

Dynamic SELECT and UPDATE statements with MuleSoft DataWeave

Why Read This Blog Post? (What problem are we solving)

There are at least three problems MuleSoft developers face when implementing a REST API where the data is backed by a relational database:

1. If any resources in the API have multiple optional query parameters that dictate a SQL query, the permutations for the WHERE clause become exponential and unmanageable.
2. Programmatically building SQL queries from dynamic input such as HTTP query parameters risks SQL injections, unless you have the proper code.
3. REST APIs should not leak implementation details, such as table or column names in a database.

This blog provides MuleSoft developers with the tools they need, including code, to create and implement APIs with multiple optional query parameters that drive SQL queries without risking SQL injection or leaking implementation details.

Objectives

When you have completed reading this blog, you should be able to:

1. Create RAML that has multiple dynamic query parameters for GET methods and various data models (types) for GET, PUT/POST, and PATCH methods.
2. Create a project based on that API (you did this in the Fundamentals and other courses)
3. How to modify the scaffolded code to:
 - a. Comply with the HTTP specification to ignore request bodies on GET requests
 - b. Create dynamic objects for GET and PATCH operations, using DataWeave, saving those objects in event variables such as `dynamicWhere` and `dynamicUpdate`.
4. How to write SQL statements using the DB module that use the dynamic objects created in Step 3.b.

Solution Overview

At a high level, to design and implement a REST API as discussed in the previous section, you need:

1. An API definition that sticks to the REST principal that each resource represents an object or collection thereof, (for example: `/cars` and `/cars/{carID}`, `/flights` and `/flights/{ID}`) and may also include relational resources between objects (for example, `/flights/{flightID}/passengers`) where the related resource (in this case `passengers`) also represents a collection of objects, an object, or a property thereof.
2. A database with a 1-to-1 relationship between resources and tables. Following the example above, the database should contain a table for cars, another for flights, another for passengers.

3. Query parameters for GET requests on a resource in the API definition that correlate to at most one column in the table associated with the resource.
4. An object model (that is, a **type** definition in RAML) that has one column in the associated table for each property in the object model. (For example, suppose the *flight* object has properties: *ID*, *number*, *aircraftID*, and *takeoffDateTime*. There must be a table that has columns for each of these properties). IMPORTANT, the name of the resource does not, and probably should not, be the same as the name of the table. Similarly, the names of the columns in said table do not, and probably should not, be the same as the names of the properties. Having those names the same leaks internal implementation details.
5. The mappings of resource names to table names, and with resources/tables, mapping between query parameters and mapping between object properties and column names.
6. A Mule application development project with an *interface.xml* file scaffolded from the API.

After creating the database table, API, and Mule project, in your code, you will:

1. Create a file with a JSON object with a map between resources, tables, query parameters and columns
2. Create a file with a JSON object with a map between resources, tables, object properties and columns
3. Create a file with the DataWeave provided in this blog post.
4. Add a Transforms Message processors at the beginning of each flow in your Interface.xml file that is for a GET or PATCH method:
 - a. In the GET transform, import the DataWeave file (Step 3) and invoke **makeDynamicWhere** passing the query parameters and the name of the resource on which the method was invoked.
 - b. In the GET transform, import the DataWeave file (Step 3) and invoke **makeDynamicUpdate** passing the query parameters and the name of the resource on which the method was invoked.
5. For each DB:SELECT processor, use the object created in Step 4 to craft the query.

Your job is done. These two functions do all the work for you. For details on how they work, read on. If you just want to see the DataWeave, skip on down to here.

A Complete Example Project

The following sections provide details of the entire project, including the database table schemas, the RAML API specification, and the Mule Design project. Specifically, this project shows how to create a complete project with dynamic query parameters and dynamic SQL statements for a project with resources of *people* and their *jobs*.

You can find the complete project on [GitHub here](#). It contains the completed Mule Design project plus a database dump. See the README.md file to see how to use the project.

