Spring.NET REST Client Framework

Reference Documentation

1.0.1

Last Updated 6/27/2011

Arjen Poutsma (Java), Bruno Baia

Chapter 1. Overview

Interacting with HTTP-based services armed only with the relatively low-level classes provided by the .NET Framework is possible, but often leads to your having to repeat the same boilerplate code over and over again as well as evolve your own set of helper methods to translate your objects to and from the text-based world of HTTP requests and responses into the object-based world of .NET consumers of HTTP services. The Spring.Rest project contains the RestTemplate helper class which allows developers to focus on the issues of delivering business value in their applications rather than focusing on low-level plumbing concerns.

RestTemplate provides higher level methods that correspond to each of the six main HTTP methods that make invoking many RESTful services a one-liner and help to enforce REST best practices. It is conceptually similar to other template classes in Spring, such as AdoTemplate and NmsTemplate. It's behavior can be customized by providing callback methods and configuring the IHttpMessageConverters used to marshal objects into the HTTP request body and to unmarshall any response back into an object.

This documentation describes how to use the RestTemplate and its associated IHttpMessageConverters.

Spring.NET REST Client Framework supports the following .NET Frameworks :

- .NET 2.0
- .NET Client Profile 3.5 and 4.0
- Silverlight 3.0 and 4.0
- Windows Phone 7.0

Note

RestTemplate has been available in the Java version of the Spring Framework for many years and has recently been adapted to use proper .NET idioms.

Chapter 2. RestTemplate

Invoking RESTful services in .NET is typically done using the HttpWebRequest class. For common REST operations this approach is too low level as shown below.

```
Uri address = new Uri("http://example.com/hotels/1/bookings");

HttpWebRequest request = WebRequest.Create(address) as HttpWebRequest;
request.Method = "POST";
string requestBody = // create booking request content

byte[] byteData = UTF8Encoding.UTF8.GetBytes(requestBody);
request.ContentLength = byteData.Length;
using (Stream requestStream = request.GetRequestStream())
{
  requestStream.Write(byteData, 0, byteData.Length);
}

using (HttpWebResponse response = request.GetResponse() as HttpWebResponse)
{
  if (response.StatusCode == HttpStatusCode.Created)
  {
    string location = response.Headers["Location"];
    if (location != null)
    {
      Console.WriteLine("Created new booking at: " + location);
    }
  }
}
```

Note

There is another class called WebClient but does not support HTTP headers and HTTP status code/ description.

RestTemplate provides higher level methods that correspond to each of the six main HTTP methods that make invoking many RESTful services a one-liner and enforce REST best practices.

2.1. REST operations

Table 2.1. Overview of RestTemplate methods

HTTP Method	RestTemplate Method
DELETE	Delete(Uri url) and 2 more
GET	GetForObject<T>(Uri url) and 2 more
	GetForMessage<T>(Uri url) and 2 more
HEAD	HeadForHeaders(Uri url) and 2 more
OPTIONS	OptionsForAllow(Uri url) and 2 more
POST	PostForLocation<T>(Uri url, object request) and 2 more
	PostForObject<T>(Uri url, object request) and 2 more
	PostForMessage<T>(Uri url, object request) and 2 more
	PostForMessage(Uri url, object request) and 2 more

PUT	Put(Uri url, object request) and 2 more

The two other methods mentioned take URI template arguments in two forms, either as a `object` variable length argument or an `IDictionary<string, object>`.
For example,

```
// using variable length arguments
string result = restTemplate.GetForObject<string>("http://example.com/hotels/{hotel}/bookings/{booking}", 42,
 21);

// using a IDictionary<string, object>
IDictionary<string, object> vars = new Dictionary<string, object>(1);
vars.Add("hotel", 42);
string result = restTemplate.GetForObject<string>("http://example.com/hotels/{hotel}/rooms/{hotel}", vars);
```

The names of RestTemplate methods follow a naming convention, the first part indicates what HTTP method is being invoked and the second part indicates what is returned. For example,

- The method `GetForObject<T>()` will perform a GET, and return the HTTP response body converted into an object type of your choice.
- The method `PostForLocation()` will do a POST, converting the given object into a HTTP request and return the response HTTP Location header where the newly created object can be found.
- The method `PostForMessage<T>()` will do a POST, converting the given object into a HTTP request and return the full HTTP response message composed of the status code and description, the response headers and the response body converted into an object type of your choice.

