An Introduction to Using Oracle Database 12*c* as a No-SQL JSON Document Store Part1: Using SODA for REST with Oracle Database 12c

Mark Drake
Manager, Product Management
Oracle XML DB & JSON

Contents

Using SODA for REST with Oracle Database 12c

Purpose

This Lab will walk you through the basics of using Oracle Database as a JSON document store. The Lab is organized into 2 sections.

- The first section provides an introduction to SODA for REST; an API that allows REST based applications to use the Oracle Database as a JSON document store.

- The second section provides an introduction in how to use SQL to work with JSON documents stored in the Oracle Database.

The entire laboratory should take approximately 60 minutes to complete.

Introduction

The first part of the HOL will introduce a simple REST based API that allows JSON documents to be stored, retrieved and queried without requiring any knowledge of SQL. This API is an implementation the Oracle Simple Document Access or SODA specification. An implementation of the SODA API for Java developers is also available, and implementations for other development environments are forthcoming.

REST (Representational state transfer**)**

Representational state transfer (REST) is an architectural style that was used to define HTTP 1.1 and Uniform Resource Identifiers. Consequently a REST based API will have a very strong resemblance to the basic functionality provided by an HTTP Server, and most REST based systems will be implemented using an HTTP client and an HTTP Server. The typical REST implementation maps CRUD (Create, Retrieve. Update and Delete) operations to the HTTP Verbs, POST, GET, PUT and DELETE.

One of the key characteristics of a REST based system is that it is stateless. That is, the server itself does not track or manage object state. Each operation performed against a REST based server is atomic, and is considered a transaction in its own right. It should also be noted that many of the facilities that that are taken for granted in relational development environment, such as locking and concurrency control, are left to the application to manage in a REST based world.

One of the main advantages of implementing a REST based architecture using HTTP is that it allows the services to be consumed from just about any modern programming platform. The includes traditional programming languages such as ‘C’, C#, C++, JAVA and PL/SQL as well as most of the modern scripting languages including JavaScript, Perl, Python, Ruby etc.

Oracle Database Document Store and Document Collections

Oracle Database 12c (12.1.0.2.0) introduces a document store backed by the Oracle Database. Documents are stored using one or more Collections. A Collection typically manages a set of related documents, in much the same way that a table manages a set of related rows. Unlike a table, where each row must have exactly the same structure, A collections can contain heterogeneous or homogeneous documents. Each document in the collection is identified by a unique id. Typically this id is assigned by the server when the document is created, although client assigned ids are also permitted. A document collection may optionally track other metadata about the document, including the date and time it was created and the date and time it was last modified.

Document Centric API: SODA for REST

SODA for REST delivers a set of HTTP based services that conform to the REST paradigm and which can be used to store and retrieve JSON documents using an Oracle database 12c based document store. The services provided by SODA for REST include:

1. List the Available collections
2. Create and Drop a collection
3. Insert a document into a collection
4. Retrieve a document from a collection
5. Update one of the documents in a collection
6. Delete a document from a collection
7. List the contents of a collection
8. Search a collection
9. Index a collection
10. Bulk insert operations on a collection

These API's provide application developers with a persistence mechanism for the JSON documents created by their applications. When an application needs to preserve the state on an object it will use the development environment's JSON serialization Developer's will use the JSON serialization support provided by their environment to generate JSON documents that represent the application's objects and state and then use SODA for REST to store those documents in the database. When application state needs to be recovered, SODA for REST will be used to retrieve the required documents from the document store, and then the native JSON parsing capabilities of the development environment will be used to recreate application objects from the JSON documents.

Complete documentation for SODA for REST can be found here

Real World Use-Cases

The Hands-on-Lab is based on a very simple HTML application that allows the student to exercise the various services provided by SOADA for REST. The aim of this lab is to demonstrate how each of the SODA for REST services can be invoked; what arguments they expect, and what type of responses they generate.

This lab is not meant to teach the student how to develop an application using JavaScript and HTML. The purpose of the HTML application is simply to set up the parameters required to invoke each of the services and view the details of the HTTP Request and HTTP Response.

When using API's that require a JSON document to be submitted as part of the API call, the JSON will be loaded from files stored in the local file system. This allows the student to avoid having to create syntactically correct JSON in order to complete the exercises. In the real-world JSON will be machine generated by the application that is making use of SODA for REST.

Conversely when using the application to retrieve JSON documents from a Document Collection, the processing that is performed on the JSON is rendering it as HTML, so that the structure of the JSON can be examined. In some cases, such as when processing lists of documents, the JSON may also be used to populate List Boxes. In real world use-cases an application would convert the JSON back into objects that the application operates on, using the JSON parser native to the development environment.

The Hands-on-Lab also focuses on using JavaScript. However, as was mentioned in the introduction to REST, it should be noted that the SODA for REST can be invoked from any programming environment that is capable of generating HTTP PUT, GET, POST and DELETE operations. This includes all modern programming and scripting languages.

Using the SODA for REST from JavaScript

This Hands-on-Lab will show how SODA for REST services can be invoked from JavaScript. An HTML page will provide a framework to allow the student to select the service to be invoked and to examine the HTTP Request and HTTP Response. When working in JavaScript the basic functionality to perform an HTTP operation is provided by the XMLHTTPRequest object. This object is defined by the W3C, and the specification can be found here http://www.w3.org/TR/XMLHttpRequest/. The XMLHTTPRequest object is supported natively by all modern browsers.

Often JavaScript developers will choose to make use of a framework, such as Angular.js or JQuery when working with REST based services. These frameworks provide a higher level interface to the functionality provided by the RESTAPI, simplifying the coding effort required to make use of the API. However all of these frameworks are simply providing wrappers over the functionality provided the XMLHTTPRequest object.

Making HTTP Requests using an XMLHTTPRequest object

In order to make an HTTP request using an XMLHTTPRequest object the basic structure of the JavaScript is as follows:

1. Instantiate the XMLHTTRequest object.
2. Specific the METHOD and URI to be accessed
3. Send the Request, supplying data as required
4. Process the Response.

The following code snippet shows how a simple JavaScript function called getDocument() which performs a “GET” operation. The function expects to be passed two arguments. The first is the URL which identifies the document to be fetched; the second is the value of the id that identifies the HTML element which will be updated using the content of JSON document returned by the request.

```
function getDocument (URL,id) {

        var jsonObject = null;
        var XHR = new XMLHttpRequest();
        XHR.open ("GET", URL, false);
        XHR.send(null);
        if (XHR.status == 200) {
          jsonObject = JSON.parse(XHR.responseText);
         renderJSON(id,jsonObject)
        }
}
```

The function performs the following tasks:

1. Instantiates an XMLHTTPRequest object

2. Invokes the open method, specifying that a GET operation is to be performed on the specified URL. The third argument to the open method indicates that the operation will be performed synchronously. This means that the browser will wait for the operation to complete before continuing to execute the JavaScript function. Note that using the XMLHTTPRequest object in its synchronous mode is considered to be extremely bad practice.

3. Invokes the send method to actually make the request to the server. Since we are making a synchronous call the function waits for the response to become available before executing the next statement.

4. Checks the status attribute of the XMLHTTPRequest object to be sure that the operation completed successfully

5. Invokes the Parse method of the Browser’s JSON object to parse the responseText attribute of the XMLHTTPRequest object. This creates a JavaScript object from the JSON document.

6. Invokes function renderJSON to process the JSON object.

All of the JavaScript functions used in the Lab perform asynchronous operations.

Now that we understand the basics using the XMLHTTPRequest object to make HTTP requests from JavaScript we are ready to see how these techniques can be used to invoke the services provided by SODA for REST.

Synchronous operations are considered bad practice since they cause the runtime environment to wait for the operation to complete. This means that when using a Browser the Browser would become unresponsive to user input until the operation was complete.

In order to avoid the issues associated with synchronous operations, best practices recommend the use of asynchronous operations. Asynchronous operations require the developer to supply a callback function that will be automatically invoked when the status of the HTTP request changes. The callback function will be responsible for testing the current status of the HTTP operation and processing the results of the operation once it has completed.

The following example shows a modified version of the getDocument() function that performs an asynchronous HTTP request and supplies a callback function that performs the processing of the response.

```
function getDocument (URL,id) {

        var callback = function (XHR,id) {
          jsonObject = JSON.parse(XHR.responseText);
         renderJSON(id,jsonObject)
        }

        var jsonObject = null;
        var XHR = new XMLHttpRequest();
        XHR.open ("GET", URL, true);
        XHR.onreadystatechange=function() { if( XHR.readyState==4 ) { callback(XHR,id) } };
        XHR.send(null);
}
```

In this example the processing that is to be performed on completion of the XMLHTTPRequest is defined as an anonymous function. The function is invoked from the callback function that is attached to the onreadystatechange event of the XMLHTTPRequest object.

During asynchronous processing of an XMLHTTPRequest, the value of the readyState attribute is updated to indicate the progress of the HTTP operation. Each time the value of the readyState attribute changes the function associated with the onreadystatechange event is invoked. When the value of the readyState property is 4, the request has completed and the results are available.

The function performs the following tasks:

1. Define a callback function. The callback function expects two arguments, the XMLHTTPRequest object and the id of the HTML element that is to be updated once the HTTP request is complete. The callback function performs the following processing

    a. Invoke the Parse method of the Browser’s JSON object to parse the responseText attribute of the XMLHTTPRequest object. This creates a JavaScript object from the JSON document.

    b. Invokes function renderJSON to process the JSON object.

2. Instantiates an XMLHTTPRequest object

3. Invokes the open method, specifying that a GET operation is to be performed on the specified URL. The third argument to the open method indicates that the operation will be performed asynchronously.

4. Sets up the onreadystatechange function which checks the value of the readyState attribute and invokes the callback function once the value of readyState attribute is 4.

5. Invokes the send method to actually make the request to the server. Since we are making a synchronous call the function waits for the response to become available before executing the next statement.

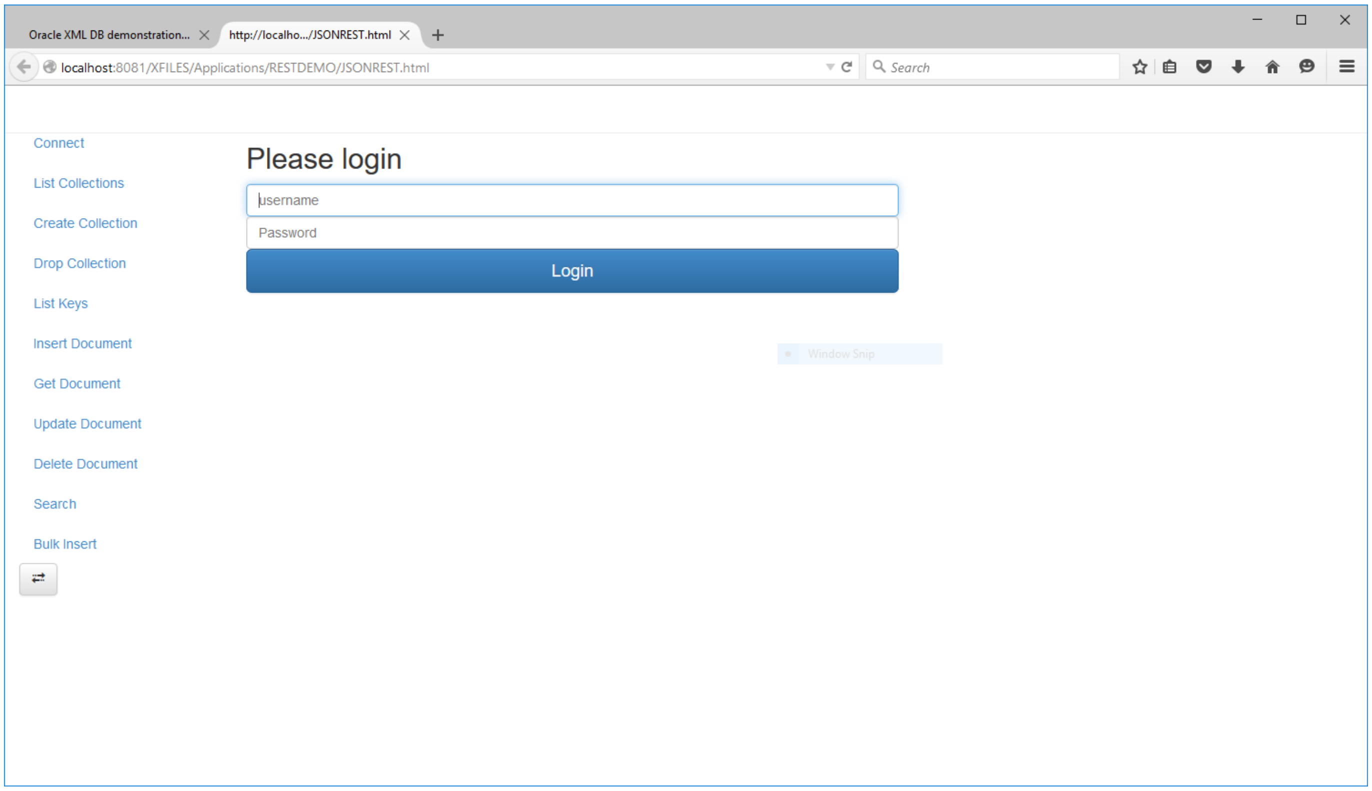
At this point the getDocument() function terminates and control is passed to back to caller. Processing will resume when the onreadystatechange function invokes the callback.

Step 1: Connecting to the database

In this HOL we are running using the SODA for REST servlet under the Oracle Database's native HTTP server. This means that the only supported authentication mechanism is database username and password. In a production environment it is expected that the Rest Services will be installed at part of Oracle Rest Database Services (ORDS). ORDS provides support for many different authentication schemes.

Open the URL http://localhost:8081/XFILES/Applications/RESTDEMO/JSONREST.html and click on the **connect** tab

This will display the following web page

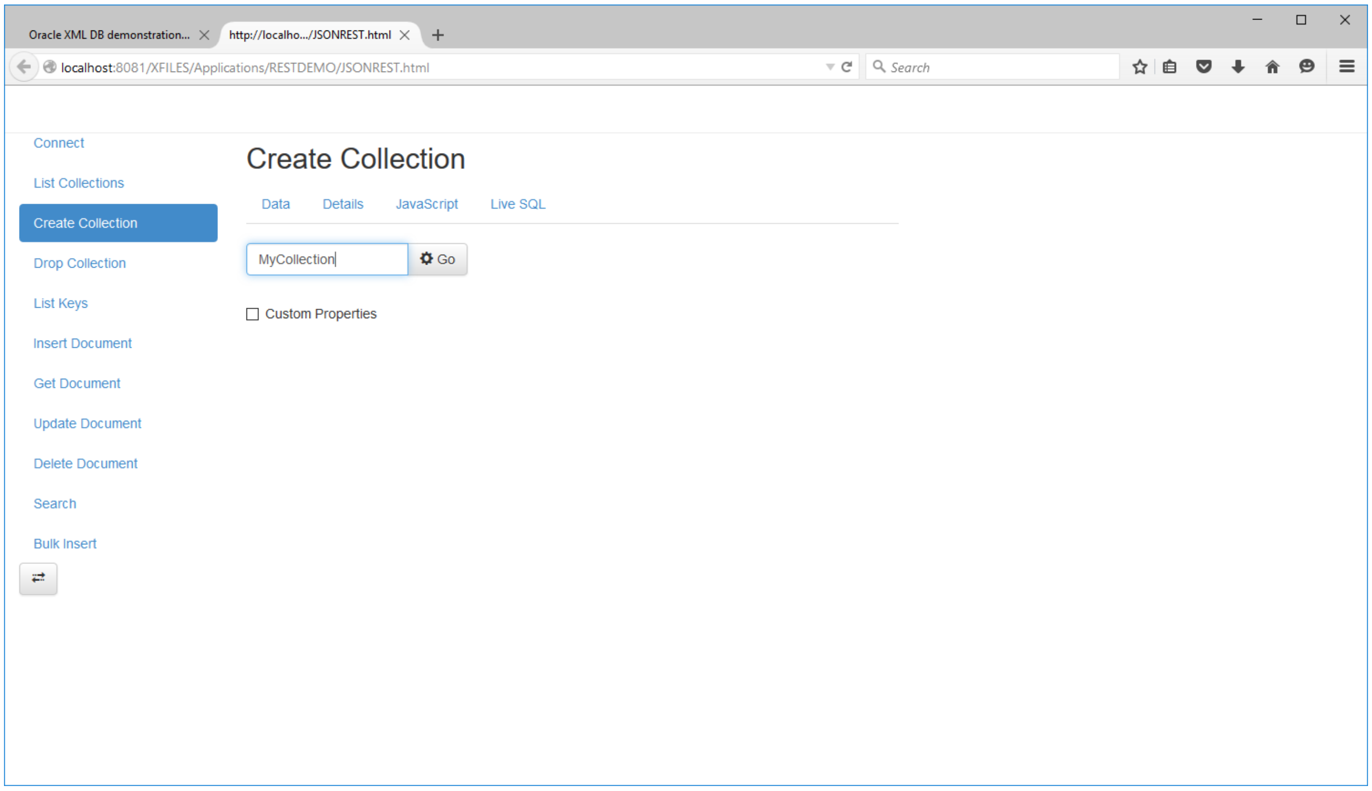


Login as user SCOTT, password <password>. Since we are using the Database's Native HTTP Server with Digest authentication enabled both user and password are case sensitive.