The Database Schema

The database that this project will query has the following three tables:

- people whose columns are:

Column Name	#	Data Type	Not Null	Auto Increment
ABC ID	1	varchar(100)	[v]	[]
ABC firstName	2	varchar(100)	[]	[]
ABC lastName	3	varchar(100)	[]	[]
ABC address1	4	varchar(100)	[]	[]

- jobs whose columns are:

Column Name	#	Data Type	Not Null	Auto Increment
ABC ID	1	varchar(100)	[v]	[]
ABC companyName	2	varchar(100)	[v]	[]
ABC companyAddress	3	varchar(100)	[]	[]
ABC Title	4	varchar(100)	[]	[]
ABC workAddress	5	varchar(100)	[]	[]
123 ratePerHour	6	double	[]	[]

- personJob that provides N:M mapping of people to jobs and whose columns are:

Column Name	#	Data Type	Not Null	Auto Increment
ABC Jid	2	varchar(100)	[v]	[]
ABC pID	1	varchar(100)	[v]	[]

The RAML

We need a root RAML file with our resources, libraries with separate data models for input, output, and patching the resources, and ideally resourceTypes for collections and member resources. For details on working smarter with libraries and resourceTypes, see [this blog](#).

The ROOT RAML

Here is a simplified version of our root RAML:

```

##RAML 1.0
title: example
mediaType:
  - application/json

uses:
  person: libraries/personLib.raml
  job: libraries/jobLib.raml

/people:
  description: actions on a collection of **people**
  # shortened RAML on the collection to demonstrate basics
  post:
    body:
      type: person.personIn
  get:
    queryParameters:
      # one query parameter for each property on which we allow query.
      # all are optional
      fname?:
      lname?:
      address?:
    responses:
      200:
        description: successfully retrieved some set of people
        body:
          type: person.personOut[]
/people/{personID}:
  # shortened RAML on member to demonstrate RAML basics
  get:
    responses:
      200:
        body:
          type: person.personOut
  put:
    body:
      type: person.personIn
  patch:
    body:
      type: person.personPatch
/jobs:
  description: jobs held by the selected person
  get:
    queryParameters:
      ratePerHour?:
        type: number
        format: double
      title?: string
      companyName?: string
/jobs:
  post:
  get:
    queryParameters:
      ratePerHour?: number
      title?: string
      companyName: String

```

The RAML libraries

We need two libraries, one that describes objects for people (a person), and one for jobs. Each library needs a separate object model (type) for input (POST/PUT), output (GET), and update (PATCH) operations.

Here is the library describing jobs:

##RAML 1.0 Library

usage: contains objects for CRUD operations on **job** objects

types:

jobIn:

properties:

companyName: string

companyAddress: string

title: string

workAddress: string

ratePerHour: number

jobOut:

type: jobIn

properties:

ID: string

jobPatch:

properties:

companyName?: string

companyAddress?: string

title?: string

workAddress?: string

ratePerHour?: string

Here is the library describing person(s) (people):

```

#%RAML 1.0 Library
usage: contains objects for CRUD operations on **person** objects

uses:
  job: jobLib.raml
types:
  personIn:
    properties:
      firstName: string
      lastName: string
      address1: string
      jobs: job.jobIn[]

  personOut:
    type: personIn
    properties:
      ID: string
      jobs: job.jobOut[]

  personPatch:
    type: object
    properties:
      firstName?: string
      lastName?: string
      address1?: string

```

The key take-aways from this RAML are that:

1. The API has GET on /people that has query parameters: fname, lname, and address, to query only for people that have the specified first / last names and/or address.
2. The API has PATCH on /people/{personID} to enable modification of a person's first / last names and address.

NOTE1: the API is abbreviated for this blog. A full API definition would include, at the least:

- A set of results with body, etc. for the GET on /people/{personID}/jobs
- Resources for /jobs/{jobID} with GET / PUT / PATCH methods similar to those operations on /people/{personID}
- A set of resourceTypes to shorten the root RAML. They are left out for clarity.

