How to build a RESTful API in DataFlex 19.1

Mike Peat - Unicorn InterGlobal

So what exactly is REST?

- **REST** stands for **RE**presentatonal **S**tate **T**ransfer - coined by Dr Roy Fielding in his PhD dissertation in 2000
- To unpack that a little: an *aspect* of the state (of either the server *or* the client) is transferred from one to the other using some representation (i.e. HTML or XML or JSON)
- Fielding based his thesis on the work he had previously done on the internet standards: HTTP, URI/URL and HTML
- In essence, to say something is RESTful means that it has *an architecture like the web*

What does that mean for us?

Essentially always over HTTP (or more likely HTTPS)

Identifies course-grained resources through URLs

Operates on those resources through HTTP verbs

Transfers representations of those resources (as JSON in our case)

Returns HTTP response statuses (200 OK, 404 Not Found, etc.)

Identifies content-type (application/json, in our case.)

Uses query-string paramerers (...?param1=x&param2=y, etc.)

Employs hypertext links (href)

Hypertext Links (href elements)

In his thesis Fielding wrote: "*REST is defined by … hypermedia as the engine of application state*" (now the horrible acronym: *HATEOAS*)

Wikipedia on HATEOAS: "*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*!)

REST is not CRUD?

Many people say this, but at a basic level, there is a pretty close correspondence with the operations we generally need to perform on databases:

- **C**reating rows
- **R**eading one or more rows
- **U**pdating a row
- **D**eleting a row

REST gives us the tools to do that

The HTTP Verbs

Verbs used by RESTful APIs: GET, POST, PUT, PATCH and DELETE

GET is simple: it returns a representation of a resource - and must also be "**safe**": have no side-effects

PUT is problematic, because it is supposed to be **IDEMPOTENT**

We will therefore use **POST** to create and **PATCH** to update

DELETE is obvious - it is also supposed to be Idempotent, but that's OK: you can delete a given resource as often as you like… it's just that after the first you will get "404 Not Found" errors

Resources

Since HTTP is providing the verbs, resources should be nouns

In a REST context, there are really only two types of resources:

- Collections (the naming of which should be plural to indicate that they *are* collections of items)
- Instances - individual items within those collections

For us - at a basic level at least - these can correspond to tables (collections) and rows within them (instances)

Why would I need an API anyway?

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

Why would I need an API anyway?

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 use your API
- If both parties have APIs then there will be a negotiation: 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

Why would I need an API anyway?

Every other system which uses an API into yours is another anchor in the ground

Increasingly your application, its data, and, perhaps most importantly, its business logic becomes the heart of a software ecosystem

Your application is the core component that all those others rely on

Does it have to be RESTful?

Well, no… it doesn't…

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

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

The DataFlex REST Library

The library basically contains three main classes (plus one sub-class)

All of these should turn up on your Class Palette when you use the Library in your project Workspaces, which makes using them easy - just drag-and-drop them in

- cRESTfulService
- cRESTResourceHandler
- cRESTApiObject

The cRESTFulService class

This will be the provider of your RESTful web service

It is based on Data Access's **cWebHttpHandler** class (new in DF 19.1)

It has a primary method (procedure) **ProcessHttpRequest** which will drive everything else through a **Case** statement which will direct requests to the appropriate cRESTResourceHandler object

It acts as a container to hold your cRESTResouceHandler objects

It is analogous to the oClientArea in a Windows app or the oWebApp in a web app (although it will actually be <u>in</u> the latter)

The cRESTResourceHandler class

Objects of this class hold your Data Dictionary structures for the various purposes you may have

They also hold the cRESTApiObjects which do the actual heavy lifting in delivering the aspects of your API

They are analogous to "Views" or "Web Views", containing DDO structures and controls

The cRESTApiObject class

These are the workhorses of your API - analogous to controls

They provides the core functionality to **List** a collection, detail an **Instance**, **Create** an instance, **Update** an instance or **Delete** an instance

They have a **phoDD** property, which must be set to a DDO, and which will determine (through that DDO's Main_File property) **which table it will operate on**

They have a **psCollName** (collection name) property **which will determine which HTTP requests (URL) they will respond to**

The cRESTApiObject class

They have a **psInstName** property (largely cosmetic <g>)

They have a **RequiredConditions** function which can be used to pull other required records into the DDOs when <u>creating</u> records, based on the passed JSON

