ASP.NET Web API 2 - Documentation

Table of Contents

1. Advanced ASP.NET Web API

In the last webinar we were introduced to the ASP.NET Web API from Microsoft. Lets continue with some advanced topics that will help us to improve the code quality.

1.1 Asynchronous Requests

By design, all Web API actions are multithreaded, so that concurrent requests are possible. Similar top ASP.NET MVC, the Web API uses a thread pool to serve requests. Threads from a pool can be reused, which saves a lot of hardware resources (memory and CPU).

This design leads us to a new problem. As soon as data comes from an external source (e.g. from the database or from an other service) chances are high that the code is **not** written in a scalable fashion. While waiting for the external resource, the thread can't be reused and blocks a free slot. If there are numerous requests for operations that take a long time to complete, more and more threads will be created, consuming additional memory and negatively affecting the overall performance.

Thanks to support for asynchronous programming, it is very easy to write asynchronous methods to overcome threading bottlenecks. C# 5.0 introduces the keywords `async` and `await` as well as the return Type `Task<T>`. Thanks to the fresh design from the ASP.NET Web API, no additional tasks are required.

This will block our code as long as we are waiting for the result:

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

The same code rewritten using async/await:

```
public async Task<IEnumerable<Note>> GetAll()
{
    return await NoteRepository.ReadAllAsync();
}
```

The compiler ensures, that there is a trail of await keywords until we reach an awaitable framework function. Thanks to the updated .NET framework, libraries like ADO.NET, Entity Framework or the new HttpClient already have inbuilt support. Here is an example for the new

HttpClient:

```
public async Task<IEnumerable<Note>> GetAllWebinar()
{
    HttpClient client = new HttpClient();
    HttpResponseMessage response = await client.GetAsync("http://johanneshoppe.github.io/DeveloperMediaSlides/webinar.json");
    response.EnsureSuccessStatusCode();

    return await response.Content.ReadAsAsync<IEnumerable<Note>>();
}
```

You can read more about consuming REST services via the HttpClient on the official ASP.NET website.

1.2 Action Filter

Last time some people insisted on the model validation example. They were totally right, validation is a typical cross cutting concern.

> You should see my German video which addresses aspect oriented programming (AOP) and cross cutting concerns in detail!

The following action filter must be written once but can be applied to all actions:

```
/// <summary>
/// If model validation fails, this filter returns an HTTP error response that contains the validation errors.
/// In that case, the controller action is NOT invoked.
/// </summary>
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(HttpActionContext actionContext)
    {
        if (!actionContext.ModelState.IsValid)
        {
            actionContext.Response = actionContext.Request.CreateErrorResponse(
                HttpStatusCode.BadRequest,
                actionContext.ModelState);
        }
    }
}
```

You can ether apply this filter to all API controllers:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Filters.Add(new ValidateModelAttribute());
    }
}
```

or decorate individual controllers or controller actions:

```
[ValidateModel]
public HttpResponseMessage Put(Note note)
{
}
```

1.3 Inversion of control

Inversion of control is a very common principle in modern software development. Martin Folwer published a good introduction to IoC. In short: IoC is usually done via dependency injection. It decouples software, it promotes contracts (via interfaces) and it reduces side effects. I recommend Autofac for both ASP.NET MVC as well as ASP.NET Web API. It already contains an implementation of System.Web.Mvc.IDependencyResolver (MVC) as well as an implementation of System.Web.Http.Dependencies.IDependencyResolver (Web API) so that we do not have to concentrate on the internal details.

```
Install-Package Autofac.Mvc4
Install-Package Autofac.WebApi
```

Add the following file to your project to set up Autofac:

```
public static class ContainerConfig
{
    public static void RegisterDependencyResolver()
    {
        var builder = new ContainerBuilder();

        // add more types here!
        builder.RegisterType<NoteRepository>().As<INoteRepository>()
               .InstancePerHttpRequest()
               .InstancePerApiRequest();

        builder.RegisterControllers(Assembly.GetExecutingAssembly());
        builder.RegisterApiControllers(Assembly.GetExecutingAssembly());

        IContainer container = builder.Build();

        SetMvcResolver(container);
        SetWebApiResolver(container);
    }

    /// <summary>
    /// Set the dependency resolver for Web API.
    /// </summary>
    private static void SetMvcResolver(ILifetimeScope container)
    {
        var mvcResolver = new AutofacDependencyResolver(container);
        DependencyResolver.SetResolver(mvcResolver);
    }