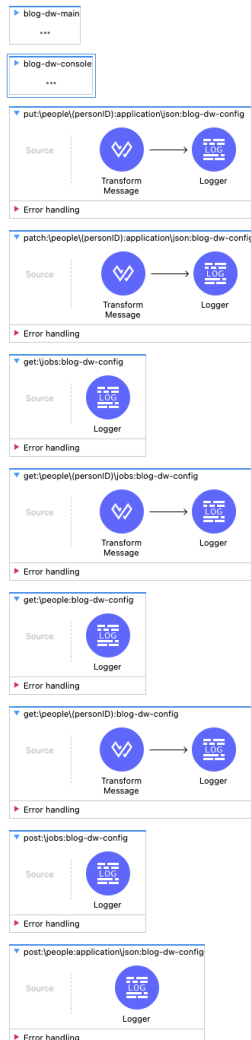
NOTE2: you can see the API definition included in the Mule Design project, imported as an asset from Exchange.

The Interface File

This section shows the code in the *interface* file. Recall from the Mule 4 Fundamentals course that the interface file is created from scaffolding an API. It starts out with:

- A *main* flow with an HTTP(s) listener and an APIKit Router.

- A *console* flow with an HTTP(s) listener and an APIKit Console Router
- One flow per *endpoint* (method : resource pair) in the API specification. Our API has eight (8) endpoints, so there are eight (8) flows beyond the *main* and *console* flows. It starts out, graphically, looking like the following:



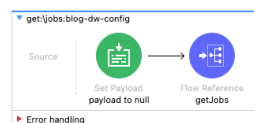
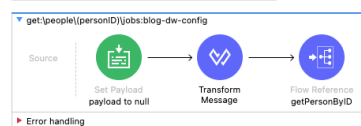
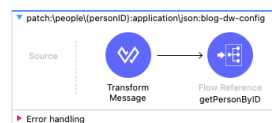
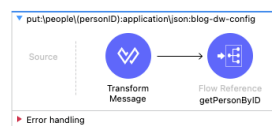
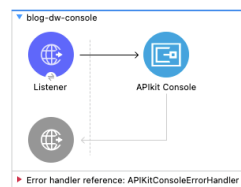
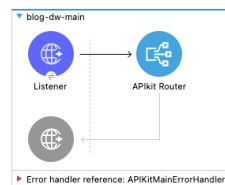
NOTE: The flows are in ordered first alphabetically by method, and then by resource. That is, it has all of the GET methods before the POST before the PUT etc.

Standard cleanup and rearrangement of the *interface.xml* file

Whenever I scaffold an API, there are several things I do in order to *clean things up*, including:

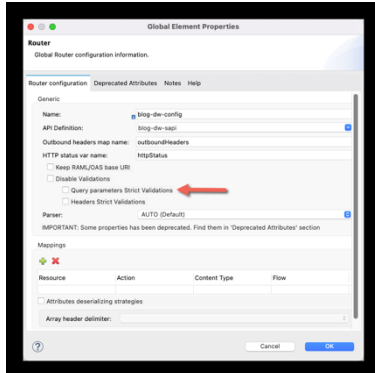
1. In order to comply with the HTTP protocol specification, where all GET methods should ignore any incoming *body* send by the client, to all flows for GET methods, add a **set payload** processor at the front of said flows, setting payload to null.
2. I move the error handling from the *main* and *console* flows to the file *errors.xml* which I create just to hold these and other error handlers.
3. Finally, I reorganize the order of the flows in the *interface* file to match the order in which they are declared in the API.

The resulting *interface* file looks like this:



Configuring Query Parameters Strict Validations

The DataWeave functions presented later in this blog do not (currently) handle the condition that you provide a query parameter that doesn't have a mapping. You can fix that with appropriate application of **if/else** or **try** expressions. For simplicity, in this project, simply edit the configuration of the APIKit Router, displayed here:



Ensure you check **Query parameters Strict Validations** to which the arrow points. This causes the APIKit Router to reject (as a bad request) any requests with query parameters not defined in the API.

The Mapping JSON files

The Dataweave functions (shown and discussed in the next section) for creating dynamic WHERE clauses and dynamic UPDATE clauses each require a JSON object that, for each resource in the API (the example has *people* and *jobs* as resources):

- The name of the table containing the collection of resources (records)
- For WHERE clauses, the mapping of query parameter names to column names
- For UPDATE clauses, the mapping of object properties to column names.

