Baeldung

Build Your API with Spring

Table of Contents

1: Bootstrap a Web Application with Spring 5

2: Build a REST API with Spring and Java Config

Table of Contents

3: Http Message Converters with the Spring Framework

Table of Contents

4: Spring's RequestBody and ResponseBody Annotations

5: Entity To DTO Conversion for a Spring REST API

6: Error Handling for REST with Spring

7: REST API Discoverability and HATEOAS

8: An Intro to Spring HATEOAS

9: REST Pagination in Spring

10: Test a REST API with Java

1: Bootstrap a Web Application with Spring 5

1. Overview

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:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.1.1.RELEASE</version>
</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*:

```
@SpringBootApplication
public class SpringBootRestApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootRestApplication.class, args);
    }
}
```

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.
4.  }
```

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:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.0.0.RELEASE</version>
</dependency>
```

3.2. The Java-based Web Configuration

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

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.baeldung.controller")
public class WebConfig {

}
```

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 ServletException {
5.          AnnotationConfigWebApplicationContext context = new
6.  AnnotationConfigWebApplicationContext();
7.          context.scan("com.baeldung");
8.          container.addListener(new ContextLoaderListener(context));
9.
10.         ServletRegistration.Dynamic dispatcher =
11.           container.addServlet("mvc", new DispatcherServlet(context));
12.         dispatcher.setLoadOnStartup(1);
13.         dispatcher.addMapping("/");
14.     }
15. }
```

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:

```
1.  <context:component-scan base-package="com.baeldung.controller" />
2.  <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.

5. Conclusion

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

1. Overview

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

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 one would expect from a RESTful service.

```
@Configuration
@EnableWebMvc
public class WebConfig{
    //
}
```

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.

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.    }
10. }
```

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*:

```
@RunWith(SpringRunner.class)
@WebMvcTest(FooController.class)
public class FooControllerWebLayerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private IFooService service;

    @Test()
    public void whenTestMvcController_thenRetrieveExpectedResult() throws Exception {
        // ...

        this.mockMvc.perform(get(“/foos”)
            .andExpect(...);
    }
}
```

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*:

```
@RestController
@RequestMapping("/foos")
class FooController {

    @Autowired
    private IFooService service;

    @GetMapping
    public List<Foo> findAll() {
        return service.findAll();
    }

    @GetMapping(value = "/{id}")
    public Foo findById(@PathVariable("id") Long id) {
        return RestPreconditions.checkFound(service.findById(id));
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Long create(@RequestBody Foo resource) {
        Preconditions.checkNotNull(resource);
        return service.create(resource);
    }

    @PutMapping(value = "/{id}")
    @ResponseStatus(HttpStatus.OK)
    public void update(@PathVariable( "id" ) Long id, @RequestBody Foo resource) {
        Preconditions.checkNotNull(resource);
        RestPreconditions.checkNotNull(service.getById(resource.getId()));
        service.update(resource);
    }

    @DeleteMapping(value = "/{id}")
    @ResponseStatus(HttpStatus.OK)
    public void delete(@PathVariable("id") Long id) {
        service.deleteById(id);
    }
}
```

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

```
1.  public class RestPreconditions {
2.      public static <T> T checkFound(T resource) {
3.          if (resource == null) {
4.              throw new MyResourceNotFoundException();
5.          }
6.          return resource;
7.      }
8.  }
```

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 *Accept*header, although other HTTP headers may be used to determine the representation as well.

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:

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
public class BadRequestException extends RuntimeException {
    //
}
@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
    //
}
```

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 declares 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:

```
<dependencies>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.9.8</version>
    </dependency>
    <dependency>
        <groupId>javax.xml.bind</groupId>
        <artifactId>jaxb-api</artifactId>
        <version>2.3.1</version>
        <scope>runtime</scope>
    </dependency>
</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:**

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.1.2.RELEASE</version>
</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.

