RAMONE 1.1

A Web API and REST Service Client for C#

Jørn Wildt 2012 - 2013

Content

Introduction

Ramone is a C\# client side library that simplifies access to HTTP based Web API s and REST services. It has a strong focus on REST and implements elements of the Uniform Interface as first class citizens of the API.

This means natural support for

- URIs as identifiers for resources.
- The standard HTTP methods GET, POST, PUT and more.
- Multiple media types (XML, JSON, HTML, ATOM and more).
- User defined media types.
- Hyper media controls (linking and key/value forms).
- Proper status code handling (redirects, authentication and so on).

What Ramone does is to wrap the inner workings of HTTP (using .NET's `HttpWebRequest`/`HttpWebResponse` classes) and make encoding and decoding easier through the use of codecs for the various formats used on the web.

Ramone supports plain XML, JSON, and Atom out of the box, but where it really excels is when it comes to working with user defined objects, custo media types and hyper media elements.

Basic operations

The very first GET of a resource

The simplest possible operation we can do with Ramone is a GET of a resource from a well known URL and then decode that resource into a client side object.

As an example we can GET the Twitter time line for user JornWildt (see the Twitter specification at https://dev.twitter.com/docs/api/1/get/statuses/user_timeline). This piece of code is also available as a demo Visual Studio project in the source code directory (https://github.com/JornWildt/Ramone/tree/master/TwitterDemo):

```
// Fixed URL to timeline of JornWildt
string url = "/1/statuses/user_timeline.json?screen_name=JornWildt";

// All interaction with Ramone goes through a session
ISession session
  = RamoneConfiguration.NewSession(new Uri("https://api.twitter.com"));

// Create a Ramone request by binding URL to session
Request request = session.Bind(url);

// GET response from Twitter
using (Response response = request.Get())
{
  // Extract payload as a C# dynamic created from the JSON response
  dynamic timeline = response.Body;

  // Write response to console ...
  Console.WriteLine("This is the timeline for JornWildt:");
  foreach (dynamic tweet in timeline)
    Console.WriteLine("* {0}.", tweet.text);
}
```

Lets go through that code step by step and see what is going on:

- The first step creates a session. Sessions are containers for things like authorization, base URL and cookies and must always be referenced somehow when creating a new request.
- The next step binds the Twitter URL to the session and creates a request.
- Then the request is activated by calling one of the HTTP methods – in this case GET.
- The “using” statement is there to ensure proper release of connections for the .NET connection pool management.
- Ramone takes care of content negotiation – but in this case it is not really needed since Twitter always returns application/json.
- Ramone decodes application/json using one of a set of predefined codecs. In this case it returns a C# dynamic which reflects the JSON response.
- At last the returned tweets are printed.

Sessions, Services and Configuration

The context for any request in Ramone is a session – a placeholder for a base URL, cookies, authorization, default encoding and media types and much more. The defaults for these values are stored in the static class `RamoneConfiguration` and from this it is possible to get a new session by calling `NewSession(baseUrl)`.

In some case it may be necessary to work with multiple web APIs at the same time – for instance when mashing up two sets of data. In this case it may not be desirable with a global settings class since each of the web APIs might easily have different requirements. For this reason Ramone introduces the concept of a "service" which stores pretty much the same as the configuration class but inside an instance class instead of a static class.

Different HTTP verbs

Most the standard verbs exists as methods on a Ramone request – you can GET, POST, DELETE, PUT and so on directly from the request object.

Here is an example where some kind of new "item" is POSTed to a resource:

```
// Fixed path to resource
string path = "...";

// All interaction with Ramone goes through a session
ISession session
  = RamoneConfiguration.NewSession(new Uri("https://example.com"));

// Create a Ramone request by binding URL to session
Request request = session.Bind(path);

// Create POST payload as anonymous object
var payload = new
{
  Title = "A new item",
  Description = "Description of item."
};

// POST data
Response response = request.AsJson().Post(payload).Dispose();
```

Any other HTTP verb can be issued with `Execute(...)` as here:

```
Response response = request.Execute("OtherMethod", payload).Dispose();
```

Fluent API – chaining method calls

Ramone has a "fluent" style API for setting up requests and working with responses. This means you can stack method calls like this:

```
Response response = request.AsMultipartFormData()
                           .AcceptJson()
                           .Post(payload);
```

Each of the three method calls modifies the containing object and returns either the same instance of the request – or a new instance of a response like Post(...) does.

Serialization and media types

Data sent over HTTP must always be assigned a media type. This is much like a file name extension on Windows that indicates the kind of data contained in the file – or, in case of HTTP, the format of the data on the wire. Often seen examples of media types are text/html, application/pdf, application/xml and application/json but of course there exists a lot more than just these.

