

### **Spring MVC Annotations**

## **@RequestBody Annotation**

**@RequestBody a**nnotation indicating a method parameter should be bound to the body of the web request. The body of the request is passed through an HttpMessageConverter to resolve the method argument depending on the content type of the request. Optionally, automatic validation can be applied by annotating the argument with @Valid.

For example, the employee JSON object is converted into Java employee object using **@RequestBody** annotation.

@RestController@RequestMapping("/api/v1")public class EmployeeController { @Autowired private EmployeeRepository employeeRepository; @PostMapping("/employees") public Employee createEmployee(@Valid @RequestBody Employee employee) { return employeeRepository.save(employee); }

*Read more at* [**Spring @RequestBody and @ResponseBody Annotations**](http://www.javaguides.net/2018/11/spring-requestbody-and-responsebody-annotations.html)

## **@RequestMapping**

**@RequestMapping** annotation for mapping web requests onto methods in request-handling classes with flexible method signatures.

Both Spring MVC and Spring WebFlux support this annotation through a RequestMappingHandlerMapping and RequestMappingHandlerAdapter in their respective modules and package structure.

**@RequestMapping** marks request handler methods inside [**@Controller**](http://www.javaguides.net/2018/11/the-spring-controller-and-restcontroller-annotations-with-examples.html) **classes; it can be configured using:**

* path, or its aliases, name, and value: which URL the method is mapped to
* method: compatible HTTP methods
* params: filters requests based on the presence, absence, or value of HTTP parameters
* headers: filters requests based on the presence, absence, or value of HTTP headers
* consumes: which media types the method can consume in the HTTP request body
* produces: which media types the method can produce in the HTTP response body Here’s a quick example of what that looks like:

@Controllerclass EmployeeController { @RequestMapping(value = "/employees/home", method = RequestMethod.GET) String home() { return "home"; }}

We can provide default settings for all handler methods in a [**@Controller**](http://www.javaguides.net/2018/11/the-spring-controller-and-restcontroller-annotations-with-examples.html) **class if we apply this annotation to the class level. The only exception is the URL which Spring won’t override with method level settings but appends the two path parts.**

For example, the following configuration has the same effect as the one above:

@Controller@RequestMapping(value = "/employees", method = RequestMethod.GET)class EmployeeController { @RequestMapping("/home") String home() { return "home"; }}

## **@GetMapping**

**@GetMapping** annotation for mapping HTTP GET requests onto specific handler methods.

Specifically, **@GetMapping** is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET).

Example:

@GetMapping("/employees")public List<Employee> getAllEmployees() { return employeeRepository.findAll();}@GetMapping("/employees/{id}")public ResponseEntity<Employee> getEmployeeById(@PathVariable(value = "id") Long employeeId) throws ResourceNotFoundException { Employee employee = employeeRepository.findById(employeeId) .orElseThrow(() -> new ResourceNotFoundException("Employee not found for this id :: " + employeeId)); return ResponseEntity.ok().body(employee);}

## **@PostMapping**

**@PostMapping a**nnotation for mapping HTTP POST requests onto specific handler methods.

Specifically, **@PostMapping** is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.POST).

Example:

@PostMapping("/employees")public Employee createEmployee(@Valid @RequestBody Employee employee) { return employeeRepository.save(employee);}

## **@PutMapping**

**@PutMapping** annotation for mapping HTTP PUT requests onto specific handler methods.

Specifically, **@PutMapping** is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.PUT).

Example:

@PutMapping("/employees/{id}")public ResponseEntity<Employee> updateEmployee(@PathVariable(value = "id") Long employeeId, @Valid @RequestBody Employee employeeDetails) throws ResourceNotFoundException { Employee employee = employeeRepository.findById(employeeId) .orElseThrow(() -> new ResourceNotFoundException("Employee not found for this id :: " + employeeId)); employee.setEmailId(employeeDetails.getEmailId()); employee.setLastName(employeeDetails.getLastName()); employee.setFirstName(employeeDetails.getFirstName()); final Employee updatedEmployee = employeeRepository.save(employee); return ResponseEntity.ok(updatedEmployee);}

## **@DeleteMapping**

**@DeleteMapping** annotation for mapping HTTP DELETE requests onto specific handler methods.

Specifically, **@DeleteMapping** is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.DELETE).

Example:

@DeleteMapping("/employees/{id}")public Map<String, Boolean> deleteEmployee(@PathVariable(value = "id") Long employeeId) throws ResourceNotFoundException { Employee employee = employeeRepository.findById(employeeId) .orElseThrow(() -> new ResourceNotFoundException("Employee not found for this id :: " + employeeId)); employeeRepository.delete(employee); Map<String, Boolean> response = new HashMap<>(); response.put("deleted", Boolean.TRUE); return response;}

## **@PatchMapping**

**@PatchMapping** annotation for mapping HTTP PATCH requests onto specific handler methods.

Specifically, **@PatchMapping** is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.PATCH).

Example:

@PatchMapping("/patch")public @ResponseBody ResponseEntity<String> patch() { return new ResponseEntity<String>("PATCH Response", HttpStatus.OK);}

## **@ControllerAdvice**

**@ControllerAdvice** annotation is a specialization of @Component. The classes annotated with **@ControllerAdvice**are auto-detected by classpath scanning.

The use of **@ControllerAdvice** is advising all or selected controllers for @ExceptionHandler, @InitBinder, and @ModelAttribute. What we have to do is create a class annotated with @ControllerAdvice and create a required method which will be annotated with @ExceptionHandler for global exception handling, @InitBinder for global init binding and @ModelAttribute for global model attributes addition. Whenever a request comes to a controller and its method with @RequestMapping and if there is no locally defined **@ExceptionHandler**, @InitBinder and @ModelAttribute, the globally defined class annotated with @ControllerAdvice is served.

