**Developing Applications with Oracle JET and Oracle Database**

Learn to divide an Oracle JavaScript Extension Toolkit (Oracle JET) application's logic that works with the data between the client-side and database tiers.

by Yuli Vasiliev

When choosing to build a web application on top of a relational database, as opposed to a flat file or files, you gain a number of benefits and capabilities when it comes to content-related operations. Not only can database data have different formats and complex relationships, but also it can be more efficiently processed within the database itself, which reduces the need to have data move between the application's front end and the underlying content source, thereby eliminating redundancy, hiding data complexity, improving efficient data processing, and maintaining data security.

In this article, you will look at an example of an application whose front end is implemented with Oracle JET and which uses Oracle Database as the back end, employing Oracle REST Data Services as the middle tier. The most interesting part is that the database tier of the application is used not only for storing and retrieving data, but also for data processing, thus allowing you to perform data manipulation operations closer to the data.

How It Works

Being entirely a client-side toolkit, Oracle JET does not support direct binding to interact with data sources, such as an Oracle database. Instead, data interaction is allowed only via web services, including JSON-based REST services, WebSockets, and server-sent events (SSEs). If your Oracle JET application's source data is stored in Oracle Database, the natural choice is to use [Oracle REST Data Services](http://www.oracle.com/technetwork/developer-tools/rest-data-services/overview/index.html), which allows you to map HTTP operations to database transactions with the help of simple wizards.

The figure below provides a visual depiction of how an Oracle JET application front end can interact with an Oracle database through a RESTful service created with Oracle REST Data Services.

Figure 1: REST calls to an Oracle database with Oracle JET and Oracle REST Data Services.

Of course, the above diagram shows a simplified model of how an Oracle JET application front end can interact with an Oracle database, using Oracle REST Data Services as the middleman. In fact, Oracle REST Data Services allows you to define a number of resource templates (each can be considered an individual RESTful service) to interact with different datasets. For each resource template, you can create a dedicated set of HTTP method handlers.

A method handler can be implemented as a SQL query or PL/SQL block for handling a particular HTTP method. Each resource template may have only one handler per HTTP method (GET, POST, PUT, DELETE, and so on). For details on RESTful services terminology, refer to the "[RESTful Services Terminology](https://docs.oracle.com/cd/E56351_01/doc.30/e87809/developing-REST-applications.htm" \l "AELIG90011" \t "_blank)" section in the [*REST Data Services Installation, Configuration, and Development Guide*](https://docs.oracle.com/cd/E56351_01/doc.30/e87809/toc.htm).

As an alternative to manually creating resource handlers with SQL or PL/SQL, you can use the AutoREST feature as a quick and easy way to REST-expose database objects, such as a database table, view or PL/SQL function, procedure, or entire PL/SQL package. Although AutoREST does not provide enough flexibility and customizability when it comes to extra validation or format manipulations, it lets you REST-expose data with just a few clicks using a simple wizard in Oracle SQL Developer.

You will see the AutoREST feature in action in the "[Creating a RESTful Service in the Database](https://developer.oracle.com/databases/jet-database#creating-service)" section later in this article. Then, in the "[Customizing Query Filtering on the Database Side](https://developer.oracle.com/databases/jet-database#query)" section, you'll look at an example of how the approach based on using SQL or PL/SQL to manually create resource handlers can be implemented, illustrating how you can push your application's data manipulation operations closer to the data.

Using Oracle JET Site Cookbook Demos

Like many other things, the concept of moving data processing to the data can be best understood by example. So, it might be interesting to look at an Oracle JET application that implements some data processing logic in the front end, and then look at a similar application that implements the same functionality but on the server side. Such a pair of Oracle JET demo applications can be found in the [Oracle JET Site Cookbook](http://www.oracle.com/webfolder/technetwork/jet/jetCookbook.html) by navigating to the following locations: [Framework > Common Model > Filtering > Fully-Fetched](http://www.oracle.com/webfolder/technetwork/jet/jetCookbook.html?component=filter&demo=full) and [Framework > Common Model > Filtering > Virtual](http://www.oracle.com/webfolder/technetwork/jet/jetCookbook.html?component=filter&demo=virtual).

