

**Lab Center – Hands-on Lab**

**Session 3548**

**Session Title Learn how to Build Application with the IBM Cognos TM1, OData Compliant, REST API**

Guido Tejeda Davila, IBM, gtejeda@us.ibm.com

Hubert Heijkers, IBM, hubert.heijkers@nl.ibm.com

**Table of Contents**

[Getting ready 3](#_Toc464259908)

[Introducing the OData compliant RESTful API 4](#_Toc464259909)

[A first peek at TM1’s RESTful API 4](#_Toc464259910)

[Explore the REST API 6](#_Toc464259911)

[A real life HTML/JavaScript based TM1 client app: TM1Top Lite 6](#_Toc464259912)

[Building your first model using the REST API 8](#_Toc464259913)

[Setting up a new TM1 server 8](#_Toc464259914)

[Building the model using the REST API 9](#_Toc464259915)

[Getting ready to do some coding 9](#_Toc464259916)

[Getting familiar with what’s there already 10](#_Toc464259917)

[Bringing it all together into the builder app 12](#_Toc464259918)

[Building a web app that uses your model 16](#_Toc464259919)

[Testing your model directly using the REST API 16](#_Toc464259920)

[We Value Your Feedback! 17](#_Toc464259921)

# Getting ready

As always there are a couple of late minute changes that need to be made, updates to the examples, instruction documents that aren’t ready in time etc. etc. Happens every time we do one of these Hands-On Labs as we want you to experience the latest and greatest of our software, and the last coolest ideas we’ve come up with.

As such there are a couple of little steps that need to be executed before your machine is ready.

1 – Grabbing the latest files for the update

The latest versions of the files needed on this box, and the sources you’ll be working with in this lab, are all kept together in one GIT repository on github.com.

Not only that, you’ll be working with Go, a.k.a. Golang, during this lab and Go has support for project, dependency and build management build in. So we’ll grab the latest by executing the following command in a command box:

go get github.com/hubert-heijkers/wow2016/hol3548

After loading the content of the repository you will receive an error message referring to the fact that there are no buildable Go source files in it, but that error can be ignored.

2 – Updating the Virtual Machine

Next we’ll execute a little batch file the updates a bunch of files and does some set up needed for the lab later on. This update can be executed by typing the following command in the command box:

%GOPATH%\src\github.com\hubert-heijkers\wow2016\hol3548\vmupdate\vmupdate.bat

That’s it, enjoy the lab!

# Introducing the OData compliant RESTful API

TM1 Servers, as of version 10.2 RP2, exposes an OData compliant, RESTful API. This was the first PUBLIC version of a RESTful API. In the meantime, a number of fix packs have been released with additional improvements and extensions to the REST API. It is safe to say that, even though still relatively new, it is very mature and the best performing API we have available for TM1. And in case you had any doubt, it will be THE API for the TM1 Server going forward.

Now you might wonder what the being “OData compliant” is all about. Well, ignoring the fact that a bunch of people from various companies, organized in an OASIS technical committee have brought together all the knowledge and experience they had to offer, it is simply said a set of specifications which we obey by that specify how a service describes what is available to a consumer, how a consumer formulates a request for a compliant server and how the service formats the response to the request. So it’s not just saying use the telephone and speak English but, more importantly, having agreed on the ‘topic’, a syntax for any requests, formulation the responses and the dictionary being used in the conversation.

OData, short for Open-Data, has been developed over a number of years as well and the latest version, v4 errata 3, is an OASIS standard. The OData standards has also made it to ISO standard in the meantime as well. For more information about OData standard and the documents describing it please visit the OData.org website at: <http://www.odata.org>. For a quick introduction to the OData standard have a look at the ‘[Understanding OData in 6 steps](http://www.odata.org/getting-started/understand-odata-in-6-steps/)’ webpage.

## A first peek at TM1’s RESTful API

So let’s start with having look at the metadata of the TM1 server first.