Here’s a quick example of what that looks like:

@ControllerAdvice(basePackages = {"com.javaguides.springmvc.controller"} )public class GlobalControllerAdvice { @InitBinder public void dataBinding(WebDataBinder binder) { SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy"); dateFormat.setLenient(false); binder.registerCustomEditor(Date.class, "dob", new CustomDateEditor(dateFormat, true)); } @ModelAttribute public void globalAttributes(Model model) { model.addAttribute("msg", "Welcome to My World!"); } @ExceptionHandler(FileNotFoundException.class) public ModelAndView myError(Exception exception) { ModelAndView mav = new ModelAndView(); mav.addObject("exception", exception); mav.setViewName("error"); return mav; }}

*Read complete example at* [**Spring MVC Exception Handling**](http://www.javaguides.net/2018/09/spring-boot-2-exception-handling-for-rest-apis.html)

## **@ResponseBody Annotation**

When you use the **@ResponseBody** annotation on a method, Spring converts the return value and writes it to the HTTP response automatically. Each method in the Controller class must be annotated with **@ResponseBody**.

The **@ResponseBody** annotation tells a controller that the object returned is automatically serialized into JSON and passed back into the HttpResponse object.

For example,

@ResponseBody@RequestMapping("/hello")String hello() { return "Hello World!";}

Spring 4.0 introduced [**@RestController**](http://www.javaguides.net/2018/11/the-spring-controller-and-restcontroller-annotations-with-examples.html)**, a specialized version of the controller which is a convenience annotation that does nothing more than adding the** [**@Controller**](http://www.javaguides.net/2018/11/the-spring-controller-and-restcontroller-annotations-with-examples.html) **and** [**@ResponseBody**](http://www.javaguides.net/2018/11/spring-requestbody-and-responsebody-annotations.html) **annotations.**

## **@ExceptionHandler**

**@ExceptionHandler** annotation for handling exceptions in specific handler classes and/or handler methods.

Handler methods which are annotated with this annotation are allowed to have very flexible signatures.

Spring calls this method when a request handler method throws any of the specified exceptions. The caught exception can be passed to the method as an argument:

@ExceptionHandler(ResourceNotFoundException.class)public ResponseEntity<?> resourceNotFoundException(ResourceNotFoundException ex, WebRequest request) { ErrorDetails errorDetails = new ErrorDetails(new Date(), ex.getMessage(), request.getDescription(false)); return new ResponseEntity<>(errorDetails, HttpStatus.NOT\_FOUND);}

## **@ResponseStatus**

We can specify the desired HTTP status of the response if we annotate a request handler method with this annotation. We can declare the status code with the code argument, or its alias, the value argument.

Also, we can provide a reason using the reason argument.

We also can use it along with **@ExceptionHandler**:

@ResponseStatus(HttpStatus.BAD\_REQUEST)@ExceptionHandler(ResourceNotFoundException.class)public ResponseEntity<?> resourceNotFoundException(ResourceNotFoundException ex, WebRequest request) { ErrorDetails errorDetails = new ErrorDetails(new Date(), ex.getMessage(), request.getDescription(false)); return new ResponseEntity<>(errorDetails, HttpStatus.NOT\_FOUND);}

## **@PathVariable**

This annotation indicates that a method argument is bound to a URI template variable. We can specify the URI template with the @RequestMapping annotation and bind a method argument to one of the template parts with @PathVariable.

We can achieve this with the name or its alias, the value argument:

@RequestMapping("/{id}")public User getUser(@PathVariable("id") long id) { // ...}

If the name of the part in the template matches the name of the method argument, we don’t have to specify it in the annotation:

@RequestMapping("/{id}")public User getUser(@PathVariable long id) { // ...}

Moreover, we can mark a path variable optional by setting the argument required to false:

@RequestMapping("/{id}")public User getUser(@PathVariable(required = false) long id) { // ...}

## **@RequestParam**

**@RequestParam** annotation which indicates that a method parameter should be bound to a web request parameter. We use **@RequestParam** for accessing HTTP request parameters:

@RequestMappingVehicle getVehicleByParam(@RequestParam("id") long id) { // ...}

It has the same configuration options as the **@PathVariable** annotation.

In addition to those settings, with **@RequestParam** we can specify an injected value when Spring finds no or empty value in the request. To achieve this, we have to set the default value argument.

Providing a default value implicitly sets required to false:

@RequestMapping("/buy")Car buyCar(@RequestParam(defaultValue = "5") int seatCount) { // ...}

Read more at [**https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestParam.html**](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestParam.html)

## @Controller

This annotation is simply a specialization of the @Component class and allows implementation classes to be autodetected through the classpath scanning.

We can define a Spring MVC controller with @Controller. Read more at [**The Spring @Controller and @RestController Annotations with Examples**](http://www.javaguides.net/2018/11/the-spring-controller-and-restcontroller-annotations-with-examples.html)

For example,

@Controller@RequestMapping("/api/v1")public class EmployeeController { @Autowired private EmployeeRepository employeeRepository; @GetMapping("/employees") public List<Employee> getAllEmployees() { return employeeRepository.findAll(); }}

## @RestController

Spring 4.0 introduced [**@RestController**](http://www.javaguides.net/2018/11/the-spring-controller-and-restcontroller-annotations-with-examples.html)**, a specialized version of the controller which is a convenience annotation that does nothing more than adding the** [**@Controller**](http://www.javaguides.net/2018/11/the-spring-controller-and-restcontroller-annotations-with-examples.html) **and** [**@ResponseBody**](http://www.javaguides.net/2018/11/spring-requestbody-and-responsebody-annotations.html) **annotations. By annotating the controller class with @RestController annotation, you no longer need to add** [**@ResponseBody**](http://www.javaguides.net/2018/11/spring-requestbody-and-responsebody-annotations.html) **to all the request mapping methods. The @ResponseBody annotation is active by default.**