**Note**: If you are already using Oracle JET, you are probably familiar with Oracle JET Site Cookbook, which provides a great number of demo applications, along with their source code in both JavaScript and HTML.

The first of the demos mentioned above performs client-side filtering, illustrating how all the functionality required to filter the displayed data based on text box input can be implemented in the front end—solely with JavaScript.

The second demo assumes that the filtering functionality is implemented on the server side. The implementation is not provided, but every user input in the filter text box is sent to the underlying server, implying that the server is supposed to perform filtering and then return a filtered dataset to the front end. In the following sections, you will see how this server-side filtering functionality can be implemented in an Oracle database with the help of Oracle REST Data Services.

The figure below shows a screenshot of this cookbook demo:

Figure 2: A screenshot of the Framework > Common Model > Filtering > Virtual cookbook demo.

To simplify things, this article's sample application is built on top of this server-side filtering cookbook demo, using it as a template for the front end. With the approach of using an existing application as a template on which to base your new application, you don't need to develop each and every part of a new application from scratch; you can reuse some blocks of existing code without having to change them.

If you already have some experience with Oracle JET, you know that developing the client side of an Oracle JET application assumes coding in both JavaScript and HTML. In this particular case, you can borrow the main section of the default HTML file (index.html) from the cookbook demo as is, without any changes. The JavaScript part will require some changes to adapt it to your environment, however.

In particular, you will need to remove the references to the mock REST server used in the demo to simulate a RESTful service and instead connect your Oracle JET code to the RESTful service that you will create in an Oracle database, as described in the "[Creating a RESTful Service in the Database](https://developer.oracle.com/databases/jet-database#creating-service)" section later in this article. All the details that cover creating the article's sample front end are provided in the "[Incorporating a Cookbook App's Code into an Oracle JET Starter Template](https://developer.oracle.com/databases/jet-database#starter)" section later in this article.

First, however, before you can create a RESTful service in the database, you need to decide on a database object whose data you want to be REST-exposed.

Creating a Database Object on Which to Define a RESTful Service

So, your first step is to define a database object (table or view) upon which you will then create a RESTful service. If your data is JSON, as the case in this example, the most natural way to go is first to create a relational view upon your JSON data, shredding it into relational rows. Then, you can REST-enable this relational view with the help of a simple wizard in Oracle SQL Developer.

It is interesting to note that Oracle Database allows you to create a relational view upon JSON data stored outside of the database. If you recall, the cookbook demo used for the article's sample as a template consumes data from the accompanying departments.json file. Because this file does not seem to be publicly accessible, you can create a file with the same name in your system and then paste into it the content of the demo's file, which is available from the [web page for this demo](http://www.oracle.com/webfolder/technetwork/jet/jetCookbook.html?component=filter&demo=virtual): find and click the departments.json link in the Virtual section of the page to display the JSON data, which should look as follows:

{

"Departments" : [ {

"DepartmentId" : 10,

"DepartmentName" : "Administration",

"ManagerId" : null,

"LocationId" : null

},

{

"DepartmentId" : 20,

"DepartmentName" : "Marketing",

"ManagerId" : null,

"LocationId" : null

},

{

...

