How To Consume a WebAPI with RestSharp

The open source library RestSharp is designed to make consuming APIs from .NET easy and work across platforms.

By Ondrej Balas **10/08/2015**

GET CODE DOWNLOAD

Interfacing with a public API from a .NET application is something that many developers find themselves needing to do. They can take several different approaches, but my favorite is an open source library called RestSharp (restsharp.org). RestSharp makes it easy by abstracting away some of the complication of dealing with raw HTTP requests and providing a developer-friendly way of getting at what's important: the data.

Communicating with APIs

At its core, communicating with an API simply involves creating and sending an HTTP request to the API, and then doing something with the response. Most modern APIs will return JSON, though you may run into APIs that return only XML. Many also allow the caller to specify which format they'd like to receive data in by specifying the preferred format in the Accept header of the HTTP request.

Crafting an HTTP request can be done in several ways, with the most common way being the use of System.Net.Http.HttpClient or the simpler System.Net.WebClient. System.Net.HttpWebRequest is also worth mentioning, though it is flagged as obsolete and not recommended unless you're doing something specific that requires lower-level access.

HttpClient is the newest one, requiring the Microsoft .NET Framework 4.5 and exposing only asynchronous methods for sending requests. It's the most similar to RestSharp in that it's easy to use (not overly abstrac and not overly complicated either), but because it requires the .NET Framework 4.5, not every application can take advantage of it.

WebClient is very simple and a great choice for just making GET requests, but once you need to start usin verbs like PUT or DELETE the code starts to get quite a bit messier.

RestSharp

The reason I prefer to use RestSharp is it's a library that is both easy to use and available across many platforms. It needs only .NET Framework 3.5 (though there is a .NET Framework 2.0 fork of RestSharp on GitHub), and supports Windows Phone and Xamarin. It also has some features allowing automatic deserialization of responses, making it a pleasure to work with.

To demonstrate RestSharp, I've created a sample solution, which you can find in the code download for this article, that contains two projects. The first is a Web API project to which I've added a ProductsController. The controller supports all the expected REST calls, persisting changes to an in-memory list of products abstracted behind the interface, like so:

```
public interface IProductRepository
{
  IEnumerable<Product> GetAll();
  Product Get(int id);
  void Create(Product product);
  void Update(int id, Product product);
  void Delete(int id);
}
```

The second is a console application that calls the API using RestSharp. GET requests

Once you've downloaded and referenced RestSharp (from NuGet or otherwise), to begin using it you'll need an instance of a RestClient. It has a default constructor, but a good practice is to pass in your base URL as either a string or Uri object. Because my Web API is listening on port 9075, I will instantiate a RestClient like this:

```
private RestClient client = new RestClient("http://localhost:9075/api/");
```

The next step is to instantiate a RestRequest. This can also be created in a number of ways, but the recommended way is to pass in the resource and method. For a Get request I create my RestRequest like this:

```
RestRequest request = new RestRequest("Products", Method.GET);
```

The final step is to use the client to execute the request, which I do like this:

```
IRestResponse<List<Product>> response = client.Execute<List<Product>>(request);
```

Notice that I'm using a generic overload of Execute, which examines the format of the response and tries t convert it to the given type. There are a lot of other methods that execute the request, too. Some specify th HTTP method, while others allow for asynchronous requests.

The response has quite a few properties that can be used to get information about the response, but to get the data you'll want to access the Data property. Doing this, my GetProducts method ends up looking like this:

```
public List<Product> GetProducts()
{
  RestRequest request = new RestRequest("Products", Method.GET);
  IRestResponse<List<Product>> response = client.Execute<List<Product>>(request);
  return response.Data;
}
```

And the corresponding Get method on my ProductsController looks like this:

```
// GET api/products
public IEnumerable<Product> Get()
{
  return _repository.GetAll();
}
```

Other Verbs

Some requests, like POST requests, for example, will require a body to be sent as part of the request. To create a product through the API, I use the following method:

```
public void Create(Product product)
{
  var request = new RestRequest("Products", Method.POST);
  request.AddJsonBody(product);
  client.Execute(request);
}
```

I first instantiate a RestRequest but before executing it I use the AddJsonBody method, passing in an instance of the product I want to create. This will cause the execution of the request to serialize my product to JSON before sending it to the API. The Execute method that I use on the client for this is the non-generic one, because I don't expect there to be a response that should be deserialized to anything specific.

Updates and deletes work similarly, but because my API expects the Id of the object to be updated or deleted in the URL, I have to append it to the resource string with which I create the RestRequest. My Update and Delete methods are shown here:

```
public void Update(int id, Product product)
{
  var request = new RestRequest("Products/" + id, Method.PUT);
  request.AddJsonBody(product);
```

```
public void Delete(int id)
{
  var request = new RestRequest("Products/" + id, Method.DELETE);
  client.Execute(request);
}
```

If you're interested in seeing what the API controller looks like, it's listed in its entirety in **Listing 1**.

Listing 1: The ProductsController Class

```
public class ProductsController : ApiController
{
  private readonly IProductRepository _repository;

  public ProductsController()
  {
    _repository = new InMemoryProductRepository();
  }

  // GET api/products
  public IEnumerable<Product> Get()
  {
    return _repository.GetAll();
  }

  // GET api/products/5
  public Product Get(int id)
  {
```

Cookies and Other Headers

Some APIs require cookies for authentication. To enable cookie support with RestSharp, after newing up th RestClient you'll need to set its CookieContainer property. This can be done like this:

```
RestClient client = new RestClient("http://localhost:9075/api/");
client.CookieContainer = new CookieContainer();
```

The CookieContainer can then be filled with pre-set cookies, but keep in mind that cookies will be added or removed as specified by the responses to any HTTP requests made.

Additional headers can be added to a request by AddHeader method on the request. The method expects a name and value, which will be sent as part of the HTTP request when it's executed. If I wanted to add a header to my request, I could add one like this:

```
request.AddHeader("User-Agent", "RestSharpDemo");
```

Wrapping Up

Hopefully this article will serve as a useful guide to getting started with RestSharp. The included code

About the Author

Ondrej Balas owns UseTech Design, a Michigan development company focused on .NET and Microsoft technologies. Ondrej is a Microsoft MVP in Visual Studio and Development Technologies and an active contributor to the Michigan software development community. He works across many industries -- finance healthcare, manufacturing, and logistics -- and has expertise with large data sets, algorithm design, distributed architecture, and software development practices.