The following two subsections contain the maps required in our example.

IMPORTANT: both of these files must be placed in the `.../src/main/resources` folder of your Mule Design project.

The *qpsColumnMap.json* file

This file maps query parameters to column names. The format is:

```
{
  "resource": {
    "table": <tablename>,
    "queryParams": {
      "queryParam1": "columnForQueryParam1",
      "queryParam2": "columnForQueryParam2" [, ...]
    }
  }
}
```

Here is the mapping for this API and its Database:

```
{
  "people": {
    "table": "people",
    "queryParams": {
      "fname": "firstName",
      "lname": "lastName",
      "address": "address1"
    }
  },
  "jobs": {
    "table": "jobs",
    "queryParams": {
      "ratePerHour": "ratePerHour",
      "companyName": "companyName",
      "jobTitle": "title"
    }
  }
}
```

NOTES:

- The resources and table names happen to be the same. This is understandable. You may change the table name if this bothers you.
- In the jobs table, the query parameter name happens to be the same as the column name. You may change the column name or query parameter name if this bothers you.

The *objectPropsColumnMap.json* file

This file maps the objects in a PATCH version of an object (see the earlier discussion on libraries), specifically the properties thereof, to columns in the corresponding table. The format for this object is almost the same as that for the *qpsColumnMap.json* file, except:

- Use the word “props” instead of “queryParams” in the pier property to “table”
- The left-side of the map is the property name instead of the query parameter name. The right-side is still the column name.

Here is the *objectPropsColumnMap.json* file for this example project:

```
{
  "people": {
    "table": "people",
    "props": {
      "firstName": "firstName",
      "lastName": "lastName",
      "address1": "address1"
    }
  },
  "jobs": {
    "table": "jobs",
    "props": {
      "companyName": "companyName",
      "companyAddress": "companyAddress",
      "title": "title",
      "workAddress": "workAddress",
      "ratePerHour": "ratePerHour"
    }
  }
}
```

NOTES:

- In this example, the table names are the same as the resources and property names the same as the column names. This may or may not be the case and often is forced not to be the case in the name of security, not leaking internal information. You can see from the previous example, and this one, that they need not be the same, which is why this object exists. In your own code, make them different to hide implementation details.
- Only list in this object the properties that are defined in the PATCH version of your object.

The DataWeave

Now, here is the DataWeave that creates dynamic WHERE clauses and UPDATE statements for dynamic SQL for GET (collection) and PATCH (member) API endpoints. Because of its size, we need to split the code across two pages. First, here are the helper functions:

```
%dw 2.0
import * from dw::core::Strings

// this file must contain ONE JSON object with keys of query parameters and
// values of the corresponding column name.
var mappingObject = readUrl("classpath://objectPropsColumnMap.json", "application/json")
var mappingQPs = readUrl("classpath://qpsColumnMap.json", "application/json")

// Turn obj values into what we need for the DB
// for now, this is a noop. Might change later.
// Originally thought strings needed to be quoted. Not now.
var makeValue = (val) ->
    //if (isNumeric(val)) val as Number else val
    val

// Create both the dynamic where clause with input parameters,
// and the inputParameter object.
var createSetClause = (acc: Object, objectProp: Object | Null | String, objName: String) ->
    if(objectProp is Object)
        do {
            var objInfo = mappingObject[objName]
            var propColumnMap = mappingObject[objName].props
            var colName = propColumnMap[namesOf(objectProp)[0]]
            ---
            if (colName == null) // there is no match, just return current acc
                acc
            else
                if(acc.set == '') { // first time through
                    set: 'SET $(colName) = :$(colName)',
                    params: { (colName): makeValue(valuesOf(objectProp)[0]) }
                }
                else { // second+ time through, add
                    set: "$(acc.set), $(colName) = :$(colName)",
                    params: acc.params ++
                        { (namesOf(objectProp)[0]): makeValue(valuesOf(objectProp)[0]) }
                }
        } // do
    else { set: acc.set, params: acc.params }

// Create both the dynamic where clause with input parameters,
// and the inputParameter object.
var createWhereClause = (acc: Object, qpo: Object | Null | String, objName: String) ->
    if(qpo is Object)
        do {
            var objInfo = mappingQPs[objName]
            var qpColumnMap = mappingQPs[objName].queryParams
            var colName = qpColumnMap[namesOf(qpo)[0]]
            ---
            if (colName == null) // there is no match, just return current acc
                acc
            else
                if(acc.where == '') {
                    where: 'WHERE $(colName) = :$(colName)',
                    params: { (colName): makeValue(valuesOf(qpo)[0]) } }
                else {
                    where: "$(acc.where) AND $(colName) = :$(namesOf(qpo)[0])",
                    params: acc.params ++
                        { namesOf(qpo)[0]: makeValue(valuesOf(qpo)[0]) }
                }
        } // do
    else { where: "", params: "" }

//var processObject =
//  1. turn queryParams or some other object into an array of key-value pairs
//  that can then be processed by reduce
//  2. use 'reduce' to iterate over each query parameter
//  and call createSetClause or createWhereClause to create:
//  a. a SET or WHERE clause with parameterized inputs
//  b. an inputParameter object with key/value pairs for the parameters in 2.a.
var processObject = (anObject: Object, objName: String, clause: String) ->
    // turn updateObject object into an array of key-value pairs
    // that can then be processed by reduce
    (anObject pluck (v, k) ->
        (k): v)
    // use 'reduce' to iterate over each query parameter
    // and create the Where-clause string and input parameter object
    reduce (prop, acc={set: '', where: '', params: ''}) ->
        clause match {
            case 'SET' -> createSetClause(acc, prop, objName)
            case 'WHERE' -> createWhereClause(acc, prop, objName)
```

