Spring MVC Interview Questions

Last modified: December 29, 2018

by baeldung (/author/baeldung/)

Spring MVC (/category/spring/spring-web/spring-mvc/)

Interview (/tag/interview/)

1. Introduction

Spring MVC is the original web framework from Spring built on the Servlet API. It provides Model-View-Controller architecture that can be used to develop flexible web applications.

In this tutorial, we'll focus on the questions related to it, as it is often a topic on a Spring developer job interview.

For more questions on the Spring Framework, you can check out another Spring related article (https://www.baeldung.com/spring-interview-questions) of our interview questions series (https://www.baeldung.com/tag/interview/).

2. Basic Spring MVC Questions

Q1. Why Should We Use Spring MVC?

Spring MVC implements a clear separation of concerns that allows us to develop and unit test our applications easily.

The concepts like:

- Dispatcher Servlet
- Controllers
- View Resolvers
- Views, Models
- *ModelAndView*
- Model and Session Attributes

are completely independent of each other, and they are responsible for one thing only.

Therefore, **MVC gives us quite big flexibility**. It's based on interfaces (with provided implementation classes), and we can configure every part of the framework by using custom interfaces.

Another important thing is that we aren't tied to a specific view technology (for example, JSP), but we have the option to choose from the ones we like the most.

Also, **we don't use Spring MVC only in web applications development but in the creation of RESTful web services as well**.

Q2. What is the Role of the *@Autowired* Annotation?

The *@Autowired* annotation can be used with fields or methods for injecting a bean by type. This annotation allows Spring to resolve and inject collaborating beans into your bean.

For more details, please refer to the tutorial about *@Autowired* in Spring (https://www.baeldung.com/spring-autowire).

Q3. Explain a Model Attribute

The *@ModelAttribute* annotation is one of the most important annotations in Spring MVC. **It binds a method parameter or a method return value to a named model attribute and then exposes it to a web view**.

If we use it at the method level, it indicates the purpose of that method is to add one or more model attributes.

On the other hand, when used as a method argument, it indicates the argument should be retrieved from the model. When not present, we should first instantiate it and then add it to the model. Once present in the model, we should populate the arguments fields from all request parameters that have matching names.

More about this annotation can be found in our article related to the *@ModelAttribute* annotation (https://www.baeldung.com/spring-mvc-and-the-modelattribute-annotation).

Q4. Explain the Difference Between *@Controller* and *@RestController*?

The main difference between the *@Controller* and *@RestController* annotations is that **the *@ResponseBody* annotation is automatically included in the *@RestController***. This means that we don't need to annotate our handler methods with the *@ResponseBody*. We need to do this in a *@Controller* class if we want to write response type directly to the HTTP response body.

Q5. Describe a *PathVariable*

We can use the *@PathVariable* annotation as a handler method parameter in order to extract the value of a URI template variable.

For example, if we want to fetch a user by id from the *www.mysite.com/user/123*, we should map our method in the controller as */user/{id}*:

```
@RequestMapping("/user/{id}")
public String handleRequest(@PathVariable("id") String userId, Model map
```

The *@PathVariable* has only one element named *value*. It's optional and we use it to define the URI template variable name. If we omit the value element, then the URI template variable name must match the method parameter name.

It's also allowed to have multiple *@PathVariable* annotations, either by declaring them one after another:

```
@RequestMapping("/user/{userId}/name/{userName}")
public String handleRequest(@PathVariable String userId,
  @PathVariable String userName, Model map) {}
```

or putting them all in a *Map<String, String>* or *MultiValueMap<String, String>*:

```
@RequestMapping("/user/{userId}/name/{userName}")
public String handleRequest(@PathVariable Map<String, String> varsMap, M
```

Q6. Validation Using Spring MVC

Spring MVC supports JSR-303 specifications by default. We need to add JSR-303 and its implementation dependencies to our Spring MVC application. Hibernate Validator, for example, is one of the JSR-303 implementations at our disposal.

JSR-303 is a specification of the Java API for bean validation, part of JavaEE and JavaSE, which ensures that properties of a bean meet specific criteria, using annotations such as *@NotNull*, *@Min*, and *@Max*. More about validation is available in the Java Bean Validation Basics (https://www.baeldung.com/javax-validation) article.

