org.springframework.http.converter

Interface HttpMessageConverter<T>

All Known Subinterfaces:

GenericHttpMessageConverter<T>

All Known Implementing Classes:

AbstractGenericHttpMessageConverter, AbstractHttpMessageConverter, AbstractJackson2HttpMessageConverter, AbstractJaxb2HttpMessageConverter, AbstractWireFeedHttpMessageConverter, AbstractXmlHttpMessageConverter, AllEncompassingFormHttpMessageConverter, AtomFeedHttpMessageConverter, BufferedImageHttpMessageConverter, ByteArrayHttpMessageConverter, FormHttpMessageConverter, GsonHttpMessageConverter, Jaxb2CollectionHttpMessageConverter, Jaxb2RootElementHttpMessageConverter, MappingJackson2HttpMessageConverter, MappingJackson2XmlHttpMessageConverter, MarshallingHttpMessageConverter, ObjectToStringHttpMessageConverter, ProtobufHttpMessageConverter, ResourceHttpMessageConverter, ResourceRegionHttpMessageConverter, RssChannelHttpMessageConverter, SourceHttpMessageConverter, StringHttpMessageConverter, XmlAwareFormHttpMessageConverter

---

public interface **HttpMessageConverter<T>**

Strategy interface that specifies a converter that can convert from and to HTTP requests and responses.

Since:

3.0

Author:

Arjen Poutsma, Juergen Hoeller

Method Summary

All Methods Instance Methods Abstract Methods

Modifier and Type	Method and Description
boolean	**canRead**(**Class**<?> clazz, **MediaType** mediaType) Indicates whether the given class can be read by this converter.
boolean	**canWrite**(**Class**<?> clazz, **MediaType** mediaType) Indicates whether the given class can be written by this converter.

List<MediaType>	**getSupportedMediaTypes**() Return the list of **MediaType** objects supported by this converter.
T	**read**(Class<? extends T> clazz, **HttpInputMessage** inputMessage) Read an object of the given type from the given input message, and returns it.
void	**write**(T t, **MediaType** contentType, **HttpOutputMessage** outputMessage) Write an given object to the given output message.

Method Detail

canRead

```
boolean canRead(Class<?> clazz,
                MediaType mediaType)
```

Indicates whether the given class can be read by this converter.

Parameters:

clazz - the class to test for readability

mediaType - the media type to read (can be null if not specified); typically the value of a Content-Type header.

Returns:

true if readable; false otherwise

canWrite

```
boolean canWrite(Class<?> clazz,
                 MediaType mediaType)
```

Indicates whether the given class can be written by this converter.

Parameters:

clazz - the class to test for writability

mediaType - the media type to write (can be null if not specified); typically the value of an Accept header.

Returns:

true if writable; false otherwise

getSupportedMediaTypes

```
List<MediaType> getSupportedMediaTypes()
```

Return the list of MediaType objects supported by this converter.

Returns:

the list of supported media types

read

```
T read(Class<? extends T> clazz,
       HttpInputMessage inputMessage)
throws IOException,
       HttpMessageNotReadableException
```

Read an object of the given type from the given input message, and returns it.

Parameters:

clazz - the type of object to return. This type must have previously been passed to the canRead method of this interface, which must have returned true.

inputMessage - the HTTP input message to read from

Returns:

the converted object

Throws:

IOException - in case of I/O errors

HttpMessageNotReadableException - in case of conversion errors

write

```
void write(T t,
           MediaType contentType,
           HttpOutputMessage outputMessage)
    throws IOException,
           HttpMessageNotWritableException
```

Write an given object to the given output message.

Parameters:

t - the object to write to the output message. The type of this object must have previously been passed to the canWrite method of this interface, which must have returned true.

contentType - the content type to use when writing. May be null to indicate that the default content type of the converter must be used. If not null, this media type must have previously been passed to the canWrite method of this interface, which must have returned true.

outputMessage - the message to write to

Throws:

IOException - in case of I/O errors

`HttpMessageNotWritableException` - in case of conversion errors

(http://baeldung.com)

Http Message Converters with the Spring Framework

Last modified: July 19, 2017

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

Spring (http://www.baeldung.com/category/spring/) +

1. Overview

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

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

Further reading:

Spring MVC Content Negotiation (http://www.baeldung.com/spring-mvc-content-negotiation-json-xml)

A guide to configuring content negotiation in a Spring MVC application and on enabling and disabling the various available strategies.