Here is the rest of the `dynamicQueryFunctions.dwl` file, containing the two main functions:

```
var makeDynamicWhere = (queryParams: Object, objName: String) ->
do {
  var dynamic = processObject(queryParams, objName, 'WHERE')
  ---
  if (dynamic.params == '') { where: "", params: "" }
  else dynamic
} // do

var makeDynamicUpdate = (updateObject: Object, objName: String) ->
do {
  var dynamic = processObject(updateObject, objName, 'SET')
  ---
  if (dynamic.params == '') { set: "", params: "",
    start: "UPDATE $(mappingObject[objName].table as String) " }
  else dynamic ++
    "start": "UPDATE $(mappingObject[objName].table as String)" }
```

Explanation of the Dataweave

The two main functions: *`makeDynamicWhere()`* and *`makeDynamicUpdate()`*

The two main functions are **`makeDynamicWhere()`** and **`makeDynamicUpdate()`**.

Use **`makeDynamicWhere()`** to create objects necessary to do dynamic SQL SELECT statements from optional query parameters such as those associated with a GET method.

Use **`makeDynamicUpdate()`** to create objects necessary to create a dynamic SQL UPDATE statement from an object with optional properties such as one associated with a PATCH method.

Here is what the functions do:

1. Both methods take as their first parameter an object that is either the query parameters for a GET or the object in the body of a PATCH. As a second parameter, they both take the name of the resource on which the method was invoked (in our case 'people' or 'jobs').
2. Both methods create an object assigned to variable *dynamic* that has several properties that get modified during call to *processObject()* (*discussed next*).
 - a. The *makeDynamicWhere*'s object has *where* and *params* properties.
 - i. *where* becomes a string with a dynamically created SQL WHERE clause, with parameterized inputs, for example, "WHERE xx = :yy" where xx is the name of a column associated with a query parameter (see the map object discussed previously), and yy is a placeholder for the associated query parameter's value.
 - ii. *params* becomes an object with keys matching the placeholders (e.g. "yy" in the example) and the placeholder's value in key: value pairs, for example: { yy: "value1" }.
 - b. The *makeDynamicUpdate*'s object has three properties: **set**, **params**, and **start**:
 - i. Set becomes a string analogous to the *where* property described above, with a list of fields to be set based on the properties present in the body of the PATCH request. It uses property placeholders just as the WHERE clause discussed in the preceding object. For example, this can become: "SET xx = :yy" where xx is the column name associated with a property in the PATCH object, and :yy is a placeholder for the associated value.