1. Start Google Chrome.
2. Retrieve the metadata document by typing the following URL in the address bar: <http://tm1server:8000/api/v1/$metadata>

The metadata for the TM1 server will be shown in your browser. It’s and XML document formatted according to the CSDL specification which is part of the OData standard. It describes all the types, entity and complex types, all entity sets and relationships between entity and complex types in the service. For example, the ‘Dimension’ entity is described as (excluding the documentation annotations):

<EntityType Name="Dimension">  
 <Key>  
 <PropertyRef Name="Name"/>  
 </Key>  
 <Property Name="Name" Type="Edm.String" Nullable="false"/>  
 <Property Name="UniqueName" Type="Edm.String"/>  
 <Property Name="Attributes" Type="ibm.tm1.api.v1.Attributes"/>  
 <NavigationProperty Name="Hierarchies" Type="Collection(ibm.tm1.api.v1.Hierarchy)" Partner="Dimension" ContainsTarget="true"/>  
 <NavigationProperty Name="DefaultHierarchy" Type="ibm.tm1.api.v1.Hierarchy"/>  
 <NavigationProperty Name="LocalizedAttributes" Type="Collection(ibm.tm1.api.v1.LocalizedAttributes)" ContainsTarget="true"/>  
</EntityType>

This is telling us that one of the types that the service exposes is a ‘Dimension’ and that it has a couple of properties among which is its Name, UniqueName and a set of Hierarchies (note: the version of TM1 you are using still only allows one Hierarchy per Dimensions, which has to have the same name then the dimension, but the REST API is ‘future proof’ in that it already supports alternate hierarchies, a feature which is tentatively planned for the upcoming v11 release). The Name is the property that uniquely identifies the Dimensions, and as such acts as the key.

As you scan the metadata file you’ll see all the types available and how they relate to each other and it is this metadata document that consumers of the service will use to understand what is available in the REST API.

Let’s be a consumer for a sec and, knowing what’s available in the service, start retrieving some data from the service. So let’s look at the list of the ‘Dimensions’ that we have in the service and, while at it, ignore those control dimensions (the dimensions starting with the ‘}’ character).

1. Retrieve those dimensions not being control dimensions by typing the following URL: [http://tm1server:8000/api/v1/Dimensions?$filter=not startswith(Name,'}')](http://tm1server:8000/api/v1/Dimensions?$filter=not%20startswith(Name,'%7d'))
2. If this is the first time you are accessing a secured resource, you’ll be challenged for a username and password. If this happens use the famous “admin” and “apple” pair.