    /// <summary>
    /// Set the dependency resolver for MVC.
    /// </summary>
    private static void SetWebApiResolver(ILifetimeScope container)
    {
        var webApiResolver = new AutofacWebApiDependencyResolver(container);
        GlobalConfiguration.Configuration.DependencyResolver = webApiResolver;
    }
}
```

Now controllers are able to get dependencies via constructor injection:

```
public class NoteController : ApiController
{
    private readonly INoteRepository _repository;

    public NoteController(INoteRepository repository)
    {
        _repository = repository;
    }
}
```

1.4. Unit Tests (NEW: Web API 2)

Simple cases with method parameters and own return types (e.g. our Note) were never tricky to unit test. In Web API v1 there were two ways of creating a response from an API action. Either returning a specific **object instance** (the Web API pipeline converts it automatically to a default HttpResponseMessage) or returning a raw **HttpResponseMessage**. This snippet shows is a typical example:

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

[Subject(typeof(NoteController))]
public class When_getting_notes
{
    static NoteController controller;
    static List<Note> result;

    Establish context = () =>
    {
        INoteRepository repository = Substitute.For<INoteRepository>();
        repository.ReadAll().Returns(TestData.Notes);
        controller = new NoteController(repository);
    };

    Because of = () => result = controller.GetAll().ToList();

    It should_return_three_notes = () => result.Count.ShouldEqual(3);
}
```

ASP.NET Web API 2 introduces **IHttpActionResult**. It is effectively a factory for HttpResponseMessage and by implementing the interface you provide instructions on how a new response should be constructed. It is **the new way** of creating responses. The internal code changes behind the introduction of IHttpActionResult offer more convenience. More parts of the framework can be manipulated for Unit Tests while nothing breaks by default (great!). We are free to mock every part of an API controller. Here is an other example that uses the new interface and changes some bits:

```
public class NoteController : ApiController
{
    public IHttpActionResult GetSearch(string titlePart)
    {
        var result = _repository.ReadAll().Where(x => x.Title.Contains(titlePart)).ToList();

        if (!result.Any())
        {
            return NotFound(); // NEW
        }

        return Content(HttpStatusCode.OK, result); // NEW
    }
}

[Subject(typeof(NoteController))]
public class When_searching_for_an_unknown_note
{
    static NoteController controller;
    static HttpResponseMessage result;

    Establish context = () =>
    {
        INoteRepository repository = Substitute.For<INoteRepository>();
        controller = new NoteController(repository)
                        {
                            Request = new HttpRequestMessage(HttpMethod.Get, "http://test/api/note/search/test"),
                            Configuration = new HttpConfiguration()
                        };

        repository.ReadAll().Returns(TestData.Notes);
    };

    private Because of = () =>
                            {
                                Task<HttpResponseMessage> task = controller.GetSearch("test").ExecuteAsync(new CancellationTo
                                task.Wait();
                                result = task.Result;
                            };

    It should_respond_with_not_found_status_code = () => result.StatusCode.ShouldEqual(HttpStatusCode.NotFound);
}
```

2. REST

2.1. Introduction to REST

As we have seen, the ASP.NET Web API is a great toolbox that helps us to create HTTP-based application interfaces. We were able to offer

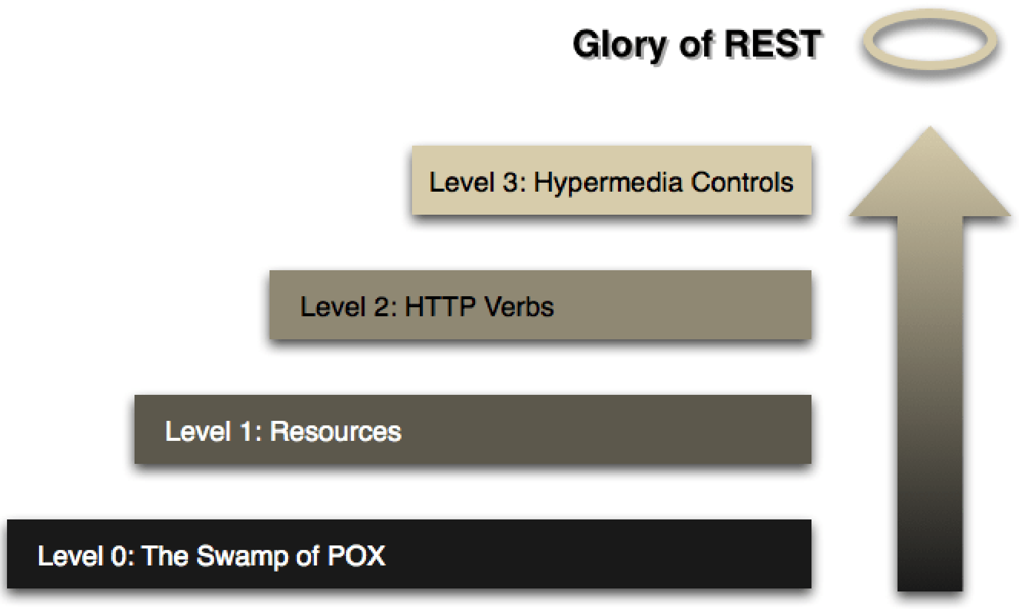
CRUD operations an a simple resource (which represented some sticky notes), but until not we did not investigated time in the general ideas behind REST at all. It's time to change this!