To use [**@RestController**](http://www.javaguides.net/2018/11/the-spring-controller-and-restcontroller-annotations-with-examples.html) **in our example, all we need to do is modify the** [**@Controller**](http://www.javaguides.net/2018/11/the-spring-controller-and-restcontroller-annotations-with-examples.html) **to** [**@RestController**](http://www.javaguides.net/2018/11/the-spring-controller-and-restcontroller-annotations-with-examples.html) **and remove the** [**@ResponseBody**](http://www.javaguides.net/2018/11/spring-requestbody-and-responsebody-annotations.html) **from each method. The resultant class should look like the following:**

@RestController@RequestMapping("/api/v1")public class EmployeeController { @Autowired private EmployeeRepository employeeRepository; @GetMapping("/employees") public List<Employee> getAllEmployees() { return employeeRepository.findAll(); }}

*Read more at* [**The Spring @Controller and @RestController Annotations with Examples**](http://www.javaguides.net/2018/11/the-spring-controller-and-restcontroller-annotations-with-examples.html)

## **@ModelAttribute**

With this annotation we can access elements that are already in the model of an MVC **@Controller, by providing the model key:**

@PostMapping("/users")void saveUser(@ModelAttribute("user") User user) { // ...}

Like with @PathVariable and @RequestParam, we don’t have to specify the model key if the argument has the same name:

@PostMapping("/users")void saveUser(@ModelAttribute User user) { // ...}

Besides, @ModelAttribute has another use: if we annotate a method with it, Spring will automatically add the method’s return value to the model:

@ModelAttribute("vehicle")User getUser() { // ...}

Like before, we don’t have to specify the model key, Spring uses the method’s name by default:

@ModelAttributeUser user() { // ...}

Before Spring calls a request handler method, it invokes all @ModelAttribute annotated methods in the class.

## **@CrossOrigin**

**@CrossOrigin** enables cross-domain communication for the annotated request handler methods:

@CrossOrigin@RequestMapping("/hello")String hello() { return "Hello World!";}

If we mark a class with it, it applies to all request handler methods in it.

We can fine-tune CORS behavior with this annotation’s arguments.

## **@InitBinder**

**@InitBinder** annotation that identifies methods which initialize the WebDataBinder which will be used for populating command and form object arguments of annotated handler methods.

Such init-binder methods support all arguments that RequestMapping supports, except for command/form objects and corresponding validation result objects. Init-binder methods must not have a return value; they are usually declared as void.

For example,

// add an initbinder ... to convert trim input strings // remove leading and trailing whitespace // resolve issue for our validation @InitBinder public void initBinder(WebDataBinder dataBinder) { StringTrimmerEditor stringTrimmerEditor = new StringTrimmerEditor(true); dataBinder.registerCustomEditor(String.class, stringTrimmerEditor); }

### **Spring Core Annotations**

## **@Autowired**

We can use the @Autowired to mark a dependency which Spring is going to resolve and inject. We can use this annotation with a constructor, setter, or field injection.

Constructor Injection:

@RestControllerpublic class CustomerController { private CustomerService customerService; @Autowired public CustomerController(CustomerService customerService) { this.customerService = customerService; }}

Setter Injection:

import org.springframework.beans.factory.annotation.Autowired;import org.springframework.web.bind.annotation.RestController;@RestControllerpublic class CustomerController { private CustomerService customerService; @Autowired public void setCustomerService(CustomerService customerService) { this.customerService = customerService; }}

Field Injection:

import org.springframework.beans.factory.annotation.Autowired;import org.springframework.web.bind.annotation.RestController;@RestControllerpublic class CustomerController { @Autowired private CustomerService customerService;}

*For more details visit our articles about* [@Autowired](http://www.javaguides.net/2018/09/spring-autowired-annotation-with-example.html) and [Guide to Dependency Injection in Spring](http://www.javaguides.net/2018/06/guide-to-dependency-injection-in-spring.html).

## **@Bean**

* @Bean is a method-level annotation and a direct analog of the XML element. The annotation supports some of the attributes offered by, such as init-method, destroy-method, autowiring and name.
* You can use the [@Bean](http://www.javaguides.net/2018/09/spring-bean-annotation-with-example.html) annotation in a [@Configuration-annotated](http://www.javaguides.net/2018/09/spring-configuration-annotation-with-example.html) or in a @Component-annotated class.

The following is a simple example of a [@Bean](http://www.javaguides.net/2018/09/spring-bean-annotation-with-example.html) method declaration:

import org.springframework.context.annotation.Bean;import org.springframework.context.annotation.Configuration;import com.companyname.projectname.customer.CustomerService;import com.companyname.projectname.order.OrderService;@Configurationpublic class Application { @Bean public CustomerService customerService() { return new CustomerService(); } @Bean public OrderService orderService() { return new OrderService(); }}

The preceding configuration is exactly equivalent to the following Spring XML:

<beans> <bean id="customerService" class="com.companyname.projectname.CustomerService"/> <bean id="orderService" class="com.companyname.projectname.OrderService"/></beans>

*Read more about @Bean annotation on* [Spring @Bean Annotation with Example](http://www.javaguides.net/2018/09/spring-bean-annotation-with-example.html).

## **@Qualifier**

This annotation helps fine-tune annotation-based autowiring. There may be scenarios when we create more than one bean of the same type and want to wire only one of them with a property. This can be controlled using @Qualifier annotation along with the @Autowired annotation.

Example: Consider EmailService and SMSService classes implements single MessageService interface.