Step 2: Creating Collections

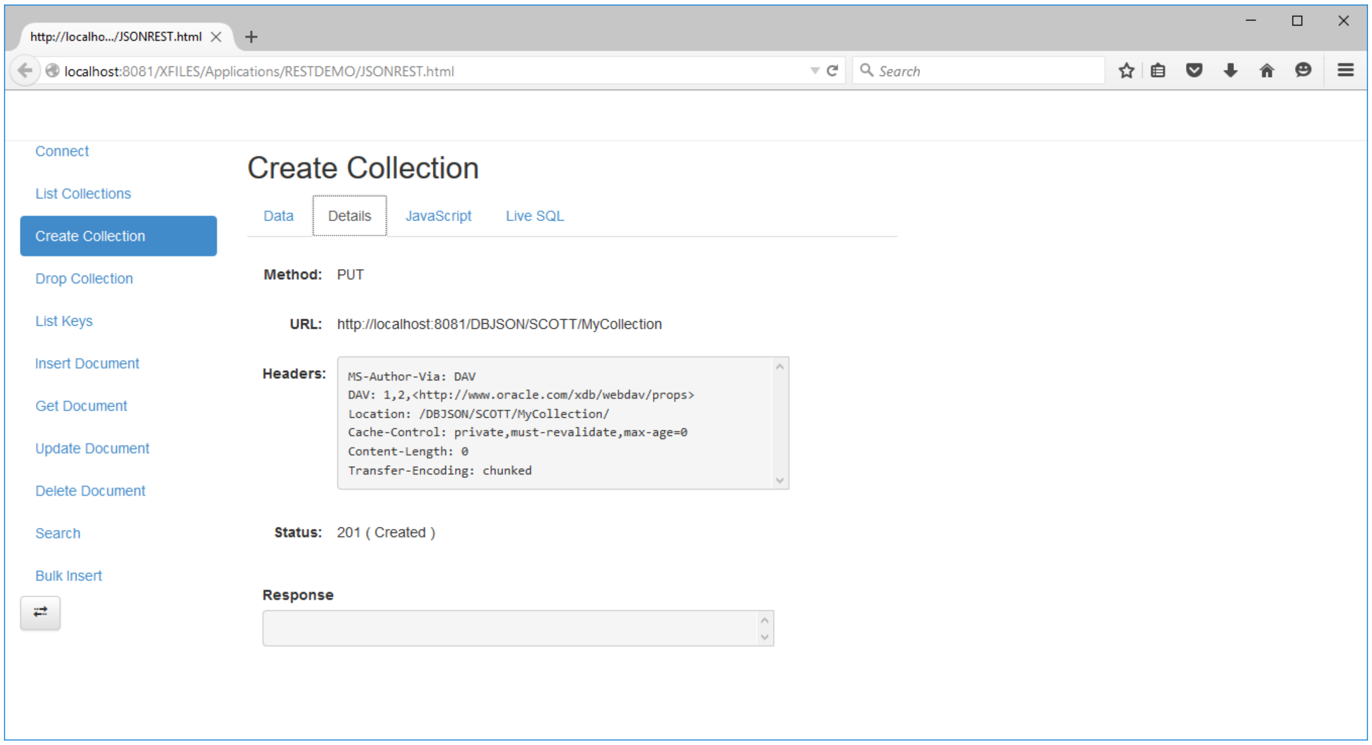
Click the Create Collection option to load the Create Collection form. The Create Collection form demonstrates the use of the SODA for REST Create Collection service. This service is used to create a new Collection.

To invoke the service enter the name of the collection to be created, for example MyCollection, and then click GO.



This will invoke the Create Collection service and create the specified collection.

Once the collection has been created switch to the Details tab to see information about the HTTP request used to invoke the Create Collection service.



Method: PUT

URL: http://localhost:8081/DBJSON/SCOTT/MyCollection

Headers:

```
MS-Author-Via: DAV
DAV: 1,2,<http://www.oracle.com/xdb/webdav/props>
Location: /DBJSON/SCOTT/MyCollection/
Cache-Control: private,must-revalidate,max-age=0
Content-Length: 0
Transfer-Encoding: chunked
```

Status: 201 (Created)

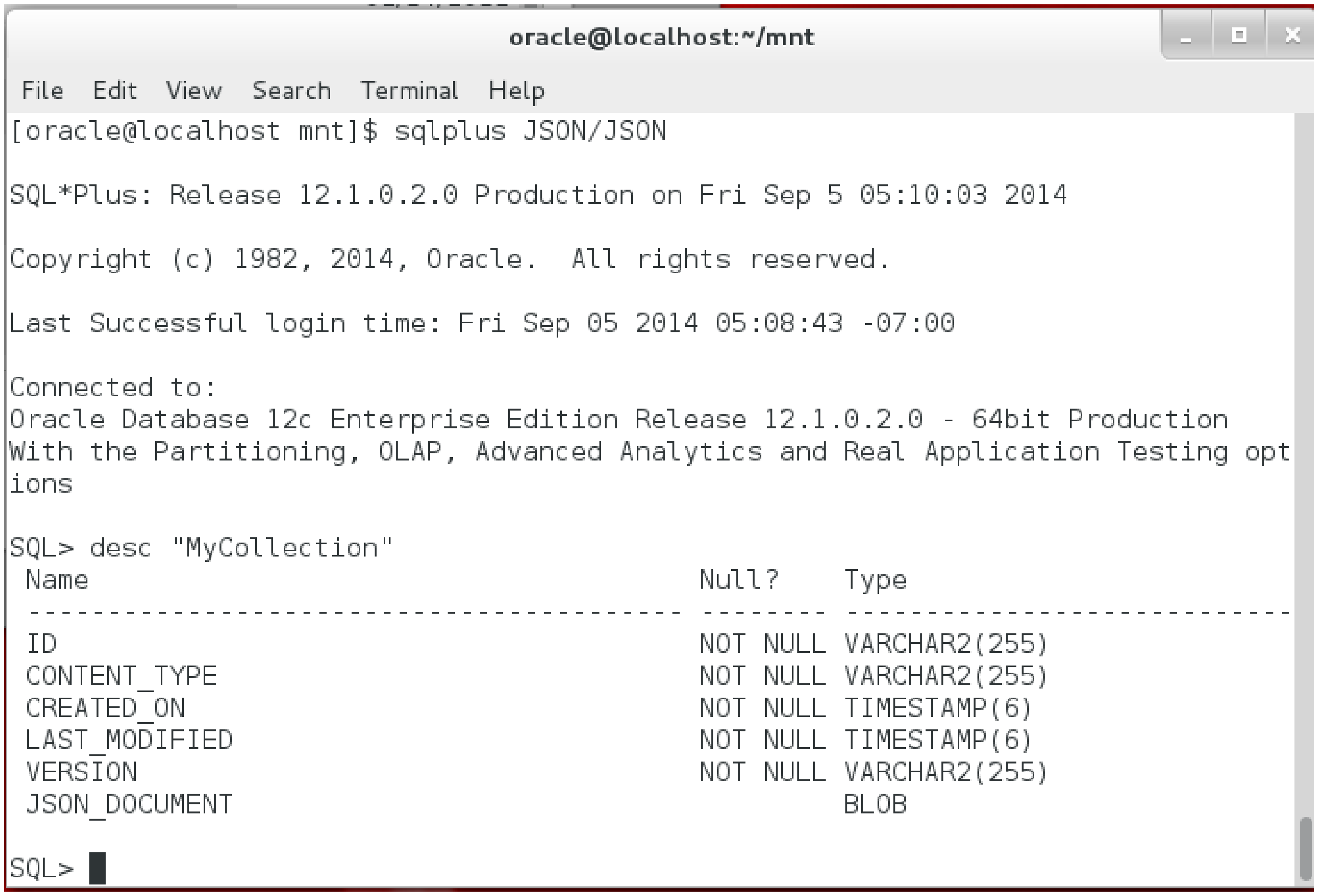
The Create Collection service was invoked by performing a PUT operation on the URL http://localhost:8081/DBJSON/SCOTT/MyCollection

The PUT operation returned status 201 indicating that the Create Collection service created a new collection object in the document store. If the specified collection was already present in the document the service returns status 200, rather than 201. The Create Collection service does not return a response document.

The header “Location” returns the URL that can be used to access the collection.

Switch to the JavaScript tab to see the JavaScript code that was used to execute the HTTP operation. To see a description of the table that was created by the Create Collection service switch to the Live SQL tab and click GO.

The details of the table that was created by the Create Collection service can also be seen using a SQL*PLUS “describe” command.

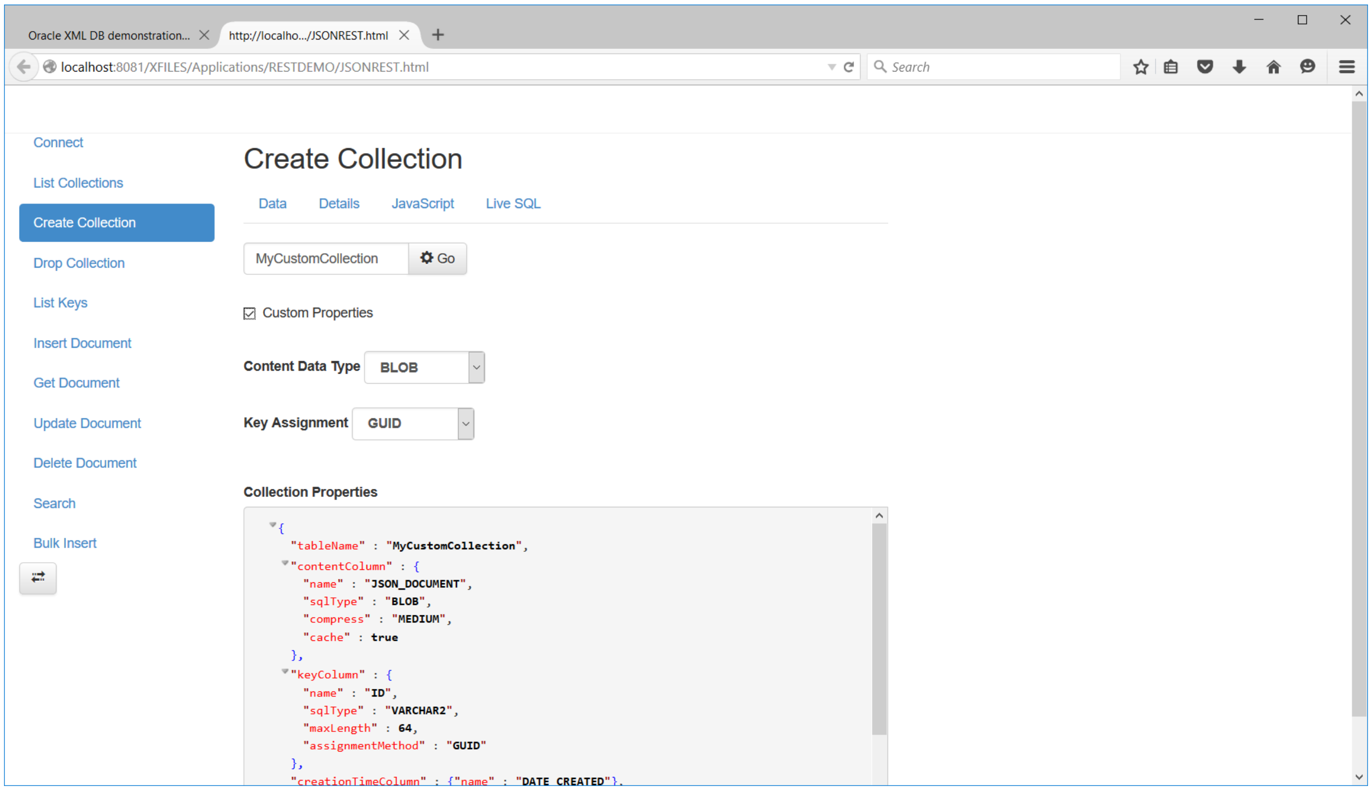


```
[oracle@localhost mnt]$ sqlplus JSON/JSON

SQL*Plus: Release 12.1.0.2.0 Production on Fri Sep 5 05:10:03 2014

Copyright (c) 1982, 2014, Oracle.  All rights reserved.

Last Successful login time: Fri Sep 05 2014 05:08:43 -07:00

Connected to:
Oracle Database 12c Enterprise Edition Release 12.1.0.2.0 - 64bit Production
With the Partitioning, OLAP, Advanced Analytics and Real Application Testing opt
ions

SQL> desc "MyCollection"
 Name                                       Null?    Type
 ----------------------------------------- -------- ----------------------------
 ID                                         NOT NULL VARCHAR2(255)
 CONTENT_TYPE                               NOT NULL VARCHAR2(255)
 CREATED_ON                                 NOT NULL TIMESTAMP(6)
 LAST_MODIFIED                              NOT NULL TIMESTAMP(6)
 VERSION                                    NOT NULL VARCHAR2(255)
 JSON_DOCUMENT                                       BLOB

SQL> 
```

Developers can control certain aspects of the table by supply a Collection Properties document to the Create Collection service. The document is supplied as the body of the PUT operation. The Collection Properties document can be used to specify a number of options, including

- The Table Name.
- The Name and Data Type of the column that contains the unique ID for each document.
- The Name and Data Type of the column that contains the content for each document.
- The Names of any housekeeping columns the system is expected to maintain. Housekeeping information includes content-type, date created, date last modified.
- The Name and algorithm used to track document version.
- How ID's are assigned (client or server) and the algorithm for generating them.
- The Drop policy for the collection.
- Whether the collection is read-only as far as the Document API's are concerned

Switch to the Data tab and enter a new collection name, e.g. MyCustomCollection. Click the Custom Properties checkbox to expose the Custom Properties form. The form shows an example of a Collection Properties document and provides a simple UI that can be used to manipulate some of content of the Collection Properties document.



Create Collection

MyCustomCollection

☑ Custom Properties

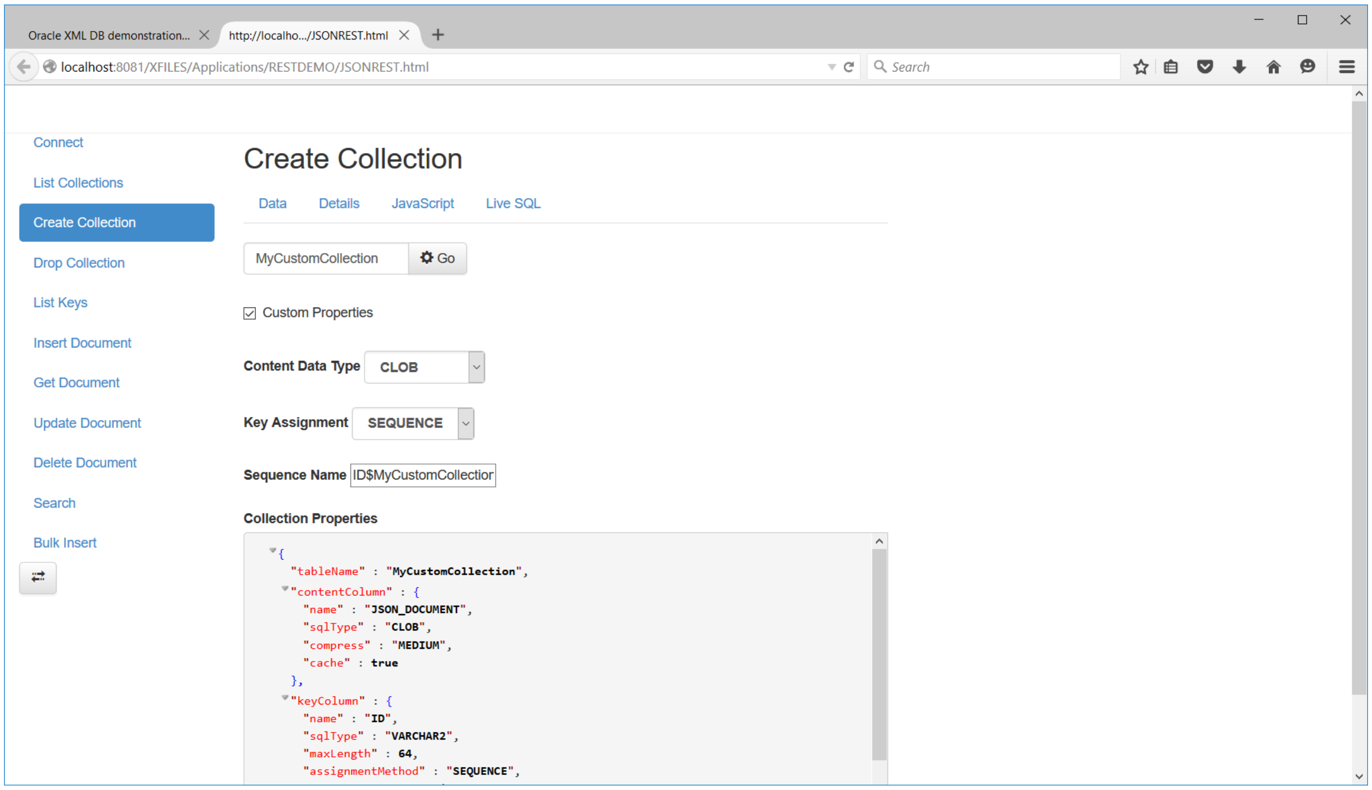
Content Data Type: BLOB

Key Assignment: GUID

Collection Properties

```
{
  "tableName" : "MyCustomCollection",
  "contentColumn" : {
    "name" : "JSON_DOCUMENT",
    "sqlType" : "BLOB",
    "compress" : "MEDIUM",
    "cache" : true
  },
  "keyColumn" : {
    "name" : "ID",
    "sqlType" : "VARCHAR2",
    "maxLength" : 64,
    "assignmentMethod" : "GUID"
  },
  "creationTimeColumn" : {"name" : "DATE_CREATED"},
```

The Custom Properties form provides controls that allow you to select the Data Type for the column that will store the JSON document and the algorithm that will be used to assign a key to each document in the collection. The collection properties document is modified whenever these controls are modified. Use the Content Data Type menu to store the JSON content using a CLOB, rather than a BLOB. Use the Key Assignment menu to use a SEQUENCE to assign IDs to the documents in the collection.