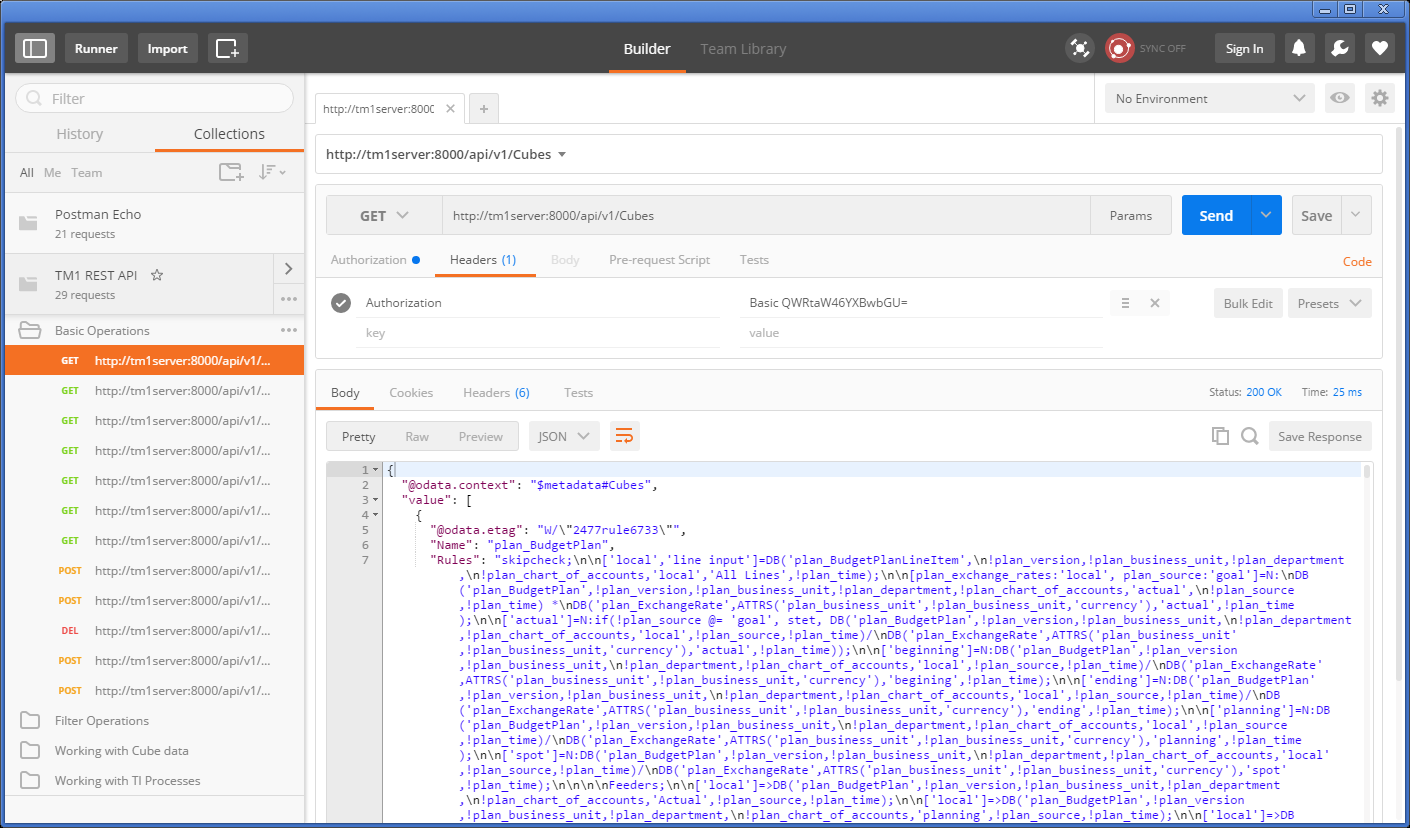
You’ll get the list of dimensions available shown in your browser nicely formatted because we installed the JSONView plug-in for Chrome.



If you want to see what went over ‘the wire’ you can start Fiddler, by clicking the  icon in the taskbar. Once Fiddler is up it’ll start recording HTTP traffic and you can look at the requests going to and the responses returned by the server. This way you’ll see for example that the JSON going over the wire is pretty compact and that we, provided the client supports it, apply compression to the response.

## Explore the REST API

Ok, it’s time for some more examples. To make it easier to interact with our, any for that matter, HTTP/REST based service we use Postman. Click on the  icon in the taskbar to start Postman.



After starting Postman you’ll find, under the Collections tab on the left, a collection named ‘TM1 REST API’. A bunch of examples have been included in this collection to give you an initial feel of what the REST API can do for you and how it works.

Note: If you don’t see the ‘TM1 REST API’ collection, not to worry, hit the ‘Import’ button on the top, open a file explorer, locate the ‘C:\HOL-TM1SDK\postman\_collections’ folder and drop the ‘TM1%20REST%20API.json.postman\_collection’ file in the screen that opened up.

After selecting an example, you can see the definition of the request on the right. Hitting the ‘Send’ button will execute the request after which the response will be shown to you in the output window. Don’t forget to look at the Cookies and Headers tabs in the output pane to see what more is being send forth and back between the client, Postman in this case, and the TM1 Server.