The request object to be POSTed or PUTed, may be a HttpEntity instance in order to add additional HTTP headers. An example is shown below.

```
Booking requestBody = // create booking request content
HttpEntity entity = new HttpEntity(requestBody);
entity.Headers["MyRequestHeader"] = "MyValue";

template.PostForLocation("http://example.com/hotels/{id}/bookings", entity, 1);
```

Note

These operations are synchronous and are not available for Silverlight and Windows Phone because all related network calls have to be asynchronous.

2.1.1. Asynchronous operations

All REST operations are also available for asynchronous calls. The asynchronous methods are suffixed by the word 'Async' based on common .NET naming conventions.

Note, that asynchronous call were initially added to support Silverlight and Windows Phone, but nothing prevents you from using them for Windows Forms and WPF applications to not freeze the UI when invoking RESTful services.
An example using an asynchronous method is shown below :

```
template.GetForObjectAsync<string>("http://example.com/hotels/bookings",
  r =>
  {
    if (r.Error != null)
    {
      Console.WriteLine(r.Error);
    }
    else
    {
      Console.WriteLine(r.Response);
    }
  });
```

2.2. Configuring the RestTemplate

2.2.1. Base address

In some cases it may be useful to set up the base url of the request once. This is possible by setting the base address in the constructor or by setting the property `BaseAddress`.
For example:

```
RestTemplate template = new RestTemplate("http://example.com");
Booking booking1 = template.GetForObject<Booking>("/hotels/{id}/bookings", 1);
Booking booking2 = template.GetForObject<Booking>("/hotels/{id}/bookings", 2);
```

2.2.2. HTTP message converters

Objects passed to and returned from REST operations are converted to and from HTTP messages by IHttpMessageConverter instances.

Converters for the main mime types are registered by default, but you can also write your own converter and register it via the `MessageConverters` property.

The default converter instances registered with the template, depending of the target Framework, are ByteArrayHttpMessageConverter, StringHttpMessageConverter, FormHttpMessageConverter, XmlDocumentHttpMessageConverter, XElementHttpMessageConverter, Atom10FeedHttpMessageConverter and Rss20FeedHttpMessageConverter.

You can override these defaults using the `MessageConverters` property. This is required if you want to use the XmlSerializableHttpMessageConverter/DataContractHttpMessageConverter or JsonHttpMessageConverter/NJsonHttpMessageConverter.

For example :

```
// Add a new converter to the default list
RestTemplate template = new RestTemplate("http://twitter.com");
template.MessageConverters.Add(new JsonHttpMessageConverter());
```

See HTTP message conversion chapter for detailed description of each converter.

2.2.3. Error handling

In case of an exception processing the HTTP method, an exception of the type RestClientException will be thrown. The interface IResponseErrorHandler allows you to determine whether a particular response has an error. The default implementation throws an exception when a HTTP client or server error happens (HTTP status code 4xx or 5xx).

The default behavior can be changed by plugging in another implementation into the RestTemplate via the `ErrorHandler` property. The example below shows a custom IResponseErrorHandler implementation which gives the user total control over the response.

```
public class MyResponseErrorHandler : IResponseErrorHandler
```

```
{
  public bool HasError(IClientHttpResponse response)
  {
    return false;
  }

  public void HandleError(IClientHttpResponse response)
  {
    // This method should not be called because HasError returns false.
    throw new InvalidOperationException();
  }
}

RestTemplate template = new RestTemplate("http://example.com");
template.ErrorHandler = new MyResponseErrorHandler();

HttpResponseMessage responseMessage = template.PostForMessage("/notfound", null); // throw
 HttpClientErrorException with default implementation
if (responseMessage.StatusCode == HttpStatusCode.NotFound)
{
  // ...
}
```

2.2.4. Request factory

RestTemplate uses a request factory to create instances of the IClientHttpRequest interface. Default implementation uses the .NET Framework class HttpWebRequest. This can be overridden by specifying an implementation of IClientHttpRequestFactory via the `RequestFactory` property.