Spring offers the *@Validator* annotation and the *BindingResult* class. The *Validator* implementation will raise errors in the controller request handler method when we have invalid data. Then we may use the *BindingResult* class to get those errors.

Besides using the existing implementations, we can make our own. To do so, we create an annotation that conforms to the JSR-303 specifications first. Then, we implement the *Validator* class. Another way would be to implement Spring's *Validator (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/validation/Validator.html)* interface and set it as the validator via *@InitBinder* annotation in *Controller* class.

To check out how to implement and use your own validations, please see the tutorial regarding Custom Validation in Spring MVC (https://www.baeldung.com/spring-mvc-custom-validator).

Q7. What are the *@RequestBody* and the *@ResponseBody*?

The *@RequestBody* annotation, used as a handler method parameter, binds the HTTP Request body to a transfer or a domain object. Spring automatically deserializes incoming HTTP Request to the Java object using Http Message Converters.

When we use the *@ResponseBody* annotation on a handler method in the Spring MVC controller, it indicates that we'll write the return type of the method directly to the HTTP response body. We'll not put it in a *Model*, and Spring won't interpret as a view name.

Please check out the article on *@RequestBody* and *@ResponseBody* (https://www.baeldung.com/spring-request-response-body) to see more details about these annotations.

Q8. Explain *Model*, *ModelMap* and *ModelAndView*?

The *Model* interface defines a holder for model attributes. **The *ModelMap* has a similar purpose, with the ability to pass a collection of values**. It then treats those values as if they were within a *Map*. We should note that in *Model* (*ModelMap*) we can only store data. We put data in and return a view name.

On the other hand, **with the *ModelAndView*, we return the object itself**. We set all the required information, like the data and the view name, in the object we're returning.

You can find more details in the article on *Model, ModelMap*, and *ModelView* (https://www.baeldung.com/spring-mvc-model-model-map-model-view).

Q9. Explain *SessionAttributes* and *SessionAttribute*

The *@SessionAttributes* annotation is used for storing the model attribute in the user's session. We use it at the controller class level, as shown in our article about the Session Attributes in Spring MVC (https://www.baeldung.com/spring-mvc-session-attributes):

```
@Controller
@RequestMapping("/sessionattributes")
@SessionAttributes("todos")
public class TodoControllerWithSessionAttributes {

    @GetMapping("/form")
    public String showForm(Model model,
      @ModelAttribute("todos") TodoList todos) {
        // method body
        return "sessionattributesform";
    }

    // other methods
}
```

In the previous example, the model attribute '*todos*' will be added to the session if the *@ModelAttribute* and the *@SessionAttributes* have the same name attribute.

If we want to retrieve the existing attribute from a session that is managed globally, we'll use *@SessionAttribute* annotation as a method parameter:

```
@GetMapping
public String getTodos(@SessionAttribute("todos") TodoList todos) {
    // method body
    return "todoView";
}
```

Q10. What is the Purpose of *@EnableWebMVC*?

The *@EnableWebMvc* annotation's purpose is to enable Spring MVC via Java configuration. It's equivalent to *<mvc: annotation-driven>* in an XML configuration. This annotation imports Spring MVC Configuration from *WebMvcConfigurationSupport*. It enables support for *@Controller*-annotated classes that use *@RequestMapping* to map incoming requests to a handler method.

You can learn more about this and similar annotations in our Guide to the Spring *@Enable* Annotations (https://www.baeldung.com/spring-enable-annotations).

Q11. What is *ViewResolver* in Spring?

The *ViewResolver* enables an application to render models in the browser – without tying the implementation to a specific view technology – **by mapping view names to actual views**.

For more details about the *ViewResolver*, have a look at our Guide to the ViewResolver in Spring MVC (https://www.baeldung.com/spring-mvc-view-resolver-tutorial).

Q12. What is the *BindingResult*?

***BindingResult* is an interface from *org.springframework.validation* package that represents binding results. We can use it to detect and report errors in the submitted form**. It's easy to invoke — we just need to ensure that we put it as a parameter right after the form object we're validating. The optional *Model* parameter should come after the *BindingResult*, as it can be seen in the custom validator tutorial (https://www.baeldung.com/spring-mvc-custom-validator):

```java
@PostMapping("/user")
public String submitForm(@Valid NewUserForm newUserForm,
  BindingResult result, Model model) {
    if (result.hasErrors()) {
        return "userHome";
    }
    model.addAttribute("message", "Valid form");
    return "userHome";
}
```

When Spring sees the *@Valid* annotation, it'll first try to find the validator for the object being validated. Then it'll pick up the validation annotations and invoke the validator. Finally, it'll put found errors in the *BindingResult* and add the latter to the view model.