Different media types has different purposes – there are media types for images, text documents, spreadsheets and so on – and media types for working with web APIs such as application/json, application/xml and others that are well suited for machine-to-machine integration over HTTP.

Ramone encodes and decodes data using media type codecs – classes that transforms between internal .NET objects and the various media type formats. You can easily add codecs for new media types as well as replace existing ones with codecs that fits your needs better.

Payload serialization

Ramone has a few different ways to assign media types to internal data structures. You can either associate a specific data structure with a codec that handles a certain media type, or you can specify the media type when setting up the request.

Lets take an example where we want to POST a set of Customer data using JSON and accept a JSON result:

```
// Anonymous type representing a customer
var customer = new { Name = "George", Address = "Far Away 12" };

// POST data as JSON, accept JSON as result
Response response = request.AsJson()
                           .AcceptJson()
                           .Post(customer);
```

The methods AsJson() and AcceptJson() are shorthands for the most commonly used formats like JSON, XML and more.

Here is a list of the available AsXxxx and AcceptXxxx methods:

- `AsJson()`
- `AsXml()`
- `AsFormUrlEncoded()`
- `AsMultipartFormData()`
- `AcceptJson()`
- `AcceptXml()`
- `AcceptFormUrlEncoded()`
- `AcceptMultipartFormData()`

Media type codecs

Ramone comes with a set of predefined codecs for the most used web API media types. These codecs are registered when creating a new service (or session) from the `RamoneConfiguration` class. If you do not want them then set `RamoneConfiguration.UseStandardCodecs = false`.

If you need codecs for other media types then you create your own and register them in the codec manager associated with your service.

Predefined codecs

Application/json

The built-in codec for JSON uses JsonFx (https://github.com/jsonfx/jsonfx) to transform between .NET data and JSON.

JsonFx supports reading and writing of plain CLR classes, dynamic types and anonymous types.

Application/xml

The built-in codec for XML uses .NETs own `XmlSerializer` class and supports all the standard C# attributes for customizing the XML format.

Application/json-patch

The built-in codec for JSON patch documents can both read and write patch documents in a type safe way. See page 11 for further information on this topic.

Custom codec registration

Another way to specify media types is to register codecs for well known data structures using the codec manager associated with the current service:

```
// Arguments for creating a new customer
class NewCustomer
{
  public string Name { get; set; }
  public string Address { get; set; }
}

// Customer representation
class Customer
{
  public long Id { get; set; }
  public string Name { get; set; }
  public string Address { get; set; }
}

// New customer argument is sent as application/x-www-form-urlencoded
session.Service.CodecManager.AddFormUrlEncoded<NewCustomer>();

// Customer is represented using JSON
session.Service.CodecManager.AddJson<Customer>();

var newCustomer = new NewCustomer { Name = "George", Address = "Far Away
12" };

// POST NewCustomer args as form-data, accept JSON as returned Customer
using (var response = request.Post<Customer>(newCustomer))
  Customer customer = response.Body;
```

There is a large set of methods to register codecs in various ways.

Custom codecs

Creating new codecs for media types that Ramone does not support is as easy as implementing `ImediaTypeWriter` or `ImediaTypeReader`.

If the new media type is text based (like XML or JSON) then it may save some time to inherit from the class Ramone.MediaTypes.TextCodecBase<T> which will setup stream readers and writers for you with the right character set encoding.

I suggest you take a look at the existing codecs at GitHub to see how a codec is implemented. The basic idea

Hypermedia linking

A basic building stone of REST services is the concept of links that can be followed by the client – often referred to as "the hypermedia constraint" or HATEOAS (Hypermedia As The Engine Of Application State).

The actual format of the links depend on the chosen media type – HTML uses anchors as in `<a href="..." rel="...">...</a>` whereas Atom uses `<atom:link href="..." rel="..."/>`.

Ramone makes it simple to work with such links through the use of a common `ILink` interface. This interface includes a HRef property for the URL and a few others for link relation name, media type and more. With the addition of a few helper methods it becomes very easy to follow hypermedia links in your code. Take for example:

```
ILink link = GetMyLink();

using (var response = link.Follow(Session).Get<MyData>())
{
  MyData x = response.Body;
}
```

What happens here is:

1. Somehow fetch a link representation from somewhere – more on that later.
2. *Follow* the link which means "create a request based on the URL of the link".
3. Do a GET on that request and decode the response to an instance of `MyData`.

An instance of `ILink` can be obtained in many different ways depending on the media type it is embedded in.

