

IBM Analytics University 2017

Fast track your insights

Berlin: October 10-13 | New Orleans: October 17-19

Brought to you by *LearnQuest*

IBM Planning Analytics 2.0 SDK

How TM1 Server's REST APIs Can Be Applied to Create Integrated Solutions

Guido Tejeda Davila, IBM, gtejeda@us.ibm.com (Berlin)

Hubert Heijkers, IBM, hubert.heijkers@nl.ibm.com (New Orleans)



Table of Contents

Getting ready	3
Introducing the OData compliant RESTful API	4
A first peek at TM1's RESTful API	4
Explore the REST API	6
Working with TM1's REST API using Swagger tooling	7
A real life HTML/JavaScript based TM1 client app: TM1Top Lite	8
Building your own Alternate Hierarchies using TI	9
Let's create a new dimension with its first hierarchy	9
Validate that the new dimension got created successfully using the REST API	11
Now let's add a second hierarchy to our newly created dimension	11
Validate that we now have a dimension with two, correction, three hierarchies!	12
Adding an 'attribute based' hierarchy	13
Creating and populating the attributes	13
Creating a hierarchy based on attribute values	14
Add a rule with string data dependency and explore its behavior	16
Building a model using the REST API	19
Setting up a new TM1 server	19
Building the model using the REST API	20
Getting ready to do some coding	20
Getting familiar with what's there already	21
Bringing it all together into the builder app	23
Having a look at the results	29
Processing Logs using the REST API	31
We Value Your Feedback!	33
Acknowledgements and Disclaimers	34



Getting ready

To be able to give you the best experience possible, and to allow us, authors, to be able to make last minute changes to the setup, samples and instructions for this Hands-On Lab, and because in our experience there is always something that we want to change last minute 😊, we've build in a 'get out of jail free card'.

As such there are a couple of 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.

Open a command box and execute the following command to grab the content of this repository:

```
git clone https://github.com/hubert-heijkers/iau17hol
```

Now let's go to the folder holding the actual update:

```
cd iau17hol\vmupdate
```

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. This update can be executed by typing the following command in the command box:

```
vmupdate.bat
```

Your VM is now up to date. You can now find the latest version of this document here:

```
C:\HOL-TM1SDK\Documents
```

Having an electronic copy of the instructions, most notably the Word document, might come in handy later when you'll be 'writing' some code;-).

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, many fix packs and a new major release have been released, all with additional improvements and extensions to the REST API. It is safe to say that by now it is a very mature, not to forget best performing, API we have available for TM1. And in case anybody still doubts it, 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 such 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](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="AllLeavesHierarchyName" 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="HierarchyAttributes"
Type="Collection(ibm.tm1.api.v1.AttributeDefinition)" ContainsTarget="true"/>
  <NavigationProperty Name="LocalizedAttributes"
Type="Collection(ibm.tm1.api.v1.LocalizedAttributes)" ContainsTarget="true"/>
</EntityType>
```



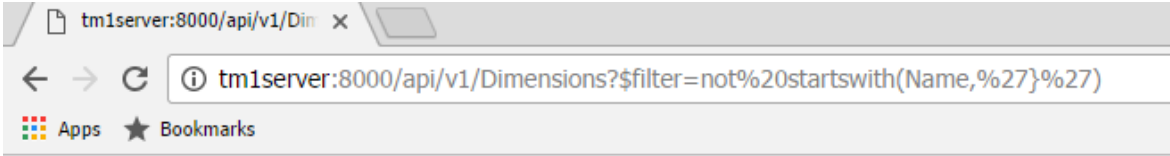
This is telling us that one of the types that the service exposes is a 'Dimension' and that it has a couple of properties among which is its Name, UniqueName and a set of Hierarchies. The Name is the property that uniquely identifies a Dimension, and as such is declared to be the key. And since in this lab you'll be working with the latest and greatest TM1 Server, version 11, this version now has support for alternate hierarchies! This lab assumes you know what alternate hierarchies are, but if you don't, just think about them as separate hierarchies rolling up, consolidating if you will, the same set of leaf elements. And after adding a second alternate hierarchy you'll notice a, system maintained, 'all leaves' hierarchy show up as well. This hierarchy contains a flat list of all the leaves introduced/used across all alternate hierarchies. Please take note of the fact that not all leaves need to be used in all alternate hierarchies although in a typical case they would be. The new 'AllLeavesHierarchyName' property of a dimension, as the name already implies, can be used to overwrite the default "All Leaves" name of this all leaves hierarchy.

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 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%20startswith\(Name,'%27}%27'\)](http://tm1server:8000/api/v1/Dimensions?$filter=not%20startswith(Name,'%27}%27'))
- 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 infamous "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.