Create MessageService interface for multiple message service implementations.

public interface MessageService { public void sendMsg(String message);}

Create implementations - EmailService and SMSService.

public class EmailService implements MessageService{ public void sendMsg(String message) { System.out.println(message); }}

public class SMSService implements MessageService{ public void sendMsg(String message) { System.out.println(message); }}

It's time to see the usage of @Qualifier annotation.

public interface MessageProcessor { public void processMsg(String message);}public class MessageProcessorImpl implements MessageProcessor { private MessageService messageService; // setter based DI @Autowired @Qualifier("emailService") public void setMessageService(MessageService messageService) { this.messageService = messageService; } // constructor based DI @Autowired public MessageProcessorImpl(@Qualifier("emailService") MessageService messageService) { this.messageService = messageService; } public void processMsg(String message) { messageService.sendMsg(message); }}

*Read more about this annotation on* [Spring @Qualifier Annotation Example](http://www.javaguides.net/2018/06/spring-qualifier-annotation-example.html).

## **@Required**

The @Required annotation is method-level annotation and applied to the setter method of a bean.

This annotation simply indicates that the setter method must be configured to be dependency-injected with a value at configuration time.

For example, @Required on setter methods to mark dependencies that we want to populate through XML:

@Requiredvoid setColor(String color) { this.color = color;}

<bean class="com.javaguides.spring.Car"> <property name="color" value="green" /></bean>

Otherwise, BeanInitializationException will be thrown.

## **@Value**

Spring @Value annotation is used to assign default values to variables and method arguments. We can read spring environment variables as well as system variables using @Value annotation.

Spring @Value annotation also supports SpEL. Let’s look at some of the examples of using @Value annotation.

**Examples:** We can assign a default value to a class property using @Value annotation.

@Value("Default DBConfiguration")private String defaultName;

@Value annotation argument can be a string only, but spring tries to convert it to the specified type. Below code will work fine and assign the boolean and integer values to the variable.

@Value("true")private boolean defaultBoolean;@Value("10")private int defaultInt;

Spring @Value – Spring Environment Property

@Value("${APP\_NAME\_NOT\_FOUND}")private String defaultAppName;

Assign system variables using @Value annotation.

@Value("${java.home}")private String javaHome; @Value("${HOME}")private String homeDir;

Spring @Value – SpEL

@Value("#{systemProperties['java.home']}")private String javaHome;

## **@DependsOn**

The @DependsOn annotation can force Spring IoC container to initialize one or more beans before the bean which is annotated by @DependsOn annotation.

The @DependsOn annotation may be used on any class directly or indirectly annotated with @Component or on methods annotated with [@Bean](http://www.javaguides.net/2018/09/spring-bean-annotation-with-example.html).

Example: Let's create FirstBean and SecondBean classes. In this example, the SecondBean is initialized before bean FirstBean.

public class FirstBean { @Autowired private SecondBean secondBean;}public class SecondBean { public SecondBean() { System.out.println("SecondBean Initialized via Constuctor"); }}

Declare the above beans in java based configuration class.

@Configurationpublic class AppConfig { @Bean("firstBean") @DependsOn(value = { "secondBean" }) public FirstBean firstBean() { return new FirstBean(); } @Bean("secondBean") public SecondBean secondBean() { return new SecondBean(); }}

*Read more about @DependsOn annotation on* [Spring - @DependsOn Annotation Example](http://www.javaguides.net/2018/10/spring-dependson-annotation-example.html).

## **@Lazy**

By default, the [Spring IoC container](http://www.javaguides.net/2018/10/spring-ioc-container-overview.html) creates and initializes all singleton beans at time of application startup. We can prevent this pre-initialization of a singleton bean by using the @Lazy annotation.

The @Lazy annotation may be used on any class directly or indirectly annotated with @Component or on methods annotated with [@Bean](http://www.javaguides.net/2018/09/spring-bean-annotation-with-example.html).

Example: Consider we have below two beans - FirstBean and SecondBean. In this example, we will explicitly load FirstBean using @Lazy annotation.

public class FirstBean { public void test() { System.out.println("Method of FirstBean Class"); }}

public class SecondBean { public void test() { System.out.println("Method of SecondBean Class"); }}

Declare the above beans in java based configuration class.

@Configurationpublic class AppConfig { @Lazy(value = true) @Bean public FirstBean firstBean() { return new FirstBean(); } @Bean public SecondBean secondBean() { return new SecondBean(); }}

As we can see, bean **secondBean** is initialized by Spring container while bean **firstBean** is initialized explicitly.

*Read more about @Lazy annotation with a complete example on* [Spring - @Lazy Annotation Example](http://www.javaguides.net/2018/10/spring-lazy-annotation-example.html).

## **@Lookup**

A method annotated with @Lookup tells Spring to return an instance of the method’s return type when we invoke it.

*Detailed information about the annotation can be found in* [Spring @LookUp Annotation](https://www.baeldung.com/spring-lookup).

## **@Primary**

We use @Primary to give higher preference to a bean when there are multiple beans of the same type.

@Component@Primaryclass Car implements Vehicle {} @Componentclass Bike implements Vehicle {} @Componentclass Driver { @Autowired Vehicle vehicle;} @Componentclass Biker { @Autowired @Qualifier("bike") Vehicle vehicle;}

*Read more about this annotation on* [Spring - @Primary Annotation Example](http://www.javaguides.net/2018/10/spring-primary-annotation-example.html).

## **@Scope**

We use @Scope to define the scope of a @Component class or a [@Bean](http://www.javaguides.net/2018/09/spring-bean-annotation-with-example.html) definition. It can be either singleton, prototype, request, session, globalSession or some custom scope.

For example:

@Component@Scope(value = ConfigurableBeanFactory.SCOPE\_SINGLETON)public class TwitterMessageService implements MessageService {}@Component@Scope(value = ConfigurableBeanFactory.SCOPE\_PROTOTYPE)public class TwitterMessageService implements MessageService {}

*Read more about @Scope annotations on* [Spring @Scope annotation with Singleton Scope Example](http://www.javaguides.net/2018/10/spring-scope-annotation-with-singleton-scope-example.html) and [Spring @Scope annotation with Prototype Scope Example](http://www.javaguides.net/2018/10/spring-scope-annotation-with-prototype.html).