The default implementation WebClientHttpRequestFactory uses an instance of HttpWebRequest which can be configured with credentials information or proxy settings. An example setting the proxy is shown below :

```
WebClientHttpRequestFactory requestFactory = new WebClientHttpRequestFactory();
requestFactory.Proxy = new WebProxy("http://proxy.example.com:8080");
requestFactory.Proxy.Credentials = new NetworkCredential("userName", "password", "domain");

RestTemplate template = new RestTemplate("http://example.com");
template.RequestFactory = requestFactory;
```

2.2.4.1. Silverlight support

In Silverlight, HTTP handling can be performed by the browser or the client. See How to: Specify Browser or Client HTTP Handling on MSDN.
By default, WebClientHttpRequestFactory will use the browser HTTP stack for HTTP methods GET and POST, and force the client HTTP stack for other HTTP methods.
This can be overridden by setting the `WebRequestCreator` property.

```
RestTemplate template = new RestTemplate("http://example.com");
((WebClientHttpRequestFactory)rt.RequestFactory).WebRequestCreator = WebRequestCreatorType.ClientHttp;
```

2.2.5. Request interceptors

RestTemplate allows you to intercept HTTP request creation and/or execution. You can create your own interceptor and register it via the `RequestInterceptors` property.
Four types of interceptors are provided :

- IClientHttpRequestFactoryInterceptor will intercept request creation, allowing to modify the HTTP URI and method, and to customize the newly created request.

- IClientHttpRequestBeforeInterceptor will intercept request before its execution, allowing to modify the HTTP headers and body. This interceptor supports both synchronous and asynchronous requests.

- IClientHttpRequestSyncInterceptor will intercept synchronous request execution, giving total control of the execution (error management, logging, perf, etc.).
- IClientHttpRequestAsyncInterceptor will intercept asynchronous request execution, giving total control of the execution (error management, logging, perf, etc.).

An example of an interceptor measuring HTTP request execution time is shown below.

```
public class PerfRequestSyncInterceptor : IClientHttpRequestSyncInterceptor
{
  public IClientHttpResponse Execute(IClientHttpRequestSyncExecution execution)
  {
    Stopwatch stopwatch = Stopwatch.StartNew();
    IClientHttpResponse response = execution.Execute();
    stopwatch.Stop();

    Console.WriteLine(String.Format(
      "Sync {0} request for '{1}' took {2}ms and resulted in {3:d} - {3}",
      execution.Method,
      execution.Uri,
      stopwatch.ElapsedMilliseconds,
      response.StatusCode));

    return response;
  }
}

RestTemplate template = new RestTemplate();
template.RequestInterceptors.Add(new PerfRequestSyncInterceptor());
```

Note that you can also make PerfRequestSyncInterceptor implement IClientHttpRequestAsyncInterceptor to support both synchronous and asynchronous requests.

2.2.6. Using the Spring.NET container

RestTemplate, as any other class can be configured in the Spring.NET container.
For example, using XML configuration :

```
<object id="RestTemplate" type="Spring.Rest.Client.RestTemplate, Spring.Rest">
  <constructor-arg name="baseAddress" value="http://example.com"/>
  <property name="ErrorHandler">
    <object type="MyErrorHandler"/>
  </property>
  <property name="MessageConverters">
    <list>
      <object type="MyHttpMessageConverter"/>
    </list>
  </property>
  <property name="RequestInterceptors">
    <list>
      <object type="MyClientHttpRequestInterceptor"/>
    </list>
  </property>
  <property name="RequestFactory.UseDefaultCredentials" value="true"/>
  <property name="RequestFactory.Timeout" value="10000"/>
</object>
```

2.3. Authenticating requests

2.3.1. Using HttpWebRequest .NET class

The default request factory implementation WebClientHttpRequestFactory, that uses the HttpWebRequest class, can be used to authenticate the HTTP request. Supported authentication schemes include Basic, Digest, Negotiate (SPNEGO), Kerberos, NTLM, and Certificates.

```
WebClientHttpRequestFactory requestFactory = new WebClientHttpRequestFactory();
requestFactory.Credentials = new NetworkCredential("userName", "password", "domain");

RestTemplate template = new RestTemplate("http://example.com");
template.RequestFactory = requestFactory;
```

You can also directly cast the default request factory to WebClientHttpRequestFactory :

```
RestTemplate template = new RestTemplate("http://example.com");
((WebClientHttpRequestFactory)template.RequestFactory).UseDefaultCredentials = true;
```

2.3.2. Basic authentication

Basic authentication is supported by the WebClientHttpRequestFactory (see previous section), but this implementation will wait the challenge response from server before to send the 'Authorization' header value. As this can be an issue in some cases, Spring provides a custom request interceptor named BasicSigningRequestInterceptor that forces Basic authentication. This is shown below

```
RestTemplate template = new RestTemplate("http://example.com");
template.RequestInterceptors.Add(new BasicSigningRequestInterceptor("login", "password"));
```

2.3.3. OAuth

This will be supported in a future version based on the Spring.Social Java project. The implementation is based on request interceptors.