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

Returning 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) →

Binary Data Formats in a Spring REST API (http://www.baeldung.com/spring-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* – converts Java objects to/from XML (added only if JAXB2 is present on the classpath)
- *MappingJackson2HttpMessageConverter* – converts JSON (added only if Jackson 2 is present on the classpath)
- *MappingJacksonHttpMessageConverter* – converts JSON (added only if Jackson is present on the classpath)
- *AtomFeedHttpMessageConverter* – converts Atom feeds (added only if Rome is present on the classpath)
- *RssChannelHttpMessageConverter* – converts RSS feeds *(added only if Rome is present on the classpath)*

3. Client-Server Communication – JSON only

3.1. High Level Content Negotiation

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

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

It will then try to find a registered converter that is capable of handling that specific media type – and it will use it to **convert the entity** and send back the response.

The process is similar for receiving a request which contains JSON information – the framework will **use the "*Content-Type*" header to determine the media type** of the request body.

It will then search for a *HttpMessageConverter* that can **convert the body** sent by the client to a Java Object.

Let's clarify this with **a quick example**:

- the Client sends a GET request to */foos* with the ***Accept*** header set to *application/json* – to get all *Foo* resources as Json
- the *Foo* Spring Controller is hit and returns the corresponding *Foo* Java entities
- Spring then uses one of the Jackson message converters to marshall the entities to json

Let's now look at the specifics of how this works – and how we should leverage the *@ResponseBody* and *@RequestBody* annotations.

3.2. *@ResponseBody*

@ResponseBody on a Controller method indicates to Spring that **the return value of the method is serialized directly to the body of the HTTP Response**. As discussed above, the "*Accept*" header specified by the Client will be used to choose the appropriate Http Converter to marshall the entity.

Let's look at a simple example*:*

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

Now, the client will specify the "**Accept**" header to **application/json** in the request – example *curl* command:

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

The *Foo* class:

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

And the Http Response Body:

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

3.3. *@RequestBody*

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

Let's look at **an example**:

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

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

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

We get back a 200 OK – a successful response:

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

4. Custom Converters Configuration

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

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

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

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

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

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

        return xmlConverter;
    }
}
```

And here is the corresponding XML configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans (http://www.springframework.org/schema/beans)"
  xmlns:mvc="http://www.springframework.org/schema/mvc (http://www.springframework.org/schema/mvc)"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance (http://www.w3.org/2001/XMLSchema-instance)"
  xmlns:context="http://www.springframework.org/schema/context (http://www.springframework.org/schema/context)"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans (http://www.springframework.org/schema/beans)
      http://www.springframework.org/schema/beans/spring-beans.xsd (http://www.springframework.org/schema/beans/spring-bea
    http://www.springframework.org/schema/context (http://www.springframework.org/schema/context)
      http://www.springframework.org/schema/context/spring-context.xsd (http://www.springframework.org/schema/context/spri
    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.springframework.org/schema/mvc/spring-mvc-4

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

    <mvc:annotation-driven>
        <mvc:message-converters>
           <bean
             class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"/>

           <bean class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter">
               <property name="marshaller" ref="xstreamMarshaller" />
               <property name="unmarshaller" ref="xstreamMarshaller" />
           </bean>
        </mvc:message-converters>
    </mvc:annotation-driven>

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

</beans>
```

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

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

Let's go over this example – we are creating a new converter – the *MarshallingHttpMessageConverter* – and we're using the Spring XStream support to configure it. This allows a great deal of flexibility since **we're working with the low level APIs of the underlying marshalling framework** – in this case XStream – and we can configure that however we want.

We can of course now do the same for Jackson – by defining our own *MappingJackson2HttpMessageConverter* we can now set a custom *ObjectMapper* on this converter and have it configured as we need to.

In this case XStream was the selected marshaller/unmarshaller implementation, but others like *CastorMarshaller* can be used to – refer to Spring api documentation for full list of available marshallers.

At this point – with XML enabled on the back end – we can consume the API with XML Representations:

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

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

As well as with the server side, Http Message Conversion can be configured in the client side on the Spring *RestTemplate*.