## **@Profile**

If we want Spring to use a @Component class or a [@Bean](http://www.javaguides.net/2018/09/spring-bean-annotation-with-example.html) method only when a specific profile is active, we can mark it with @Profile. We can configure the name of the profile with the value argument of the annotation:

@Component@Profile("sportDay")class Bike implements Vehicle {}

*You can read more about profiles in this* [Spring Profiles](https://www.baeldung.com/spring-profiles).

## **@Import**

The @Import annotation indicates one or more [@Configuration](http://www.javaguides.net/2018/09/spring-configuration-annotation-with-example.html) classes to import.

For example: In a Java-based configuration, Spring provides the @Import annotation which allows for loading [@Bean](http://www.javaguides.net/2018/09/spring-bean-annotation-with-example.html) definitions from another configuration class.

@Configurationpublic class ConfigA { @Bean public A a() { return new A(); }}@Configuration@Import(ConfigA.class)public class ConfigB { @Bean public B b() { return new B(); }}

Now, rather than needing to specify both **ConfigA** class and **ConfigB** class when instantiating the context, only **ConfigB** needs to be supplied explicitly.

*Read more about @Import annotation on* [Spring @Import Annotation](http://www.javaguides.net/2018/09/spring-import-annotation-with-example.html).

## **@ImportResource**

Spring provides a @ImportResource annotation is used to load beans from an applicationContext.xml file into an **ApplicationContext**. For example: Consider we have applicationContext.xml spring bean configuration XML file on the classpath.

@Configuration@ImportResource({"classpath\*:applicationContext.xml"})public class XmlConfiguration {}

*Read more about this annotation with a complete example on* [Spring @ImportResource Annotation](http://www.javaguides.net/2018/09/spring-importresource-annotation-example.html).

## **@PropertySource**

The @PropertySource annotation provides a convenient and declarative mechanism for adding a PropertySource to Spring’s Environment. To be used in conjunction with [@Configuration](http://www.javaguides.net/2018/09/spring-configuration-annotation-with-example.html) classes.

For example: In this example, we are reading database configuration from file config.propertiesfile and set these property values to **DataSourceConfig** class using Environment.

import org.springframework.beans.factory.InitializingBean;import org.springframework.beans.factory.annotation.Autowired;import org.springframework.context.annotation.Configuration;import org.springframework.context.annotation.PropertySource;import org.springframework.core.env.Environment;@Configuration@PropertySource("classpath:config.properties")public class ProperySourceDemo implements InitializingBean { @Autowired Environment env; @Override public void afterPropertiesSet() throws Exception { setDatabaseConfig(); } private void setDatabaseConfig() { DataSourceConfig config = new DataSourceConfig(); config.setDriver(env.getProperty("jdbc.driver")); config.setUrl(env.getProperty("jdbc.url")); config.setUsername(env.getProperty("jdbc.username")); config.setPassword(env.getProperty("jdbc.password")); System.out.println(config.toString()); }}

*Read more about this annotation on* [Spring @PropertySource Annotation with Example](http://www.javaguides.net/2018/09/spring-propertysource-annotation-with-example.html).

## **@PropertySources**

We can use this annotation to specify multiple @PropertySource configurations:

@PropertySources({ @PropertySource("classpath:config.properties"), @PropertySource("classpath:db.properties") }) public class AppConfig { //... }

In Spring, you can use @PropertySource annotation to externalize your configuration to a properties file. In this article, we will discuss how to use @PropertySource to read a properties file and display the values with @Valueand Environment.

The @PropertySource annotation provides a convenient and declarative mechanism for adding a PropertySourceto Spring’s Environment. To be used in conjunction with [@Configuration](http://www.javaguides.net/2018/09/spring-configuration-annotation-with-example.html) classes.

## Spring @PropertySource Annotation with Simple Example

In this example, we are reading database configuration from file config.properties file and set these property values to DataSourceConfig class using Environment.

import org.springframework.beans.factory.InitializingBean;import org.springframework.beans.factory.annotation.Autowired;import org.springframework.context.annotation.Configuration;import org.springframework.context.annotation.PropertySource;import org.springframework.core.env.Environment;@Configuration@PropertySource("classpath:config.properties")public class ProperySourceDemo implements InitializingBean { @Autowired Environment env; @Override public void afterPropertiesSet() throws Exception { setDatabaseConfig(); } private void setDatabaseConfig() { DataSourceConfig config = new DataSourceConfig(); config.setDriver(env.getProperty("jdbc.driver")); config.setUrl(env.getProperty("jdbc.url")); config.setUsername(env.getProperty("jdbc.username")); config.setPassword(env.getProperty("jdbc.password")); System.out.println(config.toString()); }}

## Spring @PropertySource Annotation Placeholders Example

Any ${…} placeholders present in a @PropertySource resource location will be resolved against the set of property sources already registered against the environment.

For example:

import org.springframework.beans.factory.InitializingBean;import org.springframework.beans.factory.annotation.Autowired;import org.springframework.context.annotation.Configuration;import org.springframework.context.annotation.PropertySource;import org.springframework.core.env.Environment;@Configuration@PropertySource("classpath:/com/${my.placeholder:default/path}/config.properties")public class ProperySourceDemo implements InitializingBean { @Autowired Environment env; @Override public void afterPropertiesSet() throws Exception { setDatabaseConfig(); } private void setDatabaseConfig() { DataSourceConfig config = new DataSourceConfig(); config.setDriver(env.getProperty("jdbc.driver")); config.setUrl(env.getProperty("jdbc.url")); config.setUsername(env.getProperty("jdbc.username")); config.setPassword(env.getProperty("jdbc.password")); System.out.println(config.toString()); }}

Assuming that "my.placeholder" is present in one of the property sources already registered, e.g. system properties or environment variables, the placeholder will be resolved to the corresponding value. If not, then "default/path" will be used as a default. If no default is specified and a property cannot be resolved, an IllegalArgumentException will be thrown

## @PropertySources Annotation - Include multiple properties files

Introduces new @PropertySources to support Java 8 and a better way to include multiple properties files.