Q13. What is a Form Backing Object?

The form backing object or a Command Object is just a POJO that collects data from the form we're submitting.

We should keep in mind that it doesn't contain any logic, only data.

To learn how to use form backing object with the forms in Spring MVC, please take a look at our article about Forms in Spring MVC (https://www.baeldung.com/spring-mvc-form-tutorial).

Q14. What is the Role of the *@Qualifier* Annotation?

It is used simultaneously with the *@Autowired* annotation to avoid confusion when multiple instances of a bean type are present.

Let's see an example. We declared two similar beans in XML config:

```xml
<bean id="person1" class="com.baeldung.Person" >
    <property name="name" value="Joe" />
</bean>
<bean id="person2" class="com.baeldung.Person" >
    <property name="name" value="Doe" />
</bean>
```

When we try to wire the bean, we'll get an *org.springframework.beans.factory.NoSuchBeanDefinitionException.* To fix it, we need to use *@Qualifier* to tell Spring about which bean should be wired:

```java
@Autowired
@Qualifier("person1")
private Person person;
```

Q15. What is the Role of the *@Required* Annotation?

The *@Required* annotation is used on setter methods, and it indicates that the bean property that has this annotation must be populated at configuration time. Otherwise, the Spring container will throw a *BeanInitializationException* exception.

Also, *@Required* differs from *@Autowired* – as it is limited to a setter, whereas *@Autowired* is not. *@Autowired* can be used to wire with a constructor and a field as well, while *@Required* only checks if the property is set.

Let's see an example:

```java
public class Person {
    private String name;

    @Required
    public void setName(String name) {
        this.name = name;
    }
}
```

Now, the *name* of the *Person* bean needs to be set in XML config like this:

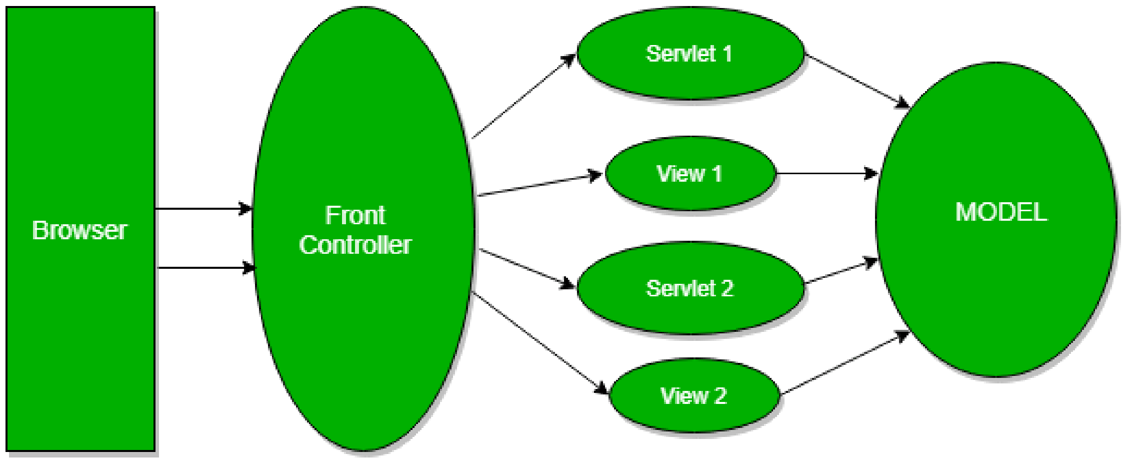
```xml
<bean id="person" class="com.baeldung.Person">
    <property name="name" value="Joe" />
</bean>
```

Please note that ***@Required* doesn't work with Java based *@Configuration* classes by default**. If you need to make sure that all your properties are set, you can do so when you create the bean in the *@Bean* annotated methods.

