Baeldung

Construya su API

1: Bootstrap una aplicación web con Spring 5

2: Crear una API REST con Spring y Java Config

3: Conversores de mensajes http con Spring Framework

4: Anotaciones RequestBody y ResponseBody de Spring

5: Conversión de entidad a DTO para una API REST de Spring

6: Gestión de errores para REST con Spring

7: Descubrimiento de API REST y HATEOAS

8: Introducción a los HATEOAS de primavera

9: Paginación REST en Spring

10: Probar una API REST con Java

1: Bootstrap una aplicación web con Spring 5

1. Visión general

El capítulo ilustra cómo **hacer Bootstrap de una aplicación Web con Spring**.

Vamos a ver la solución Spring Boot para bootstrapping la aplicación y también ver un enfoque no Spring Boot.

Utilizaremos principalmente la configuración Java, pero también echaremos un vistazo a su configuración XML equivalente.

2.1. Dependencia de Maven

En primer lugar, necesitaremos la dependencia spring-boot-starter-web:

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

Este arranque incluye:

• *spring-web* y el módulo *spring-webmvc* que necesitamos para nuestra aplicación web Spring
• un Tomcat starter para que podamos ejecutar nuestra aplicación web directamente sin instalar explícitamente ningún servidor

2.2. Creación de una aplicación Spring Boot

La forma más sencilla de empezar a utilizar Spring Boot es crear una clase principal y anótelo con *@SpringBootApplication*:

```
@SpringBootApplication
public clase SpringBootRestApplication {

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

Esta única anotación equivale a utilizar *@Conflguration*, *@EnableAutoConflguration* y *@ComponentScan.*

Por defecto, escaneará todos los componentes del mismo paquete o inferiores.

A continuación, para la configuración basada en Java de los beans de Spring, tenemos que crear una clase config y
anótelo con la anotación *@Configuration*:

```
@Configuración
public clase WebConfig
{
}
```

Esta anotación es el principal artefacto utilizado por la configuración de Spring basada en Java; está a su vez meta-anotada con *@Component*, lo que hace que las clases anotadas sean beans estándar y, como tales, también candidatas para el escaneo de componentes.

El propósito principal de las clases *@Conflguration* es ser fuentes de definiciones de beans para el Spring IoC Container. Para una descripción más detallada, consulte la documentación oficial.

Echemos también un vistazo a una solución utilizando la librería core *spring-webmvc*.

3.1. Dependencias de Maven

En primer lugar, necesitamos la dependencia spring-webmvc:

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

3.2. La configuración web basada en Java

A continuación, añadiremos la clase de configuración que tiene la anotación @Configuration:

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

}
```

Aquí, a diferencia de la solución Spring Boot, tendremos que definir explícitamente @EnableWebMvc para establecer las Configuraciones MVC de Spring por defecto y @ComponentScan para especificar los paquetes a escanear en busca de componentes.

La anotación *@EnableWebMvc* proporciona la configuración de Spring Web MVC, como la configuración del servlet dispatcher, la habilitación de las anotaciones @Controller y *@RequestMapping* y la configuración de otros valores predeterminados.

@ComponentScan configura la directiva de escaneo de componentes, especificando los paquetes a
escanear.

3.3. La clase inicializadora

A continuación, tenemos que **añadir una clase que implemente la interfaz *WebApplicationInitializer*:**

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

Aquí, estamos creando un contexto Spring usando la clase *AnnotationConflgWebApplicationContext*, lo que significa que estamos usando sólo configuración basada en anotaciones. A continuación, especificamos los paquetes para buscar componentes y clases de configuración.

Por último, estamos definiendo el punto de entrada para la aplicación web - *el*

DispatcherServlet. Esta clase puede reemplazar completamente el archivo *web.xml*

de las versiones Servlet <3.0

4. Configuración XML

Echemos también un vistazo a la configuración web XML equivalente:

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

Podemos reemplazar este archivo XML con la clase *YcbConflg* anterior.

Para iniciar la aplicación, podemos utilizar una clase Initializer que cargue la configuración XML o un fichero web.xml.

5. Conclusión

En este capítulo, hemos visto dos soluciones populares para arrancar una aplicación web Spring,
uno usando el Spring Boot web starter y otro usando la librería core spring-webmvc.

Como siempre, el código presentado en este capítulo está disponible en Github.

2: Crear una API REST con Spring y Java
Configurar

1. Visión general

Este capítulo muestra cómo **configurar REST en Spring**: el controlador y los códigos de respuesta HTTP, la configuración de la carga útil y la negociación de contenidos.

2. Entender REST en Spring

El framework Spring admite dos formas de crear servicios RESTful:

- utilizando MVC con ModelAndView
- Uso de convertidores de mensajes HTTP

El enfoque *ModclAndVicw* es más antiguo y está mucho mejor documentado, pero también es más verboso y pesa más en la configuración. Intenta meter con calzador el paradigma REST en el modelo antiguo, lo que no está exento de problemas. El equipo de Spring lo comprendió y proporcionó soporte REST de primera clase a partir de Spring 3.0.

El nuevo enfoque, basado en *HttpMessageConverter* y anotaciones, es mucho más ligero y fácil de implementar. La configuración es mínima y proporciona valores predeterminados razonables para lo que cabría esperar de un servicio RESTful.

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

La nueva anotación *@EnablcYcbMvc* hace algunas cosas útiles - específicamente, en el caso de REST, detecta la existencia de Jackson y JAXB 2 en el classpath y automáticamente crea y registra convertidores JSON y XML por defecto. La funcionalidad de la anotación es equivalente a la versión XML:

<mvc:annotation-drivcn />

Esto es un atajo, y aunque puede ser útil en muchas situaciones, no es perfecto. Cuando se necesite una configuración más compleja, elimine la anotación y extienda *YcbMvcConflgurationSupport* directamente.

3.1. Uso de Spring Boot

Si estamos usando la anotación *@SpringBootApplication* y la librería *spring-webmvc* está en el directorio
classpath, entonces la anotación *@EnablcYcbMvc* se añade automáticamente con una autoconfiguración por defecto.

Todavía podemos añadir funcionalidad MVC a esta configuración implementando la interfaz *YcbMvcConflgurcr* en una clase anotada *@Conflguration*. También podemos utilizar una instancia de *YcbMvcRcgistrationsAdaptcr* para proporcionar nuestras propias implementaciones de *RcqucstMappingHandlcrMapping*, *RcqucstMappingHandlcrAdaptcr* o *ExccptionHandlcrExccptionRcsolvcr*.

Por último, si queremos descartar las características MVC de Spring Boot y declarar una configuración personalizada,
podemos hacerlo utilizando la anotación *@EnablcYcbMvc*.

A partir de Spring 3.1, disponemos de soporte de pruebas de primera clase para las clases @Configuration:

```
1.  @RunWith(SpringJUnit4ClassRunner.class)
2.  @ConfiguraciónContexto(
3.    classes = {WebConfig.class,
4.    PersistenceConfig.class}, loader =
5.    AnnotationConfigContextLoader.class)
6.  public clase SpringTest {
7.     @Prueba
8.     public void
9.     whenSpringContextIsInstantiated_entoncesNoExceptions(){
10.       // Cuando
11. }  }
```

Estamos especificando las clases de configuración Java con la anotación *@ContcxtConflguration*. La nueva *AnnotationConflgContcxtLoadcr* carga las definiciones de bean de las clases *@Conflguration*.

Observe que la clase de configuración WebConfig no se incluyó en la prueba porque necesita ejecutarse en un contexto Servlet, que no se proporciona.

4.1. Uso de Spring Boot

Spring Boot proporciona varias anotaciones para configurar el Spring *ApplicationContcxt* para nuestras pruebas de una forma más intuitiva.

Podemos cargar sólo una porción concreta de la configuración de la aplicación, o podemos simular el
todo el proceso de inicio del contexto.

Por ejemplo, podemos utilizar la anotación *@SpringBootTcst* si queremos que todo el contexto se cree sin iniciar el servidor.