2.4. Dealing with HTTP messages

Besides the REST operations described in the previous section, the RestTemplate also has the `Exchange()` method, which can be used for arbitrary HTTP method execution based on HTTP messages. The method takes as arguments the HTTP request message composed of the request Uri, the HTTP method and the HTTP entity (headers and body) and returns the HTTP response message composed of the status code, status description and the HTTP entity (headers and body).

```
HttpResponseMessage<T> Exchange<T>(Uri url, HttpMethod method, HttpEntity requestEntity) where T : class;

// also has 2 overloads for URI template based URL.
```

Perhaps most importantly, the `Exchange()` method can be used to add request headers and read response headers for every REST operation. For example:

```
HttpEntity requestEntity = new HttpEntity();
requestEntity.Headers["MyRequestHeader"] = "MyValue";

HttpResponseMessage<string> response = template.Exchange<string>("/hotels/{hotel}", HttpMethod.GET,
 requestEntity, 42);

string responseHeader = response.Headers["MyResponseHeader"];
string body = response.Body;
HttpStatusCode statusCode = response.StatusCode;
string statusDescription = response.StatusDescription;
```

In the above example, we first prepare a request message that contains the `MyRequestHeader` header. We then retrieve the response message, and read the `MyResponseHeader`.

2.5. Under the hood...

Last but not least, the `Execute()` method is behind everything RestTemplate does.

```
T Execute<T>(Uri url, HttpMethod method, IRequestCallback requestCallback, IResponseExtractor<T>
 responseExtractor) where T : class;

// also has 2 overloads for URI template based URL.
```

The IRequestCallback interface is defined as

```
public interface IRequestCallback
{
  void DoWithRequest(IClientHttpRequest request);
}
```

The IResponseExtractor<T> interface is defined as

```
public interface IResponseExtractor<T> where T : class
{
  T ExtractData(IClientHttpResponse response);
}
```

The `Execute` method allow you to manipulate the request/response headers, write to the request body and read from the response body.

Note, when using the execute method you do not have to worry about any resource management, the template will always close the request and handle any errors. Refer to the API documentation for more information on using the execute method and the meaning of its other method arguments.

Chapter 3. HTTP Message Conversion

Objects passed to and returned from some REST operations are converted to HTTP requests and from HTTP responses by IHttpMessageConverters. The IHttpMessageConverter interface is shown below to give you a better feel for its functionality.

```
public interface IHttpMessageConverters
{
  // Indicate whether the given class and media type can be read by this converter.
  bool CanRead(Type type, MediaType mediaType);

  // Indicate whether the given class and media type can be written by this converter.
  bool CanWrite(Type type, MediaType mediaType);

  // Return the list of MediaType objects supported by this converter.
  IList<MediaType>SupportedMediaTypes { get; }

  // Read an object of the given type from the given input message, and returns it.
  T Read<T>(IHttpInputMessage message) where T : class;

  // Write an given object to the given output message.
  void Write(object content, MediaType contentType, IHttpOutputMessage message);
}
```

Concrete implementations for the main media (mime) types are provided in the framework and most are registered by default with the RestTemplate on the client-side.