8. Conclusion

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

1. Overview

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 *String*
- *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:

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

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:

```
public class Foo {
    private long id;
    private String name;
}
```

And the Http Response Body:

```
{
    "id": 1,
    "name": "Paul",
}
```

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 String
3.  id) {
4.      fooService.update(foo);
5.  }
```

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

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

We get back a 200 OK – a successful response:

```
1.  HTTP/1.1 200 OK
2.  Server: Apache-Coyote/1.1
3.  Content-Length: 0
4.  Date: Fri, 10 Jan 2014 11:18:54 GMT
```

We can also **customize the message converters by implementing the *WebMvcConfigurer* interface** & 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 MappingJackson2HttpMessageConverter());
12.     }
13.     private HttpMessageConverter<Object> createXmlHttpMessageConverter() {
14.         MarshallingHttpMessageConverter xmlConverter =
15.           new MarshallingHttpMessageConverter();
16.
17.         XStreamMarshaller xstreamMarshaller = new XStreamMarshaller();
18.         xmlConverter.setMarshaller(xstreamMarshaller);
19.         xmlConverter.setUnmarshaller(xstreamMarshaller);
20.
21.         return xmlConverter;
22.     }
23. }
```

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.oxm.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:

```
@Bean
public HttpMessageConverter<Object> createXmlHttpMessageConverter() {
    MarshallingHttpMessageConverter xmlConverter = new
MarshallingHttpMessageConverter();

    // ...

    return xmlConverter;
}
```

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.  @Test
2.  public void testGetFoo() {
3.      String URI = "http://localhost:8080/spring-boot-rest/foos/{id}";
4.      RestTemplate restTemplate = new RestTemplate();
5.      Foo foo = restTemplate.getForObject(URI, Foo.class, "1");
6.      Assert.assertEquals(new Integer(1), foo.getId());
7.  }
```

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:

```
@Test
public void givenConsumingXml_whenReadingTheFoo_thenCorrect() {
    String URI = BASE_URI + "foos/{id}";
    RestTemplate restTemplate = new RestTemplate();
    restTemplate.setMessageConverters(getMessageConverters());

    HttpHeaders headers = new HttpHeaders();
    headers.setAccept(Arrays.asList(MediaType.APPLICATION_XML));
    HttpEntity<String> entity = new HttpEntity<String>(headers);

    ResponseEntity<Foo> response =
      restTemplate.exchange(URI, HttpMethod.GET, entity, Foo.class, "1");
    Foo resource = response.getBody();

    assertThat(resource, notNullValue());
}
private List<HttpMessageConverter<?>> getMessageConverters() {
    XStreamMarshaller marshaller = new XStreamMarshaller();
    MarshallingHttpMessageConverter marshallingConverter =
      new MarshallingHttpMessageConverter(marshaller);

    List<HttpMessageConverter<?>> converters =
      ArrayList<HttpMessageConverter<?>>();
    converters.add(marshallingConverter);
    return converters;
}
```

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

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

```
@Test
public void givenConsumingJson_whenReadingTheFoo_thenCorrect() {
    String URI = BASE_URI + "foos/{id}";

    RestTemplate restTemplate = new RestTemplate();
    restTemplate.setMessageConverters(getMessageConverters());

    HttpHeaders headers = new HttpHeaders();
    headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    HttpEntity<String> entity = new HttpEntity<String>(headers);

    ResponseEntity<Foo> response =
    restTemplate.exchange(URI, HttpMethod.GET, entity, Foo.class, "1");
    Foo resource = response.getBody();

    assertThat(resource, notNullValue());
}
private List<HttpMessageConverter<?>> getMessageConverters() {
    List<HttpMessageConverter<?>> converters =
      new ArrayList<HttpMessageConverter<?>>();
    converters.add(new MappingJackson2HttpMessageConverter());
    return converters;
}
```

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, "jason");
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 marshall/unmarshall 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

1. Introduction

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

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:

```
@PostMapping("/request")
public ResponseEntity postController(
  @RequestBody LoginForm loginForm) {

    exampleService.fakeAuthenticate(loginForm);
    return ResponseEntity.ok(HttpStatus.OK);
}
```

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:

```
public class LoginForm {
    private String username;
    private String password;
    // ...
}
```

Here, the object we use to represent the *HttpRequest* body maps to our *LoginForm* object. Let's test this using cURL:

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

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

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 *@RequestBody* and *@ResponseBody* annotations.

As always code samples are available over on GitHub.

5: Entity To DTO Conversion for a Spring REST API

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*:

```
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>2.3.2</version>
</dependency>
```

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

```
@Bean
public ModelMapper modelMapper() {
    return new ModelMapper();
}
```

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

```
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 ParseException {
16.         dateFormat.setTimeZone(TimeZone.getTimeZone(timezone));
17.         return dateFormat.parse(this.date);
18.     }
19.
20.     public void setSubmissionDate(Date date, String timezone) {
21.         dateFormat.setTimeZone(TimeZone.getTimeZone(timezone));
22.         this.date = dateFormat.format(date);
23.     }
24.
25.     // standard getters and setters
26. }
```

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

getSubmissionDateConverted() method converts a date String into a date in server's timezone to use it in persisting Post entity

setSubmissionDate() method is to set *PostDto's* date to Post's Date instance in the current user's timezone

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), sort);
6.
7.      Page<Post> posts = postRepository
8.        .findByUser(userService.getCurrentUser(), pageReq);
9.      return posts.getContent();
10. }
```

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.

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, sort);
18.         return posts.stream()
19.           .map(post -> convertToDto(post))
20.           .collect(Collectors.toList());
21.     }
22.
23.     @RequestMapping(method = RequestMethod.POST)
24.     @ResponseStatus(HttpStatus.CREATED)
25.     @ResponseBody
26.     public PostDto createPost(@RequestBody PostDto postDto) {
27.         Post post = convertToEntity(postDto);
28.         Post postCreated = postService.createPost(post));
29.         return convertToDto(postCreated);
30.     }
```

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET)
    @ResponseBody
    public PostDto getPost(@PathVariable("id") Long id) {
        return convertToDto(postService.getPostById(id));
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
    @ResponseStatus(HttpStatus.OK)
    public void updatePost(@RequestBody PostDto postDto) {
        Post post = convertToEntity(postDto);
        postService.updatePost(post);
    }
}
```

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

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

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

```
private Post convertToEntity(PostDto postDto) throws ParseException {
    Post post = modelMapper.map(postDto, Post.class);
    post.setSubmissionDate(postDto.getSubmissionDateConverted(
      userService.getCurrentUser().getPreference().getTimezone()));

    if (postDto.getId() != null) {
        Post oldPost = postService.getPostById(postDto.getId());
        post.setRedditID(oldPost.getRedditID());
        post.setSent(oldPost.isSent());
    }
    return post;
}
```

So, as you can see, with the help of the model mapper, **the conversion logic is quick and simple** – we're using the 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. }
```

7. Conclusion

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

1. Overview

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 needs.

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.      public void handleException() {
6.          //
7.      }
8.  }
```

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.

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. *ResponseStatusExceptionResolver*

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:

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException() {
        super();
    }
    public ResourceNotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
    public ResourceNotFoundException(String message) {
        super(message);
    }
    public ResourceNotFoundException(Throwable cause) {
        super(cause);
    }
}
```

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. *SimpleMappingExceptionResolver* and *AnnotationMethodHandlerExceptionResolver*

The *SimpleMappingExceptionResolver* has been around for quite some time – it comes out of the older Spring MVC model and is **not very relevant for a REST Service**. We basically use it to map exception class names to view names. The *AnnotationMethodHandlerExceptionResolver* was introduced in Spring 3.0 to handle exceptions through the *@ExceptionHandler* annotation but has been deprecated by *ExceptionHandlerExceptionResolver* as of Spring 3.2.

3.5. Custom *HandlerExceptionResolver*

The combination of *DefaultHandlerExceptionResolver* and *ResponseStatusExceptionResolver* 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 RestResponseStatusExceptionResolver
3.    extends AbstractHandlerExceptionResolver {
4.
5.      @Override
6.      protected ModelAndView doResolveException(
7.        HttpServletRequest request,
8.        HttpServletResponse response,
9.        Object handler,
10.       Exception ex) {
11.         try {
12.             if (ex instanceof IllegalArgumentException) {
13.                 return handleIllegalArgument(
14.                   (IllegalArgumentException) ex,
15.                   response,
16.                   handler);
17.             }
18.             ...
19.         } catch (Exception handlerException) {
20.             logger.warn("Handling of [" + ex.getClass().getName() +
21.               "] resulted in Exception", handlerException);
22.         }
23.         return null;
24.     }
```

```
25.         private ModelAndView handleIllegalArgument(
26.           IllegalArgumentException ex,
27.           HttpServletResponse response) throws IOException {
28.             response.sendError(HttpServletResponse.SC_CONFLICT);
29.             String accept = request.getHeader(HttpHeaders.ACCEPT);
30.             ...
31.             return new ModelAndView();
32.         }
33.     }
```

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 *HtttpServletResponse* and it fits into the old MVC model which uses *ModelAndView* – so there's still room for improvement.

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*:

```
@ControllerAdvice
public class RestResponseEntityExceptionHandler
  extends ResponseEntityExceptionHandler {

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

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*:
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: ...
```

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

```
@GetMapping(value = "/{id}")
public Foo findById(@PathVariable("id") Long id, HttpServletResponse response) {
    try {
        Foo resourceById = RestPreconditions.checkFound(service.findOne(id));

        eventPublisher.publishEvent(new SingleResourceRetrievedEvent(this,
response));
        return resourceById;
     }
    catch (MyResourceNotFoundException exc) {
         throw new ResponseStatusException(
           HttpStatus.NOT_FOUND, "Foo Not Found", exc);
    }
}
```

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.

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

6.1. MVC – Custom Error Page

First, let's look at the MVC style of the solution and see how to customize an error page for Access Denied:

The XML configuration:

```
<http>
 <intercept-url pattern="/admin/*" access="hasAnyRole('ROLE_ADMIN')"/>
    ...
  <access-denied-handler error-page="/my-error-page" />
</http>
```

And the Java configuration:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin/*").hasAnyRole("ROLE_ADMIN")
        ...
        .and()
        .exceptionHandling().accessDeniedPage("/my-error-page");
}
```

When users try to access a resource without having enough authorities, they will be redirected to */my-error-page*.

6.2. Custom *AccessDeniedHandler*

Next, let's see how to write our custom *AccessDeniedHandler*:

```
@Component
public class CustomAccessDeniedHandler implements AccessDeniedHandler {
    @Override
    public void handle
      (HttpServletRequest request, HttpServletResponse response,
AccessDeniedException ex)
      throws IOException, ServletException {
        response.sendRedirect("/my-error-page");
    }
}
```

And now let's configure it using **XML Configuration**:

```
<http>
    <intercept-url pattern="/admin/*" access="hasAnyRole('ROLE_ADMIN')"/>
    ...
    <access-denied-handler ref="customAccessDeniedHandler" />
</http>
```

Or using Java Configuration:

```
@Autowired
private CustomAccessDeniedHandler accessDeniedHandler;

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin/*").hasAnyRole("ROLE_ADMIN")
        ...
        .and()
        .exceptionHandling().accessDeniedHandler(accessDeniedHandler)
}
```

Note how – in our *CustomAccessDeniedHandler*, we can customize the response as we wish by redirecting or display a custom error message.

6.3. REST and Method Level Security

Finally, let's see how to handle method-level security *using @PreAuthorize*, *@PostAuthorize*, and *@Secured* Access Denied.

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

