



Build Your API with Spring

1: Bootstrap a Web Application with Spring 5

1. Overview.....	1
2. Bootstrapping Using Spring Boot.....	2
2.1. Maven Dependency.....	2
2.2. Creating a Spring Boot Application.....	2
3. Bootstrapping Using spring-webmvc.....	4
3.1. Maven Dependencies.....	4
3.2. The Java-based Web Configuration.....	4
3.3. The Initializer Class.....	4
4. XML Configuration.....	6
5. Conclusion.....	7

2: Build a REST API with Spring and Java Config

1. Overview.....	9
2. Understanding REST in Spring.....	10
3. The Java Configuration.....	11
3.1. Using Spring Boot.....	11
4. Testing the Spring Context.....	12
4.1. Using Spring Boot.....	12

5. The Controller.....	14
6. Mapping the HTTP Response Codes.....	16
6.1. Unmapped Requests.....	16
6.2. Valid Mapped Requests.....	16
6.3. Client Error.....	16
6.4. Using <i>@ExceptionHandler</i>	17
7. Additional Maven Dependencies.....	18
7.1. Using Spring Boot.....	18
8. Conclusion.....	19

3: Http Message Converters with the Spring Framework

1. Overview.....	21
2. The Basics.....	22
2.1. Enable Web MVC.....	22
2.2. The Default Message Converters.....	22
3. Client-Server Communication – JSON only.....	23
3.1. High-Level Content Negotiation.....	23
3.2. <i>@ResponseBody</i>	24
3.3. <i>@RequestBody</i>	24

4. Custom Converters Configuration.....	26
4.1. Spring Boot Support.....	27
5. Using Spring's RestTemplate with Http Message Converters.....	28
5.1. Retrieving the Resource with no <i>Accept</i> Header.....	28
5.2. Retrieving a Resource with <i>application/xml</i> <i>Accept</i> Header.....	28
5.3. Retrieving a Resource with <i>application/json</i> <i>Accept</i> Header.....	29
5.4. Update a Resource with XML <i>Content-Type</i>	30
6. Conclusion.....	31

4: Spring's RequestBody and ResponseBody Annotations

1. Introduction.....	33
2. <i>@RequestBody</i>	34
3. <i>@ResponseBody</i>	35
4. Conclusion.....	36

5: Entity To DTO Conversion for a Spring REST API

1. Overview.....	38
2. Model Mapper.....	39

3. The DTO	40
4. The Service Layer.....	41
5. The Controller Layer.....	42
6. Unit Testing.....	44
7. Conclusion.....	45

6: Error Handling for REST with Spring

1. Overview.....	47
2. Solution 1 – The Controller level <i>@ExceptionHandler</i>	48
3. Solution 2 – <i>The HandlerExceptionHandlerResolver</i>	49
3.1. <i>ExceptionHandlerExceptionHandlerResolver</i>	49
3.2. <i>DefaultHandlerExceptionHandlerResolver</i>	49
3.3. <i>ResponseStatusExceptionHandlerResolver</i>	49
3.4. <i>Custom HandlerExceptionHandlerResolver</i>	50
4. Solution 3 – <i>@ControllerAdvice</i>	52
5. Solution 4 – <i>ResponseStatusException</i>	53
6. Handle the Access Denied in Spring Security.....	54
6.1. REST and Method Level Security.....	54

7. Spring Boot Support.....	55
8. Conclusion.....	57

7: REST API Discoverability and HATEOAS

1. Overview.....	59
2. Why Make the API Discoverable.....	60
3. Discoverability Scenarios (Driven by Tests).....	61
3.1. Discover the Valid HTTP Methods.....	61
3.2. Discover the URI of Newly Created Resource.....	62
3.2. Discover the URI of Newly Created Resource.....	62
3.3. Discover the URI to GET All Resources of That Type.....	63
4. Other Potential Discoverable URIs and Microformats.....	64
5. Conclusion.....	65

8: An Intro to Spring HATEOAS

1. Overview.....	67
2. Spring-HATEOAS.....	68
3. Preparation.....	69

4. Adding HATEOAS Support.....	70
4.1. Adding Hypermedia Support to a Resource.....	70
4.2. Creating Links.....	70
4.3. Creating Better Links.....	71
5. Relations.....	72
6. Links to Controller Methods.....	73
7. Spring HATEOAS in Action.....	74
8. Conclusion.....	77

9: REST Pagination in Spring

1. Overview.....	79
2. Page as Resource vs Page as Representation.....	80
3. The Controller.....	81
4. Discoverability for REST Pagination.....	82
5. Test Driving Pagination.....	84
6. Test Driving Pagination Discoverability.....	85
7. Getting All Resources.....	86

8. REST Paging with Range HTTP Headers.....	87
9. Spring Data REST Pagination.....	88
10. Conclusion.....	89

10: Test a REST API with Java

1. Overview.....	91
2. Testing the Status Code.....	92
3. Testing the Media Type.....	93
4. Testing the JSON Payload.....	94
5. Utilities for Testing.....	95
6. Dependencies.....	96
7. Conclusion.....	97



1: Bootstrap a Web Application with Spring 5



The chapter illustrates how to **Bootstrap a Web Application with Spring**.

We'll look into the Spring Boot solution for bootstrapping the application and also see a non-Spring Boot approach.

We'll primarily use Java configuration, but also have a look at their equivalent XML configuration.



2.1. Maven Dependency

First, we'll need the spring-boot-starter-web dependency:

```
1. <dependency>
2.     <groupId>org.springframework.boot</groupId>
3.     <artifactId>spring-boot-starter-web</artifactId>
4.     <version>2.1.1.RELEASE</version>
5. </dependency>
```

This starter includes:

- *spring-web* and the *spring-webmvc* module that we need for our Spring web application
- a Tomcat starter so that we can run our web application directly without explicitly installing any server

2.2. Creating a Spring Boot Application

The most straight-forward way to get started using Spring Boot is to create a main class and annotate it with **@SpringBootApplication**:

```
1. @SpringBootApplication
2. public class SpringBootRestApplication {
3.
4.     public static void main(String[] args) {
5.         SpringApplication.run(SpringBootRestApplication.class, args);
6.     }
7. }
```

This single annotation is equivalent to using *@Configuration*, *@EnableAutoConfiguration*, and *@ComponentScan*.

By default, it will scan all the components in the same package or below.

Next, for Java-based configuration of Spring beans, we need to create a config class and annotate it with **@Configuration** annotation:

```
1. @Configuration
2. public class WebConfig {
3. }
```

This annotation is the main artifact used by the Java-based Spring configuration; it is itself meta-annotated with `@Component`, which makes the annotated classes standard beans and as such, also candidates for component-scanning.

The main purpose of `@Configuration` classes is to be sources of bean definitions for the Spring IoC Container. For a more detailed description, see the [official docs](#).

Let's also have a look at a solution using the core *spring-webmvc* library.



3.1. Maven Dependencies

First, we need the spring-webmvc dependency:

```
1. <dependency>
2.     <groupId>org.springframework</groupId>
3.     <artifactId>spring-webmvc</artifactId>
4.     <version>5.0.0.RELEASE</version>
5. </dependency>
```

3.2. The Java-based Web Configuration

Next, we'll add the configuration class that has the `@Configuration` annotation:

```
1. @Configuration
2. @EnableWebMvc
3. @ComponentScan(basePackages = "com.baeldung.controller")
4. public class WebConfig {
5.
6. }
```

Here, unlike the Spring Boot solution, we'll have to explicitly define `@EnableWebMvc` for setting up default Spring MVC Configurations and `@ComponentScan` to specify packages to scan for components.

The `@EnableWebMvc` annotation provides the Spring Web MVC configuration such as setting up the dispatcher servlet, enabling the `@Controller` and the `@RequestMapping` annotations and setting up other defaults.

`@ComponentScan` configures the component scanning directive, specifying the packages to scan.

3.3. The Initializer Class

Next, we need to add a class that implements the `WebApplicationInitializer` interface:

```

1. public class AppInitializer implements WebApplicationInitializer {
2.
3.     @Override
4.     public void onStartup(ServletContext container) throws
5.     ServletException {
6.         AnnotationConfigWebApplicationContext context = new
7.         AnnotationConfigWebApplicationContext();
8.         context.scan("com.baeldung");
9.         container.addListener(new ContextLoaderListener(context));
10.
11.         ServletRegistration.Dynamic dispatcher =
12.             container.addServlet("mvc", new DispatcherServlet(context));
13.         dispatcher.setLoadOnStartup(1);
14.         dispatcher.addMapping("/");
15.     }
16. }

```

Here, we're creating a Spring context using the *AnnotationConfigWebApplicationContext* class, which means we're using only annotation-based configuration. Then, we're specifying the packages to scan for components and configuration classes.

Finally, we're defining the entry point for the web application – *the DispatcherServlet*.

This class can entirely replace the *web.xml* file from <3.0 Servlet versions



Let's also have a quick look at the equivalent XML web configuration:

```
<context:component-scan base-package="com.baeldung.controller" />  
<mvc:annotation-driven />
```

We can replace this XML file with the *WebConfig* class above.

To start the application, we can use an Initializer class that loads the XML configuration or a web.xml file.



In this chapter, we looked into two popular solutions for bootstrapping a Spring web application, one using the Spring Boot web starter and other using the core spring-webmvc library.

As always, the code presented in this chapter is available [over on Github](#).



2: Build a REST API with Spring and Java Config



This chapter shows how to **set up REST in Spring** – the Controller and HTTP response codes, configuration of payload marshalling and content negotiation.

2. Understanding REST in Spring



The Spring framework supports two ways of creating RESTful services:

- using MVC with `ModelAndView`
- using HTTP message converters

The *ModelAndView* approach is older and much better documented, but also more verbose and configuration heavy. It tries to shoehorn the REST paradigm into the old model, which is not without problems. The Spring team understood this and provided first-class REST support starting with Spring 3.0.

The new approach, based on *HttpMessageConverter* and annotations, is much more **lightweight and easy to implement**. Configuration is minimal, and it provides sensible defaults for what you would expect from a RESTful service.



```
1. @Configuration
2. @EnableWebMvc
3. public class WebConfig{
4.     //
5. }
```

The new `@EnableWebMvc` annotation does some useful things – specifically, in the case of REST, it detects the existence of Jackson and JAXB 2 on the classpath and automatically creates and registers default JSON and XML converters. The functionality of the annotation is equivalent to the XML version:

`<mvc:annotation-driven />`

This is a shortcut, and though it may be useful in many situations, it's not perfect. When more complex configuration is needed, remove the annotation and extend `WebMvcConfigurationSupport` directly.

3.1. Using Spring Boot

If we're using the `@SpringBootApplication` annotation and the `spring-webmvc` library is on the classpath, then the `@EnableWebMvc` annotation is added automatically with [a default autoconfiguration](#).