Q16. Describe the Front Controller Pattern

In the Front Controller pattern, all requests will first go to the front controller instead of the servlet. **It'll make sure that the responses are ready and will send them back to the browser. This way we have one place where we control everything that comes from the outside world.**

The front controller will identify the servlet that should handle the request first. Then, when it gets the data back from the servlet, it'll decide which view to render and, finally, it'll send the rendered view back as a response:

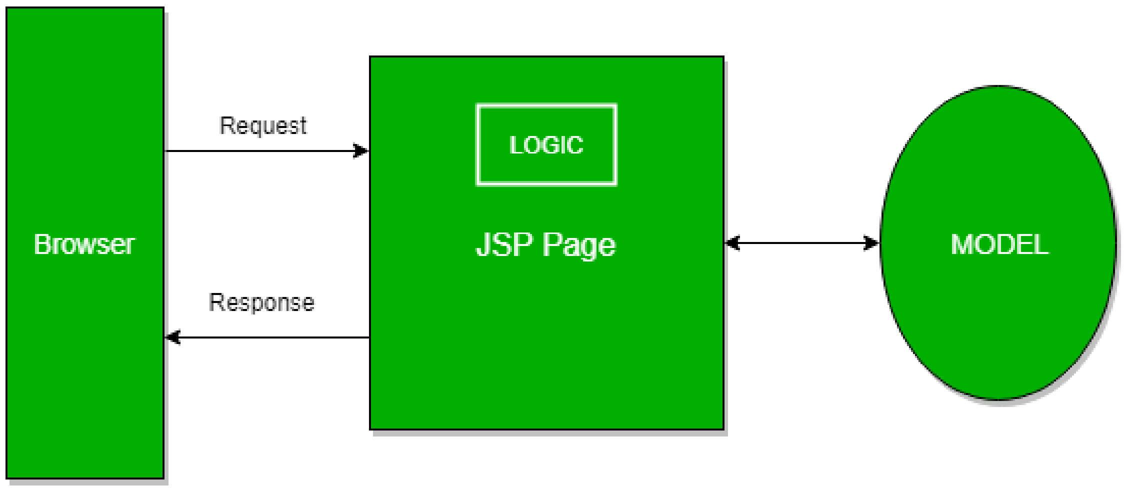


(https://www.baeldung.com/wp-content/uploads/2018/12/front_end_controller.png)
To see the implementation details, please check out our Guide to the Front Controller Pattern in Java (https://www.baeldung.com/java-front-controller-pattern).

Q17. What are Model 1 and Model 2 Architectures?

Model 1 and Model 2 represent two frequently used design models when it comes to designing Java Web Applications.

In Model 1, a request comes to a servlet or JSP where it gets handled. The servlet or the JSP processes the request, handles business logic, retrieves and validates data, and generates the response:



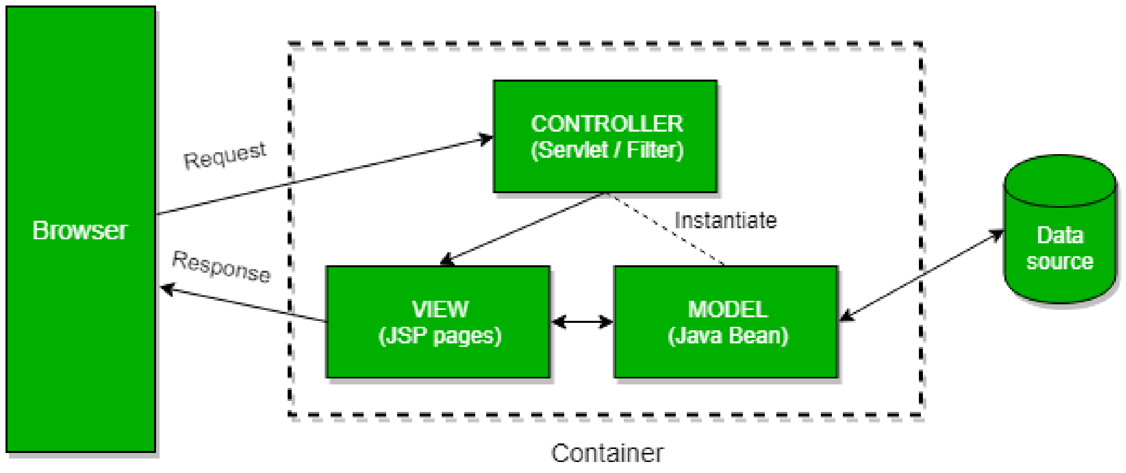
(https://www.baeldung.com/wp-content/uploads/2018/12/Model_1-1.png)
Since this architecture is easy to implement, we usually use it in small and simple applications.

