RequestHeader annotation.

For example: HEADER:

Host localhost:8080

Accept text/html,application/xhtml+xml,application/xml;q=0.9

Accept-Language fr,en-gb;q=0.7,en;q=0.3

Accept-Encoding gzip,deflate

Accept-Charset ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Keep-Alive 300

CONTROLLER:

@RequestMapping("/displayHeaderInfo.do")

public void displayHeaderInfo(@RequestHeader("Accept-Encoding") String encoding,

@RequestHeader("Keep-Alive") long keepAlive) {

//...

}

### @ResponseBody

Configuration for responding to Requests can also be simplified with the Spring MVC Framework through the use of the @ResponseBody annotation. This annotation can be used to automatically marshal java objects into a JSON String.

CONTROLLER:

@Controller

public class LibraryController {

@Autowired

private LibraryService libraryService;

...

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

public @ResponseBody List<Book> getAllBooks() {

return libraryService.getAllBooks();

}

}

This method would return a list of Book Objects as a JSON string, allowing the response to be parsed and displayed on a webpage.

### @RestController

Commonly, Spring Web Controllers implement a REST API, thus serving JSON, XML or custom MediaType content. As such, the @RestController annotation is a convenience stereotype annotation which combines the features of @RequestMapping with the @ResponseBody annotations. This allows the use of HTTP method specific Mapping annotations which automatically produce an XML, JSON or other response.

CONTROLLER:

@RestController

@RequestMapping("rest-library")

public class LibraryRestController {

@Autowired

private LibraryService libraryService;

...

@GetMapping("/all")

public List<Book> getAllBooks() {

return libraryService.getAllBooks();

}

@GetMapping("/{isbn}")

public Book getBook(@PathVariable int id) {

return libraryService.getBookByISBN(isbn);

}

**MVC Architecture and Annotations**

# **Spring Module - MVC Architecture**

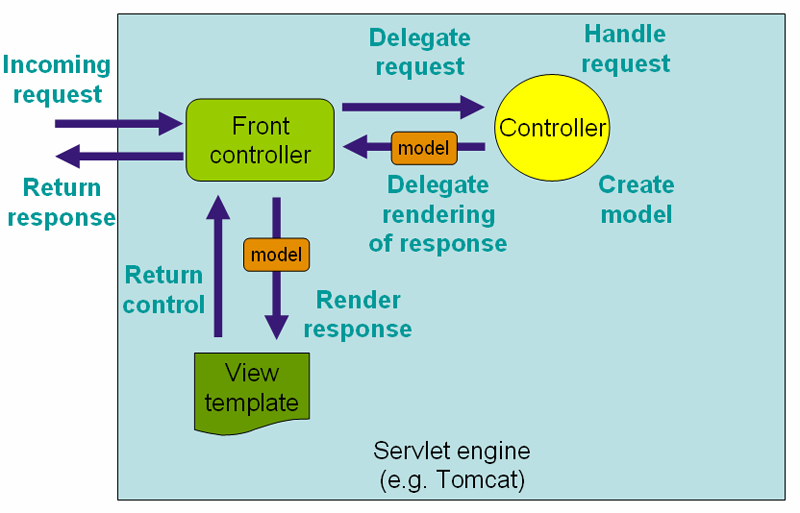
This page details an overview of the Spring MVC Framework and explains how to utilize this framework to create a Spring based Web application

### References

* [Spring 4.0.x Documentation - Web MVC Framework](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/mvc.html)
* [Spring 4.0.x API Docs](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/overview-summary.html)

## Spring Model View Controller Architecture

The Spring MVC Framework, like many other MVC Frameworks, is a request driven structure, designed around a central Servlet which dispatches requests to controllers and offers functionality to support the web application. As a breif recap, Models represent data to be stored and/or retrieved from a database or environment, the View is the user interface and displays model data and the controller handles requests between the model and responds with appropriate views. Unlike other MVC frameworks, the Spring DispatcherServlet is completely itegrated with the Spring IoC container, allowing the use of every other Spring feature in addition to this functionality. A high level overview of this request processing structure can be viewed below:



### Dispatcher Servlet and MVC

The Spring Web Model-View-Controller (MVC) Framework is based on the use of a DispatcherServlet, which uses a FrontController design pattern. This DispatcherServlet routes requests to configurable handlers, view resolutions, and also provides support for uploading files. The default controller is based on the @Controller stereotype annotation, as well as the @RequestMapping annotation, both of which are detailed in the notes below.

The DispatcherServlet, as you may guess, is an actual Servlet. As such, it must inherit it's functionality from the HttpServlet base class, and is declared in the web.xml file of your application. Any requests you want the DispatcherServlet to handle must also be mapped in the same web.xml file. As a reminder, Servlets require a top-level Servlet element in which servlet-name, servlet-class and other optional elements (such as load-on-startup) are nested. Request Mappings utilize the top-level servlet-mapping element with nested servlet-name and url-pattern elements

For Example:

<web-app>

<servlet>

<description>This is an optional description of this servlet. An example servlet</description>

<servlet-name>exampleServlet</servlet-name>

<servlet-class>com.revature.web.Dispatcher</servlet-class>

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

</servlet>

<servlet-mapping>

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

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

</servlet-mapping>

</web-app>

In this example, all requests which start with /example will be handled by the Dispatcher, which is named exampleServlet.

### ViewResolvers and InternalResourceView

Before Discussing annotations, it is worth briefly discussing how views are resolved. All Spring Web MVC controllers must resolve to a logical view name, either explicitly (for example, returning a String, View or ModelAndView) or implicitly (i.e. based on conventions). In Spring, these views are resolved by a view resolver using a logical view name. Spring has a number of view resolvers, though we will only breifly discuss a few (more information can be found in the [Spring Documentation](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/mvc.html#mvc-viewresolver)):

* AbstractCachingViewResolver: Abstract view resolver which, when extended, can cache views.
* XmlViewResolver: Implementation of ViewResolver that accepts an XML configuration file with the same DTD as Spring’s XML bean factories. The default configuration file is /WEB-INF/views.xml.
* UrlBasedViewResolver: Implementation of the ViewResolver which directly resolves view names to URLs, without an explicit mapping definition. Useful if if your logical names match the names of your views.
* InternalResourceViewResolver: Subclass of UrlBasedViewResolver that supports InternalResourceView (such as Servlets and JSPs) and subclasses (such as JstlView and TilesView). View classes for all views generated by this resolver can be set using setViewClass(..) Method. See the [UrlBasedViewResolver javadocs](https://docs.spring.io/spring-framework/docs/4.0.x/javadoc-api/org/springframework/web/servlet/view/UrlBasedViewResolver.html) for more details.

The InternalResourceView is a Wrapper class for JSP or other resources within the same web application. It can be used to expose model objects as request attributes and forward requests to a specified resource URL using the RequestDispatcher. In particular the InteralResourceViewResolver supports definitions for InternalResourceViews for a simple and straightforward resolution of @Controller class methods.

For example:

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

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

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

</bean>

### Spring Web Annotations

As discussed in the [stereotype annotation lecture notes](https://app.revature.com/), the @Controller annotation specifies a class as a controller, a special type of bean used particularly to handle web application requests. The introduction of the @Controller annotation, as of Spring 3.0, allows for the creation of RESTful Web services and applications through the use of the @PathVariable annotation as well as other flexible features. The dispatcher will scan these @Controller-annotated and @RequestMapping-annotated classes and detect any @RequestMapping-annotated methods. Depending on the context, this @RequestMapping annotation can be configured with url mapping, http request method types, url values, and other parameters.

For example (Example from Spring Documentation):

@Controller

@RequestMapping("/appointments")

public class AppointmentsController {

private final AppointmentBook appointmentBook;

@Autowired

public AppointmentsController(AppointmentBook appointmentBook) {

this.appointmentBook = appointmentBook;

}

@RequestMapping(method = RequestMethod.GET)

public Map<String, Appointment> get() {

return appointmentBook.getAppointmentsForToday();

}

@RequestMapping(value="/{day}", method = RequestMethod.GET)

public Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO.DATE) Date day, Model model) {

return appointmentBook.getAppointmentsForDay(day);

}

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

public AppointmentForm getNewForm() {

return new AppointmentForm();

}

@RequestMapping(method = RequestMethod.POST)

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

if (result.hasErrors()) {

return "appointments/new";

}

appointmentBook.addAppointment(appointment);

return "redirect:/appointments";

}

}

In this example, requests to the AppointmentsController are indicated by the "/appointments" url, and may handle a combination of GET as well as POST HTTP methods depending on the information provided. If a GET request is made simply to /appointments, then the controller will return a Map using the getAppointmentsForToday() method. If the GET request is made to the url "/appointments/new", then an AppointmentForm will be returned using the AppointmentForm() method.

Additionally, the special redirect prefix used here allows for a UrlBasedViewResolver (and subclasses) to recognize that a redirect is needed, and may resolve the view using the logical name. When using this redirect prefix supports both relative and absolute URLs: relative\*:

return: "redirect:appointments/new";

\*We used a relative path above

absolute

return: "https://revature.com"";

### @RequestBody

After defining a Request Mapping, there are a number of ways to detect and utilize variables sent with the request. For information within the body of the request, you should use @RequestBody, which allows Spring to map data from the body of a request to a defined java object.

REQUEST:

<form action="loginExample" method="POST">

<input type="text" name="username" value="Revature">

<input type="password" name="password" value="password">

<input type="submit" value="Post Data">

</form>

LOGIN OBJECT:

public class LoginForm {

private String username;

private String password;

// constructors, getters, setters, etc...

}

CONTROLLER:

@Controller("examplePost")

public class ExampleController {

...

@RequestMapping(value="/loginExample", method=RequestMethod.POST)

@ResponseStatus(value = HttpStatus.OK)

public void postData(@RequestBody LoginForm loginForm) {

System.out.println("Login values [username]: " + loginForm.getUsername()

+ ", [password]: " + loginForm.getPassword());

}

}

### @ResponseStatus

When a specific view is not returned, the logical name used is based on the value of the @RequestMapping, however you should still provide some response. The @ResponseStatus annotation marks a method or exception class with an HTTP response status code, as well as the reason that should be returned. These examples use this annotation primarily for debugging purposes, but this annotation can be useful when you do not expect to return a particular view from a controller.

### @RequestParam

Alternatively the @RequestParam annotation should be used to bind request parameters to specific values in the mapped method's signature. You may specify values when using @RequestParam either using the name attribute, or simply with the name (or value) of the parameter. REQUEST:

<form action="loginExample" method="POST">

<input type="text" name="username" value="Revature">

<input type="password" name="password" value="password">

<input type="submit" value="Post Data">

</form>

CONTROLLER:

@Controller("examplePost")

public class ExampleController {

...