We're going to configure the template with the "*Accept*" and "*Content-Type*" headers when appropriate and we're going to try to consume the REST API with full marshalling and unmarshalling of the *Foo* Resource – both with JSON and with XML.

5.1. Retrieving the Resource with no Accept Header

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

5.2. Retrieving a Resource with *application/xml* Accept header

Let's now explicitly retrieve the Resource as an XML Representation – we're going to define a set of Converters – same way we did previously – and we're going to set these on the RestTemplate.

Because we're consuming XML, we're going to use the same XStream marshaller as before:

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

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

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

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

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

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

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

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

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

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

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

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

5.4. Update a Resource with XML *Content-Type*

Finally, let's also send JSON data to the REST API and specify the media type of that data via the *Content-Type* header:

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

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

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

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

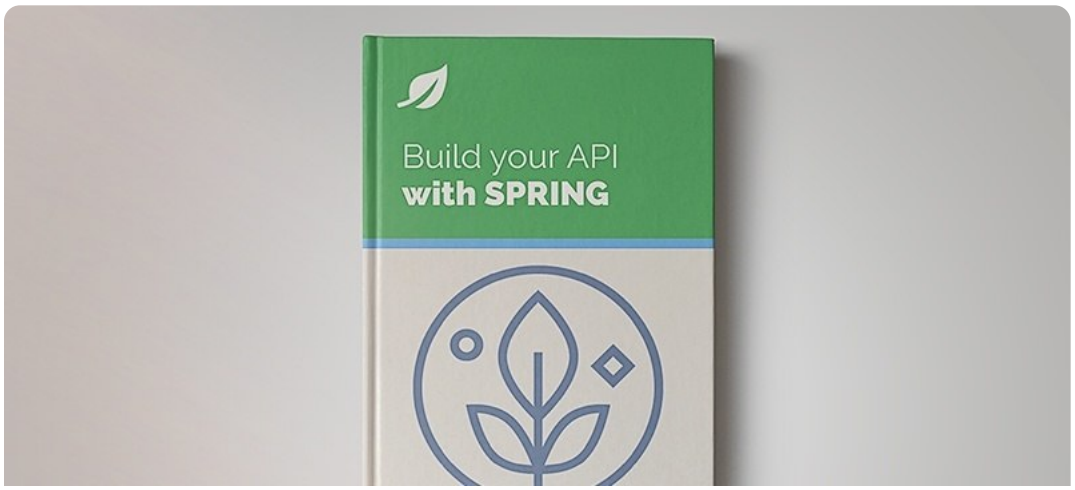
What's interesting here is that we're able to mix the media types – **we are sending XML data but we're waiting for JSON data back from the server**. This shows just how powerful the Spring conversion mechanism really is.

6. Conclusion

In this tutorial, we looked at how Spring MVC allows us to specify and fully customize Http Message Converters to **automatically marshall/unmarshall Java Entities to and from XML or JSON**. This is of course a simplistic definition, and there is so much more that the message conversion mechanism can do – as we can see from the last test example.

We have also looked at how to leverage the same powerful mechanism with the *RestTemplate* client – leading to a fully type-safe way of consuming the API.

As always, the code presented in this article is available over on Github (https://github.com/eugenp/tutorials/tree/master/spring-rest). This is a Maven based project, so it should be easy to import and run as it is.



Sort by: newest|oldest|most voted

Sheng

Guest

Great article.
How Can I Git clone this project? Is https://github.com/eugenp/tutorials/tree/master/spring-rest (https://github.com/eugenp/tutorials/tree/master/spring-rest) the right url? Thanks.

0

3 years 10 months ago

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

Guest

Cheers,
Eugen.

Yes, that is the project covering this article.

0

3 years 10 months ago

Guest

Pranav Sharma

Hi Eugen , Great article mate!! I would say best on this topic

0

3 years 3 months ago

Guest

Venkat

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 6 months ago

Guest

Venkat

Hi

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

0

3 years 6 months ago

Guest

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

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 6 months ago

Guest

Pokuri

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 4 months ago

Guest

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

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 4 months ago

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 2 months 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 2 months ago

Bill

Guest

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

2 years 10 months 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

2 years 9 months ago

Load More Comments (http://www.baeldung.com/spring-httpmessageconverter-rest?wpdParentID=8153)

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/)

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)
TERMS OF SERVICE (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)