Table of Contents

Bootstrap a Web Application with Spring 4 1
Table of Contents 1
- **1. Overview** 2
- **2. The Maven pom.xml** 2
- **3. The Java based Web Configuration** 3
- **4. Conclusion** 5

Spring Security Basic Authentication 6
1. Overview 6
- **2. The Spring Security Configuration** 6
- **3. Consuming The Secured Application** 7
- **4. Further Configuration – The Entry Point** 8
- **5. The Maven Dependencies** 8
- **6. Conclusion** 9

REST Pagination in Spring 9
Table of Contents 9
- **1. Overview** 10
- **2. Page as Resource vs Page as Representation** 10
- **3. The Controller** 10
- **4. Discoverability for REST pagination** 11
- **5. Test Driving Pagination** 12
- **6. Test Driving Pagination Discoverability** 12
- **7. Getting All Resources** 13
- **8. REST Paging with Range HTTP headers** 13
- **9. Conclusion** 13

Bootstrap a Web Application with Spring 4

Return to Content

Contents

- Table of Contents
- 1. Overview
- 2. The Maven pom.xml
- 3. The Java based Web Configuration
- 4. Conclusion

If you're new here, you may want to get my "REST APIs with Spring" eBook [https://my.leadpages.net/leadbox/146382273f72a2%3A13a71ac76b46dc/5735865741475840/]. Thanks for visiting!

Table of Contents

- **1.** Overview
- **2.** The Maven pom.xml

- **2.1.** Justification of the *cglib* dependency
- **2.2.** The *cglib* dependency in Spring 3.2 and beyond
- **3.** The Java based web configuration
- **3.1.** The web.xml
- **4.** Conclusion

1. Overview

The tutorial illustrates how to **Bootstrap a Web Application with Spring** and also discusses how to make the jump **from XML to Java** without having to completely migrate the entire XML configuration.

2. The Maven pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="
 http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>org</groupId>
 <artifactId>rest</artifactId>
 <version>0.0.1-SNAPSHOT</version>
 <packaging>war</packaging>

 <dependencies>

 <dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-webmvc</artifactId>
 <version>${spring.version}</version>
 <exclusions>
 <exclusion>
 <artifactId>commons-logging</artifactId>
 <groupId>commons-logging</groupId>
 </exclusion>
 </exclusions>
 </dependency>

 </dependencies>

 <build>
 <finalName>rest</finalName>

 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-compiler-plugin</artifactId>
 <version>3.1</version>
 <configuration>
 <source>1.6</source>
 <target>1.6</target>
 <encoding>UTF-8</encoding>
 </configuration>
 </plugin>
 </plugins>
 </build>

 <properties>
 <spring.version>4.0.5.RELEASE</spring.version>
 </properties>

</project>
```

2.1. The cglib dependency before Spring 3.2

You may wonder why *cglib* is a dependency – it turns out there is a valid reason to include it – the entire configuration cannot function without it. If removed, Spring will throw:

Caused by: java.lang.IllegalStateException: CGLIB is required to process @Configuration classes. Either add CGLIB to the classpath or remove the following @Configuration bean definitions

The reason this happens is explained by the way Spring deals with *@Configuration* classes. These classes are effectively beans, and because of this they need to be aware of the Context, and respect scope and other bean semantics. This is achieved by dynamically creating a cglib proxy with this awareness for each *@Configuration* class, hence the cglib dependency.

Also, because of this, there are a few restrictions for *Configuration* annotated classes:

- Configuration classes **should not be final**
- They should have a constructor with no arguments

2.2. The cglib dependency in Spring 3.2 and beyond

Starting with Spring 3.2, it is **no longer necessary to add cglib as an explicit dependency**. This is because Spring is in now inlining *cglib* – which will ensure that all class based proxying functionality will work out of the box with Spring 3.2.

The new cglib code is placed under the Spring package: *org.springframework.cglib* (replacing the original *net.sf.cglib*). The reason for the package change is to avoid conflicts with any *cglib* versions already existing on the classpath.

Also, the new cglib 3.0 is now used, upgraded from the older 2.2 dependency (see this JIRA issue [https://jira.springsource.org/browse/SPR-9669] for more details).

Finally, now that Spring 4.0 is out in the wild, changes like this one (removing the cglib dependency) are to be expected with Java 8 just around the corner – you can watch this Spring Jira [https://jira.springsource.org/browse/SPR-9639] to keep track of the Spring support, and the Java 8 Resources page [http://www.baeldung.com/java8] to keep tabs on the that.

3. The Java based Web Configuration

