DataFlex to new heights

Creating RESTful JSON Web Services with DataFlex

What, Why and How

Mike Peat
Unicorn InterGlobal Limited

So what is REST?

REST - coined by Dr Roy Fielding (PhD dissertation: "*Architectural Styles and the Design of Network-based Software Architectures*" in 2000 - Chapter 5)

Based on his work on the Web protocols: HTTP, HTML and URI/URL

The principles for such an architectural style:

- Client-Server
- Stateless
- Cacheable
- Uniform Interface
- Layered System
- Code-On-Demand

Essentially, saying something is "RESTful" means that it is designed *like the Web*

RESTful web services

Web services emerged in the 90s for disparate systems communication:

- CORBA and ORBs (remember those? OK - so I'm old!)
- XML-RPC
- SOAP

Complicated and difficult to use

To simplify, and thus encourage adoption, eBay then based their API on web services designed according to Fielding's principals, aiming to become "*The operating system for e-commerce on the Web*"… Amazon quickly followed suit and soon others like Google were doing the same kind of thing

Today REST-style web services are everywhere, with JSON replacing XML as the data exchange (*representation*) format of choice

Why?

Fielding identified six benefits accruing from the use of a RESTful design:

- Scalability
- Generality
- Independence
- Low-Latency
- Security
- Encapsulation

The adoption of REST was driven by companies like eBay, Amazon, Flickr and Google which were aiming at "Internet-Scale" adoption of their APIs which those benefits facilitated

So what are RESTful web services?

- Not required, but in practice always over HTTP
- Identify coarse-grained resources through URIs (URLs)
- Transfer representations of those resources (as JSON)
- Manipulate those resources using HTTP verbs
- Use response status (200 OK, 404 Not Found, etc.)
- Use content-type (application/json, etc.)
- Use query string params (for filtering, ordering, etc.)
- Use hypertext/hypermedia links (href)

Linking

Fielding's thesis said: "*REST is defined by … hypermedia as the engine of application state*"

Which has become an acronym: **HATEOAS**

Of which Wikipedia says: "*A REST client enters a REST application through a simple fixed URL. All future actions the client may take are discovered within resource representations returned from the server.*"

… and Fielding again: "*If the engine of application state (and hence the API) is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API*" (so there! … at least if you are a *RESTafarian*!)

Verbs

The common set: GET, POST, PUT, PATCH, DELETE

GET retrieves a representation and must be **SAFE** (no side effects)

PUT is problematic: it is supposed to be **IDEMPOTENT** so it can only be used for creating resources if all data is known (no server-assigned IDs) or for update if complete replacement is done - no partial updates

I recommend **POST** for create and **PATCH** for update

Whatever you use, try to be consistent - makes life easier for third-party developers trying to work to your API

Resources

Since HTTP is providing the verbs, resources should be **nouns**

There are two types of resources:

- Collections - e.g. customers (collection names should be plural)
- Instances - e.g. customers/22 (collectionName/identifier)

So why would you need an API at all?

In a nutshell: System Longevity and (Job) Security

If your system has an API, which other businesses (or business units) utilise, it cannot be replaced by another system lacking a functionally *identical* API without breaking business continuity

Takeovers, mergers or new management notwithstanding, while internal staff can have a new system imposed on them from above, it is much harder to persuade business partners to rewrite parts of their systems to match the current fad

So why would you need an API at all?

Every time inter-operating with another business entity comes up:

If you have an API and they don't, it's a no-brainer: they work to your API

If both parties have APIs then there will be a negotiation about which is best to use: this is where the quality and ease of use of your API matters

But if they have an API and you don't then it is only going to go one way: your system will depend on theirs, not the other way around

All of which means that you should be developing one or more APIs for your systems **now**, before the specific business requirement emerges

Does it have to be RESTful?

Well, no… it doesn't...