Una vez hecho esto, podemos añadir *@AutoConflgurcMockMvc* para inyectar una instancia *MockMvc*
y enviar peticiones HTTP:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public clase FooControllerAppIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Prueba
    public void whenTestApp_thenEmptyResponse() throws Exception {
        this.mockMvc.perform(get("/foos")
            .andExpect(status().isOk())
            .andExpect(...);
    }

}
```

Para evitar crear todo el contexto y probar sólo nuestros controladores MVC, podemos utilizar *@YcbMvcTcst:*

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

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    servicio privado
    IFooService;
    @Prueba()
    public void whenTestMvcController_thenRetrieveExpectedResult()
throws Exception {
        // ...

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

Podemos encontrar información detallada sobre este tema en 'Testing in Spring Boot'.

5. El controlador

El *@RestController* es el artefacto central en toda la Web Tier de la API RESTful. Para el propósito de este post, el controlador está modelando un simple recurso REST - Foo:

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

Te habrás dado cuenta de que estoy usando una utilidad *RcstPrcconditions* directa, al estilo de Guava:

```
1.      public static <T> T checkFound(T resource) {
2.          if (resource == null) {
3.              lanza una nueva excepción
4.              MyResourceNotFoundException();
5.          }
6.          devolver recurso;
7.  }   }
```

La implementación del controlador no es pública porque no es necesario.

Normalmente, el controlador es el último en la cadena de dependencias. Recibe peticiones HTTP del controlador frontal de Spring (el *DispatchcrScrvlct*) y simplemente las delega hacia una capa de servicio. Si no hay ningún caso de uso en el que el controlador tenga que ser inyectado o manipulado a través de una referencia directa, entonces prefiero no declararlo como público.

Las asignaciones de solicitud son sencillas. Al igual **que con cualquier controlador, el *valor* real de la asignación, así como el método HTTP, determinan el método de destino de la solicitud.** *@RcqucstBody* vinculará los parámetros del método al cuerpo de la petición HTTP, mientras que *@RcsponscBody* hace lo mismo con la respuesta y el tipo de retorno.

El *@RcstControllcr* es una abreviatura para incluir tanto el *@RcsponscBody* y el *@Controllcr* anotaciones en nuestra clase.

También aseguran que el recurso será marshalled y unmarshalled usando el convertidor HTTP correcto. La negociación de contenido tendrá lugar para elegir cuál de los conversores activos se utilizará, basándose principalmente en el *Acccpthcadcr*, aunque también pueden utilizarse otras cabeceras HTTP para determinar la representación.

6. Asignación de los códigos de respuesta HTTP

Los códigos de estado de la respuesta HTTP son una de las partes más importantes del servicio REST, y el tema puede complicarse rápidamente. Hacerlos bien puede ser lo que haga o deshaga el servicio.

6.1. Solicitudes sin mapa

Si Spring MVC recibe una solicitud que no tiene una asignación, considera que la solicitud no está permitida y devuelve al cliente un mensaje 405 METHOD NOT ALLOWED.

También es una buena práctica incluir la cabecera HTTP Allow cuando se devuelve un 405 al cliente, para especificar qué operaciones están permitidas. Este es el comportamiento estándar de Spring MVC y no requiere ninguna configuración adicional.

6.2. Solicitudes asignadas válidas

Para cualquier solicitud que tenga una asignación, Spring MVC considera que la solicitud es válida y responde con 200 OK si no se especifica otro código de estado.

Es por esto que el controlador declara diferentes *@RcsponscStatus* para el *crcatc*, *updatc* y *dclctc* pero no para *gct*, que debería devolver el 200 OK por defecto.

6.3. Error de cliente

En caso de error de un cliente, se definen excepciones personalizadas y se asignan a la correspondiente
códigos de error.

Simplemente lanzando estas excepciones desde cualquiera de las capas de la capa web se asegurará que Spring
asigna el código de estado correspondiente en la respuesta HTTP:

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

Estas excepciones forman parte de la API REST y, como tales, sólo deben utilizarse en las capas apropiadas correspondientes a REST; si, por ejemplo, existe una capa DAO/DAL, ésta no debe utilizar las excepciones directamente.

Tenga en cuenta también que no se trata de excepciones comprobadas, sino de excepciones en tiempo de ejecución, de acuerdo con Spring
prácticas y modismos.

6.4. Uso de *@ExceptionHandler*

Otra opción para asignar excepciones personalizadas a códigos de estado específicos es utilizar la anotación *@ExccptionHandlcr* en el controlador. El problema con este enfoque es que la anotación sólo se aplica al controlador en el que se define. Esto significa que tenemos que declarar en cada controlador de forma individual.

Por supuesto, hay más formas de gestionar errores tanto en Spring como en Spring Boot que ofrecen más flexibilidad.

Además de la dependencia *spring-webmvc* necesaria para la aplicación web estándar, tendremos que configurar el marshalling y unmarshalling de contenido para la API REST:

```
<dependencias>
   <dependencia>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.9.8</version>
   </dependencia>
   <dependencia>
      <groupId>javax.xml.bind</groupId>
      <artifactId>jaxb-api</artifactId>
      <version>2.3.1</version>
      <ámbito>runtime</sámbito>
   </dependencia>
</dependencias>
```

Estas son las bibliotecas utilizadas para convertir la representación del recurso REST a JSON o XML.

7.1. Uso de Spring Boot

Si queremos recuperar recursos con formato JSON, Spring Boot proporciona soporte para diferentes librerías, concretamente Jackson, Gson y JSON-B.

La autoconfiguración se realiza simplemente incluyendo cualquiera de las librerías de mapeo

en el classpath. Normalmente, si estamos desarrollando una aplicación web**, simplemente**

añadiremos la librería *spring-boot-starter-web*
y confiar en ella para incluir todos los artefactos necesarios en nuestro proyecto:

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

Spring Boot utiliza Jackson por defecto.
Si queremos serializar nuestros recursos en un formato XML, tendremos que añadir la extensión Jackson XML (*jackson-dataformat-xml)* a nuestras dependencias, o recurrir al JAXB

(proporcionada por defecto en el JDK) utilizando la anotación *@XmlRootElcmcnt* en nuestro recurso.

7. Dependencias adicionales de Maven

Este capítulo ilustra cómo implementar y configurar un Servicio REST utilizando Spring y configuración basada en Java.

Todo el código de este capítulo está disponible en Github.

3: Conversores de mensajes http con Spring Framework

1. Visión general

Este capítulo describe **cómo configurar *HttpMessageConverters* en Spring.**

En pocas palabras, podemos utilizar convertidores de mensajes para marshall y unmarshall objetos Java desde y hacia JSON, XML, etc - a través de HTTP.

2.1. Activar Web MVC

Para empezar, la Aplicación Web necesita ser **configurada con soporte Spring MVC**. A forma cómoda y muy personalizable de hacerlo es utilizar la anotación *@EnablcYcbMvc*:

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

Tenga en cuenta que esta clase implementa *YcbMvcConflgurcr* - que nos permitirá cambiar la predeterminada
lista de Http Convertidores con la nuestra.

2.2. Los conversores de mensajes por defecto

Por defecto, las siguientes instancias de HttpMessageConverters están prehabilitadas:

- ByteArrayHttpMessageConverter - convierte matrices de bytes
- StringHttpMessageConverter - convierte cadenas de texto
- ResourceHttpMessageConverter - convierte org.springframework.core.io.Resource para cualquier
  tipo de flujo de octetos
- SourceHttpMessageConverter - convierte javax.xml.transform.Source
- FormHttpMessageConverter - convierte datos de formulario a/desde un MultiValueMap<String, String>.
- Jaxb2RootElementHttpMessageConverter - convierte objetos Java a/desde XML
  (añadido sólo si JAXB2 está presente en el classpath)
- MappingJackson2HttpMessageConverter - convierte JSON (añadido sólo si Jackson 2 es
  presente en el classpath)
- MappingJacksonHttpMessageConverter - convierte JSON (añadido sólo si Jackson está presente
  en el classpath)
- AtomFeedHttpMessageConverter - convierte feeds Atom (se añade sólo si Roma está presente en
  el classpath)
- RssChannelHttpMessageConverter - convierte los canales RSS (sólo se añade si Roma está presente en
  el classpath)

3.1. Negociación de contenidos de alto nivel

Cada implementación de *HttpMcssagcConvcrtcr* tiene uno o varios Tipos MIME asociados.

Al recibir una nueva petición**, Spring utilizará la cabecera "Accept" para determinar el medio con el que debe responder**.

A continuación, intentará encontrar un conversor registrado que sea capaz de manejar ese tipo de medio específico. Por último, lo utilizará para convertir la entidad y devolver la respuesta.

El proceso es similar para la recepción de una solicitud que contiene información JSON. El marco **utilizará la cabecera "*Content-Type*" para determinar el tipo de medio del cuerpo de la solicitud.**

A continuación, buscará un *HttpMcssagcConvcrtcr* que pueda convertir el cuerpo enviado por el cliente en un objeto Java.

Aclaremos esto con un ejemplo rápido:

- el Cliente envía una solicitud GET a */foos* con el encabezado Accept establecido en application/json - a
  obtener todos los recursos Foo como JSON
- el controlador *Foo* Spring es golpeado y devuelve las entidades Java *Foo* correspondientes
- A continuación, Spring utiliza uno de los conversores de mensajes de Jackson para convertir las entidades a JSON

Veamos ahora cómo funciona esto y cómo podemos aprovechar el potencial de la tecnología. *@RcsponscBody* y *@RcqucstBodyannotations.*

3.2. @ResponseBody

@RcsponscBody en un método de controlador indica a Spring que **el valor de retorno del método se serializa directamente en el cuerpo de la respuesta HTTP**. Como se ha comentado anteriormente, la cabecera "*Acccpt*" especificada por el Cliente se utilizará para elegir el Conversor Http apropiado para marshallar la entidad.

Veamos un ejemplo sencillo:

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

Ahora, el cliente especificará la cabecera "Accept" a *application/json* en la petición - ejemplo comando *curl*:

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

La clase Foo:

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

Y el cuerpo de la respuesta Http:

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

3.3. @CuerpoPetición

Podemos utilizar la anotación *@RcqucstBody* en el argumento de un método Controller para indicar **que el cuerpo de la petición HTTP se deserializa a esa entidad Java en particular**. Para determinar el conversor apropiado, Spring utilizará la cabecera "Content-Type" de la petición del cliente.

Veamos un ejemplo:

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

A continuación, vamos a consumir esto con un objeto JSON - estamos especificando "Content-Type" para ser
application/json:

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

Recibimos un 200 OK - una respuesta satisfactoria:

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

4. Configuración de convertidores personalizados

También podemos **personalizar los conversores de mensajes implementando la función *YcbMvcConfigurcr***
y sobreescribiendo el método *conflgurcMcssagcConvcrtcrs*:

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

Y aquí está la configuración XML correspondiente:

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

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

En este ejemplo, estamos creando un nuevo convertidor - el *MarshallingHttpMcssagcConvcrtcr* utilizando el soporte de Spring XStream para `configurarlo`. Esto permite una `gran` flexibilidad ya que **estamos trabajando con las APIs de bajo nivel del framework de marshalling subyacente** - en este caso XStream - y podemos configurarlo como queramos.

Tenga en cuenta que este ejemplo requiere añadir la biblioteca XStream al classpath.

También hay que tener en cuenta que al ampliar esta clase de soporte, **estamos perdiendo los convertidores de mensajes por defecto que estaban previamente registrados**.

Por supuesto ahora podemos hacer lo mismo para Jackson - definiendo nuestro propio *MappingJackson2HttpMcssagcConvcrtcr*. Ahora podemos establecer un *ObjcctMappcr* personalizado en este convertidor y tenerlo configurado como necesitemos.

En este caso, XStream fue la implementación de marshaller/unmarshaller seleccionada, pero otras
como CastorMarshaller también pueden utilizarse.

En este punto - con XML habilitado en el back-end - podemos consumir la API con Representaciones XML:

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

4.1. Compatibilidad con Spring Boot

Si estamos utilizando Spring Boot podemos evitar implementar el *YcbMvcConflgurcr* y añadir todos los
los Conversores de Mensajes manualmente como hicimos anteriormente.

Podemos definir diferentes beans *HttpMessageConverter* en el contexto, y Spring Boot los añadirá automáticamente a la autoconfiguración que crea:

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

Al igual que en el lado del servidor, la conversión de mensajes http puede configurarse en el lado del cliente.
en el *RcstTcmplatc de* primavera.

Vamos a configurar la plantilla con las cabeceras "*Acccpt*" y "*Contcnt-Typc*" cuando sea apropiado. Luego intentaremos consumir la API REST con marshalling y unmarshalling completo del Recurso *Foo* - tanto con JSON como con XML.

5.1. Recuperación del recurso sin cabecera *Accept*

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

    // ...

    return xmlConverter;
}
```

5.2. Recuperación de un recurso con encabezado de *aceptación application/xml*

Ahora vamos a recuperar explícitamente el Recurso como una Representación XML. Vamos a definir un conjunto
de Convertidores y fijarlos en el *RcstTcmplatc*.

Como estamos consumiendo XML, vamos a utilizar el mismo XStream marshaller que antes:

```
@Prueba
public void dadoConsumiendoXml_alLeerElFoo_entoncesCorrecto() {
    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);

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

}   assertThat(recurso, notNullValue());
```

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

5.3. Recuperación de un recurso con la cabecera *application/json* Accept

Del mismo modo, ahora vamos a consumir la API REST pidiendo JSON:

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

5.4. Actualizar un recurso con XML *Content-Type*

Por último, enviemos también datos JSON a la API REST y especifiquemos el tipo de medio de esos datos mediante
la *Contcnt-Typchcadcr:*

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

Lo interesante aquí es que somos capaces de mezclar los tipos de medios - **estamos enviando datos XML pero estamos esperando datos JSON de vuelta del servidor.** Esto demuestra lo potente que es el mecanismo de conversión de Spring.

6. Conclusión

En este capítulo hemos visto cómo Spring MVC nos permite especificar y personalizar completamente los Conversores de Mensajes Http para **marshall/unmarshall automáticamente Entidades Java desde y hacia XML o JSON**. Esta es, por supuesto, una definición simplista, y hay mucho más que el mecanismo de conversión de mensajes puede hacer - como podemos ver en el último ejemplo de prueba.

También hemos visto cómo aprovechar el mismo potente mecanismo con el *RcstTcmplatc* lo que da lugar a una forma totalmente segura de consumir la API.

Como siempre, el código presentado en este capítulo está disponible en Github.

4: Anotaciones RequestBody y ResponseBody de Spring

1. Introducción

En este breve capítulo, ofrecemos una visión general de los módulos @RequestBody y @ Anotaciones de ResponseBody.

En pocas palabras, la anotación *@RcqucstBody* asigna el cuerpo *HttpRcqucst* a una transferencia o dominio
lo que permite la deserialización automática del cuerpo *HttpRcqucst* entrante en un objeto Java.

En primer lugar, echemos un vistazo a un método de controlador de Spring:

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

Spring deserializa automáticamente el JSON en un tipo Java suponiendo que se especifique uno apropiado. Por defecto, el tipo que anotamos con la anotación *@RcqucstBody* debe corresponder al JSON enviado desde nuestro controlador del lado del cliente:

```
1.  public class LoginForm {
2.      private String nombre de
3.      usuario; private String
4.      contraseña;
5.      // ...
    }
```

Aquí, el objeto que utilizamos para representar el cuerpo *HttpRcqucst se* asigna a nuestro objeto *LoginForm*.

Vamos a probar esto usando CURL:

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

Esto es todo lo que se necesita para una API REST de Spring y un cliente Angular utilizando el *@RcqucstBody*

1. Introducción

anotación

3. @ResponseBody

La anotación *@RcsponscBody* indica a un controlador que el objeto devuelto es automáticamente serializado en JSON y devuelto al objeto *HttpRcsponsc*.

Supongamos que tenemos un objeto Respuesta personalizado:

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

A continuación, se puede implementar el controlador asociado:

```
1.  @Controller
2.  @RequestMapping("/post")
3.  public clase ExamplePostController {
4.
5.      @Autowired
6.      EjemploServicio ejemploServicio;
7.
8.      @PostMapping("/respuesta")
9.      @ResponseBody
10.     public ResponseTransfer postResponseController(
11.       @RequestBody LoginForm loginForm) {
12.         return new ResponseTransfer("¡¡¡Gracias por
13.         publicar!!!");
14. }     }
```

En la consola de desarrollador de nuestro navegador o utilizando una herramienta como Postman, podemos ver la siguiente respuesta:

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

Recuerde que no es necesario anotar los controladores *anotados con @RestController* con la propiedad
@ResponseBodyannotation ya que aquí se hace por defecto.