HTTP Link header

The HTTP protocol supports inclusion of links in the HTTP headers (see RFC 5988 – http://tools.ietf.org/html/rfc5988). It can for instance look like this (which is an example from the RFC):

```
Link: <http://example.com/TheBook/chapter2>; rel="previous";
      title="previous chapter"
```

Ramone can decode such a header like this:

```
// Arrange
Request request = Session.Bind(LinkHeaderUrl);

// Act
using (Response response = request.Get())
{
  // Get list of links from the response "Link" header
  List<WebLink> links = response.Links().ToList();
}
```

HTML

Ramone supports two different kinds of links in HTML – header links and anchors:

```
<html>
  <head>
    <link rel="..." href="..."/>
  </head>
  <body>
    <p>Click this <a rel="..." href="...">anchor</a>.
  </body>
</html>
```

In order to fetch those HTML elements we must first decode the response data as an HTML document and then extract the links:

```
using (var response = request.Get<HtmlDocument>())
{
  // Select first HTML anchor node with rel="author".
  HtmlNode body = response.Body.DocumentNode;
  HtmlNode anchor = body.SelectNodes(@"//a[@rel=""author""]").First();

  // Convert anchor node to ILink using HTML specific extension method.
  // The response parameter is needed to resolve relative links.
  ILink authorLink = anchor.Anchor(response);

  // Follow author link and get HTML document representing the author
  using (var author = authorLink.Follow(Session).Get<HtmlDocument>())
  {
    ...
  }
}
```

Atom

Patch documents

Ramone supports JSON patch documents as defined in http://tools.ietf.org/html/draft-ietf-appsawg-json-patch-08. This is yet a draft standard so Ramone's implementation may need some later tweaking to work.

A patch document represents a set of changes to a JSON resource. This can be “add” new value, “remove” value or other similar operations.

In Ramone we use the class `JsonPatchDocument` to build a patch document. For instance like this:

```
JsonPatchDocument patch = new JsonPatchDocument<MyResourceType>();

// Supply path as a string
patch.Replace("/Title", "My new title");

// Supply path as a typed referenc
patch.Replace(d => d.Title, "Another title");
```

The use of lambdas like `d => d.Title` is a simple and type safe way to supply a patch path with respect to a given resource class. Using lambdas like this ensure paths are updated when refactoring variable names in Visual Studio.

Patch documents can be sent to a server like this:

```
JsonPatchDocument patch = ... build patch document ...

Request request = Session.Bind(ResourceUrl);

using (var response = request.Patch(patch))
{
  ... do stuff with response ...
}
```

Ramone also supports reading and applying patch documents (using the “visitor” pattern). You do although have to implement the actual operations yourself—Ramone cannot do that for you. Here is one example:

```
using (TextReader r = ... get reader from some JSON input ...)
{
  // Read patch document from input
  JsonPatchDocument patch = JsonPatchDocument.Read(r);

  // Create instance of patching logic
  MyPatchVisitor visitor = new MyPatchVisitor();

  // Apply patch document
  patch.Apply(visitor);
}


// Implements "visitor" pattern where each patch operation in
// the patch document will be translated to a call to Add(...),
// Replace(...) and so on.
class MyPatchVisitor : JsonPatchDocumentVisitor
{
  public override void Add(string path, object value)
  {
    ... implement "add" logic for target data ...
  }
}
```

There is also an optional typed version of the `JsonPatchDocumentVisitor` which will simplify your operations slightly:

```
class MyTypedPatchVisitor : JsonPatchDocumentVisitor<MyResourceType>
{
  public override void Add(string path, object value)
  {
    // Test path against delegate and cast value to int if
    // they match. Then execute the action delegate.
    IfMatch<int>(r => r.Id, path, value,
      v => ... do stuff with integer "v"...);
  }
}
```

Hypermedia forms

HTML forms

HTTP Authentication

Basic authentication

OAuth-2 authentication

Ramone supports some of the many scenarios specified in the OAuth2 framework. It can do three of the basic authorization flows; Authorization code grant, Resource owner password grant and client credentials grant. For all of them you must include a `using Ramone.Oauth2` statement to gain access to the OAuth2 features.

The Authorization code grant flow makes most sense to implement for a web service – not a client side application as Ramone is optimized for. It is possible to handle that flow using Ramone but it doesn't come natural and you are probably better off using DotNetOpenAuth or similar.

All of the methods that fetches an access token from the authorization server will return an instance of `OAuth2AccessTokenResponse`. This class contains the actual token to use, but in most cases you can ignore it since the OAuth2 methods stores the access token internally and passes it on in all future request within the same session.

If you get the access token from some other source then you can assign that to the session with a call to `OAuth2_ActivateAuthorization()`.

Take a look at the GoogleDemo project to see how OAuth2 can be used to acquire access to Google's services.

Authorization code grant

This flow starts with an authorization request (HTTP GET) which in turn will redirect the client to either a page showing an authorization code to use or a redirect URL of the client's choice with the authorization code appended to it.

