Web Developer Conference 2013

# SPA Workshop - Part 2: REST

**Table of Contents**



## 1. Introduction

We decided to send and receive JSON data, lets review dedicated API that offers additional features such as a cleaner code, content-negotiation or improved routing. Both the **ASP.NET Web API** as well as the **Zend Framework** offer nearly similar features.

## 2. A RESTFul framework

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

Without any big 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)

This resource will be available under to URL **"/api/note"**.

In ASP.NET Web API as well as Zend the conventions start 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"**).

### C#

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

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

### PHP

```
class NoteController extends AbstractRestfulController {

    public function getList() {
        return new JsonModel($this->repository->readAll());
    }

    public function get($id) {
        return new JsonModel($this->repository->read($id));
    }
}
```

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

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)
});
```

# 4. REST maturity and Hypermedia

## 4.1 History

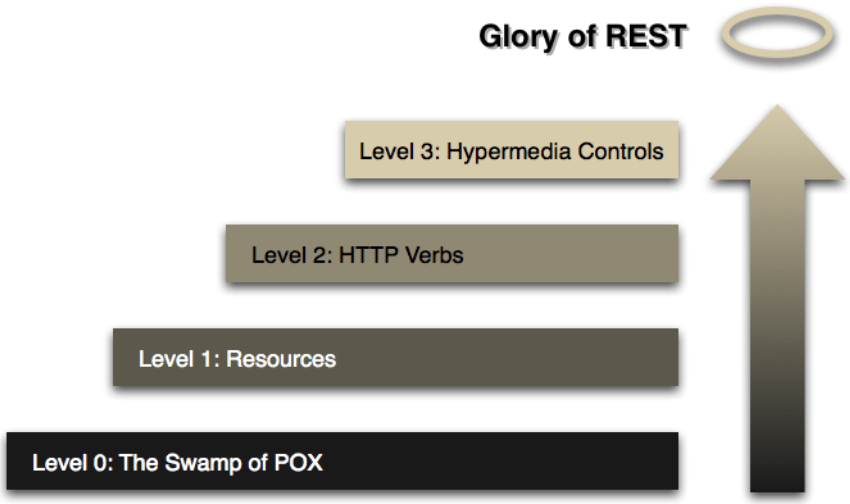
As we have seen, .NET as well as PHP offer a simple toolbox that helps us to create HTTP-based application interfaces. We were able to offer CRUD operations an a simple resource (which represented some sticky notes), but until not we did not investigated time in the general ideas behind REST at all. 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.

## 4.2 Richardson Maturity Model

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




## 4.3 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.

## 4.4 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.

## 4.5 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.

## 4.6 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?

### 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.

### 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.

### 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.

## 4.7 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).

## 4. More

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