4. Conclusión

Hemos construido un cliente Angular sencillo para la aplicación Spring que demuestra cómo utilizar las anotaciones @ RestController y @ResponseBody.

Como siempre, hay ejemplos de código disponibles en GitHub.

5: Conversión de entidad a DTO para una API REST de Spring

1. Visión general

Cada En este tutorial, vamos a manejar las conversiones que deben ocurrir **entre las entidades internas de una aplicación Spring y los DTOs externos** (Objetos de Transferencia de Datos) que se publican de nuevo al cliente.

2. Mapeador de modelos

Empecemos introduciendo la librería principal que vamos a utilizar para realizar esta conversión entidad-DTO - *ModclMappcr*.

Necesitaremos esta dependencia en el *pom.xml*:

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

A continuación definiremos el bean *ModclMappcr* en nuestra configuración de Spring:

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

A continuación, presentaremos el lado DTO de este problema de dos caras: *Post DTO*:

```
public class PostDto {
    private static final SimpleDateFormat dateFormat
      = new SimpleDateFormat("aaaa-MM-dd HH:mm");

    private Long id;

    private String title;

    private String url;

    private String date;

    private UserDto user;

    public Date getSubmissionDateConverted(String timezone) throws
ParseException {
        dateFormat.setTimeZone(TimeZone.getTimeZone(timezone));
        return dateFormat.parse(this.date);
    }

    public void setSubmissionDate(Date date, String timezone)
        {
        dateFormat.setTimeZone(TimeZone.getTimeZone(timezone));
        this.date = dateFormat.format(date);
    }

}   // getters y setters estándar
```

Tenga en cuenta que los 2 métodos personalizados relacionados con la fecha manejan la conversión de fecha de ida y vuelta entre el cliente y el servidor:

- El método *gctSubmissionDatcConvcrtcd()* convierte ts Date String en una fecha en la zona horaria del servidor para utilizarla en la persistencia de la entidad Post
- El método *sctSubmissionDatc()* es para establecer la fecha del DTO a la Fecha de Publicación en la zona horaria actual del usuario.

Veamos ahora una operación a nivel de servicio - que obviamente funcionará con la Entidad (no con el DTO):

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

Vamos a echar un vistazo a la capa por encima de servicio siguiente - la capa de controlador. Aquí es donde el
conversión también se producirá realmente.

Veamos ahora la implementación de un controlador estándar, exponiendo la API REST simple para el recurso Post. Vamos a mostrar aquí unas cuantas operaciones CRUD sencillas: crear, actualizar, obtener uno y obtener todos. Y dado que las operaciones son bastante sencillas, **estamos especialmente interesados en los aspectos de conversión Entidad-DTO**:

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

39.     @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
40.     @ResponseStatus(HttpStatus.OK)
41.     public void updatePost(@RequestBody PostDto postDto) {
42.         Post post = convertToEntity(postDto);
43.         postService.updatePost(post);
44.     }
45. }
```

Y aquí está **nuestra conversión de entidad *Post* a *PostDto***:

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

Y aquí está la conversión **de DTO a una entidad**:

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

Así que, como puedes ver, con la ayuda del mapeador de modelos, **la lógica de conversión es rápida y sencilla**: estamos utilizando la API de *mapas* del mapeador y obteniendo los datos convertidos sin escribir ni una sola línea de lógica de conversión.

Por último, hagamos una prueba muy sencilla para asegurarnos de que las conversiones entre la entidad y el DTO funcionan bien:

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

7. Conclusión

Este fue un capítulo sobre la simplificación de la conversión de Entity a DTO y de DTO a Entity en una API REST de Spring, mediante el uso de la biblioteca model mapper en lugar de escribir estas conversiones a mano.

El código fuente completo de los ejemplos está disponible en el proyecto GitHub.

6: Gestión de errores para REST con Spring

1. Visión general

Este capítulo ilustrará **cómo implementar el Manejo de Excepciones con Spring para una API REST**. También haremos un repaso histórico y veremos qué nuevas opciones introdujeron las diferentes versiones.

Antes de Spring 3.2, los dos enfoques principales para manejar excepciones en una aplicación Spring MVC eran: *HandlerExceptionResolver* o la anotación *@ExceptionHandler*. Ambos tienen algunas desventajas claras.

Desde la versión 3.2 disponemos de la anotación *@ControllerAdvice* para solventar las limitaciones de las dos soluciones anteriores y promover un manejo unificado de las excepciones en toda la aplicación.

Ahora, **Spring 5 introduce la clase *ResponseStatusException***: Una forma rápida de gestionar errores básicos en nuestras APIs REST.

Todas ellas tienen algo en común: manejan muy bien la **separación de preocupaciones**. La aplicación puede lanzar excepciones normalmente para indicar un fallo de algún tipo, excepciones que se tratarán por separado.

Por último, veremos qué aporta Spring Boot y cómo podemos configurarlo para que se adapte a nuestras necesidades.

2. Solución 1 - El *@ExceptionHandler* a nivel de controlador

La primera solución funciona en el *@Controllcr* lcvcl - definiremos un método para manejar excepciones,
y anótelo con *@ExccptionHandlcr*:

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

Este enfoque tiene un inconveniente importante - **el método anotado *@ExceptionHandler* sólo está activo para ese controlador en particular,** no globalmente para toda la aplicación. Por supuesto, añadir esto a cada controlador hace que no sea adecuado para un mecanismo general de manejo de excepciones.

Podemos evitar esta limitación haciendo que **todos los Controladores extiendan una clase Controlador Base** - sin embargo, esto puede ser un problema para aplicaciones donde, por cualquier razón, esto no es posible. Por ejemplo, los Controladores pueden ya extender de otra clase base que puede estar en otro jar o no directamente modificable, o pueden ellos mismos no ser directamente modificables.

A continuación, veremos otra forma de resolver el problema de la gestión de excepciones, una que es global
y no incluye ningún cambio en los artefactos existentes, como los controladores.

La segunda solución es definir un *HandlcrExccptionRcsolvcr* - esto resolverá cualquier excepción lanzada por la aplicación. También nos permitirá implementar un **mecanismo uniforme de gestión de excepciones** en nuestra API REST.

Antes de optar por un resolver personalizado, repasemos las implementaciones existentes.

3.1. *ExceptionHandlerExceptionResolver*

Este resolver fue introducido en Spring 3.1 y está habilitado por defecto en el *DispatchcrScrvlct*. En realidad, es el componente central del funcionamiento del mecanismo *@ExccptionHandlcr* presentado anteriormente.

3.2. *DefaultHandlerExceptionResolver*

Esta resolución se introdujo en Spring 3.0 y está activada por defecto en *DispatchcrScrvlct*. Se utiliza para resolver excepciones estándar de Spring a sus correspondientes códigos de estado HTTP, es decir, error del cliente - 4xx y error del servidor - códigos de estado 5xx. Aquí está la lista completa de las excepciones de Spring que maneja, y cómo se asignan a los códigos de estado.

Aunque establece el Código de Estado de la Respuesta correctamente, una **limitación es que no establece nada en el cuerpo de la Respuesta**. Y para una API REST - el Código de Estado no es realmente suficiente información para presentar al Cliente - la respuesta tiene que tener un cuerpo también, para permitir a la aplicación dar información adicional sobre el fallo.

Esto puede resolverse configurando la resolución de la vista y renderizando el contenido de error a través de *ModclAndVicw*, pero la solución no es claramente óptima. Por eso Spring 3.2 introdujo una opción mejor que discutiremos en una sección posterior.

3.3. *ResponseStatusExceptionResolver*

Este resolver también se introdujo en Spring 3.0 y está habilitado por defecto en el *DispatchcrScrvlct*. Su principal responsabilidad es utilizar la anotación *@RcsponscStatus* disponible en excepciones personalizadas y asignar estas excepciones a códigos de estado HTTP.

Una excepción personalizada de este tipo puede tener el siguiente aspecto:

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

Al igual que *DcfaultHandlcrExccptionRcsolvcr*, este resolver está limitado en la forma en que trata el cuerpo de la respuesta - mapea el Código de Estado en la respuesta, pero el cuerpo sigue siendo *nulo*.

3.4. *HandlerExceptionResolver personalizado*

La combinación de *DcfaultHandlcrExccptionRcsolvcr* y *RcsponscStatusExccptionRcsolvcr* contribuye en gran medida a proporcionar un buen mecanismo de gestión de errores para un servicio RESTful de Spring. La desventaja es - como se mencionó antes - **no hay control sobre el cuerpo de la respuesta**.

Idealmente, nos gustaría ser capaces de dar salida a JSON o XML, dependiendo del formato del cliente.
ha solicitado (a través de la cabecera *Acccpt*).

Esto justifica por sí solo la creación de un **nuevo solucionador de excepciones personalizado**:

```
@Componente
public class RestResponseStatusExceptionResolver extends
AbstractHandlerExceptionResolver {

    @Override
    protected ModelAndView doResolveException
      (HttpServletRequest request, HttpServletResponse response, Object
handler, Exception ex) {
        intentar {
            if (ex instanceof IllegalArgumentException) {
                return handleIllegalArgument((IllegalArgumentException)
ex, response, handler);
            }
            ...
        } catch (Exception handlerException) {
            logger.warn("Manipulación de [" + ex.getClass().getName() +
            "]
              resultó en Excepción", handlerException);
        }
        devolver null;
    }

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

Un detalle a notar aquí es que tenemos acceso al propio *rcqucst*, por lo que podemos considerar el
valor del *Acccpthcadcr* enviado por el cliente.

Por ejemplo, si el cliente pide *application/json,* entonces, en caso de una condición de error, podríamos
queremos asegurarnos de que devolvemos un cuerpo de respuesta codificado con *application/json.*

El otro detalle importante de la implementación es **que devolvemos un *ModelAndView* - este es el cuerpo
de la respuesta** y nos permitirá establecer lo que sea necesario en ella.

Este enfoque es un mecanismo coherente y fácilmente configurable para la gestión de errores de un servicio Spring REST. Sin embargo, tiene limitaciones: está interactuando con el módulo de bajo nivel
HtttpScrvlctRcsponsc y encaja en el viejo modelo MVC que utiliza *ModclAndVicw* - así que hay aún se puede mejorar.

4. Solución 3 - *@ControllerAdvice*

Spring 3.2 ofrece soporte para un ***@ExceptionHandler*** global con la anotación ***@ControllerAdvice***. Esto permite un mecanismo que rompe con el antiguo modelo MVC y hace uso de *RcsponscEntity* junto con la seguridad de tipos y la flexibilidad de *@ExccptionHandlcr*:

```
1.  @ControllerAdvice
2.  public class RestResponseEntityExceptionHandler
3.    extends ResponseEntityExceptionHandler {
4.
5.      @ExceptionHandler(valor
6.        = { IllegalArgumentException.class, IllegalStateException.class })
7.      protected EntidadRespuesta<Objeto> handleConflict(
8.        RuntimeException ex, WebRequest request) {
9.          String bodyOfResponse = "Esto debería ser específico de la
10.         aplicación";
11.         return handleExceptionInternal(ex, bodyOfResponse,
12.           new HttpHeaders(), HttpStatus.CONFLICT, request);
13. }   }
```

La anotación@ControllcrAdvicc nos permite **consolidar nuestras múltiples y dispersas *@ExceptionHandlers* de antes en un único componente global de gestión de errores.**

El mecanismo real es extremadamente sencillo, pero también muy flexible. Nos da:

- Control total sobre el cuerpo de la respuesta y el código de estado
- Asignación de varias excepciones al mismo método, que se tratarán conjuntamente, y
- Hace un buen uso de la nueva respuesta RESTful ResposeEntity

Una cosa a tener en cuenta aquí es **hacer coincidir las excepciones declaradas con *@ExceptionHandler* con la excepción utilizada como argumento del método**. Si no coinciden, el compilador no se quejará - no hay razón para que lo haga, y Spring tampoco se quejará.

Sin embargo, cuando la excepción se lanza realmente en tiempo de ejecución, **el mecanismo de resolución de excepciones fallará con:**

```
java.lang.IllegalStateException: No hay resolución adecuada para el
argumento [0] [type=...]
Detalles de HandlerMethod: ...
```

5. Solución 4 - *ResponseStatusException*

Spring 5 introdujo la clase ResponseStatusException. Podemos crear una instancia de la misma proporcionando un HttpStatus y opcionalmente una razón y una causa:

```
1.  @GetMapping(valor = "/{id}")
2.  public Foo findById(@PathVariable("id") Long id, HttpServletResponse
3.  respuesta) {
4.      intentar {
5.          Foo resourceById = RestPreconditions.checkFound(service.
6.  findOne(id));
7.
8.          eventPublisher.publishEvent(new
9.  SingleResourceRetrievedEvent(this, response));
10.         devolver resourceById;
11.      }
12.     catch (MyResourceNotFoundException exc) {
13.          throw new ResponseStatusException(
14.            HttpStatus.NOT_FOUND, "Foo no encontrado",
15.            exc);
16.     }
    }
```

¿Cuáles son las ventajas de utilizar *RcsponscStatusExccption?*

- Excelente para la creación de prototipos: Podemos implementar una solución básica con bastante rapidez
- Un tipo, varios códigos de estado: Un tipo de excepción puede dar lugar a varios respuestas. **Esto reduce el acoplamiento estrecho en comparación con el *@ExceptionHandler***
- No tendremos que crear tantas clases de excepción personalizadas
- **Mayor control sobre la gestión de** excepciones, ya que éstas pueden crearse mediante programación.

¿Y qué hay de las compensaciones?

- No hay una forma unificada de manejar las excepciones: Es más difícil imponer algunas convenciones para toda la aplicación, a diferencia de *@ControllcrAdvicc*, que proporciona un enfoque global.
- Duplicación de código: Podemos encontrarnos replicando código en múltiples controladores

También hay que tener en cuenta que es posible combinar distintos enfoques en una misma aplicación.

Por ejemplo, podemos implementar un *@ControllerAdvice* globalmente, pero también *ResponseStatusExceptions* localmente. Sin embargo, debemos tener cuidado: Si la misma excepción puede ser manejada de múltiples maneras, podemos notar algún comportamiento

sorprendente. Una posible convención es manejar un tipo específico de excepción siempre de una manera.

El acceso denegado se produce cuando un usuario autenticado intenta acceder a recursos que él
no tiene suficientes autoridades para acceder.

6.1. REST y seguridad a nivel de método

Por último, veamos cómo manejar la excepción de acceso denegado lanzada por la seguridad a nivel de método
anotaciones - *@PrcAuthorizc*, *@PostAuthorizc*, y *@Sccurc*.

Por supuesto, utilizaremos el mecanismo de gestión global de excepciones que hemos comentado antes para
manejar también el *AcccssDcnicdExccption*:

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

7. Compatibilidad con Spring Boot

Spring Boot proporciona una implementación de *ErrorController* para manejar los errores de una manera sensata.

En pocas palabras, sirve una página de error alternativa para los navegadores (también conocida como página de error de etiqueta blanca) y una respuesta JSON para las solicitudes RESTful sin HTML:

```
{
    "timestamp": “2019-01-17T16:12:45.977+0000”,
    "estado": 500,
    "error": "Error interno del servidor",
    "mensaje": "¡Error al procesar la
    petición!", "path": "/mi-punto-con-
}   excepciones"
```

Como es habitual, Spring Boot permite configurar estas características con propiedades:

- *scrvcr.crror.whitclabcl.cnablcd:* puede utilizarse para desactivar la página de error de etiqueta blanca y confiar en que el contenedor de servlets proporcione un mensaje de error HTML.
- server.error.include-stacktrace: con un valor always, incluye el stacktrace tanto en la respuesta HTML como en la JSON por defecto.

Aparte de estas propiedades, **podemos proporcionar nuestra propia asignación vista-resolución para *error*,**
anulando la página de etiqueta blanca.

También podemos personalizar los atributos que queremos mostrar en la respuesta incluyendo un bean ErrorAttributes en el contexto. Podemos extender la clase DefaultErrorAttributes proporcionada por Spring Boot para facilitar las cosas:

```
@Componente
public class AtributosDeErrorPersonalizados extends
AtributosDeErrorPredeterminados {

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

        //...

        return errorAttributes;
}   }
```

Si queremos ir más allá y definir (o anular) cómo la aplicación manejará los errores para un tipo de contenido en particular, podemos registrar un bean ErrorController.