After you have copied the JSON data into a file located on your system, you can define a view in the database to expose that data relationally, without actually having to load the data into the database. First, however, you might want to create a database schema for this view. For that, you can use Oracle SQL Developer or another SQL tool of choice. Connect with DBA privileges and issue the following commands:

CREATE USER usr IDENTIFIED BY pswd;

GRANT CONNECT, RESOURCE TO usr;

GRANT CREATE VIEW TO usr;

Then, you can connect as the newly created user usr and create the view. This can be done as follows. The example below assumes that you have placed your departments.json file in your web server documents directory:

CREATE OR REPLACE VIEW dept\_v AS

SELECT d.\* FROM JSON\_TABLE(

httpuritype('http://localhost/departments.json').getCLOB(), '$.Departments[\*]'

COLUMNS (

DepartmentId NUMBER(3) PATH '$.DepartmentId',

DepartmentName VARCHAR2(100 CHAR) PATH '$.DepartmentName',

ManagerId NUMBER(5) PATH '$.ManagerId',

LocationId NUMBER(5) PATH '$.LocationId'

)) d;

Before you can query the above view, make sure to grant the connect privilege for host localhost to USR. To do this, you'll need to reconnect with DBA privileges and run the following PL/SQL code:

BEGIN

DBMS\_NETWORK\_ACL\_ADMIN.CREATE\_ACL(

acl => 'localhost.xml',

description => 'localhost ACL',

principal => 'USR',

is\_grant => true,

privilege => 'connect');

DBMS\_NETWORK\_ACL\_ADMIN.ASSIGN\_ACL(

acl => 'localhost.xml',

host => 'localhost');

END;

/

COMMIT;

After the above has completed successfully, you can query the view as if it were a regular relational table:

SELECT \* FROM DEPT\_V;

DEPARTMENTID DEPARTMENTNAME MANAGERID LOCATIONID

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

10 Administration

20 Marketing

30 Transportation

40 Shipping

50 Human Resources

60 Operations

70 Inventory

80 Sales 145 2500

100 Finance

110 Documentation

130 Billing 1700

140 Control And Credit1700

In the above example, you created a view that provides a relational representation of external JSON data, accessing the source file via HTTP with the help of the HTTPURITYPE function. The advantage of this approach is that it allows you to retrieve external data not only from local HTTP-accessible files but also from remote web pages and files.

As an alternative to the above, you can create an external database table of JSON data from the content of a JSON file, as described in the "[Loading External JSON Data](https://docs.oracle.com/database/122/ADJSN/loading-external-json-data.htm)" section of the [*JSON Developer's Guide*](https://docs.oracle.com/database/122/ADJSN/toc.htm). Then you can create a relational view upon this external table. An example can be found in the "Projecting JSON Data Relationally" section of the "[Oracle Database Speaks JSON Now](https://community.oracle.com/docs/DOC-994732)" article published on the [Oracle Community website](https://community.oracle.com/).

In any case, once you have a relational representation of your data in the form of a database table or view, you can easily REST-enable it with Oracle REST Data Services, as outlined in the next section.

Creating a RESTful Service in the Database

In this section, you'll look at how to develop and deploy a RESTful service in an Oracle database with Oracle REST Data Services, using the RESTful Services feature in Oracle SQL Developer. With this feature, enabling REST access to database objects is as easy as following the steps of a simple wizard.

The general steps you need to complete include

* Installing Oracle REST Data Services using Oracle SQL Developer
* Enabling Oracle REST Data Services queries to access the DEPT\_V view, created as described in the previous section
* Running the Oracle REST Data Services server in standalone mode

Before you can use Oracle SQL Developer's RESTful Services feature, however, you need to install Oracle REST Data Services. This process is described in the "[Installing Oracle REST Data Services](https://docs.oracle.com/database/sql-developer-4.2/RPTUG/sql-developer-REST.htm#GUID-581C9DD7-634D-4470-8E78-D4F2AA0CEFD)" section in the [*SQL Developer User's Guide*](https://docs.oracle.com/database/sql-developer-4.2/RPTUG/toc.htm). This section also provides the instructions on how you can then run Oracle REST Data Services in standalone mode.