Roy Fielding is one of the authors of RFC 2616 (HTTP/1.1, from 1999). In his PhD thesis (from 2000) he generalized the Web's architectural principles and presented them as an architectural style, called REST (Representational State Transfer). His research about distributed application architectures and especially the chapter about REST explains the success of the web due to client-server architecture, statelessness, cacheability or layered systems. Well known building-blocks like resources and resource identifiers (URI) or representations should be used. However, due to its academic standard, it is very hard to build an valuable API just with the help of the thesis paper. REST was long forgotten, but frameworks like Ruby on Rails made it public to a wider audience.

In theory REST principles could be applied many protocols. In practice, REST is directly related to the web as we know it with its main protocol HTTP. To make things easier, Leonard Richardson proposed a classification for services on the web. Leonard's model suggest three levels of service maturity based on the support for URIs, HTTP, and hypermedia.

2.2 Richardson Maturity Model and Microsoft technologies

The Richardson Maturity Model breaks down the principal elements of a REST approach into three steps.



2.3 Level Zero Services

Services of that level have a single URI and usually all requests are POST'ed to it. The old ASP.NET web services (*.asmx) or **Windows Communication Foundation (WCF)** web services fall into that category. Client and server are usually tightly coupled to each other. Both parties must match exactly to each other. The most common protocol is SOAP which uses HTTP as a tunnel. The tight coupling mustn't be a disadvantage, since a WSDL file can describe such an API very well. However these remote procedure calls (RPC) have nothing to do with REST and are (in the authors opinion) hard to implement between different platforms.

2.2 Level One Services: Resources

The first REST level offers many URIs but only a single HTTP verb (HTTP verb == HTTP method). Due to the lack of verbs APIs of that level tend to have numerous URIs to call. It is very easy to build such an API with plain **ASP.NET MVC** actions that return JSON. Resources are usually build with a questions mark in it, where GET-parameters are used, e.g. `GET /Home/GetAllNotes?limit=200&search=test` or `GET /Home/DeleteNote?id=22`. Level one services are very hard to explore and require a very well written documentation.

2.3 Level Two Services: Verbs

Level two services host numerous resources (identified via URIs) which offer several HTTP verbs. By definition HTTP GET is a safe operation while verbs like PUT, POST or delete are unsafe operations meant to change the resources state. Level two services should also use HTTP status codes to coordinate interactions. Without any further work, the typical **ASP.NET Web API** controller with its CRUD operations (Create, Read, Update, Delete) lives at level two.

2.4 Problems at Level Two

As long as four methods (CRUD) are really enough, the default ASP.NET Web API plays very well. But for our stick note example: how should we architect a simple "deactivate" operation?

2.4.1 Simple Solution - URI

We could switch back to level one and build special URIs for special operations. Developers of a client will have to know in advance, in which situation those URI should be called. A well written documentation is required, but the approach works and is widely used.

2.4.2 Dirty Solution - POST

We could enhance the POSTed data with a new property. Instead of sending:

```
{
  "Title": "Note 1",
  "Description": "A long text"
}
```

we could hack this into our data:

```
{
  "Title": "Note 1",
  "Description": "A long text",
  "IsActive": false
}
```

Hell will freeze if we are designing an API in that fashion. This approach opens the door we countless pitfalls, eg. data loss or data inconsistency, wrong operations (eg. are we allowed to deactivate the resource by setting the value?) and very tight coupling. The deactivate operation is mixed within the data, so the client developers have to know everything about deactivation and the server developers have to reverse-engineer the lost intent from the given data! You should read more about this Data/Actions Impedance Mismatch in Sergey Shishkins blogpost.