Table 3.1. Supported HTTP message converters by Framework

Supported MIME types	.NET 2.0	.NET 3.5	.NET 4.0	Silverlight 3.0	Silverlight 4.0	Windows Phone 7.0
ByteArrayHttpMessageConverter	✓	✓	✓	✓	✓	✓
StringHttpMessageConverter	✓	✓	✓	✓	✓	✓
FileInfoHttpMessageConverter	✓	✓	✓	✓	✓	✓
FormHttpMessageConverter	✓	✓	✓	✓	✓	✓
XmlDocumentHttpMessageConverter	✓	✓	✓			
XElementHttpMessageConverter		✓	✓	✓ (1)	✓ (1)	✓
XmlSerializableHttpMessageConverter	✓					
DataContractHttpMessageConverter		✓	✓	✓	✓	✓
JsonHttpMessageConverter		✓	✓	✓	✓	✓
NJsonHttpMessageConverter Json.NET (Newtonsoft.Json)	✓ (3)	✓ (3)	✓ (3)	✓ (3)	✓ (3)	✓ (3)

Atom10FeedHttpMessageConverter	✓	✓	✓ (2)	✓ (2)	
Rss20FeedHttpMessageConverter	✓	✓	✓ (2)	✓ (2)	

The implementations of IHttpMessageConverters are described in the following sections.
For all converters a default media type is used but can be overridden by setting the `SupportedMediaTypes` property.

Refer to the API documentation for more information and to unit tests for example scenarios about each converters.

3.1. ByteArrayMessageConverter

Supports all the .NET Framework versions.

An IHttpMessageConverter implementation that can read and write byte arrays from the HTTP request and response.
By default, this converter supports all media types (`'*/*'`), and writes with a `Content-Type` of `'application/octet-stream'`.

3.2. StringHttpMessageConverter

Supports all the .NET Framework versions.

An IHttpMessageConverter implementation that can read and write Strings from the HTTP request and response.
By default, this converter supports all text media types (`'text/*'`), and writes with a `Content-Type` of `'text/plain'`.

3.3. FileInfoHttpMessageConverter

Supports all the .NET Framework versions.

An IHttpMessageConverter implementation that can write file content to the HTTP request.
By default, this converter supports all text media types (`'*/*'`).

A mapping between file extension and mime types is used to determine the Content-Type of written files.
If no `Content-Type` is available, `'application/octet-stream'` is used.

Table 3.2. Default mime mapping registered with the converter

File extension	Mime type
.bmp	image/bmp
.jpg	image/jpg
.jpeg	image/jpeg
.pdf	application/pdf
.png	image/png

.tif	image/tiff
.txt	text/plain
.zip	application/x-zip-compressed

You can override these defaults using the `MimeMapping` property.

You can also set the `Content-Type` header directly as shown in the following example with a file upload :

```
RestTemplate template = new RestTemplate(); // FormHttpMessageConverter is configured by default
IDictionary<string, object> parts = new Dictionary<string, object>();
HttpEntity entity = new HttpEntity(new FileInfo(@"C:\myFile.xls"));
entity.Headers["Content-Type"] = "application/vnd.ms-excel";
parts.Add("file", entity);
template.PostForLocation("http://example.com/myFileUpload", parts);
```

3.4. FormHttpMessageConverter

Supports all the .NET Framework versions.

An IHttpMessageConverter implementation that can handle form data from the HTTP request and response.
By default, this converter reads and writes the media type `'application/x-www-form-urlencoded'`.
Form data is read from and written into a `NameValueCollection`.

```
RestTemplate template = new RestTemplate(); // FormHttpMessageConverter is configured by default
NameValueCollection form = new NameValueCollection();
form.Add("field 1", "value 1");
form.Add("field 2", "value 2");
form.Add("field 2", "value 3");
template.PostForLocation("http://example.com/myForm", form);
```

This converter also writes multipart form data (i.e. file uploads) defined by the `'multipart/form-data'` media type.
Multipart form data is written into an `IDictionary<string, object>`.

```
RestTemplate template = new RestTemplate(); // FormHttpMessageConverter is configured by default
IDictionary<string, object> parts = new Dictionary<string, object>();
parts.Add("field 1", "value 1");
parts.Add("file", new FileInfo(@"C:\myFile.jpg"));
template.PostForLocation("http://example.com/myFileUpload", parts);
```

When writing multipart, this converter uses other IHttpMessageConverter to write the respective MIME parts.
By default, basic converters StringHttpMessageConverter and FileInfoHttpMessageConverter are registered to support `String` and `FileInfo` part types. These can be overridden by setting the `PartConverters` property.

3.5. XmlDocumentHttpMessageConverter

Supports .NET Framework 2.0, 3.5 and 4.0.

An IHttpMessageConverter implementation that can read and write XML using the .NET Framework class `XmlDocument`.
By default, this converter supports media types `'text/xml'`, `'application/xml'`, and `'application/*-xml'`.