```

{
  @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]",

```





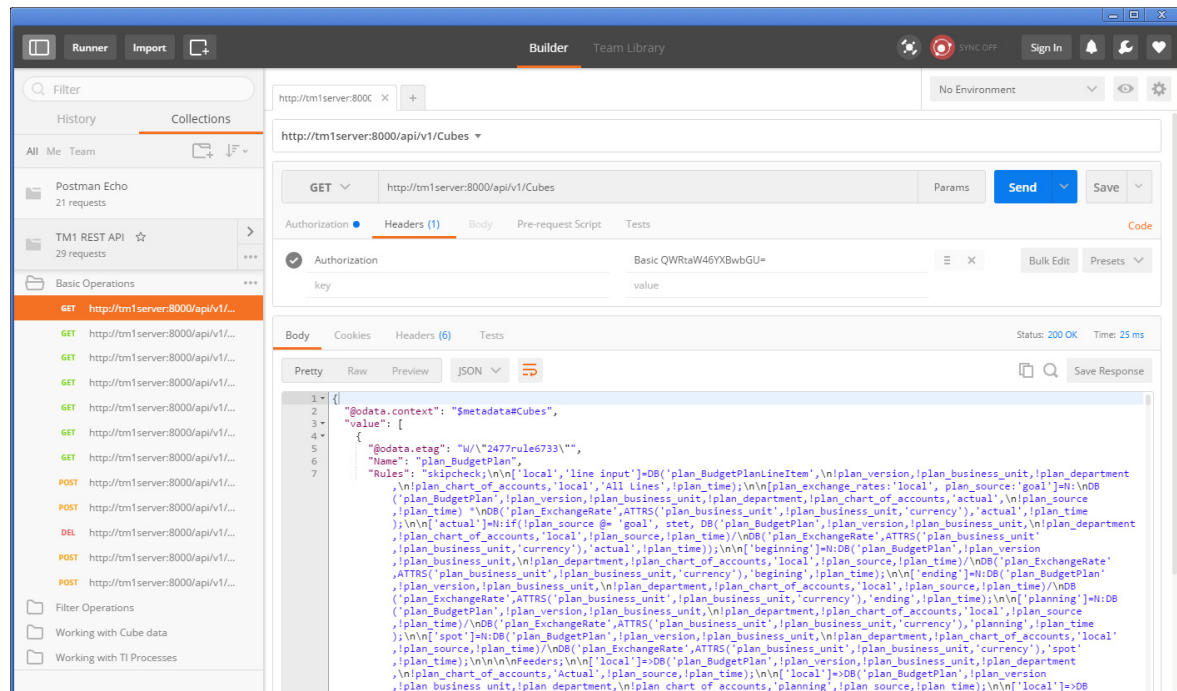
If you want to see what went over ‘the wire’ you can start Fiddler, by clicking the icon in the taskbar. Once Fiddler is up it’ll start recording HTTP traffic and you can look at the requests going to and the responses returned by the server. This way you’ll see for example that the JSON going over the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



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!

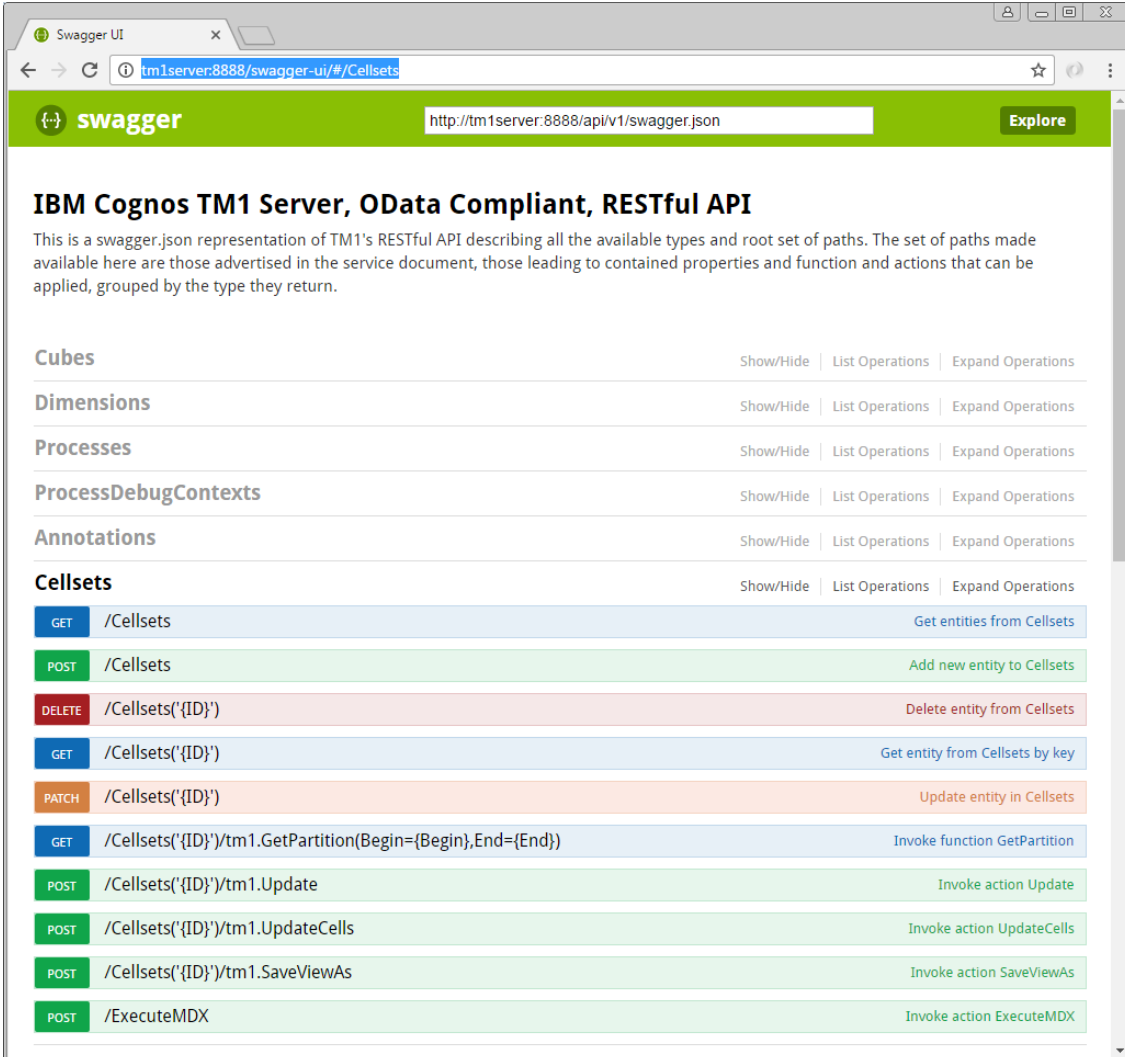


Working with TM1's REST API using Swagger tooling

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 forming of the, broadly industry backed, [Open API Initiative](#), seems to have gotten the upper hand here.

The [OData Technical Committee](#) has been working with [Swagger](#), now [OpenAPIs](#), for its JSON based CSDL format but, given limited expressiveness and support for some key OData constructs, hasn't led to any alignment between two metadata format definitions.

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'. So even though there is no loss-less translation from the OData CSDL to a Swagger definition, you can, if you are interested in using Swagger UI on your own setup, follow this article to set things up. On the lab VM however, we did the installation and configuration work for you. Simply open your browser and point it at: <http://tm1server:8888/swagger-ui>. The Swagger UI pops up and connects thru the NGINX proxy, that makes it appear as if our TM1 server itself has support for Swagger, to our TM1 server directly.



Swagger UI

tm1server:8888/swagger-ui/#/Cellsets

swagger http://tm1server:8888/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.

	Show/Hide	List Operations	Expand Operations
Cubes			
Dimensions			
Processes			
ProcessDebugContexts			
Annotations			
Cellsets			
GET /Cellsets			Get entities from Cellsets
POST /Cellsets			Add new entity to Cellsets
DELETE /Cellsets('{ID}')			Delete entity from Cellsets
GET /Cellsets('{ID}')			Get entity from Cellsets by key
PATCH /Cellsets('{ID}')			Update entity in Cellsets
GET /Cellsets('{ID}')/tm1.GetPartition(Begin={Begin},End={End})			Invoke function GetPartition
POST /Cellsets('{ID}')/tm1.Update			Invoke action Update
POST /Cellsets('{ID}')/tm1.UpdateCells			Invoke action UpdateCells
POST /Cellsets('{ID}')/tm1.SaveViewAs			Invoke action SaveViewAs
POST /ExecuteMDX			Invoke action ExecuteMDX



Note the <http://tm1server:8888/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 expose some of the missing functionality explicitly in a Swagger based environment, then you could update the swagger.json file that we provided here and update it accordingly to your own liking.

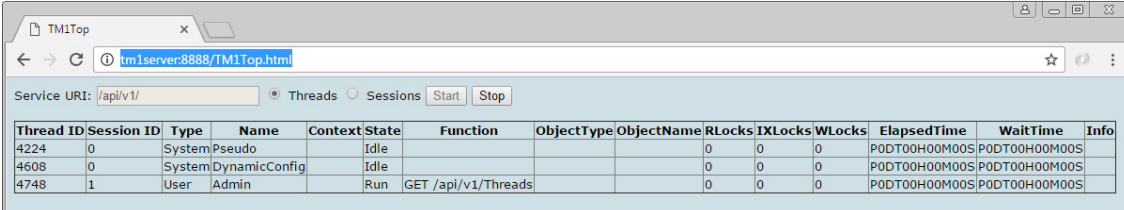
Having issues or additional questions using Swagger UI, don't hesitate to reach out to any of the lab instructors for further information hand help.

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. Not pretty but functional. Note that, since we are using a recent enough version, it is capable of showing both a session and threads view.

You can find the sample at <http://tm1server:8888/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 select 'View page source' from the pop-up menu.

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](#).



The screenshot shows a web browser window titled 'TM1Top' with the address bar displaying 'tm1server:8888/TM1Top.html'. Below the address bar, there is a 'Service URI' field set to '/api/v1/' and two radio buttons for 'Threads' (selected) and 'Sessions'. There are also 'Start' and 'Stop' buttons. The main content area displays a table with the following data:

Thread ID	Session ID	Type	Name	Context	State	Function	ObjectType	ObjectName	RLocks	IXLocks	WLocks	ElapsedTime	WaitTime	Info
4224	0	System	Pseudo		Idle				0	0	0	P0DT00H00M00S	P0DT00H00M00S	
4608	0	System	DynamicConfig		Idle				0	0	0	P0DT00H00M00S	P0DT00H00M00S	
4748	1	User	Admin		Run	GET /api/v1/Threads			0	0	0	P0DT00H00M00S	P0DT00H00M00S	

HTML/JavaScript is only one of the many ways to consume TM1's REST API. In the next section, we'll show you how to build applications that connect to TM1 using the Go programming language.

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 own Alternate Hierarchies using TI

Now that you know what Alternate Hierarchies lets dive right into creating a couple using TI, which we do by adding some hierarchies to the SData sample model that already is up and running on this box.

For your convenience, we already added some supporting files to the SData model, these being:

```
CreateRegionAtts.pro  
CreateTwoAttHierarchy.pro  
MakeDimension.pro  
MakeHierarchy.pro
```

These files can be found here: C:\Program Files\IBM\cognos\tm1_64\samples\tm1\SData

There are also three CSV files, Cities.csv, Cities2.csv and Countries.csv, which we'll be using as the source in some of our TI processes.

Let's create a new dimension with its first hierarchy

Before being able to add additional hierarchies to a dimension we need to have a dimension first. For this exercise let's create a new one based on data we have in one of our CSV files, Cities.csv in this case.

The CSV file contains all the cities we want to represent in our dimension and has four columns, three of which are describing a geographical hierarchy.

```
Helena, Montana, US, West  
Redmond, California, US, West  
Kulob, Khatlon, Tajikistan, East  
...
```

Open the MakeDimension process in Architect

Turbo Integrator: SData->MakeDimension

File Edit Help

Data Source Variables Maps Advanced Schedule

Datasource Type

☐ QDBC

☒ Text

☐ ODBC

Cube

☒ IBM Cognos TM1

Cube View

☐ IBM Cognos Package

Package

☒ None

Data Source Name: C:\Program Files\IBM\cognos\tm1_64\samples\tm1\SData\Cities.csv

Data Source Name On Server: C:\Program Files\IBM\cognos\tm1_64\samples\tm1\SData\Cities.csv

Delimiter Type

☒ Delimited

☐ Fixed Width

Delimiter

☐ Tab ☐ Space ☒ Comma

☐ Semicolon ☐ Other

Quote Char: "

Number of title records: 1

Number Delimiters

Decimal Separator: .

Thousand separator: ,

This is a standard process as you'll have seen them many times over but, go to the variables page



	Variable Name	Variable Type	Sample Value	Contents	Form
1	V1	String	Helena	Other	
2	V2	String	Montana	Other	
3	V3	String	US	Other	
4	V4	String	West	Other	

And notice that the contents is set to 'Other' to prevent Architect from injecting generated code, which you subsequently are, in Architect at least, not allowed to change. We do this because we will start using the new, alternate hierarchies based, TI functions, which, Architect being pretty much oblivious to alternate hierarchies itself, doesn't know about.

In the prolog tab we'll, as usual, destroy any existing dimension and recreate it.