MyCustomCollection

☑ Custom Properties

Content Data Type: CLOB

Key Assignment: SEQUENCE

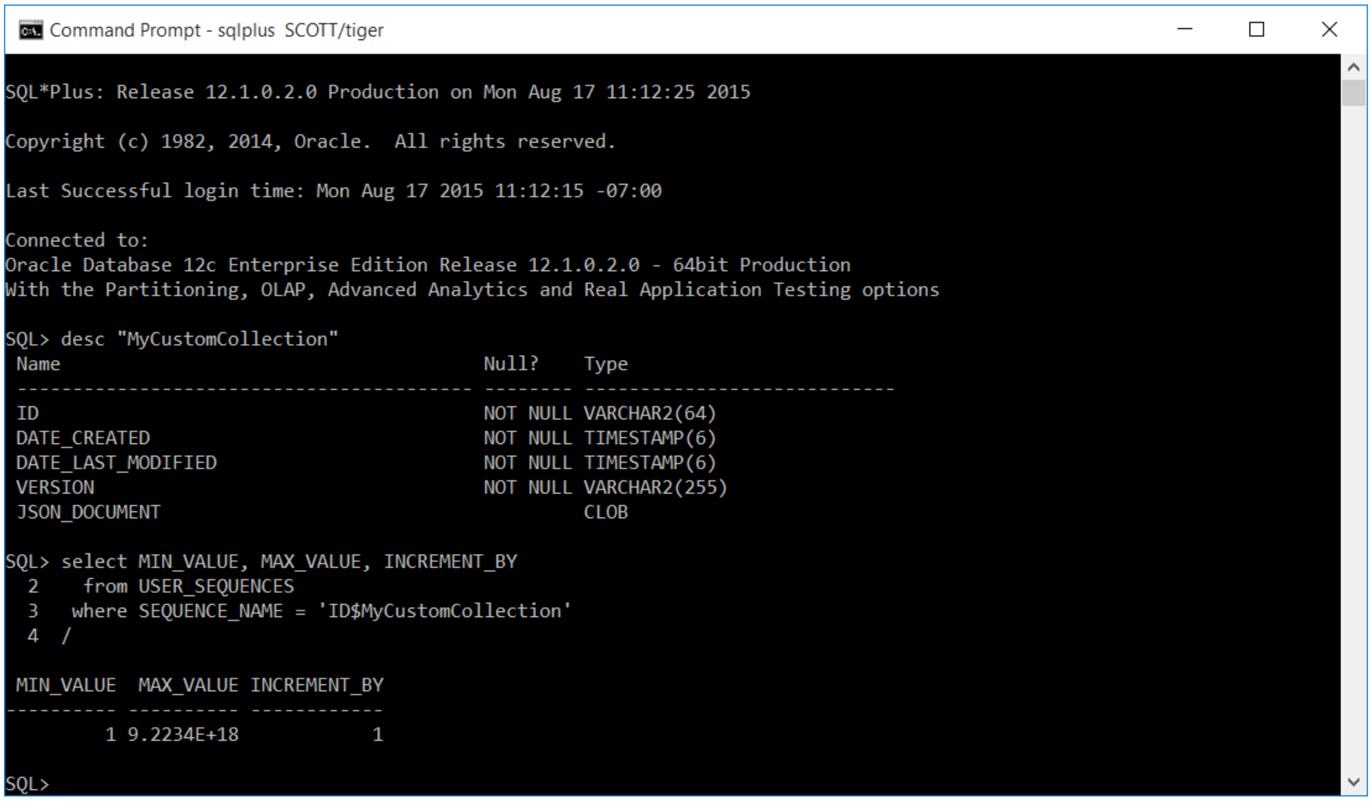
Sequence Name: ID$MyCustomCollectior

Collection Properties

```
{
  "tableName" : "MyCustomCollection",
  "contentColumn" : {
    "name" : "JSON_DOCUMENT",
    "sqlType" : "CLOB",
    "compress" : "MEDIUM",
    "cache" : true
  },
  "keyColumn" : {
    "name" : "ID",
    "sqlType" : "VARCHAR2",
    "maxLength" : 64,
    "assignmentMethod" : "SEQUENCE",
```

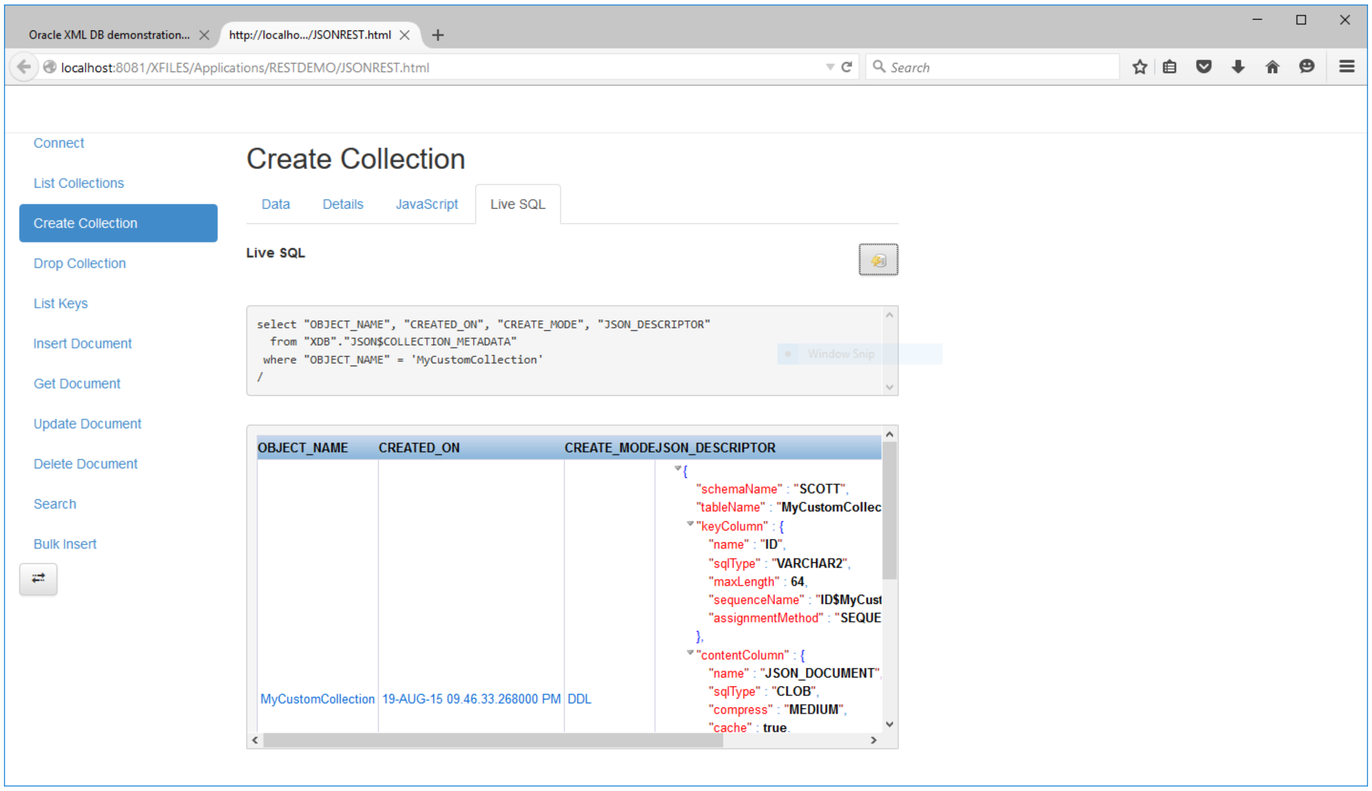
Once the custom settings have been set, click GO to create a collection based on the customized Collection Properties document.

Use SQL*Plus to confirm that your selections have been honored by the Create Collection service.



```
SQL*Plus: Release 12.1.0.2.0 Production on Mon Aug 17 11:12:25 2015

Copyright (c) 1982, 2014, Oracle.  All rights reserved.

Last Successful login time: Mon Aug 17 2015 11:12:15 -07:00

Connected to:
Oracle Database 12c Enterprise Edition Release 12.1.0.2.0 - 64bit Production
With the Partitioning, OLAP, Advanced Analytics and Real Application Testing options

SQL> desc "MyCustomCollection"
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ID                                        NOT NULL VARCHAR2(64)
 DATE_CREATED                              NOT NULL TIMESTAMP(6)
 DATE_LAST_MODIFIED                        NOT NULL TIMESTAMP(6)
 VERSION                                   NOT NULL VARCHAR2(255)
 JSON_DOCUMENT                                      CLOB

SQL> select MIN_VALUE, MAX_VALUE, INCREMENT_BY
  2    from USER_SEQUENCES
  3   where SEQUENCE_NAME = 'ID$MyCustomCollection'
  4  /

 MIN_VALUE  MAX_VALUE INCREMENT_BY
---------- ---------- ------------
         1 9.2234E+18            1

SQL>
```

Switch to the Live-SQL tab. The Live_SQL tab shows a statement that uses the system defined view XDB.JSON$COLLECTION_METADATA to obtain information about the newly created collection. Click GO to see the information available for collection “MyCustomCollection”



Create Collection

Live SQL

```
select "OBJECT_NAME", "CREATED_ON", "CREATE_MODE", "JSON_DESCRIPTOR"
  from "XDB"."JSON$COLLECTION_METADATA"
 where "OBJECT_NAME" = 'MyCustomCollection'
/
```

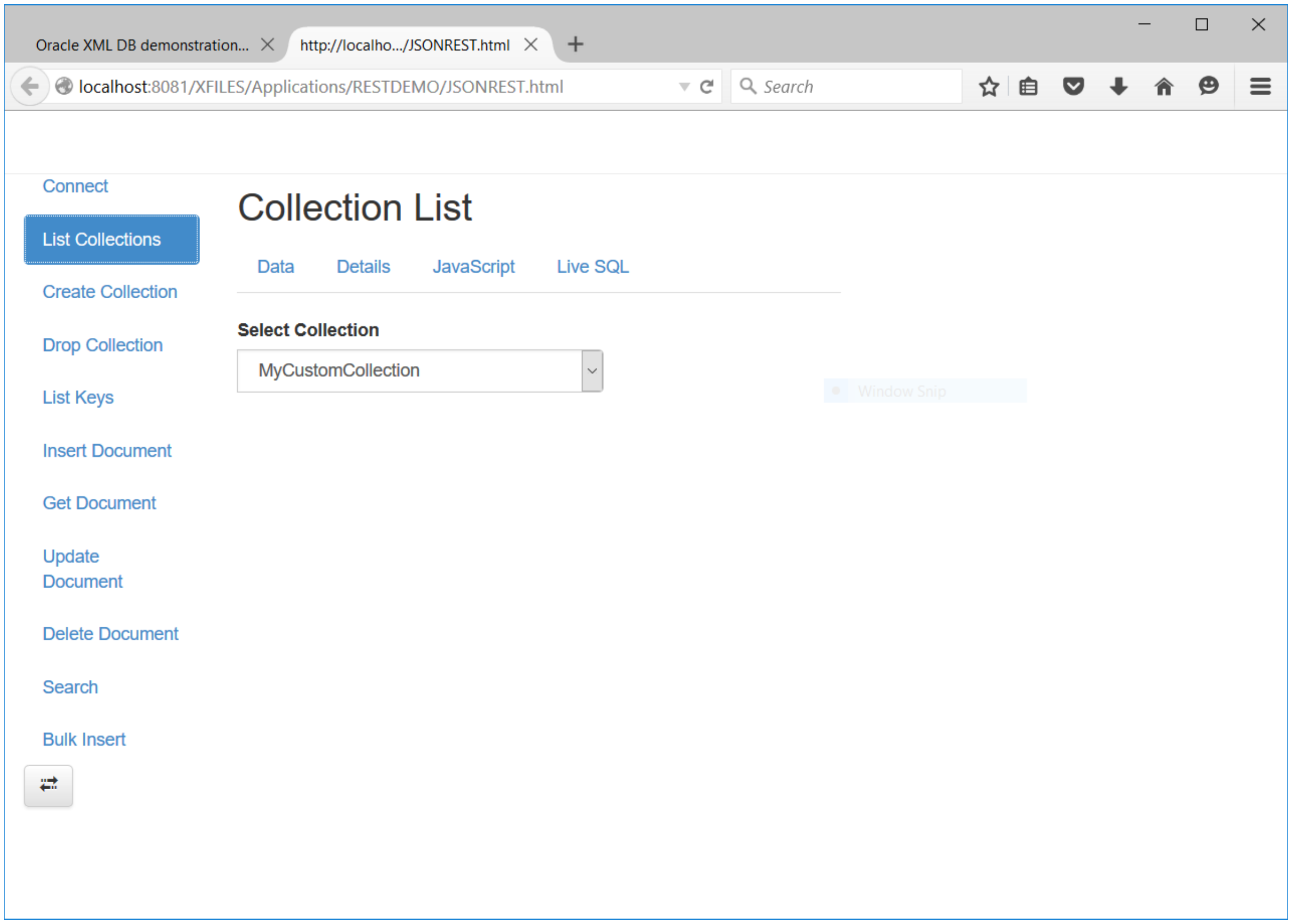
OBJECT_NAME	CREATED_ON	CREATE_MODE	JSON_DESCRIPTOR
MyCustomCollection	19-AUG-15 09.46.33.268000 PM	DDL	{ "schemaName" : "SCOTT", "tableName" : "MyCustomCollec "keyColumn" : { "name" : "ID", "sqlType" : "VARCHAR2", "maxLength" : 64, "sequenceName" : "ID$MyCust "assignmentMethod" : "SEQUE }, "contentColumn" : { "name" : "JSON_DOCUMENT", "sqlType" : "CLOB", "compress" : "MEDIUM", "cache" : true,

Hardware and Software
Engineered to Work Together

Step 3: Listing Collections

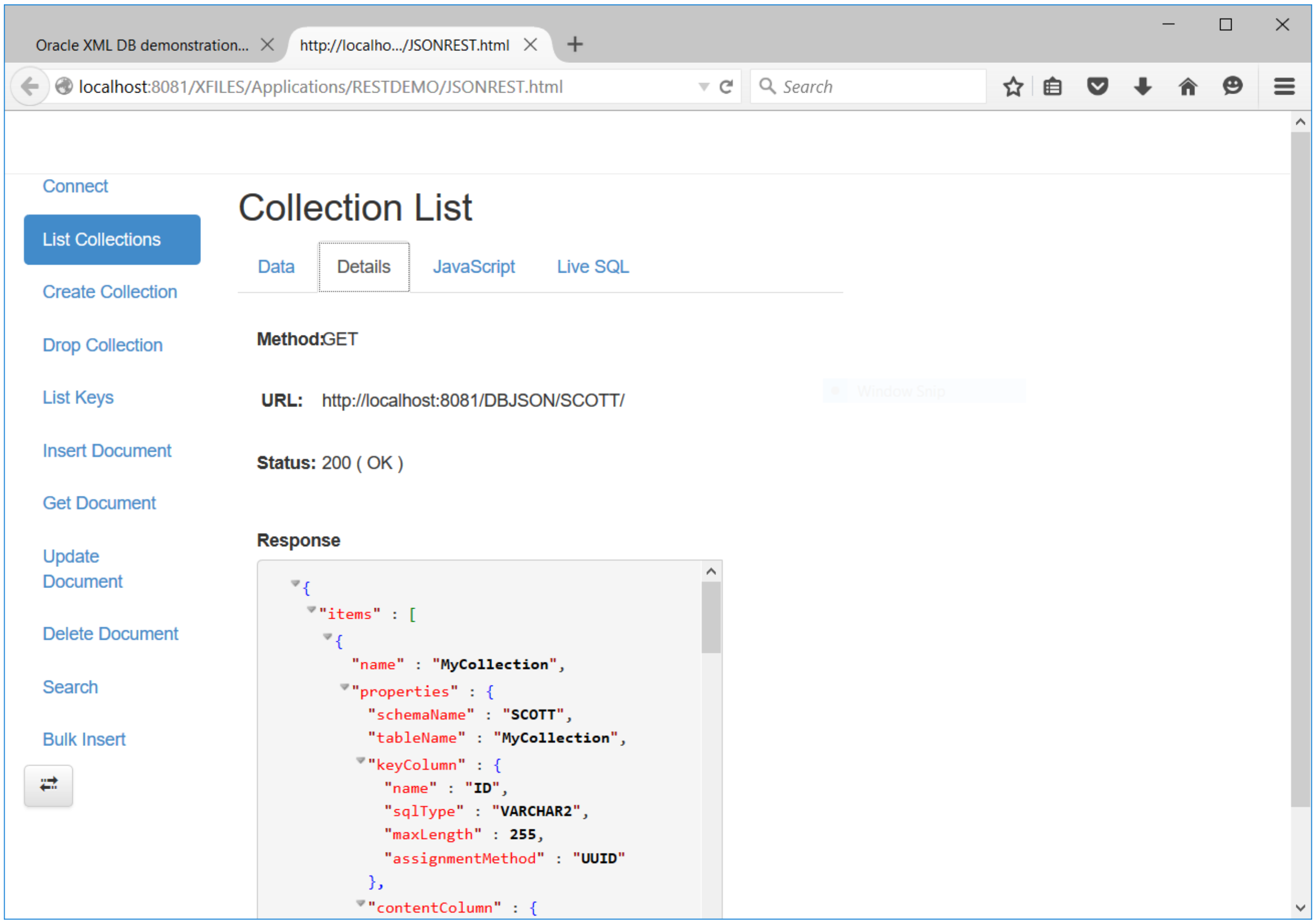
Click the List Collections option to switch to the Collection List form. This form demonstrates the use of the SODA for REST List Collection service. This service provides a list of the Document Store Collections available on the current database connection.

The form consists of the Select Collection menu which contains of a list of the collections that are available for the current document store connection. The menu is populated from the response document generated by invoking the List Collections service. The service is invoked automatically when the Collection List form is opened.



Note, if no collections are available the application automatically switches to the Create Collection form when the Select Collection option is selected.

Switch to the Details tab to see information about the HTTP request used to invoke the List Collection service.



Method:GET

URL: http://localhost:8081/DBJSON/SCOTT/

Status: 200 (OK)

Response

```
{
  "items" : [
    {
      "name" : "MyCollection",
      "properties" : {
        "schemaName" : "SCOTT",
        "tableName" : "MyCollection",
        "keyColumn" : {
          "name" : "ID",
          "sqlType" : "VARCHAR2",
          "maxLength" : 255,
          "assignmentMethod" : "UUID"
        },
        "contentColumn" : {
```

The List Collections service was invoked by performing a GET operation on the URL: http://localhost:8081/DBJSON/SCOTT.