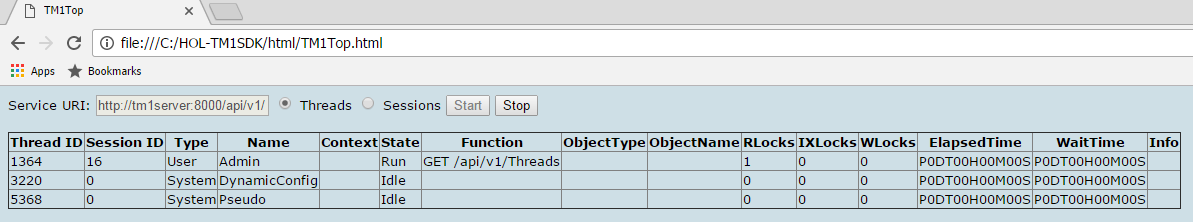
Postman is a very convenient tool to test requests. If you haven’t done so already we’d advice you to download it and install it in your environment and have a go at it. Want the collection of tests from this lab, don’t hesitate to contact any of the presenters and we’ll send it to you. Have fun!

## A real life HTML/JavaScript based TM1 client app: TM1Top Lite

To illustrate how quick and easy it is to build client applications using the new TM1 REST API, have a look at our TM1Top “Lite” sample application. It’s a simple, standalone, web client that periodically retrieves the active threads and inserts them into a table. It’s obviously not pretty but it is functional. And, since we are using a recent enough version, it is capable of showing the threads by session too.

You can find the sample at [file:///C:/HOL-TM1SDK/html/TM1Top.html](file:///C:\HOL-TM1SDK\html\TM1Top.html). Open it using Chrome. If you’re curious to see how it’s implemented, take a look at the source code by right-clicking anywhere in the web page and click “View page source”.

If you wonder what all the fiddling with security modes is about, have a look at the ‘[Using CAM Authentication with TM1’s, OData compliant, REST API](https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/W181f1083f3dd_455f_b2f8_f63c4a9c8010/page/Using%20CAM%20authentication%20with%20TM1's%2C%20OData%20compliant%2C%20RESTful%20API)’ article on [developerWorks TM1 SDK community](https://www.ibm.com/developerworks/community/groups/service/html/communitystart?communityUuid=94a0a656-48ea-436d-ac5c-c711caf02e85).



HTML/JavaScript is only one of the many ways to consume the new TM1 REST API. In the next section, we’ll show you how to build applications that connect to TM1 using C#, C++ and Java. These are simply examples, you can choose to build your applications with your choice of language/environment running on any OS as long as it supports making HTTP requests and you have means to compose and parse JSON.

# Building your first model using the REST API

Consuming data and metadata thru the REST API is one, and likely what most consumers will end up doing but it doesn’t stop there. Obviously one can create, update and delete objects, like dimensions and cubes, as well.

In this chapter you build an application, using Go, that creates all the artifacts that make up you model and, subsequently loads data, sales data in this case, into the Sales cube that you’ll be creating.

The data source for this exercise is the [NorthWind](http://services.odata.org/V4/Northwind/Northwind.svc/) database, hosted on the [OData.org website](http://www.odata.org/). This database is exposed as an OData compliant service as well.

The goal of this exercise is to learn as much about OData as it is about TM1’s REST API itself. By the end of this chapter you’ll hopefully start to see resemblances and patterns in requests being used as a result of either of these services being OData compliant, and have seen how relatively easy it is to integrate TM1 in a larger eco system of services.

In this chapter you will:

* Set up a new TM1 Server on your machine named “NorthWind”
* Written a portion of an application, named ‘builder’, that will create the model
* Ran the application and validated that the model got created successfully

Let’s get started!

## Setting up a new TM1 server

One of the things we can’t do, yet, is create a complete new model (read: server). So we’ll start with doing that the old fashion way, which means:

* Creating a data directory that is going to contain all the data for our model
* Create a tm1s.cfg file in that directory with the configuration for our new model
* Create a shortcut to start the new TM1 server representing our new model
* Start it!

On our lab VM machine we are storing the data for our TM1 models in the C:\HOL-TM1SDK\models folder. We are going to call our new service ‘NorthWind’ as per the data source name, so we’ll start with creating a new directory in the C:\HOL-TM1SDK\models folder called ‘NorthWind’.