De nuevo, podemos hacer uso del *BasicErrorControllcr* por defecto proporcionado por Spring Boot para ayudarnos.

Por ejemplo, imaginemos que queremos personalizar la forma en que nuestra aplicación maneja los errores desencadenados en endpoints XML. Todo lo que tenemos que hacer es definir un método público usando el *@RcqucstMapping* e indicando que produce el tipo de medio *application/xml*:

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

8. Conclusión

En este tutorial se han tratado varias formas de implementar un mecanismo de gestión de excepciones para una API REST en Spring, empezando por el mecanismo más antiguo y continuando con la compatibilidad con Spring 3.2 y hasta 4.x y 5.x.

Como siempre, el código presentado en este capítulo está disponible en Github.

7: Descubrimiento de API REST y HATEOAS

1. Visión general

Este capítulo se centrará en la **descubribilidad de la API REST, HATEOAS** y escenarios prácticos basados en pruebas.

2. Por qué hacer la API descubrible

La descubribilidad de una API es un tema que no recibe suficiente y merecida atención. Como consecuencia, muy pocas API lo hacen bien. También es algo que, si se hace correctamente, puede hacer que la API no solo sea RESTful y utilizable, sino también elegante.

Para entender la descubribilidad, tenemos que entender el hipermedia como motor de estado de la aplicación (HATEOAS). **Esta restricción de una API REST se refiere a la plena descubribilidad de las acciones/transiciones en un recurso de hipermedia (hipertexto en realidad), como único controlador del estado de la aplicación.**

Si la interacción debe ser dirigida por la API a través de la propia conversación, concretamente mediante hipertexto, entonces no puede haber documentación. Eso obligaría al cliente a hacer suposiciones que, de hecho, están fuera del contexto de la API.

En conclusión, el servidor debe ser lo suficientemente descriptivo como para indicar al cliente cómo utilizar la API únicamente a través de hipertexto. En el caso de una conversación HTTP, podríamos conseguirlo a través de la cabecera Link.

¿Qué significa que un servicio REST sea detectable?

A lo largo de esta sección, probaremos rasgos individuales de descubribilidad utilizando Junit, rest-assured y Hamcrest. Dado que el servicio REST ha sido previamente asegurado, cada prueba necesita primero autenticarse antes de consumir la API.

3.1. Descubra los métodos HTTP válidos

Cuando se consume un servicio REST con un método HTTP no válido, la respuesta debe ser 405 MÉTODO NO PERMITIDO.

La API también debe ayudar al cliente a descubrir los métodos HTTP válidos que están permitidos para ese
recurso concreto. **Para ello, podemos utilizar el encabezado HTTP *Allow* en la respuesta:**

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

3.2. Descubrir el URI de un recurso recién creado

La operación de creación de un nuevo Recurso debe incluir siempre en la respuesta el URI del recurso recién creado. Para ello, podemos utilizar el encabezado HTTP *Location*.

Ahora, si el cliente hace un GET en ese URI, el recurso debería estar disponible:

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

La prueba sigue un escenario sencillo: **crear un nuevo recurso *Foo* y, a continuación, utilizar la respuesta HTTP para descubrir la URI en la que el recurso está ahora disponible**. También hace un GET en ese URI para recuperar el recurso y lo compara con el original. Esto es para asegurarse de que se ha guardado correctamente.

3.3. Descubra el URI para OBTENER todos los recursos de ese tipo

Cuando obtenemos un *recurso Foo* en particular, deberíamos ser capaces de descubrir lo que podemos hacer a continuación: podemos listar todos los recursos Foo disponibles. Así, la operación de recuperar un recurso debería incluir siempre en su respuesta la URI donde obtener todos los recursos de ese tipo.

Para ello, podemos utilizar de nuevo el encabezado Enlace:

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

Tenga en cuenta que el código completo de bajo nivel de *cxtractURIByRcl* - responsable de extraer los URI mediante *rcl*
se muestra aquí.

Esta prueba cubre el espinoso tema de las relaciones de enlace en REST: la URI para recuperar todos los recursos
Utiliza la *sistemática rcl="collcction"*.

Este tipo de relación de enlace aún no se ha normalizado, pero ya se utiliza en varios microformatos y se ha propuesto su normalización. El uso de relaciones de enlace no normalizadas abre el debate sobre los microformatos y una semántica más rica en los servicios web RESTful.

Otros URI podrían descubrirse a través de la cabecera *Link*, pero los tipos existentes de relaciones de enlace no permiten más que eso sin pasar a un marcado semántico más rico, como la definición de relaciones de enlace personalizadas, el Atom Publishing Protocol o los microformatos.

Por ejemplo, el cliente debería ser capaz de descubrir el URI para crear nuevos Recursos al hacer un *GET* sobre un Recurso específico. Lamentablemente, no existe ninguna relación de enlace para modelar la semántica *de crcatc*.

Afortunadamente es una práctica estándar que el URI para la creación sea el mismo que el URI para GET todos los recursos de ese tipo, con la única diferencia del método POST HTTP.

5. Conclusión

Hemos visto **cómo una API REST es totalmente descubrible desde la raíz y sin conocimiento previo** - lo que significa que el cliente es capaz de navegar por ella haciendo un GET en la raíz. A partir de ahí, todos los cambios de estado son conducidos por el cliente utilizando las transiciones disponibles y descubribles que la API REST proporciona en representaciones (de ahí *Rcprcscntational Statc Transfcr*).

En este capítulo se han tratado algunos de los rasgos de la descubribilidad en el contexto de un servicio web REST, discutiendo el descubrimiento de métodos HTTP, la relación entre create y get, el descubrimiento de la URI para obtener todos los recursos, etc.

La implementación de todos estos ejemplos y fragmentos de código está disponible en GitHub.

8: Introducción a los HATEOAS de primavera

1. Visión general

Este capítulo explica el proceso de creación de un servicio web REST basado en hipermedia utilizando el proyecto HATEOAS de Spring.

2. Muelle-HATEOAS

El proyecto HATEOAS de Spring es una biblioteca de APIs que podemos utilizar para crear fácilmente representaciones REST que sigan el principio de HATEOAS (Hypertext as the Engine of Application State).

En términos generales, el principio implica que la API debe guiar al cliente a través de la aplicación devolviendo información relevante sobre los siguientes pasos potenciales, junto con cada respuesta.

En este capítulo, vamos a construir un ejemplo utilizando Spring HATEOAS con el objetivo de desacoplar el cliente y el servidor, y teóricamente permitir que la API cambie su esquema URI sin romper los clientes.

En primer lugar, vamos a añadir la dependencia HATEOAS de Spring:

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

Si no estamos usando Spring Boot podemos añadir las siguientes librerías a nuestro proyecto:

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

Como siempre, podemos buscar las últimas versiones de las dependencias starter HATEOAS, spring-hateoas y spring-plugin-core en Maven Central.

A continuación, tenemos el recurso Cliente sin soporte HATEOAS de Spring:

```
public clase Cliente {

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

    // getters y setters estándar
}
```

Por último, la representación del recurso Cliente:

```
{
    "customerId": "10A",
    "customerName": "Jane",
    "customerCompany": "Empresa ABC"
}
```

En un proyecto HATEOAS de Spring, no necesitamos ni buscar el contexto Servlet ni concatenar la variable path con el URI base.

En su lugar, **Spring HATEOAS ofrece tres abstracciones para crear el URI: *ResourceSupport*, *Link* y *ControllerLinkBuilder***. Podemos utilizarlas para crear los metadatos y asociarlos a la representación del recurso.

4.1. Añadir soporte hipermedia a un recurso

El proyecto proporciona una clase base llamada ResourceSupport de la que heredar cuando se crea una clase
representación de recursos:

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

El recurso *Customer* extiende de la clase *ResourceSupport* para heredar el método *add()*. Así, una vez que creamos un enlace, podemos establecer fácilmente ese valor en la representación del recurso sin añadirle ningún campo nuevo.