The GET operation returned status 200 indicating that the List Collections service executed successfully. The response is a JSON document which consists of an array of Collection Descriptor documents. The array contains one member for each of the available Collections. In this example a simple JavaScript function is used to populate the Select Collection menu from the contents of the response.

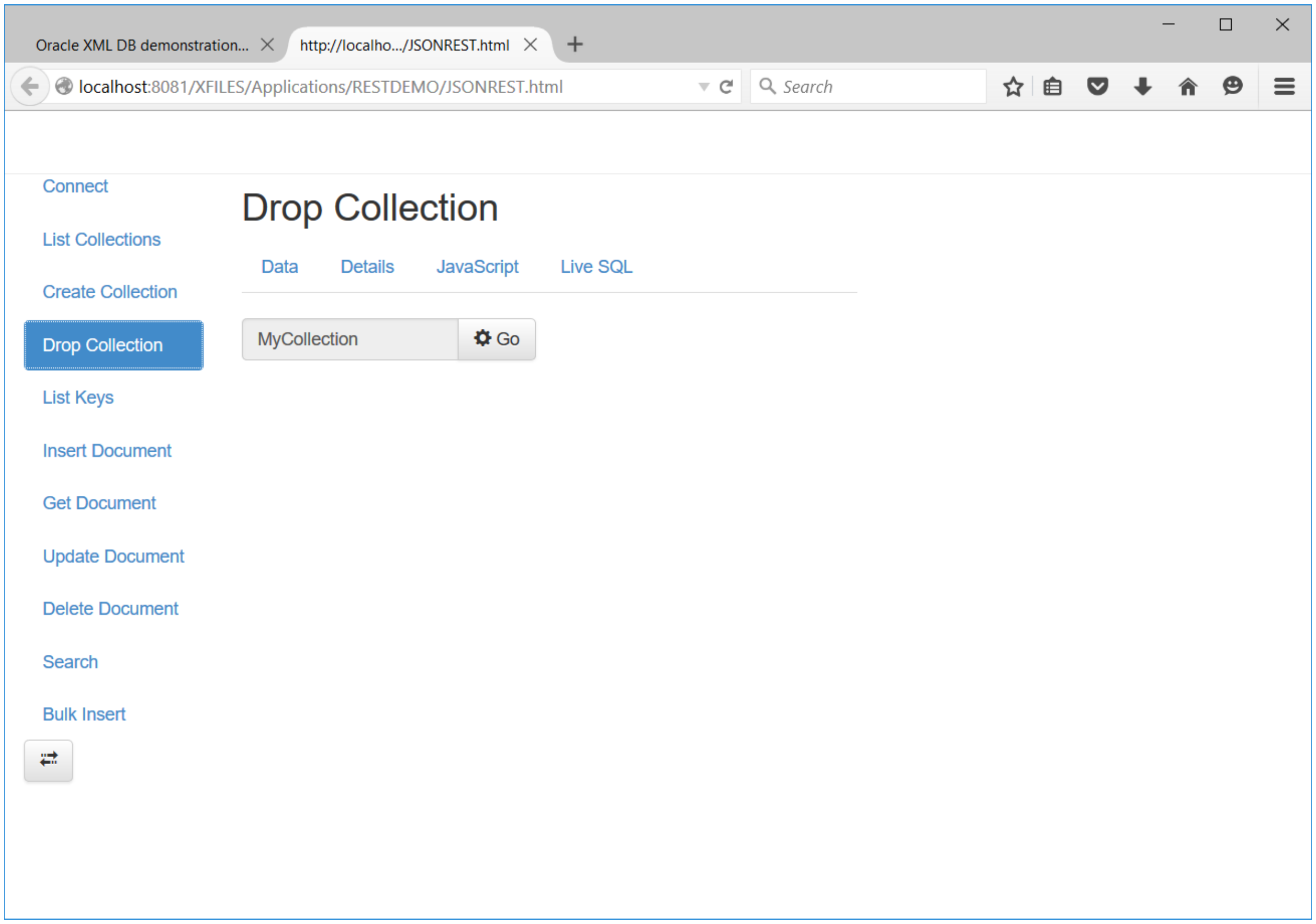
Switch to the JavaScript tab to see the JavaScript code that was used to execute the HTTP operation. Switch to the Live SQL tab to see and execute a SQL statement which shows how to use SQL to get the list of available collections by querying the view XDB.JSON$USER_COLLECTION_METADATA.

Step 4: Dropping Collections

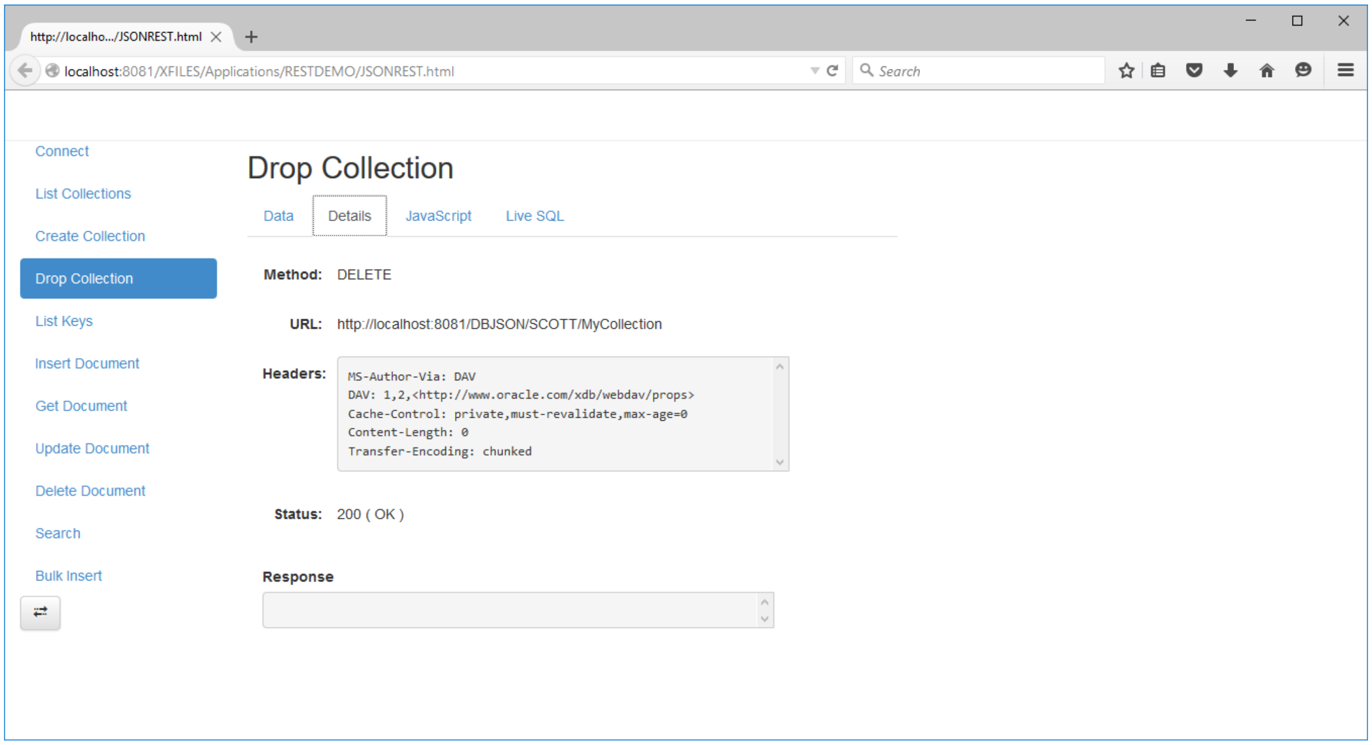
Click the Drop Collection option to switch to the Drop Collection form. This form demonstrates the use of the SODA for REST Drop Collection service. This service allows a collection to be dropped.

The form contains a single read-only field containing the name of the collection to be dropped. The field is automatically pre-populated with the name of the collection selected on the List Collections form.

Use the List Collections form to select the collection "MyCollection". Switch back to the Drop Collection form and click GO to drop the collection.



Switch to the Details tab to see information about the HTTP request used to invoke the Drop Collection service.



Method: DELETE

URL: http://localhost:8081/DBJSON/SCOTT/MyCollection

Headers:

```
MS-Author-Via: DAV
DAV: 1,2,<http://www.oracle.com/xdb/webdav/props>
Cache-Control: private,must-revalidate,max-age=0
Content-Length: 0
Transfer-Encoding: chunked
```

Status: 200 (OK)

The Drop Collection service was invoked by performing a DELETE operation on the URL: http://localhost:8081/DBJSON/SCOTT/MyCollection.

The DELETE operation returned status 200 indicating that the Drop Collection service successfully dropped the collection. The Drop Collection service does not return a response document.

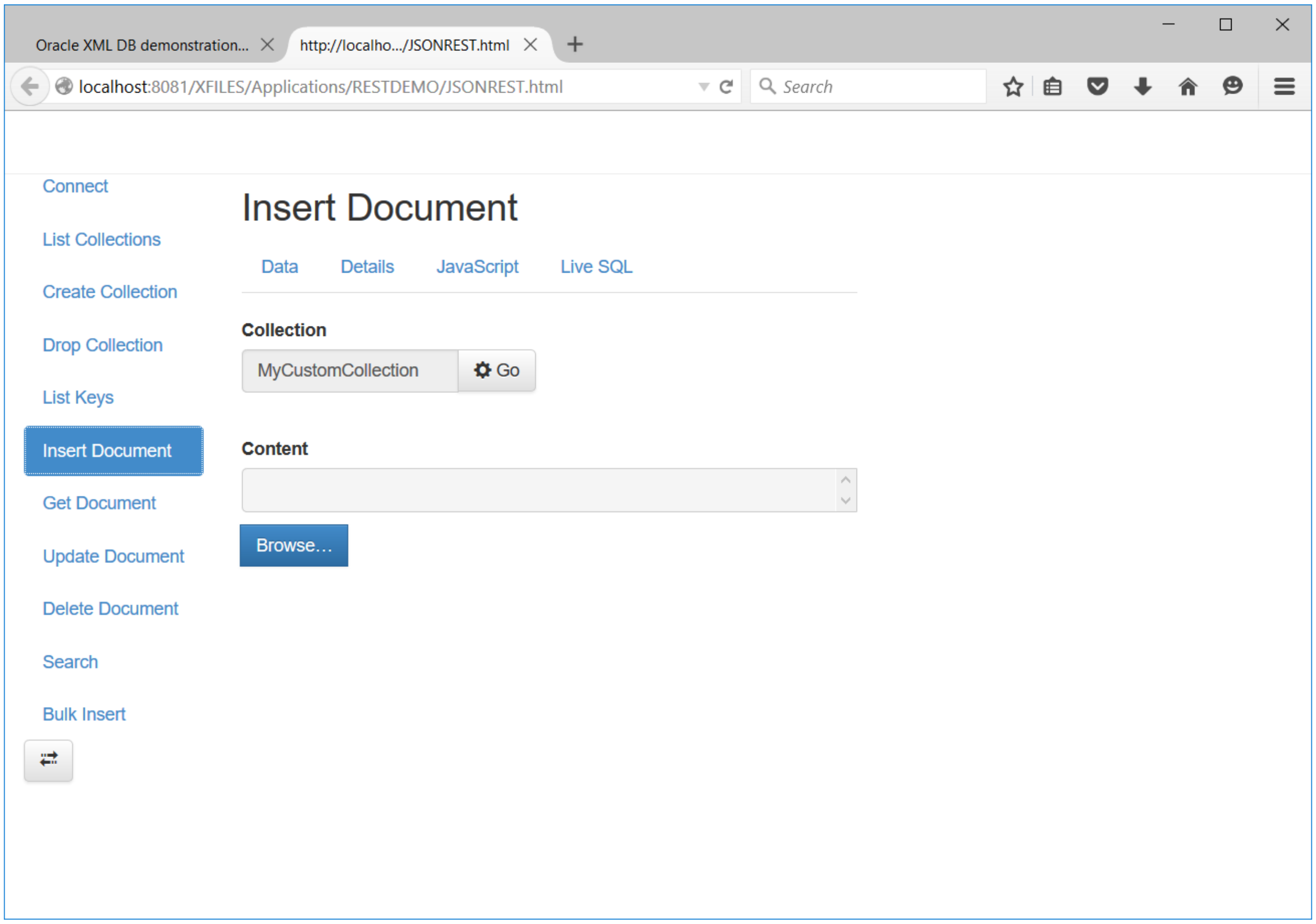
Switch to the JavaScript tab to see the JavaScript code that was used to execute the HTTP operation. To verify that the Drop Collection service successfully dropped the collection and the associated database object switch to the Live SQL tab and click GO.

Step 5: Inserting a Document

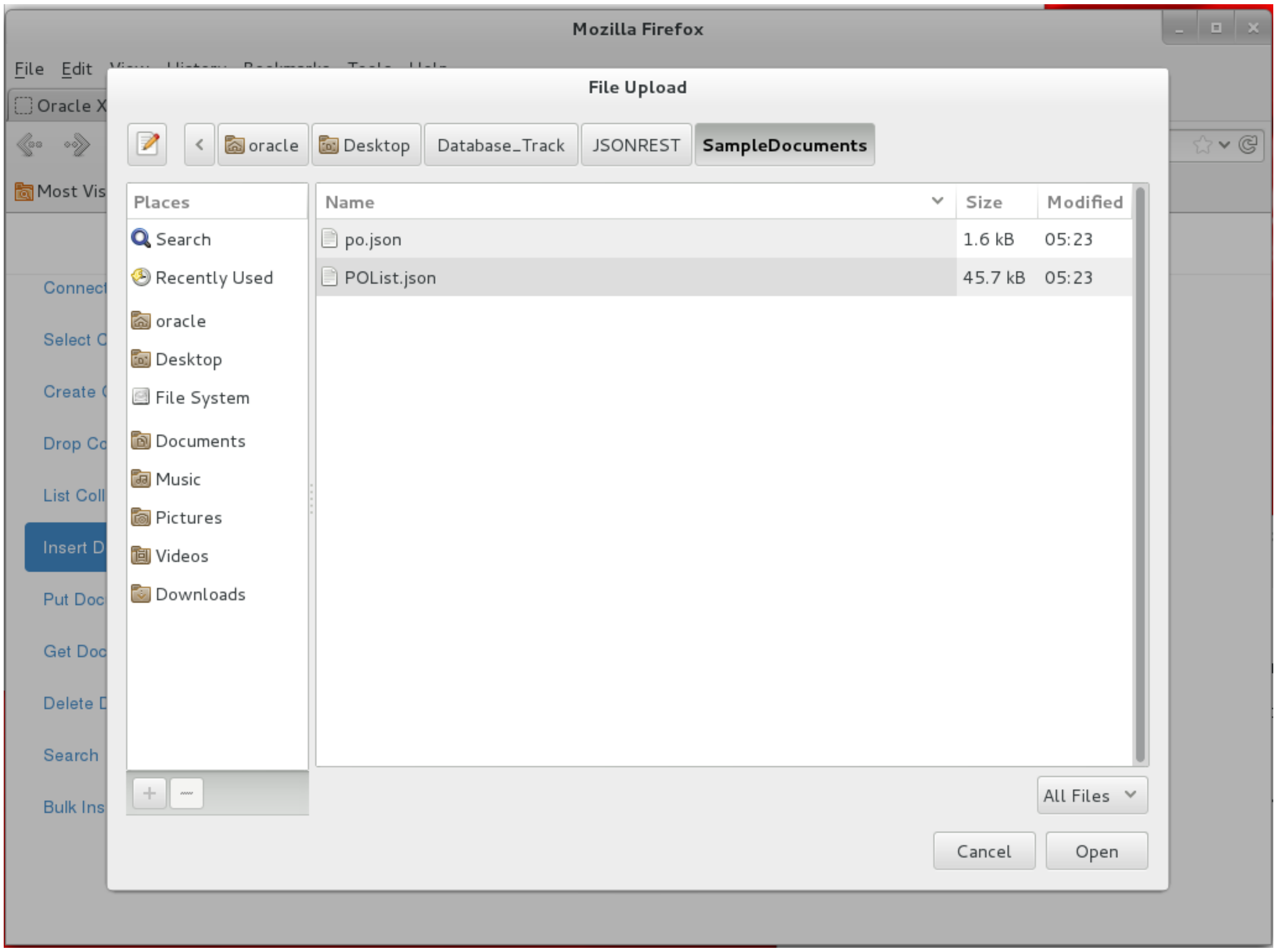
Click the Insert Document option to switch to the Insert Document form. This form demonstrates the use of the SODA for REST Insert Document service. This service allows a single document to be inserted into a collection.

The form contains two fields:

> Collection: This field is a read-only field containing the name of the collection that will be the target of the Insert operation. This field is automatically pre-populated with the name of the collection selected on the List Collections form.
>
> Content: The field is a read-only field containing the JSON content. This field is populated by uploading a document from the local file system.