@RequestMapping(value="/loginExample", method=RequestMethod.POST)

@ResponseStatus(value = HttpStatus.OK)

public void postData(@RequestParam(name="username") String username, @RequestParam("password") String password) {

System.out.println("Login values [username]: " + username + ", [password]: " + password);

}

}

The @RequestParam annotation also has a required attribute which dictates whether the specified parameter must be present for the request to complete. By default, this parameter is set to true, but can be optionally set to false: @RequestParam(name="id", required="false"). Note that all of these examples could be expressed with the value attribute instead of the name attribute, which is functionally the same: @RequestParam(value="id", required="false").

### @PathVariable

Finally, the @PathVariable annotation can identify properties directly within the URL of a request for use within a method. To map the url parameter, the parameter should be enclosed in curly braces within the RequestMapping value and the name should match the method parameter:

CONTROLLER:

@Controller

public class LibraryController {

...

@RequestMapping(value="/library/{bookISBN}", method=RequestMethod.GET)

public String getBookName(@PathVariable int bookISBN) {

switch (bookISBN) {

case 12345:

return "harry-potter";

case 54321:

return "huckleyberry-finn";

default:

return "error";

}

}

}

### @RequestHeader

In addition to request parameters and body content, header data can be viewed and mapped to a method in a similar syntax to the @RequestParam annotation. This mapping uses, as you may expect, the @RequestHeader annotation.

For example: HEADER:

Host localhost:8080

Accept text/html,application/xhtml+xml,application/xml;q=0.9

Accept-Language fr,en-gb;q=0.7,en;q=0.3

Accept-Encoding gzip,deflate

Accept-Charset ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Keep-Alive 300

CONTROLLER:

@RequestMapping("/displayHeaderInfo.do")

public void displayHeaderInfo(@RequestHeader("Accept-Encoding") String encoding,

@RequestHeader("Keep-Alive") long keepAlive) {

//...

}

### @ResponseBody

Configuration for responding to Requests can also be simplified with the Spring MVC Framework through the use of the @ResponseBody annotation. This annotation can be used to automatically marshal java objects into a JSON String.

CONTROLLER:

@Controller

public class LibraryController {

@Autowired

private LibraryService libraryService;

...

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

public @ResponseBody List<Book> getAllBooks() {

return libraryService.getAllBooks();

}

}

This method would return a list of Book Objects as a JSON string, allowing the response to be parsed and displayed on a webpage.

### @RestController

Commonly, Spring Web Controllers implement a REST API, thus serving JSON, XML or custom MediaType content. As such, the @RestController annotation is a convenience stereotype annotation which combines the features of @RequestMapping with the @ResponseBody annotations. This allows the use of HTTP method specific Mapping annotations which automatically produce an XML, JSON or other response.

CONTROLLER:

@RestController

@RequestMapping("rest-library")

public class LibraryRestController {

@Autowired

private LibraryService libraryService;

...

@GetMapping("/all")

public List<Book> getAllBooks() {

return libraryService.getAllBooks();

}

@GetMapping("/{isbn}")

public Book getBook(@PathVariable int id) {

return libraryService.getBookByISBN(isbn);

}

**Form Input with Spring**

# **Spring Module - Form Input with Spring**

This page details the basics of Gathering, parsing and returning data using HTML forms with Spring and JSP files.

### References

* [Spring With Forms Example - Baeldung](https://www.baeldung.com/spring-mvc-form-tutorial)
* [Spring With Forms Example - CodeJava](https://www.codejava.net/frameworks/spring/spring-mvc-form-handling-tutorial-and-example)
* [Spring Model Attribute](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/mvc.html#mvc-ann-modelattrib-methods)
* [Spring BindingResult](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/mvc.html#mvc-ann-methods)
* [Intro to JSP](https://www.javatpoint.com/jsp-tutorial)
* [Spring Form Tag](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/spring-form.tld.html#spring-form.tld.form)
* [Spring API Docs](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/overview-summary.html)

## Form Input and JSP

As seen in previous examples, forms are a simple and effective way to gather input from a user. Using Spring, you can easily bind form input to a modeled POJO to easily process form requests and return an appropriate response.

To examine this first create a simple Model:

**public** **class** **Account** {

**private** **long** id;

**private** **String** name;

**private** **String** type;

// Constructors, getters and setters...

}

Next, we need to create the actual form using a JSP HTML file. When establishing this file. As a refresher to Java web basics, the html views should be within a WEB-INF folder. In this example, we are not using Spring boot, so a web.xml file will still be required to configure the Spring Servlet. In your jsp file, make sure to include the spring framework form tag.

accountHome.jsp:

<!-- JSTL includes, **Spring** **Form** **Tag** -->

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

<html>

<head>

</head>

<body>

<h3>**Please** **Enter** your **Account** **Details**</h3>

<form:form method="POST"

action="/form-example/addAccount" modelAttribute="account">

<table>

<tr>

<td><form:label path="id">**Id**</form:label></td>

<td><form:input path="id"/></td>

</tr>

<tr>

<td><form:label path="name">**Name**</form:label></td>

<td><form:input path="name"/></td>

</tr>

<tr>

<td><form:label path="type">**Account** **Type**</form:label></td>

<td><form:input path="type"/></td>

</tr>

<tr>

<td><input type="submit" value="Submit"/></td>

</tr>

</table>

</form:form>

</body>

</html>

The inclusion of the Spring framework form tag is important to use the modelAttribute on the form. This modelAttribute will be used to bind our form data to our POJO through a corresponding @ModelAttribute annotation.

In addition to the modelAttribute, the various path attributes will correspond to a getter/setter method for our model (the Account class). Note that the name here is important, as the path attribute specifically looks for a method named "get" and "set". In our case, getId() and setId(), getName(), etc...

The controller should look similar to examples in the past:

**package** com.revature.controllers;

**import** com.revature.models.Account;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.ui.ModelMap;

**import** org.springframework.validation.BindingResult;

**import** org.springframework.web.bind.annotation.\*;

**import** org.springframework.web.servlet.ModelAndView;

@Controller

**public** **class** **AccountController** {

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

**public** **ModelAndView** **viewForm**() {

**return** **new** **ModelAndView**("accountHome", "account", **new** **Account**());

}

@RequestMapping(value = "/addAccount", method = **RequestMethod**.POST)

**public** **String** **submit**(@ModelAttribute("account")**Account** account,

**BindingResult** result, **ModelMap** model) {

**if** (result.**hasErrors**()) {

**return** "error";

}

model.**addAttribute**("id", account.**getId**());

model.**addAttribute**("name", account.**getName**());

model.**addAttribute**("type", account.**getType**());

**return** "accountView";

}

}

Here we are using a few new concepts, Spring's ModelAndView, BindingResult and ModelMap objects as well as the @ModelAttribute annotation.

* ModelAndView - The Spring ModelAndView object is a class which holds both a Model and View in the web MVC framework to make it possible for a controller to return both in a single return value. Note that both Model and View are still treated as distinct entities. In the example above, we are using the following overloaded version: public ModelAndView(String viewName, String modelName, Object modelObject)
* BindingResult - The BindingResult object, as its name implies, allows for binding information to a result, specifically exceptions. This allows errors to be handled by returning specific views as opposed to a thrown exception in your application.
* ModelMap - The Spring ModelMap is an implementation Java Map when building Model data.
* @ModelAttribute - As discussed previously, the @ModelAttribute annotation is used to help bind Model data to our JSP, or HTML views. Specifically @ModelAttribute can be used in two ways on a method to populate a model with attributes. The first way, as demonstrated above, adds attributes directly to the model. Alternatively you can add attributes implicitly by returning it:

@ModelAttribute

public Account addAccount(@RequestParam String id) {

return accountCreator.createAccount(id);

}

The method used depends on your current needs.

Finally, we should create two more JSPs to handle errors and view our model: error.jsp:

<html>

<head>

</head>

<body>

<h3>Oops, and Error Occurred!</h3>

<table>

<tr>

<td><a href="account">Go Back</a></td>

</tr>

</table>

</body>

</html>

accountView.jsp:

<body>

<h2>New Account Overview</h2>

<table>

<tr>

<td>ID :</td>

<td>${id}</td>

</tr>

<tr>

<td>Name :</td>

<td>${name}</td>

</tr>

<tr>

<td>Account Type :</td>

<td>${type}</td>

</tr>

</table>

</body>

**@Controller - MVC Annotations**

# **Spring Module - MVC Annotations - Controller**

This page details the @Controller and @RestController annotations used in Spring WebMVC to create Spring beans which are used to encapsulate HTTP request hanndler methods

### References

* [Spring 4.3.x Documentation - Web MVC Framework](https://docs.spring.io/spring-framework/docs/4.3.x/spring-framework-reference/html/mvc.html)
* [Spring 4.0.x API Docs (Controller)](https://docs.spring.io/spring-framework/docs/4.3.x/javadoc-api/org/springframework/stereotype/Controller.html)
* [Spring 4.0.x API Docs (Stereotypes)](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/package-summary.html)

## The Spring Annotations for MVC

The annotations used is Spring WebMVC for creating web applications are used to configure Spring beans to handle incoming request, map methods to certain request(s), bind to data in the request/response, and transform data. Below is a list of the most common annotations grouped by their purposes

* Request Handling
  + @Controller
  + @RestController
* Data Binding
  + @RequestBody
  + @PathVariable
  + @RequestParam
  + @ResponseBody
* RequestMapping
  + @RequestMapping
  + @GetMapping
  + @PostMapping
  + @PutMapping
  + @DeleteMapping
* Data Transformation
  + the consumes and produces attributes of the RequestMapping annotations.

### @Controller

A spring bean responsible for receiving requests and generating a response can be annotated with @Controller. Generally speaking a Controller is responsible for preparing a map of data that can be resolved to a view by the InternalViewResolver. This behavior can be circumvented by using the @ResponseBody data binding annotation, which then resolves the returned data directly to the body of the response and skipping the view resolution phase of the request.

@Controller is a Spring bean annotation. It is considered a specialized @Component and is part of the [Spring stereotypes](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/package-summary.html).

@Target(value=TYPE)

@Retention(value=RUNTIME)

@Documented

@Component

**public** @interface **Controller**

#### **@Controller Usage**

@Controller

**public** **class** **MyController** {...}

The code above creates a standard controller that will be expected to prepare data for view resolution by the InternalViewResolver.

@Controller

@ResponseBody

**public** **class** **MyController** {...}

The code above creates a controller in which all mapped methods will be preparing data that will be written directly to the response body.

**@RequestMapping - MVC Annotations**

# **Spring Module - MVC Annotations - Controller**

This page details the request mapping annotations used in Spring WebMVC to create Spring beans and HTTP request handler methods.