```

****Begin: Generated Statements****
****End: Generated Statements****

if (DimensionExists( 'Cities' ) > 0);
  DimensionDestroy( 'Cities' );
endif;

DimensionCreate( 'Cities' );

# In a world with alternate hierarchies one thing of a dimension of a collection of dimensions and that
# the next thing to do would be to create the first hierarchy. However TI, for old times sake supposedly,
# implicitly creates that 'same named' hierarchy already so we don't have to explicitly create it. Had we
# wanted a different name for the first hierarchy, which one typically would but which we'll refrain from
# here because Architect, at it's core is oblivious to hierarchies, can only show the same named one.
#HierarchyCreate( 'Cities', 'Cities' );

HierarchySortOrder( 'Cities', 'Cities', 'BYNAME', 'ASCENDING', 'BYHIERARCHY', "" );

```

Note that I commented out the Hierarchy create here and the comment that goes with it;-! You'll see that HierarchyCreate come back in the next process we'll run to create a second hierarchy.

In the metadata tab, we'll add the element to the hierarchy as one is used to, only this time also here we are starting to use the new, alternate hierarchies specific, TI functions. The old dimension specific functions are effectively obsolete, even if you only want one hierarchy you can use the hierarchy specific versions of the functions which will allow you to name the one and only hierarchy something else then the dimension name.



```

*****Begin: Generated Statements*****
*****End: Generated Statements*****

# insert leaf element
HierarchyElementInsert( 'Cities', 'Cities', '', V1, 'h' );

# insert consolidations
HierarchyElementInsert( 'Cities', 'Cities', '', V2, 'c' );
HierarchyElementInsert( 'Cities', 'Cities', '', V3, 'c' );
HierarchyElementInsert( 'Cities', 'Cities', '', V4, 'c' );

# connect components
HierarchyElementComponentAdd( 'Cities', 'Cities', V2, V1, 1.000000 );
HierarchyElementComponentAdd( 'Cities', 'Cities', V3, V2, 1.000000 );
HierarchyElementComponentAdd( 'Cities', 'Cities', V4, V3, 1.000000 );

```

Now let's run the process and have the 'Cities'.

Validate that the new dimension got created successfully using the REST API

Now what we have the new dimension, and earlier learned about the REST API which has had support for hierarchies since its inception, let's query for this new dimension and look at the hierarchies and element structure that we created by issuing the following request:

[https://tm1server:8010/api/v1/Dimensions\('Cities'\)?\\$select=Name&\\$expand=Hierarchies\(\\$count;\\$select=Name;\\$expand=Elements\(\\$select=Name;\\$filter=Parents/\\$count%20eq%200;\\$expand=Components\(\\$select=Name;\\$expand=Components\(\\$select=Name;\\$expand=Components\(\\$select=Name\)\)\)\)&\\$format=application/json;odata.metadata=none](https://tm1server:8010/api/v1/Dimensions('Cities')?$select=Name&$expand=Hierarchies($count;$select=Name;$expand=Elements($select=Name;$filter=Parents/$count%20eq%200;$expand=Components($select=Name;$expand=Components($select=Name;$expand=Components($select=Name))))&$format=application/json;odata.metadata=none)

Note that, to keep the response as clean as possible, I added the \$format query parameter to suppress all the e-tags and other metadata that would otherwise would have been returned.

Now let's add a second hierarchy to our newly created dimension

The second hierarchy we'll create is very much like the first one, the only difference being that instead of splitting cities across the eastern and western part of the globe we are now separating them by the northern and southern hemisphere. The data for this second hierarchy is in one of our CSV files, Cities2.csv in this case.

The CSV file contains all the cities we want to represent in our dimension and again has four columns, three of which are describing the geographical hierarchy.

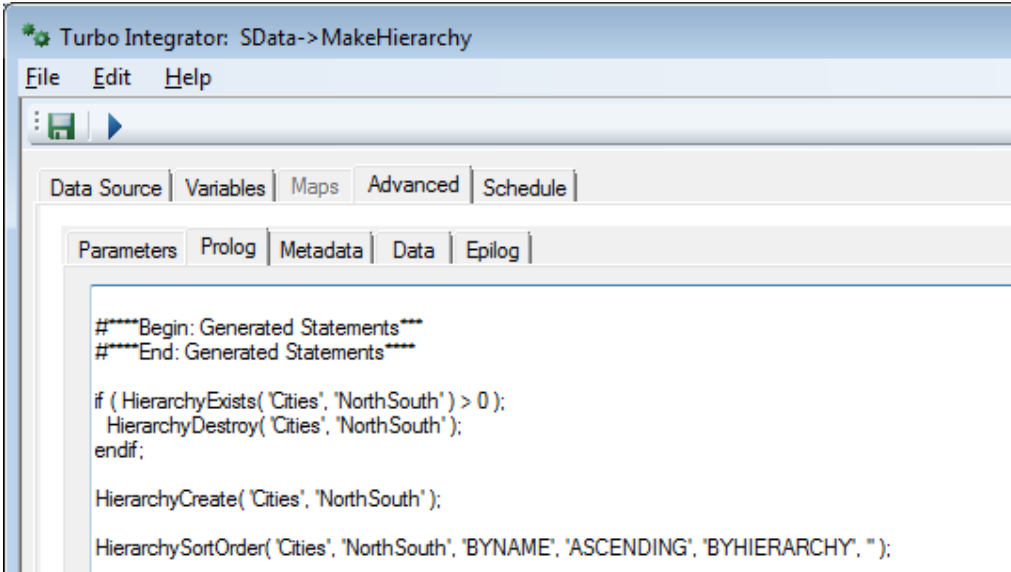
```

Helena, Montana, US, North
Redmond, California, US, North
Kulob, Khatlon, Tajikistan, North
...

```

Open the MakeHierarchy process in Architect and go to the prolog tab