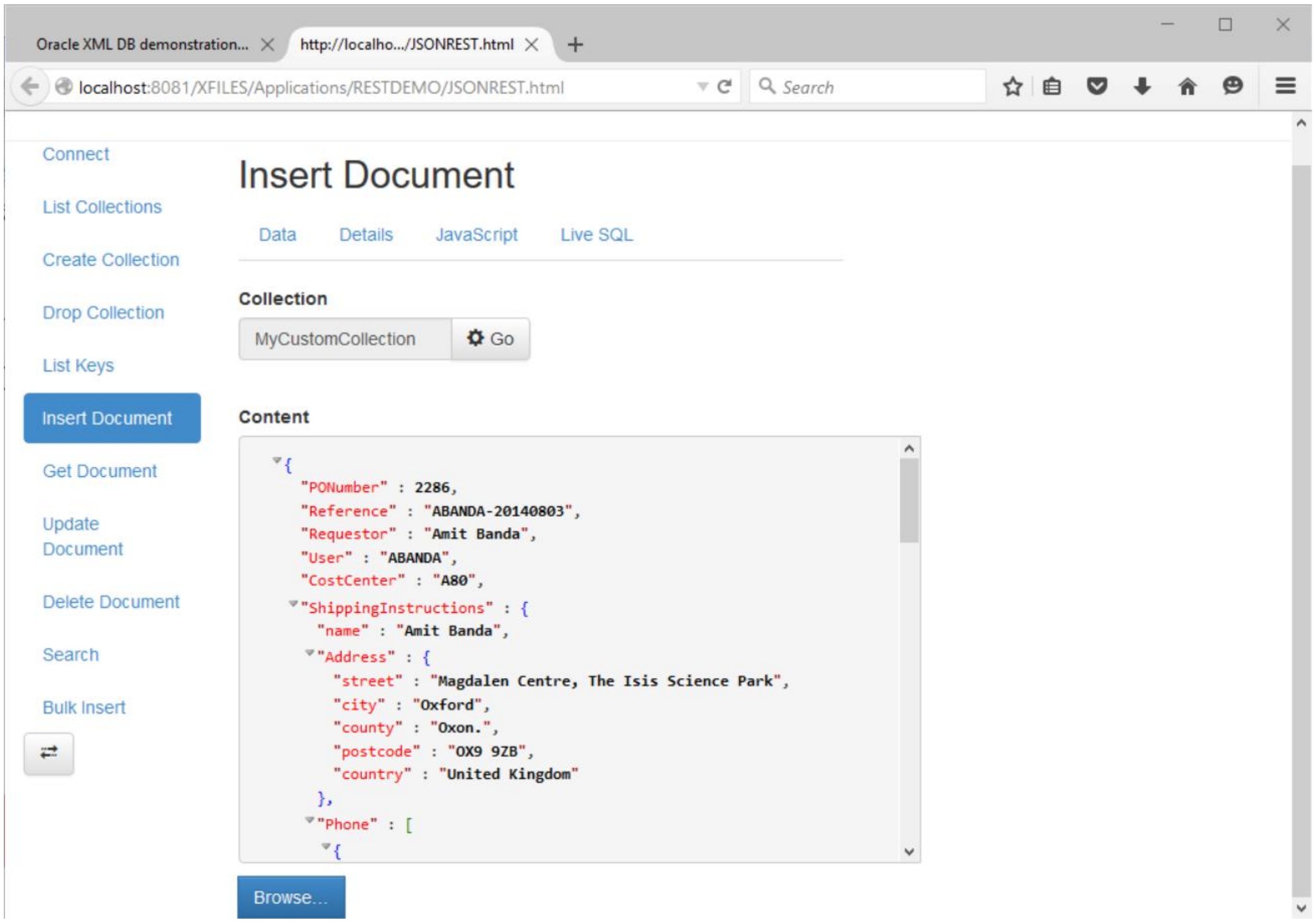
Use the List Collections form to select the collection “MyCustomCollection”. Switch back to the Insert Document form and use the Browse button to upload the file po.json which can be found in the SampleDocuments folder, located under /home/oracle/Desktop/Database_TrackJSON.



Name	Size	Modified
po.json	1.6 kB	05:23
POList.json	45.7 kB	05:23

Click Open to load the document.

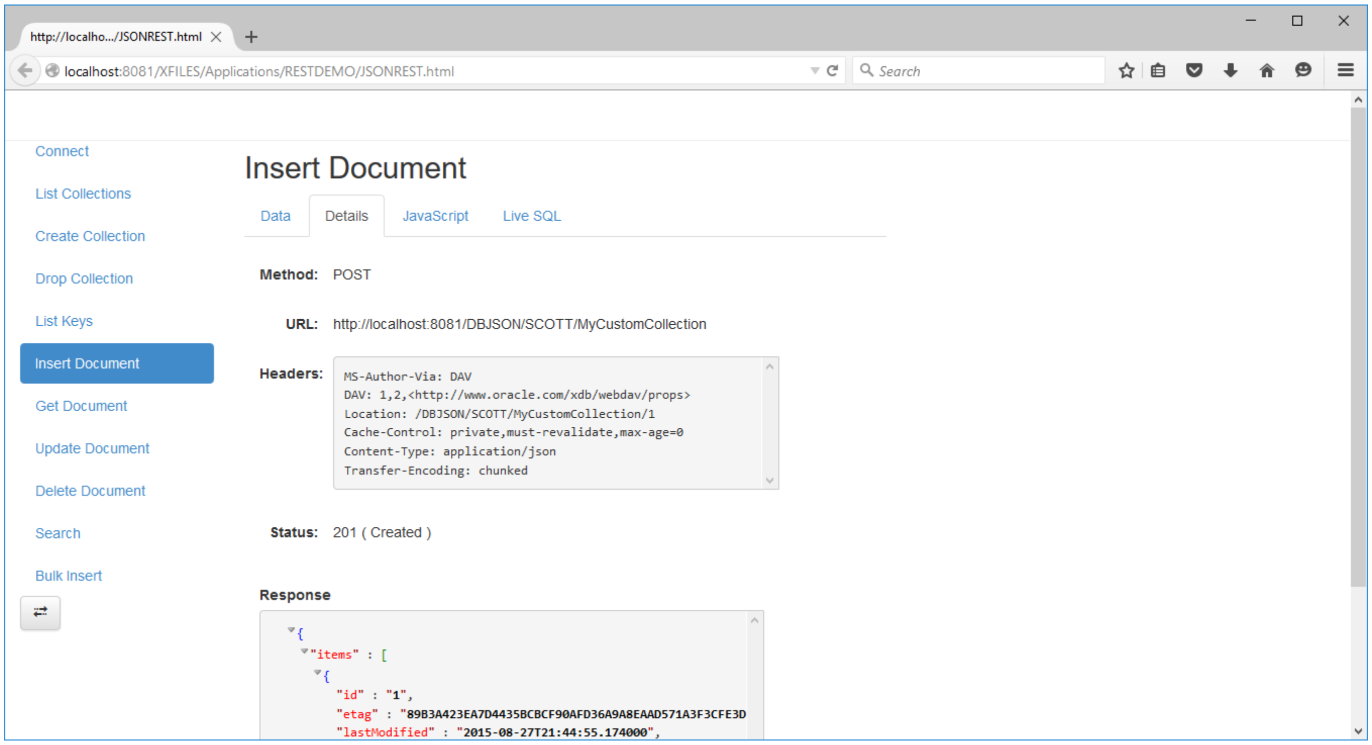
Once the document has been loaded the content will be displayed on the form.



```
{
  "PONumber" : 2286,
  "Reference" : "ABANDA-20140803",
  "Requestor" : "Amit Banda",
  "User" : "ABANDA",
  "CostCenter" : "A80",
  "ShippingInstructions" : {
    "name" : "Amit Banda",
    "Address" : {
      "street" : "Magdalen Centre, The Isis Science Park",
      "city" : "Oxford",
      "county" : "Oxon.",
      "postcode" : "OX9 9ZB",
      "country" : "United Kingdom"
    },
    "Phone" : [
      {
```

Click GO to insert the document.

Switch to the Details tab to see information about the HTTP request used to invoke the Insert Document service.



Insert Document

Method: POST

URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection

Headers:

```
MS-Author-Via: DAV
DAV: 1,2,<http://www.oracle.com/xdb/webdav/props>
Location: /DBJSON/SCOTT/MyCustomCollection/1
Cache-Control: private,must-revalidate,max-age=0
Content-Type: application/json
Transfer-Encoding: chunked
```

Status: 201 (Created)

Response

```
{
  "items" : [
    {
      "id" : "1",
      "etag" : "89B3A423EA7D4435BCBCF90AFD36A9A8EAAD571A3F3CFE3D
      "lastModified" : "2015-08-27T21:44:55.174000",
```

The Insert Document service was invoked by performing a POST operation on the URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection. The JSON document was supplied as the body of the POST request.

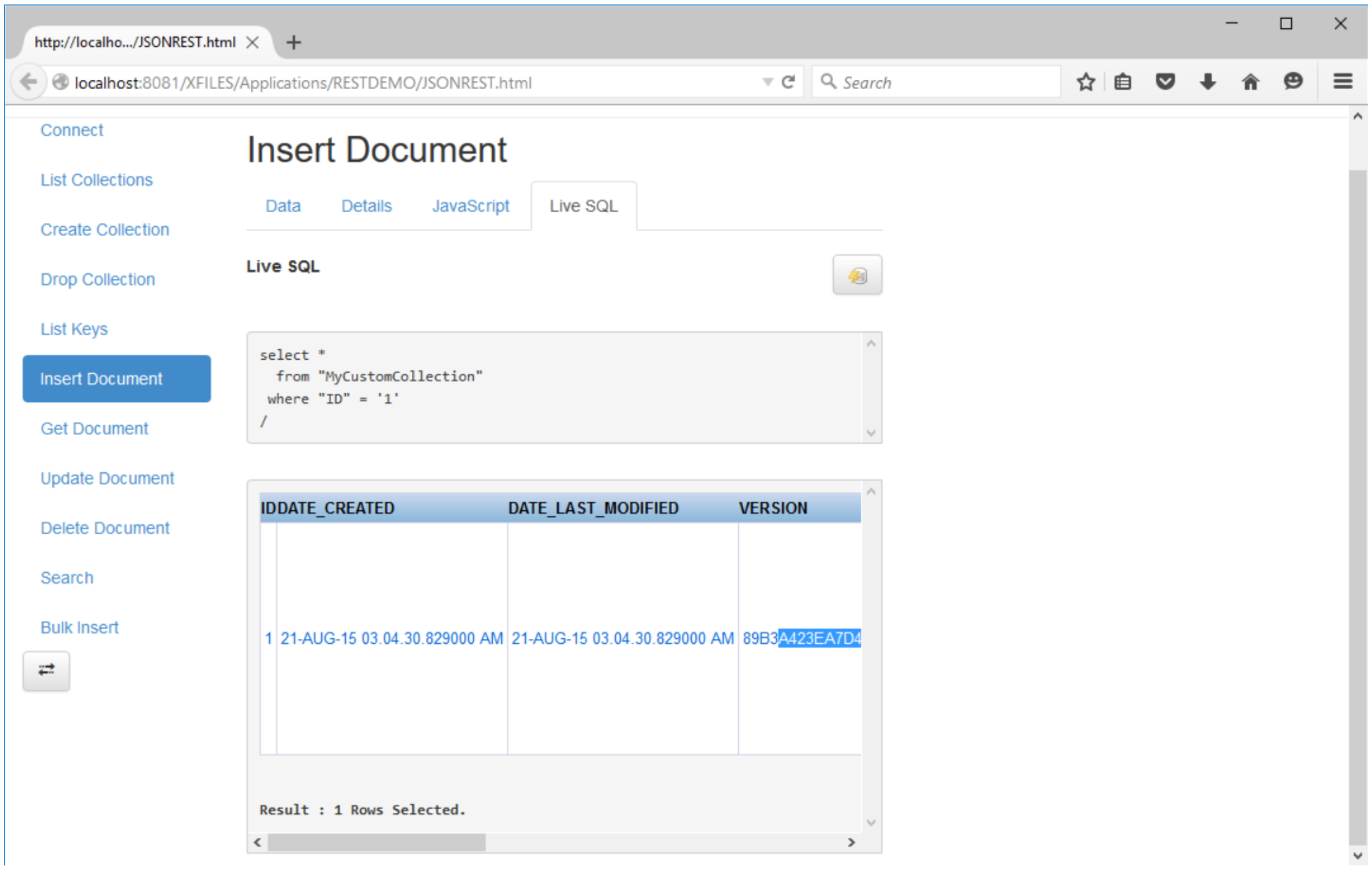
The POST operation returned status 201 indicating that the Insert Document service successfully inserted the document into the collection. The response provided by the Insert Document service consists of a JSON document containing the id that was assigned by the server, an ETAG value and creation and modification timestamps

The header location contains a URL that can be used to access the newly created document.

By default, the Oracle REST API defaults to server assigned keys. This means when document is added to a collection its key is assigned by the server as part of the insert operation. For “MyCustomCollection” the ID is generated using a database sequence.

Switch to the JavaScript tab to see the JavaScript code that was used to execute the HTTP operation.

To verify that the Insert service successfully inserted the document into the collection switch to the Live SQL tab and click GO



Insert Document

Live SQL

```
select *
  from "MyCustomCollection"
 where "ID" = '1'
/
```

ID	DATE_CREATED	DATE_LAST_MODIFIED	VERSION
1	21-AUG-15 03.04.30.829000 AM	21-AUG-15 03.04.30.829000 AM	89B3A423EA7D4

```
Result : 1 Rows Selected.
```

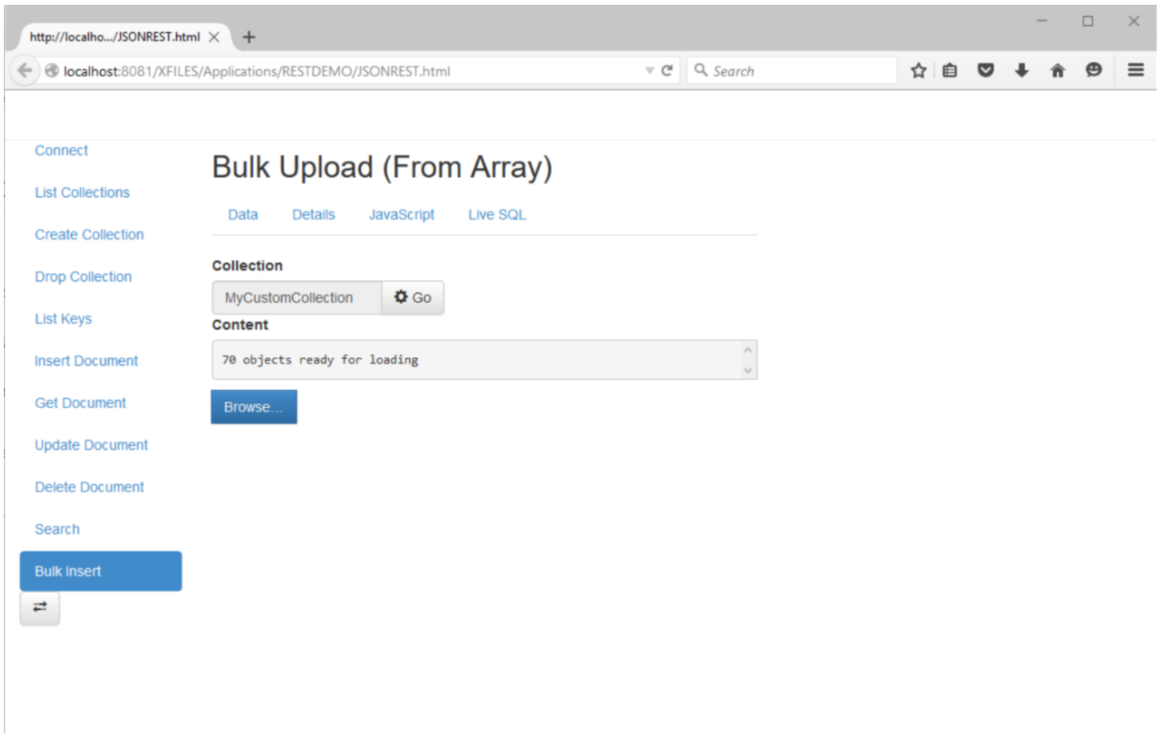
Step 6: Bulk Insert Operations

Click the Bulk Insert option to switch to the Bulk Upload form. This form demonstrates the use of the SODA for REST Bulk Insert service. This service allows a set of documents to be created from a JSON array using a single round-trip.

The form contains two fields:

> Collection: This field is a read-only field containing the name of the collection that will be the target of the Insert operation. This field is automatically pre-populated with the name of the collection selected on the List Collections form.
>
> Content: The field is a read-only field containing the JSON content. This field is populated by uploading a document from the local file system.

Use the List Collections form to select the collection "MyCustomCollection". Switch back to the Bulk Insert form and use the Browse button to upload the file poList.json which can be found in the SampleDocuments folder, located under /home/oracle/Desktop/Database_TrackJSON. This file contains a JSON array with70 PurchaseOrder objects. Once the document has been uploaded the Content field will show a message indicating the number of objects in the array.

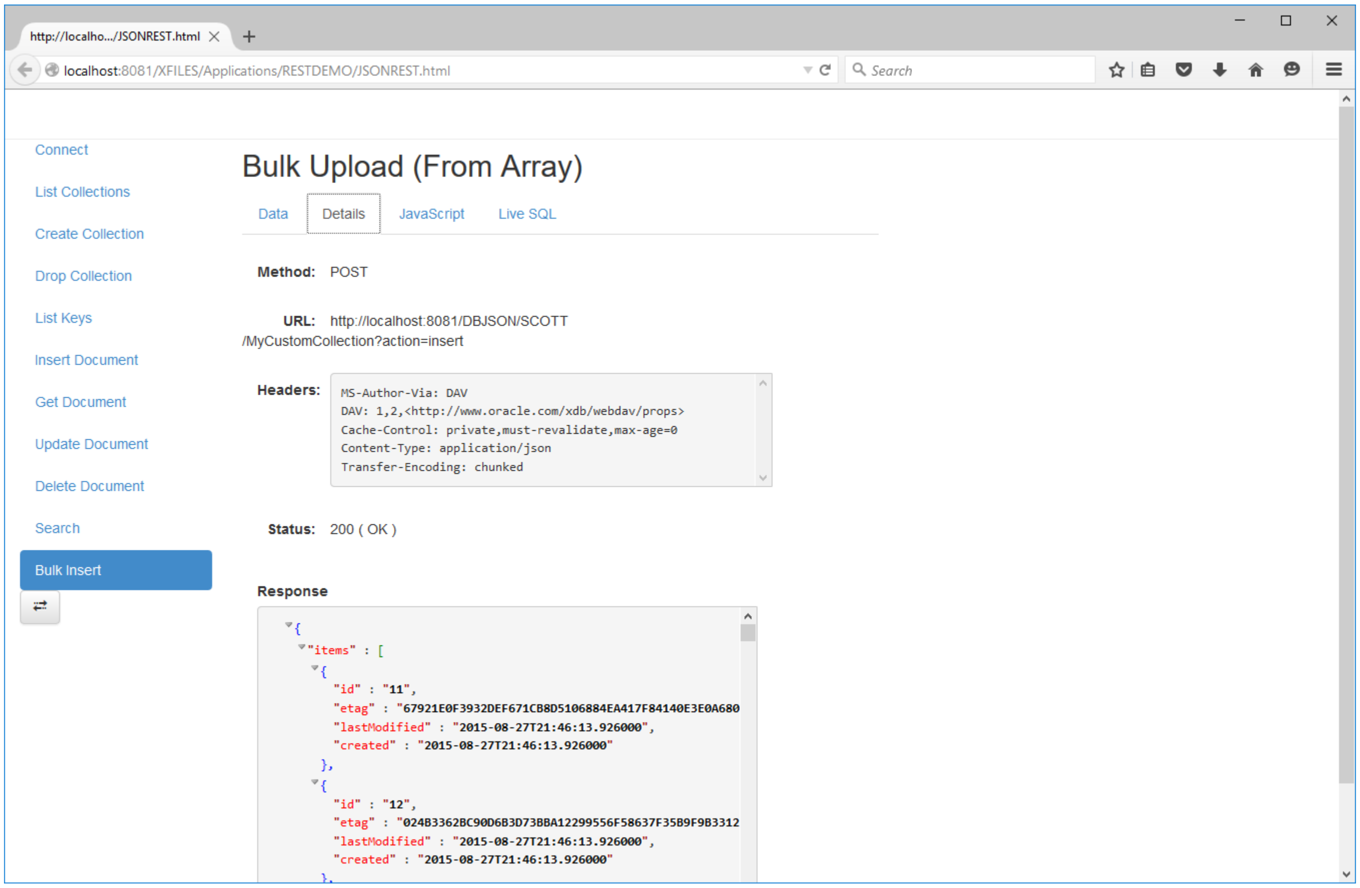


Collection: MyCustomCollection

Content: 70 objects ready for loading

Clicks GO to upload the document.