@Configuration @PropertySources({ @PropertySource("classpath:config.properties"), @PropertySource("classpath:db.properties") }) public class AppConfig { //... }

Allow @PropertySource to ignore the not found properties file.

@Configuration @PropertySource("classpath:missing.properties") public class AppConfig { //... }

If missing.properties is not found, the system is unable to start and throws FileNotFoundException

Caused by: java.io.FileNotFoundException: classpath resource [missiong.properties] cannot be opened because it does not exist

In Spring 4, you can use ignoreResourceNotFound to ignore the not found properties file

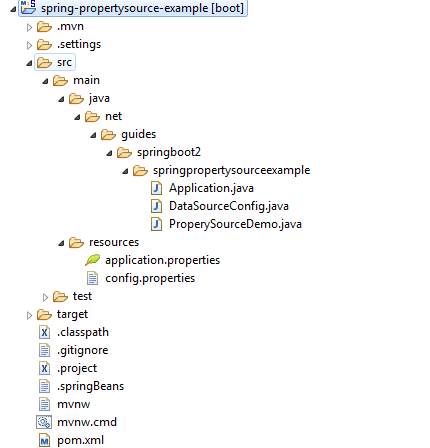
@Configuration @PropertySource(value="classpath:missing.properties", ignoreResourceNotFound=true) public class AppConfig { //... } @PropertySources({ @PropertySource(value = "classpath:missing.properties", ignoreResourceNotFound=true), @PropertySource("classpath:config.properties") })

## Spring @PropertySource Annotation Complete Example

Let's create a simple Spring boot maven project to bootstrap quickly.

In this example, we are reading database configuration from file config.properties file and set these property values to DataSourceConfig class.

Create a maven project using Spring Initializr at <http://start.spring.io/>, which is an online Spring Boot application generator.



Create a packing structure as per above diagram.

## The pom.xml File

<?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 <http://maven.apache.org/xsd/maven-4.0.0.xsd>"> <modelVersion>4.0.0</modelVersion> <groupId>net.guides.springboot2</groupId> <artifactId>spring-propertysource-example</artifactId> <version>0.0.1-SNAPSHOT</version> <packaging>jar</packaging> <name>spring-propertysource-example</name> <description>Demo project for Spring Boot</description> <parent> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-parent</artifactId> <version>2.0.5.RELEASE</version> <relativePath/> <!-- lookup parent from repository --> </parent> <properties> <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding> <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding> <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> </dependency> </dependencies> <build> <plugins> <plugin> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-maven-plugin</artifactId> </plugin> </plugins> </build></project>

## Create a config.properties File

Let's create a config.properties file in classpath and we will use @PropertySource annotation to read a properties file and display the values with @Value and Environment.

jdbc.driver=com.mysql.jdbc.Driverjdbc.url=jdbc:mysql://localhost:3306/dev\_dbjdbc.username=rootjdbc.password=root

## Create DataSourceConfig.java File

package net.guides.springboot2.springpropertysourceexample;public class DataSourceConfig { private String driver; private String url; private String username; private String password; @Override public String toString() { return "DataSourceConfig [driver=" + driver + ", url=" + url + ", username=" + username + "]"; } public String getDriver() { return driver; } public void setDriver(String driver) { this.driver = driver; } public String getUrl() { return url; } public void setUrl(String url) { this.url = url; } public String getUsername() { return username; } public void setUsername(String username) { this.username = username; } public String getPassword() { return password; } public void setPassword(String password) { this.password = password; }}

## Create ProperySourceDemo.java File

package net.guides.springboot2.springpropertysourceexample;import org.slf4j.Logger;import org.slf4j.LoggerFactory;import org.springframework.beans.factory.InitializingBean;import org.springframework.beans.factory.annotation.Autowired;import org.springframework.beans.factory.annotation.Value;import org.springframework.context.annotation.Configuration;import org.springframework.context.annotation.PropertySource;import org.springframework.core.env.Environment;@Configuration@PropertySource("classpath:config.properties")public class ProperySourceDemo implements InitializingBean { private static final Logger LOGGER = LoggerFactory.getLogger(ProperySourceDemo.class); @Value("${jdbc.driver}") private String driver; @Value("${jdbc.url}") private String url; @Value("${jdbc.username}") private String username; @Value("${jdbc.password}") private String password; @Autowired Environment env; @Override public void afterPropertiesSet() throws Exception { LOGGER.info(driver); LOGGER.info(url); LOGGER.info(password); LOGGER.info(username); setDatabaseConfig(); } private void setDatabaseConfig() { DataSourceConfig config = new DataSourceConfig(); config.setDriver(env.getProperty("jdbc.driver")); config.setUrl(env.getProperty("jdbc.url")); config.setUsername(env.getProperty("jdbc.username")); config.setPassword(env.getProperty("jdbc.password")); System.out.println(config.toString()); }}

## The Application.java File

This spring boot application has an entry point Java class called Application.java with the public static void main(String[] args) method, which you can run to start the application.

package net.guides.springboot2.springpropertysourceexample;import org.springframework.boot.SpringApplication;import org.springframework.boot.autoconfigure.SpringBootApplication;@SpringBootApplicationpublic class Application { public static void main(String[] args) { SpringApplication.run(Application.class, args); }}

[@SpringBootApplication](https://www.blogger.com/) is a convenience annotation that adds all of the following:

[@Configuration](http://www.javaguides.net/2018/09/spring-configuration-annotation-with-example.html) tags the class as a source of bean definitions for the application context.