**Note**: Apart from the standalone mode, which is suitable for test and development use only, the deployment options for Oracle REST Data Services include Oracle WebLogic Server, GlassFish Server, and Apache Tomcat. For further details, refer to the [*REST Data Services Installation, Configuration, and Development Guide*](https://docs.oracle.com/cd/E56351_01/doc.30/e87809/toc.htm). Another important note is that using Oracle SQL Developer to install Oracle REST Data Services in standalone mode is optional. A command-line alternative is described in the "[Installing Oracle REST Data Services](https://docs.oracle.com/cd/E56351_01/doc.30/e87809/installing-REST-data-services.htm#GUID-A16BCCA2-8081-4062-A635-9F7C36FC394)" chapter in the above guide.

After you have installed Oracle REST Data Services, you can enable REST access to a database object with the help of the RESTful Services Wizard in Oracle SQL Developer. Thus, to REST-enable the DEPT\_V view created in the previous section, find it in the Connections navigator, expanding **Tables (Filtered)** under **USR**. Then, right-click **DEPT\_V** and select **Enable REST Service** from the pop-up menu to launch the RESTful Services Wizard. Select and deselect the checkboxes, as shown in the screenshot below:

Figure 3: A screenshot of the first screen of the RESTful Services Wizard.

Then, click **Next** to move on to the Summary screen of the wizard. If you click the **SQL** tab in this screen, you'll see a PL/SQL block that will be executed upon the completion of the wizard:

Figure 4: A screenshot of the SQL tab of the RESTful Summary screen.

This reveals that the wizard uses a PL/SQL API behind the scenes. Of course, you can use this API explicitly in any SQL tool of choice as an alternative to the Oracle REST Data Services wizards available in Oracle SQL Developer. For details on using the Oracle REST Data Services PL/SQL API, refer to the "[Using the Oracle REST Data Services PL/SQL API](https://docs.oracle.com/cd/E56351_01/doc.30/e87809/developing-REST-applications.htm#AELIG90176)" section in the *REST Data Services Installation, Configuration, and Development Guide*, as well as to the "[ORDS PL/SQL Package Reference](https://docs.oracle.com/cd/E56351_01/doc.30/e87809/ORDS-reference.htm)" section.

After you have REST-enabled the DEPT\_V view, you can access it via the following URL:

http://yourdbhost:9090/ords/usr/dept\_v/

Before you can use it, however, make sure that the Oracle REST Data Services server is running. In Oracle SQL Developer, check the Processes pane. If the Oracle REST Data Services server is running, it will be displayed in this pane. Otherwise, you have to launch it. This can be done with the wizard that you can invoke by selecting the **Tools > REST Data Services > Run** menu item.

If you are not using Oracle SQL Developer, you can launch the Oracle REST Data Services server in standalone mode from the command line, as described in the "[Running in Standalone Mode](https://docs.oracle.com/cd/E56351_01/doc.30/e87809/installing-REST-data-services.htm#AELIG7022)" section in the *REST Data Services Installation, Configuration, and Development Guide*.

Once you have the Oracle REST Data Services server running, you can test your RESTful service. The simplest way to do that is to point your browser to the service URL. In this particular example, the result should look like the following figure:

Figure 5: Testing the RESTful service that enables REST access to the DEPT\_V database view.

As you will learn in the next section, even the auto-REST enabling used in this example allows you to take advantage of the "Filtering in queries" Oracle REST Data Services feature, which enables limiting a collection resource just like the WHERE clause in SQL, providing the simplest way of moving filtering logic to the database.

Incorporating a Cookbook App's Code into an Oracle JET Starter Template

Now that you have a REST data source to be consumed in your application, you can proceed to coding the front end using the Oracle JET framework.

Before you can use Oracle JET, you need to download it from the Oracle Technology Network (OTN) at the [Oracle JavaScript Extension Toolkit Download page](http://www.oracle.com/technetwork/developer-tools/jet/downloads/index.html). As an option, you can download Oracle JET with a QuickStart template that already includes an index.html file preconfigured to work with the Oracle JET libraries. For this article's sample, you might choose [Web Starter Template – Basic](http://download.oracle.com/otn/JET/400/JET-Template-Web-Basic.zip), which is available as a zip file.