You *could* go on using the familiar SOAP web services for your API

If the other party needs to work with JSON you might even manage that using the JSON/debug feature of the DataFlex SOAP web service (perhaps disguised behind some clever URL Rewrite manipulation to look less ugly)

However that still leaves you tied to the fixed structs that the DataFlex SOAP services deal in - using JSON objects can be much more flexible, which often matters

Also RESTful services are what people and companies expect to deal with these days - if you are going to do it, it makes sense to go with that flow

Creating a RESTful API in DataFlex

Need to use an ASP file (only 12 lines):

```
<%
   Dim sData, iLen, sResp
   iLen = Request.TotalBytes
   If (iLen > 0) Then sData = oRestService.UTF8ToString(Request.BinaryRead(iLen))
   sResp = oRestService.call("get_ProcessCall", sData, iLen)
   iPos = InStr(sResp, Chr(31))
   If (iPos > 0) Then
      Response.Status = Mid(sResp, 1, (iPos - 1))
      sResp         = Mid(sResp, (iPos + 1))
   End If
   If (Len(sResp) > 0) Then
      Response.BinaryWrite oRestService.StringToUTF8(sResp)
   End If
%>
```

Creating a RESTful API in DataFlex

Changes in IIS Manager:

Need URL Rewrite module

Add server variable "ORIGINAL_REQUEST"

Add rewrite rule to redirect requests to our ASP file, putting the remainder of the call-path into our server variable

Creating a RESTful API in DataFlex

Class cRESTfulService (version 2) - create a WebApp, then a Web Service object within it and change that object's class to be cRESTfulService (also change Use)

Basic infrastructure for accepting and returning JSON

RouteCall function must be overridden to call business functions which have a return type of **Variant** and Function_Return a call to **ReturnJson** (or **ReturnError)**: Function_Return (ReturnJson(Self, ???))

BeforeProcessing function - override to take action prior to RouteCall (e.g. to perform authentication)

AfterProcessing procedure - hook to allow you to modify the JSON response prior to it being returned

Support in cRESTfulService

Some helpful properties and methods of the class:

- **psVerb** - the HTTP verb used to make the call
- **RequestPart** - returns specific parts of the call (0, 1, 2, etc.)
- **phoJsonData** - handle of the object containing passed JSON (if any)
- **QueryValue** - returns the value of a query string parameter
- **HttpHeaderValue** - returns the value of a passed HTTP header
- **DDUpdateFromJson** - updates DD with column values in passed JSON
- **JsonFromDD** - returns a JSON object with (selected) DD values
- **OriginalURL** - returns the full URL which was called
- **MatchesFilterString** - complicated, but supports constraining on filters

There are quite a few more - whenever I find myself writing similar code, I make that a parameterised method… some of those were generic enough that I made them methods of the class

Other aspects of the class

pbDFErrsToJson - set True: if there are errors during a call it will return those in the JSON - useful both for debugging and in helping users of your API understand why things went wrong (Set **pbVerboseErrors** False for production)

psResponseStatus (and **pbReturnRespStatus** - True by default) - allows setting the HTTP response status - defaults to "200 OK" (HttpResponseStatus.pkg: C_http*NNN*)

pbRESTDir - needs to be set to your pseudo-directory (which may be "") for OriginalURL (and the BaseURL it calls) to work properly - default "/REST"

pbReturnBinary - if True will allow return of Binary database fields as base64

API design considerations

What to return to the client?