The form will respond with the message “Bulk Load Successful: 70 Documents loaded”. Dismiss the message and switch to the Details tab to see information about the HTTP request used to invoke the Bulk Insert service.



Bulk Upload (From Array)

Method: POST

URL: http://localhost:8081/DBJSON/SCOTT
/MyCustomCollection?action=insert

Headers:

```
MS-Author-Via: DAV
DAV: 1,2,<http://www.oracle.com/xdb/webdav/props>
Cache-Control: private,must-revalidate,max-age=0
Content-Type: application/json
Transfer-Encoding: chunked
```

Status: 200 (OK)

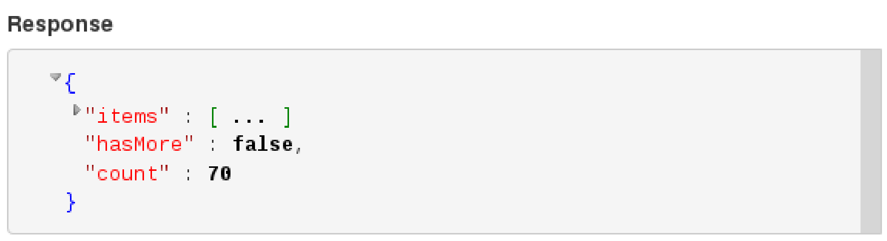
Response

```
{
  "items" : [
    {
      "id" : "11",
      "etag" : "67921E0F3932DEF671CB8D5106884EA417F84140E3E0A680
      "lastModified" : "2015-08-27T21:46:13.926000",
      "created" : "2015-08-27T21:46:13.926000"
    },
    {
      "id" : "12",
      "etag" : "024B3362BC90D6B3D73BBA12299556F58637F35B9F9B3312
      "lastModified" : "2015-08-27T21:46:13.926000",
      "created" : "2015-08-27T21:46:13.926000"
    },
```

The Bulk Insert service was invoked by performing a POST operation on the URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection?action=insert. The JSON document was supplied as the body of the POST request. Adding the query string **action=insert** causes the SODA for REST bulk insert to be invoked, which creates a separate document for each member of the array.

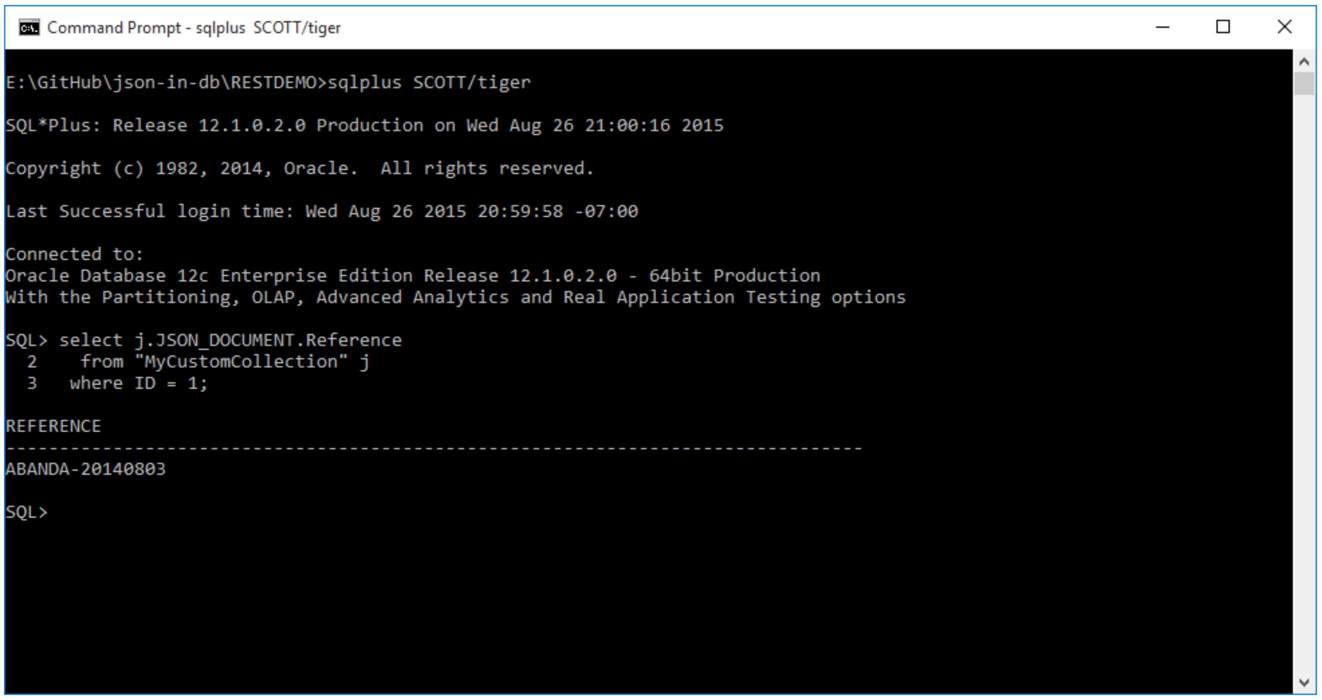
The POST operation returned status 200 indicating that the Bulk Insert service successfully processed the contents of the document. The response provided by the Insert Document service consists of a JSON document containing an array called “items” that provides information about the documents that were created from the array. The array contains one member for each document created. Each member of the array provides id, ETAG and timestamp information for one document.

The response also contains a count of the number of documents that were created. This can be seen by collapsing the items array.



```
{
  "items" : [ ... ]
  "hasMore" : false,
  "count" : 70
}
```

Highlight one of the ids's returned by Bulk Insert and copy it to the clipboard. Open a command prompt and use SQL*PLUS to query the collection using the selected ID.

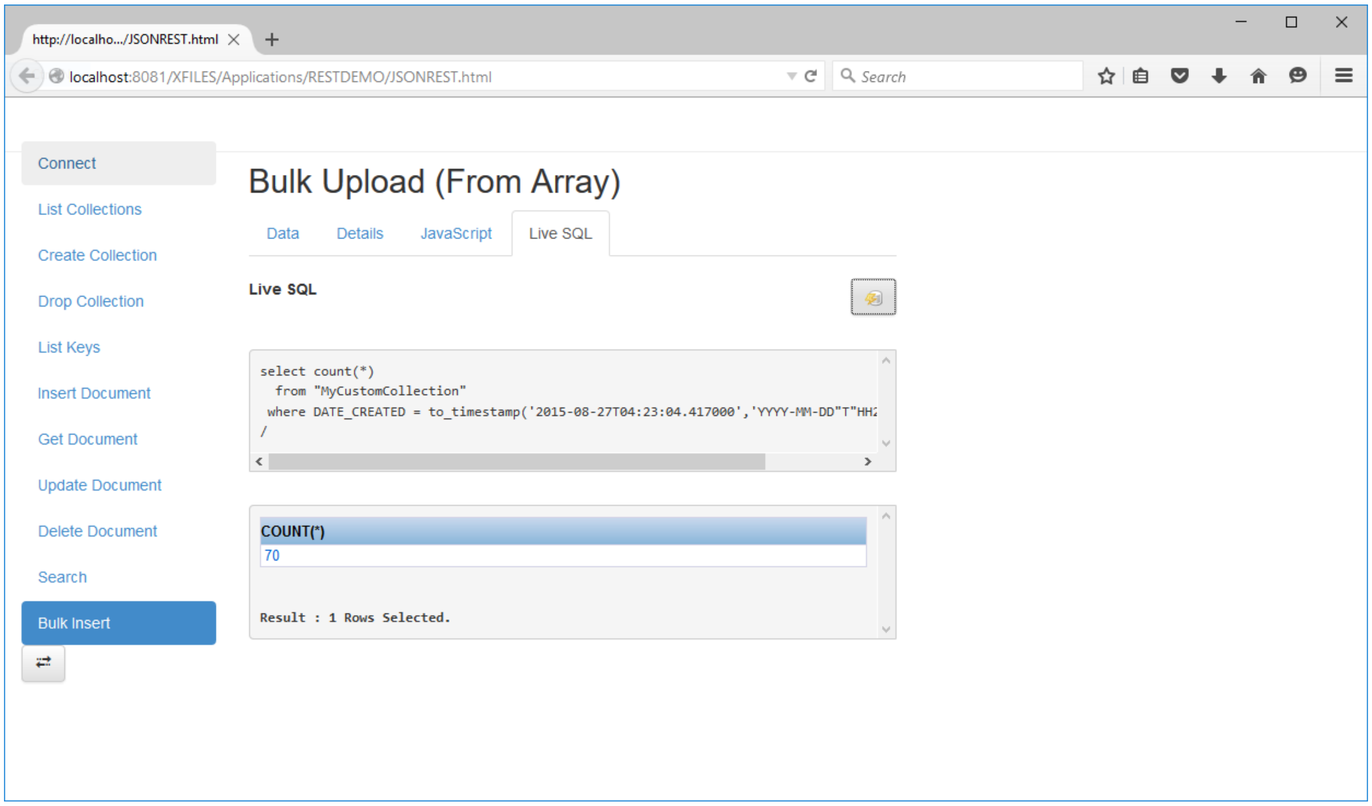


```
E:\GitHub\json-in-db\RESTDEMO>sqlplus SCOTT/tiger

SQL*Plus: Release 12.1.0.2.0 Production on Wed Aug 26 21:00:16 2015

Copyright (c) 1982, 2014, Oracle.  All rights reserved.

Last Successful login time: Wed Aug 26 2015 20:59:58 -07:00

Connected to:
Oracle Database 12c Enterprise Edition Release 12.1.0.2.0 - 64bit Production
With the Partitioning, OLAP, Advanced Analytics and Real Application Testing options

SQL> select j.JSON_DOCUMENT.Reference
  2    from "MyCustomCollection" j
  3   where ID = 1;

REFERENCE
--------------------------------------------------------------------------------
ABANDA-20140803

SQL>
```

Note that the table name has to be provided as a quoted identifier in SQL, since it was given as a mixed case name when the collection was created.

Switch to the JavaScript tab to see the JavaScript code that was used to execute the HTTP operation.

To verify that the Bulk Insert service successfully inserted all documents into the collection switch to the Live SQL tab and click GO



Bulk Upload (From Array)

Live SQL

```
select count(*)
  from "MyCustomCollection"
 where DATE_CREATED = to_timestamp('2015-08-27T04:23:04.417000','YYYY-MM-DD"T"HH2
/
```

COUNT(*)
70

```
Result : 1 Rows Selected.
```

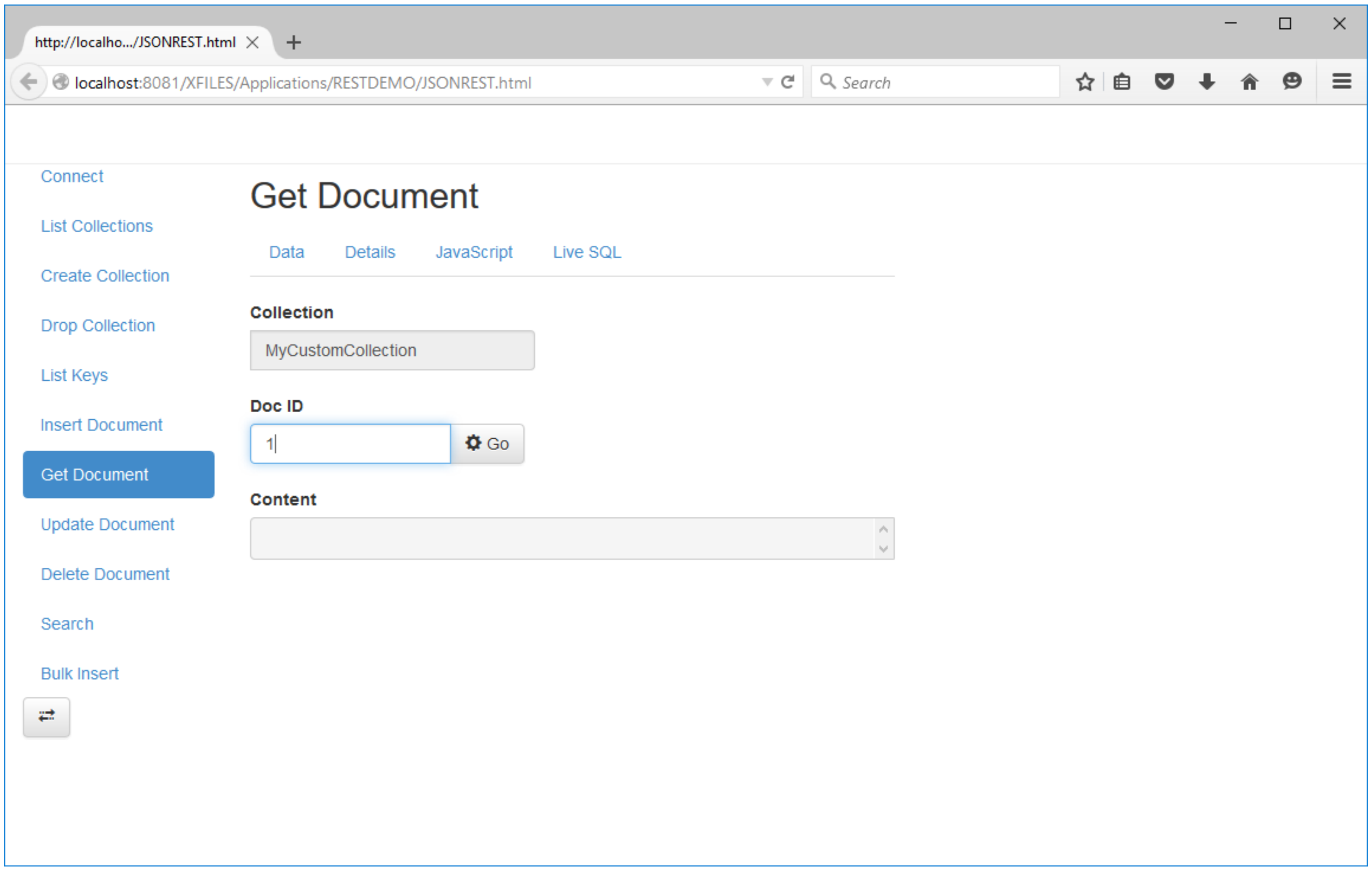
Step 7: Retrieving a Document

Click the Get Document option to switch to the Get Document form. This form demonstrates the use of the SODA for REST Get Document service. This service allows a document to be retrieved using its ID.

The form contains two fields:

> Collection: This field is a read-only field containing the name of the collection that will be the target of the Get operation. This field is automatically pre-populated with the name of the collection selected on the List Collections form.
>
> Document Id: This field is used to specify the unique ID of the document be fetched.

Use the List Collections form to select the collection "MyCustomCollection". Switch back to the Get Document form and enter a known document id or paste the id copied from the response document returned by the Bulk Insert operation into the Document ID field.