Unpack JET-Template-Web-Basic.zip on your local machine and then customize the unpacked template application, incorporating the code from the cookbook demo discussed earlier and making the changes as described in the rest of this section.

So, change to the unpacked directory and open the index.html file for editing. Find the main div element in the HTML document (highlighted in bold below):

...

</header>

**<div role="main" class="oj-web-applayout-max-width oj-web-applayout-content">**

**</div>**

<footer class="oj-web-applayout-footer" role="contentinfo">

...

And insert the following tags inside it, thus copying the content of the main div element taken from the demo's index.html file:

<div id="quickFilter" class="frame">

<oj-input-text id="filterDepartName" maxlength="30"

placeholder="Type to filter department name" value="{{filter}}">

</oj-input-text>

</div>

<oj-data-grid

id="datagrid"

style="height:300px; max-width:320px"

data="[[datasource]]"

selection-mode.row="single"

header.column.style="width:150px">

</oj-data-grid>

That is all you need to do in index.html.

The next step is to modify the JavaScript code. For that, change to the js directory of the unpacked template application and open the appController.js file for editing.

To start with, in appController.js, change the define line as follows:

define(['ojs/ojcore', 'knockout',

'jquery','ojs/ojknockout','ojs/ojmodel', 'ojs/ojdatagrid',

'ojs/ojcollectiondatagriddatasource', 'ojs/ojinputtext',

'ojs/ojbutton']

Then, in this same appController.js file, move on to the ControllerViewModel function, and insert the following code (taken from the viewModel function in demo.js) right between the Header and the Footer blocks. The code below already reflects the changes (highlighted in bold) you have to make so that it can interact with the RESTful service created in the previous section:

// Header

...

self.filter = ko.observable('');

var latest = "";

self.handleKeyUp = function(event) {

latest = event.target.value;

var myCollection = self.DeptCol();

myCollection.refresh();

};

document.getElementById('datagrid').addEventListener('keyup', self.handleKeyUp);

function filterDepartments() {

var filter = latest;

if (filter === "") {

return "";

}

**return '{"DepartmentName": {"$instr":"'+ filter + '"}}';**

**//return "DepartmentName=@"+filter;**

};

// Keep the filter on based on the UI settings, with every query

// including those generated by the component

function getURL(operation, collection, options) {

var url = self.serviceURL;

if (options.fetchSize !== undefined) {

url += "?limit="+options.fetchSize;

sawOpt = true;

}

if (options.startIndex !== undefined) {

url += "&offset="+options.startIndex;

}

var q = filterDepartments();

if (q !== undefined && q !== "") {

url +="&q=" + q;

}

url +="&totalResults=true";

return url;

};

self.serviceURL = '**http://yourdbhost:9090/ords/usr/dept\_v/**';

self.DeptCol = ko.observable();

self.datasource = ko.observable();

self.parseDept = function(response) {

return {DepartmentId: response['**departmentid**'],

DepartmentName: response['**departmentname**']};

};

self.Department = oj.Model.extend({

urlRoot: self.serviceURL,

parse: self.parseDept,

idAttribute: 'DepartmentId'

});

self.myDept = new self.Department();

self.DeptCollection = oj.Collection.extend({

customURL: getURL,

model: self.myDept,

**fetchSize:50,**

comparator: "DepartmentId"

});

self.DeptCol(new self.DeptCollection());

var elem = document.getElementById("filterDepartName");

elem.addEventListener('keyup', self.handleKeyUp);

$.getJSON("**http://yourdbhost:9090/ords/usr/dept\_v/**",

function (data) {

// **new MockPagingRESTServer(data, {id:"DepartmentId", collProp:"Departments"});**

self.datasource(new oj.CollectionDataGridDataSource(self.DeptCol()));

});

// Footer

...

