Lab Center – Hands-on Lab

Session 3548

Session Title Learn how to build Applications with the IBM Cognos TM1, OData Compliant, REST API

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

Table of Contents

Getting ready

As always there are a couple of late minute changes that need to be made, updates to examples, instruction documents that aren't ready in time etc. etc. It 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 built 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 that 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
```

Note, now that your VM is updated you'll find the latest version of this document in the doc folder:

```
%GOPATH%\src\github.com\hubert-heijkers\wow2016\hol3548\doc
```

Having an electronic copy might come in handy later on when you'll be 'writing' some code;-).

3 – Start the 'Planning Sample' TM1 server

Most of the tooling we introduce in the first chapter uses our (in-)famous 'Planning Sample' sample database. On the desktop you'll find 'TM1 – Planning Sample' shortcut. Double click it to start that TM1 server.

That's all, 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 a 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, OData builds on a strong foundation with very clear protocol semantics, URL conventions, a concise metadata definition and a, JSON based, format. OData, albeit coming from a strong data driven background, is all but limited to exposing data in a web friendly way. In laymen's terms, it is set of specifications which we obey by that specify how a service describes what is available to a consumer, how a consumer needs to formulate a request for such server and how the service formats the response to the request.

OData, short for Open-Data, has been developed over a number of years and the latest version, v4 errata 3, is an OASIS standard. The OData standard has also made it to ISO standard in the meantime as well. For more information about the 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' 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 an 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>
```

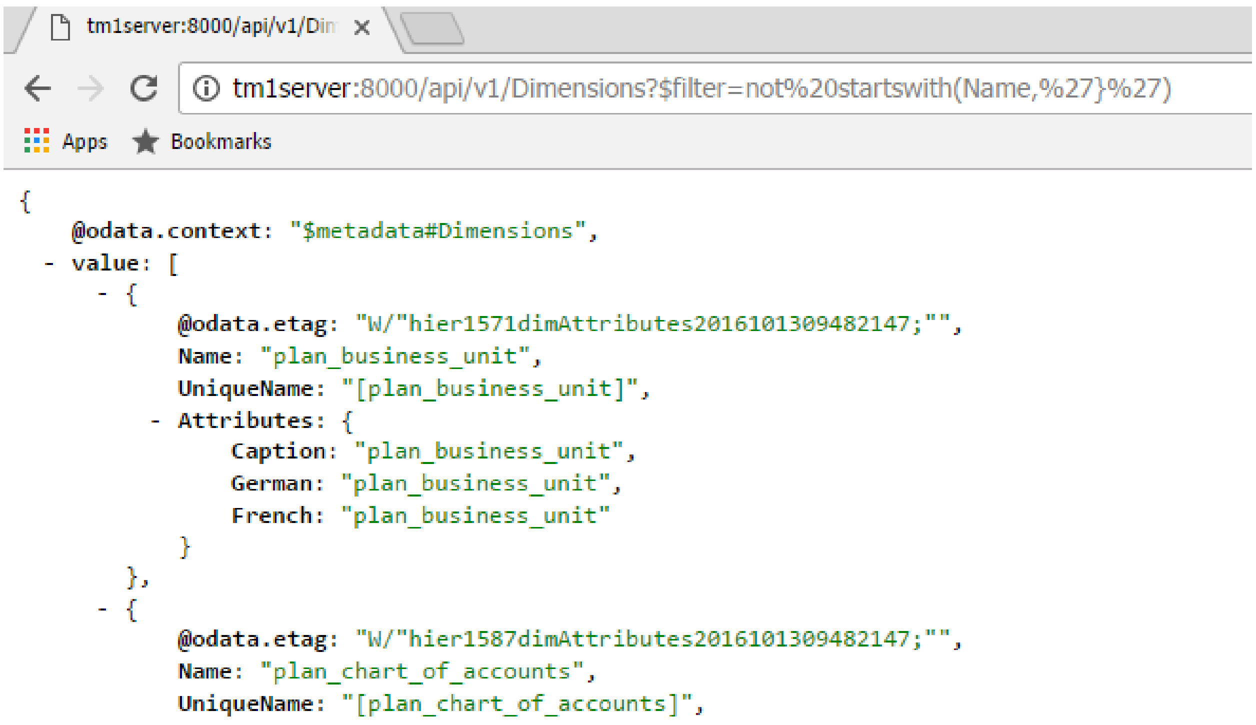
This is telling us that one of the types that the service exposes is a 'Dimension' and that it has a couple of properties among which is its Name, UniqueName and a set of Hierarchies. Note: the version of TM1 you are using still only allows one Hierarchy per Dimensions, which has to have the same name 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).

3) Retrieve those dimensions not being control dimensions by typing the following URL: `http://tm1server:8000/api/v1/Dimensions?$filter=not startswith(Name,'}')`
4) 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.