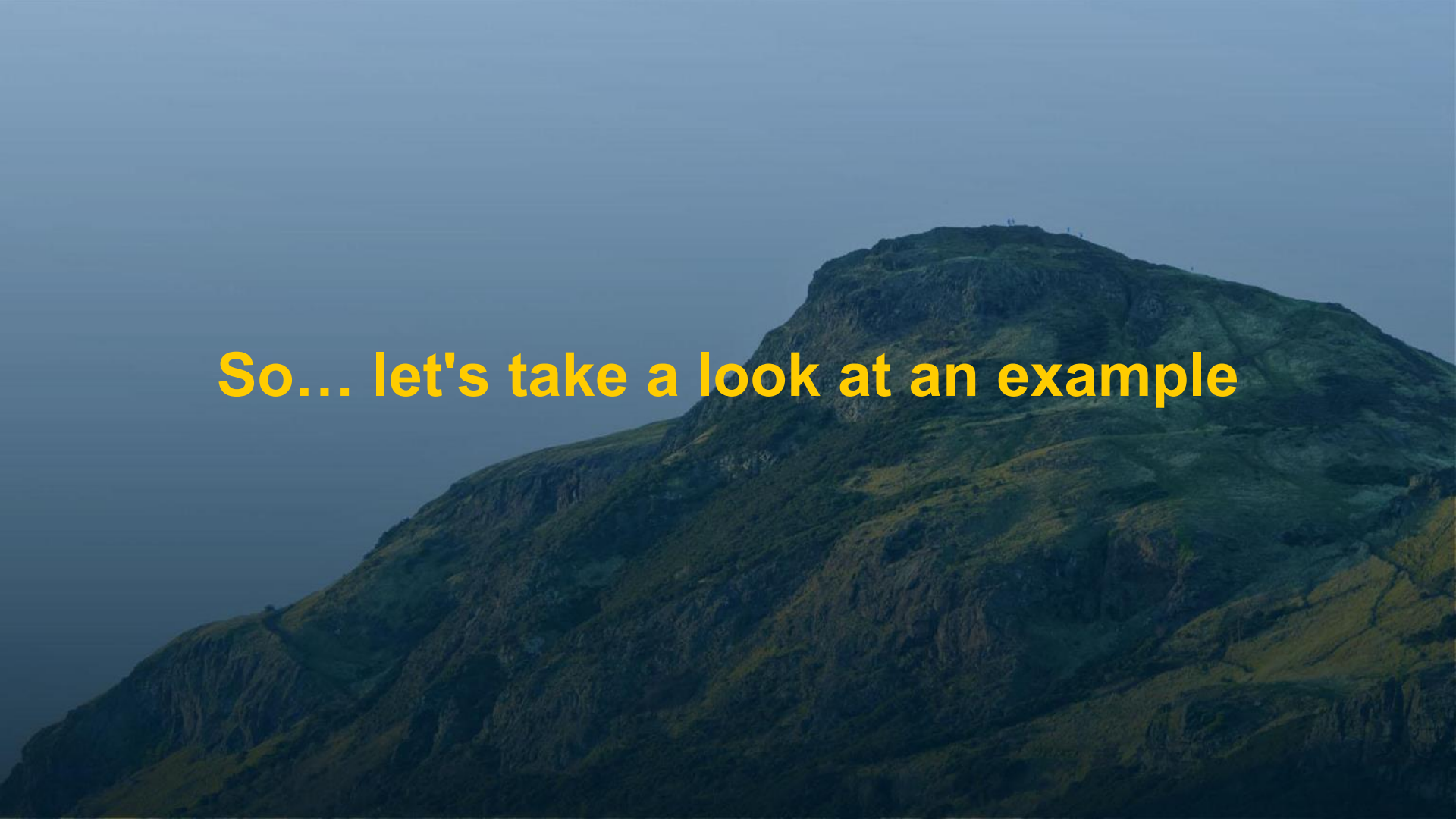
GET is obvious: they asked for something - give it to them (with 200 - OK)

However **POST** (for create), **PATCH** (for update) and **DELETE** are less obvious:

- Nothing? (except a response status of 201 Created or 200 OK)
- A message saying they were successful?
- A representation of the resource the call was acting upon?

I like the last option best - especially helpful in the case of an inadvertent DELETE!

In the event of an **error** the class offers **ReturnError** with a error number, a message and a fuller description, but you may choose a different approach


So… let's take a look at an example

In the real world

That was obviously not a *real* API, however, but it has the basic features of one

When designing, try to avoid "RPC" style thinking: not easy

Many "RESTful" APIs are actually little more than JSON-RPC using GET and POST to call RPC-style functions

A properly "**RESTful**" API must be built around resources (nouns) identified by URIs, the operations (verbs) to be performed on them and links (hrefs) between them

Otherwise Dr Roy will cross you off his Christmas card list and one night a mob of angry torch-and-pitchfork welding **RESTafarians** will come and burn your house down!

Further information

I wrote a white paper last year on some of the mechanics of doing this, published on our web site www.UnicornInterGlobal.com: go to Company → White Papers to find it at:

http://www.unicornInterglobal.com/Company-White-Papers-Creating-RESTful-JSON-Web-Services-in-DataFlex-868

However... I have learned quite a bit more about the niceties of being properly *RESTful* since then - that article only covers the basics and I recommend doing things a little differently now

I should probably write a new one!


European DataFlex users' conference · Edinburgh Scotland
EDUC
2018
DataFlex to new heights
Thank you!
Are there any questions?