To be able to start a new TM1 server we need, at a minimum, a configuration file, tm1s.cfg. Create a new, text, file, in the newly create NorthWind folder. Open it in an editor and make sure it has at least the following configuration settings set:

[TM1S]  
ServerName=NorthWind  
DataBaseDirectory=.  
HTTPPortNumber=8088  
HTTPSessionTimeoutMinutes=180  
PortNumber=12222  
UseSSL=F  
IntegratedSecurityMode=1

The most important things in here, apart from the server name, is the HTTPPortNumber, instructing the server what port to use to host the REST API on, and secondly, the UseSSL setting which we’ve set to false implying that we’ll not be using SSL on our connections which, for our REST API, implies we’ll be using HTTP instead of HTTPS. Note that in normal installation you would not turn SSL off and, preferably, you’d always use your own certificate, as opposed to using the one provided with the install.

Now that we have a data folder and configuration down we’ll, on the desktop, create yet another TM1 server shortcut for our NorthWind server. So right-click on the desktop and select ‘New’ > ‘Shortcut’ from the pop-up menu. In the dialog that shows up type in the following in the location box:

"C:\Program Files\ibm\cognos\tm1\_64\bin64\tm1s.exe" -z "C:\HOL-TM1SDK\models\NorthWind"

The first portion is the location of the tm1s.exe file, which is in the default install location, and the -z option tells the server where to go look for the configuration file, which is in the folder you just created.

Now that you have the shortcut you can double-click and start your new, empty, TM1 server.

Note: If at any point later in this chapter you needed a ‘reset’, for example if you end up building only a part of your model and wanted to start from scratch again, just stop the TM1 Server, remove all the files from the NorthWind folder with the exception of the tm1s.cfg file, and start the server again.

## Building the model using the REST API

Now that we have a server up and running it is time to create some dimensions, create a cube and load some data into that cube. For that you’ll be creating an application, written in Go, that does exactly that. And, to make it easy for you, we’ve already gone ahead, created a project and wrote the code that would help you implement this application, including the skeleton of the application itself.

### Getting ready to do some coding

So before we’ll write some code let’s get familiar with the project and learn how to build and run it.

You’ll be using Go, and as mentioned before, it has built in support for dependency management, building, testing etc. All the files it works with however therefore need to be organized in places where it knows where to find it. The root of all those locations is the so called GOPATH. On the lab VM the GOPATH is set to ‘C:\Users\Student\Go’. The sources and their dependencies, which it manages the organization of for you, all reside under the ‘src’ subfolder and once it’s done building, and installing, an application the binary for that application ends up in the ‘bin’ subfolder.

The source for our project, named ‘builder’, under the github.com\hubert-heijkers\wow2016\lab3548 repository therefore can be found here:

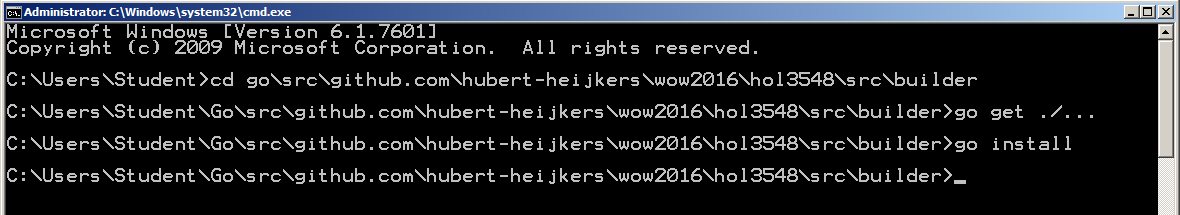
C:\Users\Student\Go\src\github.com\hubert-heijkers\wow2016\hol3548\src\builder

Whereas the ‘builder’ app, read: builder.exe, will end up being put into:

C:\Users\Student\Go\bin

Now let’s go ahead and open a command box and change the directory to builder folder.

cd Go\src\github.com\hubert-heijkers\wow2016\hol3548\src\builder



The code that has been written and you are going to write, directly or indirectly, has dependencies on some third party packages. So before we can compile anything we need to get those dependencies so let’s to that right here, using the following command:

go get ./…

And, while we are at it, let’s build the application as well, and have it ‘installed’ in the bin folder, using:

go install

Congratulations, you’ve build the app. Now go and have a peek in the go bin folder, C:\Users\Student\Go\bin. It should now contain the binary for your application, builder.exe.