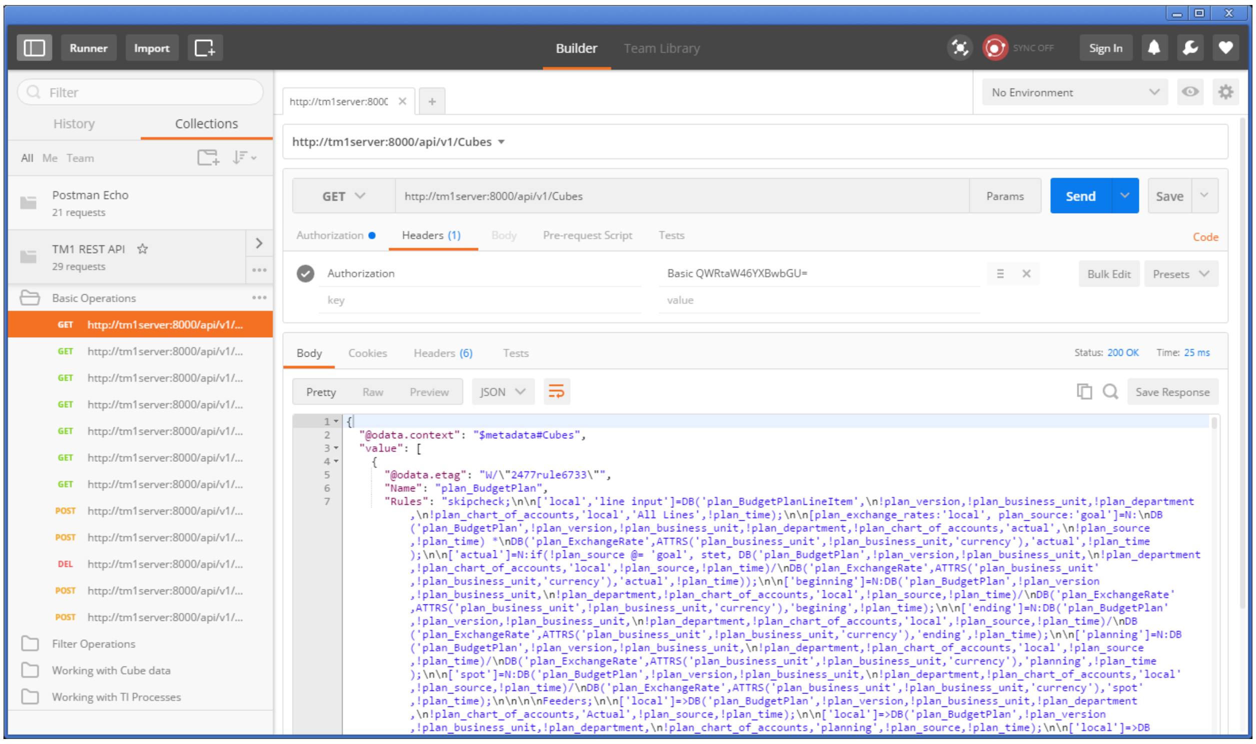

tm1server:8000/api/v1/Dim
tm1server:8000/api/v1/Dimensions?$filter=not%20startswith(Name,%27}%27)
Apps
Bookmarks

```
{
    @odata.context: "$metadata#Dimensions",
  - value: [
      - {
            @odata.etag: "W/"hier1571dimAttributes2016101309482147;"",
            Name: "plan_business_unit",
            UniqueName: "[plan_business_unit]",
          - Attributes: {
                Caption: "plan_business_unit",
                German: "plan_business_unit",
                French: "plan_business_unit"
            }
        },
      - {
            @odata.etag: "W/"hier1587dimAttributes2016101309482147;"",
            Name: "plan_chart_of_accounts",
            UniqueName: "[plan_chart_of_accounts]",
```

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.



GET http://tm1server:8000/api/v1/Cubes

Authorization: Basic QWRtaW46YXBwbGU=

Status: 200 OK Time: 25 ms

```
{
  "@odata.context": "$metadata#Cubes",
  "value": [
    {
      "@odata.etag": "W/\"2477rule6733\"",
      "Name": "plan_BudgetPlan",
      "Rules": "skipcheck;\n\n['local','line input']=DB('plan_BudgetPlanLineItem',\n!plan_version,!plan_business_unit,!plan_department
        ,\n!plan_chart_of_accounts,'local','All Lines',!plan_time);\n\n[plan_exchange_rates:'local', plan_source:'goal']=N:\nDB
        ('plan_BudgetPlan',!plan_version,!plan_business_unit,!plan_department,!plan_chart_of_accounts,'actual',\n!plan_source
        ,!plan_time) *\nDB('plan_ExchangeRate',ATTRS('plan_business_unit',!plan_business_unit,'currency'),'actual',!plan_time
        );\n\n['actual']=N:if(!plan_source @= 'goal', stet, DB('plan_BudgetPlan',!plan_version,!plan_business_unit,\n!plan_department
        ,!plan_chart_of_accounts,'local',!plan_source,!plan_time)/\nDB('plan_ExchangeRate',ATTRS('plan_business_unit'
        ,!plan_business_unit,'currency'),'actual',!plan_time));\n\n['beginning']=N:DB('plan_BudgetPlan',!plan_version
        ,!plan_business_unit,\n!plan_department,!plan_chart_of_accounts,'local',!plan_source,!plan_time)/\nDB('plan_ExchangeRate'
        ,ATTRS('plan_business_unit',!plan_business_unit,'currency'),'begining',!plan_time);\n\n['ending']=N:DB('plan_BudgetPlan'
        ,!plan_version,!plan_business_unit,\n!plan_department,!plan_chart_of_accounts,'local',!plan_source,!plan_time)/\nDB
        ('plan_ExchangeRate',ATTRS('plan_business_unit',!plan_business_unit,'currency'),'ending',!plan_time);\n\n['planning']=N:DB
        ('plan_BudgetPlan',!plan_version,!plan_business_unit,\n!plan_department,!plan_chart_of_accounts,'local',!plan_source
        ,!plan_time)/\nDB('plan_ExchangeRate',ATTRS('plan_business_unit',!plan_business_unit,'currency'),'planning',!plan_time
        );\n\n['spot']=N:DB('plan_BudgetPlan',!plan_version,!plan_business_unit,\n!plan_department,!plan_chart_of_accounts,'local'
        ,!plan_source,!plan_time)/\nDB('plan_ExchangeRate',ATTRS('plan_business_unit',!plan_business_unit,'currency'),'spot'
        ,!plan_time);\n\n\n\nFeeders;\n\n['local']=>DB('plan_BudgetPlan',!plan_version,!plan_business_unit,!plan_department
        ,\n!plan_chart_of_accounts,'Actual',!plan_source,!plan_time);\n\n['local']=>DB('plan_BudgetPlan',!plan_version
        ,!plan_business_unit,!plan_department,\n!plan_chart_of_accounts,'planning',!plan_source,!plan_time);\n\n['local']=>DB
```

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.