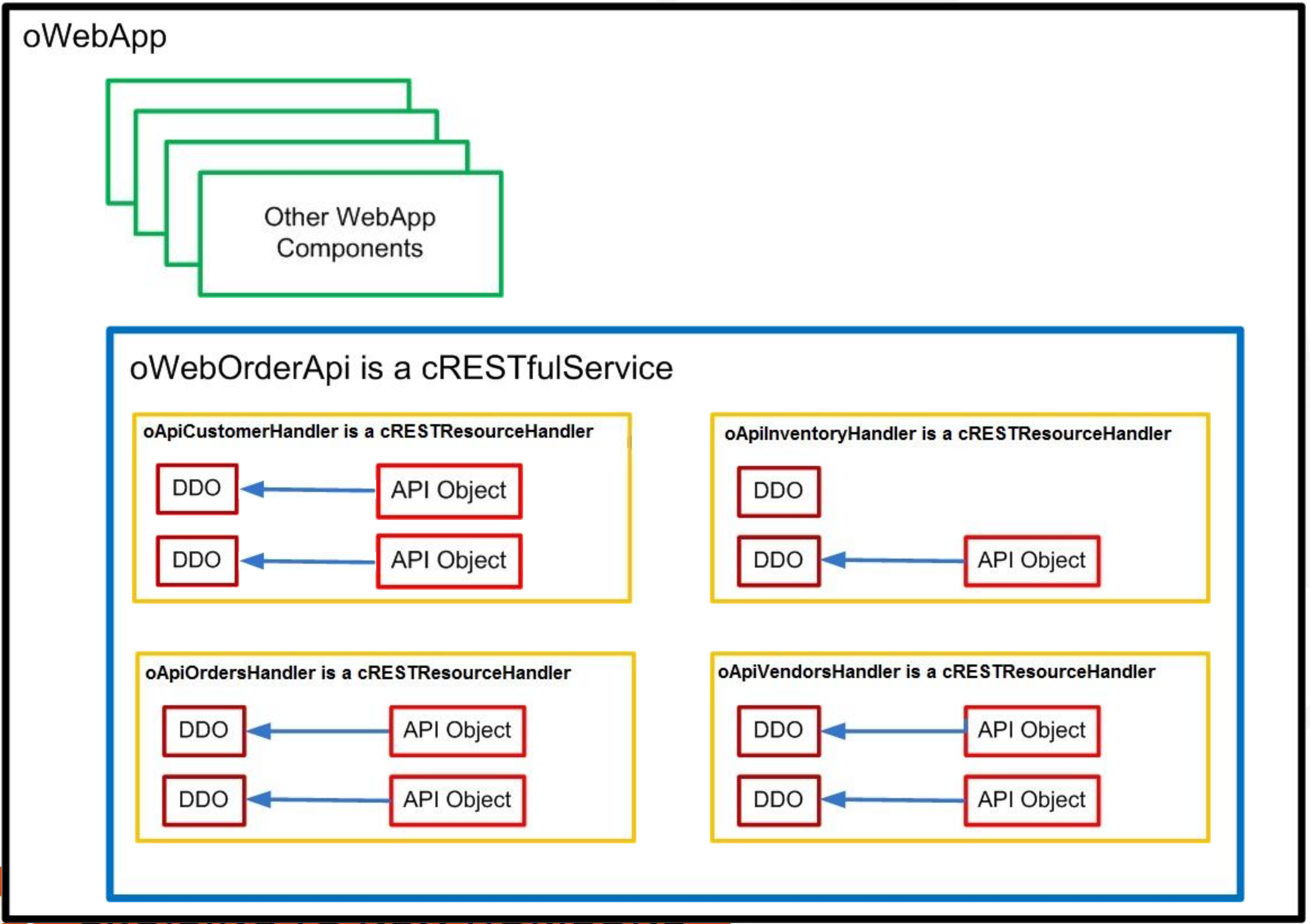
They have has a **PostProc** procedure which can be added to to "decorate" instances with links to dependant collections (think Orders → Details) or other "href" links

The cRESTApiObject class

They have a **PreProc** function which ensures that the correct *constraining* records - based on the parts of the path - are pulled into parent DDOs automatically, but which can be **overridden** in the event that multi-segment relationships are involved

By the same token, they have **InstID** and **FindInstance** functions which will correctly deal with simple cases automatically, but can be overridden in the event that the required key has multiple segments

API Structure within oWebApp


oWebApp
Other WebApp Components
oWebOrderApi is a cRESTfulService
oApiCustomerHandler is a cRESTResourceHandler
DDO
API Object
DDO
API Object
oApiInventoryHandler is a cRESTResourceHandler
DDO
DDO
API Object
oApiOrdersHandler is a cRESTResourceHandler
DDO
API Object
DDO
API Object
oApiVendorsHandler is a cRESTResourceHandler
DDO
API Object
DDO
API Object

The API

Between these classes, we will be able to quickly build access to our database on a kind of "*maintenance view*" level

A little like using DataBase Explorer, but all operations are mediated by our data dictionaries and their business rules

The classes do support *somewhat* more control than that - fields can be excluded or marked as read-only and interfaces can be set to read-only or no-delete or no-update

For higher-level business operations you will still have to write your own code - this should provide a good start however

Demonstration!

Thank you!

Any Questions?