### References

* [Spring 4.3.x Documentation - Web MVC Framework](https://docs.spring.io/spring-framework/docs/4.3.x/spring-framework-reference/html/mvc.html)
* [Spring 4.3.x API Docs (RequestMapping)](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html)

## @RequestMapping

Methods and classes can be annotated with @RequestMapping to bind the class and/or method(s) to certain url paths of an incoming request. @RequestMapping also provides metadata to the DefaultHandlerMapping, which is used during the request handshake to determine with mapped method will handle the current request. The most common attributes used with @RequestMapping are:

* path - the url path(s) to which the mapped class/method will be bound
* method - the http request method(s) the class/method will be bound
* consumes - the incoming data format(s) the class/method will expect
* produces - the outgoing data format(s) the class/method can return
* headers - headers which are expected to be on the request
* params - parameters which are expected to be on the request

The @RequestMapping annotation is extended by the following semantic annotations which should be used for mapping methods rather than the general @RequestMapping.

* @GetMapping
* @PostMapping
* @PutMapping
* @DeleteMapping

The semantic annotations are configured with an @RequestMapping with the appropriate method for the request.

@Target(value=METHOD)

@Retention(value=RUNTIME)

@Documented

@RequestMapping(method=GET)

**public** @interface **GetMapping**

### @RequestMapping usage

@Controller

@RequestMapping("my\_controller")

**public** **class** **MyController** {...}

The above code creates a standard spring controller bound to the url path /my\_controller. In order to connect to this controller a request must be made to http://<domain>/<dispatcher\_mapping>/my\_controller.

@Controller

@RequestMapping("my\_controller")

**public** **class** **MyController** {

@RequestMapping(path="my\_data", method=**HttpMethod**.GET, produces=**MediaType**.APPLICATION\_JSON\_VALUE)

@ResponseBody

**public** **ResponseEntity** **getMyData**() {

...

}

}

The above code creates a controller bound to the /my\_controller url path with a handler method bound to the /my\_data url path. In order to connect to this method a request must be made to http://<domain>/<dispatcher\_mapping>/my\_controller/my\_data.

General practice is to add @RequestMapping to the controller class for mapping relative to a REST resource for example /users, /books, /employees. For the methods it is generally better to use the extended mapping annotations @GetMapping, @PostMapping, @PutMapping, and/or @DeleteMapping

@Controller

@RequestMapping("my\_controller")

**public** **class** **MyController** {

@GetMapping(path="my\_data", produces=**MediaType**.APPLICATION\_JSON\_VALUE)

@ResponseBody

**public** **ResponseEntity** **getMyData**() {

...

}

}

The above code is similar to the last example, but uses the @GetMapping annotation to map this method to HTTP get requests. Multiple extended request mappings can be used on a single method.

**Path Variables and Request Params**

# **Spring Module - MVC Annotations - Path Variable and Request Mapping**

This page details the @PathVariable and @RequestParam annotations used in Spring WebMVC to bind data to Java objects for request handling

### References

* [Spring 4.3.x Documentation - Web MVC Framework](https://docs.spring.io/spring-framework/docs/4.3.x/spring-framework-reference/html/mvc.html)
* [Spring 4.3.x API Docs (PathVariable)](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/PathVariable.html)
* [Spring 4.3.x API Docs (RequestParam)](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestParam.html)

## @PathVariable

REST endpoints can be bound to particular REST resources. Generally speaking the URL is used to refer to a uniquely addressable object. @PathVariable can be used to build dynamic routes to uniquely identified REST resources. Using @PathVariable is a two step process. First the variable path segment must be identified in the url path of the handler request mapping, then the segment is bound to a parameter in the handler's parameter list using @PathVariable Following is the list of @PathVariable attributes

* name - name of the path segment to bind to
* required - boolean to flag the variable as required
* value - alias for name

Path matching is done with regular expresions. It is possible to have simple expressions like {id}, {user\_name} or something more elegant like {id : [0-9]{3}-[0-9]{2}-[0-9]{2}-[0-9]{3}}. The first example ({id}) will match any value in the segement, but the second example will only match that specific pattern (xxx-xx-xx-xxx where 'x' is a digit 0-9).

### @PathVariable usage

@Controller

@RequestMapping("my\_controller")

@ResponseBody

**public** **class** **MyController** {

@GetMapping(path="my\_data/{id}") // {id} is the path variable

**public** **ResponseEntity** **getDataById**(@PathVariable **Integer** id){...}

}

The code above creates a controller and a handler. The handler method includes a path with a variable segment identified by {id}. The handler method's parameter list includes a parameter name id that is bound to the {id} segment of the url. Spring will automatically bind variables and segments that have the same name. It is possible to have a url segment and a binding variable with different names, if this is the case @PathVariable can be configured to use the correct segment.

@Controller

@RequestMapping("my\_controller")

@ResponseBody

**public** **class** **MyController** {

@GetMapping(path="my\_data/{id}") // {id} is the path variable

**public** **ResponseEntity** **getDataById**(@PathVariable("id") **Integer** dataId){...}

}

The above code is very similar to the previous example, but this time the segment and binding variable have different names. @PathVariable is configured to bind to the correct url segment with either the value attribute or name attribute.

@Controller

@RequestMapping("my\_controller")

@ResponseBody

**public** **class** **MyController** {

@GetMapping(path="my\_data/{id}") // {id} is the path variable

**public** **ResponseEntity** **getDataById**(@PathVariable(name="id", required=false) **Integer** dataId){

**if**(dataId != null) {

// path if id was included

} **else** {

// path if id was not included

}

}

}

The above code shows how handler methods can be written to react to optional path variables. This is useful if you have common logic that run regardless of the presence of the optional variable. IMPORTANT NOTE!!! By default all path variables are required. If a required path variable is not present the DefaultHandlerMapping will still map to this method [(REST Precendence Rules)](https://www.oreilly.com/library/view/restful-java-with/9781449361433/ch04.html#precedence_rules), but Spring WebMVC will return a 400 status for the bad request.

## @RequestParam

Query string data in an HTTP request is used to further configure and filter the request and data coming in the response. For example a request like /vehicles?make=Chevrolet&model=Silverado&color=blue is a request for a list of vehicles, but the query string further configures the request to only get vehicles with a matching make, model, and color. @RequestParam bindings are similar to @PathVarialbe binding, however the query string mapping is not included in the handler mapping path attribute. The bindings are included in the parameter list much like @PathVariable. Below is a list of @RequestParam attributes.

* defaultValue - fallback value if the is not provided or present.
* name - the name of the parameter to bind to
* required - boolean to flag the variable as required
* value - alias for name

### @RequestParam Usage

@Controller

@RequestMapping("my\_controller")

@ResponseBody

**public** **class** **MyController** {

@GetMapping(path="my\_data") // query parameters aren't included in the mapping

**public** **ResponseEntity** **getData**(@RequestParam **LocalDateTime** createdOn){...}

}

The code above creates a controller and a handler. The handler method mapping doesn't include the query parameter declarations. The parameter list however, does include the bindings with @RequestParam. Spring will automatically bind handler parameters to query parameters with the same name, but binding names can be qualified just like @PathVariable (@RequestParam("make") String manufacturer).

@Controller

@RequestMapping("my\_controller")

@ResponseBody

**public** **class** **MyController** {

@GetMapping(path="my\_data")

**public** **ResponseEntity** **getData**(@RequestParam("created") **LocalDateTime** dateCreatedOn){...}

}

The above code is very similar to the previous example, but this time we assume the query parameter and handler parameter have different names. The binding names are qualified with the name or value attribute of @RequestParam

@Controller

@RequestMapping("my\_controller")

@ResponseBody

**public** **class** **MyController** {

@GetMapping(path="my\_data")

**public** **ResponseEntity** **getDataById**(@RequestParam(name="create", required=false, defaultValue=**LocalDateTime**.**now**()) **LocalDateTime** dateCreatedOn){

...

}

}

The above code makes the create query parameter optional and assigns a default value if it isn't present in the request. This is useful to ensure proper filtering can take place, to reduce the number of query manipulations, or reduce the number of validation checks. IMPORTANT NOTE!!! By default all bound request parameter variables are required. If a required request variable is not present the DefaultHandlerMapping will still map to this method, but Spring WebMVC will return a 400 status for the bad request unless a defaultValue is provided.

Day -3

**Spring Data**

# **JPA Interfaces**

This page details the Spring Data Java Persistence API (JPA) module and its central interfaces which provides default configurations for simple CRUD applications, as well as manual custom query support.

### References

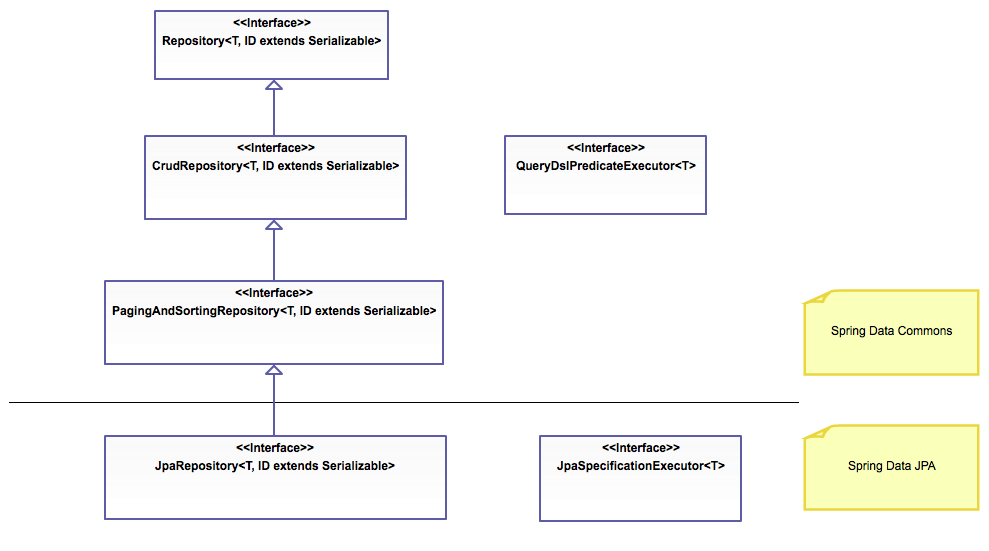
* [Spring Data JPA - Reference Documentation](https://docs.spring.io/spring-data/jpa/docs/2.3.2.RELEASE/reference/html/#preface)
* [Spring JPA - API Docs](https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html)
* [Persistence with Spring Data JPA - Baeldung](https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa)
* [Spring Data JPA - Github Repository](https://github.com/spring-projects/spring-data-jpa)
* [QueryDSL](http://www.querydsl.com/)

## Spring Data JPA

At this point, we have already utilized the Data Access Object (DAO) Layer to handle communication of data between our application and our repository. This DAO layer usually consists of a lot of boilerplate code, and as such can and should be simplified. There are numerous reasons to simplfy this code, including a decrease in the number of artifacts that need to be defined and maintained, consistency of data access patters, consistency of configuration, and quicker implementation for updated repository data.