```

Turbo Integrator: SData->MakeHierarchy
File Edit Help

Data Source Variables Maps Advanced Schedule

Parameters Prolog Metadata Data Epilog

****Begin: Generated Statements****
****End: Generated Statements****

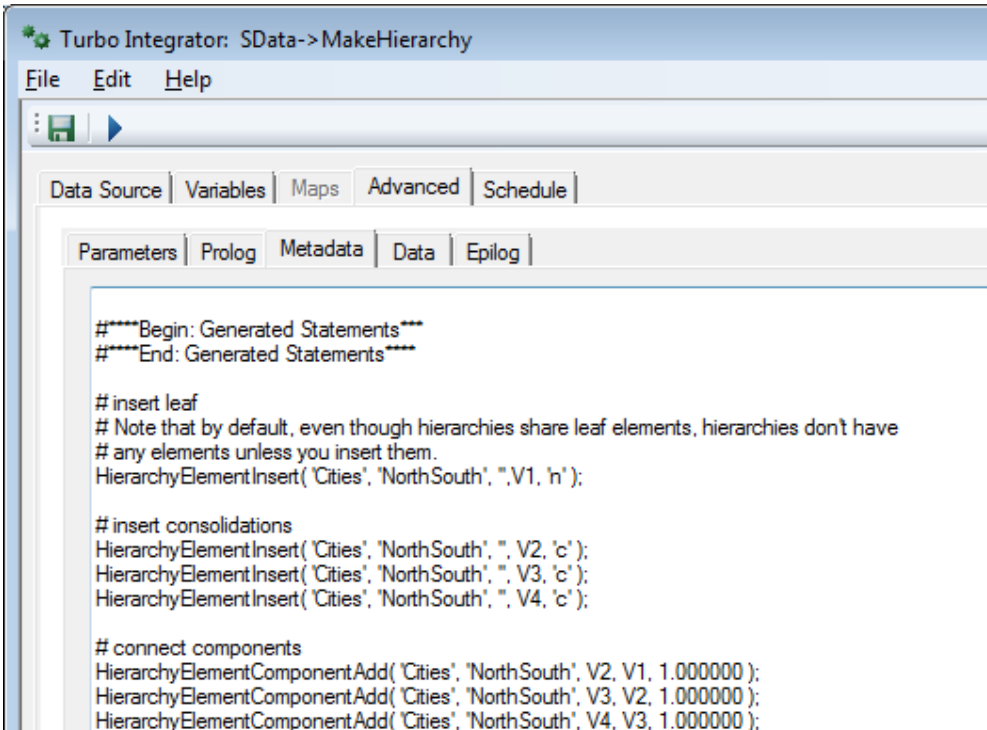
if ( HierarchyExists( 'Cities', 'NorthSouth' ) > 0 );
  HierarchyDestroy( 'Cities', 'NorthSouth' );
endif;

HierarchyCreate( 'Cities', 'NorthSouth' );

HierarchySortOrder( 'Cities', 'NorthSouth', 'BYNAME', 'ASCENDING', 'BYHIERARCHY', '' );

```

Notice that now we do create a new hierarchy, in the previously created 'Cities' dimension, named 'NorthSouth'. The metadata tab looks effectively the same as in our previous process, the only difference is effectively the hierarchy name in which we are creating the elements.



```

Turbo Integrator: SData->MakeHierarchy
File Edit Help

Data Source Variables Maps Advanced Schedule

Parameters Prolog Metadata Data Epilog

****Begin: Generated Statements****
****End: Generated Statements****

# insert leaf
# Note that by default, even though hierarchies share leaf elements, hierarchies don't have
# any elements unless you insert them.
HierarchyElementInsert( 'Cities', 'NorthSouth', 'V1', 'h' );

# insert consolidations
HierarchyElementInsert( 'Cities', 'NorthSouth', 'V2', 'c' );
HierarchyElementInsert( 'Cities', 'NorthSouth', 'V3', 'c' );
HierarchyElementInsert( 'Cities', 'NorthSouth', 'V4', 'c' );