Get Document

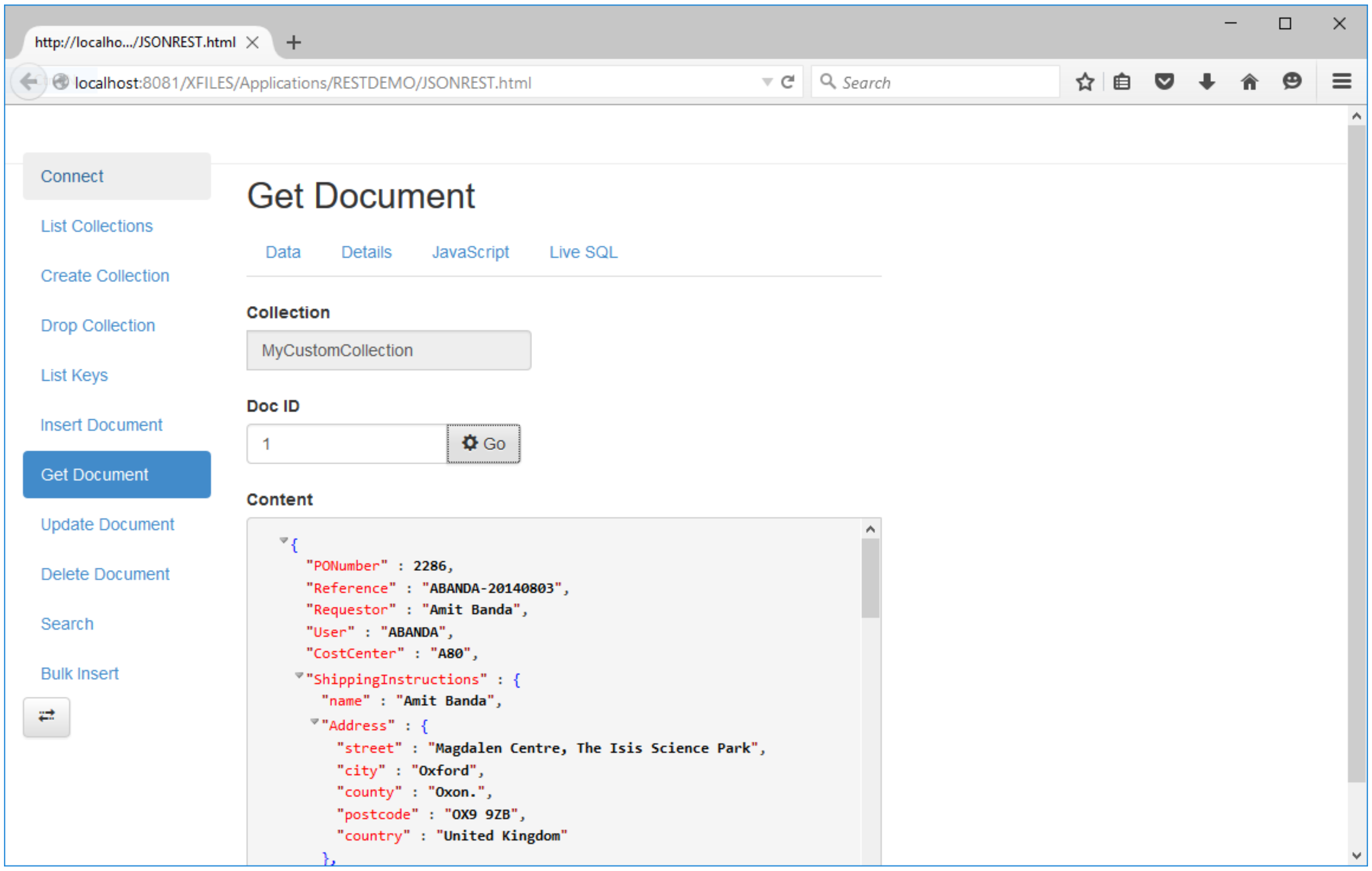
Collection: MyCustomCollection

Doc ID: 1

Content

Click GO to retrieve the document.

The document corresponding to the specified ID will be displayed.



Get Document

Collection: MyCustomCollection

Doc ID: 1

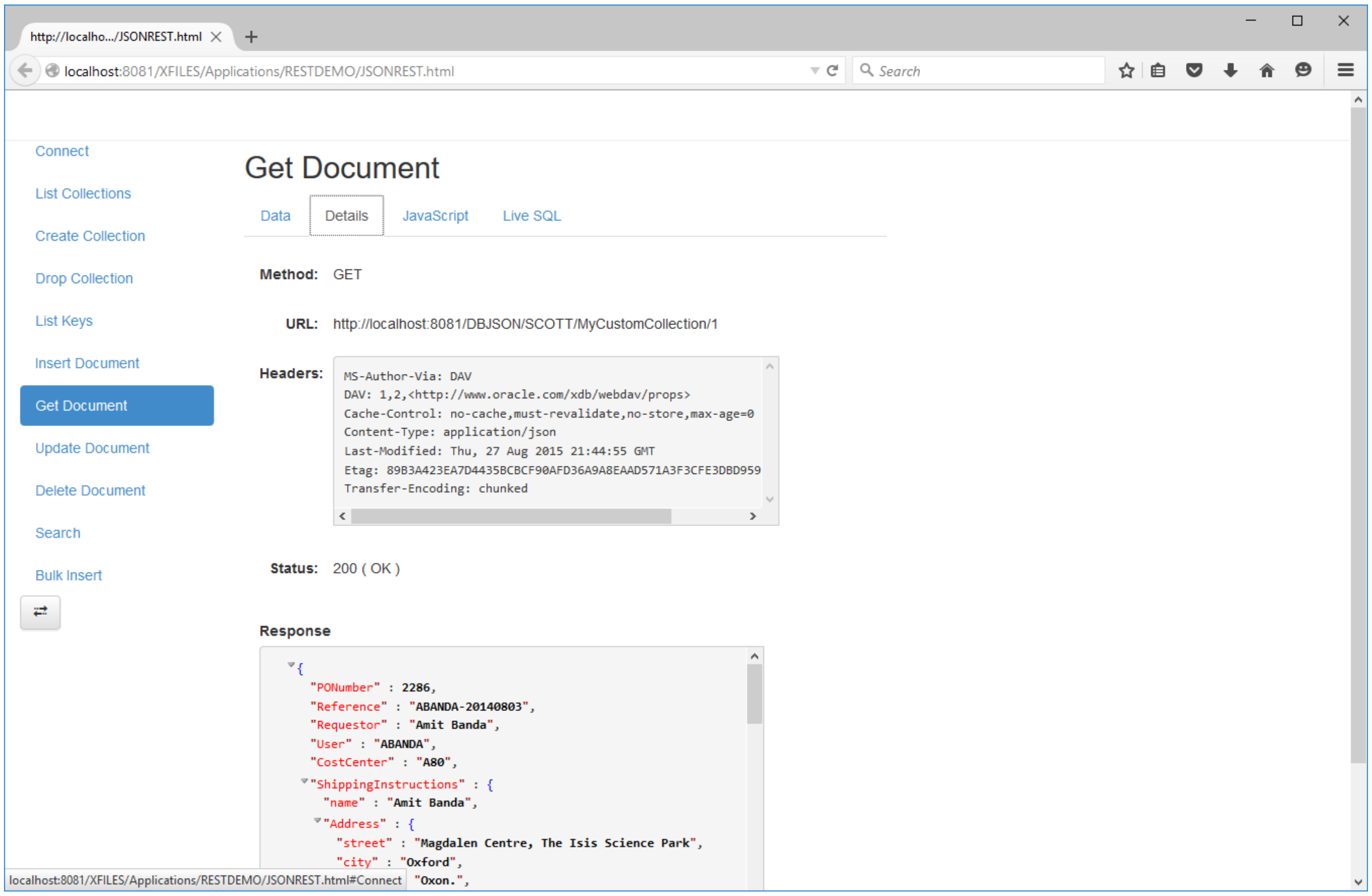
Content

```
{
  "PONumber" : 2286,
  "Reference" : "ABANDA-20140803",
  "Requestor" : "Amit Banda",
  "User" : "ABANDA",
  "CostCenter" : "A80",
  "ShippingInstructions" : {
    "name" : "Amit Banda",
    "Address" : {
      "street" : "Magdalen Centre, The Isis Science Park",
      "city" : "Oxford",
      "county" : "Oxon.",
      "postcode" : "OX9 9ZB",
      "country" : "United Kingdom"
    },
```

Hardware and Software
Engineered to Work Together

ORACLE

Switch to the Details tab to see information about the HTTP request used to invoke the Get Document service.



Get Document

Method: GET

URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection/1

Headers:

```
MS-Author-Via: DAV
DAV: 1,2,<http://www.oracle.com/xdb/webdav/props>
Cache-Control: no-cache,must-revalidate,no-store,max-age=0
Content-Type: application/json
Last-Modified: Thu, 27 Aug 2015 21:44:55 GMT
Etag: 89B3A423EA7D4435BCBCF90AFD36A9A8EAAD571A3F3CFE3DBD959
Transfer-Encoding: chunked
```

Status: 200 (OK)

Response

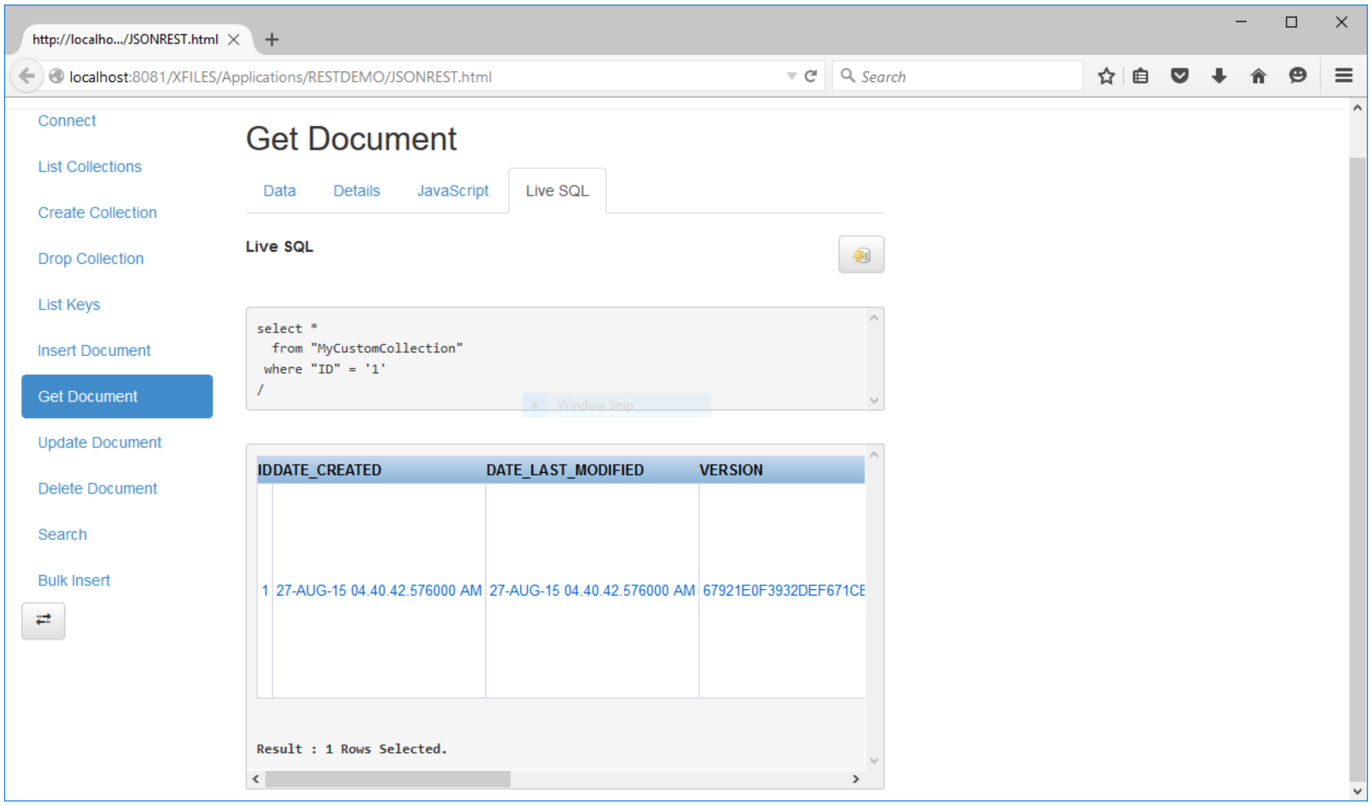
```
{
  "PONumber" : 2286,
  "Reference" : "ABANDA-20140803",
  "Requestor" : "Amit Banda",
  "User" : "ABANDA",
  "CostCenter" : "A80",
  "ShippingInstructions" : {
    "name" : "Amit Banda",
    "Address" : {
      "street" : "Magdalen Centre, The Isis Science Park",
      "city" : "Oxford",
      "Oxon.",
```

The Get Document service was invoked by performing a GET operation on the URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection/1.

The GET operation returned status 200 indicating that the document was found. The document was returned as the body of the GET response.

Switch to the JavaScript tab to see the JavaScript code that was used to execute the HTTP operation.

To see the SQL that is used to retrieve the document from the collection switch to the Live SQL tab and click GO

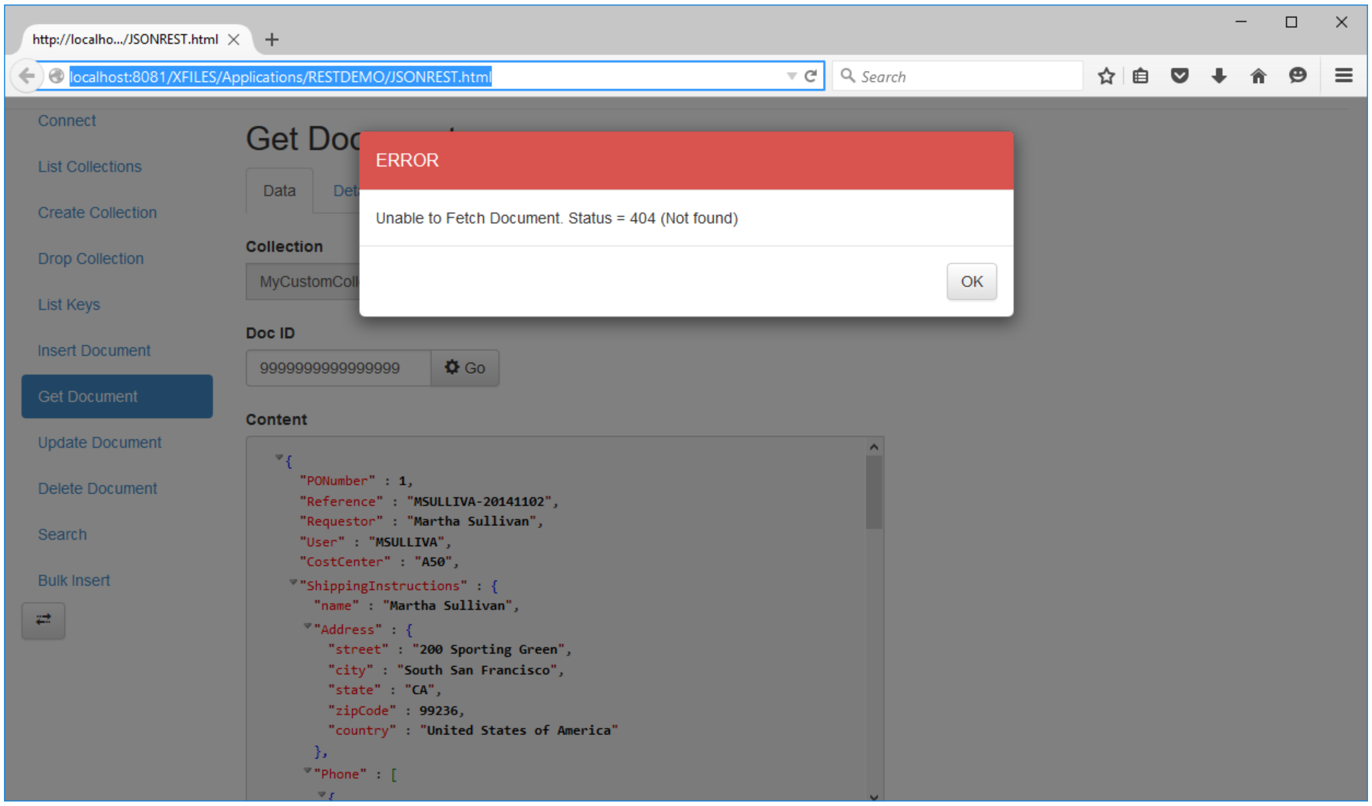


```
select *
  from "MyCustomCollection"
 where "ID" = '1'
/
```

ID	DATE_CREATED	DATE_LAST_MODIFIED	VERSION
1	27-AUG-15 04.40.42.576000 AM	27-AUG-15 04.40.42.576000 AM	67921E0F3932DEF671CE

```
Result : 1 Rows Selected.
```

Click on the **Data** tab; change the value of the Doc Id field that that the key does not match any of the Documents in the collection and click GO.

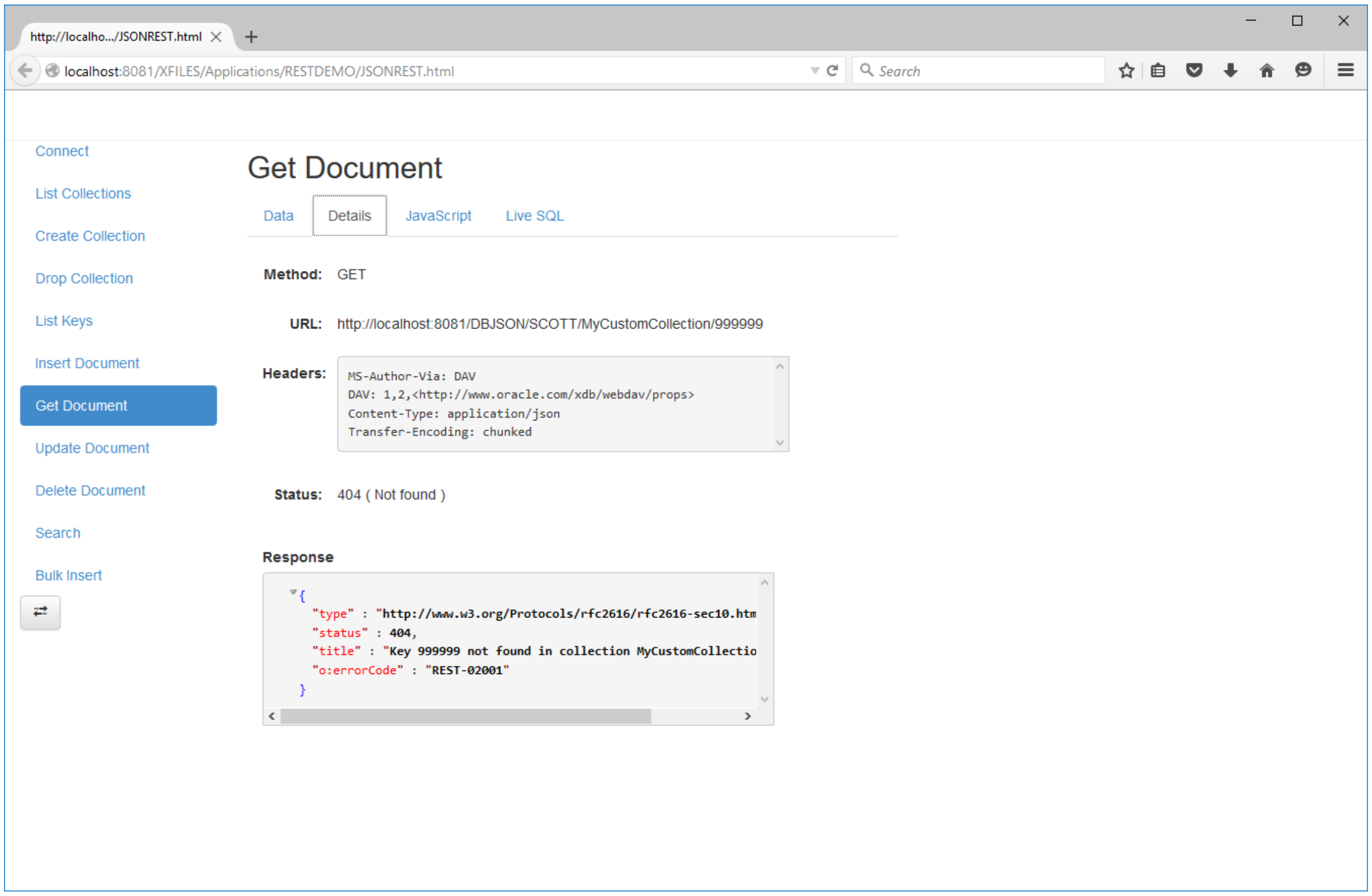


Unable to Fetch Document. Status = 404 (Not found)

```
{
  "PONumber" : 1,
  "Reference" : "MSULLIVA-20141102",
  "Requestor" : "Martha Sullivan",
  "User" : "MSULLIVA",
  "CostCenter" : "A50",
  "ShippingInstructions" : {
    "name" : "Martha Sullivan",
    "Address" : {
      "street" : "200 Sporting Green",
      "city" : "South San Francisco",
      "state" : "CA",
      "zipCode" : 99236,
      "country" : "United States of America"
    },
    "Phone" : [
      {
```

The application will report that no matching document was found.

