

**Component Scanning**

Spring can automatically scan a package for beans if component scanning is enabled.

Here’s the simple way to do that **with Java configuration**:

@Configuration

@ComponentScan("com.baeldung")

public class JavaConfig {

@Bean(name="helloWorld")

public HelloWorld helloWorld() {

return new HelloWorldImpl();

}

}

Using the the @ComponentScan annotation means that Spring will search for any annotated classes within the com.baeldung package and its sub packages.

When using XML configuration, the component scanning can be used just as easily:

<context:component-scan base-package="com.baeldung" />

<http://www.baeldung.com/spring-bean-annotations>

By default, Spring resolves *@Autowired* entries by type. If more than one beans of the same type are available in the container, the framework will throw a fatal exception indicating that more than one bean is available for auto wiring.

@Component

public class Car {

@Autowired

public Car(Engine engine, Transmission transmission) {

this.engine = engine;

this.transmission = transmission;

}

}

<http://www.baeldung.com/spring-interview-questions>

<http://www.baeldung.com/spring-mvc-tutorial>

<http://www.baeldung.com/javax-validation>

<http://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>

<https://docs.spring.io/spring/docs/current/spring-framework-reference/index.html>

@Required checks if a particular property has been set or not. If a field has been annotated with @Required annotation and that field is not set, you will get org.springframework.beans.factory.BeanInitializationException.

<https://howtodoinjava.com/spring/spring-core/spring-beans-autowiring-concepts/>

@PostConstruct annotation defines a method that will be called after a bean as been fully initialized. In other words it will be called after bean construction and all dependency injection. @PostConstruct is annotation form of init-method which is an attribute of bean tag.  
@PreDestroy annotation defines a method that will be called just before a bean is destroyed. This is usually useful for resource clean up. @PreDestroy is annotation form of destroy-method which is an attribute of bean tag.

**public** **class** **Foo** {

**public** **void** init() {

*// initialization logic*

}

}

**public** **class** **Bar** {

**public** **void** cleanup() {

*// destruction logic*

}

}

@Configuration

**public** **class** **AppConfig** {

@Bean(initMethod = "init")

**public** Foo foo() {

**return** **new** Foo();

}

@Bean(destroyMethod = "cleanup")

**public** Bar bar() {

**return** **new** Bar();

}

}



**Which is the best way of injecting beans and why?**

The recommended approach is to use constructor arguments for mandatory dependencies and setters for optional ones. Constructor injection allows injecting values to immutable fields and makes testing easier.

**What is the difference between BeanFactory and ApplicationContext?**

BeanFactory is an interface representing a container that provides and manages bean instances. The default implementation instantiates beans lazily when getBean() is called.

ApplicationContext is an interface representing a container holding all information, metadata, and beans in the application. It also extends the BeanFactory interface but the default implementation instantiates beans eagerly when the application starts. This behavior can be overridden for individual beans.

<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>

@Lazy Annotation in spring is used with @Configuration. The bean which has been declared with @Lazy annotation will not be initialized by spring container. @Lazy will be initialized by container only when that bean will be accessed somewhere in code. Find the example.

@Configuration

public class AppConf {

@Bean

@Lazy(value = true)

public A a(){

return new A();

}

}



@ComponentScan(basePackages = {

"x.y.z.service",

"x.y.z.controller "})

<context:component-scan base-package="x.y.z.service, x.y.z.controller" />

The following example shows the configuration ignoring all @Repository annotations and using "stub" repositories instead.

@Configuration

@ComponentScan(basePackages = "org.example",

includeFilters = @Filter(type = FilterType.REGEX, pattern = ".\*Stub.\*Repository"),

excludeFilters = @Filter(Repository.class))

**public** **class** **AppConfig** {

...

