(http://baeldung.com)

Convertidores de mensajes Http con Spring Framework

Última modificación: 6 de abril de 2018

por baeldung (http://www.baeldung.com/author/baeldung/)
(http://www.baeldung.com/author/baeldung/)

Primavera (http://www.baeldung.com/category/spring/) +
Spring MVC (http://www.baeldung.com/category/spring-mvc/)

1. Información general

Este artículo describe **cómo configurar *HttpMessageConverter* en Spring** .

En pocas palabras, los convertidores de mensajes se utilizan para ordenar y separar los objetos Java de JSON, XML, etc. a través de HTTP.

Otras lecturas:

Spring MVC Content Returning **Binary Data Formats**

Negociación (http://www.baeldung.com/spring-mvc-content-negotiation-json-xml)

Una guía para configurar la negociación de contenido en una aplicación Spring MVC y para habilitar y deshabilitar las diversas estrategias disponibles.

Read more (http://www.baeldung.com/spring-mvc-content-negotiation-json-xml) →

Image/Media Data with Spring MVC (http://www.baeldung.com/spring-mvc-image-media-data)

The article shows the alternatives for returning image (or other media) with Spring MVC and discusses the pros and cons of each approach.

Read more (http://www.baeldung.com/spring-mvc-image-media-data) →

in a Spring REST API (http://www.baeldung.com/rest-api-with-binary-data-formats)

In this article we explore how to configure Spring REST mechanism to utilize binary data formats which we illustrate with Kryo. Moreover we show how to support multiple data formats with Google Protocol buffers.

Read more (http://www.baeldung.com/spring-rest-api-with-binary-data-formats) →

2. The Basics

2.1. Enable Web MVC

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

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

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

2.2. The Default Message Converters

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

- *ByteArrayHttpMessageConverter* – converts byte arrays
- *StringHttpMessageConverter* – converts Strings
- *ResourceHttpMessageConverter* – converts *org.springframework.core.io.Resource* for any type of octet stream
- *SourceHttpMessageConverter* – converts *javax.xml.transform.Source*
- *FormHttpMessageConverter* – converts form data to/from a *MultiValueMap<String, String>*.

- *Jaxb2RootElementHttpMessageConverter* - convierte objetos Java a / desde XML (solo se agrega si JAXB2 está presente en el classpath)
- *MappingJackson2HttpMessageConverter* - convierte JSON (solo se agrega si Jackson 2 está presente en el classpath)
- *MappingJacksonHttpMessageConverter* - convierte JSON (agregado solo si Jackson está presente en el classpath)
- *AtomFeedHttpMessageConverter* : convierte feeds Atom (solo se agrega si Rome está presente en el classpath)
- *RssChannelHttpMessageConverter* - convierte canales RSS *(solo se agrega si Rome está presente en el classpath)*

3. Comunicación cliente-servidor: solo JSON

3.1. Negociación de contenido de alto nivel

Cada implementación de *HttpMessageConverter* tiene uno o varios tipos MIME asociados.

Al recibir una nueva solicitud, Spring **utilizará el encabezado " *Aceptar* " para determinar el tipo de medio con el** que debe responder.

Luego intentará encontrar un convertidor registrado que sea capaz de manejar ese tipo de medio específico, y lo usará para **convertir la entidad** y devolver la respuesta.

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

Luego buscará un *HttpMessageConverter* que pueda **convertir el cuerpo** enviado por el cliente a un objeto Java.

Vamos a aclarar esto con **un ejemplo rápido** :

- el Cliente envía una solicitud GET a */ foos* con el encabezado ***Aceptar*** *aceptado* en *application / json* - para obtener todos los recursos de *Foo* como Json
- el *Foo* Spring Controller recibe un golpe y devuelve las correspondientes entidades de *Foo* Java
- Spring luego usa uno de los conversores de mensajes Jackson para ordenar las entidades a json

Veamos ahora los detalles de cómo funciona esto y cómo debemos aprovechar las *anotaciones @ResponseBody* y **@** *RequestBody* .