```
@Configuration
@ImportResource( { "classpath*:/rest_config.xml" } )
@ComponentScan( basePackages = "org.rest" )
@PropertySource({ "classpath:rest.properties", "classpath:web.properties" })
public class AppConfig{

 @Bean
   public static PropertySourcesPlaceholderConfigurer properties() {
   return new PropertySourcesPlaceholderConfigurer();
   }
}
```

First, the **@Configuration** annotation – this 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 [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/beans.html#beans-java].

Then, **@ImportResource** is used to import the existing XML based Spring configuration. This may be configuration which is still being migrated from XML to Java, or simply legacy configuration that you wish to keep. Either way, importing it into the Container is essential for a successful migration, allowing small steps without to much risk. The equivalent XML annotation that is replaced is:

<import resource="classpath:/rest_config.xml" />*

Moving on to **@ComponentScan** – this configures the component scanning directive, effectively replacing the XML:

```
<context:component-scan base-package="org.rest" />
```

As of Spring 3.1, the *@Configuration* are excluded from classpath scanning by default – see this JIRA issue [https://jira.springsource.org/browse/SPR-8808]. Before Spring 3.1 though, these classes should have been excluded explicitly:

```
excludeFilters = { @ComponentScan.Filter( Configuration.class ) }
```

The *@Configuration* classes should not be autodiscovered because they are already specified and used by the Container – allowing them to be rediscovered and introduced into the Spring context will result in the following error:

Caused by: org.springframework.context.annotation.ConflictingBeanDefinitionException: Annotation-specified bean name 'webConfig' for bean class [org.rest.spring.AppConfig] conflicts with existing, non-compatible bean definition of same name and class [org.rest.spring.AppConfig]

And finally, using the **@Bean** annotation to configure the **properties support** – *PropertySourcesPlaceholderConfigurer* is initialized in a *@Bean* annotated method, indicating it will produce a Spring bean managed by the Container. This new configuration has replaced the following XML:

```
<context:property-placeholder
location="classpath:persistence.properties, classpath:web.properties"
ignore-unresolvable="true"/>
```

For a more in depth discussion on why it was necessary to manually register the *PropertySourcesPlaceholderConfigurer* bean, see the Properties with Spring Tutorial [http://www.baeldung.com/2012/02/06/properties-with-spring/].

3.1. The web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="
 http://java.sun.com/xml/ns/javaee"
     xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
 http://java.sun.com/xml/ns/javaee
 http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    id="rest" version="3.0">

 <context-param>
 <param-name>contextClass</param-name>
 <param-value>
 org.springframework.web.context.support.AnnotationConfigWebApplicationContext
 </param-value>
 </context-param>
 <context-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>org.rest.spring.root</param-value>
 </context-param>
 <listener>
       <listener-class>org.springframework.web.context.ContextLoaderListener</
listener-class>
 </listener>

 <servlet>
 <servlet-name>rest</servlet-name>
 <servlet-class>
 org.springframework.web.servlet.DispatcherServlet
 </servlet-class>
 <init-param>
 <param-name>contextClass</param-name>
 <param-value>
 org.springframework.web.context.support.AnnotationConfigWebApplicationContext
 </param-value>
 </init-param>
 <init-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>org.rest.spring.rest</param-value>
 </init-param>
 <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
 <servlet-name>rest</servlet-name>
 <url-pattern>/api/*</url-pattern>
 </servlet-mapping>

 <welcome-file-list>
 <welcome-file />
 </welcome-file-list>

</web-app>
```

First, the root context is defined and configured to use *AnnotationConfigWebApplicationContext* instead of the default *XmlWebApplicationContext*. The newer *AnnotationConfigWebApplicationContext* accepts *@Configuration* annotated classes as input for the Container configuration and is needed in order to set up the Java based context. Unlike *XmlWebApplicationContext*, it assumes no default configuration class locations, so the *"contextConfigLocation"init-param* for the Servlet must be set. This will point to the java package where the *@Configuration* classes are located; the fully qualified name(s) of the classes are also supported.