Here is how to get the authorization code directly for copy/paste from the browser:

```
// Configure OAuth2 with the stuff it needs for it's magic
OAuth2Settings settings = new OAuth2Settings
{
  AuthorizationEndpoint = new Uri("my-authorization-endpoint-url"),
  TokenEndpoint = new Uri("my-token-endpoint-url"),
  ClientID = "my-client-id",
  ClientSecret = "my-client-secret",
  // This special redirect URI means "show me a pin code"
  RedirectUri = new Uri("urn:ietf:wg:oauth:2.0:oob")
};

Session.OAuth2_Configure(settings);

// Create authorization URL
Uri authorizationUrl = Session.OAuth2_GetAuthorizationRequestUrl("my-scope");

// Open URL in browser
Process.Start(authorizationUrl.AbsoluteUri);

// Enter Google authorization code from browser authorization
string authorizationCode = Console.ReadLine();

// Get access credentials from Google
OAuth2AccessTokenResponse token =
  Session.OAuth2_GetAccessTokenFromAuthorizationCode(authorizationCode);
```

If you choose to redirect to a local web service (instead of using the redirect URI "urn:ietf:wg:oauth:2.0:oob") then you can grab the authorization code like this:

```
string authorizationCode =
  Session.OAuth2_GetAuthorizationCodeFromRedirectUrl(redirectUrl);
```

Resource owner password grant

This flow is a one-liner in Ramone assuming you have already configured OAuth2 for Ramone:

```
OAuth2AccessTokenResponse token =
  Session.Oauth2_GetAccessTokenUsingOwnerUsernamePassword(
    userName, password);
```

Client credentials grant

This flow is also a one-liner in Ramone assuming you have already configured OAuth2 for Ramone:

```
OAuth2AccessTokenResponse token =
  Session.OAuth2_GetAccessTokenUsingClientCredentials()
```

Client credentials grant with JWT and SHA256 and RSA-SHA256

It is also possible to authorize using client credentials only – but without sending the actual credentials over the wire and instead use a JSON Web Token (JWT) signed with SHA256 as well as RSA-SHA256. This is for instance how the Google APIs does it.

The methods to use are `OAuth2_GetAccessTokenFromJWT_SHA256()` and `OAuth2_GetAccessTokenFromJWT_RSASHA256()` – please see the GoogleDemo example to see how it is done (the most difficult part is to get the RSACryptoServiceProvider from a certificate).

OAuth-1 authentication

Ramone supports both OAuth1 and OAuth2 authorization, but OAuth1 became deprecated before writing the documentation, so please take a look at the TwitterDemo to see how OAuth1 works with Ramone.

Asynchronous requests

Asynchronous requests can be prepared via a call to `Async()` on the `Request` object. After this you can do the actual request using any of the GET, POST, etc. methods where a callback delegate is passed as on of the arguments. When the request completes it will call back to the delegate, passing in the response from the request.

Here is one example:

```
// Define resource type
public class Cat
{
  public string Name { get; set; }
  public DateTime DateOfBirth { get; set; }
}


// URL template (relative to service root)
const string CatUrlTemplate = "/cat/{name}";


public static void GetAsync()
{
  // Create session pointing to service root
  ISession Session = RamoneConfiguration.NewSession(...));

  // Setup HTTP request
  Request req = Session.Bind(CatUrlTemplate, new { name = "Mike" });

  Console.WriteLine("Waiting for request to finish ...");

  // Initiate asynchronous request
  req.AcceptJson().Async()
     .Get<Cat>(response =>
       {
         Console.WriteLine("Cat: {0}.", response.Body.Name);
         Console.Write("Press Enter to complete: ");
       });

  Console.ReadLine();
}
```

It is also possible to register a callback delegate for error handling; if anything goes bad with the request (or if the operation callback delegate raises an exception) then this error delegate will be called.

In the same way it is possible to register a callback delegate to be called after the request has been completed. This delegate will always be called no matter what the outcome of the operation is.

Building on the previous example, we can now include the two `OnError` and `OnComplete` handlers:

```
// Initiate asynchronous request with additional handlers
req.AcceptJson().Async()
   .OnError(e => Console.WriteLine("Failed: {0}", e.Exception.Message))
   .OnComplete(() => Console.Write("Press Enter to complete: "))
   .Get<Cat>(response =>
     {
       Console.WriteLine("Cat: {0}.", response.Body.Name);
     });
```

POST, PUT and other operations usually includes a body in the request. This can easily be added as a parameter in the asynchronous operation methods:

```
// Initiate asynchronous POST including a request body
var body = ... any data object to POST ...
req.AcceptJson().Async()
   .Post<Cat>(body, response =>
     {
       Console.WriteLine("Cat: {0}.", response.Body.Name);
     });
```

Ramone will also handle redirects, authentication and all the other standard synchronous stuff when going asynchronous.

Configuration

- Serialization formats
- User agent