# connect components
HierarchyElementComponentAdd( 'Cities', 'NorthSouth', V2, V1, 1.000000 );
HierarchyElementComponentAdd( 'Cities', 'NorthSouth', V3, V2, 1.000000 );
HierarchyElementComponentAdd( 'Cities', 'NorthSouth', V4, V3, 1.000000 );

```

Once again let's run the process and add the second hierarchy to our cities dimension.

Validate that we now have a dimension with two, correction, three hierarchies!

Once again, using the REST API, verify the structure of the dimension by issuing the same request we used earlier after we created the dimension initially. Here it is once again:

[https://tm1server:8010/api/v1/Dimensions\('Cities'\)?\\$select=Name&\\$expand=Hierarchies\(\\$count;\\$select=Name;\\$expand=Elements\(\\$select=Name;\\$filter=Parents/\\$count%20eq%200;\\$expand=Compo](https://tm1server:8010/api/v1/Dimensions('Cities')?$select=Name&$expand=Hierarchies($count;$select=Name;$expand=Elements($select=Name;$filter=Parents/$count%20eq%200;$expand=Compo)



[nents\(\\$select=Name;\\$expand=Components\(\\$select=Name;\\$expand=Components\(\\$select=Name\)\)\)\)&\\$format=application/json;odata.metadata=none](#)

Notice that you don't have two but rather three hierarchies now! Wonder where that third hierarchy came from? Well, remember that all hierarchies share the same set of leaves, but also that not every hierarchy needs to contain all the leaves? Well, after adding a second hierarchy the system injects a so called 'all leaves' hierarchy which the system maintains for you and it guarantees that it contains all the leaves used in any of the hierarchies in the dimension. The name of this hierarchy can be set by specifying the "AllLeavesHierarchyName" property of the dimension, the default name is "Leaves". Also notice that hierarchies have a "Visible" property. This is a hint, nothing more than that, to clients that, if they so choose to, they can hide that hierarchy from being shown to the users of such clients. I'd expect 'reporting' style clients to adhere to this property whereas a modeling tool would obviously ignore it but rather let you set that property instead. By default, the Visible property of the Leaves hierarchy is set to false! If you want it to show up everywhere you can simple switch that value to true.

Adding an 'attribute based' hierarchy

Notice that we've been talking about alternate hierarchies all along and, deliberately, not about 'attribute based' hierarchies. The primary reason for this is not that a hierarchy can't be generated, like dimensions can for years, based on attribute values, but because the system itself would be totally oblivious to the fact that there is a relationship between those attribute values and the hierarchy structure itself.

We therefore have reserved the term 'attribute based hierarchies' for now and plan on introducing real, attribute based, hierarchies support at some point in the future, at which point, the server will auto maintain the structure of a hierarchy, based on the referenced attributes value, given a specification for such hierarchy that also defines what to do with duplicate names, null values/raggedness, unbalanced hierarchies, level skipping etc. etc.

But that doesn't stop us from creating a hierarchy based on the values of attributes at a point in time using a TI process so let's do so.

Creating and populating the attributes

Using yet another CSV file, Countries.csv, the CreateRegionAtts TI process will create a couple of attributes in our, already existing, 'Region' dimension. The attribute data represents a hierarchical relationship. The values themselves represent 2015 import volumes of the individual countries, the first attribute being finer grained than the second one.

Open up the CreateRegionAtts TI process



And have a peek at the variables representing the columns in the CSV file

	Variable Name	Variable Type	Sample Value	Contents
1	Country	String	Argentina	Element
2	Imports2015	String	0-100B	Attribute
3	Imports2015_Top	String	Small	Attribute

And the mapping of those variables to text attributes

Attribute Variable	Sample Value	Dimension	Element Variable	Attribute	Action	Attribute Type
Imports2015	0-100B	region	Country	Imports2015	Create	Text
Imports2015_Top	Small	region	Country	Imports2015_Top	Create	Text

Now run this TI to have the two attributes created and added to the region dimension.

Creating a hierarchy based on attribute values

Now that we have attributes whose values represent a hierarchical structure for the members in the dimension we can use those attribute values to build such a hierarchy.

Instead of a CSV file however we'll use the }ElementAttributes cube associated to the dimension, the region dimension in this case, as the data source.

Note that if we wanted a hierarchy solely based on one attribute value we could use the CreateHierarchyByAttribute function, which does exactly the same, but is limited to one attribute.

This TI will create the Imports2015 hierarchy in the region dimension. But before we can do so you need to create a public view on our }ElementAttributes_region cube, let's call it "Imports2015", first, which view we will subsequently use as the source for our CreateTwoAttHierarchy TI process. The important bit: the view must have all leaf elements, and only leaf elements, for the region dimension and one, doesn't really matter which one, element from our }ElementAttributes_region dimension. Make sure that there is NO aliases in effect on the subset of elements on the region dimension. Why no aliases you wonder? Well, because we need the actual element name, not the alias, to set up the



connection to the actual leaf elements in the dimension and the elements and the consolidated elements that we'll be creating to represent the attribute values.

Open up the CreateTwoAttHierarchy process

Turbo Integrator: SData->CreateTwoAttHierarchy

File Edit Help

Data Source Variables Maps Advanced Schedule

Datasource Type

☐ ODBC

☐ Text

☐ ODBO

Cube

☒ IBM Cognos TM1

Cube View

Data Source Name: SData:}Element.Attributes_region->Imports2015

And have a peek at the variables

	Variable Name	Variable Type	Sample Value	Contents
1	region	String	Argentina	Element
2	Imports2015	String	Imports2015_Top	Element
3	Imports2015_Top	String	Small	Element

And the dimension mapping

Element Variable	Sample Value	Dimension	Order In Cube	Action	Element Type	Element Order
region	Argentina	region		As Is	Numeric	By Input
Imports2015	Imports2015_Top	}Element.Attributes_region		As Is	String	By Input
Imports2015_Top	Small	}Element.Attributes_region		As Is	String	By Input

In our prolog tab we once again see the new hierarchy being created after we have destroyed any previous version of the hierarchy, if such hierarchy already existed.

We also create the one and only root element, named 'All Imports2015' in this example, as the one and only root consolidating all imports in this hierarchy.



```

Turbo Integrator: SData->CreateTwoAttHierarchy
File Edit Help

Data Source Variables Maps Advanced Schedule

Parameters Prolog Metadata Data Epilog

#****Begin: Generated Statements****
#****End: Generated Statements****

if (HierarchyExists('region', 'Imports2015') > 0);
  HierarchyDestroy('region', 'Imports2015');
endif;

HierarchyCreate('region', 'Imports2015');
HierarchyElementInsert('region', 'Imports2015', "", 'All Imports2015', 'c');

HierarchySortOrder('region', 'Imports2015', 'BYNAME', 'ASCENDING', 'BYHIERARCHY', "");

```

In our metadata tab we use the attribute values, associated to the current region, to add the consolidated elements that build the structure.

```

Turbo Integrator: SData->CreateTwoAttHierarchy
File Edit Help

Data Source Variables Maps Advanced Schedule

Parameters Prolog Metadata Data Epilog

#****Begin: Generated Statements****
#****End: Generated Statements****

# insert leaf
HierarchyElementInsert('region', 'Imports2015', "", region, 'h');

# insert consolidations
Imports2015 = AttrS('region', region, 'Imports2015');
Imports2015_Top = AttrS('region', region, 'Imports2015_Top');
HierarchyElementInsert('region', 'Imports2015', "", Imports2015, 'c');
HierarchyElementInsert('region', 'Imports2015', "", Imports2015_Top, 'c');

# add edges
HierarchyElementComponentAdd('region', 'Imports2015', Imports2015, region, 1.0);
HierarchyElementComponentAdd('region', 'Imports2015', Imports2015_Top, Imports2015, 1.0);
HierarchyElementComponentAdd('region', 'Imports2015', 'All Imports2015', Imports2015_Top, 1.0);

```

Run the TI process and, if you don't trust us, verify, using the REST request you used twice before, only amending it to look at the 'region' dimension, that the region dimension now has two, oh no, three hierarchies as well!

[Add a rule with string data dependency and explore its behavior](#)

Now that we've added hierarchies to dimensions we might get into a situation where these hierarchies start playing a role when it comes to rule execution as well.



Let's add some hierarchy dependent rule to a new, numeric, leaf element, named "Prime" in our account1 dimension. To do so add such numeric leaf element, named "Prime" to the account1 dimension.

Next add this rule statement to the SalesCube cube:

```
# version 1 cube rule statement - uses AttrS
['Prime']=C:DB('ImportCtrl', AttrS('region', !region, 'Imports2015'), 'Units');
```

And add a feeder:

```
['Units']=>['Prime'];
```

And save the rule.

Now let's post the following, single hierarchy query, using Prime, to the server:

```
select
  { [region].[region].[level003].members } on rows,
  { [account1].[Prime] } on columns
from
  [SalesCube]
```

Note that [level003] of the region hierarchy in the region dimension represents the countries in this case. The cells in the result all have the attribute values retrieved using the AttrS in the rule.

Now let's put a consolidated element from the [region].[Imports2015] hierarchy, which we just created, into the where clause of the query, making it a multi hierarchy query:

```
select
  { [region].[region].[level003].members } on rows,
  { [account1].[Prime] } on columns
from
  [SalesCube]
where
  ( [region].[Imports2015].[1T+] )
```

You should still see prime numbers returned. AttrS understands how to handle multi hierarchy !dimension appearing as the element parameter:

```
AttrS( 'dimension', !dimension, 'attribute' )
```

is the same as:

```
ElementAttrS( 'dimension', 'dimension', !dimension, 'attribute' )
```

That is, AttrS operates on the same named hierarchy. The same named hierarchy is present in this query and AttrS extracts and uses it. It can do this because it knows the parameter represents an element, and it knows which hierarchy – we told it.

Now modify the rule statement so that it uses DB:

```
['Prime']=C:DB('ImportCtrl', DB('}ElementAttributes_region', !region,
'Imports2015'), 'Units');
```

Remove [region].[Imports2015] from the where clause and refresh the query:

```
// Version 1 query – single hierarchy
```



```

select
    { [region].[region].[level003].members } on rows,
    { [account1].[Prime] } on columns
from
    [SalesCube]

```

This still returns the same numbers.

But as soon as you put [region].[Imports2015].[1T+] back in the where clause, the rule statement will break. With AttrS you're specifying the hierarchy. With DB you'd need to hierarchy qualify in order to convey that information instead.

The next step is to make that change to the rule statement and use the hierarchy qualified current element operator with string-flavored DB expressions as in:

```

['Prime']=C:DB('ImportCtrl', DB('}ElementAttributes_region', !region:region,
'Imports2015'), 'Units');

```

This rule will work whether or not [region] is multi hierarchy in the query.



Building a 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 the Go programming language, that creates all the artifacts that make up your 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, just as any other service with an OData compliant RESTful API, into any application.

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 the only thing we need is a tm1s.cfg file containing the minimum set of configuration settings to stand up such server. In the NorthWind folder you'll find that tm1s.cfg containing the following configuration:

```
[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 that everybody that has TM1 have as well (read: don't really add much protection).

Now that we have a data folder and the configuration down we'll have to start our newly configured server. To do so we added a shortcut on the desktop named "TM1 NorthWind". Go ahead and double click it and start our new server, empty, TM1 server.

Note: If at any point in time, while working thru the later parts of this chapter, you need 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.

In this lab you'll be working with Go, a.k.a. Golang, which 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:\go-workspace'. 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\vision2017 repository therefore can be found here:

```
C:\go-workspace\src\github.com\hubert-heijkers\vision2017\src\builder
```

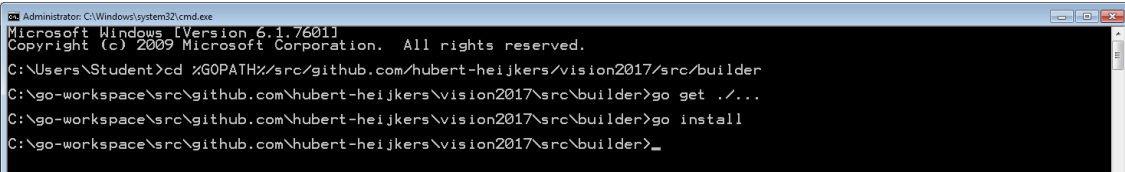
Whereas the 'builder' app, read: builder.exe, will end up being put into:

```
C:\go-workspace\bin
```

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

```
cd C:\go-workspace\src\github.com\hubert-heijkers\vision2017\src\builder or
```

```
cd %GOPATH%\src\github.com\hubert-heijkers\vision2017\src\builder
```



```
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Student>cd %GOPATH%\src\github.com\hubert-heijkers\vision2017\src\builder
C:\go-workspace\src\github.com\hubert-heijkers\vision2017\src\builder>go get ./...
C:\go-workspace\src\github.com\hubert-heijkers\vision2017\src\builder>go install
C:\go-workspace\src\github.com\hubert-heijkers\vision2017\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 do 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:\go-workspace\bin. It contains 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.

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.

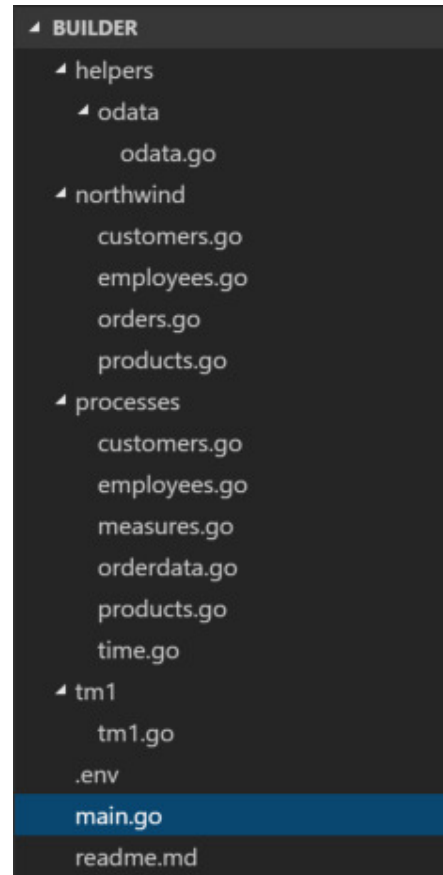
NorthWind package:

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](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. Products and customer dimensions each have one dimension while for the Employees dimension we create both a hierarchy based on geography as well as by age generation. 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\(\\$select=ProductID,ProductName;\\$orderby=ProductName\)](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](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,BirthDate&\\$orderby=Country%20asc,Region%20asc,City%20asc](http://services.odata.org/V4/Northwind/Northwind.svc/Employees?$select=EmployeeID,LastName,FirstName,TitleOfCourtesy,City,Region,Country,BirthDate&$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 date, both ascending and descending, and then only asking for the first order to be returned. Using the order date from these two orders we know the date range for which it subsequently creates a time dimension with one hierarchy containing years, quarters, months and days and additional hierarchies for years, quarters and months. 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](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](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 to you as snippets that need to go into the main.go file. Note that all the snippets contain the comments from the provided 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 Code, 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 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 dimensions spanning the cube and the rules that need to be set on the cube.

First, we'll create a collection of dimension IDs, read: OData IDs, that we'll need to pass to the cube definition. We'll do so by simply converting the dimensions array to a string array with these IDs.