3.2. *@ResponseBody*

@ResponseBody en un método de controlador indica a Spring que **el valor de retorno del método se serializa directamente al cuerpo de la respuesta HTTP** . Como se mencionó anteriormente, el encabezado " *Aceptar* " especificado por el Cliente se usará para elegir el Convertidor Http apropiado para ordenar la entidad.

Veamos un ejemplo simple *:*

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

Ahora, el cliente especificará el encabezado " **Aceptar** " a la **aplicación / json** en la solicitud - ejemplo de comando *curl* :

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

La clase *Foo* :

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

Y el cuerpo de respuesta Http:

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

3.3. *@RequestBody*

@RequestBody se usa en el argumento de un método de controlador: indica a Spring **que el cuerpo de la solicitud de HTTP se deserializa a esa entidad Java en particular** . Como se discutió previamente, el *encabezado* " *Content-Type* " especificado por el Cliente se usará para determinar el convertidor apropiado para esto.

Veamos **un ejemplo** :

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

Ahora, consumamos esto con un objeto JSON: especificamos que " **Content-Type"** sea *application / json* :

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

Recuperamos un 200 OK - una respuesta exitosa:

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

4. Configuración de convertidores personalizados

Podemos **personalizar los conversores de mensajes ampliando la clase WebMvcConfigurerAdapter** y redefiniendo el método *configureMessageConverters* :

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

    @Override
    public void configureMessageConverters(
      List<HttpMessageConverter<?>> converters) {

        messageConverters.add(createXmlHttpMessageConverter());
        messageConverters.add(new MappingJackson2HttpMessageConverter());

        super.configureMessageConverters(converters);
    }
    private HttpMessageConverter<Object> createXmlHttpMessageConverter() {
        MarshallingHttpMessageConverter xmlConverter =
          new MarshallingHttpMessageConverter();

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

        return xmlConverter;
    }
}
```

Y aquí está la configuración XML correspondiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans (http://www.springframework.org/
  xmlns:mvc="http://www.springframework.org/schema/mvc (http://www.springframework.org/sch
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance (http://www.w3.org/2001/XMLSchema-i
  xmlns:context="http://www.springframework.org/schema/context (http://www.springframework
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans (http://www.springframework.org/schema/bea
      http://www.springframework.org/schema/beans/spring-beans.xsd (http://www.springframe
    http://www.springframework.org/schema/context (http://www.springframework.org/schema/c
      http://www.springframework.org/schema/context/spring-context.xsd (http://www.springf
    http://www.springframework.org/schema/mvc (http://www.springframework.org/schema/mvc)
      http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd (http://www.springframe

    <context:component-scan base-package="org.baeldung.web" />

    <mvc:annotation-driven>
        <mvc:message-converters>
           <bean
             class="org.springframework.http.converter.json.MappingJackson2HttpMessageConv

           <bean class="org.springframework.http.converter.xml.MarshallingHttpMessageConve
              <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"

</beans>
```

Tenga en cuenta que **la biblioteca XStream ahora debe estar presente en el classpath** .

También tenga en cuenta que al extender esta clase de soporte, **estamos perdiendo los convertidores de mensajes predeterminados que previamente estaban prerregistrados** , solo tenemos lo que definimos.

Repasemos este ejemplo: estamos creando un nuevo convertidor, el *MarshallingHttpMessageConverter* , y estamos usando el soporte de Spring XStream para configurarlo. Esto permite una gran flexibilidad ya **que estamos trabajando con las API de bajo nivel del marco de clasificación subyacente** , en este caso XStream, y podemos configurar eso como queramos.

Por supuesto, ahora podemos hacer lo mismo con Jackson: al definir nuestro propio *MappingJackson2HttpMessageConverter* ahora podemos establecer un *ObjectMapper* personalizado en este convertidor y configurarlo como lo necesitamos.