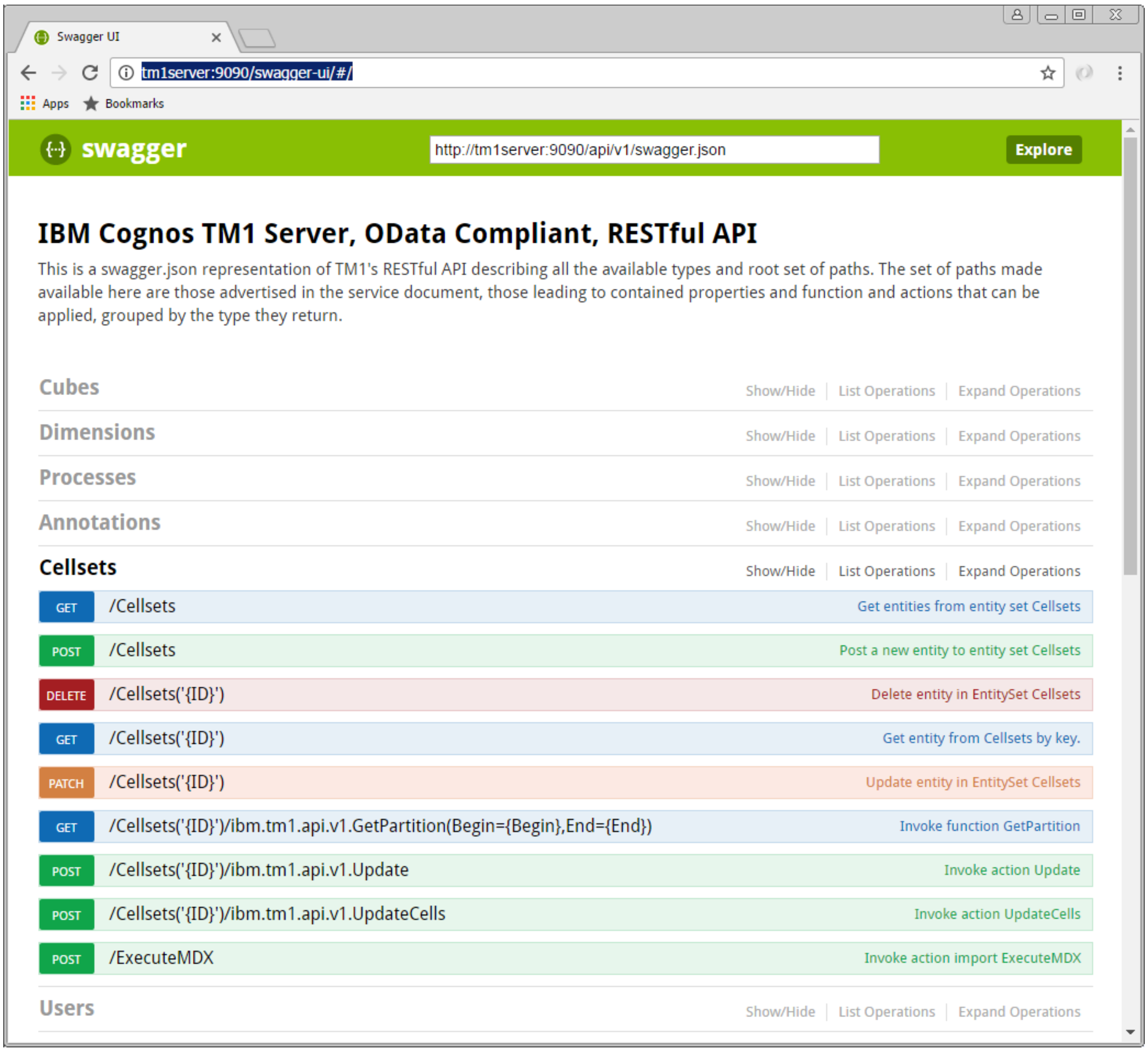
After selecting an example, you can see the definition of the request on the right. Hitting the 'Send' button will execute the request after which the response will be shown to you in the output window. Don't forget to look at the Cookies and Headers tabs in the output pane to see what more is being 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 advise you to download and install it in your environment and have a go. 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!

Working with TM1's REST API using Swagger tooling

OData is not the only attempt to 'standardize', especially the metadata side, REST APIs, others like Swagger and RAML, have been gaining popularity as well. Swagger, with the recent forming of the, broadly industry backed, Open API Initiative, seems to have gotten the upper hand here. This has resulted in the OData Technical Committee seeking alignment with Swagger, and now OpenAPIs, for its JSON based metadata definition.

One great thing about Swagger is the community and tooling around it, most notably the swagger-ui. On the IBM developerWorks community for TM1 SDK you can find an article named 'Using Swagger with TM1 server's, OData compliant, RESTful API'. If you are interested in using Swagger on your own setup this article tells you how to set it up. On the lab VM however, we did the installation and configuration work for you. The only thing you need to do is go to the C:\HOL-TM1SDK\nginx folder and start nginx.exe. After you have done that, open up a browser and point it at: http://tm1server:9090/swagger-ui. It'll open up the Swagger UI and connects thru the nginx proxy, that makes it appear as if our TM1 server now has support for Swagger proper, to our TM1 server directly.



Swagger UI
tm1server:9090/swagger-ui/#/
Apps
Bookmarks
swagger
http://tm1server:9090/api/v1/swagger.json
Explore

IBM Cognos TM1 Server, OData Compliant, RESTful API

This is a swagger.json representation of TM1's RESTful API describing all the available types and root set of paths. The set of paths made available here are those advertised in the service document, those leading to contained properties and function and actions that can be applied, grouped by the type they return.

Cubes — Show/Hide | List Operations | Expand Operations
Dimensions — Show/Hide | List Operations | Expand Operations
Processes — Show/Hide | List Operations | Expand Operations
Annotations — Show/Hide | List Operations | Expand Operations
Cellsets — Show/Hide | List Operations | Expand Operations

GET	/Cellsets	Get entities from entity set Cellsets
POST	/Cellsets	Post a new entity to entity set Cellsets
DELETE	/Cellsets('{ID}')	Delete entity in EntitySet Cellsets
GET	/Cellsets('{ID}')	Get entity from Cellsets by key.
PATCH	/Cellsets('{ID}')	Update entity in EntitySet Cellsets
GET	/Cellsets('{ID}')/ibm.tm1.api.v1.GetPartition(Begin={Begin},End={End})	Invoke function GetPartition
POST	/Cellsets('{ID}')/ibm.tm1.api.v1.Update	Invoke action Update
POST	/Cellsets('{ID}')/ibm.tm1.api.v1.UpdateCells	Invoke action UpdateCells
POST	/ExecuteMDX	Invoke action import ExecuteMDX

Users — Show/Hide | List Operations | Expand Operations

Note the http://tm1server:9090/api/v1/swagger.json link at the top of the screen, which the Swagger UI uses, and therefore you can use directly as well, to retrieve the swagger definition for TM1's REST API.

