Building a REST API with Spring 4

TABLE OF CONTENTS

1: BOOTSTRAP A WEB APPLICATION WITH SPRING 4

2: BUILD A REST API WITH SPRING 4 AND JAVA CONFIG

7: HTTP MESSAGE CONVERTERS WITH THE SPRING FRAMEWORK

8: REST API DISCOVERABILITY AND HATEOAS

1: BOOTSTRAP A WEB APPLICATION WITH SPRING 4

1. Overview

This section 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 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 to keep track of the Spring support, and the Java 8 Resources page 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.

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. 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.

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 section on REST with Spring, I cover setting up MVC in the project, configuration of the HTTP status codes, payload marshalling and content negotiation.

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

2: BUILD A REST API WITH SPRING 4 AND JAVA CONFIG

1. Overview

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

2. Understanding REST in Spring

The Spring framework supports 2 ways of creating RESTful services:

- using MVC with ModelAndView
- using HTTP message converters

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

The new approach, based on ***HttpMessageConverter*** and annotations, is much more lightweight and easy to implement. Configuration is minimal and it provides sensible defaults for what you would expect from a RESTful service. It is however newer and a a bit on the light side concerning documentation; what's , the reference doesn't go out of it's way to make the distinction and the tradeoffs between the two approaches as clear as they should be. Nevertheless, this is the way RESTful services should be build after Spring 3.0.

3. The Java Configuration

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

The new ***@EnableWebMvc*** annotation does a number of useful things – specifically, in the case of REST, it detect 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.

4. Testing the Spring Context

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

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( classes = { ApplicationConfig.class, PersistenceConfig.class },
 loader = AnnotationConfigContextLoader.class )
public class SpringTest{

  @Test
  public void whenSpringContextIsInstantiated_thenNoExceptions(){
    // When
  }
}
```

The Java configuration classes are simply specified with the ***@ContextConfiguration*** annotation and 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.

5. The Controller

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

```
@Controller
@RequestMapping( value = "/foos" )
class FooController{

  @Autowired
  IFooService service;

  @RequestMapping( method = RequestMethod.GET )
  @ResponseBody
  public List< Foo > findAll(){
    return service.findAll();
  }

  @RequestMapping( value = "/{id}", method = RequestMethod.GET )
  @ResponseBody
  public Foo findOne( @PathVariable( "id" ) Long id ){
    return RestPreconditions.checkFound( service.findOne( id ) );
  }

  @RequestMapping( method = RequestMethod.POST )
  @ResponseStatus( HttpStatus.CREATED )
  @ResponseBody
  public Long create( @RequestBody Foo resource ){
    Preconditions.checkNotNull( resource );
    return service.create( resource );
  }

  @RequestMapping( value = "/{id}", method = RequestMethod.PUT )
  @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 );
  }

  @RequestMapping( value = "/{id}", method = RequestMethod.DELETE )
  @ResponseStatus( HttpStatus.OK )
  public void delete( @PathVariable( "id" ) Long id ){
    service.deleteById( id );
  }

}
```

You may have noticed I'm using a very simple, guava style *RestPreconditions* utility:

```
public class RestPreconditions {
   public static <T> T checkFound(final T resource) {
      if (resource == null) {
         throw new MyResourceNotFoundException();
      }
      return resource;
   }
}
```

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 *DispathcerServlet*) and simply delegate them forward to a service layer. If there is 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 are used to 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. 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.

6. Mapping the HTTP response codes

The status codes of the HTTP response are one of the most important parts of the REST service, and the subject can quickly become very complex. 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 is also good practice to include the ***Allow* HTTP header** when returning a 405 to the client, in order to specify which operations **are** allowed. This is the standard behavior of Spring MVC and does not 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 is because of this that 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 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( value = HttpStatus.BAD_REQUEST )
public class BadRequestException extends RuntimeException{
  //
}
@ResponseStatus( value = 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 exist, 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 is defined, not to the entire Spring

Container, which means that it needs to be declared in each controller individually. This quickly becomes cumbersome, especially in more complex applications which many controllers.
There are a few **JIRA issues** opened with Spring at this time to handle this and other related limitations: SPR-8124, SPR-7278, SPR-8406.

7. Additional Maven dependencies

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>${jackson.version}</version>
  </dependency>
  <dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>${jaxb-api.version}</version>
    <scope>runtime</scope>
  </dependency>
</dependencies>

<properties>
  <jackson.version>2.4.0</jackson.version>
  <jaxb-api.version>2.2.11</jaxb-api.version>
</properties>
```

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

8. Conclusion

This section covered the configuration and implementation of a REST Service using Spring 4 and Java based configuration, discussing HTTP response codes, basic Content Negotiation and marshaling.

In the next sections of the series I will focus on Discoverability of the API, advanced **content negotiation** and working with **additional representations** of a Resource.

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

3: SPRING SECURITY FOR A REST API

1. Overview

This section shows how to **Secure a REST Service using Spring and Spring Security 3.1** with Java based configuration. We will focus on how to set up the Security Configuration specifically for the REST API using a Login and Cookie approach.

2. Spring Security in the *web.xml*

The architecture of Spring Security is based entirely on Servlet Filters and, as such, comes before Spring MVC in regards to the processing of HTTP requests. Keeping this in mind, to begin with, a **filter** needs to be declared in the web.xml of the application:

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

The filter must necessarily be named '*springSecurityFilterChain*' to match the default bean created by Spring Security in the container.

Note that the defined filter is not the actual class implementing the security logic but a *DelegatingFilterProxy* with the purpose of delegating the Filter's methods to an internal bean. This is done so that the target bean can still benefit from the Spring context lifecycle and flexibility.

The URL pattern used to configure the Filter is **/*** even though the entire web service is mapped to **/api/*** so that the security configuration has the option to secure other possible mappings as well, if required.

3. The Security Configuration

```
<?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"
  xmlns:sec="http://www.springframework.org/schema/security"
  xsi:schemaLocation="
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.2.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">

  <http entry-point-ref="restAuthenticationEntryPoint">
    <intercept-url pattern="/api/admin/**" access="ROLE_ADMIN"/>

    <form-login
      authentication-success-handler-ref="mySuccessHandler"
      authentication-failure-handler-ref="myFailureHandler"
    />

    <logout />
  </http>

  <beans:bean id="mySuccessHandler"
    class="org.rest.security.MySavedRequestAwareAuthenticationSuccessHandler"/>
  <beans:bean id="myFailureHandler"
    class="org.springframework.security.web.authentication.SimpleUrlAuthenticationFailureHandler"/>

  <authentication-manager alias="authenticationManager">
    <authentication-provider>
      <user-service>
        <user name="temporary" password="temporary" authorities="ROLE_ADMIN"/>
        <user name="user" password="user" authorities="ROLE_USER"/>
      </user-service>
    </authentication-provider>
  </authentication-manager>

</beans:beans>
```

Most of the configuration is done using the **security namespace** – for this to be enabled, the schema locations must be defined and pointed to the correct 3.1 or 3.2 XSD versions. The namespace is designed so that it expresses the common uses of Spring Security while still providing hooks raw beans to accommodate more advanced scenarios.

3.1. The *<http>* element

The *<http>* element is the main container element for HTTP security configuration. In the current implementation, it only secured a single mapping: */api/admin/***. Note that the mapping is **relative to the root context** of the web application, not to the rest Servlet; this is because the entire security configuration lives in the root Spring context and not in the child context of the Servlet.

3.2. The Entry Point

In a standard web application, the authentication process may be automatically triggered when the client tries to access a secured resource without being authenticated – this is usually done by redirecting to a login page so that the user can enter credentials. However, for a **REST Web Service** this behavior doesn't make much sense – Authentication should only be done by a request to the correct URI and all other requests should simply fail with a **401 UNAUTHORIZED** status code if the user is not authenticated.