On the other hand, it isn't convenient for large-scale web applications. The functionalities are often duplicated in JSPs where business and presentation logic are coupled.

The Model 2 is based on the Model View Controller design pattern and it separates the view from the logic that manipulates the content.

Furthermore, we can distinguish three modules in the MVC pattern: the model, the view, and the controller. The model is representing the dynamic data structure of an application. It's responsible for the data and business logic manipulation. The view is in charge of displaying the data, while the controller serves as an interface between the previous two.

In Model 2, a request is passed to the controller, which handles the required logic in order to get the right content that should be displayed. The controller then puts the content back into the request, typically as a JavaBean or a POJO. It also decides which view should render the content and finally passes the request to it. Then, the view renders the data:



(https://www.baeldung.com/wp-content/uploads/2018/12/Model_2.png)

3. Advanced Spring MVC Questions

Q18. What's the Difference Between *@Controller*, *@Component*, *@Repository,* and *@Service* Annotations in Spring?

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

Let's take a look at specific use cases of last three:

- **@*Controller*** – indicates that the class serves the role of a controller, and detects *@RequestMapping* annotations within the class
- **@*Service*** – indicates that the class holds business logic and calls methods in the repository layer
- **@*Repository*** – indicates that the class defines a data repository; its job is to catch platform-specific exceptions and re-throw them as one of Spring's unified unchecked exceptions

Q19. What are *DispatcherServlet* and *ContextLoaderListener*?

Simply put, in the Front Controller design pattern, a single controller is responsible for directing incoming *HttpRequests* to all of an application's other controllers and handlers.

Spring's *DispatcherServlet* implements this pattern and is, therefore, responsible for correctly coordinating the *HttpRequests* to the right handlers.

On the other hand, *ContextLoaderListener* starts up and shuts down Spring's root *WebApplicationContext*. It ties the lifecycle of *ApplicationContext* to the lifecycle of the *ServletContext.* We can use it to define shared beans working across different Spring contexts.

For more details on *DispatcherServlet*, please refer to this tutorial (/spring-dispatcherservlet).

Q20. What is a *MultipartResolver* and When Should We Use It?

The *MultipartResolver* interface is used for uploading files. The Spring framework provides one *MultipartResolver* implementation for use with Commons FileUpload and another for use with Servlet 3.0 multipart request parsing.

Using these, we can support file uploads in our web applications.

Q21. What is Spring MVC Interceptor and How to Use It?

Spring MVC Interceptors allow us to intercept a client request and process it at three places – before handling, after handling, or after completion (when the view is rendered) of a request.

The interceptor can be used for cross-cutting concerns and to avoid repetitive handler code like logging, changing globally used parameters in Spring model, etc.

For details and various implementations, take a look at Introduction to Spring MVC HandlerInterceptor (https://www.baeldung.com/spring-mvc-handlerinterceptor) article.

Q22. What is an Init Binder?

A method annotated with *@InitBinder* is used to customize a request parameter, URI template, and backing/command objects. We define it in a controller and it helps in controlling the request. **In this method, we register and configure our custom *PropertyEditor*s, a formatter, and validators**.

The annotation has the '*value*' element. If we don't set it, the *@InitBinder* annotated methods will get called on each HTTP request. If we set the value, the methods will be applied only for particular command/form attributes and/or request parameters whose names correspond to the '*value*' element.

It's important to remember that one of the arguments must be *WebDataBinder.* Other arguments can be of any type that handler methods support except for command/form objects and corresponding validation result objects.

Q23. Explain a Controller Advice

The *@ControllerAdvice* annotation allows us to write global code applicable to a wide range of controllers. We can tie the range of controllers to a chosen package or a specific annotation.

By default, ***@ControllerAdvice* applies to the classes annotated with *@Controller* (or *@RestController*)**. We also have a few properties that we use if we want to be more specific.

If we want to restrict applicable classes to a package, we should add the name of the package to the annotation:

```
@ControllerAdvice("my.package")
@ControllerAdvice(value = "my.package")
@ControllerAdvice(basePackages = "my.package")
```

It's also possible to use multiple packages, but this time we need to use an array instead of the *String*.