En este caso, XStream fue la implementación seleccionada de marshaller / unmarshaller, pero otros como *CastorMarshaller* se pueden usar para referirse a la documentación Spring api para obtener una lista completa de los marshallers disponibles.

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-rest/foos/1
```

5. Usando Spring's *RestTemplate* con Http Message Converters

Además del lado del servidor, Http Message Conversion puede configurarse en el lado del cliente en Spring *RestTemplate* .

Vamos a configurar la plantilla con los encabezados " *Aceptar* " y " *Tipo de contenido* " cuando corresponda, y vamos a tratar de consumir la API REST con clasificación completa y desasignación del Recurso *Foo* , ambos con JSON y con XML .

5.1. Recuperando el recurso sin aceptar el encabezado

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

5.2. Recuperando un recurso con la *aplicación / xml* Aceptar encabezado

Ahora recuperemos explícitamente el Recurso como una Representación XML - vamos a definir un conjunto de Convertidores - del mismo modo que lo hicimos anteriormente - y vamos a configurar estos en el RestTemplate.

Como consumimos XML, vamos a utilizar el mismo marcador de XStream que antes:

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

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

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

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

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

5.3. Recuperando un recurso con *application / json* Accept header

Del mismo modo, ahora consumamos la API REST preguntando por JSON:

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

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

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

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

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

5.4. Actualizar un recurso con tipo de *contenido* XML

Finalmente, también enviemos datos JSON a la API REST y especifiquemos el tipo de medio de esos datos a través del encabezado *Content-Type* :

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

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

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

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

Lo que es interesante aquí es que podemos mezclar los tipos de medios: **estamos enviando datos XML, pero estamos esperando que los datos JSON vuelvan del servidor** . Esto muestra lo poderoso que realmente es el mecanismo de conversión de Spring.

6. Conclusión

En este tutorial, observamos cómo Spring MVC nos permite especificar y personalizar completamente Http Message Converters para **ordenar / desasignar automáticamente las entidades Java hacia y desde XML o JSON** . Esta es, por supuesto, una definición simplista, y hay mucho más que puede hacer el mecanismo de conversión de mensajes, como podemos ver en el último ejemplo de prueba.

También hemos analizado cómo aprovechar el mismo poderoso mecanismo con el cliente *RestTemplate* , lo que conduce a una forma completamente segura de consumir API.

Como siempre, el código presentado en este artículo está disponible en Github (https://github.com/eugenp/tutorials/tree/master/spring-rest) . Este es un proyecto basado en Maven, por lo que debería ser fácil de importar y ejecutar tal cual.

Suscribir ▾ ▲ el más nuevo ▲ más antiguo ▲ el más votado

Sheng

Huésped

Excelente artículo.
¿Cómo puedo clonar este proyecto? ¿Es https://github.com/eugenp/tutorials/tree/master/springrest (https://github.com/eugenp/tutorials/tree/master/spring-rest) la URL correcta? Gracias.

+ 0 − Hace 4 años

Eugen Paraschiv (http://www.baeldung.com/)

Huésped

Sí, ese es el proyecto que cubre este artículo.
Saludos,
Eugen.

+ 0 − Hace 4 años

Pranav Sharma

Huésped

Hola Eugen, gran compañero de artículo !! Yo diría mejor sobre este tema

+ 0 − hace 3 años

Venkat

Huésped