Spring Security handles this automatic triggering of the authentication process with the concept of an **Entry Point** – this is a required part of the configuration, and can be injected via the *entry-point-ref* attribute of the *<http>* element. Keeping in mind that this functionality doesn't make sense in the context of the REST Service, the new custom entry point is defined to simply return 401 whenever it is triggered:

```
@Component( “restAuthenticationEntryPoint” )
public class RestAuthenticationEntryPoint implements AuthenticationEntryPoint{

   @Override
   public void commence( HttpServletRequest request, HttpServletResponse response,
    AuthenticationException authException ) throws IOException{
      response.sendError( HttpServletResponse.SC_UNAUTHORIZED, “Unauthorized” );
   }
}
```

A quick sidenote here is that the 401 is sent without the *WWW-Authenticate* header, as required by the HTTP Spec – we can of course set the value manually if we need to.

3.3. The Login Form for REST

There are multiple ways to do Authentication for a REST API – one of the default Spring Security provides is **Form Login** – which uses an authentication processing filter – *org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter.*

The *<form-login>* element will create this filter and will also allow us to set our custom authentication success handler on it. This can also be done manually by using the *<custom-filter>* element to register a filter at the position *FORM_LOGIN_FILTER* – but the namespace support is flexible enough.

Note that for a standard web application, the ***auto-config*** attribute of the *<http>* element is shorthand syntax for some useful security configuration. While this may be appropriate for some very simple configurations, it doesn't fit and should not be used for a REST API.

3.4. Authentication should return 200 instead of 301

By default, form login will answer a successful authentication request with a **301 MOVED PERMANENTLY** status code; this makes sense in the context of an actual login form which needs to redirect after login. For a RESTful web service however, the desired response for a successful authentication should be **200 OK**.

This is done by injecting a **custom authentication success handler** in the form login filter, to replace the default one. The new handler implements the exact same login as the default *org.springframework.security.web.authentication.SavedRequestAwareAuthenticationSuccessHandler* with one notable difference – the redirect logic is removed:

```
public class MySavedRequestAwareAuthenticationSuccessHandler
     extends SimpleUrlAuthenticationSuccessHandler {

    private RequestCache requestCache = new HttpSessionRequestCache();

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response,
     Authentication authentication) throws ServletException, IOException {
        SavedRequest savedRequest = requestCache.getRequest(request, response);

        if (savedRequest == null) {
            clearAuthenticationAttributes(request);
            return;
        }
        String targetUrlParam = getTargetUrlParameter();
        if (isAlwaysUseDefaultTargetUrl() ||
          (targetUrlParam != null &&
          StringUtils.hasText(request.getParameter(targetUrlParam)))) {
            requestCache.removeRequest(request, response);
            clearAuthenticationAttributes(request);
            return;
        }

        clearAuthenticationAttributes(request);
    }

    public void setRequestCache(RequestCache requestCache) {
        this.requestCache = requestCache;
    }
}
```

3.5. Failed Authentication should return 401 instead of 302

Similarly – we configured the authentication failure handler – same way we did with the success handler.

Luckily – in this case, we don't need to actually define a new class for this handler – the standard implementation – *SimpleUrlAuthenticationFailureHandler* – does just fine.

The only difference is that – now that we're defining this explicitly in our XML config – it's **not going to get a default *defaultFailure*Url from Spring** – and so it won't redirect.

3.6. The Authentication Manager and Provider

The authentication process uses an **in-memory provider** to perform authentication – this is meant to simplify the configuration as a production implementation of these artifacts is outside the scope of this post.

3.7 Finally – Authentication against the running REST Service

Now let’s see how we can authenticate against the REST API – the URL for login is */j_spring_security_check* – and a simple *curl* command performing login would be:

```
curl -i -X POST -d j_username=user -d j_password=userPass
http://localhost:8080/spring-security-rest/j_spring_security_check
```

This request will return the Cookie which will then be used by any subsequent request against the REST Service.

We can use curl to authentication and **store the cookie it receives in a file:**

```
curl -i -X POST -d j_username=user -d j_password=userPass -c /opt/cookies.txt
http://localhost:8080/spring-security-rest/j_spring_security_check
```

Then **we can use the cookie from the file** to do further authenticated requests:

```
curl -i --header “Accept:application/json” -X GET -b /opt/cookies.txt
http://localhost:8080/spring-security-rest/api/foos
```

This authenticated request will correctly **result in a 200 OK**:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 24 Jul 2013 20:31:13 GMT

[{“id”:0,”name”:”JbidXc”}]
```

4. Maven and other trouble

The Spring core dependencies necessary for a web application and for the REST Service have been discussed in detail. For security, we'll need to add: *spring-security-web* and *spring-security-config* – all of these have also been covered in the Maven for Spring Security tutorial.

It's worth paying close attention to the way Maven will resolve the older Spring dependencies – the resolution strategy will start causing problems once the security artifacts are added to the pom. To address this problem, some of the core dependencies will need to be overridden in order to keep them at the right version.

5. Conclusion

This section covered the basic security configuration and implementation for a RESTful Service using **Spring Security 3.1**, discussing the *web.xml*, the security configuration, the HTTP status codes for the authentication process and the Maven resolution of the security artifacts.

4: SPRING SECURITY BASIC AUTHENTICATION

1. Overview

This section shows how to set up, configure and customize **Basic Authentication with Spring**. We're going to built on top of this simple Spring MVC example, 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>
```

There is of course the option of Java based configuration as well.

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 section, 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 previous section.

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/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 – we will need both *spring-security-web* and *spring-security-config* available at runtime.

6. Conclusion

In this example we secured the existing REST 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 section can be found in the github project – 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

5: SPRING SECURITY DIGEST AUTHENTICATION

1. Overview

This section illustrates how to set up, configure and customize Digest Authentication with Spring. Similar to the previous section about Basic Authentication, we're going to built on top of the Spring REST Service from the previous sections, and secure the application with the Digest Auth mechanism provided by Spring Security.

2. The Security XML Configuration

First thing to understand about the configuration is that, while Spring Security does have full out of the box support for the Digest authentication mechanism, this support is **not as well integrated into the namespace** as Basic Authentication was.

In this case, we need to manually **define the raw beans** that are going to make up the security configuration – the *DigestAuthenticationFilter* and *the DigestAuthenticationEntryPoint*:

```
<?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">
```

```
  <beans:bean id="digestFilter"
    class="org.springframework.security.web.authentication.www.DigestAuthenticationFilter">
      <beans:property name="userDetailsService" ref="userService" />
      <beans:property name="authenticationEntryPoint" ref="digestEntryPoint" />
  </beans:bean>
  <beans:bean id="digestEntryPoint"
    class="org.springframework.security.web.authentication.www.DigestAuthenticationEntryPoint">
      <beans:property name="realmName" value="Contacts Realm via Digest Authentication" />
      <beans:property name="key" value="acegi" />
  </beans:bean>

  <!-- the security namespace configuration -->
  <http use-expressions="true" entry-point-ref="digestEntryPoint">
      <intercept-url pattern="/**" access="isAuthenticated()" />

      <custom-filter ref="digestFilter" after="BASIC_AUTH_FILTER" />
  </http>

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

</beans:beans>
```

Next, we need to integrate these beans into the overall security configuration – and in this case, the namespace is still flexible enough to allow us to do that.

The first part of this is pointing to the custom entry point bean, via the entry-point-ref attribute of the main *<http>* element.

The second part is **adding the newly defined digest filter into the security filter chain**. Since this filter is functionally equivalent to the *BasicAuthenticationFilter*, we are using the same relative position in the chain – this is specified by the *BASIC_AUTH_FILTER* alias in the overall Spring Security Standard Filters.

Finally, notice that the Digest Filter is configured to **point to the user service bean** – and here, the namespace is again very useful as it allows us to specify a bean name for the default user service created by the *<user-service>* element:

```
<user-service id="userService">
```

3. Consuming the Secured Application

We're going to be using **the *curl* command** to consume the secured application and understand how a client can interact with it.