We can still add MVC functionality to this configuration by implementing the `WebMvcConfigurer` interface on a `@Configuration` annotated class. We can also use a `WebMvcRegistrationsAdapter` instance to provide our own `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`, or `ExceptionHandlerExceptionResolver` implementations.

Finally, if we want to discard Spring Boot's MVC features and declare a custom configuration, we can do so by using the `@EnableWebMvc` annotation.

4. Testing the Spring Context



Starting with Spring 3.1, we get first-class testing support for `@Configuration` classes:

```
1. @RunWith(SpringJUnit4ClassRunner.class)
2. @ContextConfiguration(
3.     classes = {WebConfig.class, PersistenceConfig.class},
4.     loader = AnnotationConfigContextLoader.class)
5. public class SpringTest {
6.
7.     @Test
8.     public void whenSpringContextIsInstantiated_thenNoExceptions() {
9.         // When
10.    }
11. }
```

We're specifying the Java configuration classes with the `@ContextConfiguration` annotation. The new `AnnotationConfigContextLoader` loads the bean definitions from the `@Configuration` classes.

Notice that the `WebConfig` configuration class was not included in the test because it needs to run in a Servlet context, which is not provided.

4.1. Using Spring Boot

Spring Boot provides several annotations to set up the Spring *ApplicationContext* for our tests in a more intuitive way.

We can load only a particular slice of the application configuration, or we can simulate the whole context startup process.

For instance, we can use the `@SpringBootTest` annotation if we want the entire context to be created without starting the server.

With that in place, we can then add the `@AutoConfigureMockMvc` to inject a *MockMvc* instance and send HTTP requests:

```

1.  @RunWith(SpringRunner.class)
2.  @SpringBootTest
3.  @AutoConfigureMockMvc
4.  public class FooControllerAppIntegrationTest {
5.
6.      @Autowired
7.      private MockMvc mockMvc;
8.
9.      @Test
10.     public void whenTestApp_thenEmptyResponse() throws Exception {
11.         this.mockMvc.perform(get("/foos")
12.             .andExpect(status().isOk())
13.             .andExpect(...));
14.     }
15.
16. }

```

To avoid creating the whole context and test only our MVC Controllers, we can use *@WebMvcTest*:

```

1.  @RunWith(SpringRunner.class)
2.  @WebMvcTest(FooController.class)
3.  public class FooControllerWebLayerIntegrationTest {
4.
5.      @Autowired
6.      private MockMvc mockMvc;
7.
8.      @MockBean
9.      private IFooService service;
10.
11.      @Test()
12.     public void whenTestMvcController_thenRetrieveExpectedResult()
13.     throws Exception {
14.         // ...
15.
16.         this.mockMvc.perform(get("/foos")
17.             .andExpect(...));
18.     }
19. }

```

We can find detailed information on this subject on '[Testing in Spring Boot](#)'.



The **@RestController** is the central artifact in the entire Web Tier of the RESTful API. For the purpose of this post, the controller is modeling a simple REST resource – Foo:

```
1.  @RestController
2.  @RequestMapping("/foos")
3.  class FooController {
4.
5.      @Autowired
6.      private IFooService service;
7.
8.      @GetMapping
9.      public List<Foo> findAll() {
10.         return service.findAll();
11.     }
12.
13.     @GetMapping(value =("/{id}")
14.     public Foo findById(@PathVariable("id") Long id) {
15.         return RestPreconditions.checkFound(service.findById(id));
16.     }
17.
18.     @PostMapping
19.     @ResponseStatus(HttpStatus.CREATED)
20.     public Long create(@RequestBody Foo resource) {
21.         Preconditions.checkNotNull(resource);
22.         return service.create(resource);
23.     }
24.
25.     @PutMapping(value =("/{id}")
26.     @ResponseStatus(HttpStatus.OK)
27.     public void update(@PathVariable("id") Long id, @RequestBody
28. Foo resource) {
29.         Preconditions.checkNotNull(resource);
30.         RestPreconditions.checkNotNull(service.getById(resource.
31. getId()));
32.         service.update(resource);
33.     }
34.
35.     @DeleteMapping(value =("/{id}")
36.     @ResponseStatus(HttpStatus.OK)
37.     public void delete(@PathVariable("id") Long id) {
38.         service.deleteById(id);
39.     }
40.
41. }
```

You may have noticed I'm using a straightforward, Guava style *RestPreconditions* utility:

```
1. public static <T> T checkFound(T resource) {  
2.     if (resource == null) {  
3.         throw new MyResourceNotFoundException();  
4.     }  
5.     return resource;  
6. }  
7. }
```

The Controller implementation is non-public – this is because it doesn't need to be.

Usually, the controller is the last in the chain of dependencies. It receives HTTP requests from the Spring front controller (the *DispatcherServlet*) and simply delegates them forward to a service layer. If there's no use case where the controller has to be injected or manipulated through a direct reference, then I prefer not to declare it as public.

The request mappings are straightforward. **As with any controller, the actual *value* of the mapping, as well as the HTTP method, determine the target method for the request.**

@RequestBody will bind the parameters of the method to the body of the HTTP request, whereas *@ResponseBody* does the same for the response and return type.

The *@RestController* is a shorthand to include both the *@ResponseBody* and the *@Controller* annotations in our class.

They also ensure that the resource will be marshalled and unmarshalled using the correct HTTP converter. Content negotiation will take place to choose which one of the active converters will be used, based mostly on the *Acceptheader*, although other HTTP headers may be used to determine the representation as well.

6. Mapping the HTTP Response Codes



The status codes of the HTTP response are one of the most important parts of the REST service, and the subject can quickly become very complicated. Getting these right can be what makes or breaks the service.

6.1. Unmapped Requests

If Spring MVC receives a request which doesn't have a mapping, it considers the request not to be allowed and returns a 405 METHOD NOT ALLOWED back to the client.

It's also a good practice to include the Allow HTTP header when returning a 405 to the client, to specify which operations are allowed. This is the standard behavior of Spring MVC and doesn't require any additional configuration.

6.2. Valid Mapped Requests

For any request that does have a mapping, Spring MVC considers the request valid and responds with 200 OK if no other status code is specified otherwise.

It's because of this that the controller declares different *@ResponseStatus* for the *create*, *update* and *delete* actions but not for *get*, which should indeed return the default 200 OK.

6.3. Client Error

In the case of a client error, custom exceptions are defined and mapped to the appropriate error codes.

Simply throwing these exceptions from any of the layers of the web tier will ensure Spring maps the corresponding status code on the HTTP response:

```
1. @ResponseStatus(HttpStatus.BAD_REQUEST)
2. public class BadRequestException extends RuntimeException {
3.     //
4. }
5. @ResponseStatus(HttpStatus.NOT_FOUND)
6. public class ResourceNotFoundException extends RuntimeException {
7.     //
8. }
```

These exceptions are part of the REST API and, as such, should only be used in the appropriate layers corresponding to REST; if for instance, a DAO/DAL layer exists, it should not use the exceptions directly.

Note also that these are not checked exceptions but runtime exceptions – in line with Spring practices and idioms.

6.4. Using *@ExceptionHandler*

Another option to map custom exceptions on specific status codes is to use the *@ExceptionHandler* annotation in the controller. The problem with that approach is that the annotation only applies to the controller in which it's defined. This means that we need to declare it in each controller individually.

Of course, there are more [ways to handle errors](#) in both Spring and Spring Boot that offer more flexibility.



In addition to the *spring-webmvc* dependency [required for the standard web application](#), we'll need to set up content marshalling and unmarshalling for the REST API:

```
1. <dependencies>
2.   <dependency>
3.     <groupId>com.fasterxml.jackson.core</groupId>
4.     <artifactId>jackson-databind</artifactId>
5.     <version>2.9.8</version>
6.   </dependency>
7.   <dependency>
8.     <groupId>javax.xml.bind</groupId>
9.     <artifactId>jaxb-api</artifactId>
10.    <version>2.3.1</version>
11.    <scope>runtime</scope>
12.  </dependency>
13. </dependencies>
```

These are the libraries used to convert the representation of the REST resource to either JSON or XML.

7.1. Using Spring Boot

If we want to retrieve JSON-formatted resources, Spring Boot provides support for different libraries, namely Jackson, Gson and JSON-B.

Auto-configuration is carried out by just including any of the mapping libraries in the classpath.

Usually, if we're developing a web application, **we'll just add the *spring-boot-starter-web* dependency and rely on it to include all the necessary artifacts to our project:**

```
1. <dependency>
2.   <groupId>org.springframework.boot</groupId>
3.   <artifactId>spring-boot-starter-web</artifactId>
4.   <version>2.1.2.RELEASE</version>
5. </dependency>
```

Spring Boot uses Jackson by default.

If we want to serialize our resources in an XML format, we'll have to add the Jackson XML extension (*jackson-dataformat-xml*) to our dependencies, or fallback to the JAXB implementation (provided by default in the JDK) by using the *@XmlRootElement* annotation on our resource.



This chapter illustrated how to implement and configure a REST Service using Spring and Java-based configuration.

All the code of this chapter is available [over on Github](#).



3: Http Message Converters with the Spring Framework



This chapter describes **how to Configure *HttpMessageConverters* in Spring**.

Simply put, we can use message converters to marshall and unmarshall Java Objects to and from JSON, XML, etc – over HTTP.



2.1. Enable Web MVC

To start with, the Web Application needs to be **configured with Spring MVC support**. A convenient and very customizable way to do this is to use the `@EnableWebMvc` annotation:

```
1. @EnableWebMvc
2. @Configuration
3. @ComponentScan({ "com.baeldung.web" })
4. public class WebConfig implements WebMvcConfigurer {
5.     ...
6. }
```

Note that this class implements *WebMvcConfigurer* – which will allow us to change the default list of Http Converters with our own.

2.2. The Default Message Converters

By default, the following *HttpMessageConverters* instances are pre-enabled:

- *ByteArrayHttpMessageConverter* – converts byte arrays
- *StringHttpMessageConverter* – converts Strings
- *ResourceHttpMessageConverter* – converts *org.springframework.core.io.Resource* for any type of octet stream
- *SourceHttpMessageConverter* – converts *javax.xml.transform.Source*
- *FormHttpMessageConverter* – converts form data to/from a *MultiValueMap<String, String>*.
- *Jaxb2RootElementHttpMessageConverter* – converts Java objects to/from XML (added only if JAXB2 is present on the classpath)
- *MappingJackson2HttpMessageConverter* – converts JSON (added only if Jackson 2 is present on the classpath)
- *MappingJacksonHttpMessageConverter* – converts JSON (added only if Jackson is present on the classpath)
- *AtomFeedHttpMessageConverter* – converts Atom feeds (added only if Rome is present on the classpath)
- *RssChannelHttpMessageConverter* – converts RSS feeds (added only if Rome is present on the classpath)



3.1. High-Level Content Negotiation

Each *HttpMessageConverter* implementation has one or several associated MIME Types.