Looking through the above code, the first change you might notice is in the return clause of the filterDepartments function. This function creates and returns the query parameter q, which is then attached to the url passed to the underling server that processes application requests. Since the underlying server has been replaced with a different one, you had to change the query parameter syntax, according to the requirements of the new server. In this particular example, you must change it to meet the requirements of Oracle REST Data Services. Based on the content of this parameter, Oracle REST Data Services will generate a WHERE predicate for the query it issues behind the scenes against the dept\_v view.

**Note**: To understand how Oracle REST Data Services filtering in queries against REST-enabled objects works, check out the "[Filtering in Queries](https://docs.oracle.com/cd/E56351_01/doc.30/e87809/developing-REST-applications.htm#GUID-091748F8-3D14-402B-9310-25E6A9116B4)" section in the "[Developing Oracle REST Data Services Applications](https://docs.oracle.com/cd/E56351_01/doc.30/e87809/developing-REST-applications.htm#AELIG3000)" chapter of the *REST Data Services Installation, Configuration, and Development Guide*.

The next alteration in the above code is changing the service URL value to the one through which you can access your RESTful service. In this particular example, the URL is the following:

**http://yourdbhost:9090/ords/usr/dept\_v/**

Then, you must modify the DepartmentId and DepartmentName identifier names in the response object to suit those that Oracle REST Data Services generates. In particular, you must replace them with departmentid and departmentname, respectively. This incompatibility originates from the differences in naming standards used in Oracle Database and JavaScript: camelCase (for example, EmpId) in JavaScript and snake\_case (for example, emp\_id) in Oracle Database. More importantly, Oracle SQL is a case-insensitive language, unlike JavaScript. This means that you cannot expect a camelCase-named column in an Oracle Database object REST-exposed through Oracle REST Data Services.

Finally, note that the demo code employs a mock REST server that only simulates a RESTful service. Because this mock server is no longer required, the line in which it is initiated has been commented out.

The script in the main.js file does not require any changes. So you can now test the application. For that, just double-click the index.html file that was edited as described in the beginning of this section. Once it's loaded, you can type in any characters in the filter text box to see how filtering works in the newly created application:

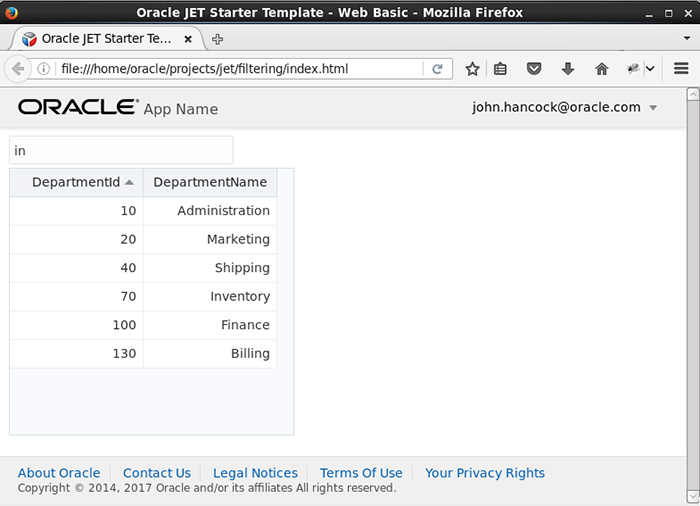


Figure 6: Checking how server-side filtering works.

As you can see, from the user's standpoint, the application you just created works just like the [client-side filtering demo](http://www.oracle.com/webfolder/technetwork/jet/jetCookbook.html?component=filter&demo=full) works. Behind the scenes, however, the main difference is that the newly created application performs filtering on the server side, using the "filtering in queries" Oracle REST Data Services feature.

Customizing Query Filtering on the Database Side

Of course, auto-REST enabling database objects and then using the "filtering in queries" feature is not the only option you have when using Oracle REST Data Services to create RESTful services that return database data. A more flexible and customizable alternative is to use manually created resource modules that require you to explicitly specify SQL or PL/SQL to process REST requests and interact with the underlying database data. Because SQL queries and PL/SQL code used in such resource modules may have complex structure and relationships, your first step is to define all necessary objects in the database.