The Spring Data module takes this simplification one step further by providing standard implementation for common DAO methods allowing for the removal of the DAO implementation and only requiring the definition of the DAO interface methods.

In order to leverage the Sping Data programming model with JPA, a DAO interface should extend the Spring JpaRepository interface from the org.springframework.data.jpa.repository package. Note that it is possible to create a basic CRUD application by extending the spring CrudRepository interface (from the org.springframework.data.repository package), though this only provides an interface for generic CRUD operations on a repository, and does not leverage the JPA.



Implementation of the Spring JpaRepository provides the following:

* Sophisticated support to build repositories based on Spring and JPA
* Support for QueryDSL predicates and thus type-safe JPA queries
  + QueryDSL is a framework which enables statically typed SQL-like queries, instead of requiring inline string queries or external XML files.
* Transparent auditing of Domain class
* Pagination (sequential numbering) support
* Dynamic query execution
* Support for integration of custom data access code
* Validation of @Query annotated queries during bootstrapping
* Support for XML based entity mapping
* JavaConfig based repository configuration by introducing @EnableJpaRepositories

Lets examine an example. First create a Spring project with the following dependencies:

* Spring Data JPA
* Spring Web (We will use this to interface with our application)
* H2 Database

If you choose to use Maven, the pom.xml file should look similar to this:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.6.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.revature</groupId>

<artifactId>JpaDemo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>JpaDemo</name>

<description>Demo project for Spring JPA - Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

If you choose to use Gradle, the build.gradle file should look similar to this:

plugins {

id 'org.springframework.boot' version '2.1.6.RELEASE'

id 'io.spring.dependency-management' version '1.0.8.RELEASE'

id 'java'

}

group = 'com.revature'

version = '0.0.1-SNAPSHOT'

sourceCompatibility = '1.8'

repositories {

mavenCentral()

}

dependencies {

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

implementation 'org.springframework.boot:spring-boot-starter-web'

runtimeOnly 'com.h2database:h2'

testImplementation('org.springframework.boot:spring-boot-starter-test')

}

test {

useJUnitPlatform()

}

Next, we must define a simple entity as a standard Bean. In this example, we will have a Customer object associated with a Customer table.

package com.revature.models;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

@Entity

public class Customer {

@Id // used to indicate field is the primary key

@GeneratedValue // used to indicate property should be auto-generated

private Integer id;

private String firstName;

private String lastName;

// default constructor - for use by Spring JPA

public Customer() {

super();

// TODO Auto-generated constructor sub

}

public Customer(String firstName, String lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

@Override

public String toString() {

return "Customer[id=" + id + ", firstName=" + firstName + ", lastName=" + lastName + "]"

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

}

Note that we have two constructors, the default no-args constructor exists for the sake of the JPA, and as such we will not use it directly. Instead, we will utilize the parameterized constructor to create instances of Customer objects to be saved to the database.

Next, we create our repository interface:

package com.revature.repository;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;

import com.revature.models.Customer;

/\*

\* Note that the generics for the JpaRepository should match the associated

\* model followed by the datatype of its id

\*/

public interface CustomerRepository extends JpaRepository<Customer, Integer> {

List<Customer> findByLastName(String lastName);

Customer findById(Integer id);

}

Next, lets create a simple controller which we will use to interface with our application

package com.revature.controllers;

import java.util.List;

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

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import com.revature.models.Customer;

import com.revature.repository.CustomerRepository;

@RestController

@RequestMapping

public class CustomerController {

@Autowired

private CustomerRepository customerRepository;

@GetMapping(value="/all")

public List<Customer> findAll(){

return customerRepository.findAll();

}

@GetMapping(value="/last/{lastName}")

public List<Customer> findByLastName(@PathVariable String lastName) {

return customerRepository.findByLastName(lastName);

}

@GetMapping(value="/id/{id}")

public Customer findById(@PathVariable Integer id) {

return customerRepository.findById(id);

}

@PostMapping(value="/add")

public Customer add(@RequestBody Customer customer) {

customerRepository.save(customer);

return customerRepository.findById(customer.getId());

}

}

Note that this controller references a findAll() and save() method on our customerRepository though we have not defined these. These methods are already predefined in the JpaRepository Interface, so the only definitions we need to provide are ones which match properties of our Customer model object. A list of JPA methods can be [found here](https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html).

Make sure you configure your application.properties file:

#Configure DB connection

spring.h2.console.enabled=true

spring.datasource.platform=h2

spring.datasource.url=jdbc:h2:mem:testdb

#Configure the port and context path for app

server.port=8082

server.servlet.context-path=/customer

Finally, create the application class:

package com.revature;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class JpaDemoApplication {

public static void main(String[] args) {

SpringApplication.run(JpaDemoApplication.class, args);

}

}

At this point you can add customers using postman to send GET and POST requests to save and retrieve customer data. If you would like to produce custom queries, you can do so using the @Query annotation in your repository interface. For example:

@Query("SELECT c FROM Customer c WHERE c.status = 1")

List<Customer> findAllActiveCustomers();

Here, we specify a "status" property for our class and utilize this to track "active" (1) or "inactive" (0) customers in our database.

Though the actual implementation of the Spring Data managed DAO is hidden (as we don't work with it directly) it is a simple enough implementation. In fact, Spring JPA provides a SimpleJpaRepository interface, which extends the JpaRepository, that defines transaction semantics using annotations. Specifically, a read-only @Transactional annotation is used at the class level, which is then overridden for the nonread-only methods. The rest of the transaction semantics are default, but can easily be overridden manually on a per-method basis.

**Spring Data Annotations**

# **Spring Data Annotations**

This page is a brief overview of the some of the Spring Data annotations. Please see the reference links for the official documentation

### References

* [Spring Data Annotations](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/annotation/package-summary.html)

## Spring Data Annotations

Spring Data abstracts away the code required for data storage solutions, allowing us to focus more on the business logic. The following is a list and brief explanation of some common Spring Data annotations which allow us to configure how the queries are executed.

| **Annotation** | **Purpose** |
| --- | --- |
| @Transactional | Configure how the database transaction behaves. See the @Transactional notes. |
| @NoRepositoryBean | Creates and interface that provides common methods for child repositories |
| @Param | Parameters can be passed to queries defined with @Query |
| @Id | Marks a field in a model class as the primary key |
| @Transient | Mark a field as transient, to be ignored by the data store engine during reads and writes |
| @CreatedBy, @LastModifiedBy | Auditing annotations that will automatically filled with the current principal |
| @CreatedDate, @LastModifiedDate | Auditing annotations that will automatically fill with current date |
| @Query | Supply a JPQL query for repository methods |

### @Transactional

See the notes on the @Transactional annotation for multiple examples

### NoRepositoryBean

@NoRepositoryBean

**public** **interface** **MyRepoBase**<T, ID **extends** **Serializable**> **extends** **JpaRepository**<T, ID> {

**Optional**<**List**<T>> **findAllByPrice**(**Double** price);

}

@Repository

**public** inteface **BookRepo** **extends** **MyRepoBase**<**Book**, **Integer**> {}

The interface MyRepoBase can now serve as a base for children repos like BookRepo. Spring will not create a bean of type MyRepoBase, but will create a BookRepo bean which will include all of the JpaRepository methods and also findAllByPrice from MyBaseRepo

### @Param

@Repository

**public** inteface **BookRepo** **extends** **MyRepoBase**<**Book**, **Integer**> {

@Query("FROM Book b WHERE b.title LIKE :title")

**Optional**<**List**<T>> **findBySimilarTitle**(@Param("title") **String** title);

}

The @Param annotation binds the method parameter to the title parameter of @Query.

### @Id

This is essentially the same as the JPA annotation.

### @Transient

This is essentially the same as the JPA annotation.

### @CreatedBy, @LastModifiedBy, @CreatedDate, @LastModifiedDate

**public** **class** **Book** {

@CreateBy

**public** **User** creator;

@LastModifiedBy

**public** **User** modifier;

@CreatedDate

**public** **Date** createdAt;

@LastModifiedDate

**public** **Date** modifiedAt;

}

### @Query

@Query("SELECT **COUNT**(\*) FROM **Book** b)

**long** **getBookInventoryCount**();

JPQL query.

@Query("FROM Book b WHERE b.title LIKE :title")

**Optional**<**List**<T>> **findBySimilarTitle**(@Param("title") **String** title);

With named parameters.

@Query("SELECT AVG(b.page\_count) FROM Book b", nativeQuery=true)

**int** **getAvgPageCount**();

Natvie SQL query.

**Validation**

# **Validation**

This page details Data Validation for a Web Application using Spring's validation package.

### References

* [Spring Validation and Data Binding](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/validation.html)
* [Spring Validation API Docs - Validator](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/validation/Validator.html)
* [Spring Validation API Docs - ValidationUtils](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/validation/ValidationUtils.html)
* [Spring Validation API Docs - Errors](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/validation/Errors.html)
* [Hibernate Validator JBoss Docs](https://docs.jboss.org/hibernate/validator/4.3/reference/en-US/html_single/)

## Validation and Spring

Validation, or the act of checking/verifying validity or accuracy of something, is an important consideration for any web application, with various pros and cons. In general validation should not be tied to the web tier, it should be easy to localize, and it should be loosely coupled with any available validators. Fortunately, Spring provides a basic yet eminently usable Validator interface which can work in every layer of an application. Additionally, when validating information, it is useful to utilize Data binding, which allows user input to be dynamically bound to custom models for each application. Within the Spring validation package, both the Validator interface as well as a DataBinder objects exist to address both. Note that Athough this package is primarily used within the MVC framework, it is not limited to it.

The Spring Validator interface, as its name implies, can be used to validate objects. The interface works using a Spring Framework Errors objects so that, while validating, validators can report these validation failures to this Errors object.

To examine this lets take a look at our Account object and create a validator for this. Account.class

**public** **class** **Account** {

**private** **long** id;

**private** **String** name;

**private** **String** type;

// Constructors, getters and setters...