}

and the equivalent using XML

<beans>

<context:component-scan base-package="org.example">

<context:include-filter type="regex"

expression=".\*Stub.\*Repository"/>

<context:exclude-filter type="annotation"

expression="org.springframework.stereotype.Repository"/>

</context:component-scan>

</beans>

### **How to define the scope of a bean?**

To set Spring Bean’s scope, we can use @Scope annotation or “scope” attribute in XML configuration files. There are five supported scopes:

* **singleton**
* **prototype**
* **request**
* **session**
* **global-session**

**Can we have multiple Spring configuration files in one project?**

Yes, in large projects, having multiple Spring configurations is recommended to increase maintainability and modularity.

You can load multiple Java-based configuration files:

@Configuration

@Import({MainConfig.class, SchedulerConfig.class})

public class AppConfig {

Or load one XML file that will contain all other config’s:

ApplicationContext context = new ClassPathXmlApplicationContext("spring-all.xml");

And inside this XML file you’ll have:

<import resource="main.xml"/>

<import resource="scheduler.xml"/>

### **Name some of the Design Patterns used in the Spring Framework?**

* **Singleton Pattern:** Singleton-scoped beans
* **Factory Pattern:** Bean Factory classes
* **Prototype Pattern:** Prototype-scoped beans
* **Adapter Pattern:** Spring Web and Spring MVC
* **Proxy Pattern:** Spring Aspect Oriented Programming support
* **Template Method Pattern:** JdbcTemplate, HibernateTemplate, etc.
* **Front Controller:** Spring MVC DispatcherServlet
* **Data Access Object:** Spring DAO support
* **Model View Controller:**Spring MVC

We define the bean with singleton scope by using the @Scope annotation:

@Bean

@Scope("singleton")

public Person personSingleton() {

return new Person();

}

We can also use a constant instead of the String value in the following manner:

@Scope(value = ConfigurableBeanFactory.SCOPE\_SINGLETON

xml

<bean id="personSingleton" class="org.baeldung.scopes.Person" scope="singleton"/>

<https://www.concretepage.com/spring/example_depends_on_spring>

**Method Injection:**

<http://www.wideskills.com/spring/method-injection-in-spring>

<http://www.logicbig.com/tutorials/spring-framework/spring-core/using-lookup-method/>

**Spring AOP**

**Aspect:** Aspect is a term used to represent a cross-cutting concerns. In Spring AOP, aspects are implemented using regular classes or regular classes annotated with the @Aspect annotation.

**Advice:** TheImplementation code of aspect is called an advice. An advice contains the programming code for a cross-cutting concerns. Different types of advice include “around,” “before” and “after” advice.

**Join Point:** a join point is a place where advices are plugged in into a business object. In Spring AOP, a join point always represents a method execution.

**Point Cut:** A point cut verifies whether a particular method of a particular class is eligible for applying advices or no.

**@Before:** Run before the method execution

**@After:** Run after the method returned a result

**@AfterReturning:** Run after the method returned a result, intercept the returned result as well.

**@AfterThrowing:** Run after the method throws an exception

**@Around:** Run around the method execution, combine all three advices above.

@Aspect

public class LoggingAspect {

@Before("execution(\* com.mkyong.customer.bo.CustomerBo.addCustomer(..))")

public void logBefore(JoinPoint joinPoint) {

}

@After("execution(\* com.mkyong.customer.bo.CustomerBo.addCustomer(..))")

public void logAfter(JoinPoint joinPoint) {

}

@AfterReturning( pointcut = "execution(\* com.mkyong.customer.bo.CustomerBo.addCustomerReturnValue(..))",

returning= "result")

public void logAfterReturning(JoinPoint joinPoint, Object result) {

}

}

<https://www.mkyong.com/spring3/spring-aop-aspectj-annotation-example/>

You may annotate the service layer with @Transactional annotations and instruct the Spring container to find these annotations and provide transactional semantics for these annotated methods.

@Transactional(readOnly = true)

public List<Product> findAllProducts() {

return this.productDao.findAllProducts();

}

ModelMap is a Map implementation, which saves you from old request.getAttribute/ request.setAttribute. It provides a way to set/get attributes from/to request or session.