2.4.3 Better Solution - PATCH

RFC 5789 (PATCH Method for HTTP) introduces a new verb that reduces the problems. A PATCH request changes just a part of the data of a resource. A PATCH in JSON format (draft!) could look like this:

```
{
  {"replace": "/IsActive", "value": false}
}
```

You can learn more about PATCH in this blog post from Mark Nottingham. However, PATCH was not intended to mask operations. We are still facing the data/actions impedance mismatch.

2.5 Level Three Services: Hypermedia

As we have seen, we need a new communication direction. Until now, the client was forced to know everything about the data and the operations that are available. A better approach would an API where the server tells the client which data and operations he wants to offer the the client. This approach leads to he most web-aware level of service supports: the notion of **hypermedia as the engine of application state** (HATEOAS). In that level resources contain URI links to other resources that might be of interest for the next action. HTML offers all required controls (such as links <a> or lists <ul>) but it was designed to present data and operations to humans, not machines. At the moment there is no (final) standard which defines an definite set of hypermedia controls, that are designed to be consumed by machines.

REST-developers must offer a starting point for a machine that wants to follow a trail of resources. To navigate through the trails we could use one of these formats. (or build our own domain-specific format)

- Collection+JSON - designed by Mike Amundsen
- **Hypertext Application Language (HAL)** - designed by Mike Kelly
- JSON-Home - designed by Mark Nottingham.

For a valuable API the RFC 6570 (URI Template) should be considered, since it describes how URI can be generated via placeholders. (e.g. `GET /api/Note?limit={limit}&search={searchPhrase}`) You should also avoid to reinvent the wheel by using already existing microformats to describe your data.

It seems that **HAL** is going to be the next widely adopted standard, since it is submitted as a internet-draft and already implemented by several libraries. The ASP.NET Web API can be easily taught to speak HAL, by implementing a dedicated MediaTypeFormatter. HAL can be expressed as JSON or XML, but until now all examples were written in JSON format, so let's continue with that.

JSON-HAL (application/hal+json) is just plain old JSON, with two reserved properties: `_links` and `_embedded`. A valid JSON-HAL representation could be this document:

```
{
  "Title": "Note 1",
  "Description": "A long text",
  "_links": {
    "self":   { "href": "/api/Note/1" },
    "next":   { "href": "/api/Note/2" },
    "search": { "href": "/api/search/Note/{searchPhrase}",
                "templated": true }
  }
}
```

The HAL draft is designed to work on GET and does not cover an easy way to describe NON-Get operations. JSON-Home goes a step further and introduces another set of vocabulary:

```
"/api/Note/1": {
   "hints": {
     "allow": ["GET", "POST", "DELETE"],
     "formats": {
       "application/json": {}
     },
     "accept-post": ["application/xml"]
   }
 }
```

I would be happy to see an API where both formats would be thrown together. Here is our domain-specific hypermedia format that would server our needs. Why should call it "application/hal+home+json-webnote"!

```
{
  "Title": "Note 1",
  "Description": "A long text",
  "_links": {
    "self":   { "href": "/api/Note/1" },
    "next":   { "href": "/api/Note/2" },
    "search": { "href": "/api/search/Note/{searchPhrase}",
                "templated": true },
    "deactivate": { "href": "/api/Note/2",
                    "allow": ["POST"],
                    "accept-post": ["application/json-post-webnote"]
                  }
  }
}
```

At the moment all formats are just proposals or drafts. As long as you document your API very well, I would consider the code-snipped above as a real RESTful API.

You can lean more about HAL by using the HAL browser, that helps to discover HAL APIs (including an inbuilt demo).

Several other formats are addressing more specialized tasks. In example the ASP.NET Web API supports the **Open Data Protocol (OData)** to discover, query and manipulate big sets of data in a standardized way. OData was invented by Microsoft and is built on the AtomPub protocol and JSON. On the Web API site, the result must have a type of **IQueryable<T>** to let the magic begin (that's very cool).

```
[Queryable]
public override IQueryable<Note> Get()
{
  return Notes.GetAll();
}
```

3. More

There are several more things to discover.
You should start by downloading the sources of the demo app.

Will be published after the talk!

A great book is "REST in Practice". It will teach you the spirit of REST while being easy to understand and read!

Note: during my speak I will probably mix up "media type" and "content type" again. "Content-Type" is the name of the HTTP header field that carries the media type in the field value. So I will use both terms interchangeable.