### Getting familiar with what’s there already

Before we start coding let’s have a peek at the code that’s already provided. If you look in builder source folder, you’ll notice there are folders, each representing a separate package, in Go speak.

Note: All the code in these packages was written with this example in mind. Shortcuts have been taken, error checking is ignored and assumptions made as such, and therefore this code is by no means meant to be complete or ‘production’ quality, yet is purely to demonstrate the principals involved.

*OData package*:  
This package implements OData specific extensions on top of the build in http package. It implements wrappers for the GET and POST methods, adding some OData specifics to the request as well as error checking, as well as an IterateCollection function that, given the URL to a collection valued OData resource, iterates the collection in one or more roundtrips, building on OData semantics.

*NorthWind package:*Go has built in support for marshalling of structures from and to JSON. In this package you’ll find the structures describing both entity types and responses, with their JSON mapping, we’ll end up consuming from the NorthWind service. If you are interested in taking a look at the metadata for the NorthWind service then, as you did with the TM1 server earlier already, query the metadata document by, like with the TM1 server, adding $metadata to the service root URL as in: <http://services.odata.org/V4/Northwind/Northwind.svc/$metadata>.

*TM1 package:*Using the same JSON mapping as mentioned above, the TM1 package describes those meta data entity types (Cube, Dimension, Hierarchy, Element, Edge etc.), again only specifying those properties that we’ll end up using, from TM1’s REST API, needed by code that we are writing to build our NorthWind model.

Note: Currently there is a difference in JSON encoding of a collection of references being received from the server and a collection of references being send, which still requires the @odata.bind annotation. This is changing in an upcoming version of the OData specification, version 4.01, but until then two separate types will be required, making it all very inconvenient to mix and match. In the code here we do not use the components to define the dimension, only in consumption cases, but rather specify edges, which use the bind notation.

*Processes package:*This is package in which the code resides that does the actual processing of the source data, and which generates the definitions of the dimensions as well as loading data into the model.

The Products, Customers and Employees dimension all follow the same pattern, they iterate the collections of [categories expanded with products](http://services.odata.org/V4/Northwind/Northwind.svc/Categories?$select=CategoryID,CategoryName&$orderby=CategoryName&$expand=Products($select=ProductID,ProductName;$orderby=ProductName)), [customers](http://services.odata.org/V4/Northwind/Northwind.svc/Customers?$orderby=Country%20asc,Region%20asc,%20City%20asc&$select=CustomerID,CompanyName,City,Region,Country) and [employees](http://services.odata.org/V4/Northwind/Northwind.svc/Employees?$select=EmployeeID,LastName,FirstName,TitleOfCourtesy,City,Region,Country&$orderby=Country%20asc,Region%20asc,City%20asc), and generate the dimension structures for the dimension representing them. Have a look at the source data.

Categories expanded with products:  
<http://services.odata.org/V4/Northwind/Northwind.svc/Categories?$select=CategoryID,CategoryName&$orderby=CategoryName&$expand=Products($select=ProductID,ProductName;$orderby=ProductName)>

Customers:  
<http://services.odata.org/V4/Northwind/Northwind.svc/Customers?$orderby=Country%20asc,Region%20asc,%20City%20asc&$select=CustomerID,CompanyName,City,Region,Country>

Employees:  
<http://services.odata.org/V4/Northwind/Northwind.svc/Employees?$select=EmployeeID,LastName,FirstName,TitleOfCourtesy,City,Region,Country&$orderby=Country%20asc,Region%20asc,City%20asc>

Note that we are using $select, $expand and $orderby to select just the data we are interested in and have the data source order them before returning them so we can build on that order.