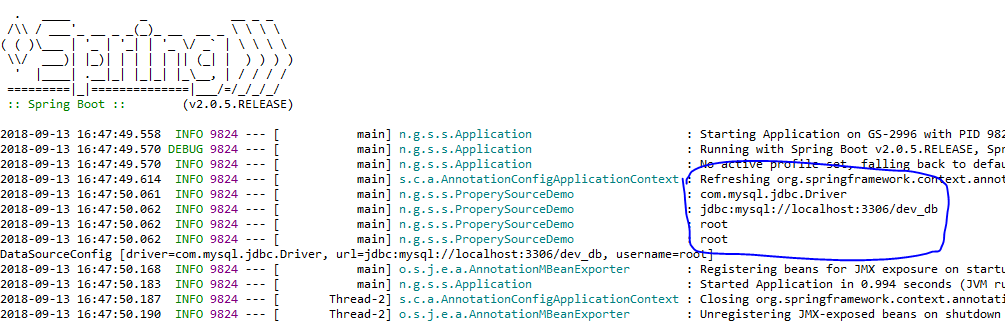
[@EnableAutoConfiguration](http://www.javaguides.net/2018/09/spring-boot-enableautoconfiguration-annotation-with-example.html) tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

Normally you would add @EnableWebMvc for a Spring MVC app, but Spring Boot adds it automatically when it sees spring-webmvc on the classpath. This flags the application as a web application and activates key behaviors such as setting up a DispatcherServlet.

@ComponentScan tells Spring to look for other components, configurations, and services in the hello package, allowing it to find the controllers.

Let's run Application.java class and oberve the console output.

## Output



Note that the database configurations are printed at the console using Environment class.

### **Spring Boot Annotations**

## **1. @SpringBootApplication**

[@SpringBootApplication](http://www.javaguides.net/2018/09/spring-boot-springbootapplication-annotation-with-example.html) annotation indicates a configuration class that declares one or more @Bean methods and also triggers auto-configuration and component scanning.

The [@SpringBootApplication](http://www.javaguides.net/2018/09/spring-boot-springbootapplication-annotation-with-example.html) annotation is equivalent to using [@Configuration](http://www.javaguides.net/2018/09/spring-configuration-annotation-with-example.html), [@EnableAutoConfiguration](http://www.javaguides.net/2018/09/spring-boot-enableautoconfiguration-annotation-with-example.html), and @ComponentScan with their default attributes.

[**@SpringBootApplication**](http://www.javaguides.net/2018/09/spring-boot-springbootapplication-annotation-with-example.html) **annotation example:** We use this annotation to mark the main class of a Spring Boot application:

import org.springframework.boot.SpringApplication;import org.springframework.boot.autoconfigure.SpringBootApplication;@SpringBootApplication // same as @Configuration @EnableAutoConfiguration @ComponentScanpublic class Application { public static void main(String[] args) { SpringApplication.run(Application.class, args); }}

Read more about [@SpringBootApplication](http://www.javaguides.net/2018/09/spring-boot-springbootapplication-annotation-with-example.html) annotation on [Spring Boot @SpringBootApplication Annotation with Example](http://www.javaguides.net/2018/09/spring-boot-springbootapplication-annotation-with-example.html).

## **2. @EnableAutoConfiguration**

[@EnableAutoConfiguration](http://www.javaguides.net/2018/09/spring-boot-enableautoconfiguration-annotation-with-example.html) annotation tells Spring Boot to “guess” how you want to configure Spring, based on the jar dependencies that you have added. Since spring-boot-starter-web dependency added to classpath leads to configure Tomcat and [Spring MVC](http://www.javaguides.net/search/label/Spring%20MVC%20Tutorial), the auto-configuration assumes that you are developing a web application and sets up Spring accordingly.

[**@EnableAutoConfiguration**](http://www.javaguides.net/2018/09/spring-boot-enableautoconfiguration-annotation-with-example.html) **annotation example:** Let's add [@EnableAutoConfiguration](http://www.javaguides.net/2018/09/spring-boot-enableautoconfiguration-annotation-with-example.html) annotation to Application class or Main class to enable an auto-configuration feature.

import org.springframework.boot.SpringApplication;import org.springframework.boot.autoconfigure.EnableAutoConfiguration;@EnableAutoConfigurationpublic class Application { public static void main(String[] args) { SpringApplication.run(Application.class, args); }}

Read more about @SpringBootApplication annotation on [Spring Boot @EnableAutoConfiguration Annotation with Example](http://www.javaguides.net/2018/09/spring-boot-enableautoconfiguration-annotation-with-example.html)

## **3. @ConditionalOnClass and @ConditionalOnMissingClass**

These annotations belong to Class conditions. The @ConditionalOnClass and @ConditionalOnMissingClassannotations let configuration be included based on the presence or absence of specific classes.

**Example:** In below example, using these conditions, Spring will only use the marked auto-configuration bean if the class in the annotation’s argument is present/absent:

@Configuration@ConditionalOnClass(DataSource.class)class OracleAutoconfiguration { //...}

## **4. @ConditionalOnBean and @ConditionalOnMissingBean**

The @ConditionalOnBean and @ConditionalOnMissingBean annotations let a bean be included based on the presence or absence of specific beans.

**@ConditionalOnBean annotation example:** Use when we want to define conditions based on the presence or absence of a specific bean:

@Bean@ConditionalOnBean(name = "dataSource")LocalContainerEntityManagerFactoryBean entityManagerFactory() { // ...}

**@ConditionalOnMissingBean annotation example:** When placed on a [@Bean](http://www.javaguides.net/2018/09/spring-bean-annotation-with-example.html) method, the target type defaults to the return type of the method, as shown in the following example:

@Configurationpublic class MyAutoConfiguration { @Bean @ConditionalOnMissingBean public MyService myService() { ... }}

In the preceding example, the myService bean is going to be created if no bean of type MyService is already contained in the ApplicationContext.