- ii. Params becomes an object with the property placeholders as keys and their associated values as values, just like in the previous object. For example, "{ yy: '1234' }"
 - iii. Start is a string that starts the UPDATE SQL statement with the table name associated with the resource (see the previous object mapping discussion).
3. Both functions call **processObject()**, passing their own two parameters, and then either "WHERE" or "SET" as a third parameters.

The processObject() function

At a high level, this function processes all of the keys in the first parameter (*anObject*) as an array of key/value pairs to create either a dynamic SET or dynamic WHERE clause using **createSetClause()** and **createWhereClause()** respectively. Specifically, it:

1. Uses **pluck()** to turn the key/value pairs of *anObject* into an array of key/value pairs.
2. It calls **reduce()** on the array created by **pluck()** with: A) the current element of the array remembered in the *prop* variable, and an accumulator that contains the previously described object with keys of *set*, *where*, and *params*.
3. It calls **createSetClause()** if the 3rd parameter (*clause*) is "SET" (create structures for a PATCH method) or **createWhereClause()** if the 3rd parameter (*clause*) is "WHERE" (create structures for a GET method).

The create[Set,Where]Clause() functions

These functions essentially do the same thing, but one creates the structure for GET methods and one for PATCH endpoints. They update different parts of the accumulator object (set vs where).

1. Both functions start with an if/else that validates that the 2nd parameter (*objectProp* or *qpo*) is an object.
 - a. If not, it returns the relevant parts of the current accumulator.
2. If it is an object, the function:
 - a. Looks up the object information in the appropriate map (*mappingObject* for **createSetClause** and *qpo* for **createWhereClause**). It stores this in *objInfo*.
 - b. Looks up the appropriate map for the object property (**createSetClause**) or query parameter (**createWhereClause**). It stores this in *propColumnMap* and *qpColumnMap* respectively.
 - c. Looks up the column from the map set in the preceding step. It remembers this in *colName*.
3. If *colName* is not null (the column lookup succeeded), each of these initialize or add to the existing accumulator object:
 - a. For **createSetClause**, the two properties of the accumulator are *set* and *params*. If the value of *set* is an empty string, then the code initializes *set* with this line of code:

```
set: "SET ${colName} = :${colName}"
```

NOTE: `$(xx)` inside a string imbeds the value of `xx` into the string.

It also sets the value of the *params* property of the accumulator to the name of the column as the key (the column name was used to create the placeholder),

and then the value set in the property of the object passed into the PATCH method.

At the end of the first time through this method, the accumulator object has one item in the SET clause that is parameterized, and one input parameter by the column name and its new value.

- b. If the value of *set* is not empty, it adds a comma and another column name / column-placeholder with this line of code:

```
set: "${acc.set}, ${colName} = :${colName}"
```

Thus, if *set* had the value "SET xx = :xx" and the new property is "yy", *set* gets the new value "SET xx = :xx, yy = :yy".

It also adds the property placeholder for the new property to the accumulator's *params* object.

- c. For *createWhereClause()*, the function performs the same logic, except it modifies the *where* property in the *accumulator* object (instead of the *set* property), using the string "AND" between query parameters / properties instead of commas.

NOTE1: Both of these methods call *makeValue()* to turn the new value into an appropriate form. For the moment, this method just returns the object passed, so it can be safely ignored, and even removed if you'd like.

NOTE2: It is entirely possible to combine the logic for these two functions into one function, by passing the "SET" or "WHERE" parameter from *processObject()* so that this function knew whether to create a *set* or *where* property in the *accumulator* object. We leave that to you, the reader, as an exercise.

Putting the DataWeave to work in the *interface.xml* file.

Recall from the Production Ready Integrations course (it may have also been discussed in your Fundamentals and Production Ready Development Practices (PRDP) courses): All code dealing with having HTTP as your interface should be handled in your *interface.xml* file, and not in your implementation file. (NOTE: The Fundamentals and PRDP courses don't actually do this, though it may be discussed). This code should not go into the *implementation.xml* code.