}

In order to create the validation behavior, we must implement the Validator interface from the org.springframework.validation package and provide a definition for the public boolean supports(Class) method and the public void validate (Object, org.springframework.validation.Errors) method

AccountValidator.class

**public** **class** **AccountValidator** **implements** **Validator** {

/\*\*

\* This Validator validates \*just\* Person instances

\*/

**public** **boolean** **supports**(**Class** clazz) {

**return** **Account**.**class**.**equals**(clazz);

}

**public** **void** **validate**(**Object** obj, **Errors** e) {

**ValidationUtils**.**rejectIfEmpty**(e, "name", "name.empty");

validationUtils.**rejectIfEmptyOrWhitespace**(e, "type", "field.required");

**Account** a = (**Account**) obj;

**if** (a.**getId**() < 0) {

e.**rejectValue**("id", "negativevalue");

}

}

}

The ValidationUtils class provides various methods which can be used to produce a variety of Errors objects. For example, the rejectIfEmpty will register to the errors object if the value name is empty, while the rejectifEmptyOrWhitespace method registers when the type is either empty or contains whitespace. Though it is possible to implement a single validator class for an application, it is better to encapsulate validation logic for each class with its own Validator implementation.

### Spring Errors

The Spring Errors interface stores and exposes information about data-binding and validation errors for objects. This interface defines a static NESTED\_PATH\_SEPARATOR String, which details the nested path for properties on a model or bean. For example: "account.id" or "account.type" correspond to the values of that data on an "account" object. If an object is more complex (such as a map, or property of an object nested within a class), this path separator can also provide this information: "customer.address.streetName". Here, the customer object has an address object with a property called "streetName".

In addition, Errors objects can output error messages by using a MessageSource. In the examples above, this is done by using the error code given when rejecting the field. Custom logic can then be provided in your Errors object. (See the [reject method](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/validation/Errors.html#reject-java.lang.String-) in the documentation for more information).

### DataBinder and @Valid

As of Spring 3, an instance of the DataBinder object can be configured with a Validator. Once configured the Validator may be invoked by invoking the validate() method of a DataBinder object. Any validation errors will then automatically be added to a BindingResult object of the DataBinder. Before viewing an example, it it worth discussing the BeanWrapper. Fundamentally, the BeanWrapper is used in a lot of places, but direct configuration is generally not needed. However, while performing data binding it is useful to examine this concept.

As you know, beans are objects used for injection and modeling information in a Spring application. These beans have various properties, like normal Java POJOs. When utilizing beans however, an important interface used in the beans package is the BeanWrapper and its corresponding implementation BeanWrapperImpl. The BeanWrapper offers critical functionality to set and get properties values, either individually or in bulk. When establishing property values, the name of a property is used as the context for that particular value on a bean. For instance, a String name property corresponds to the getName() and setName(String) methods.

Within the Bean Factory, the getPropertyValue() and setPropertyValue() methods are used as a general way to configure bean property values with this interface during bean instantiation. The following is an example of how this data may be retrieved or manipulated:

**BeanWrapper** account = **BeanWrapperImpl**(**new** **Account**());

// setting the account name..

account.**setPropertyValue**("name", "Revature Account");

// ... can also be done like this:

**PropertyValue** value = **new** **PropertyValue**("name", "Revature Account");

account.**setPropertyValue**(value);

The DataBinder object uses this concept to bind bean values to it. For example:

**Account** target = **new** **Account**();

**DataBinder** binder = **new** **DataBinder**(target);

binder.**setValidator**(**new** **AccountValidator**());

// bind to the target object

binder.**bind**(propertyValues);

// validate the target object

binder.**validate**();

// get BindingResult that includes any validation errors

**BindingResult** results = binder.**getBindingResult**();

See the [Form-input lecture notes](https://app.revature.com/) for another example of the BindingResult object.

Moreover, as of Spring 3, the Spring MVC has the ability to automatically validate @Controller inputs through the use of the @Valid annotation so long as the validator for a model has been configured:

@Controller

**public** cass **AccountController** {

@RequestMapping("/example", method=**RequestMethod**.POST)

**public** **void** validationExample (@Valid **Account** account) {

// logic goes here

}

}

### Hibernate Validator

Although you can provide your own custom validation, the Hibernate Validator is the standard validation implementation. It is also worth noting that hibernate includes its own specific constraints and annotations, and use of this validator is **not** dependent on the hibernate ORM. To utilize this validator implementation you should add version 4.3.1 the following dependencies to your project:

<dependency>

<groupId>org.hibernate.validator</groupId>

<artifactId>hibernate-validator</artifactId>

<version>4.3.1.Final</version>

</dependency>

Note that the new standard is HibernateValidator version 6 with the JSR-380 standard.

The hibernate validator provides an implementation on various, useful constraints such as @NotNull, @Min and @Max. Thankfully the names of these annotations are intuitive to their purpose. Although the implementation of these annotations comes from the hibernate validator, you should import the annotation from the javax.validation.constratins package.

@NotNull This annotation checks that an annotated value is not null.

**import** javax.validation.constratins.NotNull;

**public** **class** **Account** {

@NotNull

**private** **long** id;

**private** **String** name;

**private** **String** type;

// ...

}

@Min and @Max These annotations check for a minimum or maximum numerical value.

**import** javax.validation.constraints.Min;

**import** javax.validation.constraints.Max;

**public** **class** **Plane** {

@Min(1)

**private** **int** pioletCount;

@Max(250)

**private** **int** passengerCount;

// ...

}

A list of more annotations can be found [here](https://docs.jboss.org/hibernate/validator/4.3/reference/en-US/html_single/#validator-defineconstraints-spec).

If you intend to use Hibernate specific annotations, which you can find [here](https://docs.jboss.org/hibernate/validator/4.3/reference/en-US/html_single/#validator-defineconstraints-hv-constraints), you must import these annotations from the hibernate.validator.constraints package. Examples of these annotations include @CreditCardNumber, which allows validation of credit card number syntax (not validity of the actual credit card) or @Range to check if a numerical value lies between a specified minimum and maximum value (inclusive).

**Transaction Propagation Strategies**

# **Transaction propagation**

The page details the general purpose usage of transaction propagation strategies, but in the context of spring with hibernate and transaction management. A link for transaction propagation with JDBC can be found in the references.

### References

* [Spring Transaction Management](https://docs.spring.io/spring-framework/docs/4.3.x/spring-framework-reference/html/transaction.html)
* [Spring Propagation Types](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html)
* [JDBC Transaction Propagation](https://docs.oracle.com/cd/A87860_01/doc/java.817/a83725/trans1.htm#1108707)

## Transaction Propagation.

When designing database transactions to be performed in Java or any other programming language other than SQL, it is important to ask this question, "How will transactions behave over multiple method calls"? If Method1 calls `Method2, how will transactions propagate. Transaction propagation falls into the following categories.

| **Strategy** | **Purpose** |
| --- | --- |
| Mandatory | Support a current transaction, throw an exception if none exists. |
| Nested | Execute within a nested transaction if a current transaction exists, behave like REQUIRED otherwise. |
| Never | Execute non-transactionally, throw an exception if a transaction exists. |
| Not Supported | Execute non-transactionally, suspend the current transaction if one exists. |
| Required | Support a current transaction, create a new one if none exists. |
| Requires New | Create a new transaction, and suspend the current transaction if one exists. |
| Supports | Support a current transaction, execute non-transactionally if none exists. |

### Spring Transaction Management

Designing transactions and ensuring consistency isn't necessarily trivial, but using a transaction manager can help aleviate some of the more complex tasks such as read/write locking and thread management. As we know Spring integrates with Hibernate with reasonable ease and integrating a transaction manager is a matter of configuring Hibernate to do so. So how is transaction propagation performed? Refer to the notes on @Transactional to learn how it is used to create contextual sessions. Below will be a discussion on the propagation attribute of the annotation which is used to setup transaction propagation.

### Propagation Usage

@Service

**public** **class** **MyService** {

...

@Transactional(readonly=true, propagation=**Propagation**.NEVER)

**public** **List**<**MyData**> **readMyData**(){...}

}

The above code use the transaction manager to create a read only, transactionless transaction state. Generally speaking readonly operations are idempotent and safe so an argument for never having a transaction present can be made. This is not necessarily a best practice.

@Service

**public** **class** **MyService**{

@Autowired

**private** **MyService2** myService2;

...

@Transactional(propagation=**Propagation**.REQUIRED\_NEW)

**public** **Integer** **createNewData**(...){

myService2.**createSupportingData**(...);

....

}

}

@Service

**public** **class** **MyService2**{

...

@Transactional(propagation=**Propagation**.MANDATORY)

**public** **Integer** **createSupportingData**(...){...}

}

The above code creates a more complicated situation. From the MyService perspective, it is easy to see that the createNewData method will require a new transaction be created, but the dependency on MyService2 complicates the operation. The example simplifies the problem by assuming that createSupportingData of MyService2 not only requires a transaction, but will fail if one isn't present. Essentially the example creates a situation where the data write for MyService is contigent on the data write from MyService2 at the transaction level, using this strategy we have created an atomic multioperational transaction which will increase the likely hood of maintaining a consistent state.

## Important Note

Transaction management design is a multipart problem. Remember database transactions should be A.C.I.D. The solution to transaction management should be as organic to your problem as possible. There are no cookiecutter solutions that will fit all problems.

**@Transactional**

# **@Transactional**

The page details the use of @Transactional in Spring to utilize transaction management and contextual sessions.

### References

* [Spring @Transactional](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html)
* [Spring Transaction Management](https://docs.spring.io/spring-framework/docs/4.3.x/spring-framework-reference/html/transaction.html)
* [JDBC Transaction Propagation](https://docs.oracle.com/cd/A87860_01/doc/java.817/a83725/trans1.htm#1108707)

## Transaction Management

When designing database transactions to be performed in Java or any other programming language other than SQL, it is important to consider that multiple transactions will performed simultaneously. Managing all of these transactions can be tedious and cumbersome becuase it requires complicated strategies for read/write locking and thread management. JDBC and connection pool utilities like Hikari have built-in tools for handling these issues, but still will require a lot of boiler plate code to achieve the consistency required for your data.

## @Transactional

When using Spring and Spring ORM or Spring Data your chosen ORM can be configured to use a transaction manager that will automatically handle transaction propagation, and isolation, commit, and rollback. Assuming you understand how to configure Spring to use Hibernate and a TransactionManager, utilizing the management tools is as simple as annotating a class or method with @Transactional.

For best practice usage @Transactional should be used on your @Service beans since a transaction is a unit of work with its success or failure directly tied to the requirements of the application and not necessrarily to the model. The side-effects of a transaction can be assumed to be either commit or rollback therefore the model will be consistent, but the @Service will be required to react accordingly.