```

// Build array of dimension ids representing the dimensions making up the cube
dimensionIds := make([]string, len(dimensions))
for i, dim := range dimensions {
    dimensionIds[i] = "Dimensions('" + dim.Name + "')"
}

```

Now we'll pass these IDs 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(tm1.ServiceRootURL+"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. Note 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](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:

11.0.00200.989

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, one could envision using this value to validate a minimal version required or even, as a shortcut instead of evaluating the \$metadata document as one should, make some choices 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 authentication header, in this case we are using authentication mode 1 which translates into basic HTTP authentication, execute the request and, after checking we got a 200 – OK status, dump the content of the response, the server version, to the console:

```
// Validate that the TM1 server is accessible 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 dimension definition which we'll need to pass to the createCube function later. So, lets create an array of dimensions to store the dimension we create.

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 dimensions [5]*tm1.Dimension
dimensions[0] = createDimension(proc.GenerateProductDimension(client,
datasourceServiceRootURL, productDimensionName))
dimensions[1] = createDimension(proc.GenerateCustomerDimension(client,
datasourceServiceRootURL, customerDimensionName))
dimensions[2] = createDimension(proc.GenerateEmployeeDimension(client,
datasourceServiceRootURL, employeeDimensionName))
dimensions[3] = createDimension(proc.GenerateTimeDimension(client,
datasourceServiceRootURL, timeDimensionName))

```



```
dimensions[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 dimensions representing the dimensions spanning the cube, and a set of rules to be used by the cube. The set of dimensions 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. 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, dimensions[:],
"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. Needless to say that 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, dimensions[0], dimensions[1], dimensions[2], dimensions[3],
dimensions[4])
```

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 another console window and go to the `bin` folder with the binaries by typing:

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

which should take you to `C:\go-workspace\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
```



>> Done!

Having a look at the results

[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;\\$expand=Children\(\\$select=Name\)\)\)\)\)\)&\\$format=application/json;odata.metadata=None](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;$expand=Children($select=Name))))))&$format=application/json;odata.metadata=None)

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. Therefore we recursively expand enough times to cover the maximum depth used across all dimensions in the model.

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

File Edit View Options Help

[Base]

All Revenue All

Time

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 %GOPATH%/src/github.com/hubert-heijkers/vision2017/output/builder folder. Feel free to copy that version over to the %GOPATH%/src/github.com/hubert-heijkers/vision2017/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 vision2017 %GOPATH%/src/github.com/hubert-heijkers/vision2017/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'](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:\go-workspace\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. An example you can try and play with is available on github here:

<https://github.com/Hubert-Heijkers/tm1-log-tracker>. Feel free to have a go at it and customize it to your own liking.



We Value Your Feedback!

- Don't forget to submit your IBM Analytics University session and speaker feedback! Your feedback is very important to us – we use it to continually improve the conference.
- Access the IBM Analytics University Conference Connect tool to quickly submit your surveys from your smartphone, laptop or conference kiosk.



Acknowledgements and Disclaimers

Copyright © 2017 by International Business Machines Corporation (IBM). No part of this document may be reproduced or transmitted in any form without written permission from IBM.

U.S. Government Users Restricted Rights — use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM.

Information in these presentations (including information relating to products that have not yet been announced by IBM) has been reviewed for accuracy as of the date of initial publication and could include unintentional technical or typographical errors. IBM shall have no responsibility to update this information. **This document is distributed “as is” without any warranty, either express or implied. In no event shall IBM be liable for any damage arising from the use of this information, including but not limited to, loss of data, business interruption, loss of profit or loss of opportunity.** IBM products and services are warranted according to the terms and conditions of the agreements under which they are provided.

IBM products are manufactured from new parts or new and used parts.

In some cases, a product may not be new and may have been previously installed. Regardless, our warranty terms apply.”

Any statements regarding IBM's future direction, intent or product plans are subject to change or withdrawal without notice.

Performance data contained herein was generally obtained in a controlled, isolated environments. Customer examples are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual performance, cost, savings or other results in other operating environments may vary.

References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business.

Workshops, sessions and associated materials may have been prepared by independent session speakers, and do not necessarily reflect the views of IBM. All materials and discussions are provided for informational purposes only, and are neither intended to, nor shall constitute legal or other guidance or advice to any individual participant or their specific situation.

It is the customer's responsibility to insure its own compliance with legal requirements and to obtain advice of competent legal counsel as to the identification and interpretation of any relevant laws and regulatory requirements that may affect the customer's business and any actions the customer may need to take to comply with such laws. IBM does not provide legal advice or represent or warrant that its services or products will ensure that the customer is in compliance with any law.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products. IBM does not warrant the quality of any third-party products, or the ability of any such third-party products to interoperate with IBM's products. **IBM expressly disclaims all warranties, expressed or implied, including but not limited to, the implied warranties of merchantability and fitness for a particular, purpose.**

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents, copyrights, trademarks or other intellectual property right.

IBM, the IBM logo, ibm.com, Aspera®, Bluemix, Blueworks Live, CICS, Clearcase, Cognos®, DOORS®, Emptoris®, Enterprise Document Management System™, FASP®, FileNet®, Global Business Services®, Global Technology Services®, IBM ExperienceOne™, IBM SmartCloud®, IBM Social Business®, Information on Demand, ILOG, Maximo®, MQIntegrator®, MQSeries®, Netcool®, OMEGAMON, OpenPower, PureAnalytics™, PureApplication®, pureCluster™, PureCoverage®, PureData®, PureExperience®, PureFlex®, pureQuery®, pureScale®, PureSystems®, QRadar®, Rational®, Rhapsody®, Smarter Commerce®, SoDA, SPSS, Sterling Commerce®, StoredIQ, Tealeaf®, Tivoli® Trusteer®, Unica®, urban{code}®, Watson, WebSphere®, Worklight®, X-Force® and System z® Z/OS, are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at: www.ibm.com/legal/copytrade.shtml.