```
@ControllerAdvice
public class RestResponseEntityExceptionHandler
  extends ResponseEntityExceptionHandler {

    @ExceptionHandler({ AccessDeniedException.class })
    public ResponseEntity<Object> handleAccessDeniedException(
      Exception ex, WebRequest request) {
       return new ResponseEntity<Object>(
       "Access denied message here", new HttpHeaders(), HttpStatus.FORBIDDEN);
    }

    ...
}
```

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:

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

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*, overridding the Whitelabel Error 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:

```
@Component
public class MyCustomErrorAttributes extends DefaultErrorAttributes {

    @Override
    public Map<String, Object> getErrorAttributes(WebRequest webRequest,
boolean includeStackTrace) {
        Map<String, Object> errorAttributes = super.
getErrorAttributes(webRequest, includeStackTrace);
        errorAttributes.put("locale", webRequest.getLocale()
            .toString());
        errorAttributes.remove("error");

        //...

        return errorAttributes;
    }
}
```

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 *ErrorController* bean.

Again, we can make use of the default *BasicErrorController* 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 *using the @RequestMapping annotation and stating* it produces *application/xml* media type:

```
@Component
public class MyErrorController extends BasicErrorController {

    public MyErrorController(ErrorAttributes errorAttributes) {
        super(errorAttributes, new ErrorProperties());
    }
    @RequestMapping(produces = MediaType.APPLICATION_XML_VALUE)
    public ResponseEntity<Map<String, Object>> xmlError(HttpServletRequest request) {

    // ...
    }
}
```

8. Conclusion

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

1. Overview

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

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.

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_thenAllowHeaderListsTheAllowedActions(){
4.      // Given
5.      String uriOfExistingResource = restTemplate.createResource();
6.
7.      // When
8.      Response res = givenAuth().post(uriOfExistingResource);
9.
10.     // Then
11.     String allowHeader = res.getHeader(HttpHeaders.ALLOW);
12.     assertThat( allowHeader, AnyOf.anyOf(
13.       containsString("GET"), containsString("PUT"), containsString("DELETE") ) );
14. }
```

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, MediaType.
12. 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:

```
@Test
public void whenResourceIsRetrieved_thenUriToGetAllResourcesIsDiscoverable() {
    // Given
    String uriOfExistingResource = createAsUri();

    // When
    Response getResponse = givenAuth().get(uriOfExistingResource);

    // Then
    String uriToAllResources = HTTPLinkHeaderUtil
      .extractURIByRel(getResponse.getHeader("Link"), "collection");

    Response getAllResponse = givenAuth().get(uriToAllResources);
    assertThat(getAllResponse.getStatusCode(), is(200));
}
```

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.

5. Conclusion

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

1. Overview

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.

First, let's add the Spring HATEOAS dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-hateoas</artifactId>
    <version>2.1.4.RELEASE</version>
</dependency>
```

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

```
<dependency>
    <groupId>org.springframework.hateoas</groupId>
    <artifactId>spring-hateoas</artifactId>
    <version>0.25.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.plugin</groupId>
    <artifactId>spring-plugin-core</artifactId>
    <version>1.2.0.RELEASE</version>
</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:

```
public class Customer {

    private String customerId;
    private String customerName;
    private String companyName;

    // standard getters and setters
}
```

And we have a controller class without Spring HATEOAS support:

```
@RestController
@RequestMapping(value = “/customers”)
public class CustomerController {
    @Autowired
    private CustomerService customerService;

    @GetMapping(“/{customerId}”)
    public Customer getCustomerById(@PathVariable String customerId) {
        return customerService.getCustomerDetail(customerId);
    }
}
```

Finally, the *Customer* resource representation:

```
{
    “customerId”: “10A”,
    “customerName”: “Jane”,
    “customerCompany”: “ABC Company”
}
```