Next, the *DispatcherServlet* is configured to use the same kind of context, with the only difference that it's loading configuration classes out of a different package.

Other than this, the *web.xml* doesn't really change from a XML to a Java based configuration.

4. Conclusion

The presented approach allows for a smooth **migration of the Spring configuration** from XML to Java, mixing the old and the new. This is important for older projects, which may have a lot of XML based configuration that cannot be migrated all at once.

This way, in a migration, the XML beans can be ported in small increments.

In the next article on REST with Spring [http://www.baeldung.com/2011/10/25/building-a-restful-web-service-with-spring-3-1-and-java-based-configuration-part-2/], I cover setting up MVC in the project, configuration of the HTTP status codes, payload marshalling and content negotiation.

The implementation of this *Bootstrap a Spring Web App Tutorial* can be downloaded as a working sample project. [https://my.leadpages.net/leadbox/147e9e473f72a2%3A13a71ac76b46dc/5745710343389184/]

This is an Eclipse based project, so it should be easy to import and run as it is.

http://twitter.com/share

java [http://www.baeldung.com/tag/java-2/], Spring [http://www.baeldung.com/tag/spring/]

Spring Security Basic Authentication

Return to Content

Contents

- 1. Overview
- 2. The Spring Security Configuration
- 3. Consuming The Secured Application
- 4. Further Configuration – The Entry Point
- 5. The Maven Dependencies
- 6. Conclusion

If you're new here, you may want to get my "REST APIs with Spring" eBook [https://my.leadpages.net/leadbox/146382273f72a2%3A13a71ac76b46dc/5735865741475840/]. Thanks for visiting!

1. Overview

This tutorial shows how to set up, configure and customize **Basic Authentication with Spring**. We're going to built on top of the simple Spring MVC example [http://www.baeldung.com/spring-mvc-tutorial], and secure the UI of the MVC application with the Basic Auth mechanism provided by Spring Security.

2. The Spring Security Configuration

The Configuration for Spring Security is still XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:beans="http://www.springframework.org/schema/beans"
 xsi:schemaLocation="
 http://www.springframework.org/schema/security
 http://www.springframework.org/schema/security/spring-security-3.1.xsd
 http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">

 <http use-expressions="true">
 <intercept-url pattern="/**" access="isAuthenticated()" />

 <http-basic />
 </http>

 <authentication-manager>
 <authentication-provider>
 <user-service>
 <user name="user1" password="user1Pass" authorities="ROLE_USER" />
 </user-service>
 </authentication-provider>
 </authentication-manager>

</beans:beans>
```

This is one of the last pieces of configuration in Spring that still need XML – Java Configuration for Spring Security [https://github.com/SpringSource/spring-security-javaconfig] is still a work in progress.

What is relevant here is the *<http-basic>* element inside the main *<http>* element of the configuration – this is enough to enable Basic Authentication for the entire application. The Authentication Manager is not the focus of this tutorial, so we are using an in memory manager with the user and password defined in plaintext.

The *web.xml* of the web application enabling Spring Security has already been discussed in the Spring Logout tutorial [http://www.baeldung.com/spring-security-login#web_xml].

3. Consuming The Secured Application

The *curl* command is our go to tool for consuming the secured application.

First, let’s try to request the */homepage.html* without providing any security credentials:

```
curl -i http://localhost:8080/spring-security-mvc-basic-auth/homepage.html
```

We get back the expected *401 Unauthorized* and the Authentication Challenge [http://tools.ietf.org/html/rfc1945#section-10.16]:

```
HTTP/1.1 401 Unauthorized
Server: Apache-Coyote/1.1
Set-Cookie:     JSESSIONID=E5A8D3C16B65A0A007CFAACAEEE6916B;     Path=/spring-
security-mvc-basic-auth/; HttpOnly
WWW-Authenticate: Basic realm="Spring Security Application"
Content-Type: text/html;charset=utf-8
Content-Length: 1061
Date: Wed, 29 May 2013 15:14:08 GMT
```

The browser would interpret this challenge and prompt us for credentials with a simple dialog, but since we’re using *curl*, this isn’t the case.

Now, let’s request the same resource – the homepage – but **provide the credentials** to access it as well:

```
curl -i --user user1:user1Pass http://localhost:8080/spring-security-mvc-basic-
auth/homepage.html
```

Now, the response from the server is *200 OK* along with a *Cookie*:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie:     JSESSIONID=301225C7AE7C74B0892887389996785D;     Path=/spring-
security-mvc-basic-auth/; HttpOnly
Content-Type: text/html;charset=ISO-8859-1
Content-Language: en-US
Content-Length: 90
Date: Wed, 29 May 2013 15:19:38 GMT
```

From the browser, the application can be consumed normally – the only difference is that a login page is no longer a hard requirement since all browsers support Basic Authentication and use a dialog to prompt the user for credentials.

4. Further Configuration – The Entry Point

By default, the *BasicAuthenticationEntryPoint* provisioned by Spring Security returns a full html page for a *401 Unauthorized* response back to the client. This html representation of the error renders well in a browser, but it not well suited for other scenarios, such as a REST API where a json representation may be preferred.

The namespace is flexible enough for this new requirement as well – to address this – the entry point can be overridden:

```
<http-basic entry-point-ref="myBasicAuthenticationEntryPoint" />
```

The new entry point is defined as a standard bean:

```
@Component
public         class         MyBasicAuthenticationEntryPoint         extends
 BasicAuthenticationEntryPoint {

 @Override
 public void commence
     (HttpServletRequest      request,      HttpServletResponse      response,
 AuthenticationException authEx)
 throws IOException, ServletException {
  response.addHeader("WWW-Authenticate", "Basic realm=\"" + getRealmName() +
 "\"");
 response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
 PrintWriter writer = response.getWriter();
 writer.println("HTTP Status 401 - " + authEx.getMessage());
 }

 @Override
 public void afterPropertiesSet() throws Exception {
 setRealmName("Baeldung");
 super.afterPropertiesSet();
 }
}
```

By writing directly to the HTTP Response we now have full control over the format of the response body.

5. The Maven Dependencies

The Maven dependencies for Spring Security have been discussed before in the Spring Security with Maven article [http://www.baeldung.com/spring-security-with-maven] – we will need both *spring-security-web* and *spring-security-config* available at runtime.

6. Conclusion

In this example we secured an MVC application with Spring Security and Basic Authentication. We discussed the XML configuration and we consumed the application with simple curl commands. Finally took control of the exact error message format – moving from the standard HTML error page to a custom text or json format.

The implementation of this Spring tutorial can be found in the github project [https://github.com/eugenp/tutorials/tree/master/spring-security-basic-auth#readme] – this is an Eclipse based project, so it should be easy to import and run as it is. When the project runs locally, the sample html can be accessed at:

http://localhost:8080/spring-security-mvc-basic-auth/homepage.html

http://twitter.com/share

security [http://www.baeldung.com/tag/security/], Spring [http://www.baeldung.com/tag/spring/]

REST Pagination in Spring

Return to Content

Contents

• Table of Contents

• 1. Overview

• 2. Page as Resource vs Page as Representation

• 3. The Controller

• 4. Discoverability for REST pagination

• 5. Test Driving Pagination

• 6. Test Driving Pagination Discoverability

• 7. Getting All Resources

• 8. REST Paging with Range HTTP headers

• 9. Conclusion

If you're new here, you may want to get my "REST APIs with Spring" eBook [https://my.leadpages.net/leadbox/146382273f72a2%3A13a71ac76b46dc/5735865741475840/]. Thanks for visiting!

Table of Contents

• **1.** Overview

• **2.** Page as Resource vs Page as Representation

• **3.** The Controller

- **4.** Discoverability for REST pagination
- **5.** Test Driving Pagination
- **6.** Test Driving Pagination Discoverability
- **7.** Getting All Resources
- **8.** REST Paging with *Range* HTTP headers
- **9.** Conclusion

1. Overview

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

3. The Controller

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

```
@RequestMapping( value = "admin/foo",params = { "page", "size" },method = GET )
@ResponseBody
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 ResourceNotFoundException();
 }
 eventPublisher.publishEvent( new PaginatedResultsRetrievedEvent< Foo >
 ( Foo.class, uriBuilder, response, page, resultPage.getTotalPages(), size ) );
 return resultPage.getContent();
}
```

The two query parameters are injected into the Controller method via *@RequestParam.*

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

Finally – note that the focus of this article is only the REST and the web layer – to go deeper into the data access part of pagination you can check out this article [http://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-seven-pagination/] about Pagination with Spring Data.