4.2. Creación de enlaces

**Spring HATEOAS proporciona un objeto *Link* para almacenar los metadatos (ubicación o URI del
recurso).**

En primer lugar, crearemos manualmente un enlace sencillo:

```
Enlace enlace = nuevo Enlace("http://localhost:8080/spring-security-
rest/api/ clientes/10A");
```

El objeto *Enlace* sigue la sintaxis de enlace *Atom* y consta de un *rcl* que identifica la relación con el
y *hrcfattributc*, que es el propio enlace.

Este es el aspecto del recurso Cliente ahora que contiene el nuevo enlace:

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

El URI asociado a la respuesta se califica como enlace *sclf*. La semántica del enlace *sclf* es clara: se trata simplemente de la ubicación canónica en la que se puede acceder al Recurso.

4.3. Crear mejores enlaces

Otra abstracción muy importante que ofrece la biblioteca es **el *ControllerLinkBuilder*, que simplifica** la **creación de URI** evitando los enlaces codificados.

El siguiente fragmento muestra la construcción del autoenlace del cliente utilizando el *ControllcrLinkBuildcr* class:

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

`Echemos` un vistazo:

- el método linkTo() inspecciona la clase del controlador y obtiene su asignación raíz
- el método slash() añade el valor customerId como variable de ruta del enlace
- por último, withSelfMethod() califica la relación como autoenlace

En la sección anterior, hemos mostrado una relación de autorreferencia. Sin embargo, las relaciones
sistemas también pueden implicar otras relaciones.

Por ejemplo, un *customcr* puede tener una relación con los pedidos. Modelemos la clase *Ordcr* como una clase
recurso:

```
1.  public class Pedido extends ResourceSupport
2.      { private String orderId;
3.      private double precio;
4.      private int cantidad;
5.
6.      // getters y setters estándar
7.  }
```

En este punto, podemos extender el CustomerController con un método que devuelva todos los pedidos de un cliente en particular:

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

Nuestro método devuelve un objeto *Rcsourccs* para cumplir con el tipo de retorno HAL, así como un enlace "*_sclf"* para cada uno de los pedidos y la lista completa.

Una cosa importante a notar aquí es que el hipervínculo para los pedidos de clientes depende del mapeo del método *gctOrdcrsForCustomcr()*. Nos referiremos a este tipo de enlaces como enlaces de método y mostraremos cómo el *ControllcrLinkBuildcr* puede ayudar en su creación.

6. Enlaces a los métodos del controlador

El *ControllcrLinkBuildcr* ofrece un amplio soporte para los Controladores MVC de Spring. El siguiente ejemplo muestra cómo construir hipervínculos HATEOAS basados en el método *gctOrdcrsForCustomcr()* de la clase *CustomcrControllcr*:

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

La *función methodOn()* obtiene la asignación de métodos mediante una invocación ficticia al método de destino
en el controlador proxy y establece el *customcrId* como la variable de ruta del URI.

Pongamos el autoenlace y la creación de enlace de método todo junto en un método *gctAllCustomcrs()*:

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

A continuación, vamos a invocar el método getAllCustomers():

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

Y examina el resultado:

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

Dentro de cada representación de recursos, hay un enlace *sclf* y el enlace *allOrdcrs* para extraer todas las
pedidos de un cliente. Si un cliente no tiene pedidos, el enlace para pedidos no aparecerá.

Este ejemplo muestra cómo Spring HATEOAS fomenta el descubrimiento de API en una web de descanso
servicio. **Si el enlace existe, el cliente puede seguirlo y obtener todos los pedidos de un cliente**:

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

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

8. Conclusión

En este capítulo hemos visto cómo **crear un servicio web REST Spring basado en hipermedia utilizando el proyecto HATEOAS de Spring.**

En el ejemplo, vemos que el cliente puede tener un único punto de entrada a la aplicación y se pueden emprender otras acciones basadas en los metadatos de la representación de la respuesta.

Esto permite que el servidor cambie su esquema URI sin romper el cliente. Además, el La aplicación puede anunciar nuevas capacidades poniendo nuevos enlaces o URIs en la representación.

Por último, la aplicación completa de este capítulo puede encontrarse en el proyecto GitHub.

9: Paginación REST en Spring

1. Visión general

Este tutorial se centrará en **la implementación de la paginación en una API REST, utilizando Spring MVC**
y Spring Data.

2. Página como recurso frente a página como representación

La primera pregunta al diseñar la paginación en el contexto de una arquitectura RESTful es si considerar la **página como un Recurso real o sólo como una Representación de Recursos.**

Tratar la propia página como un recurso introduce una serie de problemas, como dejar de poder identificar de forma única los recursos entre llamadas. Esto, unido al hecho de que, en la capa de persistencia, la página no es una entidad propiamente dicha, sino un soporte que se construye cuando es necesario, hace que la elección sea sencilla: **la página forma parte de la representación**.

La siguiente cuestión en el diseño de la paginación en el contexto de REST es **dónde incluir la función**
información de paginación:

- en la ruta URI: */foo/pagc/1*
- la consulta URI: */foo?pagc=1*

Teniendo en cuenta que una **página no es un recurso**, la codificación de la información de la página en el URI no es necesaria.
ya no es una opción.

Vamos a utilizar la forma estándar de resolver este problema mediante la **codificación de la paginación**
en una consulta URI.

Ahora, para la implementación - **el Spring MVC Controller para la paginación es sencillo**:

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

En este ejemplo, estamos inyectando los dos parámetros de consulta, tamaño y página, en el controlador
a través de *@RcqucstParam*.

Alternativamente, podríamos haber utilizado un objeto *Pageable*, que asigna los parámetros de *página*, *tamaño* y *ordenación* automáticamente. Además, la entidad PagingAndSortingRepository proporciona métodos out-of-box que soportan también el uso de Pageable como parámetro.

También estamos inyectando tanto el Http Response como el *UriComponcntsBuildcr* para ayudar con Discoverability - que estamos desacoplando a través de un evento personalizado. Si ese no es el objetivo de la API, puede eliminar el evento personalizado.

Por último - ten en cuenta que el enfoque de este capítulo es sólo el REST y la capa web - para profundizar en la parte de acceso a datos de la paginación puedes consultar este artículo sobre Paginación con Spring Data.

En el ámbito de la paginación, satisfacer la **restricción HATEOAS de REST** significa permitir que el cliente de la API descubra las páginas *siguiente* y *anterior* basándose en la página actual de la navegación. Para ello, **vamos a utilizar la cabecera HTTP *Link*, junto con los tipos de relación de enlace *"next"*, "*prev"*, *"first"* y *"last"***.

En REST, **la descubribilidad es una preocupación transversal**, aplicable no sólo a operaciones específicas sino a tipos de operaciones. Por ejemplo, cada vez que se crea un Recurso, el URI de ese Recurso debe poder ser descubierto por el cliente. Dado que este requisito es relevante para la creación de CUALQUIER Recurso, lo trataremos por separado.

Desacoplaremos estas preocupaciones utilizando eventos. En el caso de la paginación, el evento - *PaginatcdRcsultsRctricvcdEvcnt* - se dispara en la capa del controlador. Luego implementaremos la descubribilidad con un receptor personalizado para este evento.

En resumen, el oyente comprobará si la navegación permite una página *ncxt*, *prcvious*, *flrst* y *last*. Si
añadirá los URI pertinentes a la respuesta como encabezado HTTP "Enlace".

Vayamos ahora paso a paso. El ***UriComponentsBuilder*** pasado desde el controlador contiene únicamente la URL base (el host, el puerto y la ruta de contexto). Por lo tanto, tendremos que añadir las secciones restantes:

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

A continuación, utilizaremos un *StringJoincr* para concatenar cada enlace. Utilizaremos *uriBuildcr* para generar la cadena
URIs. Veamos cómo procederíamos con el enlace a la página *ncxt*:

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

Veamos la lógica del método *constructNcxtPagcUri*:

```
String constructNextPageUri(UriComponentsBuilder uriBuilder, int page,
int size) {
    return uriBuilder.replaceQueryParam(PAGE, page + 1)
      .replaceQueryParam("tamaño", tamaño)
      .construir()
      .encode()
      .toUriString();
}
```

Procederemos de forma similar para el resto de URIs que queramos incluir.

Por último, añadiremos la salida como cabecera de respuesta:

```
response.addHeader("Enlace", linkHeader.toString());
```

Tenga en cuenta que, en aras de la brevedad, he incluido aquí sólo una muestra parcial del código y el código completo.