You can expand and collapse the sections in the interface. Clicking on any of the operations will expand the form for that operation and will tell you about the potential parameters in the request and the metadata about the information send and/or retrieved using that operation.

Note that not everything is necessarily exposed thru this interface, Swagger has its restrictions when it comes to metadata descriptions in comparison to OData and, for one, can't express the recursive nature of operations on types like OData can. If you had a specific need however to have expose some explicitly in an Swagger based environment then you could update the swagger.json file that we provided here and update it accordingly to your own liking.

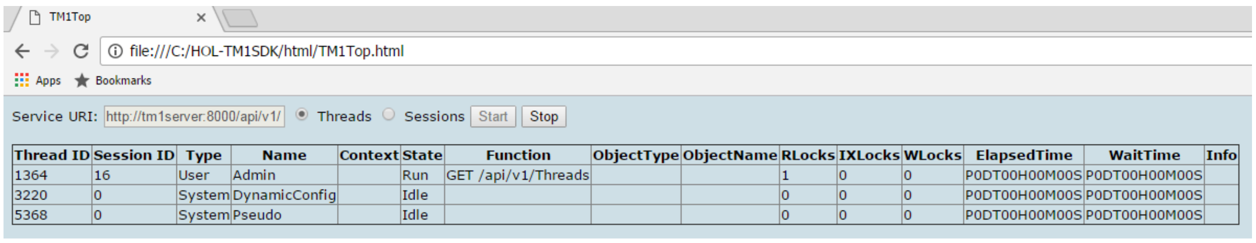
Having issues or additional questions using Swagger UI, don't hesitate to reach out to any of the lab instructors for further information hand help.

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`. 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' article on developerWorks TM1 SDK community.



Service URI: http://tm1server:8000/api/v1/ [✓ Threads] Sessions

Thread ID	Session ID	Type	Name	Context	State	Function	ObjectType	ObjectName	RLocks	IXLocks	WLocks	ElapsedTime	WaitTime	Info
1364	16	User	Admin		Run	GET /api/v1/Threads			1	0	0	P0DT00H00M00S	P0DT00H00M00S	
3220	0	System	DynamicConfig		Idle				0	0	0	P0DT00H00M00S	P0DT00H00M00S	
5368	0	System	Pseudo		Idle				0	0	0	P0DT00H00M00S	P0DT00H00M00S	

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 database, hosted on the OData.org website. 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, that is in the default install location, and the -z option tells the server to go look for the configuration file 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 Go works with need to be organized in places where it knows where to find them. 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, that Go manages the organization of, 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:

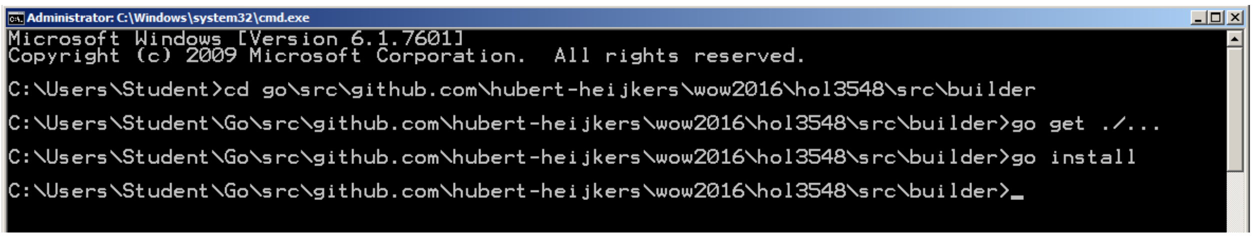
```
C:\Users\Student\Go\src\github.com\hubert-heijkers\wow2016\hol3548\src\builder
```

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
```



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Student>cd go\src\github.com\hubert-heijkers\wow2016\hol3548\src\builder

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

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

C:\Users\Student\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 ./…
```

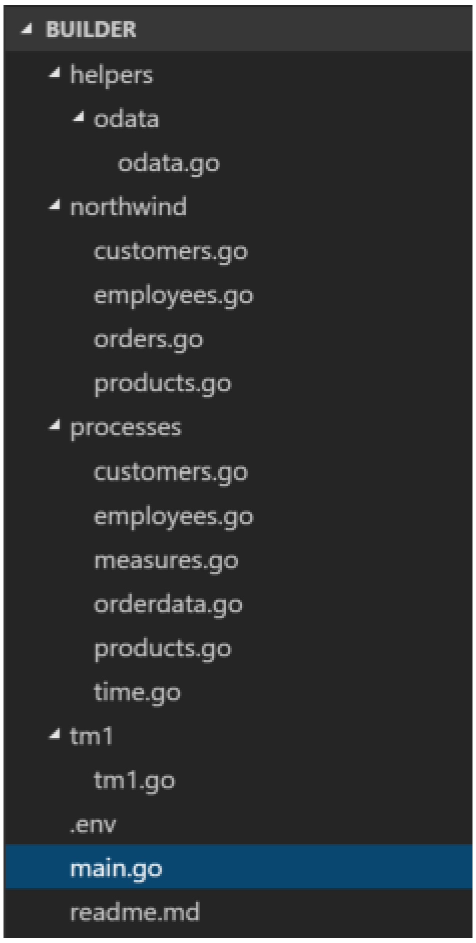
And, while we are at it, let's build the application as well, and have it 'installed' in the bin folder, using:

```
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 that representing a separate 'package' in Go speak.