The Time dimension applies a bit of a different logic. It requests the first and the last order by requesting the orders collection, ordering them by order data, both ascending and descending, and then only asking for the first order to be returned. Using the order date from these to orders it knows the date range for which it subsequently creates a time dimension with years, quarters, months and days. Want to find out yourself what the first and last order dates are then follow the following links:

First order date:  
<http://services.odata.org/V4/Northwind/Northwind.svc/Orders?$select=OrderDate&$orderby=OrderDate%20asc&$top=1>

Last order date:  
<http://services.odata.org/V4/Northwind/Northwind.svc/Orders?$select=OrderDate&$orderby=OrderDate%20desc&$top=1>

The measures dimension is not driven by source data, it simply builds a simple flat dimension with three elements: Quantity, Unit Price and Revenue. Later we’ll see that we’ll define a rule in which we calculate the, average to be exact, Unit Price from Revenue and Quantity.

Leaves us with loading the data into the TM1 server. From a consuming the source OData service it is, once again, simply iterating a collection, in this case the collection of orders but this time expanded with the order details. The processing code doesn’t generate a dimension structure this time but, in this case, we choose to build the JSON payload for the Update request directly into the processor. On the other hand, we don’t need to collect all the data that needs to be loaded but, just because we can, we choose to send an update request per chunk of orders that we receive from the source.

Note that this might not be the typical thing to do as you might want to make all these update logically as one transaction. In that case one could compose one big payload and POST one big update action request to the server or, alternatively, compose a text file with the data, upload it as a blob to the server, write a TI to process the blob and execute that TI.

### Bringing it all together into the builder app

Alright, now that we have all the basic ingredients for building the model taken care of, lets write the code that brings it all together. All the code that needs to be written, and don’t worry we don’t expect you to know, we’ll give you all the snippets, goes into the main.go file. Open up the main.go file in either Notepad++ or Visual Studio, whatever suits you best.

You’ll see that we provided the skeleton for three functions that we’ll have to implement, the main logic and two functions that contain the logic of creating a dimension and cube respectively.

#### The createDimension function

Let’s start with the createDimension function. The createDimension function is the function that makes the appropriate REST API request that, given a specification, using the structures defined in the tm1 package, result in the dimension actually being created in the TM1 server. In our example we will define and associate values to the, built-in, Caption attribute for those elements for which we’d like to show a friendlier name or representation then the, unique, name of the element. To do this we, at least currently, need three REST request, notably:

* A POST of the dimension specification to create the dimension
* A POST of the attribute definition to associate the ‘Caption’ attribute with the elements
* An Update action to update the Caption values for the elements in the dimension

Note that the last step, setting attribute values, could, arguably should, be done thru updates to the LocalizedAttributes collection of localized attribute values. However, that, to date, requires a request per element and locale we are setting values for. We therefore chose to update the element attribute cube, the one containing the ‘default’ values for the attribute, directly using the Update action.

So, let’s start filling in the skeleton. The dimension definition is passed to the function so first thing we need is a JSON representation of it. The first thing we’ll therefore do is marshal the dimension definition into JSON:

// Create a JSON representation for the dimension

jDimension, \_ := json.Marshal(dimension)

That’s all we need to POST to our TM1 server to get the dimension created as in:

// POST the dimension to the TM1 server

fmt.Println(">> Create dimension", dimension.Name)

resp := client.ExecutePOSTRequest(tm1ServiceRootURL+"Dimensions", "application/json", string(jDimension))

Note that ExecutePOSTRequest returns irrespective of the result of executing the request itself, so we’ll have to validate the actual status code that the server responded with. If the request was successful, and the dimension was created successfully, then the server responds with a 201 – created. All other status code, indicate something didn’t go as expected. Let’s add the code to validate just that and break of the process if it failed, while logging the response from the server.

// Validate that the dimension got created successfully

odata.ValidateStatusCode(resp, 201, func() string {

return "Failed to create dimension '" + dimension.Name + "'."

})

resp.Body.Close()

Next we’ll add the ‘Caption’ attribute by posting the attribute definition, which we in lined as the payload for the request here, to the dimension hierachy’s ElementAttributes collection:

// Secondly create an element attribute named 'Caption' of type 'string'

fmt.Println(">> Create 'Caption' attribute for dimension", dimension.Name)

resp = client.ExecutePOSTRequest(tm1ServiceRootURL + "Dimensions('"+dimension.Name+"')/Hierarchies('"+dimension.Name+"')/ElementAttributes", "application/json", `{"Name":"Caption","Type":"String"}`)

Again we’ll test if the request was successful and that the attribute got created successfully:

// Validate that the element attribute got created successfully as well

odata.ValidateStatusCode(resp, 201, func() string {

return "Creating element attribute 'Caption' for dimension '" + dimension.Name + "'."