In this section, you will define all the database objects needed to support the RESTful resource that will be created as described in the next section. In particular, you will create a PL/SQL pipelined table function that generates and returns a collection from the DEPT\_V view, based on an input parameter. This pipelined table function will then be called from the SQL query you define for the GET resource handler.

Before you can create the PL/SQL function, you need to create two types: an object type that represents a row from DEPT\_V and a table type of this object type. After that, you can create the function. You can create all these objects in the USR database schema, in which the DEPT\_V view was defined, as described in the "Creating a Database Object on Which to Define a RESTful Service" section earlier in this article.

So, the steps to be performed are the following:

1. Make sure to grant CREATE TYPE and CREATE PROCEDURE privileges to the USR user.

2. Once the privileges have been granted, connect to the database as the USR user.

3. Create an object that represents one row of the results:

CREATE OR REPLACE TYPE dept\_t

AS OBJECT

(

DEPARTMENTID NUMBER(3),

DEPARTMENTNAME VARCHAR2(400),

MANAGERID NUMBER(5),

LOCATIONID NUMBER(5)

);

4. Create a table type of the object type created in the previous step:

CREATE OR REPLACE TYPE dept\_tab\_t

AS TABLE OF dept\_t;

5. Create a function that returns the above table type:

CREATE OR REPLACE FUNCTION dept\_f(dept\_name VARCHAR2)

RETURN dept\_tab\_t

PIPELINED

AS

BEGIN

IF dept\_name IS null THEN

FOR r IN (SELECT \* FROM dept\_v) LOOP

PIPE ROW (dept\_t(r.DEPARTMENTID, r.DEPARTMENTNAME,

r.MANAGERID, r.LOCATIONID));

END LOOP;

ELSE

FOR r IN (SELECT \* FROM dept\_v d WHERE

regexp\_instr(d.departmentname,dept\_name,1,1,0,'i')>0) LOOP

PIPE ROW (dept\_t(r.DEPARTMENTID, r.DEPARTMENTNAME,

r.MANAGERID, r.LOCATIONID));

END LOOP;

END IF;

RETURN;

END;

/

As you can see, the above function returns an entire collection of the DEPT\_V view rows if the input parameter is null. Otherwise, it returns only those DEPT\_V rows that satisfy the filtering criteria.

Before moving on to creating a RESTful service that will use the above DEPT\_F function, you might want to test this function. For that, you might issue the following SQL statement:

SELECT \* from TABLE(dept\_f(null));

This should output all the DEPT\_V rows:

DEPARTMENTID DEPARTMENTNAME MANAGERID LOCATIONID

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

10 Administration

20 Marketing

30 Transportation

40 Shipping

50 Human Resources

...

Then, you might want to test how filtering works:

SELECT \* from TABLE(dept\_f('ad'));

This time, the output should look as follows:

DEPARTMENTID DEPARTMENTNAME MANAGERID LOCATIONID

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

10 Administration

Now you can move on and create a resource module with the GET resource handler mapped to the above query.

Creating a RESTful Service to Retrieve Data Using a Parameter

The process of creating an Oracle REST Data Services resource module in Oracle SQL Developer is described in detail in the "[Lab 3: REST Development](http://www.oracle.com/webfolder/technetwork/tutorials/obe/db/ords/r30/REST%20Development/REST%20Development.html)" hands-on lab.