When receiving a new request, **Spring will use the "Accept" header to determine the media type that it needs to respond with.**

It will then try to find a registered converter that's capable of handling that specific media type. Finally, it will use this to convert the entity and send back the response.

The process is similar for receiving a request which contains JSON information. The framework will use the **"Content-Type" header to determine the media type of the request body.**

It will then search for a *HttpMessageConverter* that can convert the body sent by the client to a Java Object.

Let's clarify this with a quick example:

- the Client sends a GET request to `/foos` with the Accept header set to `application/json` – to get all Foo resources as JSON
- the *Foo* Spring Controller is hit and returns the corresponding *Foo* Java entities
- Spring then uses one of the Jackson message converters to marshall the entities to JSON

Let's now look at the specifics of how this works – and how we can leverage the `@ResponseBody` and `@RequestBody` annotations.

3.2. @ResponseBody

`@ResponseBody` on a Controller method indicates to Spring that **the return value of the method is serialized directly to the body of the HTTP Response**. As discussed above, the "Accept" header specified by the Client will be used to choose the appropriate Http Converter to marshall the entity.

Let's look at a simple example:

```
1. @GetMapping("/{id}")
2. public @ResponseBody Foo findById(@PathVariable long id) {
3.     return fooService.findById(id);
4. }
```

Now, the client will specify the "Accept" header to *application/json* in the request – example *curl* command:

```
curl --header "Accept: application/json"
http://localhost:8080/spring-boot-rest/foos/1
```

The Foo class:

```
1. public class Foo {
2.     private long id;
3.     private String name;
4. }
```

And the Http Response Body:

```
1. {
2.     "id": 1,
3.     "name": "Paul",
4. }
```

3.3. @RequestBody

We can use the `@RequestBody` annotation on the argument of a Controller method to indicate **that the body of the HTTP Request is deserialized to that particular Java entity**. To determine the appropriate converter, Spring will use the "Content-Type" header from the client request.

Let's look at an example:

```
1. @PutMapping("/{id}")
2. public @ResponseBody void update(@RequestBody Foo foo, @PathVariable
3. String id) {
4.     fooService.update(foo);
5. }
```

Next, let's consume this with a JSON object – we're specifying "Content-Type" to be *application/json*:

```
curl -i -X PUT -H "Content-Type: application/json"
-d '{"id":"83","name":"klik"}' http://localhost:8080/spring-boot-rest/
foos/1
```

We get back a 200 OK – a successful response:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Length: 0
```

4. Custom Converters Configuration



We can also **customize the message converters by implementing the *WebMvcConfigurer* interface** and overriding the *configureMessageConverters* method:

```
1.  @EnableWebMvc
2.  @Configuration
3.  @ComponentScan({ "com.baeldung.web" })
4.  public class WebConfig implements WebMvcConfigurer {
5.
6.      @Override
7.      public void configureMessageConverters(
8.          List<HttpMessageConverter<?>> converters) {
9.
10.         messageConverters.add(createXmlHttpMessageConverter());
11.         messageConverters.add(new
12. MappingJackson2HttpMessageConverter());
13.     }
14.     private HttpMessageConverter<Object> createXmlHttpMessageConverter()
15.     {
16.         MarshallingHttpMessageConverter xmlConverter =
17.             new MarshallingHttpMessageConverter();
18.
19.         XStreamMarshaller xstreamMarshaller = new XStreamMarshaller();
20.         xmlConverter.setMarshaller(xstreamMarshaller);
21.         xmlConverter.setUnmarshaller(xstreamMarshaller);
22.
23.         return xmlConverter;
24.     }
25. }
```

And here is the corresponding XML configuration:

```
<context:component-scan base-package="org.baeldung.web" />
<mvc:annotation-driven>
    <mvc:message-converters>
        <bean class="org.springframework.http.converter.json.
MappingJackson2HttpMessageConverter" />
        <bean class="org.springframework.http.converter.xml.
MarshallingHttpMessageConverter">
            <property name="marshaller" ref="xstreamMarshaller" />
            <property name="unmarshaller" ref="xstreamMarshaller" />
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>

<bean id="xstreamMarshaller" class="org.springframework.xml.xstream.
XStreamMarshaller" />
```

In this example, we're creating a new converter – the *MarshallingHttpMessageConverter* – and using the Spring XStream support to configure it. This allows a great deal of flexibility since **we're working with the low-level APIs of the underlying marshalling framework** – in this case XStream – and we can configure that however we want.

Note that this example requires adding the XStream library to the classpath.

Also be aware that by extending this support class, **we're losing the default message converters which were previously pre-registered**.

We can of course now do the same for Jackson – by defining our own *MappingJackson2HttpMessageConverter*. We can now set a custom *ObjectMapper* on this converter and have it configured as we need to.

In this case, XStream was the selected marshaller/unmarshaller implementation, but [others](#) like CastorMarshaller can be used as well.

At this point – with XML enabled on the back end – we can consume the API with XML Representations:

```
curl --header "Accept: application/xml"
  http://localhost:8080/spring-boot-rest/foos/1
```

4.1. Spring Boot Support

If we're using Spring Boot we can avoid implementing the *WebMvcConfigurer* and adding all the Message Converters manually as we did above.

We can just define different *HttpMessageConverter* beans in the context, and Spring Boot will add them automatically to the autoconfiguration that it creates:

```
1. @Bean
2. public HttpMessageConverter<Object> createXmlHttpMessageConverter() {
3.     MarshallingHttpMessageConverter xmlConverter = new
4.     MarshallingHttpMessageConverter();
5.
6.     // ...
7.
8.     return xmlConverter;
9. }
```



As well as with the server side, Http Message Conversion can be configured in the client side on the Spring *RestTemplate*.

We're going to configure the template with the "Accept" and "Content-Type" headers when appropriate. Then we'll try to consume the REST API with full marshalling and unmarshalling of the *Foo* Resource – both with JSON and with XML.

5.1. Retrieving the Resource with no *Accept* Header

```
1. @Bean
2. public HttpMessageConverter<Object> createXmlHttpMessageConverter() {
3.     MarshallingHttpMessageConverter xmlConverter = new
4.     MarshallingHttpMessageConverter();
5.
6.     // ...
7.
8.     return xmlConverter;
9. }
```

5.2. Retrieving a Resource with *application/xml* Accept Header

Let's now explicitly retrieve the Resource as an XML Representation. We're going to define a set of Converters and set these on the *RestTemplate*.

Because we're consuming XML, we're going to use the same XStream marshaller as before:

```
1. @Test
2. public void givenConsumingXml_whenReadingTheFoo_thenCorrect() {
3.     String URI = BASE_URI + "foos/{id}";
4.     RestTemplate restTemplate = new RestTemplate();
5.     restTemplate.setMessageConverters(getMessageConverters());
6.
7.     HttpHeaders headers = new HttpHeaders();
8.     headers.setAccept(Arrays.asList(MediaType.APPLICATION_XML));
9.     HttpEntity<String> entity = new HttpEntity<String>(headers);
10.
11.     ResponseEntity<Foo> response =
12.         restTemplate.exchange(URI, HttpMethod.GET, entity, Foo.class,
13.         "1");
14.     Foo resource = response.getBody();
15.
16.     assertThat(resource, notNullValue());
17. }
```

```

1. private List<HttpMessageConverter<?>> getMessageConverters() {
2.     XStreamMarshaller marshaller = new XStreamMarshaller();
3.     MarshallingHttpMessageConverter marshallingConverter =
4.         new MarshallingHttpMessageConverter(marshaller);
5.
6.     List<HttpMessageConverter<?>> converters =
7.         ArrayList<HttpMessageConverter<?>>();
8.     converters.add(marshallingConverter);
9.     return converters;
10. }

```

5.3. Retrieving a Resource with *application/json* Accept Header

Similarly, let's now consume the REST API by asking for JSON:

```

1. @Test
2. public void givenConsumingJson_whenReadingTheFoo_thenCorrect() {
3.     String URI = BASE_URI + "foos/{id}";
4.
5.     RestTemplate restTemplate = new RestTemplate();
6.     restTemplate.setMessageConverters(getMessageConverters());
7.
8.     HttpHeaders headers = new HttpHeaders();
9.     headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
10.    HttpEntity<String> entity = new HttpEntity<String>(headers);
11.
12.    ResponseEntity<Foo> response =
13.        restTemplate.exchange(URI, HttpMethod.GET, entity, Foo.class,
14.            "1");
15.    Foo resource = response.getBody();
16.
17.    assertThat(resource, notNullValue());
18. }
19. private List<HttpMessageConverter<?>> getMessageConverters() {
20.     List<HttpMessageConverter<?>> converters =
21.         new ArrayList<HttpMessageConverter<?>>();
22.     converters.add(new MappingJackson2HttpMessageConverter());
23.     return converters;
24. }

```

5.4. Update a Resource with XML *Content-Type*

Finally, let's also send JSON data to the REST API and specify the media type of that data via the *Content-Type* header:

```
1.  @Test
2.  public void givenConsumingXml_whenWritingTheFoo_thenCorrect() {
3.      String URI = BASE_URI + "foos/{id}";
4.      RestTemplate restTemplate = new RestTemplate();
5.      restTemplate.setMessageConverters(getMessageConverters());
6.
7.      Foo resource = new Foo(4, "json");
8.      HttpHeaders headers = new HttpHeaders();
9.      headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
10.     headers.setContentType(MediaType.APPLICATION_XML);
11.     HttpEntity<Foo> entity = new HttpEntity<Foo>(resource, headers);
12.
13.     ResponseEntity<Foo> response = restTemplate.exchange(
14.         URI, HttpMethod.PUT, entity, Foo.class, resource.getId());
15.     Foo fooResponse = response.getBody();
16.
17.     Assert.assertEquals(resource.getId(), fooResponse.getId());
18. }
```

What's interesting here is that we're able to mix the media types – **we're sending XML data but we're waiting for JSON data back from the server**. This shows just how powerful the Spring conversion mechanism really is.



In this chapter, we looked at how Spring MVC allows us to specify and fully customize Http Message Converters to **automatically marshal/unmarshal Java Entities to and from XML or JSON**. This is, of course, a simplistic definition, and there is so much more that the message conversion mechanism can do – as we can see from the last test example.

We have also looked at how to leverage the same powerful mechanism with the *RestTemplate* client – leading to a fully type-safe way of consuming the API.

As always, the code presented in this chapter is available [over on Github](#).



4: Spring's RequestBody and ResponseBody Annotations



In this quick chapter, we provide a concise overview of the Spring `@RequestBody` and `@ResponseBody` annotations.

2. *@RequestBody*



Simply put, the *@RequestBody* annotation maps the *HttpRequest* body to a transfer or domain object, enabling automatic deserialization of the inbound *HttpRequest* body onto a Java object.

First, let's have a look at a Spring controller method:

```
1. @PostMapping("/request")
2. public ResponseEntity postController(
3.     @RequestBody LoginForm loginForm) {
4.
5.     exampleService.fakeAuthenticate(loginForm);
6.     return ResponseEntity.ok(HttpStatus.OK);
7. }
```

Spring automatically deserializes the JSON into a Java type assuming an appropriate one is specified. By default, the type we annotate with the *@RequestBody* annotation must correspond to the JSON sent from our client-side controller:

```
1. public class LoginForm {
2.     private String username;
3.     private String password;
4.     // ...
5. }
```

Here, the object we use to represent the *HttpRequest* body maps to our *LoginForm* object.

Let's test this using CURL:

```
1. curl -i \
2. -H "Accept: application/json" \
3. -H "Content-Type:application/json" \
4. -X POST --data
5.     '{"username": "johnny", "password": "password"}' "https://
6. localhost:8080/.../request"
```

This is all that is needed for a Spring REST API and an Angular client using the *@RequestBody* annotation!

3. @ResponseBody



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

Suppose we have a custom Response object:

```
1. public class ResponseTransfer {
2.     private String text;
3.
4.     // standard getters/setters
5. }
```

Next, the associated controller can be implemented:

```
1. @Controller
2. @RequestMapping("/post")
3. public class ExamplePostController {
4.
5.     @Autowired
6.     ExampleService exampleService;
7.
8.     @PostMapping("/response")
9.     @ResponseBody
10.    public ResponseTransfer postResponseController(
11.        @RequestBody LoginForm loginForm) {
12.        return new ResponseTransfer("Thanks For Posting!!!");
13.    }
14. }
```

In the developer console of our browser or using a tool like Postman, we can see the following response:

```
1. {"text":"Thanks For Posting!!!"}
```

Remember, we don't need to annotate the *@RestController-annotated* controllers with the *@ResponseBody* annotation since it's done by default here.

4. Conclusion



We've built a simple Angular client for the Spring app that demonstrates how to use the `@RestController` and `@ResponseBody` annotations.

As always code samples are available [over on GitHub](#).



5: Entity To DTO Conversion for a Spring REST API



Each In this tutorial, we'll handle the conversions that need to happen **between the internal entities of a Spring application and the external DTOs** (Data Transfer Objects) that are published back to the client.

2. Model Mapper



Let's start by introducing the main library that we're going to use to perform this entity-DTO conversion – [ModelMapper](#).

We will need this dependency in the *pom.xml*:

```
1. <dependency>
2.     <groupId>org.modelmapper</groupId>
3.     <artifactId>modelmapper</artifactId>
4.     <version>2.3.2</version>
5. </dependency>
```

We'll then define the *ModelMapper* bean in our Spring configuration:

```
1. @Bean
2. public ModelMapper modelMapper() {
3.     return new ModelMapper();
4. }
```


3. The DTO



Next, let's introduce the DTO side of this two-sided problem – *Post DTO*:

```
1. public class PostDto {
2.     private static final SimpleDateFormat dateFormat
3.         = new SimpleDateFormat("yyyy-MM-dd HH:mm");
4.
5.     private Long id;
6.
7.     private String title;
8.
9.     private String url;
10.
11.    private String date;
12.
13.    private UserDto user;
14.
15.    public Date getSubmissionDateConverted(String timezone) throws
16.        ParseException {
17.        dateFormat.setTimeZone(TimeZone.getTimeZone(timezone));
18.        return dateFormat.parse(this.date);
19.    }
20.
21.    public void setSubmissionDate(Date date, String timezone) {
22.        dateFormat.setTimeZone(TimeZone.getTimeZone(timezone));
23.        this.date = dateFormat.format(date);
24.    }
25.
26.    // standard getters and setters
27. }
```

Note that the 2 custom date related methods handle the date conversion back and forth between the client and the server:

- *getSubmissionDateConverted()* method converts Date String into a date in server's timezone to use it in persisting Post entity
- *setSubmissionDate()* method is to set DTO's date to Post's Date in current user timezone.

4. The Service Layer



Let's now look at a service level operation – which will obviously work with the Entity (not the DTO):

```
1. public List<Post> getPostsList(  
2.     int page, int size, String sortDir, String sort) {  
3.  
4.     PageRequest pageReq  
5.         = PageRequest.of(page, size, Sort.Direction.fromString(sortDir),  
6.     sort);  
7.  
8.     Page<Post> posts = postRepository  
9.         .findByUser(userService.getCurrentUser(), pageReq);  
10.    return posts.getContent();  
11. }
```

We're going to have a look at the layer above service next – the controller layer. This is where the conversion will actually happen as well.

5. The Controller Layer



Let's now have a look at a standard controller implementation, exposing the simple REST API for the Post resource. We're going to show here a few simple CRUD operations: create, update, get one and get all. And given the operations are pretty straightforward, **we are especially interested in the Entity-DTO conversion aspects**:

```
1. @Controller
2. class PostRestController {
3.
4.     @Autowired
5.     private IPostService postService;
6.
7.     @Autowired
8.     private IUserService userService;
9.
10.    @Autowired
11.    private ModelMapper modelMapper;
12.
13.    @RequestMapping(method = RequestMethod.GET)
14.    @ResponseBody
15.    public List<PostDto> getPosts(...) {
16.        //...
17.        List<Post> posts = postService.getPostsList(page, size, sortDir,
18. sort);
19.        return posts.stream()
20.            .map(post -> convertToDto(post))
21.            .collect(Collectors.toList());
22.    }
23.
24.    @RequestMapping(method = RequestMethod.POST)
25.    @ResponseStatus(HttpStatus.CREATED)
26.    @ResponseBody
27.    public PostDto createPost(@RequestBody PostDto postDto) {
28.        Post post = convertToEntity(postDto);
29.        Post postCreated = postService.createPost(post);
30.        return convertToDto(postCreated);
31.    }
32.
33.    @RequestMapping(value =("/{id}", method = RequestMethod.GET)
34.    @ResponseBody
35.    public PostDto getPost(@PathVariable("id") Long id) {
36.        return convertToDto(postService.getPostById(id));
37.    }
38.
39.    @RequestMapping(value =("/{id}", method = RequestMethod.PUT)
40.    @ResponseStatus(HttpStatus.OK)
41.    public void updatePost(@RequestBody PostDto postDto) {
42.        Post post = convertToEntity(postDto);
43.        postService.updatePost(post);
44.    }
45. }
```

And here is **our conversion from *Post* entity to *PostDto***:

```
1. private PostDto convertToDto(Post post) {  
2.     PostDto postDto = modelMapper.map(post, PostDto.class);  
3.     postDto.setSubmissionDate(post.getSubmissionDate(),  
4.         userService.getCurrentUser().getPreference().getTimezone());  
5.     return postDto;  
6. }
```

And here is the conversion **from DTO to an entity**:

```
1. private Post convertToEntity(PostDto postDto) throws ParseException {  
2.     Post post = modelMapper.map(postDto, Post.class);  
3.     post.setSubmissionDate(postDto.getSubmissionDateConverted(  
4.         userService.getCurrentUser().getPreference().getTimezone()));  
5.  
6.     if (postDto.getId() != null) {  
7.         Post oldPost = postService.getPostById(postDto.getId());  
8.         post.setRedditID(oldPost.getRedditID());  
9.         post.setSent(oldPost.isSent());  
10.    }  
11.    return post;  
12. }
```

So, as you can see, with the help of the model mapper, **the conversion logic is quick and simple** – we're using the *map* API of the mapper and getting the data converted without writing a single line of conversion logic.



Finally, let's do a very simple test to make sure the conversions between the entity and the DTO work well:

```
1. public class PostDtoUnitTest {
2.
3.     private ModelMapper modelMapper = new ModelMapper();
4.
5.     @Test
6.     public void whenConvertPostEntityToPostDto_thenCorrect() {
7.         Post post = new Post();
8.         post.setId(Long.valueOf(1));
9.         post.setTitle(randomAlphabetic(6));
10.        post.setUrl("www.test.com");
11.
12.        PostDto postDto = modelMapper.map(post, PostDto.class);
13.        assertEquals(post.getId(), postDto.getId());
14.        assertEquals(post.getTitle(), postDto.getTitle());
15.        assertEquals(post.getUrl(), postDto.getUrl());
16.    }
17.
18.    @Test
19.    public void whenConvertPostDtoToPostEntity_thenCorrect() {
20.        PostDto postDto = new PostDto();
21.        postDto.setId(Long.valueOf(1));
22.        postDto.setTitle(randomAlphabetic(6));
23.        postDto.setUrl("www.test.com");
24.
25.        Post post = modelMapper.map(postDto, Post.class);
26.        assertEquals(postDto.getId(), post.getId());
27.        assertEquals(postDto.getTitle(), post.getTitle());
28.        assertEquals(postDto.getUrl(), post.getUrl());
29.    }
30. }
```



This was a chapter on simplifying the conversion from Entity to DTO and from DTO to Entity in a Spring REST API, by using the model mapper library instead of writing these conversions by hand.

The full source code for the examples is available in the [GitHub project](#).



6: Error Handling for REST with Spring



This chapter will illustrate **how to implement Exception Handling with Spring for a REST API**. We'll also get a bit of historical overview and see which new options the different versions introduced.

Before Spring 3.2, the two main approaches to handling exceptions in a Spring MVC application were: *HandlerExceptionResolver* or the *@ExceptionHandler* annotation. Both of these have some clear downsides.

Since 3.2 we've had the *@ControllerAdvice* annotation to address the limitations of the previous two solutions and to promote a unified exception handling throughout a whole application.

Now, **Spring 5** introduces the *ResponseStatusException* class: A fast way for basic error handling in our REST APIs.

All of these do have one thing in common – they deal with the **separation of concerns** very well. The app can throw exception normally to indicate a failure of some kind – exceptions which will then be handled separately.

Finally, we'll see what Spring Boot brings to the table, and how we can configure it to suit our

2. Solution 1 – The Controller level *@ExceptionHandler*

The first solution works at the *@Controller level* – we will define a method to handle exceptions, and annotate that with *@ExceptionHandler*:

```
1. public class FooController{
2.
3.     //...
4.     @ExceptionHandler({ CustomException1.class, CustomException2.class
5. })
6.     public void handleException() {
7.         //
8.     }
9. }
```

This approach has a major drawback – **the *@ExceptionHandler* annotated method is only active for that particular Controller**, not globally for the entire application. Of course, adding this to every controller makes it not well suited for a general exception handling mechanism.

We can work around this limitation by having **all Controllers extend a Base Controller class** – however, this can be a problem for applications where, for whatever reason, this isn't possible. For example, the Controllers may already extend from another base class which may be in another jar or not directly modifiable, or may themselves not be directly modifiable.

Next, we'll look at another way to solve the exception handling problem – one that is global and doesn't include any changes to existing artifacts such as Controllers.

3. Solution 2 – *The HandlerExceptionResolver*



The second solution is to define an *HandlerExceptionResolver* – this will resolve any exception thrown by the application. It will also allow us to implement a **uniform exception handling mechanism** in our REST API.

Before going for a custom resolver, let's go over the existing implementations.

3.1. *ExceptionHandlerExceptionResolver*

This resolver was introduced in Spring 3.1 and is enabled by default in the *DispatcherServlet*. This is actually the core component of how the *@ExceptionHandler* mechanism presented earlier works.

3.2. *DefaultHandlerExceptionResolver*

This resolver was introduced in Spring 3.0, and it's enabled by default in the *DispatcherServlet*. It's used to resolve standard Spring exceptions to their corresponding HTTP Status Codes, namely Client error – 4xx and Server error – 5xx status codes. [Here's the full list](#) of the Spring Exceptions it handles, and how they map to status codes.

While it does set the Status Code of the Response properly, one **limitation is that it doesn't set anything to the body of the Response**. And for a REST API – the Status Code is really not enough information to present to the Client – the response has to have a body as well, to allow the application to give additional information about the failure.

This can be solved by configuring view resolution and rendering error content through *ModelAndView*, but the solution is clearly not optimal. That's why Spring 3.2 introduced a better option that we'll discuss in a later section.

3.3. *ResponseStatusExceptionHandlerResolver*

This resolver was also introduced in Spring 3.0 and is enabled by default in the *DispatcherServlet*. Its main responsibility is to use the *@ResponseStatus* annotation available on custom exceptions and to map these exceptions to HTTP status codes.

Such a custom exception may look like:

```

1.  @ResponseStatus(value = HttpStatus.NOT_FOUND)
2.  public class ResourceNotFoundException extends RuntimeException {
3.      public ResourceNotFoundException() {
4.          super();
5.      }
6.      public ResourceNotFoundException(String message, Throwable cause) {
7.          super(message, cause);
8.      }
9.      public ResourceNotFoundException(String message) {
10.         super(message);
11.     }
12.     public ResourceNotFoundException(Throwable cause) {
13.         super(cause);
14.     }
15. }

```

Same as the *DefaultHandlerExceptionResolver*, this resolver is limited in the way it deals with the body of the response – it does map the Status Code on the response, but the body is still *null*.

3.4. Custom HandlerExceptionResolver

The combination of *DefaultHandlerExceptionResolver* and *ResponseStatusExceptionHandlerResolver* goes a long way towards providing a good error handling mechanism for a Spring RESTful Service. The downside is – as mentioned before – **no control over the body of the response**.

Ideally, we'd like to be able to output either JSON or XML, depending on what format the client has asked for (via the *Accept* header).

This alone justifies creating a **new, custom exception resolver**:

```

1.  @Component
2.  public class RestResponseStatusExceptionHandlerResolver extends
3.  AbstractHandlerExceptionHandlerResolver {
4.
5.      @Override
6.      protected ModelAndView doResolveException
7.      (HttpServletRequest request, HttpServletResponse response, Object
8.  handler, Exception ex) {
9.          try {
10.             if (ex instanceof IllegalArgumentException) {
11.                 return handleIllegalArgument((IllegalArgumentException)
12.  ex, response, handler);
13.             }
14.             ...
15.         } catch (ExceptionHandlerException) {
16.             logger.warn("Handling of [" + ex.getClass().getName() + "]
17.                 resulted in Exception", handlerException);
18.         }
19.         return null;
20.     }
21.
22.     private ModelAndView handleIllegalArgument
23.     (IllegalArgumentException ex, HttpServletResponse response) throws
24.     IOException {
25.         response.sendError(HttpServletResponse.SC_CONFLICT);
26.         String accept = request.getHeader(HttpHeaders.ACCEPT);
27.         ...
28.         return new ModelAndView();
29.     }
30. }

```

One detail to notice here is that we have access to the *request* itself, so we can consider the value of the *Accept* header sent by the client.

For example, if the client asks for *application/json* then, in the case of an error condition, we'd want to make sure we return a response body encoded with *application/json*.

The other important implementation detail is that **we return a *ModelAndView* – this is the body of the response** and it will allow us to set whatever is necessary on it.

This approach is a consistent and easily configurable mechanism for the error handling of a Spring REST Service. It does, however, have limitations: it's interacting with the low-level *HttpServletResponse* and it fits into the old MVC model which uses *ModelAndView* – so there's still room for improvement.

4. Solution 3 – @ControllerAdvice



Spring 3.2 brings support for a global **@ExceptionHandler** with the **@ControllerAdvice** annotation. This enables a mechanism that breaks away from the older MVC model and makes use of *ResponseEntity* along with the type safety and flexibility of **@ExceptionHandler**:

```
1.  @ControllerAdvice
2.  public class RestResponseEntityExceptionHandler
3.      extends ResponseEntityExceptionHandler {
4.
5.      @ExceptionHandler(value
6.          = { IllegalArgumentException.class, IllegalStateException.class })
7.      protected ResponseEntity<Object> handleConflict(
8.          RuntimeException ex, WebRequest request) {
9.          String bodyOfResponse = "This should be application specific";
10.         return handleExceptionInternal(ex, bodyOfResponse,
11.             new HttpHeaders(), HttpStatus.CONFLICT, request);
12.     }
13. }
```

The **@ControllerAdvice** annotation allows us to **consolidate our multiple, scattered @ExceptionHandler**s from before into a single, global error handling component.

The actual mechanism is extremely simple but also very flexible. It gives us:

- Full control over the body of the response as well as the status code
- Mapping of several exceptions to the same method, to be handled together, and
- It makes good use of the newer RESTful *ResponseEntity* response

One thing to keep in mind here is to **match the exceptions declared with @ExceptionHandler with the exception used as the argument of the method**. If these don't match, the compiler will not complain – no reason it should, and Spring will not complain either.

However, when the exception is actually thrown at runtime, **the exception resolving mechanism will fail with:**

```
java.lang.IllegalStateException: No suitable resolver for argument [0]
[type=...]
HandlerMethod details: ...
```

5. Solution 4 – *ResponseStatusException*



Spring 5 introduced the `ResponseStatusException` class. We can create an instance of it providing an `HttpStatus` and optionally a reason and a cause:

```
1. @GetMapping(value =("/{id}")
2. public Foo findById(@PathVariable("id") Long id, HttpServletResponse
3. response) {
4.     try {
5.         Foo resourceById = RestPreconditions.checkFound(service.
6. findOne(id));
7.
8.         eventPublisher.publishEvent(new
9. SingleResourceRetrievedEvent(this, response));
10.        return resourceById;
11.    }
12.    catch (MyResourceNotFoundException exc) {
13.        throw new ResponseStatusException(
14.            HttpStatus.NOT_FOUND, "Foo Not Found", exc);
15.    }
16. }
```

What are the benefits of using *ResponseStatusException*?

- Excellent for prototyping: We can implement a basic solution quite fast
- One type, multiple status codes: One exception type can lead to multiple different responses. **This reduces tight coupling compared to the `@ExceptionHandler`**
- We won't have to create as many custom exception classes
- **More control over exception handling** since the exceptions can be created programmatically

And what about the tradeoffs?

- There's no unified way of exception handling: It's more difficult to enforce some application-wide conventions, as opposed to `@ControllerAdvice` which provides a global approach
- Code duplication: We may find ourselves replicating code in multiple controllers

We should also note that it's possible to combine different approaches within one application.

For example, we can implement a `@ControllerAdvice` globally, but also *`ResponseStatusExceptions` locally*. However, we need to be careful: If the same exception can be handled in multiple ways, we may notice some surprising behavior. A possible convention is to handle one specific kind of exception always in one way.



The Access Denied occurs when an authenticated user tries to access resources that he doesn't have enough authorities to access.

6.1. REST and Method Level Security

Finally, let's see how to handle Access Denied exception thrown by method-level security annotations - `@PreAuthorize`, `@PostAuthorize`, and `@Secure`.

Of course, we'll use the global exception handling mechanism that we discussed earlier to handle the `AccessDeniedException` as well:

```
1.  @ControllerAdvice
2.  public class RestResponseEntityExceptionHandler
3.      extends ResponseEntityExceptionHandler {
4.
5.      @ExceptionHandler({ AccessDeniedException.class })
6.      public ResponseEntity<Object> handleAccessDeniedException(
7.          Exception ex, WebRequest request) {
8.          return new ResponseEntity<Object>(
9.              "Access denied message here", new HttpHeaders(), HttpStatus.
10. FORBIDDEN);
11.      }
12.
13.      ...
14.  }
```



Spring Boot provides an *ErrorController* implementation to handle errors in a sensible way.

In a nutshell, it serves a fallback error page for browsers (aka the Whitelabel Error Page), and a JSON response for RESTful, non HTML requests:

```
1. {
2.     "timestamp": "2019-01-17T16:12:45.977+0000",
3.     "status": 500,
4.     "error": "Internal Server Error",
5.     "message": "Error processing the request!",
6.     "path": "/my-endpoint-with-exceptions"
7. }
```

As usual, Spring Boot allows configuring these features with properties:

- *server.error.whitelabel.enabled*: can be used to disable the Whitelabel Error Page and rely on the servlet container to provide an HTML error message
- *server.error.include-stacktrace*: with an always value, it includes the stacktrace in both the HTML and the JSON default response

Apart from these properties, **we can provide our own view-resolver mapping for */error*, overriding the Whitelabel Page.**

We can also customize the attributes that we want to show in the response by including an *ErrorAttributes* bean in the context. We can extend the *DefaultErrorAttributes* class provided by Spring Boot to make things easier:

```
1. @Component
2. public class MyCustomErrorAttributes extends DefaultErrorAttributes {
3.
4.     @Override
5.     public Map<String, Object> getErrorAttributes(WebRequest webRequest,
6. boolean includeStackTrace) {
7.         Map<String, Object> errorAttributes = super.
8. getErrorAttributes(webRequest, includeStackTrace);
9.         errorAttributes.put("locale", webRequest.getLocale()
10.             .toString());
11.         errorAttributes.remove("error");
12.
13.         //...
14.
15.         return errorAttributes;
16.     }
17. }
```


If we want to go further and define (or override) how the application will handle errors for a particular content type, we can register an `ExceptionHandler` bean.

Again, we can make use of the default `ExceptionHandler` provided by Spring Boot to help us out.

For example, imagine we want to customize how our application handles errors triggered in XML endpoints. All we have to do is define a public method using the `@ExceptionHandler` and stating it produces `application/xml` media type:

```
1.  @ExceptionHandler
2.  public class MyExceptionHandler extends HandlerExceptionResolver {
3.
4.      public MyExceptionHandler(ErrorAttributes errorAttributes) {
5.          super(errorAttributes, new ErrorProperties());
6.      }
7.
8.      @ExceptionHandler(produces = MediaType.APPLICATION_XML_VALUE)
9.      public ResponseEntity<Map<String, Object>>
10. xmlError(HttpServletRequest request) {
11.
12.         // ...
13.
14.     }
15. }
```



This tutorial discussed several ways to implement an exception handling mechanism for a REST API in Spring, starting with the older mechanism and continuing with the Spring 3.2 support and into 4.x and 5.x.

As always, the code presented in this chapter is available [over on Github](#).



7: REST API Discoverability and HATEOAS



This chapter will focus on **Discoverability of the REST API, HATEOAS** and practical scenarios driven by tests.

2. Why Make the API Discoverable



Discoverability of an API is a topic that doesn't get enough well-deserved attention. As a consequence, very few APIs get it right. It's also something that, if done correctly, can make the API not only RESTful and usable but also elegant.

To understand discoverability, we need to understand the Hypermedia As The Engine Of Application State (HATEOAS) constraint. **This constraint of a REST API is about full discoverability of actions/transitions on a Resource from Hypermedia (Hypertext really), as the only driver of application state.**

If the interaction is to be driven by the API through the conversation itself, concretely via Hypertext, then there can be no documentation. That would coerce the client to make assumptions that are in fact outside of the context of the API.

In conclusion, the server should be descriptive enough to instruct the client how to use the API via Hypertext only. In the case of an HTTP conversation, we could achieve this through the Link header.

3. Discoverability Scenarios (Driven by Tests)



So what does it mean for a REST service to be discoverable?

Throughout this section, we'll test individual traits of discoverability using Junit, [rest-assured](#) and [Hamcrest](#). Since [the REST Service has been previously secured](#), each test first needs to [authenticate](#) before consuming the API.

3.1. Discover the Valid HTTP Methods

When a REST Service is consumed with an invalid HTTP method, the response should be a 405 METHOD NOT ALLOWED.

The API should also help the client discover the valid HTTP methods that are allowed for that particular Resource. **For this, we can use the *Allow* HTTP Header in the response:**

```
1.  @Test
2.  public void
3.      whenInvalidPOSTIsSentToValidURIOfResource_
4.      thenAllowHeaderListsTheAllowedActions(){
5.      // Given
6.      String uriOfExistingResource = restTemplate.createResource();
7.
8.      // When
9.      Response res = givenAuth().post(uriOfExistingResource);
10.
11.     // Then
12.     String allowHeader = res.getHeader(HttpHeaders.ALLOW);
13.     assertThat( allowHeader, AnyOf.anyOf(
14.         containsString("GET"), containsString("PUT"),
15.         containsString("DELETE") ) );
16. }
```

3.2. Discover the URI of Newly Created Resource

The operation of creating a new Resource should always include the URI of the newly created resource in the response. For this, we can use the *Location* HTTP Header.

Now, if the client does a GET on that URI, the resource should be available:

```
1.  @Test
2.  public void whenResourceIsCreated_
3.  thenUriOfTheNewlyCreatedResourceIsDiscoverable() {
4.      // When
5.      Foo newResource = new Foo(randomAlphabetic(6));
6.      Response createResp = givenAuth().contentType("application/json")
7.          .body(unpersistedResource).post(getFooURL());
8.      String uriOfNewResource= createResp.getHeader(HttpHeaders.LOCATION);
9.
10.     // Then
11.     Response response = givenAuth().header(HttpHeaders.ACCEPT,
12.         MediaType.APPLICATION_JSON_VALUE)
13.         .get(uriOfNewResource);
14.
15.     Foo resourceFromServer = response.body().as(Foo.class);
16.     assertThat(newResource, equalTo(resourceFromServer));
17. }
```

The test follows a simple scenario: **creating a new *Foo* resource, then using the HTTP response to discover the URI where the Resource is now available.** It also then does a GET on that URI to retrieve the resource and compares it to the original. This is to make sure that it was correctly saved.

3.3. Discover the URI to GET All Resources of That Type

When we GET any particular *Foo* resource, we should be able to discover what we can do next: we can list all the available *Foo* resources. Thus, the operation of retrieving a resource should always include in its response the URI where to get all the resources of that type.

For this, we can again make use of the Link header:

```
1.  @Test
2.  public void whenResourceIsRetrieved_
3.  thenUriToGetAllResourcesIsDiscoverable() {
4.      // Given
5.      String uriOfExistingResource = createAsUri();
6.
7.      // When
8.      Response getResponse = givenAuth().get(uriOfExistingResource);
9.
10.     // Then
11.     String uriToAllResources = HTTPLinkHeaderUtil
12.         .extractURIByRel(getResponse.getHeader("Link"), "collection");
13.
14.     Response getAllResponse = givenAuth().get(uriToAllResources);
15.     assertThat(getAllResponse.getStatusCode(), is(200));
16. }
```

Note that the full low-level code for *extractURIByRel* – responsible for extracting the URIs by *rel* relation [is shown here](#).

This test covers the thorny subject of Link Relations in REST: the URI to retrieve all resources uses the *rel="collection" semantics*.

This type of link relation has not yet been standardized, but is already [in use](#) by several microformats and proposed for standardization. Usage of non-standard link relations opens up the discussion about microformats and richer semantics in RESTful web services.



Other URIs could potentially be discovered via the *Link* header, but there is only so much the existing types of link relations allow without moving to a richer semantic markup such as [defining custom link relations](#), the [Atom Publishing Protocol](#) or [microformats](#).

For example, the client should be able to discover the URI to create new Resources when doing a *GET* on a specific Resource. Unfortunately, there is no link relation to model *create* semantics.

Luckily it's a standard practice that the URI for creation is the same as the URI to GET all resources of that type, with the only difference being the POST HTTP method.



We've seen **how a REST API is fully discoverable from the root and with no prior knowledge** – meaning the client is able to navigate it by doing a GET on the root. Moving forward, all state changes are driven by the client using the available and discoverable transitions that the REST API provides in representations (hence *Representational State Transfer*).

This chapter covered the some of the traits of discoverability in the context of a REST web service, discussing HTTP method discovery, the relation between create and get, discovery of the URI to get all resources, etc.

The implementation of all these examples and code snippets is available [over on GitHub](#).



8: An Intro to Spring HATEOAS



This chapter explains the process of creating hypermedia-driven REST web service using the Spring HATEOAS project.



The Spring HATEOAS project is a library of APIs that we can use to easily create REST representations that follow the principle of HATEOAS (Hypertext as the Engine of Application State).

Generally speaking, the principle implies that the API should guide the client through the application by returning relevant information about the next potential steps, along with each response.

In this chapter, we're going to build an example using Spring HATEOAS with the goal of decoupling the client and server, and theoretically allowing the API to change its URI scheme without breaking clients.

3. Preparation



First, let's add the Spring HATEOAS dependency:

```
1. <dependency>
2.     <groupId>org.springframework.boot</groupId>
3.     <artifactId>spring-boot-starter-hateoas</artifactId>
4.     <version>2.1.4.RELEASE</version>
5. </dependency>
```

If we're not using Spring Boot we can add the following libraries to our project:

```
1. <dependency>
2.     <groupId>org.springframework.hateoas</groupId>
3.     <artifactId>spring-hateoas</artifactId>
4.     <version>0.25.1.RELEASE</version>
5. </dependency>
6. <dependency>
7.     <groupId>org.springframework.plugin</groupId>
8.     <artifactId>spring-plugin-core</artifactId>
9.     <version>1.2.0.RELEASE</version>
10. </dependency>
```

As always, we can search the latest versions of the starter HATEOAS, the spring-hateoas and the spring-plugin-core dependencies in Maven Central.

Next, we have the Customer resource without Spring HATEOAS support:

```
1. public class Customer {
2.
3.     private String customerId;
4.     private String customerName;
5.     private String companyName;
6.
7.     // standard getters and setters
8. }
```

Finally, the Customer resource representation:

```
1. {
2.     "customerId": "10A",
3.     "customerName": "Jane",
4.     "customerCompany": "ABC Company"
5. }
```

4. Adding HATEOAS Support



In a Spring HATEOAS project, we don't need to either look up the Servlet context nor concatenate the path variable to the base URI.

Instead, **Spring HATEOAS offers three abstractions for creating the URI – *ResourceSupport*, *Link*, and *ControllerLinkBuilder***. We can use these to create the metadata and associate it to the resource representation.

4.1. Adding Hypermedia Support to a Resource

The project provides a base class called *ResourceSupport* to inherit from when creating a resource representation:

```
1. public class Customer extends ResourceSupport {  
2.     private String customerId;  
3.     private String customerName;  
4.     private String companyName;  
5.  
6.     // standard getters and setters  
7. }
```

The *Customer* resource extends from the *ResourceSupport* class to inherit the *add()* method. So once we create a link, we can easily set that value to the resource representation without adding any new fields to it.

4.2. Creating Links

Spring HATEOAS provides a *Link* object to store the metadata (location or URI of the resource).

First, we'll create a simple link manually:

```
Link link = new Link("http://localhost:8080/spring-security-rest/api/  
customers/10A");
```

The *Link* object follows the [Atom](#) link syntax and consists of a *rel* which identifies relation to the resource and *hrefattribute* which is the actual link itself.

Here's how the Customer resource looks now that it contains the new link:

```

1.  {
2.      "customerId": "10A",
3.      "customerName": "Jane",
4.      "customerCompany": "ABC Company",
5.      "_links": {
6.          "self": {
7.              "href": "http://localhost:8080/spring-security-rest/api/
8. customers/10A"
9.          }
10.     }
11. }

```

The URI associated with the response is qualified as a *self* link. The semantics of the *self* relation is clear – it's simply the canonical location the Resource can be accessed at.

4.3. Creating Better Links

Another very important abstraction offered by the library is **the *ControllerLinkBuilder*** – which **simplifies building URIs** by avoiding hard-coded the links.

The following snippet shows building the customer self-link using the *ControllerLinkBuilder* class:

```
linkTo(CustomerController.class).slash(customer.getCustomerId()).withSelfRel();
```

Let's have a look:

- the `linkTo()` method inspects the controller class and obtains its root mapping
- the `slash()` method adds the `customerId` value as the path variable of the link
- finally, the `withSelfMethod()` qualifies the relation as a self-link



In the previous section, we've shown a self-referencing relation. However, more complex systems may involve other relations as well.

For example, a *customer* can have a relationship with orders. Let's model the *Order* class as a resource as well:

```
1. public class Order extends ResourceSupport {
2.     private String orderId;
3.     private double price;
4.     private int quantity;
5.
6.     // standard getters and setters
7. }
```

At this point, we can extend the *CustomerController* with a method that returns all orders of a particular customer:

```
1. @GetMapping(value =("/{customerId}/orders", produces = { "application/
2.     hal+json" })
3. public Resources<Order> getOrdersForCustomer(@PathVariable final String
4.     customerId) {
5.     List<Order> orders = orderService.
6.     getAllOrdersForCustomer(customerId);
7.     for (final Order order : orders) {
8.         Link selfLink = linkTo(methodOn(CustomerController.class)
9.             .getOrderById(customerId, order.getOrderId())).withSelfRel();
10.        order.add(selfLink);
11.    }
12.
13.    Link link = linkTo(methodOn(CustomerController.class)
14.        .getOrdersForCustomer(customerId)).withSelfRel();
15.    Resources<Order> result = new Resources<Order>(orders, link);
16.    return result;
17. }
```

Our method returns a *Resources* object to comply with the HAL return type, as well as a *"_self"* link for each of the orders and the full list.

An important thing to notice here is that the hyperlink for the customer orders depends on the mapping of *getOrdersForCustomer()* method. We'll refer to these types of links as method links and show how the *ControllerLinkBuilder* can assist in their creation.



The *ControllerLinkBuilder* offers rich support for Spring MVC Controllers. The following example shows how to build HATEOAS hyperlinks based on the *getOrdersForCustomer()* method of the *CustomerController* class:

```
Link ordersLink = linkTo(methodOn(CustomerController.class)
    .getOrdersForCustomer(customerId)).withRel("allOrders");
```

The *methodOn()* obtains the method mapping by making dummy invocation of the target **method** on the proxy controller and sets the *customerId* as the path variable of the URI.



Let's put the self-link and method link creation all together in a *getAllCustomers()* method:

```
1. @GetMapping(produces = { "application/hal+json" })
2. public Resources<Customer> getAllCustomers() {
3.     List<Customer> allCustomers = customerService.allCustomers();
4.
5.     for (Customer customer : allCustomers) {
6.         String customerId = customer.getCustomerId();
7.         Link selfLink = linkTo(CustomerController.class).
8. slash(customerId).withSelfRel();
9.         customer.add(selfLink);
10.        if (orderService.getAllOrdersForCustomer(customerId).size() > 0)
11.    {
12.        Link ordersLink = linkTo(methodOn(CustomerController.class)
13.            .getOrdersForCustomer(customerId)).withRel("allOrders");
14.        customer.add(ordersLink);
15.    }
16.    }
17.
18.    Link link = linkTo(CustomerController.class).withSelfRel();
19.    Resources<Customer> result = new Resources<Customer>(allCustomers,
20. link);
21.    return result;
22. }
```

Next, let's invoke the *getAllCustomers()* method:

```
curl http://localhost:8080/spring-security-rest/api/customers
```

And examine the result:

```

1.  {
2.    "_embedded": {
3.      "customerList": [{
4.        "customerId": "10A",
5.        "customerName": "Jane",
6.        "companyName": "ABC Company",
7.        "_links": {
8.          "self": {
9.            "href": "http://localhost:8080/spring-security-rest/api/
10. customers/10A"
11.          },
12.          "allOrders": {
13.            "href": "http://localhost:8080/spring-security-rest/api/
14. customers/10A/orders"
15.          }
16.        }
17.      }, {
18.        "customerId": "20B",
19.        "customerName": "Bob",
20.        "companyName": "XYZ Company",
21.        "_links": {
22.          "self": {
23.            "href": "http://localhost:8080/spring-security-rest/api/
24. customers/20B"
25.          },
26.          "allOrders": {
27.            "href": "http://localhost:8080/spring-security-rest/api/
28. customers/20B/orders"
29.          }
30.        }
31.      }, {
32.        "customerId": "30C",
33.        "customerName": "Tim",
34.        "companyName": "CKV Company",
35.        "_links": {
36.          "self": {
37.            "href": "http://localhost:8080/spring-security-rest/api/
38. customers/30C"
39.          }
40.        }
41.      }
42.    ],
43.    "_links": {
44.      "self": {
45.        "href": "http://localhost:8080/spring-security-rest/api/customers"
46.      }
47.    }
48.  }

```

Within each resource representation, there is a *self* link and the *allOrders* link to extract all orders of a customer. If a customer doesn't have orders, then the link for orders won't appear.

This example demonstrates how Spring HATEOAS fosters API discoverability in a rest web service. **If the link exists, the client can follow it and get all orders for a customer:**

```
1. curl http://localhost:8080/spring-security-rest/api/customers/10A/orders
2.
3. {
4.   "_embedded": {
5.     "orderList": [{
6.       "orderId": "001A",
7.       "price": 150,
8.       "quantity": 25,
9.       "_links": {
10.        "self": {
11.          "href": "http://localhost:8080/spring-security-rest/api/
12. customers/10A/001A"
13.        }
14.      }
15.    }, {
16.      "orderId": "002A",
17.      "price": 250,
18.      "quantity": 15,
19.      "_links": {
20.        "self": {
21.          "href": "http://localhost:8080/spring-security-rest/api/
22. customers/10A/002A"
23.        }
24.      }
25.    }]
26.   },
27.   "_links": {
28.     "self": {
29.       "href": "http://localhost:8080/spring-security-rest/api/
30. customers/10A/orders"
31.     }
32.   }
33. }
```



In this chapter, we've discussed how to **build a hypermedia-driven Spring REST web service using the Spring HATEOAS project**.

In the example, we see that the client can have a single entry point to the application and further actions can be taken based on the metadata in the response representation.

This allows the server to change its URI scheme without breaking the client. Also, the application can advertise new capabilities by putting new links or URIs in the representation.

Finally, the full implementation of this chapter can be found in the [GitHub project](#).

9: REST Pagination in Spring



This tutorial will focus on **the implementation of pagination in a REST API, using Spring MVC and Spring Data.**

2. Page as Resource vs Page as Representation



The first question when designing pagination in the context of a RESTful architecture is whether to consider the **page an actual Resource or just a Representation of Resources**.

Treating the page itself as a resource introduces a host of problems such as no longer being able to uniquely identify resources between calls. This, coupled with the fact that, in the persistence layer, the page is not a proper entity but a holder that is constructed when necessary, makes the choice straightforward: **the page is part of the representation**.

The next question in the pagination design in the context of REST is **where to include the paging information**:

- in the URI path: */foo/page/1*
- the URI query: */foo?page=1*

Keeping in mind that a **page is not a Resource**, encoding the page information in the URI is no longer an option.

We're going to use the standard way of solving this problem by **encoding the paging information in a URI query**.



Now, for the implementation – the Spring MVC Controller for pagination is straightforward:

```
1. @GetMapping(params = { "page", "size" })
2. public List<Foo> findPaginated(@RequestParam("page") int page,
3.    @RequestParam("size") int size, UriComponentsBuilder uriBuilder,
4.    HttpServletResponse response) {
5.     Page<Foo> resultPage = service.findPaginated(page, size);
6.     if (page > resultPage.getTotalPages()) {
7.         throw new MyResourceNotFoundException();
8.     }
9.     eventPublisher.publishEvent(new PaginatedResultsRetrievedEvent<Foo>(
10.        Foo.class, uriBuilder, response, page, resultPage.getTotalPages(),
11.        size));
12.
13.     return resultPage.getContent();
14. }
```

In this example, we're injecting the two query parameters, size and page, in the Controller method via *@RequestParam*.

Alternatively, we could have used a *Pageable* object, which maps the *page*, *size*, and *sort* parameters automatically. In addition, the *PagingAndSortingRepository* entity provides out-of-the-box methods that support using the *Pageable* as a parameter as well.

We're also injecting both the *Http Response* and the *UriComponentsBuilder* to help with Discoverability – which we're decoupling via a custom event. If that's not a goal of the API, you can simply remove the custom event.

Finally – note that the focus of this chapter is only the REST and the web layer – to go deeper into the data access part of pagination you can [check out this article](#) about Pagination with Spring Data.

4. Discoverability for REST Pagination



Within the scope of pagination, satisfying the **HATEOAS constraint of REST** means enabling the client of the API to discover the *next* and *previous* pages based on the current page in the navigation. For this purpose, **we're going to use the *Link* HTTP header, coupled with the "next", "prev", "first" and "last" link relation types.**

In REST, **Discoverability is a cross-cutting concern**, applicable not only to specific operations but to types of operations. For example, each time a Resource is created, the URI of that Resource should be discoverable by the client. Since this requirement is relevant for the creation of ANY Resource, we'll handle it separately.

We'll decouple these concerns using events. In the case of pagination, the event – *PaginatedResultsRetrievedEvent* – is fired in the controller layer. Then we'll implement discoverability with a custom listener for this event.

In short, the listener will check if the navigation allows for a *next*, *previous*, *first* and *last* pages. If it does – it will **add the relevant URIs to the response as a 'Link' HTTP Header.**

Let's go step by step now. The *UriComponentsBuilder* passed from the controller contains only the base URL (the host, the port and the context path). Therefore, we'll have to add the remaining sections:

```
1. void addLinkHeaderOnPagedResourceRetrieval(  
2.     UriComponentsBuilder uriBuilder, HttpServletResponse response,  
3.     Class clazz, int page, int totalPages, int size ){  
4.  
5.     String resourceName = clazz.getSimpleName().toString().toLowerCase();  
6.     uriBuilder.path( "/admin/" + resourceName );  
7.  
8.     // ...  
9.  
10. }
```

Next, we'll use a *StringJoiner* to concatenate each link. We'll use the *uriBuilder* to generate the URIs. Let's see how we'd proceed with the link to the *next* page:

```
1. StringJoiner linkHeader = new StringJoiner(", ");  
2. if (hasNextPage(page, totalPages)){  
3.     String uriForNextPage = constructNextPageUri(uriBuilder, page,  
4.     size);  
5.     linkHeader.add(createLinkHeader(uriForNextPage, "next"));  
6. }
```

Let's have a look at the logic of the *constructNextPageUri* method:

```
1. String constructNextPageUri(UriComponentsBuilder uriBuilder, int page,  
2. int size) {  
3.     return uriBuilder.replaceQueryParam(PAGE, page + 1)  
4.         .replaceQueryParam("size", size)  
5.         .build()  
6.         .encode()  
7.         .toUriString();  
8. }
```

We'll proceed similarly for the rest of the URIs that we want to include.

Finally, we'll add the output as a response header:

```
1. response.addHeader("Link", linkHeader.toString());
```

Note that, for brevity, I included only a partial code sample and [the full code here](#).

5. Test Driving Pagination



Both the main logic of pagination and discoverability are covered by small, focused integration tests. We'll use the [REST-assured library](#) to consume the REST service and to verify the results.

These are a few examples of pagination integration tests; for a full test suite, check out the GitHub project (link at the end of the chapter):

```
1.  @Test
2.  public void whenResourcesAreRetrievedPaged_then200IsReceived(){
3.      Response response = RestAssured.get(paths.getFooURL() +
4.      "?page=0&size=2");
5.
6.      assertThat(response.getStatusCode(), is(200));
7.  }
8.  @Test
9.  public void whenPageOfResourcesAreRetrievedOutOfBounds_
10. then404IsReceived(){
11.     String url = getFooURL() + "?page=" + randomNumeric(5) + "&size=2";
12.     Response response = RestAssured.get.get(url);
13.
14.     assertThat(response.getStatusCode(), is(404));
15. }
16. @Test
17. public void givenResourcesExist_whenFirstPageIsRetrieved_
18. thenPageContainsResources(){
19.     createResource();
20.     Response response = RestAssured.get(paths.getFooURL() +
21.     "?page=0&size=2");
22.
23.     assertFalse(response.body().as(List.class).isEmpty());
24. }
```

6. Test Driving Pagination Discoverability



Testing that pagination is discoverable by a client is relatively straightforward, although there is a lot of ground to cover.

The tests will focus on the position of the current page in navigation and the different URIs that should be discoverable from each position:

```
1.  @Test
2.  public void whenFirstPageOfResourcesAreRetrieved_thenSecondPageIsNext() {
3.      Response response = RestAssured.get(getFooURL()+"?page=0&size=2");
4.
5.      String uriToNextPage = extractURIByRel(response.getHeader("Link"),
6.      "next");
7.      assertEquals(getFooURL()+"?page=1&size=2", uriToNextPage);
8.  }
9.  @Test
10. public void whenFirstPageOfResourcesAreRetrieved_thenNoPreviousPage() {
11.     Response response = RestAssured.get(getFooURL()+"?page=0&size=2");
12.
13.     String uriToPrevPage = extractURIByRel(response.getHeader("Link"),
14.     "prev");
15.     assertNull(uriToPrevPage);
16. }
17. @Test
18. public void whenSecondPageOfResourcesAreRetrieved_
19. thenFirstPageIsPrevious() {
20.     Response response = RestAssured.get(getFooURL()+"?page=1&size=2");
21.
22.     String uriToPrevPage = extractURIByRel(response.getHeader("Link"),
23.     "prev");
24.     assertEquals(getFooURL()+"?page=0&size=2", uriToPrevPage);
25. }
26. @Test
27. public void whenLastPageOfResourcesIsRetrieved_
28. thenNoNextPageIsDiscoverable() {
29.     Response first = RestAssured.get(getFooURL()+"?page=0&size=2");
30.     String uriToLastPage = extractURIByRel(first.getHeader("Link"),
31.     "last");
32.
33.     Response response = RestAssured.get(uriToLastPage);
34.
35.     String uriToNextPage = extractURIByRel(response.getHeader("Link"),
36.     "next");
37.     assertNull(uriToNextPage);
38. }
```

Note that the full low-level code for *extractURIByRel* – responsible for extracting the URIs by *rel* relation is [here](#).



On the same topic of pagination and discoverability, **the choice must be made if a client is allowed to retrieve all the Resources in the system at once, or if the client must ask for them paginated.**

If the choice is made that the client cannot retrieve all Resources with a single request, and pagination is not optional but required, then several options are available for the response to a get all request. One option is to return a 404 (*Not Found*) and use the *Link* header to make the first page discoverable:

```
Link=<http://localhost:8080/rest/api/admin/foo?page=0&size=2>; rel="first",  
<http://localhost:8080/rest/api/admin/foo?page=103&size=2>; rel="last"
```

Another option is to return redirect – 303 (*See Other*) – to the first page. A more conservative route would be to simply return to the client a 405 (*Method Not Allowed*) for the GET request.

8. REST Paging with Range HTTP Headers



A relatively different way of implementing pagination is to work with the **HTTP Range headers** – *Range*, *Content-Range*, *If-Range*, *Accept-Ranges* – and **HTTP status codes** – 206 (*Partial Content*), 413 (*Request Entity Too Large*), 416 (*Requested Range Not Satisfiable*).

One view on this approach is that the HTTP Range extensions were not intended for pagination and that they should be managed by the Server, not by the Application. Implementing pagination based on the HTTP Range header extensions is nevertheless technically possible, although not nearly as common as the implementation discussed in this chapter.

9. Spring Data REST Pagination



In Spring Data, if we need to return a few results from the complete data set, we can use any *Pageable* repository method, as it will always return a *Page*. The results will be returned based on the page number, page size, and sorting direction.

Spring Data REST automatically recognizes URL parameters like *page*, *size*, *sort* etc. To use paging methods of any repository we need to extend *PagingAndSortingRepository*:

```
public interface SubjectRepository extends
PagingAndSortingRepository<Subject, Long>{}
```

If we call *http://localhost:8080/subjects* Spring automatically adds the *page*, *size*, *sort* parameters suggestions with the API:

```
1.  "_links" : {
2.    "self" : {
3.      "href" : "http://localhost:8080/subjects{?page,size,sort}",
4.      "templated" : true
5.    }
6.  }
```

By default, the page size is 20 but we can change it by calling something like *http://localhost:8080/subjects?page=10*. If we want to implement paging into our own custom repository API we need to pass an additional *Pageable* parameter and make sure that API returns a *Page*:

```
@RestResource(path = "nameContains")
public Page<Subject> findByNameContaining(@Param("name") String name, Pageable
p);
```

Whenever we add a custom API a */search* endpoint gets added to the generated links. So if we call *http://localhost:8080/subjects/search* we will see a pagination capable endpoint:

```
1.  "findByNameContaining" : {
2.    "href" : "http://localhost:8080/subjects/search/
3.    nameContains{?name,page,size,sort}",
4.    "templated" : true
5.  }
```

All APIs that implement *PagingAndSortingRepository* will return a *Page*. If we need to return the list of the results from the *Page*, the *getContent()* API of *Page* provides the list of records fetched as a result of the Spring Data REST API.



This chapter illustrated how to implement Pagination in a REST API using Spring, and discussed how to set up and test Discoverability.

If you want to go in depth on pagination in the persistence level, check out my [JPA](#) or [Hibernate](#) pagination tutorials.

The implementation of all these examples and code snippets can be found in the [GitHub project](#).

10: Test a REST API with Java



This tutorial focuses on the basic principles and mechanics of **testing a REST API with live Integration Tests** (with a JSON payload).

The main goal is to provide an introduction to testing the basic correctness of the API – and we're going to be using the latest version of the [GitHub REST API](#) for the examples.

For an internal application, this kind of testing will usually run as a late step in a Continuous Integration process, consuming the REST API after it has already been deployed.

When testing a REST resource, there are usually a few orthogonal responsibilities the tests should focus on:

- the HTTP **response code**
- other HTTP **headers** in the response
- the **payload** (JSON, XML)

Each test should only focus on a single responsibility and include a single assertion.

Focusing on a clear separation always has benefits, but when doing this kind of black box testing is even more important, as the general tendency is to write complex test scenarios in the very beginning.

Another important aspect of the integration tests is adherence to the *Single Level of Abstraction Principle* – the logic within a test should be written at a high level. Details such as creating the request, sending the HTTP request to the server, dealing with IO, etc should not be done inline but via utility methods.

2. Testing the Status Code



```
1.  @Test
2.  public void givenUserDoesNotExists_whenUserInfoIsRetrieved_
3.  then404IsReceived()
4.      throws ClientProtocolException, IOException {
5.
6.      // Given
7.      String name = RandomStringUtils.randomAlphabetic( 8 );
8.      HttpRequest request = new HttpGet( "https://api.github.com/
9.  users/" + name );
10.
11.     // When
12.     HttpResponse httpResponse = HttpClientBuilder.create().build().
13. execute( request );
14.
15.     // Then
16.     assertThat(
17.         httpResponse.getStatusLine().getStatusCode(),
18.         equalTo(HttpStatus.SC_NOT_FOUND) );
19. }
```

This is a rather simple test – **it verifies that a basic happy path is working**, without adding too much complexity to the test suite.

If for whatever reason, it fails, then there is no need to look at any other test for this URL until this is fixed.

3. Testing the Media Type



```
1.  @Test
2.  public void
3.  givenRequestWithNoAcceptHeader_whenRequestIsExecuted_
4.  thenDefaultResponseContentTypeIsJson()
5.      throws ClientProtocolException, IOException {
6.
7.      // Given
8.      String jsonMimeType = "application/json";
9.      HttpRequest request = new HttpGet( "https://api.github.com/users/
10. eugenp" );
11.
12.      // When
13.      HttpResponse response = HttpClientBuilder.create().build().execute(
14. request );
15.
16.      // Then
17.      String mimeType = ContentType.getDefault(response.getEntity()).
18. getMimeType();
19.      assertEquals( jsonMimeType, mimeType );
20. }
```

This ensures that the Response actually contains JSON data.

As you might have noticed, **we're following a logical progression of tests** – first the Response Status Code (to ensure that the request was OK), then the Media Type of the Response, and only in the next test will we look at the actual JSON payload.

4. Testing the JSON Payload



```
1.  @Test
2.  public void
3.      givenUserExists_whenUserInformationIsRetrieved_
4.      thenRetrievedResourceIsCorrect()
5.      throws ClientProtocolException, IOException {
6.
7.          // Given
8.          HttpRequest request = new HttpGet( "https://api.github.com/users/
9.      eugenp" );
10.
11.         // When
12.         HttpResponse response = HttpClientBuilder.create().build().execute(
13.     request );
14.
15.         // Then
16.         GitHubUser resource = RetrieveUtil.retrieveResourceFromResponse(
17.             response, GitHubUser.class);
18.         assertThat( "eugenp", Matchers.is( resource.getLogin() ) );
19.     }
```

In this case, I know the default representation of GitHub resources is JSON, but usually, the *Content-Type* header of the response should be tested alongside the *Accept* header of the request – the client asks for a particular type of representation via *Accept*, which the server should honor.

5. Utilities for Testing



We're going to use Jackson 2 to unmarshall the raw JSON String into a type-safe Java Entity:

```
1. public class GitHubUser {  
2.  
3.     private String login;  
4.  
5.     // standard getters and setters  
6. }
```

We're only using a simple utility to keep the tests clean, readable and at a high level of abstraction:

```
1. public static <T> T retrieveResourceFromResponse(HttpResponse response,  
2. Class<T> clazz)  
3.     throws IOException {  
4.  
5.     String jsonFromResponse = EntityUtils.toString(response.  
6. getEntity());  
7.     ObjectMapper mapper = new ObjectMapper()  
8.         .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,  
9. false);  
10.     return mapper.readValue(jsonFromResponse, clazz);  
11. }
```

Notice that [Jackson is ignoring unknown properties](#) that the GitHub API is sending our way – that's simply because the Representation of a User Resource on GitHub gets pretty complex – and we don't need any of that information here.



The utilities and tests make use of the following libraries, all available in Maven central:

- [HttpClient](#)
- [Jackson 2](#)
- [Hamcrest](#) (optional)



This is only one part of what the complete integration testing suite should be. The tests focus on **ensuring basic correctness for the REST API**, without going into more complex scenarios,

For example, the following are not covered: Discoverability of the API, consumption of different representations for the same Resource, etc.

The implementation of all these examples and code snippets can be found [over on Github](#).