Let's start by requesting the homepage – **without providing security credentials** in the request:

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

As expected, we get back a response with a 401 Unauthorized status code:

```
HTTP/1.1 401 Unauthorized
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=CF0233C...; Path=/spring-security-mvc-digest-auth/; HttpOnly
WWW-Authenticate: Digest realm="Contacts Realm via Digest Authentication", qop="auth",
  nonce="MTM3MzYzODE2NTg3OTo3MmYxN2JkOWYxZTc4MzdmMzBiN2Q0YmY0ZTU0N2RkZg=="
Content-Type: text/html;charset=utf-8
Content-Length: 1061
Date: Fri, 12 Jul 2013 14:04:25 GMT
```

If this request were sent by the browser, the authentication challenge would prompt the user for credentials using a simple user/password dialog.

Let's now **provide the correct credentials** and send the request again:

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

Notice that we are enabling Digest Authentication for the *curl* command via the *–digest* flag.

The first response from the server will be the same – the *401 Unauthorized* – but the challenge will now be interpreted and acted upon by a second request – which will succeed with a *200 OK*:

```
HTTP/1.1 401 Unauthorized
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=A961E0D...; Path=/spring-security-mvc-digest-auth/; HttpOnly
WWW-Authenticate: Digest realm="Contacts Realm via Digest Authentication", qop="auth",
  MTM3MzYzODgyOTczMTo3YjM4OWQzMGU0YTgwZDg0YmYwZjRlZWJjMDQzZWZkOA=="
Content-Type: text/html;charset=utf-8
Content-Length: 1061
Date: Fri, 12 Jul 2013 14:15:29 GMT

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=55F996B...; Path=/spring-security-mvc-digest-auth/; HttpOnly
Content-Type: text/html;charset=ISO-8859-1
Content-Language: en-US
Content-Length: 90
Date: Fri, 12 Jul 2013 14:15:29 GMT

<html>
<head></head>

<body>
  <h1>This is the homepage</h1>
</body>
</html>
```

A final note on this interaction is that a client can **preemptively send the correct *Authorization* header** with the first request, and thus entirely avoid the server security challenge and the second request.

4. The Maven Dependencies

The security dependencies are discussed in depth in the Spring Security Maven tutorial. In short, we will need to define *spring-security-web* and *spring-security-config* as dependencies in our *pom.xml*.

5. Conclusion

In this section we introduce security into a simple Spring REST API by leveraging the Digest Authentication support in the framework.

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

When the project runs locally, the homepage html can be accessed at (or, with minimal Tomcat configuration, on port 80):

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

Finally, in the next section we'll see that there is no reason an application needs to choose between Basic and Digest authentication – **both can be set up simultaneously on the same URI structure**, in such a way that the client can pick between the two mechanisms when consuming the web application.

6: BASIC AND DIGEST AUTHENTICATION FOR A REST SERVICE WITH SPRING SECURITY

1. Overview

This section discusses how to **set up both Basic and Digest Authentication on the same URI structure of a REST API**. In the two previous sections, we discussed another method of securing the REST Service – form based authentication, so Basic and Digest authentication is the natural alternative, as well as the more RESTful one.

2. Configuration of Basic Authentication

The main reason that form based authentication is not ideal for a RESTful Service is that Spring Security will **make use of Sessions** – this is of course state on the server, so **the statelessness constraints in REST** is practically ignored.

We'll start by setting up Basic Authentication – first we remove the old custom entry point and filter from the main *<http>* security element:

```
<http create-session="stateless">
   <intercept-url pattern="/api/admin/**" access="ROLE_ADMIN" />

   <http-basic />
</http>
```

Note how support for basic authentication has been added with a single configuration line – *<http-basic />* – which handles the creation and wiring of both the *BasicAuthenticationFilter* and the *BasicAuthenticationEntryPoint*.

2.1. Satisfying the stateless constraint – getting rid of sessions

One of the main constraints of the RESTful architectural style is that the client-server communication is fully stateless, as the original dissertation reads:

5.1.3 Stateless
We next add a constraint to the client-server interaction: communication must be stateless in nature, as in the client-stateless-server (CSS) style of Section 3.4.3 (Figure 5-3), such that each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. ***Session state is therefore kept entirely on the client.***

The concept of **Session** on the server is one with a long history in Spring Security, and removing it entirely has been difficult until now, especially when configuration was done by using the namespace. However, Spring Security 3.1 augments the namespace configuration with a **new *stateless* option** for session creation, which effectively guarantees that no session will be created or used by Spring. What this new option does is completely removes all session related filters from the security filter chain, ensuring that authentication is performed for each request.

3. Configuration of Digest Authentication

Starting with the previous configuration, the filter and entry point necessary to set up digest authentication will be defined as beans. Then, the **digest entry point** will override the one created by *<http-basic>* behind the scenes. Finally, the custom **digest filter** will be introduced in the security filter chain using the after semantics of the security namespace to position it directly after the basic authentication filter.

```
<http create-session=”stateless” entry-point-ref=”digestEntryPoint”>
  <intercept-url pattern=”/api/admin/**” access=”ROLE_ADMIN” />

  <http-basic />
  <custom-filter ref=”digestFilter” after=”BASIC_AUTH_FILTER” />
</http>

<beans:bean id=”digestFilter” class=
 “org.springframework.security.web.authentication.www.DigestAuthenticationFilter”>
  <beans:property name=”userDetailsService” ref=”userService” />
  <beans:property name=”authenticationEntryPoint” ref=”digestEntryPoint” />
</beans:bean>

<beans:bean id=”digestEntryPoint” class=
 “org.springframework.security.web.authentication.www.DigestAuthenticationEntryPoint”>
  <beans:property name=”realmName” value=”Contacts Realm via Digest Authentication”/>
  <beans:property name=”key” value=”acegi” />
</beans:bean>

<authentication-manager>
  <authentication-provider>
    <user-service id=”userService”>
      <user name=”eparaschiv” password=”eparaschiv” authorities=”ROLE_ADMIN” />
      <user name=”user” password=”user” authorities=”ROLE_USER” />
    </user-service>
  </authentication-provider>
</authentication-manager>
```

Unfortunately there is no support in the security namespace to automatically configure the digest authentication the way basic authentication can be configured with *<http-basic>*. Because of that, the necessary beans had to be defined and wired manually into the security configuration.

4. Supporting both authentication protocols in the same RESTful service

Basic or Digest authentication alone can be easily implemented in Spring Security 3.x; it is supporting both of them for the same RESTful web service, on the same URI mappings that introduces a new level of complexity into the configuration and testing of the service.

4.1. Anonymous request

With both basic and digest filters in the security chain, the way a **anonymous request** – a request containing no authentication credentials (*Authorization* HTTP header) – is processed by Spring Security is – the two authentication filters will find **no credentials** and will continue execution of the filter chain. Then, seeing how the request wasn't authenticated, an *AccessDeniedException* is thrown and caught in the *ExceptionTranslationFilter*, which commences the digest entry point, prompting the client for credentials.

The responsibilities of both the basic and digest filters are very narrow – they will continue to execute the security filter chain if they are unable to identify the type of authentication credentials in the request. It is because of this that Spring Security can have the flexibility to be configured with support for multiple authentication protocols on the same URI.

When a request is made containing the correct authentication credentials – either basic or digest – that protocol will be rightly used. However, for an anonymous request, the client will get prompted only for digest authentication credentials. This is because the digest entry point is configured as the main and single entry point of the Spring Security chain; as such **digest authentication can be considered the default.**

4.2. Request with authentication credentials

A **request with credentials** for Basic authentication will be identified by the *Authorization* header starting with the prefix *"Basic"*. When processing such a request, the credentials will be decoded in the basic authentication filter and the request will be authorized. Similarly, a request with credentials for Digest authentication will use the prefix *"Digest"* for it's *Authorization* header.