To summarize, you'll need to accomplish the following general steps:

1. Create a new REST development connection.

2. Define a resource module: URI Prefix: **jet/**.

3. Define a resource template: URI Pattern: **dept\_v**/.

4. Define a resource handler: Method: **GET**.

5. Add a query to the resource handler:

**SELECT \* from TABLE(dept\_f(:DepartmentName));**

6. Upload the resource module.

The following figure summarizes what you should get as the result:

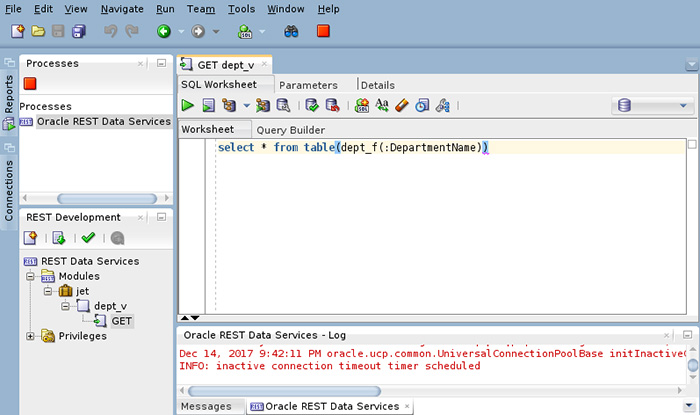


Figure 7: The GET resource handler for the article's sample.

In this particular example, there is no need to create any other resource handlers (for example, PUT, POST, and so on), because only the GET method is going to be used.

Adjusting Oracle JET Code

To make use of the RESTful service created in the previous section, some adjustments in the application Oracle JET code are required. In particular, you need to edit the application's appController.js file as follows (the changes to be made are highlighted in bold):

...

function filterDepartments() {

var filter = latest;

if (filter === "") {

return "";

}

**return "&DepartmentName=" + filter;**

**//return '{"DepartmentName": {"$instr":"'+ filter + '"}}';**

};

...

function getURL(operation, collection, options) {

var url = self.serviceURL;

if (options.fetchSize !== undefined) {

url += "?limit="+options.fetchSize;

sawOpt = true;

}

if (options.startIndex !== undefined) {

url += "&offset="+options.startIndex;

}

var q = filterDepartments();

if (q !== undefined && q !== "") {

**// url +="&q=" + q;**

**url += q;**

}

url +="&totalResults=true";

return url;

};

**// self.serviceURL = 'http://yourdbhost:9090/ords/usr/dept\_v/';**

**self.serviceURL = 'http://yourdbhost:9090/ords/usr/jet/dept\_v';**

...

**// $.getJSON("http://yourdbhost:9090/ords/usr/dept\_v/",**

**$.getJSON("http://yourdbhost:9090/ords/usr/jet/dept\_v",**

function (data) {

// new MockPagingRESTServer(data, {id:"DepartmentId", collProp:"Departments"});

self.datasource(new oj.CollectionDataGridDataSource(self.DeptCol()));

});

...

After applying the above changes to appController.js, you can test the updated application. Once again, from the user's standpoint, it should work as before.

Conclusion

By following the example in this article, you learned how an Oracle JET application's logic can be pushed into the database, giving you the benefits of Oracle Database functionality.

See Also

* "[Oracle JET and Oracle Database – a Beautiful Combination](http://www.oracle.com/technetwork/articles/database/oracle-db-oraclejet-combination-3941880.html)"
* "[Working With REST in Oracle JET](https://dzone.com/articles/working-with-rest-in-oracle-jet)"
* [Oracle JET Site Cookbook](http://www.oracle.com/webfolder/technetwork/jet/jetCookbook.html)
* [Oracle REST Data Services OTN website](http://www.oracle.com/technetwork/developer-tools/rest-data-services/overview/index.html)

About the Author

**Yuli Vasiliev** is a software developer, freelance author, and consultant currently specializing in open source development, databases, business intelligence (BI), machine learning, and virtualization. He is the author of a series of books on Oracle technology, including *Oracle Business Intelligence: An introduction to Business Analysis and Reporting* (Packt Publishing) and *PHP Oracle Web Development: Data processing, Security, Caching, XML, Web Services, and Ajax* (Packt Publishing).