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:

```
{
    "customerId": "10A",
    "customerName": "Jane",
    "customerCompany": "ABC Company",
    "_links":{
       "self":{
         "href":"http://localhost:8080/spring-security-rest/api/customers/10A"
        }
    }
}
```

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

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:

```
1. Link ordersLink = linkTo(methodOn(CustomerController.class)
2.   .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).slash(customerId).
8.  withSelfRel();
9.          customer.add(selfLink);
10.         if (orderService.getAllOrdersForCustomer(customerId).size() > 0) {
11.             Link ordersLink = linkTo(methodOn(CustomerController.class)
12.               .getOrdersForCustomer(customerId)).withRel("allOrders");
13.             customer.add(ordersLink);
14.         }
15.     }
16.
17.     Link link = linkTo(CustomerController.class).withSelfRel();
18.     Resources<Customer> result = new Resources<Customer>(allCustomers, link);
19.     return result;
20. }
```

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

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

And examine the result:

```
{
  "_embedded": {
    "customerList": [{
       "customerId": "10A",
       "customerName": "Jane",
       "companyName": "ABC Company",
       "_links": {
         "self": {
           "href": "http://localhost:8080/spring-security-rest/api/customers/10A"
         },
         "allOrders": {
           "href": "http://localhost:8080/spring-security-rest/api/customers/10A/orders"
         }
       }
     },{
       "customerId": "20B",
       "customerName": "Bob",
       "companyName": "XYZ Company",
       "_links": {
         "self": {
           "href": "http://localhost:8080/spring-security-rest/api/customers/20B"
         },
         "allOrders": {
           "href": "http://localhost:8080/spring-security-rest/api/customers/20B/orders"
         }
       }
     },{
       "customerId": "30C",
       "customerName": "Tim",
       "companyName": "CKV Company",
       "_links": {
         "self": {
           "href": "http://localhost:8080/spring-security-rest/api/customers/30C"
         }
       }
     }]
  },
  "_links": {
    "self": {
      "href": "http://localhost:8080/spring-security-rest/api/customers"
    }
  }
}
```

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**:

```
curl http://localhost:8080/spring-security-rest/api/customers/10A/orders

{
  "_embedded": {
    "orderList": [{
        "orderId": "001A",
        "price": 150,
        "quantity": 25,
        "_links": {
          "self": {
            "href": "http://localhost:8080/spring-security-rest/api/
customers/10A/001A"
          }
        }
      },{
        "orderId": "002A",
        "price": 250,
        "quantity": 15,
        "_links": {
          "self": {
            "href": "http://localhost:8080/spring-security-rest/api/
customers/10A/002A"
          }
        }
      }]
  },
  "_links": {
    "self": {
      "href": "http://localhost:8080/spring-security-rest/api/customers/10A/
orders"
    }
  }
}
```

8. Conclusion

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

1. Overview

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

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**:

```
@GetMapping(params = { "page", "size" })
public List<Foo> findPaginated(@RequestParam("page") int page,
  @RequestParam("size") int size, UriComponentsBuilder uriBuilder,
  HttpServletResponse response) {
    Page<Foo> resultPage = service.findPaginated(page, size);
    if (page > resultPage.getTotalPages()) {
        throw new MyResourceNotFoundException();
    }
    eventPublisher.publishEvent(new PaginatedResultsRetrievedEvent<Foo>(
     Foo.class, uriBuilder, response, page, resultPage.getTotalPages(), size));

    return resultPage.getContent();
}
```

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.

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, size);
4.      linkHeader.add(createLinkHeader(uriForNextPage, "next"));
5.  }
```

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

```
1.  String constructNextPageUri(UriComponentsBuilder uriBuilder, int page, int
2.  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.