5. Paginación de prueba

Tanto la lógica principal de la paginación como la descubribilidad están cubiertas por una integración pequeña y centrada
pruebas. Utilizaremos la biblioteca REST-assured para consumir el servicio REST y verificar los resultados.

Estos son algunos ejemplos de pruebas de integración de la paginación; para obtener un conjunto de pruebas completo, consulte el proyecto de GitHub (enlace al final del capítulo):

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

Comprobar que la paginación es detectable por un cliente es relativamente sencillo, aunque hay
mucho terreno que cubrir.

Las pruebas se centrarán en la posición de la página actual en la navegación y en los distintos URI que deberían poder descubrirse desde cada posición:

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

Tenga en cuenta que el código completo de bajo nivel de *cxtractURIByRcl* - responsable de extraer los URI mediante *rcl*
relación está aquí.

Sobre el mismo tema de la paginación y la descubribilidad, **hay que elegir si se permite a un cliente recuperar todos los Recursos del sistema a la vez, o si el cliente debe pedirlos paginados.**

Si se decide que el cliente no puede recuperar todos los recursos con una sola solicitud, y la paginación no es opcional sino obligatoria, existen varias opciones para la respuesta a una solicitud de obtención de todos los recursos. Una opción es devolver un 404 (*Not Found*) y utilizar la cabecera *Link* para hacer que la primera página sea descubrible:

```
Enlace=<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"
```

Otra opción es devolver la redirección - 303 (*Scc Othcr*) - a la primera página. Una ruta más conservadora sería simplemente devolver al cliente un 405 (*Mcthod Not Allowcd*) para la petición GET.

8. Paginación REST con cabeceras HTTP de rango

Una forma relativamente diferente de implementar la paginación es trabajar con las **cabeceras HTTP *Range*** - *Rangc*, *Contcnt-Rangc*, *If-Rangc, Acccpt-Rangcs* - y los **códigos de estado HTTP** - 206 (*Partial Contcnt*), 413 (*Rcqucst Entity Too Largc*), 416 (*Rcqucstcd Rangc Not Satisflablc*).

Una opinión sobre este enfoque es que las extensiones HTTP Range no estaban pensadas para la paginación y que deberían ser gestionadas por el Servidor, no por la Aplicación. Implementar la paginación basada en las extensiones de cabecera HTTP Range es técnicamente posible, aunque no tan común como la implementación discutida en este capítulo.

En Spring Data, si necesitamos devolver unos pocos resultados del conjunto de datos completo, podemos utilizar cualquier método del repositorio *Pagcablc*, ya que siempre devolverá un *Pagc*. Los resultados se devolverán en función del número de página, el tamaño de página y la dirección de ordenación.

Spring Data REST reconoce automáticamente parámetros URL como *page*, *size*, *sort*, etc. Para utilizar los métodos de paginación de cualquier repositorio necesitamos extender *PagingAndSortingRcpository*:

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

Si llamamos a *http://localhost:8080/subjccts* Spring añade automáticamente el *pagc*, *sizc*, *sort* sugerencias de parámetros con la API:

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

Por defecto, el tamaño de página es 20 pero podemos cambiarlo llamando a algo como *http://localhost:8080/subjccts?pagc=10*. Si queremos implementar la paginación en nuestra propia API de repositorio personalizada necesitamos pasar un *Pagcablcparamctcr* adicional y asegurarnos de que la API devuelve un *Pagc*:

```
@RestResource(ruta = "nombreContiene")
public Page<Subject> findByNameContaining(@Param("nombre") String nombre,
Pageable
p);
```

Siempre que añadimos una API personalizada se añade un endpoint /search a los enlaces generados. Así, si
llamamos a http://localhost:8080/subjccts/scarch veremos un endpoint con capacidad de paginación:

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

10. Conclusión

Todas las APIs que implementan PagingAndSortingRepository devolverán una Página. Si necesitamos devolver la lista de resultados de la página, la API getContent() de Page proporciona la lista de registros obtenidos como resultado de la API REST de Spring Data.

10. Conclusión

Este capítulo ilustra cómo implementar la Paginación en una API REST utilizando Spring, y discute cómo configurar y probar la Descubribilidad.

Si quieres profundizar en la paginación en el nivel de persistencia, echa un vistazo a mis tutoriales de paginación en JPA o Hibernate.

La implementación de todos estos ejemplos y fragmentos de código puede encontrarse en el proyecto GitHub.

10: Probar una API REST con Java

1. Visión general

Este tutorial se centra en los principios básicos y la mecánica de las **pruebas de una API REST con pruebas de integración en vivo** (con una carga útil JSON).

El objetivo principal es proporcionar una introducción a la comprobación de la corrección básica de la API - y
vamos a utilizar la última versión de la API REST de GitHub para los ejemplos.

Para una aplicación interna, este tipo de prueba se ejecutará normalmente como un paso tardío en un proceso de Integración Continua, consumiendo la API REST después de que ya se haya desplegado.
Al probar un recurso REST, suele haber algunas responsabilidades ortogonales en las que deben centrarse las pruebas:

- el **código de respuesta** HTTP
- otras **cabeceras** HTTP en la respuesta
- la **carga útil** (JSON, XML)

Cada prueba debe centrarse en una única responsabilidad e incluir una única afirmación. Centrarse en una separación clara siempre tiene ventajas, pero al realizar este tipo de pruebas de caja negra es aún más importante, ya que la tendencia general es escribir escenarios de prueba complejos desde el principio.

Otro aspecto importante de las pruebas de integración es el cumplimiento *del principio de abstracción*: la lógica de una prueba debe escribirse a alto nivel. Detalles como la creación de la solicitud, el envío de la solicitud HTTP al servidor, la gestión de la entrada/salida, etc. no deben realizarse en línea, sino mediante métodos de utilidad.

```
@Prueba
public void givenUserDoesNotExists_whenUserInfoIsRetrieved_
then404IsReceived()
  throws ClientProtocolException, IOException {

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

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

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

Esta es una prueba bastante simple - **verifica que una ruta feliz básica está funcionando**, sin añadir demasiado
mucha complejidad al conjunto de pruebas.

Si por alguna razón falla, no es necesario realizar ninguna otra prueba para esta URL hasta que se solucione.

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

   // Dado
   String jsonMimeType = "application/json";
   HttpUriRequest request = new HttpGet( "https://api.github.com/users/
eugenp" );

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

   // Entonces
   String mimeType = ContentType.getOrDefault(response.getEntity()).
getMimeType();
   assertEquals( jsonMimeType, mimeType );
}
```

Esto garantiza que la respuesta contiene realmente datos JSON.

Como habrás notado, estamos **siguiendo una progresión lógica de pruebas** - primero el Código de Estado de la Respuesta (para asegurar que la solicitud fue OK), luego el Tipo de Medio de la Respuesta, y sólo en la siguiente prueba miraremos la carga útil JSON real.

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

En este caso, sé que la representación por defecto de los recursos de GitHub es JSON, pero normalmente, la cabecera *Contcnt-Typc* de la respuesta debería comprobarse junto con la cabecera *Acccpt* de la petición - el cliente pide un tipo particular de representación a través de *Acccpt*, que el servidor debería respetar.

Vamos a utilizar Jackson 2 para descomponer la cadena JSON sin procesar en una entidad Java de tipo seguro:

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

Sólo estamos utilizando una simple utilidad para mantener las pruebas limpias, legibles y en un alto nivel de abstracción:

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

Fíjate en que Jackson está ignorando propiedades desconocidas que la API de GitHub nos está enviando - eso es simplemente porque la Representación de un Recurso de Usuario en GitHub es bastante compleja - y no necesitamos nada de esa información aquí.

6. Dependencias

Las utilidades y pruebas hacen uso de las siguientes librerías, todas disponibles en Maven central:

- HttpClient
- Jackson 2
- Hamcrest (opcional)

7. Conclusión

Esto es sólo una parte de lo que debe ser el conjunto completo de pruebas de integración. Las pruebas se centran en **garantizar la corrección básica para la API REST,** sin entrar en escenarios más complejos,

Por ejemplo, no se contempla lo siguiente Descubribilidad de la API, consumo de diferentes representaciones para un mismo Recurso, etc.

La implementación de todos estos ejemplos y fragmentos de código se puede encontrar en Github.