3.6. XElementHttpMessageConverter

Supports .NET Framework 3.5, 4.0. and Windows Phone.

Supports Silverlight using `Spring.Http.Converters.Xml.Linq.dll`.

An IHttpMessageConverter implementation that can read and write XML using the .NET Framework class `XElement` (Linq to Xml support).
By default, this converter supports media types `'text/xml'`, `'application/xml'`, and `'application/*-xml'`.

3.6.1. Silverlight support

This converter requires a reference to `System.Xml.Linq.dll` which is not part of the runtime installed with the Silverlight plug-in.
The dll is included with the SDK, and will be packaged into the Xap file downloaded by the client.
To optimize weight of the `Spring.Rest.dll` file, XElementHttpMessageConverter is available as an external dll `Spring.Http.Converters.Xml.Linq.dll`.

Taking as an example the RestSilverlightQuickstart, the XAP file size increases from 29,5Kb to 76,4Kb using Linq to Xml, hence the importance of providing this converter separately.

To use this converter with the RestTemplate, you will have to add it to the message converters it :

```
template.MessageConverters.Add(new XElementHttpMessageConverter());
```

3.7. XmlSerializableHttpMessageConverter

Supports .NET Framework 2.0, 3.5, 4.0 and Windows Phone.

An IHttpMessageConverter implementation that can read and write XML from .NET classes.
The converter uses the `XmlSerializer` .NET Framework class.
By default, this converter supports media types `'text/xml'`, `'application/xml'`, and `'application/*-xml'`.

3.8. DataContractHttpMessageConverter

Supports .NET Framework 3.5, 4.0, Silverlight and Windows Phone.

An IHttpMessageConverter implementation that can read and write XML from .NET classes.
The converter uses the `DataContractSerializer` .NET Framework class.
By default, this converter supports media types `'text/xml'`, `'application/xml'`, and `'application/*-xml'`.

3.9. JsonHttpMessageConverter

Supports .NET Framework 3.5, 4.0, Silverlight and Windows Phone.

An IHttpMessageConverter implementation that can read and write JSON from .NET classes.
The converter uses the `DataContractJsonSerializer` .NET Framework class.
By default, this converter supports media type `'application/json'`.

3.10. NJsonHttpMessageConverter

Supports all the .NET Framework versions using `Spring.Http.Converters.NJson.dll`.

An IHttpMessageConverter implementation that can read and write JSON using the Json.NET (Newtonsoft.Json) library.
By default, this converter supports media type `'application/json'`.

Unlike `DataContractJsonSerializer` .NET Framework class used by JsonHttpMessageConverter, this library supports getting/setting values from JSON directly, without the need to deserialize/serialize to a .NET class.

To use this converter with the RestTemplate, you will have to add it to the message converters list :

```
template.MessageConverters.Add(new NJsonHttpMessageConverter());
```

See the Windows Phone quick start for an example of use.

3.11. Feed converters

Supports .NET Framework 3.5, 4.0.
Supports Silverlight using `Spring.Http.Converters.Feed.dll`.

3.11.1. Atom10FeedHttpMessageConverter

An IHttpMessageConverter implementation that can read and write Atom feeds using .NET Framework classes `SyndicationFeed` and `SyndicationItem`.
By default, this converter supports media type `'application/atom+xml'`.

3.11.2. Rss20FeedHttpMessageConverter

An IHttpMessageConverter implementation that can read and write RSS feeds using .NET Framework classes `SyndicationFeed` and `SyndicationItem`.
By default, this converter supports media type `'application/rss+xml'`.

3.11.3. Silverlight support

These converters require a reference to `System.ServiceModel.Syndication.dll` which is not part of the runtime intalled with the Silverlight plug-in.
The dll is included with the SDK, and will be packaged into the Xap file downloaded by the client.
To optimize weight of the `Spring.Rest.dll` file, feed converters are available as an external dll `Spring.Http.Converters.Feed.dll`.

Taking as an example the RestSilverlightQuickstart, the XAP file size increases from 29,5Ko to 189Kb using feeds, hence the importance of providing these converters separately.

To use these converters with the RestTemplate, you will have to add them to the message converters list :

```
template.MessageConverters.Add(new Atom10FeedHttpMessageConverter());
template.MessageConverters.Add(new Rss20FeedHttpMessageConverter());
```