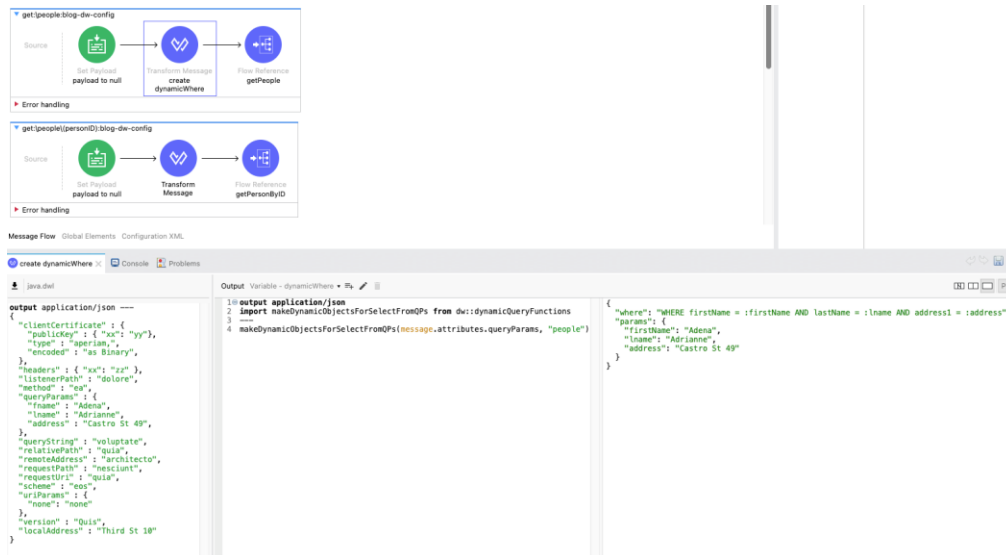
In each flow in your *interface.xml* file for a GET with query parameters, or PATCH, add a transform message that invokes the appropriate function, **makeDynamic{Where,Update}**, respectively. Pass `message.attributes.queryParams` and the resource, or `message.payload` and the resource, to the functions, respectively, saving the results in a variable, e.g. **dynamicWhere** or **dynamicUpdate**. For example:

- For `get:\people:.....` in a new transform, put this code:

```
Output application/java
Import makeDynamicWhere from dw::dynamicQueryFunctions.xml
---
makeDynamicWhere(message.attributes.queryParams, "people")
```
- For `patch:\people\((personID):...` in the new transform, put this code:

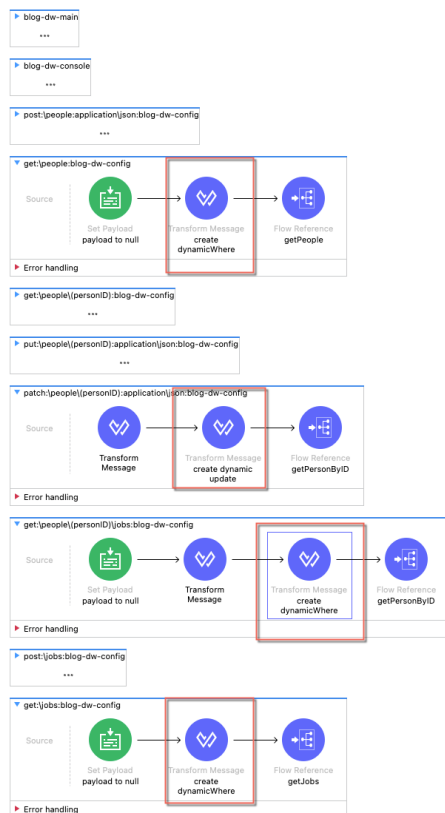
```
Output application/java
Import makeDynamicUpdate from dw::dynamicQueryFunctions.xml
---
makeDynamicUpdate(message.payload, "people")
```

This is an example of the transform message for `get:\people` (NOTE: I added sample attributes):



You must add similar transform message components in the `get:\jobs` and `patch:\jobs\{jobID}` flows in the `interface.xml` file.