Below is a list of the attributes for @Transactional

| **Attribute** | **Purpose** |
| --- | --- |
| isolation | The transaction isolation level. |
| noRollbackFor | Defines zero (0) or more exception Classes, which must be subclasses of Throwable, indicating which exception types must not cause a transaction rollback. |
| noRollbackForClassName | Defines zero (0) or more exception names (for exceptions which must be a subclass of Throwable) indicating which exception types must not cause a transaction rollback. |
| propagation | The transaction propagation type. |
| readOnly | A boolean flag that can be set to true if the transaction is effectively read-only, allowing for corresponding optimizations at runtime. |
| rollbackFor | Defines zero (0) or more exception classes, which must be subclasses of Throwable, indicating which exception types must cause a transaction rollback. |
| rollbackForClassName | Defines zero (0) or more exception names (for exceptions which must be a subclass of Throwable), indicating which exception types must cause a transaction rollback. |
| timeout | The timeout for this transaction (in seconds). |
| transactionManager | A qualifier value for the specified transaction. |

### @Transactional Usage

The following examples assume the usage of Spring ORM since the query structure and operations of Spring Data are abstracted away.

@Service

@Transactional // all methods in this class will utilize the tx manager and contextual sessions

**public** **class** **MyService** {

...

**public** **List**<**MyData**> **readMyData**(...){...}

}

The above code creates a service bean that utilizes the transaction manager. By annotating the class, all methods are implicated in the use of the manager.

@Service

@Transactional // all methods in this class will utilize the tx manager and contextual sessions with default configuration

**public** **class** **MyService** {

...

@Transactional(readonly=true, isolation=**Isolation**.READ\_COMMITTED, propagation=**Propagation**.NEVER) //override the default config

**public** **List**<**MyData**> **readMyData**(...){...}

}

The above code expands on the previous example. @Transactional can be used on classes or methods or both. In this example, the class uses the transaction manager with default configuration, but the method overrides the default configuration. If you are unfamiliar with the attributes being used, don't worry to much about that right now.

@Service

@Transactional // all methods in this class will utilize the tx manager and contextual sessions

**public** **class** **MyService** {

...

@Transactional(isolation=**Isolation**.SERIALIZABLE, rollbackFor={**ConstraintViolationException**.**class**}, propagation=propagation.REQUIRES\_NEW) // override when the transaction should rollback.

**public** **Integer** **createNewData**(...){...}

}

In the above example the code demonstrates how to configure the transaction manager with a list of exceptions that must cause a rollback to be triggered. Normally the transaction manager will always rollback for any RuntimeException or RuntimeException subclass (unchecked exceptions) however, the transaction manager will not automatically rollback for Exception (checked exceptions)

## Important Note

Transaction management design is a multipart problem. Remember database transactions should be A.C.I.D. The solution to transaction management should be as organic to your problem as possible. There are no cookiecutter solutions that will fit all problems.

Day 4

**Advice**

# **Spring Module - Advice**

This page details the concept of Advice with regard to Aspect Oriented Programming and Spring.

### References

* [Spring 4.0.x Documentation - Advice](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/aop.html#aop-advice)

### Prerequisites

* [AspectJ with Spring](https://app.revature.com/)
* [Pointcuts and Joinpoints](https://app.revature.com/)

## Types of Advice

Advice is specific actions taken, defined as a method, at a particular point during the execution of a program. There are five (5) types of advice, which controls at what point the action will take place.

* **Before** - Advice that will execute before a join point, but does not have the capability to halt the normal execution of the proceeding join point (unless an exception is thrown)
* **After Returning** - Advice that will execute after a join point completes without throwing an exception
* **After Throwing** - Advice that will execute if a join point throws an exception.
* **After (finally)** - Advice that will execute regardless of how the join point completes, whether normal or by throwing an exception.
* **Around** - Advice that will execute before and after the join point. Around advice is the most general, but also most powerful kind of advice, as it can perform custom behavior before and after method invocation, and can be responsible for choosing whether to proceed to the join point method execution or shortcut the advised method by returning its own value or throwing an exception. In general it is recommended to use the least powerful type of advise for any particular task. For instance, if you only need to update a cache with a value returned from a method, it is best to use the After Returning advice type.

All of these annotations, including the @Aspect annotation must be imported from the org.aspectj.lang.annotation pacakage.

### Before

Before advice is declared using the @Before annotation:

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

@Aspect

public class BeforeExample {

@Before("com.revature.example.dataAccessOperation()")

public void doAccessCheck() {

// ...

}

}

### After Returning

After Returning advice is declared using the @AfterReturning annotation:

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.AfterReturning;

@Aspect

public class AfterReturningExample {

@AfterReturning("com.revature.example.dataAccessOperation()")

public void doAccessCheck() {

// ...

}

}

Note that you can also bind @AfterReturning a returned value to the advice in order to access information from it (such as storing its value, or printing information specific to the returned data). This binding is declared using the returning attribute in the pointcut:

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.AfterReturning;

@Aspect

public class AfterReturningExample {

@AfterReturning(

pointcut="com.revature.example.dataAccessOperation()",

returning="retVal")

public void doAccessCheck(Object retVal) {

// ...

}

}

Here, "retVal" is the name given to the object being returned, and is referenced in the method signature of the doAccessCheck method by the same name.

### After Throwing

As you may assume, After Throwing advice is declared with the @After Throwing annotation:

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.AfterThrowing;

@Aspect

public class AfterThrowingExample {

@AfterThrowing("com.revature.example.dataAccessOperation()")

public void doRecoveryActions() {

// ...

}

}

Similarly to @AfterReturning, you may reference a thrown exception within the advice through the use of the throwing attribute, which allows you to reference the exception within the method signature of the advice by corresponding an argument in the method signature with the name declared with the throwing attribute:

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.AfterThrowing;

@Aspect

public class AfterThrowingExample {

@AfterThrowing(

pointcut="com.revature.example.dataAccessOperation()",

throwing="ex")

public void doRecoveryActions(DataAccessException ex) {

// ...

}

}

### After (finally)

After advice is declared using the @After annotation. Since it will execute after normal or exceptional completion of a method, you must make sure any @After advice is configured to handle both conditions:

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.After;

@Aspect

public class AfterFinallyExample {

@After("com.revature.example.dataAccessOperation()")

public void doReleaseLock() {

// ...

}

}

### Around

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Around;

import org.springframework.stereotype.Component;

@Component

@Aspect

public class AroundExample {

private static Logger log = Logger.getLogget(LoggingAspect.class);

@After("com.revature.example.dataAccessOperation()")

public void performAroundAdvice() {

log.info("This is Around Advice");

}

}

**Pointcut and Join Points**

# **Spring Module - Pointcut and Join Points**

This page details pointcuts and join points in Spring.

### References

* [Spring 4.0.x Documentation - Pointcuts](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/aop.html#aop-pointcuts)

### Prerequisites

* [AspectJ with Spring](https://app.revature.com/)

## Pointcuts and Join Points

Recall the pointcuts determine join points, in that they define which methods in our application advice ought to be injected into or around, while join points are the specific moment during the execution of a program in which the advice can be taken. Though these may sound similar, you can think of the relationship between pointcuts and join points in that pointcuts determine which methods in your application advice should take place and join points determine which package or class that advice should take place.

Pointcuts are comprised of two parts: a name, and any parameters. Additionally, keep in mind that pointcut expression determine exactly which method execution we are interested in. In Spring AOP, pointcut expressions use annotations with regular method signatures, and are indicated using the @Pointcut annotation. To use this annotation you must import the org.aspectj.lang.annotation.Pointcut class. Note that methods serving as the pointcut signature must have a return type of void.

The Spring AOP supports the following pointcut designators:

* execution - Used to match a method execution join point. This is the primary pointcut designator used. More detail on this expression can be found below.

@Pointcut("execution(public String getName())")

public void executionExample() {}

* within - Limits method execution to those of a matching type.

@Pointcut("within(com.revature.examples.\*)")

public void executionExample() {}

* this - Limits method execution where the bean referenced is an instance of the specified type.

@Pointcut("this(com.revature.examples.ExampleService)")

public void executionExample() {}

* target - Limits method execution where the target, proxied object, in an instance of a given type.

@Pointcut("target(com.revature.examples.ExampleService)")

public void executionExample() {}

* args - Limits method execution where the arguments are instances of the given type.

@Pointcut("java.io.Serializable")

public void executionExample() {}

* @target - Limits method execution where the class of the executing boject has an annotation of the given type.

@Pointcut("@target(org.springframework.transaction.annotation.Transactional)")

public void executionExample() {}

* @args - Limits method execution where the runtime type of the actual arguments passed have annotations of a given type.

@Pointcut("@args(org.springframework.transaction.annotation.Transactional)")

public void executionExample() {}

* @within - Limits method execution within types that have a given annotation.

@Pointcut("@within(org.springframework.transaction.annotation.Transactional)")

public void executionExample() {}

* @annotation - Limits method execution where the join point has a given annotation.

@Pointcut("@annotation(org.springframework.transaction.annotation.Transactional)")

public void executionExample() {}

## Parameter Patterns for execution expression

As stated previously, the execution expression is the most widely used pointcut designator. As such, it is key to understand the specific syntax for this expression:

execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern)

throws-pattern?)

Note that only the return type pattern, name pattern and parameter patterns are required. Additionally, when declaring these patterns it is possible to use wildcard indicators (\*) to substitute all or parts of these patterns. As you may guess, the return type pattern indicates a specific type of data returned by the method must match in order for the join point to be matched. Often this return type will use the , or wildcard, to specify any return type for a given method. The name pattern specifies that a name must match. Note that this name may also use a \* to substitute part or all of the name for a given method. The parameters pattern is slightly more complex. () indicates a method that takes no parameters, whereas (..) specifies a method with any number of parameters (zero or more). The wildcard \* specifies the type of parameter, while listing the specific type requires the type to match at that parameter index. For instance: (,String) matches methods which take two parameters, the first can be any type, while the second must be a String.

execution(public String getName())

This point cut specifies a public method (note this public access modifier is optional) named getName that returns String data.

execution(public \* getName())

This pointcut specifies any public method that returns any type of data, so long as the method is named getName.

execution(\* get\*())

Here, the pointcut matches a method of any return type whose name begins with 'get'

execution(\* set\*(\*))

This pointcut matches methods whose name begins with 'set' which take one argument of any type

execution(void com.revature.\*.set\*(..))

Note that you can also use fully qualified type names, and wildcards for these names as well. This pointcut matches any method who name begins with 'set' (which returns void) within any package from com.revature (such as com.revature.service, com.revature.dao, com.revature.examples, etc...), and can take any number of parameters.

**Overview of AOP and cross-cutting concerns**

# **Aspect Oriented Programming and Cross Cutting Concerns**

This page details Aspect Oriented Programming to address Cross Cutting Concerns.