## **5. @ConditionalOnProperty**

The @ConditionalOnProperty annotation lets configuration be included based on a Spring Environmentproperty.

**@ConditionalOnProperty annotation example:** With this annotation, we can make conditions on the values of properties:

@Bean@ConditionalOnProperty( name = "usemysql", havingValue = "local")DataSource dataSource() { // ...}

## **6. @ConditionalOnResource**

The @ConditionalOnResource annotation lets configuration be included only when a specific resource is present:

@ConditionalOnResource(resources = "classpath:mysql.properties")Properties additionalProperties() { // ...}

## **7. @ConditionalOnWebApplication and @ConditionalOnNotWebApplication**

The @ConditionalOnWebApplication and @ConditionalOnNotWebApplication annotations let configuration be included depending on whether the application is a “web application”. A web application is an application that uses a Spring WebApplicationContext, defines a session scope, or has a StandardServletEnvironment.

**@ConditionalOnWebApplication annotation sample code:** With these annotations, we can create conditions based on if the current application is or isn’t a web application:

@ConditionalOnWebApplicationHealthCheckController healthCheckController() { // ...}

## **8. @ConditionalExpression**

We can use this annotation in more complex situations. Spring will use the marked definition when the SpEL expression is evaluated to true:

@Bean@ConditionalOnExpression("${usemysql} && ${mysqlserver == 'local'}")DataSource dataSource() { // ...}

## **9. @Conditional**

For even more complex conditions, we can create a class evaluating the custom condition. We tell Spring to use this custom condition with @Conditional:

@Conditional(HibernateCondition.class)Properties additionalProperties() { //...}

### **Spring Scheduling Annotations**

## **1. @Scheduled Annotation**

The [@Scheduled](http://www.javaguides.net/2018/10/spring-boot-2-scheduling-tasks.html) annotation is added to a method along with some information about when to execute it, and Spring takes care of the rest.

For example, the following method would be invoked every 5 seconds with a fixed delay, meaning that the period will be measured from the completion time of each preceding invocation.

@Scheduled(fixedDelay=5000)public void doSomething() { // something that should execute periodically}

If a fixed rate execution is desired, simply change the property name specified within the annotation. The following would be executed every 5 seconds measured between the successive start times of each invocation.

@Scheduled(fixedRate=5000)public void doSomething() { // something that should execute periodically}

For fixed-delay and fixed-rate tasks, an initial delay may be specified indicating the number of milliseconds to wait before the first execution of the method.

@Scheduled(initialDelay=1000, fixedRate=5000)public void doSomething() { // something that should execute periodically}

If simple periodic scheduling is not expressive enough, then a cron expression may be provided. For example, the following will only execute on weekdays.

@Scheduled(cron="\*/5 \* \* \* \* MON-FRI")public void doSomething() { // something that should execute on weekdays only}

## **2. @EnableScheduling Annotation**

@EnableScheduling annotation is used to enable scheduling in the application. We also have to use it in conjunction with [@Configuration](http://www.javaguides.net/2018/09/spring-configuration-annotation-with-example.html):

@Configuration@EnableSchedulingpublic class Appconfig{}

As a result, we can now run methods periodically with [@Scheduled](http://www.javaguides.net/2018/10/spring-boot-2-scheduling-tasks.html).

## **3. @Async Annotation**

The @Async annotation can be provided on a method so that invocation of that method will occur asynchronously. In other words, the caller will return immediately upon invocation and the actual execution of the method will occur in a task that has been submitted to a Spring TaskExecutor. In the simplest case, the annotation may be applied to a void-returning method.

@Asyncvoid doSomething() { // this will be executed asynchronously}

Unlike the methods annotated with the [@Scheduled](http://www.javaguides.net/2018/10/spring-boot-2-scheduling-tasks.html) annotation, these methods can expect arguments, because they will be invoked in the "normal" way by callers at runtime rather than from a scheduled task being managed by the container.

For example, the following is a legitimate application of the @Async annotation.

@Asyncvoid doSomething(String s) { // this will be executed asynchronously}

Even methods that return a value can be invoked asynchronously. However, such methods are required to have a Future typed return value. This still provides the benefit of asynchronous execution so that the caller can perform other tasks prior to calling get() on that Future.

@AsyncFuture<String> returnSomething(int i) { // this will be executed asynchronously}

@Async cannot be used in conjunction with lifecycle callbacks such as @PostConstruct. To asynchronously initialize Spring beans you currently have to use a separate initializing Spring bean that invokes the @Asyncannotated method on the target then.

public class SampleBeanImpl implements SampleBean { @Async void doSomething() { // ... }}public class SampleBeanInitializer { private final SampleBean bean; public SampleBeanInitializer(SampleBean bean) { this.bean = bean; } @PostConstruct public void initialize() { bean.doSomething(); }}

## **4. @EnableAsync Annotation**

With this annotation, we can enable asynchronous functionality in Spring. We must use it with [@Configuration](http://www.javaguides.net/2018/09/spring-configuration-annotation-with-example.html):

@Configuration@EnableAsyncpublic class AppConfig{}

Now, that we enabled asynchronous calls, we can use @Async to define the methods supporting it.

## **5. @Schedules Annotation**

We can use this annotation to specify multiple [@Scheduled](http://www.javaguides.net/2018/10/spring-boot-2-scheduling-tasks.html) rules:

@Schedules({ @Scheduled(fixedRate = 10000), @Scheduled(cron = "0 \* \* \* \* MON-FRI")})void checkVehicle() { // ...}