5. Testing both scenarios

The tests will consume the REST service by creating a new resource after authenticating with either basic or digest:

```
@Test
public void givenAuthenticatedByBasicAuth_whenAResourceIsCreated_then201IsReceived(){
  // Given
  // When
  Response response = given()
   .auth().preemptive().basic( ADMIN_USERNAME, ADMIN_PASSWORD )
   .contentType( HttpConstants.MIME_JSON ).body( new Foo( randomAlphabetic( 6 ) ) )
   .post( paths.getFooURL() );

  // Then
  assertThat( response.getStatusCode(), is( 201 ) );
}
@Test
public void givenAuthenticatedByDigestAuth_whenAResourceIsCreated_then201IsReceived(){
  // Given
  // When
  Response response = given()
   .auth().digest( ADMIN_USERNAME, ADMIN_PASSWORD )
   .contentType( HttpConstants.MIME_JSON ).body( new Foo( randomAlphabetic( 6 ) ) )
   .post( paths.getFooURL() );

  // Then
  assertThat( response.getStatusCode(), is( 201 ) );
}
```

Note that the test using basic authentication adds credentials to the request preemptively, regardless if the server has challenged for authentication or not. This is to ensure that the server doesn't need to challenge the client for credentials, because if it did, the challenge would be for Digest credentials, since that is the default.

6. Conclusion

This section covered the configuration and implementation of both Basic and Digest authentication for a RESTful service, using mostly Spring Security 3.0 namespace support as well as some new features added by Spring Security 3.1.

For the full implementation, check out the github project.

7: HTTP MESSAGE CONVERTERS WITH THE SPRING FRAMEWORK

1. Overview

This section describes **how to Configure *HttpMessageConverter* in Spring.**

Simply put, message converters are used to marshall and unmarshall Java Objects to and from JSON, XML, etc – over HTTP.

2. The Basics

2.1. Enable Web MVC

The Web Application needs to be **configured with Spring MVC support** – one convenient and very customizable way to do this is to use the *@EnableWebMvc* annotation:

```
@EnableWebMvc
@Configuration
@ComponentScan({ “org.baeldung.web” })
public class WebConfig extends WebMvcConfigurerAdapter {
    ...
}
```

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

2.2. The Default Message Converters

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

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

3. Client-Server Communication – JSON only

3.1. High Level Content Negotiation

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

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

It will then try to find a registered converter that is capable of handling that specific media type – and it will use it 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 should 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:

```
@RequestMapping(method=RequestMethod.GET, value=”/foos/{id}”)
public @ResponseBody Foo findById(@PathVariable long id) {
   return fooService.get(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-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

@RequestBody is used on the argument of a Controller method – it indicates to Spring **that the body of the HTTP Request is deserialized to that particular Java entity**. As discussed previously, the *“Content-Type”* header specified by the Client will be used to determine the appropriate converter for this.

Let’s look at **an example**:

```
@RequestMapping(method=RequestMethod.PUT, value=”/foos/{id}”)
public @ResponseBody void updateFoo(@RequestBody Foo foo, @PathVariable String id) {
   fooService.update(foo);
}
```

Now, let’s consume this with a JSON object – we’re specifying **“Content-Type”** to be *application/json*:

```
curl -i -X PUT -H “Content-Type: application/json” \
-d ‘{“id”:”83”,”name”:”klik”}’ http://localhost:8080/spring-rest/foos/1
```

We get back a 200 OK – a successful response:

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

4. Custom Converters Configuration – adding XML Support

We can **customize the message converters by extending the WebMvcConfigurerAdapter class** and overriding the *configureMessageConverters* method:

```
@EnableWebMvc
@Configuration
@ComponentScan({ “org.baeldung.web” })
public class WebConfig extends WebMvcConfigurerAdapter {

   @Override
   public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
      messageConverters.add(createXmlHttpMessageConverter());
      messageConverters.add(new MappingJackson2HttpMessageConverter());

      super.configureMessageConverters(converters);
   }
```

```
    private HttpMessageConverter<Object> createXmlHttpMessageConverter() {
        MarshallingHttpMessageConverter xmlConverter =
          new MarshallingHttpMessageConverter();

        XStreamMarshaller xstreamMarshaller = new XStreamMarshaller();
        xmlConverter.setMarshaller(xstreamMarshaller);
        xmlConverter.setUnmarshaller(xstreamMarshaller);

        return xmlConverter;
    }
}
```

Note that **the XStream library now needs to be present on the classpath**.

Also be aware that by extending this support class, **we are losing the default message converters which were previously pre-registered** – we only have what we define.

Let's go over this example – we are creating a new converter – the *MarshallingHttpMessageConverter* – and we're 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.

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 to – refer to Spring api documentation for full list of available marshallers.

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-rest/foos/1
```

5. Using Spring's *RestTemplate* with Http Message Converters

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 and we're going to 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

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

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 – same way we did previously – and we're going to 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);
   converters.add(marshallingConverter);
```

```
   List<HttpMessageConverter<?>> converters = new ArrayList<HttpMessageConverter<?>>();
   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:

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

    Foo resource = new Foo(4, "jason");
    HttpHeaders headers = new HttpHeaders();
    headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    headers.setContentType((MediaType.APPLICATION_XML));
    HttpEntity<Foo> entity = new HttpEntity<Foo>(resource, headers);

    ResponseEntity<Foo> response =
      restTemplate.exchange(URI, HttpMethod.PUT, entity, Foo.class, resource.getId());
    Foo fooResponse = response.getBody();

    Assert.assertEquals(resource.getId(), fooResponse.getId());
}
```

What's interesting here is that we're able to mix the media types – **we are 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.

6. Conclusion

In this section, 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.

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

8: REST API DISCOVERABILITY AND HATEOAS

1. Overview

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

2. Why make the API Discoverable

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

To understand **discoverability** you need to understand that constraint that is Hypermedia As The Engine Of Application State (HATEOAS); 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 interaction is to be **driven** by the API through the conversation itself, concretely via Hypertext, then there can be **no documentation**, as 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, which, in the case of a HTTP conversation, may be the *Link* header.

3. Discoverability Scenarios (Driven by tests)

So what does it mean for a REST service to be **discoverable**? Throughout this section, we will test individual traits of discoverability using Junit, rest-assured and Hamcrest. Since the REST Service has been previously secured, each test first need 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**; in addition, it should also help the client **discover** the valid HTTP methods that are allowed for that particular Resource, using the ***Allow*** HTTP Header in the response:

```
@Test
public void
 whenInvalidPOSTIsSentToValidURIOfResource_thenAllowHeaderListsTheAllowedActions(){
   // Given
   String uriOfExistingResource = restTemplate.createResource();

   // When
   Response res = givenAuth().post( uriOfExistingResource );

   // Then
   String allowHeader = res.getHeader( HttpHeaders.ALLOW );
   assertThat( allowHeader, AnyOf.<String> anyOf(
    containsString("GET"), containsString("PUT"), containsString("DELETE") ) );
}
```

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, using the ***Location*** HTTP Header. If the client does a GET on that URI, the resource should be available:

```
@Test
public void whenResourceIsCreated_thenUriOfTheNewlyCreatedResourceIsDiscoverable() {
   // When
   Foo newResource = new Foo(randomAlphabetic(6));
   Response createResp = givenAuth().contentType("application/json")
     .body(unpersistedResource).post(getFooURL());
   String uriOfNewResource= createResp.getHeader(HttpHeaders.LOCATION);

   // Then
   Response response = givenAuth().header(HttpHeaders.ACCEPT, "application/json")
     .get(uriOfNewResource);

   Foo resourceFromServer = response.body().as(Foo.class);
   assertThat(newResource, equalTo(resourceFromServer));
}
```

The test follows a simple scenario: a new *Foo* resource is created and the HTTP response is used to **discover the URI** where the Resource is now accessible. The tests then goes one step further and does a GET on that URI to retrieve the resource and compares it to the original, to make sure that it has been correctly persisted.

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 an resource should always include in its response the URI where to get all the resources of that type, again making 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.

4. Other potential discoverable URIs and microformats

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 is 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. Forms can also be used to achieve this.

5. Conclusion

We have 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 section 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 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.

9: HATEOAS FOR A SPRING REST SERVICE

1. Overview

This section will focus on the **implementation of discoverability in a Spring REST Service** and on satisfying the HATEOAS constraint.

2. Decouple Discoverability through events

Discoverability as a **separate aspect or concern** of the web layer should be decoupled from the controller handling the HTTP request. In order to do so, the Controller will fire off events for all the actions that require additional manipulation of the HTTP response.

First, the events:

```
public class SingleResourceRetrieved extends ApplicationEvent {
    private HttpServletResponse response;

    public SingleResourceRetrieved(Object source,
      HttpServletResponse response) {
        super(source);

        this.response = response;
    }

    public HttpServletResponse getResponse() {
        return response;
    }
}
public class ResourceCreated extends ApplicationEvent {
    private HttpServletResponse response;
    private long idOfNewResource;

    public ResourceCreated(Object source,
      HttpServletResponse response, long idOfNewResource) {
        super(source);
        this.response = response;
        this.idOfNewResource = idOfNewResource;
    }

    public HttpServletResponse getResponse() {
        return response;
    }
    public long getIdOfNewResource() {
        return idOfNewResource;
    }
}
```

Then, the Controller, with **2 simple operations –** ***find by id and create:***

```
@Controller
@RequestMapping(value = "/foos")
public class FooController {

    @Autowired
    private ApplicationEventPublisher eventPublisher;

    @Autowired
    private IFooService service;

    @RequestMapping(value = "foos/{id}", method = RequestMethod.GET)
    @ResponseBody
    public Foo findById(@PathVariable("id") Long id, HttpServletResponse response) {
        Foo resourceById = Preconditions.checkNotNull(service.findOne(id));

        eventPublisher.publishEvent(new SingleResourceRetrieved(this, response));
        return resourceById;
    }

    @RequestMapping(method = RequestMethod.POST)
    @ResponseStatus(HttpStatus.CREATED)
    public void create(@RequestBody Foo resource, HttpServletResponse response) {
        Preconditions.checkNotNull(resource);
        Long newId = service.create(resource).getId();

        eventPublisher.publishEvent(new ResourceCreated(this, response, newId));
    }
}
```

These events can then be handled by any number of ***decoupled listeners***, each focusing on it's own particular case and each moving towards satisfying the overall HATEOAS constraint.

The listeners should be the last objects in the call stack and no direct access to them is necessary; as such they are not public.

3. Make the URI of a newly created Resource discoverable

As discussed in the previous section on HATEOAS, the operation of creating a new Resource should return the URI of that resource in the ***Location*** HTTP header of the response; this is handled by a listener:

```
@Component
class ResourceCreatedDiscoverabilityListener
  implements ApplicationListener< ResourceCreated >{

    @Override
    public void onApplicationEvent( ResourceCreated resourceCreatedEvent ){
       Preconditions.checkNotNull( resourceCreatedEvent );

       HttpServletResponse response = resourceCreatedEvent.getResponse();
       long idOfNewResource = resourceCreatedEvent.getIdOfNewResource();

       addLinkHeaderOnResourceCreation( response, idOfNewResource );
   }
   void addLinkHeaderOnResourceCreation
     ( HttpServletResponse response, long idOfNewResource ){
       URI uri = ServletUriComponentsBuilder.fromCurrentRequestUri().
         path("/{idOfNewResource}").buildAndExpand(idOfNewResource).toUri();
       response.setHeader( "Location", uri.toASCIIString() );
    }
}
```

We are now making use of the *ServletUriComponentsBuilder* – this was introduced in Spring 3.1 to **help with using the current Request**. This way, we don't need to pass anything around and we can simply access this statically.

If the API would return *ResponseEntity* – we could also use the Location support introduced here.

4. Get of single Resource

On retrieving a single Resource, the client should be able to discover the URI to get all Resources of that particular type:

```
@Component
class SingleResourceRetrievedDiscoverabilityListener
 implements ApplicationListener< SingleResourceRetrieved >{

   @Override
   public void onApplicationEvent( SingleResourceRetrieved resourceRetrievedEvent ){
      Preconditions.checkNotNull( resourceRetrievedEvent );

      HttpServletResponse response = resourceRetrievedEvent.getResponse();
      addLinkHeaderOnSingleResourceRetrieval( request, response );
   }
   void addLinkHeaderOnSingleResourceRetrieval ( HttpServletResponse response ){
      String requestURL = ServletUriComponentsBuilder.fromCurrentRequestUri().
        build().toUri().toASCIIString();
      int positionOfLastSlash = requestURL.lastIndexOf( "/" );
      String uriForResourceCreation = requestURL.substring( 0, positionOfLastSlash );

      String linkHeaderValue = LinkUtil
        .createLinkHeader( uriForResourceCreation, "collection" );
      response.addHeader( LINK_HEADER, linkHeaderValue );
   }
}
```

Note that the semantics of the link relation make use of the *"collection"* relation type, specified and used in several microformats, but not yet standardized.

The ***Link*** header is one of the most used HTTP header for the purposes of discoverability. The utility to create this header is simple enough:

```
public final class LinkUtil {
   public static String createLinkHeader(final String uri, final String rel) {
      return "<" + uri + ">; rel=\"" + rel + "\"";
   }
}
```

5. Discoverability at the Root

The root is the entry point in the entire service – it is what the client comes into contact with when consuming the API for the first time. If the HATEOAS constraint is to be considered and implemented throughout, then this is the place to start. The fact that **all the main URIs of the system have to be discoverable from the root** shouldn't come as much of a surprise by this point.

Let’s now look at the controller for this:

```
@RequestMapping( value = “admin”,method = RequestMethod.GET )
@ResponseStatus( value = HttpStatus.NO_CONTENT )
public void adminRoot( HttpServletRequest request, HttpServletResponse response ){
   String rootUri = request.getRequestURL().toString();

   URI fooUri = new UriTemplate( “{rootUri}/{resource}” ).expand( rootUri, “foo” );
   String linkToFoo = LinkUtil.createLinkHeader
     ( fooUri.toASCIIString(), “collection” );
   response.addHeader( “Link”, linkToFoo );
}
```

This is of course an illustration of the concept, focusing on a single, sample URI, for Foo Resources – a real implementation should add, similarly, URIs for all the Resources published to the client.

5.1. Discoverability is not about changing URIs

This can be a controversial point – one the one hand, the purpose of HATOAS is to have the client discover the URIs of the API and not rely on hardcoded values. On the other hand – this is not how the web works: yes, URIs are discovered, but they are also bookmarked.

A subtle but important distinction is evolution of the API – the old URIs should still work, but any client that will discover the API should discover the new URIs – which allows the API to change dynamically, and good clients to work well even when the API changes.

In conclusion – just because all URIs of the RESTful web service should be considered cool URIs (and cool URIs don’t change) – that doesn’t mean that adhering to the HATEOAS constraint isn’t extremely useful when evolving the API.

6. Caveats of Discoverability

As some of the discussions around the previous articles state, the first goal of discoverability is to make minimal or no use of **documentation** and have the client learn and understand how to use the API via the responses it gets. In fact, this shouldn’t be regarded as such a far fetched ideal – it is how we consume every new web page – **without any documentation**. So, if the concept is more problematic in the context of REST, then it must be a matter of technical implementation, not of a question of whether or not it's possible.

That being said, technically, we are still far from the a fully working solution – the specification and framework support are still evolving, and because of that, some compromises may have to be made; these are nevertheless compromises and should be regarded as such.

7. Conclusion

In this section we covered the implementation of some of the traits of discoverability in the context of a RESTful Service with Spring MVC and touched on the concept of discoverability at the root.

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.

10: ETAGS FOR REST WITH SPRING

1. Overview

This section will focus on **working with ETags in Spring**, integration testing of the REST API and consumption scenarios with *curl*.

2. REST and ETags

From the official Spring documentation on ETag support:

An ETag (entity tag) is an HTTP response header returned by an HTTP/1.1 compliant web server used to determine change in content at a given URL.

ETags are used for two things – caching and conditional requests. The **ETag value can be though as a hash** computed out of the bytes of the Response body. Because a cryptographic hash function is likely used, even the smallest modification of the body will drastically change the output and thus the value of the ETag. This is only true for strong ETags – the protocol does provide a weak Etag as well.

Using an ***If-**** **header** turns a standard GET request into a **conditional GET**. The two *If-** headers that are using with ETags are "If-None-Match" and "If-Match" – each with it's own semantics as discussed later in this section.

3. Client–Server communication with *curl*

A simple Client-Server communication involving ETags can be broken down into the steps:

- **first**, the Client makes a REST API call – the Response includes the ETag header to be stored for further use:

```
curl -H “Accept: application/json” -i http://localhost:8080/rest-sec/api/resources/1
```

```
HTTP/1.1 200 OK
ETag: “f88dd058fe004909615a64f01be66a7”
Content-Type: application/json;charset=UTF-8
Content-Length: 52
```

- **next** request the Client makes to the REST API includes the *If-None-Match* request header with the ETag value from the previous step; if the Resource has not changed on the Server, the Response will contain **no body** and a status code of *304 – Not Modified:*

```
curl -H “Accept: application/json” -H ‘If-None-Match: “f88dd058fe004909615a64f01be66a7”’
 -i http://localhost:8080/rest-sec/api/resources/1
?
```

```
HTTP/1.1 304 Not Modified
ETag: “f88dd058fe004909615a64f01be66a7”
```

- **now**, before retrieving the Resource again, we will change it by performing an update:

```
curl --user admin@fake.com:adminpass -H “Content-Type: application/json” -i
  -X PUT --data ‘{ “id”:1, “name”:”newRoleName2”, “description”:”theNewDescription” }’
```

```
http://localhost:8080/rest-sec/api/resources/1
```

```
HTTP/1.1 200 OK
ETag: “d41d8cd98f00b204e9800998ecf8427e”
Content-Length: 0
```

- **finally**, we send out the the last request to retrieve the Privilege again; keep in mind that it has been updated since the last time it was retrieved, so the previous ETag value should no longer work – the response will contain the new data and a new ETag which, again, can be stored for further use:

```
curl -H "Accept: application/json" -H 'If-None-Match: "f88dd058fe004909615a64f01be66a7"' -i
```

```
http://localhost:8080/rest-sec/api/resources/1
```

```
HTTP/1.1 200 OK
ETag: "03cb37ca667706c68c0aad4cb04c3a211"
Content-Type: application/json;charset=UTF-8
Content-Length: 56
```

And there you have it – ETags in the wild and saving bandwidth.

4. ETag support in Spring

On to the Spring support – to use ETag in Spring is extremely easy to set up and completely **transparent** for the application. The support is enabled by adding a simple *Filter* in the web.xml:

```
<filter>
   <filter-name>etagFilter</filter-name>
   <filter-class>org.springframework.web.filter.ShallowEtagHeaderFilter</filter-class>
</filter>
<filter-mapping>
   <filter-name>etagFilter</filter-name>
   <url-pattern>/api/*</url-pattern>
</filter-mapping>
```

The filter is mapped on the same URI pattern as the RESTful API itself. The filter itself is the standard implementation of ETag functionality since Spring 3.0.

The implementation is a **shallow** one – the ETag is calculated based on the response, which will **save bandwidth** but **not server performance**. So, a request that will benefit from the ETag support will still be processed as a standard request, consume any resource that it would normally consume (database connections, etc) and only before having it's response returned back to the client will the ETag support kick in.

At that point the ETag will be calculated out of the Response body and set on the Resource itself; also, if the *If-None-Match* header was set on the Request, it will be handled as well.

A **deeper implementation** of the ETag mechanism could potentially provide much greater benefits – such as serving some requests from the cache and not having to perform the computation at all – but the implementation would most definitely not be as simple, nor as pluggable as the shallow approach described here.

5. Testing ETags

Let's start simple – we need to verify that the response of a simple request retrieving a single Resource will actually return the *"ETag"* header:

```
@Test
public void givenResourceExists_whenRetrievingResource_thenEtagIsAlsoReturned() {
   // Given
   String uriOfResource = createAsUri();

   // When
   Response findOneResponse = RestAssured.given().header("Accept", "application/json").get(uriOfResource);

   // Then
   assertNotNull(findOneResponse.getHeader("ETag"));
}
```

Next, we verify the happy path of the ETag behaviour – if the Request to retrieve the *Resource* from the server uses the correct *ETag* value, then the Resource is no longer returned.

```
@Test
public void givenResourceWasRetrieved_whenRetrievingAgainWithEtag_thenNotModifiedReturned() {
   // Given
   String uriOfResource = createAsUri();
   Response findOneResponse = RestAssured.given().
     header("Accept", "application/json").get(uriOfResource);
   String etagValue = findOneResponse.getHeader(HttpHeaders.ETAG);
```

```
    // When
    Response secondFindOneResponse= RestAssured.given().
      header("Accept", "application/json").headers("If-None-Match", etagValue)
      .get(uriOfResource);

    // Then
    assertTrue(secondFindOneResponse.getStatusCode() == 304);
}
```

Step by step:

- a *Resource* is first created and then retrieved – the *ETag* value is stored for further use
- a new retrieve request is sent, this time with the *"If-None-Match"* header specifying the *ETag* value previously stored
- on this second request, the server simply returns a **304 Not Modified**, since the Resource itself has indeed not beeing modified between the two retrieval operations

Finally, we verify the case where the Resource is **changed** between the first and the second retrieval requests:

```
@Test
public void
  givenResourceWasRetrievedThenModified_whenRetrievingAgainWithEtag_thenResourceIsReturned() {
    // Given
    String uriOfResource = createAsUri();
    Response findOneResponse = RestAssured.given().
      header("Accept", "application/json").get(uriOfResource);
    String etagValue = findOneResponse.getHeader(HttpHeaders.ETAG);

    existingResource.setName(randomAlphabetic(6));
    update(existingResource);

    // When
    Response secondFindOneResponse= RestAssured.given().
      header("Accept", "application/json").headers("If-None-Match", etagValue)
      .get(uriOfResource);

    // Then
    assertTrue(secondFindOneResponse.getStatusCode() == 200);
}
```

Step by step:

- a *Resource* is first created and then retrieved – the *ETag* value is stored for further use
- the same *Resource* is then updated
- a new retrieve request is sent, this time with the *"If-None-Match"* header specifying the

ETag value previously stored
- on this second request, the server will returns a ***200 OK*** along with the full Resource, since the *ETag* value is no longer correct, as the Resource has been updated in the meantime

Finally, the last test – which is **not going to work** because the functionality has not yet been implemented in Spring – is the support for the *If-Match HTTP header*.

```
@Test
public void givenResourceExists_whenRetrievedWithIfMatchIncorrectEtag_then412IsReceived() {
   // Given
   T existingResource = getApi().create(createNewEntity());

   // When
   String uriOfResource = baseUri + "/" + existingResource.getId();
   Response findOneResponse = RestAssured.given().header("Accept", "application/json").
     headers("If-Match", randomAlphabetic(8)).get(uriOfResource);

   // Then
   assertTrue(findOneResponse.getStatusCode() == 412);
}
```

Step by step:

- a Resource is first created
- the Resource is then retrieved with the *"If-Match"* header specifying an incorrect ETag value – this is a **conditional GET request**
- the server should return a ***412 Precondition Failed***

6. ETags are BIG

We have only used ETags for **read operations** – a RFC exists trying to clarify how implementations should deal with ETags on **write operations** – this is not standard, but is an interesting read.

There are of course other possible uses of the ETag mechanism, such an for an Optimistic Locking Mechanism using Spring 3.1 as well as dealing with the related "Lost Update Problem".

There are also several known potential pitfalls and caveats to be aware of when using ETags.

7. Conclusion

In this section, we only scratched the surface with what's possible with Spring and ETags.

For a full implementation of an ETag enabled RESTful service, along with integration tests verifying the ETag behavior, **check out** the github project – this is an Eclipse based project, so it should be easy to import and run as it is.

11: REST PAGINATION IN SPRING

1. Overview

This section 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 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 ***"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 section focused on Discoverability of the 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.

5. Test Driving Pagination

Both the main logic of pagination and discoverability are covered by small, focused integration tests; as in the previous section, the rest-assured library 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 section):

```
@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.