Really nice article with good explanation. I have one use case, could you please help me. class Persons implements Serializable{ private String nameSummary; private String addressSummary; private String bioSummary; } Each field in above class contains json data. Below is my Controller class. @RequestMapping(value = {"/getSummary"}, method = {RequestMethod.POST}, produces=MediaType.APPLICATION_JSON_VALUE) @ResponseBody public Persons getSummary(@RequestBody long empId) { } But in the response i am getting JSON but with escape characters like below: { "nameSummary": "{ "_id" : 3242242 , "name" : "juan"}", "addressSummary": "{ "_id" : 3242243 , "name" : "john"}", "bioSummary": "{ "_id" : 3242244 , "name" : "eric"}"... Read more »

+ 0 − 3 years ago

Venkat

Guest

Hi

Return Type might not be object,, even List is fine for me.

+ 0 − 3 years ago

Eugen Paraschiv (http://www.baeldung.com/)

Guest

Hey Venkat – first – a github project reproducing the problem in a test would make sense.
Second – you can look at the Jackson 2 – JsonFactory – CharacterEscapes to control the way Jackson will escape characters on serialization. Cheers,
Eugen.

+ 0 − 3 years ago

Pokuri

Guest

Helpful atricle. I have a question! When we have redirect(or)forward request then response from one first controller marshalled and then redirected or will redirect and handle the marshalling at the end? I want to wrap final response/returnValue from controller into anotehr class bafore handling it to message converter

+ 0 −

3 years ago

Eugen Paraschiv (http://www.baeldung.com/)

Guest

Hey Pokuri – two notes on this. First – doing a redirect/forward is quite the corner case for an API. Second – I would have to take a quick look at a working example (github) – but from the info you're providing, I'd say that it's the final controller in the chain that will then return the response body, and it's on that body that marshalling will occur. Hope that helps. Cheers,
Eugen.

+ 0 −

3 years ago

Istiti

Guest

Istiti

Hi, I'm new for all this. Your article helps me. But for the json format I have, I can' retrieve what I want. Here is what I did : curl localhost:9090/players. Then the result is : { "_links" : { "search" : { "href" : "http://localhost:9090/players/search" } }, "_embedded" : { "players" : [{ "club" : "club1", "gender" : "H", "licence" : "245891", "lastname" : "lastname1", "firstname" : "firstname1", "clssingle" : "C4", "clsdouble" : "C4", "clsmixte" : "C4", "playerStatus" : "ACTIF", "play1" : "SH1", "play2" : "MX1", "_links" : { "self" : { "href" : "http://localhost:9090/players/10" } } },... Read more »

+ 0 −

3 years ago

Eugen Paraschiv (http://www.baeldung.com/)

Guest

Hey Istiti – first – it's not really possible to solve this here in the comments – I would need to look at some actual code. If you have a github repo that I can look at, feel free to contact me on email and I'd be happy to take a look. Cheers,
Eugen.

+ 1 −

3 years ago

Cuenta

Guest

Bill

Eugen –

Again, great informative article. I'm having and issue with overriding configureMessageConverters in my config class; it never seems to get called. If I add an override for requestMappingHandlerAdapter and call its super, then my configureMessageConverters gets called. So, its working, but why won't the configureMessageConverters get called on its own? I'm using spring 4.0.6.

Bill

+ 0 −

3 years ago

Eugen Paraschiv (http://www.baeldung.com/)

Guest

Hey Bill – without actually looking at your code, I would say that – a config class (that extends WebMvcConfigurerAdapter) and overrides configureMessageConverters should and will definitely be called. I recommend simply starting with the code samples of the article – I just tried it now and the method does get called. Hope that helps. Cheers,
Eugen.

+ 0 −

3 years ago

Load More Comments

CATEGORIES

SPRING (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
HTTPCLIENT 4 TUTORIAL (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
SPRING PERSISTENCE TUTORIAL (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SECURITY WITH SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
THE COURSES (HTTP://COURSES.BAELDUNG.COM)
CONSULTING WORK (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
WRITE FOR BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACT (HTTP://WWW.BAELDUNG.COM/CONTACT)
COMPANY INFO (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
#8898 (NO TITLE) (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
PRIVACY POLICY (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)
EDITORS (HTTP://WWW.BAELDUNG.COM/EDITORS)
KIT DE MEDIOS (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)