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):

```
@Test
public void whenResourcesAreRetrievedPaged_then200IsReceived(){
    Response response = RestAssured.get(paths.getFooURL() +
"?page=0&size=2");

    assertThat(response.getStatusCode(), is(200));
}
@Test
public void whenPageOfResourcesAreRetrievedOutOfBounds_then404IsReceived(){
    String url = getFooURL() + "?page=" + randomNumeric(5) + "&size=2";
    Response response = RestAssured.get.get(url);

    assertThat(response.getStatusCode(), is(404));
}
@Test
public void givenResourcesExist_whenFirstPageIsRetrieved_
thenPageContainsResources(){
   createResource();
   Response response = RestAssured.get(paths.getFooURL() + "?page=0&size=2");

   assertFalse(response.body().as(List.class).isEmpty());
}
```

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:

```
@Test
public void whenFirstPageOfResourcesAreRetrieved_thenSecondPageIsNext(){
   Response response = RestAssured.get(getFooURL()+"?page=0&size=2");

   String uriToNextPage = extractURIByRel(response.getHeader("Link"), "next");
   assertEquals(getFooURL()+"?page=1&size=2", uriToNextPage);
}
@Test
public void whenFirstPageOfResourcesAreRetrieved_thenNoPreviousPage(){
   Response response = RestAssured.get(getFooURL()+"?page=0&size=2");

   String uriToPrevPage = extractURIByRel(response.getHeader("Link"), "prev");
   assertNull(uriToPrevPage );
}
@Test
public void whenSecondPageOfResourcesAreRetrieved_thenFirstPageIsPrevious(){
   Response response = RestAssured.get(getFooURL()+"?page=1&size=2");

   String uriToPrevPage = extractURIByRel(response.getHeader("Link"), "prev");
   assertEquals(getFooURL()+"?page=0&size=2", uriToPrevPage);
}
@Test
public void whenLastPageOfResourcesIsRetrieved_thenNoNextPageIsDiscoverable(){
   Response first = RestAssured.get(getFooURL()+"?page=0&size=2");
   String uriToLastPage = extractURIByRel(first.getHeader("Link"), "last");

   Response response = RestAssured.get(uriToLastPage);

   String uriToNextPage = extractURIByRel(response.getHeader("Link"), "next");
   assertNull(uriToNextPage);
}
```

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.

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.

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*:

```
1.  public interface SubjectRepository extends
2.  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*:

```
1.  @RestResource(path = "nameContains")
2.  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/nameContains{?name,page,size,sort}”,
3.    “templated” : true
4.  }
```

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.

The code in this section is available in the *spring-data-rest* project.

10. Conclusion

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

1. Overview

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 I/O, etc. should not be done inline but via utility methods.

```
@Test
public void givenUserDoesNotExists_whenUserInfoIsRetrieved_then404IsReceived()
  throws ClientProtocolException, IOException {

    // Given
    String name = RandomStringUtils.randomAlphabetic( 8 );
    HttpUriRequest request = new HttpGet( "https://api.github.com/users/" +
name );

    // When
    HttpResponse httpResponse = HttpClientBuilder.create().build().execute(
request );

    // Then
    assertThat(
      httpResponse.getStatusLine().getStatusCode(),
      equalTo(HttpStatus.SC_NOT_FOUND));
}
```

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.

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

```
@Test
public void
  givenUserExists_whenUserInformationIsRetrieved_
thenRetrievedResourceIsCorrect()
  throws ClientProtocolException, IOException {

    // Given
    HttpUriRequest request = new HttpGet( "https://api.github.com/users/eugenp"
);

    // When
    HttpResponse response = HttpClientBuilder.create().build().execute( request
);

    // Then
    GitHubUser resource = RetrieveUtil.retrieveResourceFromResponse(
      response, GitHubUser.class);
    assertThat( "eugenp", Matchers.is( resource.getLogin() ) );
}
```

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.

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.getEntity());
6.      ObjectMapper mapper = new ObjectMapper()
7.        .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
8.      return mapper.readValue(jsonFromResponse, clazz);
9.  }
```

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)

7. Conclusion

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.