})

resp.Body.Close()

Now that the Caption attribute has been created we can associate Caption values with the elements in the newly created dimension. As mentioned before we will do so by making an update against the element attributes cube, associate with the dimension.

// Now that the caption attribute exists lets set the captions accordingly for this

// we'll simply update the }ElementAttributes\_DIMENSION cube directly, updating the

// default value. Note: TM1 Server doesn't support passing the attribute values as

// part of the dimension definition just yet (should shortly), so for now this is the

// easiest way around that. Alternatively, one could have updated the attribute

// values for elements one by one by POSTing to or PATCHing the LocalizedAttributes

// of the individual elements.

fmt.Println(">> Set 'Caption' attribute values for elements in dimension", dimension.Name)

resp = client.ExecutePOSTRequest(tm1ServiceRootURL + "Cubes('}ElementAttributes\_"+dimension.Name+"')/tm1.Update", "application/json", dimension.GetAttributesJSON())

The payload that in this request is generated based on a map of captions that we keep track of in the dimension definition, but is formatted for and update request against the element attributes cube.

After the update we’ll, once again, make sure that the update of the caption values succeeded.

// Validate that the update executed successfully (by default an empty response is expected, hence the 204).

odata.ValidateStatusCode(resp, 204, func() string {

return "Setting Caption values for elements in dimension '" + dimension.Name + "'."

})

resp.Body.Close()

The function ends with returning the OData id, which in services that follow convention (which TM1 does) is equal to the canonical URL of the resource, to the newly created dimension.

#### The createCube function

The createCube function makes the appropriate REST API request that, given the specification for a cube, result in the cube to be created in the TM1 server. This only requires one REST request, notably:

* A POST of the cube specification to create the cube

The function takes a cube name, the list of OData ids of the dimensions spanning the cube and the rules that need to be set on the cube. We’ll pass these into a structure defined in the tm1 package, which we subsequently use to marshal into the JSON specification:

// Create a JSON representation for the cube

jCube, \_ := json.Marshal(tm1.CubePost{Name: name, DimensionIds: dimensionIds, Rules: rules})

This JSON specification we subsequently POST to our TM1 server to get the cube created using:

// POST the dimension to the TM1 server

fmt.Println(">> Create cube", name)

resp := client.ExecutePOSTRequest(tm1ServiceRootURL+"Cubes", "application/json", string(jCube))

We obviously want to validate that the cube got created successfully before continuing. Once again we expect the server to respond with a 201 – created. All other status code, indicate something didn’t go as expected. Let’s add the code to validate just that:.

// Validate that the cube got created successfully

odata.ValidateStatusCode(resp, 201, func() string {

return "Failed to create cube '" + name + "'."

})

resp.Body.Close()

The function ends with returning the OData id, which in services that follow convention (which TM1 does) is equal to the canonical URL of the resource, to the newly created cube.

#### The main function

As the name already suggest, main is the function that is executed when the application gets executed and is therefore the function where we kick off the work that we do to build the model.

Note: The complete version of the main.go file is also provided in the wow2016/hol3548/output/builder folder. Feel free to copy that version over to the src/builder folder and safe some time.

# Building a web app that uses your model

------------------------------------------------------------------

THIS IS WHERE WE BUILD A WEB APP USING CANVAS.

------------------------------------------------------------------

# Testing your model directly using the REST API

------------------------------------------------------------------

Using RESTit/v2 validate some assumptions of the created model using the REST API

i.e. The customer dimension should contain x elements of which y are leaves.

------------------------------------------------------------------

# We Value Your Feedback!

* Don’t forget to submit your World of Watson session and speaker feedback! Your feedback is very important to us – we use it to continually improve the conference.
* Access the World of Watson Conference Connect tool to quickly submit your surveys from your smartphone, laptop or conference kiosk.