### References

* [Spring 4.0.x Framework Documentation - AOP with Spring](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/aop.html)
* [Spring 4.0.x Framework Documentation - AOP API](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/aop-api.html)
* [JavaDocs API - Spring 4.0.x](https://docs.spring.io/spring/docs/4.0.x/javadoc-api/overview-summary.html)
* [AspectJ API Docs](https://www.eclipse.org/aspectj/doc/released/runtime-api/index.html)
* [AspectJ API Docs - Annotations](https://www.eclipse.org/aspectj/doc/released/aspectj5rt-api/index.html)

## Aspect Oriented Programming and Cross Cutting Concerns

Aspect Oriented Programming is another way to think about structuring your program, which can be used complementary to Object Oriented Programming. In object oriented programming, classes are used as the key component used to drive the creation of objects, which serve the purpose of representing concrete ideas or things with states and behaviors. In Aspect Oriented Programming, the key component are aspects, which modularizes particular transactional concerns which can present across multiple classes, known as cross-cutting concerns. Examples of these concerns are, Database Access (security for a database), data entities (transactions to take place), error handling, or logging system messages. The defining characteristic of these cross-cutting concerns is that they are actions that can take place across your classes, regardless of the class function or structure. In traditional object oriented programming, this would result in code redundancy, as the same code must be called multiple times throughout an application to perform these actions. Aspect Oriented Programming works to eliminate this redundancy by transferring responsibility of these common problems to aspects. Note that the Spring IoC Container is not dependent on AOP; however, the Spring AOP framework complements the IoC Container by providing a capable middleware solution for concerns.

### Important Terminology

* Aspect - A representation of a concern which cuts across multiple classes.
* Weaving - The process of linking aspects with other objects, such as beans, to create advised objects. This can be done at compile time (such as when using the AspectJ compiler), load time or runtime. Spring performs weaving at Runtime.
* Join Point - A specified moment during the execution of a program, such as the invocation of a method, in which actions can be taken.
* \_Advice \_ - Action taken by an aspect at a specified Join point.
* Pointcut - A definition of which methods in our application advice ought to be injected into or around, for example, when a method of a certain name is executed. By default, spring uses AspectJ pointcut expression language.
* Introduction - Declaration of new interfaces and corresponding implementations in subclasses of any advised object. Introductions use the @DeclareParents annotation with the defaultImpl attribute to define a default concrete class for the bean definition. For Example:

**IFunction Interface**:

package com.revature.examples;

public interface IFunctional {

public void function();

}

**FunctionalDefault**:

package com.revature.examples;

public class FunctionalDefault implements IFunctional {

public void function() {

System.out.println("This is the default function");

}

}

**Aspect**:

package com.revature.examples;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.DeclareParents;

import org.springframework.stereotype.Component;

@Component

@Aspect

public class AspectExample {

@DeclareParents(value="com.revature.examples.\*+",defaultImpl=FunctionalDefault.class)

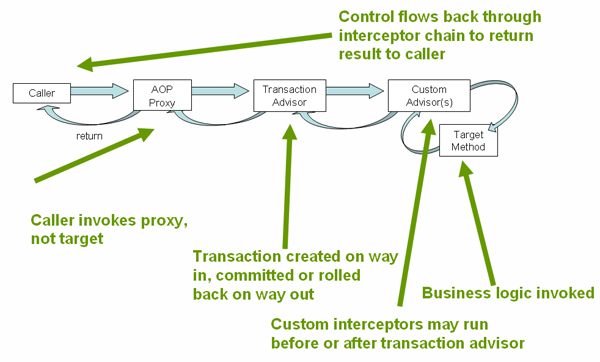
public static IFunctional iFunctional;

}

The '+' after the package is used to match the class (or interface) and all subclasses. This pointcut expression will match any interfaces or classes defined in the package com.revature.examples and also any subclass or implementing class of these classes, even if the implementing class or subclass is in another package.

* Target Object - The object which is being advised by one or more aspects. The Spring AoP is implemented using proxies, and as such, this target object will always be a proxied object.
* AOP Proxy - An object created in an AoP framework in order to implement advice defined in aspects. The utilization of these proxies allows for loose coupling between your advice targets. The Spring AoP framework supports JDK dynamic proxies (which is interface based) as well as a CGLIB proxy (which is class based).

Below is a visual representation of how the Spring Transaction proxy works:



At a high-level, when a method is invoked an AOP Proxy is informed, which informs the associated Advisor (in this case a transaction advisor). This advisor implementation can then inject advice as needed (based on configuration) before, or after invoking the target method. Note that additional custom advisors can come before or after the transaction advisor is run.

**Aspect-J**

# **Spring Module - @AspectJ support**

This page details AspectJ support in Spring and how to configure Spring to utilize AspectJ with Spring.

### References

* [Spring 4.0.x Documentation - AspectJ](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/aop.html#aop-ataspectj)
* [AspectJ Project](https://www.eclipse.org/aspectj/)
* [AspectJ API Documentation - lang package](https://www.eclipse.org/aspectj/doc/released/runtime-api/index.html)
* [AspectJ API Documentation - annotations](https://www.eclipse.org/aspectj/doc/released/aspectj5rt-api/index.html)

## @AspectJ

@AspectJ refers to a style of declaring aspects through the use of regular Java classes with annotations. @AspectJ style was introduced in the AspectJ 5 release of the AspectJ Project. Note that Spring interprets the same annotations as AspectJ 5, using a library supplied by AspjectJ, however Spring's AOP runtime is not dependent on the AspectJ compiler or weaver.

## Enabling @AspectJ

To utilize @AspectJ aspects in a Spring configuration, you need to enable support for configuring the Spring AOP based on @AspectJ aspects and autoproxying beans which are advised by those aspects. Autoproxying means that Spring will automatically generate a proxy to intercept method invocations when a bean is advised by one or more aspects to ensure that advice is executed as needed.

@AspectJ configurations can be enabled using either XML or Java-based Spring configuration. In either case, you will need to make sure that the AspectJ aspectweaver.jar is on your application's classpath (version 1.6.8 or later is required for Spring 4.0.x).

This can be done by adding the following dependency within your Maven Pom file:

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>1.6.8</version> <!-- or a later version... -->

</dependency>

For XML based configurations you must use the aop:aspectj-autoproxy element:

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

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

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

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

xmlns:aop="http://www.springframework.org/schema/aop"

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

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.3.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">

<!-- This element enables AspectJ autoproxy -->

<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

<!-- other configurations below... -->

</beans>

For Java based configurations, AspectJ is supported through the use of the @EnableAspectJAutoProxy annotation within your @Configuration annotated class:

@Configuration

@EnableAspectJAutoProxy

public class AppConfig {

}

**Stereotypes**

# **Spring Module - Stereotype Annotations**

This page details Stereotype Annotations and their importance in Spring Bean Configurations.

### References

* [Spring Stereotype Annotation Documentation](https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html#beans-stereotype-annotations)
* [Spring Stereotype API Docs](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/package-summary.html)

## Stereotype Annotations Overview

Stereotypes are a number of built in annotations which are used as markers to declare obects as a bean, define their name and clarify to a developer what the bean will be used for.

### @Component

The @Component annotation is a generic stereotype used to declare an objects as a bean, which will be nijected into other classes/beans at some point in time. Spring also provides more specific use cases for @Component beans when the class serves a specific purpose, such as the persistance of data, a service or presentation. These annotations are @Repository, @Service and @Controller respectively.

@Component("account");

public class Account {

private int id;

@Override

public class toString() {

return "Account [id=" + id +"]";

}

public Account() {

super();

}

public Account (int id) {

super();

this.id = id;

}

public int getId() {

return this.id;

}

public void setId(int id) {

this.id = id;

}

}

### @Repository

The @Repository annotation marks a class to be used as a for use with storing data within a repository or database. Specifically, the @Repository annotation provides benefits for objects that would otherwise be utilized as a Data Access Object (DAO).

@Repository("accountRepository")

public class AccountRepositoryImpl implments AccountRepository {

Connection connection = null;

PreparedStatement stmt = null;

@Override

public List<Account> getAllAccounts() {

List<Account> accounts = new ArrayList<Account>();

try {

connection = DAOUtilities.getConnection();

String sql = "SELECT \* FROM accounts";

stmt = connection.prepareStatement(sql);

ResultSet rs = stmt.executeQuery();

while (rs.next()) {

Account account = new account(rs.getInt("id"));

accounts.add(account);

}

rs.close();

} catch (SQLException e) {

e.printStackTrace();

} finally {

closeResources();

}

return accounts;

}

}

### @Service

The @Service annotation marks a class as a Service for an application

@Service("accountService");

public class AccountServiceImpl implements AccountService {

@Autowired

private AccountRepository accountRepository;

public List<Account> findAll() {

return accountRepository.getAllAcounts();

}

}

### @Controller

The @Controller annotation marks a class as a [Spring MVC Controller](https://app.revature.com/) which allow the use of handlder mapping annotations. Classes annotated with @Controller are autodeteced through classpath scanning, and when used in comination with @RequestMapping, allows for quick configurations of a web application controller. More detail on @RequestMapping as well as @ResponseBody can be found in the [MVC lecture notes](https://app.revature.com/).

@Controller("accountController")

public class accountControllerImpl {

@Autowired

private AccountService accountService;

@RequestMapping(value="/accounts/all")

public @ResponseBody List<Account> getAllAccounts() {

Return accountService.findAll();

}

}

In addition to readability, use of these stereotypes makes classes more suited for other processing tools, such as [aspects](https://app.revature.com/) or marking targets for [pointcuts](https://app.revature.com/).

**Lombok**

# **Lombok**

This page details the use of Project Lombok with Spring to reduce writing/generating boilerplate code..

### References

* [Lombok](https://projectlombok.org/)

## What is Lombok

According to the official website, "Project Lombok is a java library that automatically plugs into your editor and build tools, spicing up your java. Never write another getter or equals method again, with one annotation your class has a fully featured builder, Automate your logging variables, and much more." Project Lombok isn't unique to Spring. Lombok can be used with vanilla Java however, Lombok and Spring have a common goal; reduce the amount of boilerplate code by autogenerating the required code elements at compiletime.

## Why use Lombok

First lets define boilerplate code. Boilerplate code is code that must be included in many places with little to no alteration. Prime examples of Java boilerplate code.

* Getters and Setters
* No Arg Constructors
* toString
* Equals
* Hashcode
* Logging Variables

Next, why do we want to reduApplicationContextce writing boilerplate code? Writing boilerplate code is tedious, easily overlooked, reduces testing coverage, and muddies up code. Lombok exposes a host of annotations that makes generating this code easy while aleviating the above issues.

## Lombok usage

The following assumes the use of Lombok with Spring. It is required to used the library with Spring, but this is where we are right now. Please refer to the official documentation for learning how to install the library and required plugins for your IDE and project workflow. In the examples below I have the plugin for my IDE (IntelliJ) and I have Lombok as a project dependency in the pom.xml.