4. Discoverability for REST pagination

Withing 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 official [http://www.iana.org/assignments/link-relations/link-relations.xml] "***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, it should be dealt with separately and decoupled from the main Controller flow.

With Spring, this **decoupling is done with Events**, as was thoroughly discussed in the previous article focusing on Discoverability [http://www.baeldung.com/2011/11/13/rest-service-discoverability-with-spring-part-5/] of a REST Service. In the case of pagination, the event – *PaginatedResultsRetrievedEvent* – is fired in the controller layer, and discoverability is implemented with a custom listener for this event:

```
void addLinkHeaderOnPagedResourceRetrieval(
 UriComponentsBuilder uriBuilder, HttpServletResponse response,
 Class clazz, int page, int totalPages, int size ){

 String resourceName = clazz.getSimpleName().toString().toLowerCase();
 uriBuilder.path( "/admin/" + resourceName );

 StringBuilder linkHeader = new StringBuilder();
 if( hasNextPage( page, totalPages ) ){
 String uriNextPage = constructNextPageUri( uriBuilder, page, size );
 linkHeader.append( createLinkHeader( uriNextPage, "next" ) );
 }
 if( hasPreviousPage( page ) ){
 String uriPrevPage = constructPrevPageUri( uriBuilder, page, size );
 appendCommaIfNecessary( linkHeader );
 linkHeader.append( createLinkHeader( uriPrevPage, "prev" ) );
 }
 if( hasFirstPage( page ) ){
 String uriFirstPage = constructFirstPageUri( uriBuilder, size );
 appendCommaIfNecessary( linkHeader );
 linkHeader.append( createLinkHeader( uriFirstPage, "first" ) );
 }
 if( hasLastPage( page, totalPages ) ){
 String uriLastPage = constructLastPageUri( uriBuilder, totalPages, size );
 appendCommaIfNecessary( linkHeader );
 linkHeader.append( createLinkHeader( uriLastPage, "last" ) );
 }
 response.addHeader( "Link", linkHeader.toString() );
}
```

In short, the listener checks if the navigation allows for a *next*, *previous*, *first* and *last* pages and – if it does – **adds the relevant URIs to the Link HTTP Header**.

Note that, for brevity, I included only a partial code sample and the full code here [https://gist.github.com/1622997].

5. Test Driving Pagination

Both the main logic of pagination and discoverability are covered by small, focused integration tests; as in the previous article [http://www.baeldung.com/2011/11/06/restful-web-service-discoverability-part-4/], the rest-assured library [http://code.google.com/p/rest-assured/] is used to consume the REST service and to verify the results.

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

```
@Test
public void whenResourcesAreRetrievedPaged_then200IsReceived(){
 Response response = givenAuth().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 = givenAuth().get(url);

 assertThat( response.getStatusCode(), is( 404 ) );
}
@Test
public                                                                  void
 givenResourcesExist_whenFirstPageIsRetrieved_thenPageContainsResources(){
 createResource();

 Response response = givenAuth().get( paths.getFooURL() + "?page=0&size=2" );

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

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

 Response response = givenAuth().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 [https://gist.github.com/eugenp/8269915].

7. Getting All Resources

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 article.

9. Conclusion

This tutorial 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 [http://www.baeldung.com/jpa-pagination] or Hibernate [http://www.baeldung.com/hibernate-pagination] pagination tutorials.

The implementation of all these examples and code snippets **can be found in my github project** – this is an Eclipse based project, so it should be easy to import and run as it is.

http://twitter.com/share

HATEOAS [http://www.baeldung.com/tag/hateoas/], java [http://www.baeldung.com/tag/java-2/], REST [http://www.baeldung.com/tag/rest/], testing [http://www.baeldung.com/tag/testing/]