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

    public String sayHelloAgain(ModelMap model) {

        model.addAttribute("greeting", "Hello World Again, from Spring 4 MVC");

        return "welcome";

}

@Configuration

@EnableWebMvc

@ComponentScan(basePackages = "com.websystique.springmvc")

public class HelloWorldConfiguration {

    @Bean

    public ViewResolver viewResolver() {

        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();

        viewResolver.setViewClass(JstlView.class);

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

        viewResolver.setSuffix(".jsp");

        return viewResolver;

    }

}

@EnableWebMvc is equivalent to mvc:annotation-driven in XML.

1. The @RequestParam is used to extract query parameters while @PathVariable is used to extract data right from the URI.

URL: http://localhost:8080/eportal/trades?tradeId=2001  
@RequestMapping("/trades")  
public String showTradeDetails(@RequestParam String tradeId,  
                               Model model){  
 model.addAttribute("tradeId", tradeId);

return "tradeDetails";

}

1. @RequestParam is more useful on a traditional web application where data is mostly passed in the query abatements while @PathVariable is more suitable for RESTful web services where URL contains values e.g. http://localhost:8080/book/9783827319333, here data, which is ISBN number is part of URI.

@RequestMapping(value="/book/{ISBN}", method= RequestMethod.GET)  
public String showBookDetails(@PathVariable String ISBN,   
                              Model model){  
 model.addAttribute("ISBN",ISBN);  
 return "bookDetails";

}

**Validation Annotations:**

public class User {

    @NotNull(message = "Name cannot be null")

    private String name;

    @AssertTrue

    private boolean working;

    @Size(min = 10, max = 200, message = "About Me must be between 10 and 200 characters")

    private String aboutMe;

    @Min(value = 18, message = "Age should not be less than 18")

    @Max(value = 150, message = "Age should not be greater than 150")

    private int age;

    @Email(message = "Email should be valid")

    private String email;

    // standard setters and getters

}

<http://www.baeldung.com/javax-validation>

<http://www.baeldung.com/spring-requestmapping>

@ModelAttribute is used to access the command class in controller class.

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

public String submit(@ModelAttribute("employee") Employee employee) {

    // Code that uses the employee object

    return "employeeView";

}

**Exception Handling:**

@ExceptionHandler(CustomGenericException.class)

public ModelAndView handleCustomException(CustomGenericException ex) {

ModelAndView model = new ModelAndView("error/generic\_error");

model.addObject("errCode", ex.getErrCode());

model.addObject("errMsg", ex.getErrMsg());

return model;

}

@ExceptionHandler(Exception.class)

public ModelAndView handleAllException(Exception ex) {

ModelAndView model = new ModelAndView("error/generic\_error");

model.addObject("errMsg", "this is Exception.class");

return model;

}

@ControllerAdvice

public class GlobalExceptionHandler {

private static final Logger logger = LoggerFactory.getLogger(GlobalExceptionHandler.class);

@ExceptionHandler(SQLException.class)

public ModelAndView handleSQLException(HttpServletRequest request, Exception ex){

logger.info("SQLException Occured:: URL="+request.getRequestURL());

return new ModelAndView();

}

@ResponseStatus(value=HttpStatus.NOT\_FOUND, reason="IOException occured")

@ExceptionHandler(IOException.class)

public ModelAndView handleIOException(){

logger.error("IOException handler executed");

//returning 404 error code

return new ModelAndView();

}

}

**Does Spring singleton beans are thread-safe?**

No, Spring singleton beans are not thread-safe. Singleton doesn't mean bean would be[thread-safe](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html).

**stateless beans:** beans that are singleton and are initialized only once. The only state they have is a shared state. These beans are created while the ApplicationContext is being initialized. The SAME bean instance will be returned/injected during the lifetime of this ApplicationContext. .

**stateful beans:** beans that can carry state (instance variables). These are created every time an object is required.

As a rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans.