



# ASP.NET Web API - Documentation

## Table of Contents

1. [Introduction](#)
2. [Preparing with a simple model](#)
3. [Requesting data with jQuery](#)
4. [Debugging with Fiddler](#)
5. [Validation with Data Annotations](#)
6. [Attribute routing \(Web API 2\)](#)
7. [More](#)

## 1. Introduction

---

The last two [Webinars](#) concentrated on client-side JavaScript. Let's review another important part of a modern website: the communication between client and server. It should be lightweight (no SOAP) and standard conform (RESTful). On client-side we are already well equipped with jQuery and Knockout. On the server side we have an ASP.NET MVC 4 website that serves more or less static HTML content. Of course, we could use ASP.NET MVC to send and receive JSON data, but Microsoft offers a dedicated API that offers additional features such as a cleaner code, content-negotiation or improved routing. The bad news for WCF-enthusiasts: all WCF-bits were removed. The good news for all ASP.NET guys: many ASP.NET MVC knowledge can be still required.

## 2. Preparing a simple model

---

We will use again a very simple model. It is a single C# class that represents a sticky post. It can have categories, which are just plain strings for simplicity.

```
public class Note
{
    public Note()
    {
        Categories = new List<string>();
    }

    public int Id { get; set; }

    public string Title { get; set; }

    public string Message { get; set; }

    public DateTime Added { get; set; }

    public IEnumerable<string> Categories { get; set; }
}
```

We will use this model to show a editable list of sticky notes:



Let's assume that we have created a new project from the installed Visual Studio 2012 templates. (ASP.NET MVC 4 Web Application > Web API) This template has all the required Nuget packages included, ships with some tweaks in the web.config and contains the required registration of a default Web API route. These steps can be added to any existing ASP.NET project, too. The only hard dependency is the targeted version of ASP.NET 4.5.

In contrast to other technologies (e.g. the WCF - Windows Communication Foundation) the ASP.NET Web API is built with respect to the **"Convention Over Configuration"** principle.

Without any further configuration-ceremony we can just add a class to the project to offer one or more notes as a resource. (a resource is a source of specific information in a RESTful architecture) By convention the class must inherit from `System.Web.Http.ApiController` and must end with the word "Controller". This naming convention is well-known in the ASP.NET MVC world, too. The file can be located everywhere in the project (it's loaded via reflection), but it is a good start to add it in the namespace & folder "Controllers".

```
public class NoteController : ApiController
{
}
```

This resource will be available under to URL **"/api/note"** as specified in the default route:

```
config.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

The conventions continue with the naming of the methods. To respond to a HTTP GET request, the method should start with the prefix "Get". Since the id was marked as optional, we can add two methods: one for a request without an ID in the URL and one for the optional id (eg. **"api/note/222"**).

```
public class NoteController : ApiController
{
    public IEnumerable<Note> GetAll()
    {
        return NoteRepository.ReadAll();
    }

    public Note Get(int id)
```

```

{
    return NoteRepository.Read(id);
}
}

```

The exact method name does not matter, just the prefix and the parameter signature is important. Additional to "GET" [RFC 2616](#) defines some more HTTP methods (also known as "verbs"). For the daily work the following HTTP verbs should be known.

HTTP method	Possible usage
GET	retrieve information
POST	create a new resource
PUT	update a resource
DELETE	delete a resource

### 3. Requesting data with jQuery

The ASP.NET Web API was clearly made for serving AJAX driven websites. A very easy-to use [AJAX api is provided by jQuery](#). Our simple Web API can be consumed with the following lines of code:

```

$.getJSON('/api/note').done(function (xhr) {
    console.log("AJAX Result: ", xhr)
});

```

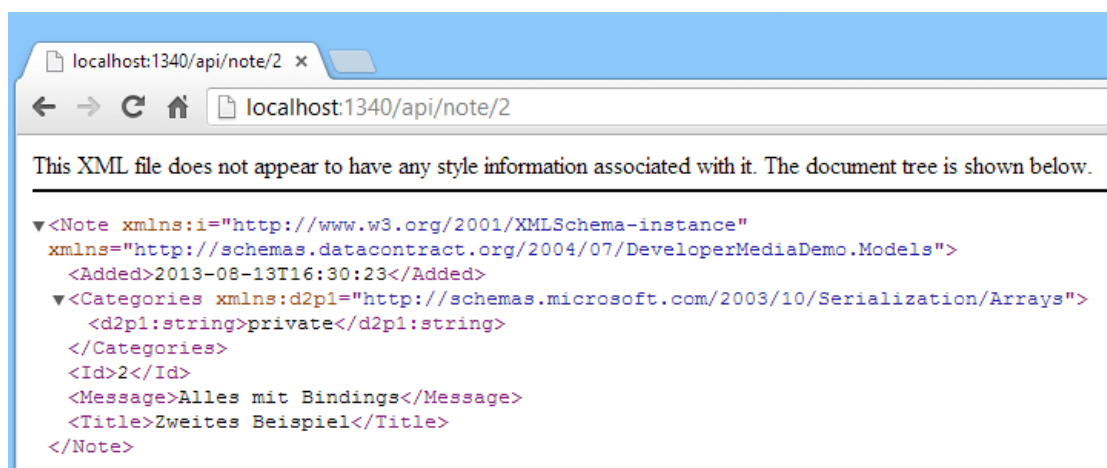
The Web API will respond with [JSON-formatted data](#) so that jQuery can automatically convert the data to a JavaScript object. To get the note with the number 2, we just have to change the URL:

```

$.getJSON('/api/note/2').done(function (xhr) {
    console.log("AJAX Result: ", xhr)
});

```

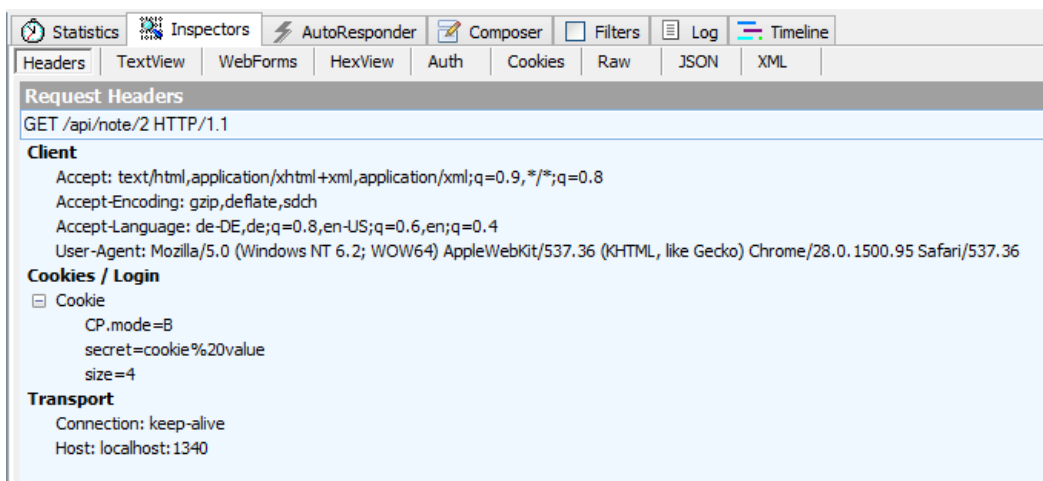
We are working with normal HTTP requests - just like any other website. It's time to review the response directly in the browser. Let's open the Web API URL directly in the browser! (eg. <http://localhost/api/note/2>) The result might be surprising. Instead of JSON-formatted data we are receiving some unexpected XML! We should debug that issue.



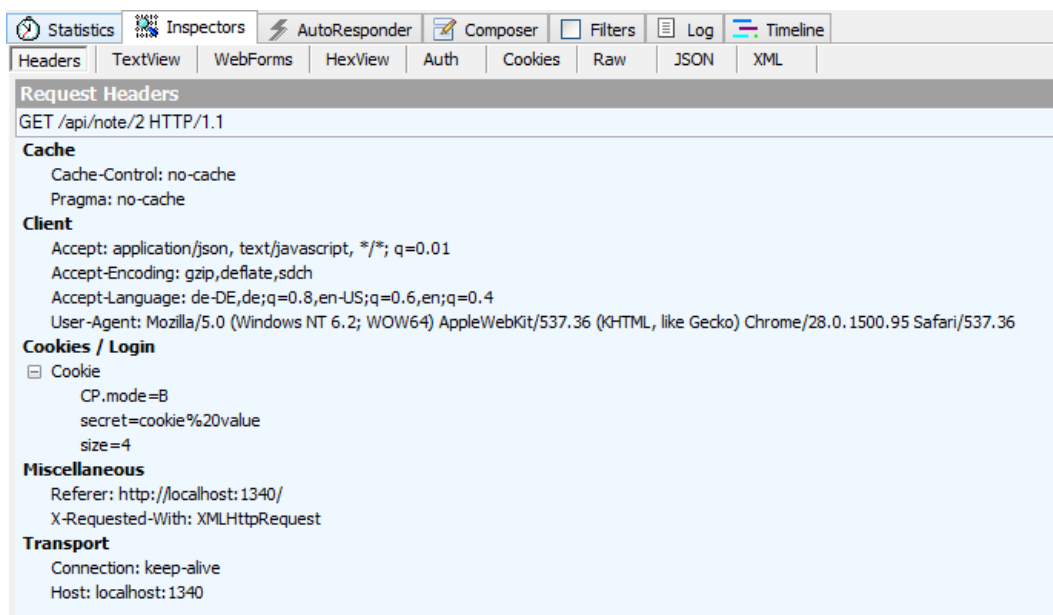
### 4. Debugging with Fiddler

[Fiddler](#) is a web debugging proxy which logs all HTTP traffic between your computer and the internet (or your local server). Today, all modern browsers offer a network debugger, too (press F12 key). But Fiddler is still a very convenient tool when it comes to manipulate the requests or responses.

This is the request we created, while surfing the URL directly with the browser:

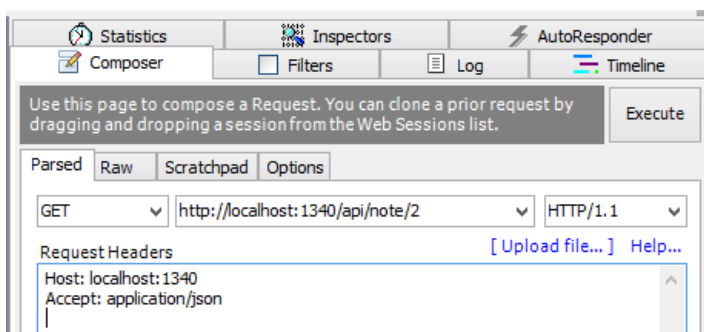


In comparison to that, this is the request made by the jQuery AJAX call:

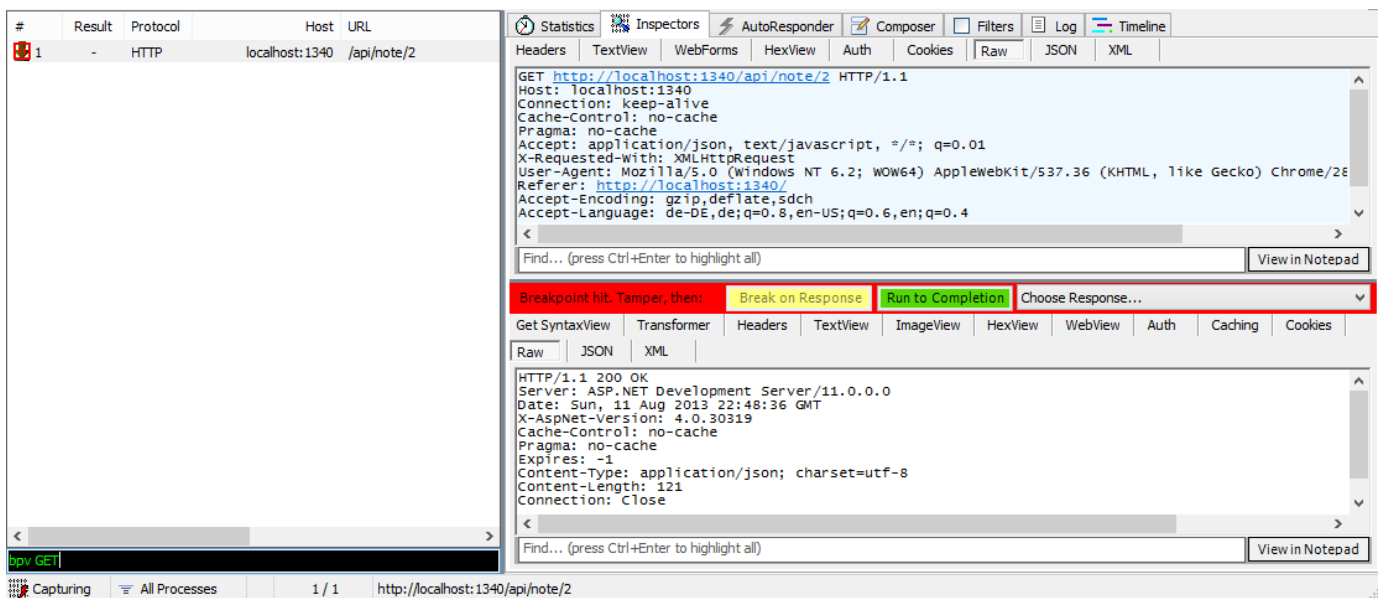


You will notice, that we have send two different Accept-Headers. When surfing around the web, we probably want the see a HTML-formatted document. That's why we are sending a "text/html" header. But according to the browser, it would be also ok to get a document in any XML-format. This "application/xml" **media type** is recognized and build-in to the ASP.NET Web API. As requested, the Web API responds with a XML-formatted document. This mechanism is called **content negotiation**.

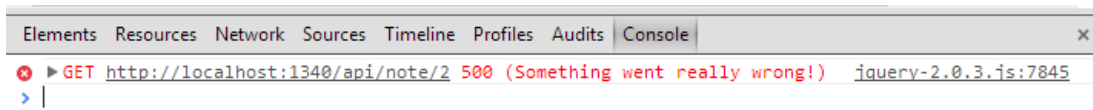
We can approve that behavior by manipulation the first request with Fiddler. Switch to the "Composer" tab and drag&drop the first request to that tab. Change the accept header to "Accept: application/xml" as shown in the following screenshot and start a new request by clicking on execute.



It also possible to manipulate any response to quickly confirm the correct behavior of the client. For that Fiddler offers a set of "QuickExec" commands. Just type `bpv GET` to stop on every GET request.



On the next GET request you will see the screen shown above. Click on the request in the list, go to the inspector and let the call go through by clicking the yellow **"Break on Response"** button. We are now free to change virtually anything of that response. For example, change the status code from "HTTP/1.1 200 OK" to "HTTP/1.1 500 Something went really wrong!" and hit **"Run to Completion"**. As expected, the browser will receive an error:



By just entering `bpv` the breakpoint is removed again.

## 5. Validation with Data Annotations

Data Annotations (attributes from the namespace [System.ComponentModel.DataAnnotations](#)) were originally created for the validation of data sent to an ASP.NET MVC controller. Now the same technology serves the ASP.NET Web API, too. By decorating the data transfer object, simple rules can be applied. More complex rules can be created by inheriting from [ValidationAttribute](#).

```
public class Note
{
    [Required]
    [MinLength(5)]
    public string Title { get; set; }

    [Required]
    [MinLength(10, ErrorMessage = "Don't be lazy!")]
    public string Message { get; set; }
}
```

However, you should know that MVC relies on the "DefaultModelBinder" while the Web API has a more complex pipeline.

Good to know: By default validation will be invoked by the [ModelBinderParameterBinding](#) (works on the URL) or [FormatterParameterBinding](#) (works on the content body [FromBody]). See the big [ASP.NET Web API Poster](#) for more information.

Web API controllers can check if the rule was satisfied by evaluating **ModelState.IsValid**.

```
public class NoteController : ApiController
{
    public IEnumerable<Note> GetAll()
    {
        return NoteRepository.ReadAll();
    }

    public Note Get(int id)
    {
        return NoteRepository.Read(id);
    }
}
```

```

public HttpResponseMessage Post()
{
    var newNote = new Note();
    NoteRepository.Create(newNote);
    return Request.CreateResponse(HttpStatusCode.Created, newNote.Id);
}

public HttpResponseMessage Put(Note note)
{
    if (ModelState.IsValid)
    {
        NoteRepository.Update(note);
        return Request.CreateResponse(HttpStatusCode.OK);
    }
    return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ModelState);
}

public void Delete(int id)
{
    NoteRepository.Delete(id);
}
}

```

## 6. Attribute routing (Web API 2)

In v1 of the Web API all routes were defined in a central place. This has advantages for global routes, but leads to dirty code as soon as more and more special routes are required. Things get messy as soon as controllers change but old route definitions stay orphaned. But now the ASP.NET Web API v2 supports [attribute routing](#). With attribute routing you can specify your Web API routes by annotating your actions and controller directly.

You can either install the [Visual Studio 2013 Preview](#) or use the existing Visual Studio 2012 installation and download the newest version from Nuget: (currently 5.0.0-beta2)

```
Install-Package Microsoft.AspNet.WebApi -Pre
```

In the `Global.asx.cs` (or wherever `GlobalConfiguration.Configuration` is altered) the only change is this new line of code:

```
config.MapHttpAttributeRoutes();
```

To search for a note with a special title we can now annotate a new method with `[HttpGet]` and the desired route:

```

public class NoteController : ApiController
{
    public Note Get(int id)
    {
        return NoteRepository.Read(id);
    }

    [HttpGet("note/search/{titlePart}")]
    public IEnumerable<Note> GetSearch(string titlePart)
    {
        return NoteRepository.ReadAll().Where(x => x.Title.Contains(titlePart));
    }
}

```

In the previous version of the Web API we the same solution would require a dedicated config as well as a explicitly named action:

```

config.Routes.MapHttpRoute(
    name: "SearchNoteApi",
    routeTemplate: "api/{controller}/{action}/{titlePart}",
    defaults: null,
    constraints: new { controller = "Note|OrMaybeAnotherController" }
);

public class NoteController : ApiController
{
    [ActionName("search")]
    public IEnumerable<Note> GetSearch(string titlePart)
    {

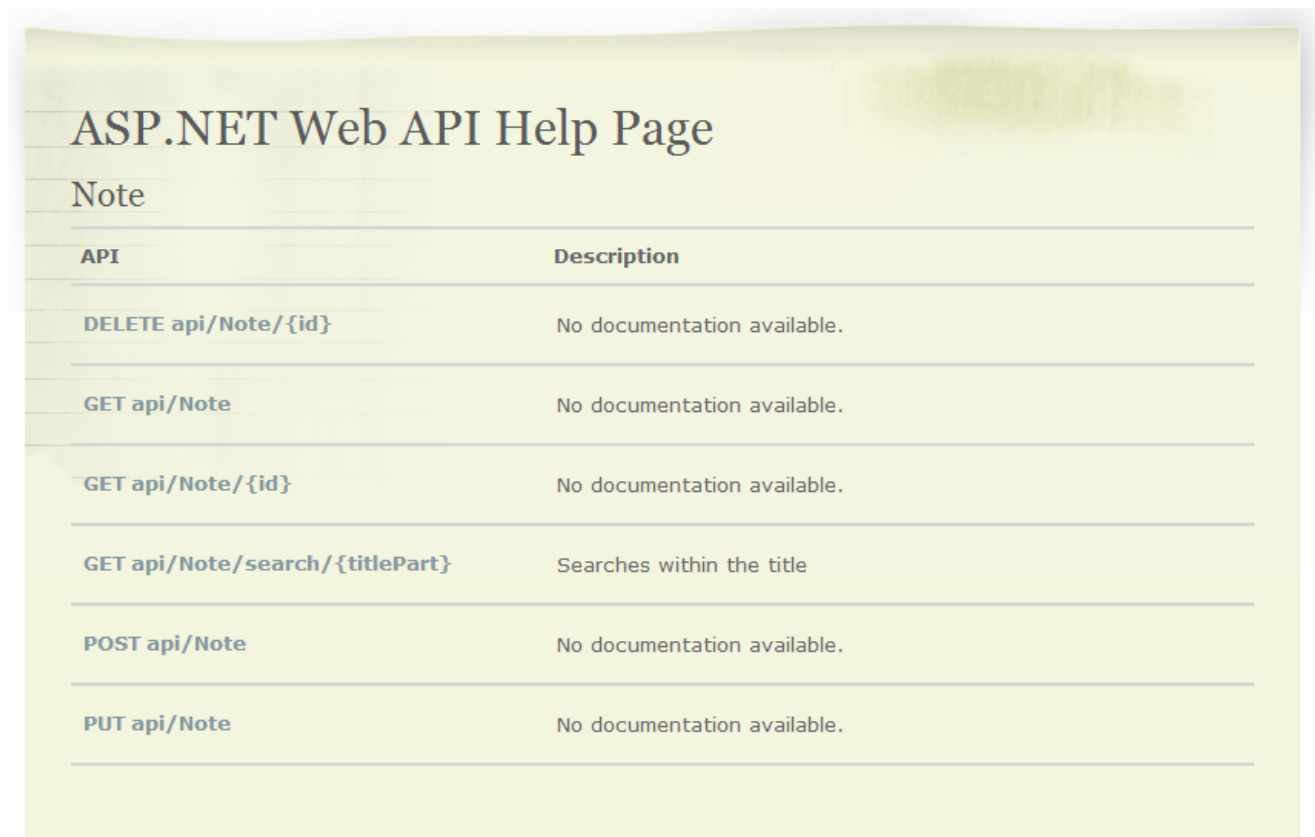
```

```

    return NoteRepository.ReadAll().Where(x => x.Title.Contains(titlePart));
}
}

```

To review all new routes we can install the "Microsoft ASP.NET Web API Help Page" via Nuget  
`Install-Package Microsoft.AspNet.WebApi.HelpPage -Pre` and access the new page at `/help`.



API	Description
DELETE api/Note/{id}	No documentation available.
GET api/Note	No documentation available.
GET api/Note/{id}	No documentation available.
GET api/Note/search/{titlePart}	Searches within the title
POST api/Note	No documentation available.
PUT api/Note	No documentation available.

Previously it was a bit tricky to switch between actions which are only separated by their type. Now you can add "Route Constraints" to separate them. Regular expressions are available, too.

```

public class NoteController : ApiController
{
    /// <summary>
    /// Searches for a the answer to Life, universe and everything
    /// </summary>
    [HttpGet("api/Note/search/{what:regex(^answer$)}", RouteOrder = 1)]
    public int GetSearchFor42(string what)
    {
        return 42;
    }

    /// <summary>
    /// Searches within the title (only alpha-characters "[A-Za-z]*$")
    /// </summary>
    [HttpGet("api/Note/search/{titlePart:alpha}", RouteOrder = 2)]
    public IEnumerable<Note> GetSearch(string titlePart)
    {
        return NoteRepository.ReadAll().Where(x => x.Title.Contains(titlePart));
    }

    /// <summary>
    /// Searches for a year
    /// </summary>
    [HttpGet("api/Note/search/{year:int}")]
    public IEnumerable<Note> GetSearch(int year)
    {
        return NoteRepository.ReadAll().Where(x => x.Added.Year == year);
    }
}

```

## 7. More

---

There are several more things to discover.

You should start by downloading the sources of the demo app.

[» Download Demo-Code \(.zip\)](#)

---

© 2013, *Johannes Hoppe*