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 section.

9. Conclusion

We 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** my github project – this is an Eclipse based project, so it should be easy to import and run as it is.

12: ERROR HANDLING FOR REST WITH SPRING

1. Overview

This section will focus on **how to implement Exception Handling with Spring for a REST API**. We'll look at both the recommended solution with Spring 3.2 and 4.x but also at the older options as well.

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.

After 3.2 we now have the new *@ControllerAdvice* annotation to address the limitations of the previous two solutions.

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.

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

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

```
public class FooController{
    ...
    @ExceptionHandler({ CustomException1.class, CustomException2.class })
    public void handleException() {
        //
    }
}
```

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.

The limitation is often worked around by having **all Controllers extend a Base Controller class** – however, this can be a problem for applications where, for whatever reasons, the Controllers cannot be made to extend from such a class. 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 does not include any changes to existing artifacts such as Controllers.

3. Solution 2 – The *HandlerExceptionResolver*

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

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

3.1. ExceptionHandlerExceptionResolver

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

3.2. *DefaultHandlerExceptionResolver*

This resolver was introduced in Spring 3.0 and is enabled by default in the *DispatcherServlet*. It is used to resolve standard Spring exceptions to their corresponding HTTP Status Codes, namely Client error – 4xx and Server error – 5xx status codes. Here is the **full list** of the Spring Exceptions it handles, and how these are mapped 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 – which is why a better option has been made available with Spring 3.2 – we'll talk about that in the latter part of this section.

3.3. *ResponseStatusExceptionResolver*

This resolver was also introduced in Spring 3.0 and is enabled by default in the *DispatcherServlet*. It's 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 final 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**. It is used 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**:

```
@Component
public class RestResponseStatusExceptionResolver extends AbstractHandlerExceptionResolver {

    @Override
    protected ModelAndView doResolveException
      (HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {
        try {
            if (ex instanceof IllegalArgumentException) {
                return handleIllegalArgument((IllegalArgumentException) ex, response, handler);
            }
            ...
        } catch (Exception handlerException) {
            logger.warn("Handling of [" + ex.getClass().getName() + "]
              resulted in Exception", handlerException);
        }
        return null;
    }

    private ModelAndView handleIllegalArgument
      (IllegalArgumentException ex, HttpServletResponse response) throws IOException {
        response.sendError(HttpServletResponse.SC_CONFLICT);
        String accept = request.getHeader(HttpHeaders.ACCEPT);
        ...
        return new ModelAndView();
    }
}
```

One detail to notice here is the Request itself is available, so the application can consider the value of the *Accept* header sent by the client. For example, if the client asks for *application/json* then, in case of an error condition, the application should still return a response body encoded with *application/json*.

The other important implementation detail is that a *ModelAndView* is returned – this is the **body of the response** and it will allow the application 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 is 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.

4. Solution 3 – The New *@ControllerAdvice* (Spring 3.2 And Above)

Spring 3.2 brings support for a global *@ExceptionHandler* with the new *@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);
   }
}
```

The new annotation allows the multiple scattered *@ExceptionHandler* from before to be consolidated into a **single, global error handling component**.

The actual mechanism is extremely simple but also very flexible:

- it allows full control over the body of the response as well as the status code
- it allows mapping of several exceptions to the same method, to be handled together
- it makes good use of the newer *RESTful ResposeEntity* response

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

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

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

5. Conclusion

We discussed here 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 4.0 and 4.1.

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

13: VERSIONING A REST API

1. The Problem

Evolving a REST API is a difficult problem – one for which many options are available. This section discusses through some of these options.

2. What is in the Contract?

Before anything else, we need to answer one simple question: ***What is the Contract between the API and the Client?***

2.1. URIs part of the Contract?

Let's first consider **the URI structure of the REST API** – is that part of the contract? Should clients bookmark, hardcode and generally rely on URIs of the API?

If they would, then the interaction of the Client with the REST Service would no longer be driven by the Service itself, but by what Roy Fielding calls ***out-of-band*** **information:**

A REST API should be entered with no prior knowledge beyond the initial URI (bookmark) and set of standardized media types that are appropriate for the intended audience…Failure here implies that out-of-band information is driving interaction instead of hypertext.

So clearly **URIs are not part of the contract!** The client should only know a single URI – the entry point to the API – all other URIs should be discovered while consuming the API.

2.2. Media Types part of the Contract?

What about **the Media Type information** used for the representations of Resources – are these part of the contract between the Client and the Service?

In order to successfully consume the API, the Client **must have prior knowledge of these Media Types** – in fact, the definition of these media types represents the entire contract, and is where the REST Service should focus the most:

A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types.

So the **Media Type definitions are part of the contract** and should be prior knowledge for the client that consumes the API – this is where standardization comes in.

We now have a good idea of what the contract is, let's move on to how to actually tackle the versioning problem.

3. High Level Options

Let's now discuss the high level approaches to versioning the REST API:

- **URI Versioning** – version the URI space using version indicators
- **Media Type Versioning** – version the Representation of the Resource

When we introduce **the version in the URI space**, the Representations of Resources are considered immutable, so when changes need to be introduced in the API, a new URI space needs to be created.

For example, say an API publishes the following resources – users and privileges:

```
http://host/v1/users
http://host/v1/privileges
```

Now, let's consider that a breaking change in the users API requires **a second version to be introduced**:

```
http://host/v2/users
http://host/v2/privileges
```

When we **version the Media Type and extend the language**, we go through Content Negotiation based on this header. The REST API would make use of custom vendor MIME media types instead of generic media types such as *application/json*. It is these media types that are going to be versioned instead of the URIs.

For example:

```
===>
GET /users/3 HTTP/1.1
Accept: application/vnd.myname.v1+json
<===
HTTP/1.1 200 OK
Content-Type: application/vnd.myname.v1+json
{
   "user": {
      "name": "John Smith"
   }
}
```

What is important to understand here is that **the client makes no assumptions about the structure of the response** beyond what is defined in the media type. This is why generic media types are not ideal – these **do not provide enough semantic information** and force the client to use require additional hints to process the actual representation of the resource.

An exception to this is using some other way of uniquely identifying the semantics of the content – such as an XML schema.

4. Advantages and Disadvantages

Now that we have a clear concept of what is part of the Contract between the Client and the Service, as well as a high level overview of the options to version the API, let's discuss the advantages and disadvantages of each approach.

First, introducing version identifiers in the URI leads to a **very large URI footprint**. This is due to the fact that any breaking change in any of the published APIs will introduce a whole new tree of representations for the entire API. Over time, this becomes a burden to maintain as well as a problem for the client – which now has more options to choose from.

Version identifiers in the URI is also **severely inflexible** – there is no way to simply evolve the API of a single Resource, or a small subset of the overall API. As we mentioned before, this is an all or nothing approach – if part of the API moves to the new version, then the entire API has to move along with it. This also makes upgrading clients from v1 to v2 a major undertaking – which leads to slower upgrades and much longer sunset periods for the old versions.

HTTP Caching is also a major concern when it comes to versioning.

From the **perspective of proxy caches in the middle**, each approach has advantages and disadvantages: if the URI is versioned, then the cache will need to keep multiple copies of each Resource – one for every version of the API. This puts load on the cache and decreases the cache hit rate, since different clients will use different versions. Also, some cache invalidation mechanisms will no longer work. If the media type is the one that is versioned, then both the Client and the Service need to support the Vary HTTP header to indicate that there are multiple versions being cached.

From the **perspective of client caching** however, the solution that versions the media type involves slightly more work than the one where URIs contain the version identifier. This is because it's simply easier to cache something when its key is an URL than a media type.

Let's end this section with defining some goals (straight out of API Evolution):

- keep compatible changes out of names
- avoid new major versions
- makes changes backwards-compatible
- think about forwards-compatibility

5. Possible Changes to the API

Next, let's consider the types of changes to the REST API – these are introduced here:

- representation format changes
- resource changes

5.1. Adding to the Representation of a Resource

The format documentation of the media type should be designed with forward compatibility in mind; specifically – a client should ignore information that it doesn't understand (which JSON does better than XML).

Now, adding information in the Representation of a resource **will not break existing clients** if these are correctly implemented.

To continue our earlier **example**, adding the amount in the representation of the user will not be a breaking change:

```
{
   "user": {
      "name": "John Smith",
      "amount": "300"
   }
}
```

5.2. Removing or changing an existing Representation

Removing, renaming or generally restructuring information in the design of existing representations is a breaking change for clients, because they already understand and rely on the old format.

This is where Content Negotiation comes in – for such changes, **a new vendor MIME media type** needs to be introduced.

Let's continue with the previous **example** – say we want to break the name of the user into *firstname* and *lastname*:

```
===>
GET /users/3 HTTP/1.1
Accept: application/vnd.myname.v2+json
<===
HTTP/1.1 200 OK
Content-Type: application/vnd.myname.v2+json
{
   "user": {
      "firstname": "John",
      "lastname": "Smith",
      "amount": "300"
   }
}
```

As such, this does represent an incompatible change for the Client – which will have to request the new Representation and understand the new semantics, but the URI space will remain stable and will not be affected.

5.3. Major Semantic Changes

These are changes in the meaning of the Resources, the relations between them or what the map to in the backend. This kinds of changes may require a new media type, or they may require publishing a new, sibling Resource next to the old one and making use of linking to point to it.

While this sounds like using version identifiers in the URI all over again, the important distinction is that the new Resource i**s published independently of any other Resources in the API** and will not fork the entire API at the root.

The REST API should adhere to **the HATEOAS constraint** – most of the URIs should be DISCOVERED by Clients, not hardcoded. Changing such an URI should not be considered an incompatible change – the new URI can replace the old one and Clients will be able to re-discover the URI and still function.

It is worth noting however that, while using version identifiers in the URI is problematic for all of these reasons, **it is not un-RESTful** in any way.

6. Conclusion

This section tried to provide an overview of the very diverse and difficult problem of **evolving a REST Service**. We discussed the two common solutions, advantages and disadvantages of each one, and ways to reason about these approaches in the context of REST. The material concludes by making the case for the second solution – **versioning the media types**, while examining the possible changes to a RESTful API.

14: TESTING REST WITH MULTIPLE MIME TYPES

1. Overview

This final section will focus on **testing a REST Service with multiple Media Types/ representations**.

We will write integration tests capable of switching between the multiple types of Representations supported by the API. The goal is to be able to run the exact same test consuming the exact same URIs of the service, just asking for a different Media Type.

2. Goals

Any REST API needs to expose it's Resources as representations using some sort of Media Type, and in many cases more than a single one. **The client will set the *Accept* header** to choose the type of representation it asks for from the service.

Since the Resource can have multiple representations, the server will have to implement a mechanism responsible with choosing the right representation – also known as **Content Negotiation**. Thus, if the client asks for *application/xml*, then it should get an XML representation of the Resource, and if it asks for *application/json*, then it should get JSON.

3. Testing Infrastructure

We'll begin by defining a simple interface for a **marshaller** – this will be the main abstraction that will allow the test to switch between different Media Types:

```
public interface IMarshaller {
   ...
   String getMime();
}
```

Then we need a way to initialize the right marshaller based on some form of external configuration. For this mechanism, we will use a Spring FactoryBean to initialize the marshaller and a simple **property** to determine which marshaller to use:

```
@Component
@Profile("test")
public class TestMarshallerFactory implements FactoryBean<IMarshaller> {

   @Autowired
   private Environment env;

   public IMarshaller getObject() {
      String testMime = env.getProperty("test.mime");
      if (testMime != null) {
         switch (testMime) {
         case "json":
            return new JacksonMarshaller();
         case "xml":
            return new XStreamMarshaller();
         default:
            throw new IllegalStateException();
         }
      }

      return new JacksonMarshaller();
   }

   public Class<IMarshaller> getObjectType() {
      return IMarshaller.class;
   }

   public boolean isSingleton() {
      return true;
   }
}
```

Let's look over this:

- first, the new *Environment* abstraction introduced in Spring 3.1 is used here – for more on this check out the detailed article on using Properties with Spring
- the ***test.mime* property** is retrieved from the environment and used to determine

which marshaller to create – some Java 7 *switch on String* syntax at work here
- next, the **default marshaller**, in case the property is not defined at all, is going to be the Jackson marshaller for JSON support
- finally – this *BeanFactory* is only active in a test scenario, as the new *@Profile* support, also introduced in Spring 3.1 is used

That's it – the mechanism is able to switch between marshallers based on whatever the value of the *test.mime* property is.

4. The JSON and XML Marshallers

Moving on, we'll need the actual marshaller implementation – one for each supported Media Type.

For JSON we'll use *Jackson* as the underlying library:

```
public class JacksonMarshaller implements IMarshaller {
   private ObjectMapper objectMapper;

   public JacksonMarshaller() {
      super();
      objectMapper = new ObjectMapper();
   }

   ...

   @Override
   public String getMime() {
      return MediaType.APPLICATION_JSON.toString();
   }
}
```

For the XML support, the marshaller uses *XStream*:

```
public class XStreamMarshaller implements IMarshaller {
   private XStream xstream;

   public XStreamMarshaller() {
      super();
      xstream = new XStream();
   }
```

```
    ...

    public String getMime() {
       return MediaType.APPLICATION_XML.toString();
    }
}
```

Note that **these marshallers are not Spring beans themselves.** The reason for that is they will be bootstrapped into the Spring context by the *TestMarshallerFactory*, so there is no need to make them components directly.

5. Consuming the Service with both JSON and XML

At this point we should be able to run a full integration test against the deployed service. Using the marshaller is straightforward – an *IMarshaller* is simply injected directly into the test:

```
@ActiveProfiles({ "test" })
public abstract class SomeRestLiveTest {

    @Autowired
    private IMarshaller marshaller;

    // tests
    ...

}
```

The exact marshaller that will be injected by Spring will of course be decided by the value of *test.mime property*; this could be picked up from a properties file or simply set on the test environment manually. If however a value is **not provided** for this property, the *TestMarshallerFactory* will simply fall back on the default marshaller – the JSON marshaller.

6. Maven and Jenkins

If Maven is set up to run integration tests against an already deployed REST Service, then it can be run like this:

```
mvn test -Dtest.mime=xml
```

Or, if this the build uses the integration-test phase of the Maven lifecycle:

```
mvn integration-test -Dtest.mime=xml
```

For more details about how to use these phases and how to set up the a Maven build so that it will bind the deployment of the application to the *pre-integration-test goal*, run the integration tests in the integration-test goal and then shut down the deployed service in on *post-integration-test*, see the Integration Testing with Maven article.

With **Jenkins**, the job must be configured with:

```
This build is parametrized
```

And the *String parameter: **test.mime=xml*** added.

A common Jenkins configuration would be having to jobs running the same set of integration tests against the deployed service – one with XML and the other with JSON representations.

7. Conclusion

This section showed how to properly test a REST API that works with multiple representations. Most APIs do publish their Resources under multiple Representations, so testing all of these is vital; the fact that we can use the exact same tests across all of them is just cool.

The full implementation of this mechanism – using actual integration tests and verifying both the XML and JSON representations – can be found in the github project. This is an Eclipse based project, so it should be easy to import and run as it is.

Building a REST API with Spring 4