```
BUILDER
  helpers
    odata
      odata.go
  northwind
    customers.go
    employees.go
    orders.go
    products.go
  processes
    customers.go
    employees.go
    measures.go
    orderdata.go
    products.go
    time.go
  tm1
    tm1.go
  .env
  main.go
  readme.md
```

Note: All 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. For starters it implements wrappers for the GET and POST methods, adding some OData specifics to the request as well as error checking. The IterateCollection function, given the URL to a collection valued OData resource, iterates that 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.). 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 sent, which for now 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, that use the bind notation.

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

The Products, Customers and Employees dimensions all follow the same pattern, they iterate the collections of categories expanded with products, customers and employees, 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 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.

This 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 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 needed to be loaded but, 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 may want to make all these update logically as one transaction. In that case one could compose one large payload and POST one update action request to the server. Alternatively, one could also compose a text file with the data, upload it as a blob to the server, write a TI to process the data contained in the and execute that TI process.

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 how to write Go code, we'll give you all the snippets that need to go into the main.go file. Note that all the snippets contain the comments from the provide skeletons as well, so you don't need to type them and know where the code should go;-). 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 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 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’ status. All other status codes 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+"')/ElementAttribu
tes", "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, representing 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 will 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

Now that we got all the ingredients for our application lets write the main function, the function that gets executed when our application is started. If you look at the skeleton of the function as provided you'll see that it starts by initializing a couple of variables that get loaded from 'environment' variables which themselves get initialized by loading them from the ".env" file.

The steps in the getting ready portion made sure you have a ".env" file in the right location, the go/bin folder in this case, and that it has the correct values to initialize these variables, in this particular case the service root URLs for both the source, our NorthWind database hosted on odata.org, and our target, the TM1 server that you created at the beginning of this exercise.

First we'll create an instance of an http client which we use to execute our HTTP requests. In this case we'll use one that we extended ourselves in our OData package, which allows us to generically take care of some of the OData specifics when making HTTP requests to an OData service. We also need to make sure that once we've been authenticated to a service that any cookies, in TM1's case the TM1SessionId cookie representing our session, are retained for the duration of our session. To do so we'll have to initialize a so called cookie jar as well. Not that this is a very common pattern in any http library in any language you'll end up using. With initializing some form of cookie storage it is often very hard, if not impossible, to retain/manage your session. Here is the code you need to inject to do exactly that:

```
// Create the one and only http client we'll be using, with a cookie jar enabled
to keep reusing our session
client = &odata.Client{}
cookieJar, _ := cookiejar.New(nil)
client.Jar = cookieJar
```

Next we'll make sure that we connect to the TM1 server. We'll write out this first request here as we'll have to add credentials to authenticate with our server and thereby trigger the server to give us the session cookie for the authenticated user. The request we'll use is a simple request for purely the server version. You can do this in a browser directly to by following this URL:

http://tm1server:8088/api/v1/Configuration/ProductVersion/$value

Note the /$value at the end of the URL. This, OData defined, path segment instructs the server to return the value for the property, in this case the product version, in raw, text in this case, format. In this case:

```
10.2.20600.66
```

We don't use this value for anything other than dumping it out to the console to show which version of TM1 server we are working with but, on could envision using this to validate a minimal version required or even, as a shortcut instead of evaluating the $metadata document as one should, make some chooses as to what to support or how to implement knowing what version it was. So here is the code we need to set up the request, set the, in this case because we use authentication mode 1, basic authentication, execute the request and, dump, after checking we got a 200 – OK status, the content of the response, the server version, to the console:

```
// Validate that the TM1 server is accessable by requesting the version of the
server
req, _ := http.NewRequest("GET",
tm1ServiceRootURL+"Configuration/ProductVersion/$value", nil)

// Since this is our initial request we'll have to provide a user name and
// password, also conveniently stored in the environment variables, to
authenticate.
// Note: using authentication mode 1, TM1 authentication, which maps to basic
// authentication in HTTP[S]
req.SetBasicAuth(os.Getenv("TM1_USER"), os.Getenv("TM1_PASSWORD"))

// We'll expect text back in this case but we'll simply dump the content out and
// won't do any content type verification here
req.Header.Add("Accept", "*/*")

// Let's execute the request
resp, err := client.Do(req)
if err != nil {
	// Execution of the request failed, log the error and terminate
	log.Fatal(err)
}

// Validate that the request executed successfully
odata.ValidateStatusCode(resp, 200, func() string {
	return "Server responded with an unexpected result while asking for its
version number."
})

// The body simply contains the version number of the server
version, _ := ioutil.ReadAll(resp.Body)
resp.Body.Close()

// which we'll simply dump to the console
fmt.Println("Using TM1 Server version", string(version))
```

Once again, after having executed this request the server also returns a new cookie, named TM1SessionId, which got stored in the cookie jar we created earlier. Note that, especially in browsers, if you end up writing code in JavaScript for example like the TM1Top example earlier, you will not have direct access to these cookies and will depend on the underlying http client to handle these correctly.

Alright, now that we know we can establish a connection to our TM1 server and are authenticated let’s run some of our ‘processes’ to create some dimensions to being with. You might recall that the createDimension function returned the OData id, a.k.a. the reference which in TM1’s case always happens to be the canonical URL as well, which we’ll need to pass to the createCube function later. So we’ll create an array here and store those dimension ids in it.

Creating the dimensions themselves has become ‘as simple as’ calling the function that generates the specification for it, as described earlier, and passing that definition to the createDimension function that we wrote just now. The only parameters we’ll pass to those generation functions are the http client we are using, the service root URL from our data source, the NorthWind database in our case, and the name of the dimension to be generated. In code this looks like:

```
// Now let's build some Dimensions. The definition of the dimension is based on
data
// in the NorthWind database, a data source hosted on odata.org which can be
queried
// using its OData complaint REST API.
var dimensionIds [5]string
dimensionIds[0] = createDimension(proc.GenerateProductDimension(client,
datasourceServiceRootURL, productDimensionName))
```

```
dimensionIds[1] = createDimension(proc.GenerateCustomerDimension(client,
datasourceServiceRootURL, customerDimensionName))
dimensionIds[2] = createDimension(proc.GenerateEmployeeDimension(client,
datasourceServiceRootURL, employeeDimensionName))
dimensionIds[3] = createDimension(proc.GenerateTimeDimension(client,
datasourceServiceRootURL, timeDimensionName))
dimensionIds[4] = createDimension(proc.GenerateMeasuresDimension(client,
datasourceServiceRootURL, measuresDimensionName))
```

Now that we have the dimension we need to create a cube, which we'll do using the createCube function you wrote a little earlier. This function takes a name, the set of dimension ids representing the dimensions spanning the cube, and a set of rules to be used by the cube. The set of dimension ids we created above, the only remaining thing is the rules. As you might have seen in the measures dimension generation code already, we create three measures, Quantity, Unit Price and Revenue. Even though the data from the orders we'll be loading has Quantity and Unit Price, there is no easy way to aggregate those if we incrementally load data the way we do. We therefore store Quantity and Revenue, as a simple multiplication of Quantity * Unit Price, and we'll add a rule that calculates our Unit Price later on, an average in this case. We'll also add a feeder to make sure that Unit Price doesn't get suppressed if null/empty suppression is request. So the rules we'll be using are:

```
UNDEFVALS;
SKIPCHECK;

['UnitPrice']=['Revenue']\['Quantity'];

FEEDERS;
['Quantity']=>['UnitPrice'];
```

Ok, now that we have everything let's have the server create that cube!