Here is a view of the `interface.xml` file with all but the modified flows collapsed:



NOTE: The added transformations are outlined for emphasis.

After creating these transforms, ensure you propagate metadata through the flow references to the called flows. Propagating the meta-data allows you to see said meta-data in the called flows, i.e. in your implementatin flows.

Using the Dynamic Objects in your SQL statements

With the API done, the scaffolding done, the DataWeave and mapping objects done, the modifications made to the scaffolding code as discussed in the immediately preceding section, you are finally ready to put all this work to use in your flows that implement your endpoints, especially in the DB connector processors.

To use all this work, all you need to do is to add DB processors in your implementation flows that use the *vars.dynamicWhere* and *vars.dynamicSet* objects created in the Transforms you created in the applicable flows in your *interface.xml* file.

To create a dynamic SELECT statement for collections, write code that looks like this (in XML):

```
<db:select doc:name="Select" doc:id="bb38a8b3-533b-43eb-b912-79db2683797a" config-ref="Database_Config">
  <db:sql ><![CDATA[#['select * from people $(vars.dynamicWhere.where)' ]]]></db:sql>
  <db:input-parameters >
    <![CDATA[#['if (isEmpty(vars.dynamicWhere.params)) {} else vars.dynamicWhere.params]]]>
  </db:input-parameters>
</db:select>
```

NOTES:

- The input-parameters use an if/else
 - If there are no parameters, use an empty object
 - If there are parameters, use those
- The SELECT statement uses string interpolation to include *vars.dynamicWhere.where*, assuming the variable *vars.dynamicWhere* was set in the get:\people... flow in the *interface.xml* file.

To create a dynamic PATCH statement for members of a collection, write code that looks similar to this (in XML):

```
<db:update doc:name="Update according to patch data" doc:id="e407bd8e-f8e7-4130-bb72-de86a877bdc"
config-ref="Database_Config">
  <db:sql ><![CDATA[#['update people $(vars.dynamicUpdate.set) WHERE ID = :uriID' ]]]></db:sql>
  <db:input-parameters >
    <![CDATA[#['if (isEmpty(vars.dynamicUpdate.params)) { uriID: vars.personID }
      else vars.dynamicUpdate.params ++ { uriID: vars.personID}]]]>
  </db:input-parameters>
</db:update>
```

NOTES:

- The **update** statement has to have a WHERE clause (not created by the **makeDynamicUpdate()** function) that only updates the record whose ID is that passed in the URI Parameter for the ID (for example: personID, jobID, etc.)
- The URI parameter needs to be added as an input Parameter. The code above uses the ++ function to add the URI parameter to the other input Parameters, if any.
- To handle the situation that no input parameters were provided, the input-parameters code has an if/else statement to indicate which input-parameter object to use: just the URI parameter, or the other inputs plus the URI parameter.

Here is a graphical view of the implementation of this small project:



NOTES:

- Not specifically related to this blog's main topic, there is a flow at the top that validates that a person exists by the ID provided in the personID URI parameter. You would need an analogous flow for jobs and other collections. This flow raises and propagates an error if no person exists by the provided ID. It also sets the httpStatus to 404.
- Each flow for xxxByID (getPersonByID, patchPersonByID, etc.) first calls the validatePersonID flow before proceeding to other work, such as retrieving or updating the appropriate record(s) in from the database.
- The flows for GET methods where the GET takes query parameters all have dynamic SQL whose XML was shown earlier in this blog.
- The flows for PATCH methods all have dynamic SQL whose XML was also shown earlier in this blog.

Summary

This blog showed you how to:

5. Create RAML that has multiple dynamic query parameters for GET methods and various data models (types) for GET, PUT/POST, and PATCH methods.
6. Create a project based on that API (you did this in the Fundamentals and other courses)
7. How to modify the scaffolded code to:
 - a. Comply with the HTTP specification to ignore request bodies on GET requests
 - b. Create dynamic objects for GET and PATCH operations, saving those objects in event variables such as dynamicWhere and dynamicUpdate.

8. How to write SQL statements using the DB module that use the dynamic objects created in Step 3.b.