Besides restricting to the package by its name, we can do it by using one of the classes or interfaces from that package:

```
@ControllerAdvice(basePackageClasses = MyClass.class)
```

The '*assignableTypes*' element applies the *@ControllerAdvice* to the specific classes, while '*annotations*' does it for particular annotations.

It's noteworthy to remember that we should use it along with *@ExceptionHandler*. This combination will enable us to configure a global and more specific error handling mechanism without the need to implement it every time for every controller class.

Q24. What Does the *@ExceptionHandler* Annotation Do?

The *@ExceptionHandler* annotation allows us to define a method that will handle the exceptions. **We may use the annotation independently, but it's a far better option to use it together with the *@ControllerAdvice*.** Thus, we can set up a global error handling mechanism. **In this way, we don't need to write the code for the exception handling within every controller**.

Let's take a look at the example from our article about Error Handling for REST with Spring (https://www.baeldung.com/exception-handling-for-rest-with-spring):

```
@ControllerAdvice
public class RestResponseEntityExceptionHandler
  extends ResponseEntityExceptionHandler {

    @ExceptionHandler(value = { IllegalArgumentException.class,
      IllegalStateException.class })
    protected ResponseEntity<Object> handleConflict(RuntimeException ex
      WebRequest request) {
        String bodyOfResponse = "This should be application specific";
        return handleExceptionInternal(ex, bodyOfResponse, new HttpHead
          HttpStatus.CONFLICT, request);
    }
}
```

We should also note that this will provide *@ExceptionHandler* methods to all controllers that throw *IllegalArgumentException* or *IllegalStateException*. The exceptions declared with *@ExceptionHandler* should match the exception used as the argument of the method. Otherwise, the exception resolving mechanism will fail at runtime.

One thing to keep in mind here is that it's possible to define more than one *@ExceptionHandler* for the same exception. We can't do it in the same class though since Spring would complain by throwing an exception and failing on startup.

On the other hand, **if we define those in two separate classes, the application will start, but it'll use the first handler it finds, possibly the wrong one**.

Q25. Exception Handling in Web Applications

We have three options for exceptions handling in Spring MVC:

- per exception
- per controller
- globally

If an unhandled exception is thrown during web request processing, the server will return an HTTP 500 response. To prevent this, **we should annotate any of our custom exceptions with the *@ResponseStatus* annotation**. **This kind of exceptions is resolved by *HandlerExceptionResolver***.

This will cause the server to return an appropriate HTTP response with the specified status code when a controller method throws our exception. We should keep in mind that we shouldn't handle our exception somewhere else for this approach to work.

Another way to handle the exceptions is by using the *@ExceptionHandler* annotation. We add *@ExceptionHandler* methods to any controller and use them to handle the exceptions thrown from inside that controller. These methods can handle exceptions without the *@ResponseStatus* annotation, redirect the user to a dedicated error view, or build a totally custom error response.

We can also pass in the servlet-related objects (*HttpServletRequest*, *HttpServletResponse*, *HttpSession*, and *Principal*) as the parameters of the handler methods. But, we should remember that we can't put the *Model* object as the parameter directly.

The third option for handling errors is by *@ControllerAdvice* classes. It'll allow us to apply the same techniques, only this time at the application level and not only to the particular controller. To enable this, we need to use the *@ControllerAdvice* and the *@ExceptionHandler* together. This way exception handlers will handle exceptions thrown by any controller.

For more detailed information on this topic, go through the Error Handling for REST with Spring (https://www.baeldung.com/exception-handling-for-rest-with-spring) article.

4. Conclusion

In this article, we've explored some of the Spring MVC related questions that could come up at the technical interview for Spring developers. You should take these questions into account as a starting point for further research since this is by no means an exhaustive list.

We wish you good luck in any upcoming interviews!

Leave a Reply

Start the discussion...

Subscribe

CATEGORIES

SPRING (/CATEGORY/SPRING/)
REST (/CATEGORY/REST/)
JAVA (/CATEGORY/JAVA/)
SECURITY (/CATEGORY/SECURITY-2/)
PERSISTENCE (/CATEGORY/PERSISTENCE/)
JACKSON (/CATEGORY/JSON/JACKSON/)
HTTP CLIENT (/CATEGORY/HTTP/)
KOTLIN (/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE COURSES (HTTPS://COURSES.BAELDUNG.COM)
CONSULTING WORK (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS)
ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)