```
// Now that we have all our dimensions, let's create cube
createCube(ordersCubeName, dimensionIds[:],
"UNDEFVALS;\nSKIPCHECK;\n\n['UnitPrice']=['Revenue']\\['Quantity'];\n\nFEEDERS;\n[
'Quantity']=>['UnitPrice'];")
```

Now that we have a cube we can start loading data into it. This we'll do using the LoadOrderData function, implemented in the processes packages as discussed earlier. We'll again simply pass the service root URLs, the cube and dimension names on to the function. Obviously this load function was written with this particular target cube in mind but we wanted to keep the names for both cube and dimensions configurable, while not reusing or building on anything that happened necessarily before in the same process. Here is how to call that load function:

```
// Load the data in the cube
proc.LoadOrderData(client, datasourceServiceRootURL, tm1ServiceRootURL,
ordersCubeName, productDimensionName, customerDimensionName,
employeeDimensionName, timeDimensionName, measuresDimensionName)
```

That concludes the code writing portion of this exercise. Now go back to the console window you opened earlier and build and install your app by typing:

```
go install
```

After successful compilation of the code you can now run the app. Open up another console window and go to the bin folder with the binaries by typing:

```
cd %GOPATH%\bin
```

which should take you to C:\Users\Student\Go\bin. In this folder you now find builder.exe and the .env file that was dropped there while we got ready for the lab. Type:

```
builder
```

And your application should fire up and, after telling you which version of TM1 you are using, start firing requests to the NorthWind database and TM1 server to build your model. All the requests, the request payloads in case of POST requests and the responses from GET requests are dumped out to the console for you to see what really happens under the covers. Once it's done processing everything successfully you should see a last line like this one in your output:

```
>> Done!
```

If you get an error message that you can't resolve yourself, please ask one of the instructors in the room for help.

Having a look at the results

A quick way to have a peek at the cube and dimensions we created is by simply querying them and visually validating the returned JSON. Try executing the following request:

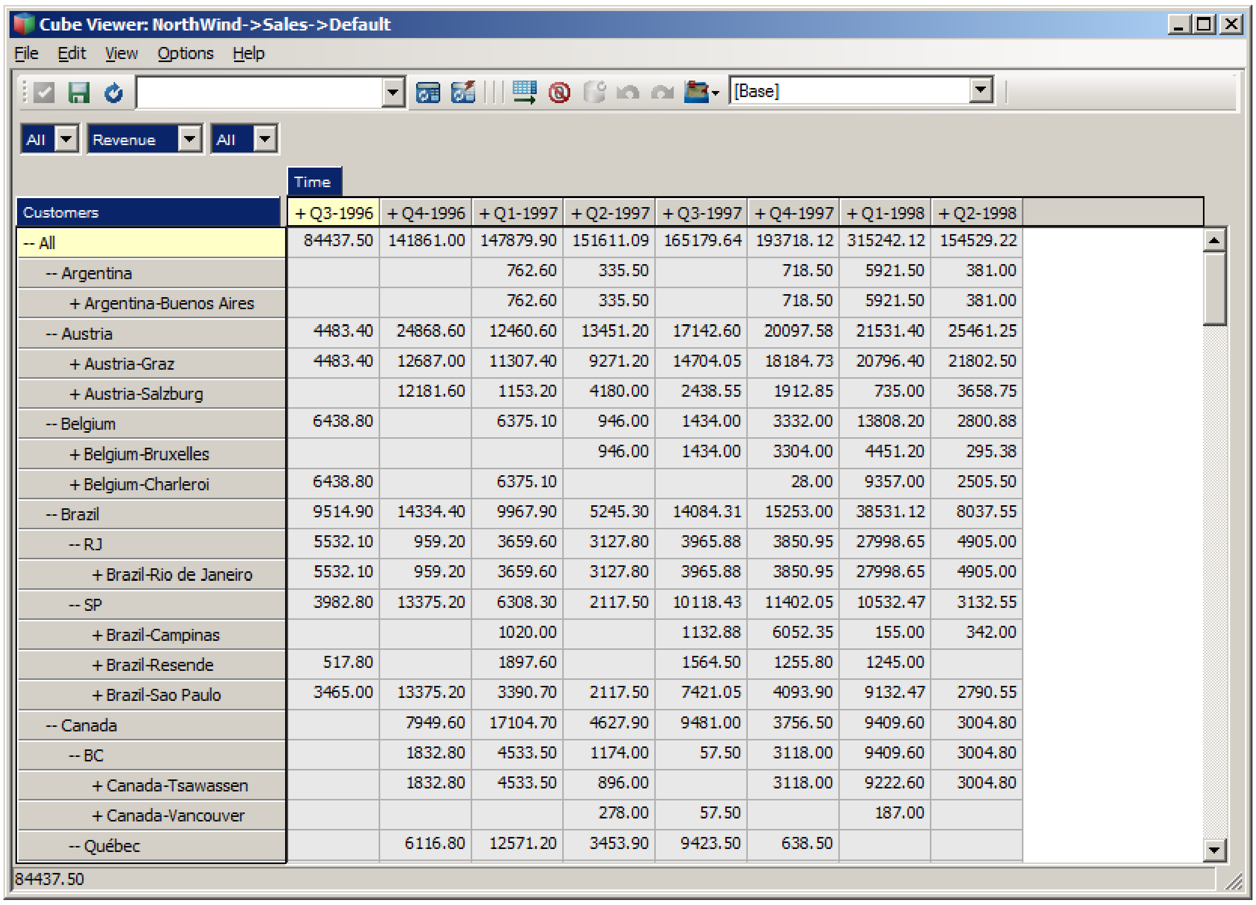
http://tm1server:8088/api/v1/Cubes('Sales')/Dimensions?$select=Name&$expand=Hierarchies($select=Name;$expand=Members($filter=Parent%20eq%20null;$select=Name;$expand=Children($select=Name;$expand=Children($select=Name;$expand=Children($select=Name;$expand=Children($select=Name))))))&$format=application/json;odata.metadata=none

Notice that we are using the Cube's dimensions collection to limit the set of dimensions to just those referenced by the cube. You'll also notice that we are specifying the $format query option, as defined in the OData specification. This query options overwrites the 'Accept' header. In this case we add the 'odata.metadata' parameter, which defaults to minimal, to none here to minimize any additional information one doesn't need in the response. In this case it only removes a couple of etag annotations but, when requesting entities without including their key properties, which often happens in the TM1 case when choosing the, ironically, more descriptive, unique names, the odata.id annotations will be removed as well. This can be handy in keeping responses readable and small.

Lastly you'll notice that, once at the element level, we'll first filter the elements to just the roots, by checking if the parent is equal to null or not, and then expand the components recursively. One that would have read the OData specification might wonder why not use $level here. Well, the answer is that TM1, as is, doesn't, yet, support $level in $expand constructs. This is why we recursively expand enough times to cover the maximum depth used across all dimensions in the model.

Obviously you can use any of the tools we discussed earlier already and connect any of them to the newly created server and have a look at what is in there.

For example, open up architect, connect it to our NorthWind server and create a view and explore the data that is in the model that you just created.


Cube Viewer: NorthWind->Sales->Default

All | Revenue | All

Customers	+ Q3-1996	+ Q4-1996	+ Q1-1997	+ Q2-1997	+ Q3-1997	+ Q4-1997	+ Q1-1998	+ Q2-1998
-- All	84437.50	141861.00	147879.90	151611.09	165179.64	193718.12	315242.12	154529.22
-- Argentina			762.60	335.50		718.50	5921.50	381.00
+ Argentina-Buenos Aires			762.60	335.50		718.50	5921.50	381.00
-- Austria	4483.40	24868.60	12460.60	13451.20	17142.60	20097.58	21531.40	25461.25
+ Austria-Graz	4483.40	12687.00	11307.40	9271.20	14704.05	18184.73	20796.40	21802.50
+ Austria-Salzburg		12181.60	1153.20	4180.00	2438.55	1912.85	735.00	3658.75
-- Belgium	6438.80		6375.10	946.00	1434.00	3332.00	13808.20	2800.88
+ Belgium-Bruxelles				946.00	1434.00	3304.00	4451.20	295.38
+ Belgium-Charleroi	6438.80		6375.10			28.00	9357.00	2505.50
-- Brazil	9514.90	14334.40	9967.90	5245.30	14084.31	15253.00	38531.12	8037.55
-- RJ	5532.10	959.20	3659.60	3127.80	3965.88	3850.95	27998.65	4905.00
+ Brazil-Rio de Janeiro	5532.10	959.20	3659.60	3127.80	3965.88	3850.95	27998.65	4905.00
-- SP	3982.80	13375.20	6308.30	2117.50	10118.43	11402.05	10532.47	3132.55
+ Brazil-Campinas			1020.00		1132.88	6052.35	155.00	342.00
+ Brazil-Resende	517.80		1897.60		1564.50	1255.80	1245.00	
+ Brazil-Sao Paulo	3465.00	13375.20	3390.70	2117.50	7421.05	4093.90	9132.47	2790.55
-- Canada		7949.60	17104.70	4627.90	9481.00	3756.50	9409.60	3004.80
-- BC		1832.80	4533.50	1174.00	57.50	3118.00	9409.60	3004.80
+ Canada-Tsawassen		1832.80	4533.50	896.00		3118.00	9222.60	3004.80
+ Canada-Vancouver				278.00	57.50		187.00	
-- Québec		6116.80	12571.20	3453.90	9423.50	638.50		

84437.50

I hope this exercise was helpful in getting some insight in using OData, using REST API programmatically and how to use the REST API to manipulate data and metadata in TM1.

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 save time.

If you want to look at the end result without having to build it yourself, or if for some reason like technical difficulties, the data for the NorthWind model can also be found in the wow2016/hol3548/output/NorthWind.

Processing Logs using the REST API

If you made it here you've learned already how to use and manipulate data and metadata in a TM1 server. So now it's, time permitting, time for a more advanced topic.

In this chapter we'll discuss a second app called the 'watcher', once again written in Go. This app will 'monitor' the transaction log in this case, and will dump any new changes recorded in it to the console.

The goal of this example is to make you familiar with the support, for now on our transaction and message logs, of deltas. For more specifics about delta and the odata.track-changes preference, see the OData Protocol document, specifically section 11.3 Requesting Changes.

This application, which we provide the code for, is building on the some of the helper code from the example in the previous example, most notably the TM1 and OData helper packages. The TM1 server we are targeting in this example is the server you just build in the previous chapter.

Ok, let's have a look at the code.

You might have noticed there were a couple of additional structures and functions in the TM1 and OData packages.

The OData package has a TrackCollection function, arguably the most interesting function in this example. The TrackCollection function, like the IterateCollection, iterates the collection specified by the URL. However there are two differences. It adds the odata.track-changes preference in the request, by means of the 'Prefer' header which, as a result of this header being added, will trigger the service, in this case TM1, to add a so called delta link, using the odata.deltaLink annotation, to the end of the payload, containing the URL that can be used at a later point in time to retrieve any changes/deltas to the collection requested in the initial request. The TrackCollection continues requesting changes after the specified interval, a duration, passed to the function.

In the TM1 package you might have noticed that there was already an TransactionLogEntry and TransactionLogEntriesRequest representing the TransactionLogEntry entity and a response that returns a collection of this entity. These are used to unmarshal the response from the TM1 server on any of these requests.

The sample itself only contains one single file, the main.go file. The main function in this file is pretty much the same as the one you wrote for the builder app in the previous chapter with the exception of the last couple of lines. This is where the TrackCollection function is called with the URL to the transaction log entries. Note that we are asking the server to only return those entries for the Sales cube we created earlier, and to repeat it every second. Lastly the processTransactionLogEntries is being passed as the function to be called with the response of any of the requests, which is the only other function in our main source file.

Want to have a quick peek at what such result looks like? Execute the following request in a browser:

http://tm1server:8088/api/v1/TransactionLogEntries?$filter=Cube%20eq%20'Sales'

Note that the transaction log contains process messages as well but those records don't have a value for the Cube property. As a result of filtering for the 'Sales' cube the process messages are filtered out too.

The processTransactionLogEntries function, which is very similar to the process function in our dimension build processes in our previous builder sample, unmarshals the response from the server, iterates the entries in the collection and builds a nicely readable representation and prints it to the console. It returns any next link or delta link that is in the response, note there, as per the OData conventions always either a next link or a delta link, never both.

To build the application, like before, go to your console window, make sure you are now in the 'watcher' folder and use Go to build and install your app by typing:

```
go install
```

After the successful compilation of the code you can now run the app. Open another console window and go to the bin folder with the binaries by typing:

```
cd %GOPATH%\bin
```

which should take you to C:\Users\Student\Go\bin. In this folder you now find watcher.exe and the .env file that was dropped there while we got ready for the lab. Type:

```
watcher
```

And your application should fire up and, after telling you which version of TM1 you are using, start showing you any transaction log entries your server already has followed by any new changes as they are coming in.

One way of testing is to open up Architect once again and make some changes to the data in the Sales cube. If you spread data, you'll actually see multiple changes happen as the transaction log records the leaf level changes.

Now that you have this app you could actually start after creating the new server in the previous chapter but before actually running the builder. Once you start the builder you'll see transaction flow in while the builder is loading and writing them. Pretty cool eh?

As mentioned earlier, TM1 supports tracking changes on both transaction log and message log. If you are interested in processing message log entries as they happen, you could use this sample as the base for writing that message log processor. Want to react to users logging in or out? Want to write the message log, or transaction log, entries to a database? It has all become relatively easy to do this in real time.

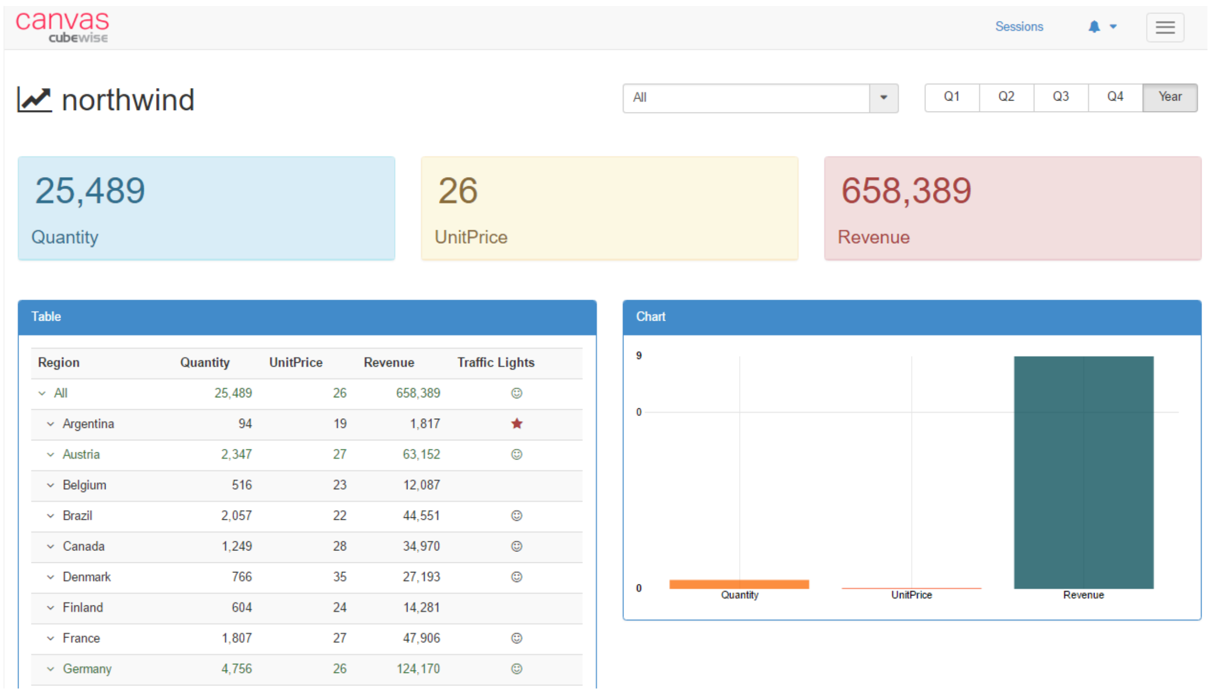
Building a web app ‘on top of’ TM1

The most asked question when talking about the REST API and web apps most likely is: Does this mean I need to use TM1’s REST API from within the browser? Java-Script? On the client?

The truth of the matter is that there isn’t a yes or no answer. Essentially any request to a service from within a client side script is technically the same then a request something like TM1’s REST API but does one need to consume the REST API from the client? No. Could one, most definitely, one example of using TM1’s REST API you saw already earlier when you had a peek at the TM1Top Lite example. If you didn’t look at the script in the HTML file back then, have a peek at it now and notice that, like you ended up doing in the code of the builder and watcher apps, it makes requests to TM1’s REST API.

More often than not the client to TM1 server will end up being a ‘service’ itself again. These services typically combine functionality for a specific purpose or expose frameworks other people can more easily build upon again. A very nice example of this is the ‘Canvas for TM1’ framework developed by a business partner in Australia, Cubewise, an early adopter of TM1’s REST API.

Cubewise allowed us to use their Canvas framework for this Hands-On Lab, allowing you to build dashboards on your newly creates server like the one in the following screen shot:



25,489 Quantity | 26 UnitPrice | 658,389 Revenue

Table

Region	Quantity	UnitPrice	Revenue	Traffic Lights
All	25,489	26	658,389	☺
Argentina	94	19	1,817	★
Austria	2,347	27	63,152	☺
Belgium	516	23	12,087	
Brazil	2,057	22	44,551	☺
Canada	1,249	28	34,970	☺
Denmark	766	35	27,193	☺
Finland	604	24	14,281	
France	1,807	27	47,906	☺
Germany	4,756	26	124,170	☺

The only thing you need to do to get started is to execute the canvas-download.exe, located in the C:\HOL-TM1SDK folder, to have the Canvas framework and services downloaded and installed.

All the required components will be put in a newly created folder named ‘canvas’ in the C:\HOL-TM1SDK. You are now ready to start building your first Canvas based dashboard.

The detailed instructions on how to start Canvas and build your first Canvas dashboard can be found in the Canvas instructions document located in the wow2016/hol3548/doc folder, next to the digital version of this document.

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.