@Getter

@Setter

@NoArgsConstructor

@ToString

@EqualsAndHashCode

**public** **class** **Book** {

**private** **String** title;

**private** **String** isbn;

**private** **String**[] authors;

**private** **LocalDateTime** publicationDate;

}

The above code creates a Java model class for a book. The use of the Lombok annotations will automatically generate the require code elements during compile time. In this case since the annotations are above the class, specifically the @Getter and @Setter annotations, the generated code will cover the whole class. So a getter and setter for each field.

| **Annotation** | **Purpose** |
| --- | --- |
| @Getter | Generates a getter method for each field |
| @Setter | Generate a setter method for each field |
| @NoArgsConstructor | Generate a no argument constructor for the class |
| @ToString | Generate a toString method for the class |
| @EqualsAndHasCode | Generate and equals and hashCode method for the class |

@Getter

@Setter

@NoArgsConstructor

@ToString

@EqualsAndHashCode

**public** **class** **Book** {

**private** **String** title;

**private** **String** isbn;

**private** **String**[] authors;

**private** **LocalDateTime** publicationDate;

@Getter(**AccessLevel**.NONE)

@Setter

**private** **Double** commissionRate;

}

The above code example is similar to the previous one meaning Lombok will provide a getter and setter for each field. However the commissionRate field's getter method will not be accessible. You can pass an AccessLevel to either the @Getter and @Setter or both. This is useful when you want to create readonly or writeonly fields.

The previous examples uses a lot of the Lombok annoations and it will be easy to assume that the Lombok annotations will become boilerplate code in short order. @Data is a shortcut annotation that includes the following annotations

* @Getter
* @Setter
* @ToString
* @EqualsAndHashCode
* @RequiredArgsConstructor

### What about Autowiring with Spring

Although Spring and Lombok won't clash too much in most occassions, there is one feature of Spring that Lombok will not handle by default, and this will cause problems at runtime, Autowiring. Because the code is generated for you at compile time, you will have to tell Lombok to add the @Autowired annotation.

@Repository

@NoArgsConstructor

**public** **class** **BookRepository** {...}

@Service

@RequiredArgsConstructor(onConstructor=@\_\_(@Autowired))

**public** **class** **BookService**{

**private** **BookRepository** bookRepository

}

The above code creates the BookRepository and BookService classes. There is a dependency graph BookRepository -> BookService. The BookService requires the BookRepository, but the constructor is going to be generated for us. The onConstructor attribute takes a list of annotations to add to the constructor's signature.

@Repository

@NoArgsConstructor

**public** **class** **BookRepository** {...}

@Service

@NoArgsConstructor

**public** **class** **BookService**{

@Setter(onMethod=@\_\_(@Autowired))

**private** **BookRepository** bookRepository

}

The above example is similar to the previous example, but uses setter injection.

WEIRD NOTE!!! The syntax @\_\_ is for a backward compatibility issue with javac 7 [Read More on @\_\_](https://projectlombok.org/features/experimental/onX.html).

Day – 5

**Spring Boot Actuator**

# **Spring Boot Module - Spring Boot Actuator**

This page details Spring Boot Actuator to monitor and manage web applications

Spring Boot has really become popular as more and more companies create or convert their systems using microservice patterns.

### References

* [Spring Boot Actuator Production Features](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-features.html)
* [Spring Boot Actuator Web API](https://docs.spring.io/spring-boot/docs/current/actuator-api/html/)

## Spring Boot Actuator

Spring Boot Actuator is a Spring library that exposes tools for monitoring and gathering metrics about a running application. By using Spring Boot Actuator boilerplate operation tools like health, metrics, env, etc.. can be automatically included in an application through either HTTP or JMX (Java Management Extensions).

## Endpoints

Spring Boot Actuator comes with a lot of predefined endpoints to utilize its operational features. Below is a list of some of the more common endpoints. Check the docs for more information.

| **Endpoint** | **Use** |
| --- | --- |
| /health | Shows application health information |
| /beans | Display a list of available Beans in the BeanFactory |
| /env | Return the current application properties or a single property |
| /info | Return general information can be custom data |
| /logfile | Returns content of the logfile |
| /loggers | Show and/or modify logger configurations |
| /mappings | Returns a collated list of all @RequestMappings |
| /metrics | Show metric information for the current application |
| /shutdown | Perform an application shutdown |

### Spring Boot Actuator Usage

To include Actuator in your Spring Boot project simply add it to the pom.xml after the project creation or select it during the project creation process with Spring Boot Initializr.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

By Default all endpoints except for shutdown are enabled. To enable and endpoint simply set its management.endpoints.<id>.enabled property.

management.endpoints.shutdown.enabled=true

management.endpoints.metrics.enabled=false

Endpoints can be configured to be opt-in rather than opt-out by setting the management.endpoints.enabled-by-default to false. Then all endpoints will be disabled and you must opt-in to use each endpoint.

management.endpoints.enabled-by-default=false

management.endpoints.health.enabled=true

management.endpoints.mappings.enabled=true

IMPORTANT NOTE!!! Some of these endpoints will show sensitive information or perform dangerous operations. Actuator endpoints should be included in your security chain and/or excluded from exposure. Depending on what you're using, HTTP or JMX some endpoints are exposed by default, see the docs for more information. To include or exclude a(n) endpoint(s), simply set the include or exclude property of the messaging protocol being used.

management.endpoints.jmx.exclude=shutdown, loggers

management.endpoints.web.exclude=shutdown, loggers

management.endpoints.jmx.include=info, health

management.endpoints.web.include=info, health

Once your endpoints are enabled and exposed you can access the information by sending a request to /actuator/<endpoint-id>.

Acutator is used to handle many management and monitoring production operations. This document is a very short overview of the use. Please refer to the documentation for more details.

**Spring Boot DevTools**

# **Spring Boot Module - Spring Boot DevTools**

This page details the use of Spring Boot DevTools

### References

* [Spring Boot DevTools](https://docs.spring.io/spring-boot/docs/1.5.16.RELEASE/reference/html/using-boot-devtools.html)

## Spring Boot DevTools

Spring Boot applications can include a set of tools to ease development more. spring-boot-devtools. Spring Boot DevTools offer 2 major features that improve development workflow.

* Disabled caching
* Automatic restarts

Spring Boot DevTools are only used during development and are automatically disabled when running a fully packaged application.

## Disabled caching

Some Spring modules use caching to imporve performance. While useful during production deployments, caching can hinder development workflow. Spring Boot DevTools automatically disables caching with these modules.

## Automatic Restarts

Manual restart of an application during development can be cumbersome and frustrating. If making small, frequent changes it can be easy to overlook a restart. DevTools will automatically restart the application when a file on the classpath changes.

Spring Boot DevTools optimizes restarts by using separate ClassLoaders. The first ClassLoader is used for classes that won't change. The second ClassLoader is used for your classes. This approach reduces the number of classes that need to be loaded during startup making automatic restarts faster. There are a number of ways to trigger restarts. Refer to the docs for more information.

## DevTools Usage

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<optional>true</optional>

</dependency>

</dependencies>

Simply include spring-boot-devtools dependency in your project file. Please refer to the documentation for any special configurations that you may want to change.

It is best practice to flag the dependency as optional in Mave or compileOnly in Gradle to prevent transitive inclusion in other dependent modules.

**Using RestTemplate to consume web services**

# **Spring Boot Module - Rest Template Requests**

This page details using RestTemplate to make web requests from a Java/Spring Application. The example code assumes the use of Spring Boot, but RestTemplate is available to all Java/Spring applications.

### References

* [RestTemplate](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html)

NOTE: RestTemplate will be deprecated in Spring Framework 5.

## RestTemplate

In Spring RestTemplate acts as a web client to make requests to web services. RestTemplate offers 3 types of methods for exchanging data with web service:

| **API** | **Purpose** | **Example** |
| --- | --- | --- |
| \*ForEntity | Request for the native exchange format returned by the server (JSON, XML, Text) | getForEntity(...) |
| \*ForObject | Deserialize the response into a Java pojo | getForObject(...) |
| exchange | Generic API for making web requests | exchange(...) |

## RequestTemplate Usage

### getForEntity

**RestTemplate** rest = **new** **RestTemplate**();

**String** resourceUrl = "http://localhost:8080/myservice/data";

**ResponseEntity**<**String**> response = rest.**getForEntity**(resourceUrl + "/1", **String**.**class**);

The above code creates and instance of a RestTemplate and makes a request to the resourceUrl. The method used for the request here is one of the overloadedgetForEntity methods. Here the request is being made for the native data format being sent back by the server. The response is encapsulated in a ResponseEntity object with which we can retrived the response status, body, and headers.

### getForObject

**public** **class** **DataModel**{

...

**private** **String** value;

}

**RestTemplate** rest = **new** **RestTemplate**();

**String** resourceUrl = "http://localhost:8080/myservice/data";

**DataModel** model = rest.**getForObject**(resourceUrl + "/1", **DataModel**.**class**);

The above code is similar to the previouse example, however this time the code uses getForObject. RestTemplate will attempt to deserialize the response body using a message converter bean based on the Content-Type header in the response.

### exchange API

The exchange API offers a more generic manner of make web request through RestTemplate.

**RestTemplate** rest = **new** **RestTemplate**();

**HttpEntity**<**Data**> request = **new** **HttpEntity<>**(**new** **DataModel**("value"));

**String** resourceUrl = "http://localhost:8080/myservice/data";

**ResponseEntity**<**DataModel**> response = rest.**exchange**(resourceUrl, **HttpMethod**.POST, request, **DataModel**.**class**);

The above code uses the exchange API to perform an POST request. The major differences in the signature are the HttpMethod argument and the request argument. The HttpMethod argument configures which type of request to make, and the request argument will become the serialized body of the request.

**RestTemplate** rest = **new** **RestTemplate**();

**HttpEntity**<**Data**> request = **new** **HttpEntity<>**(**new** **DataModel**("value"));

**String** resourceUrl = "http://localhost:8080/myservice/data";

**ResponseEntity**<**DataModel**> response = rest.**postForLocation**(resourceUrl, request);

This above code is similar to the previous example, but uses postForLocation. REST spec states that a Post request can return an empty body with the Location header set to the new resource url. postForLocation will return the value of the Location header in the response.

### Form Data

**RestTemplate** rest = **new** **RestTemplate**();

**HttpHeaders** headers = **new** **HttpHeaders**();

headers.**setContentType**(**MediaType**.APPLICATION\_FORM\_URLENCODED);

**MultiValueMap**<**String**, **String**> map = **new** **LinkedMultiValueMap<>**();

map.**add**("value": "test");

**HttpEntity**<**MultiValueMap**<**String**, **String**>> request = **new** **HttpEntity<>**(map, headers);

**String** resourceUrl = "http://localhost:8080/myservice/data";

**ResponseEntity**<**DataModel**> response = rest.**postForLocation**(resourceUrl, request);