Switch to the Details tab to see information about the HTTP request used to invoke the Get Document service.



Get Document

Method: GET

URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection/999999

Headers:

```
MS-Author-Via: DAV
DAV: 1,2,<http://www.oracle.com/xdb/webdav/props>
Content-Type: application/json
Transfer-Encoding: chunked
```

Status: 404 (Not found)

Response

```
{
  "type" : "http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.htm
  "status" : 404,
  "title" : "Key 999999 not found in collection MyCustomCollectio
  "o:errorCode" : "REST-02001"
}
```

The GET operation returned status 404 indicating that the required document was not found. The response consists of a JSON document that provides further details about the error.

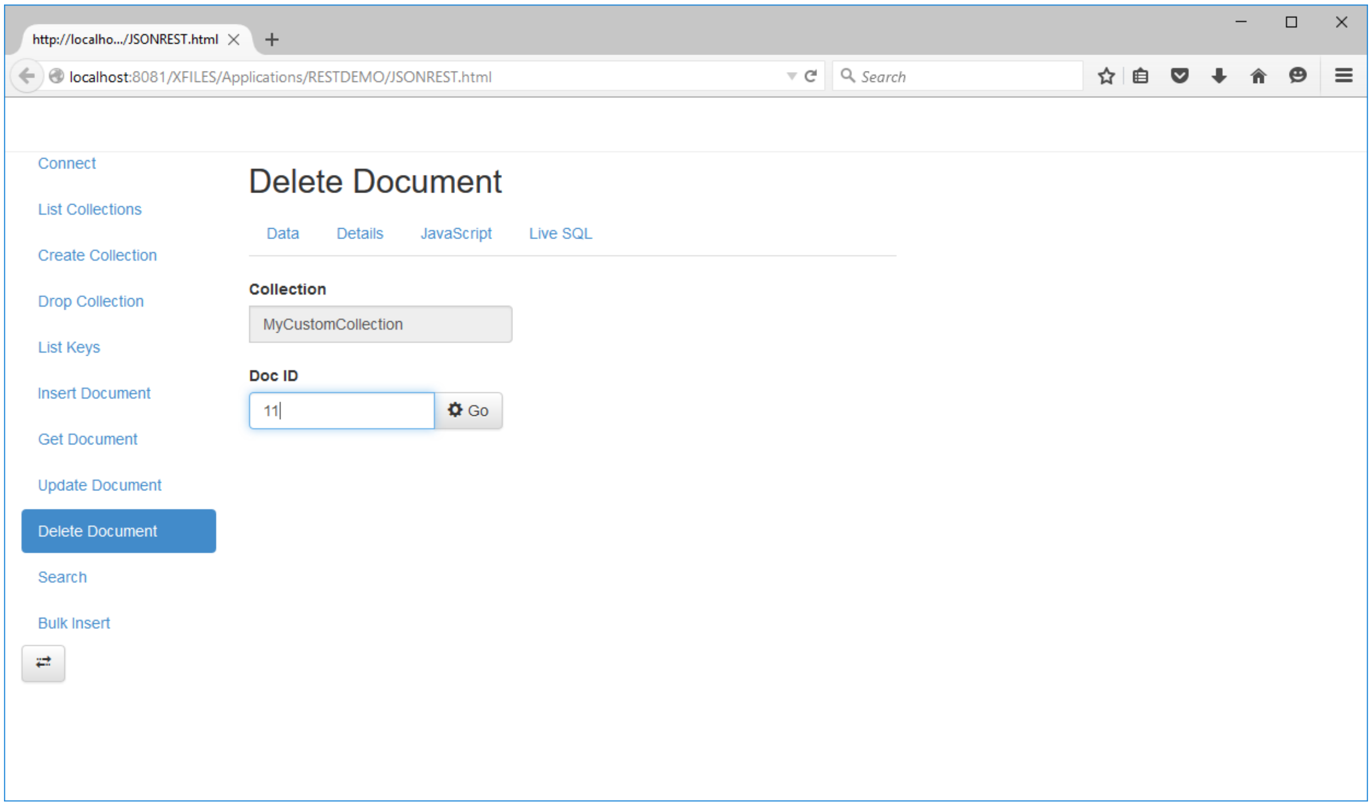
Step 8: Deleting a Document

Click the Delete Document option to switch to the Delete Document form. This form demonstrates the use of the SODA for REST Delete Document service. This service allows a document to be deleted using its ID.

The form contains two fields:

> Collection: This field is a read-only field containing the name of the collection that will be the target of the Delete operation. This field is automatically pre-populated with the name of the collection selected on the List Collections form.
>
> Document Id: This field is used to specify the unique ID of the document be deleted.

Use the List Collections form to select the collection "MyCustomCollection". Switch back to the Delete Document form and enter a known document id or paste the id copied from the response document returned by the Bulk Insert operation into the Document ID field.

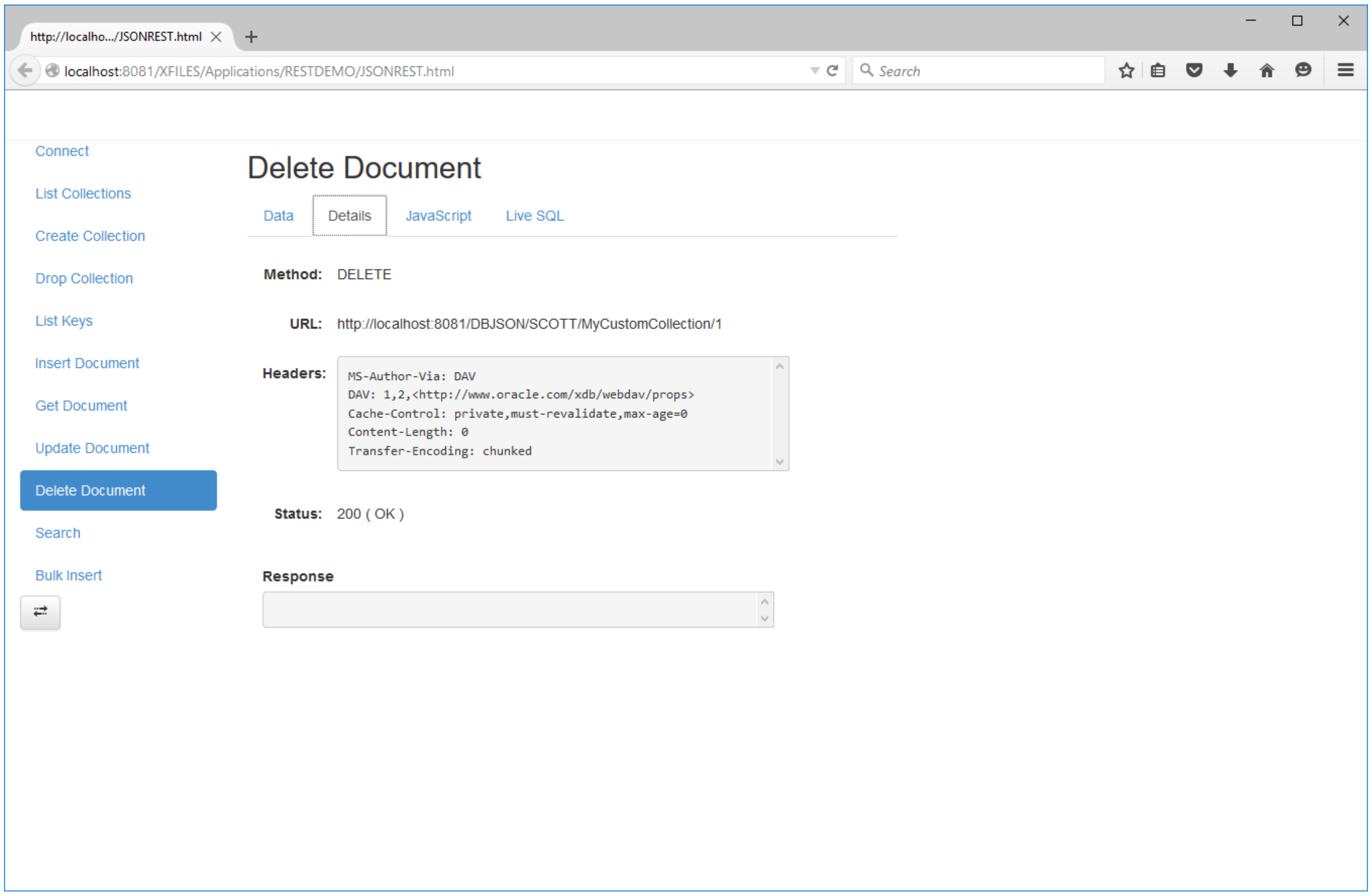


Collection: MyCustomCollection

Doc ID: 11

Click GO to delete the document.

Switch to the Details tab to see information about the HTTP request used to invoke the Delete Document service.



Delete Document

Method: DELETE

URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection/1

Headers:

```
MS-Author-Via: DAV
DAV: 1,2,<http://www.oracle.com/xdb/webdav/props>
Cache-Control: private,must-revalidate,max-age=0
Content-Length: 0
Transfer-Encoding: chunked
```

Status: 200 (OK)

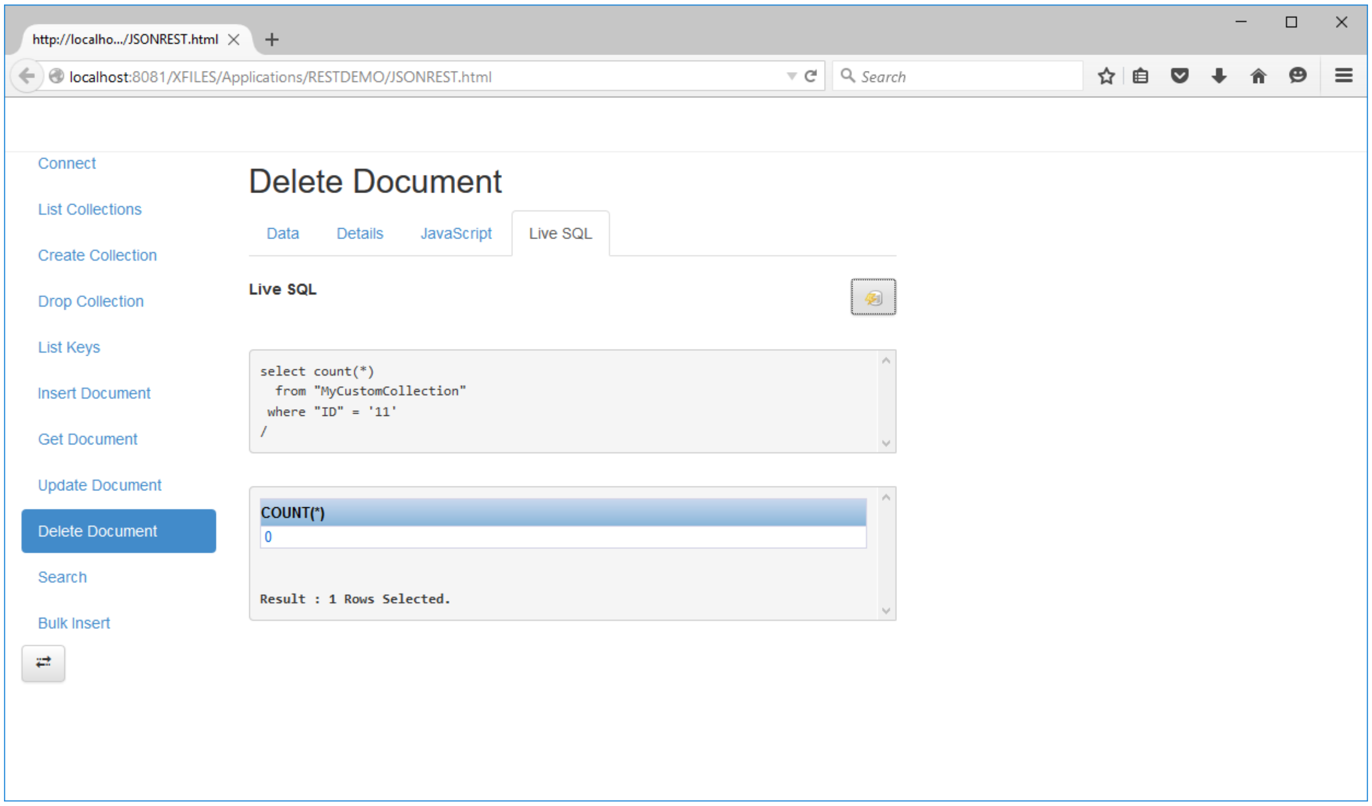
Response

The Delete Document service was invoked by performing a DELETE operation on the URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection/11.

The DELETE operation returned status 200 indicating that the document was deleted. There is no response for a successful Delete document operation. If there is no matching document then the operation will return status 404. The response consists of a JSON document that provides further details about the error.

Switch to the JavaScript tab to see the JavaScript code that was used to execute the HTTP operation.

To use SQL to verify that the document has been deleted from the collection switch to the Live SQL tab and click GO



Delete Document

Live SQL

```
select count(*)
  from "MyCustomCollection"
 where "ID" = '11'
/
```

COUNT(*)
0

```
Result : 1 Rows Selected.
```

Step 9: Updating a Document

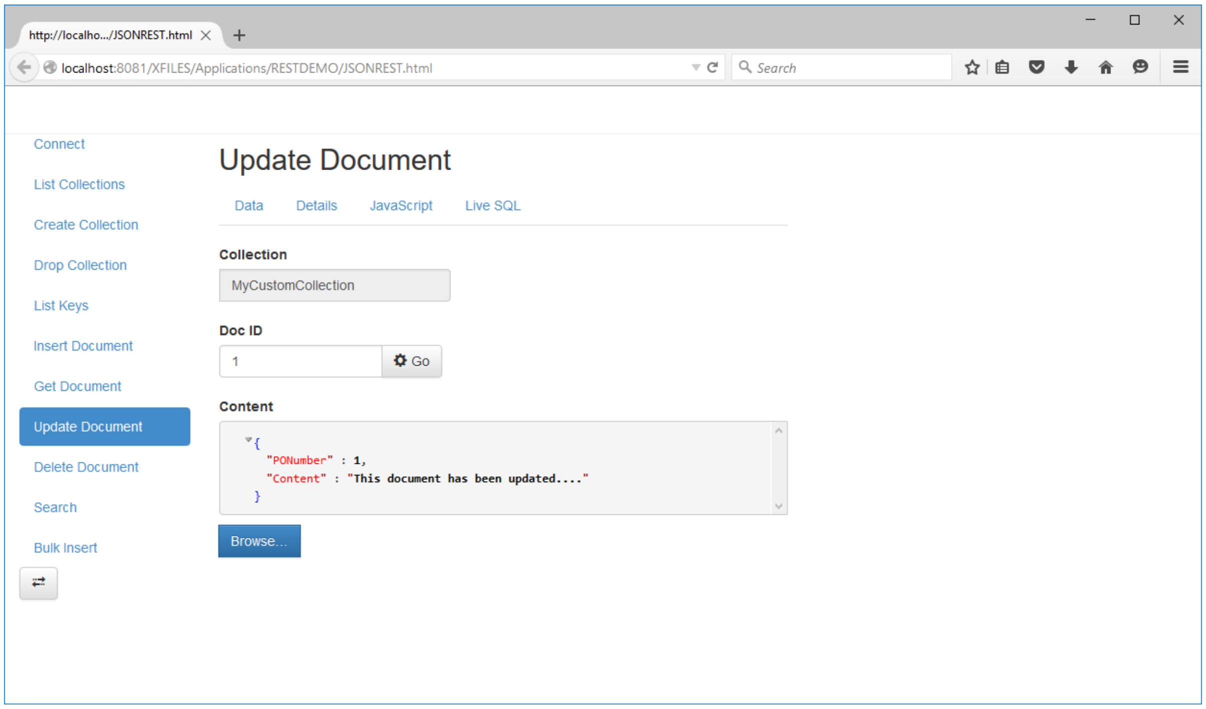
Click the Update Document option to switch to the Update Document form. This form demonstrates the use of the SODA for REST Update Document service. This service allows the content of a document to be updated using its ID.

The form contains three fields:

> Collection: This field is a read-only field containing the name of the collection that will be the target of the Update operation. This field is automatically pre-populated with the name of the collection selected on the List Collections form.
>
> Document Id: This field is used to specify the unique ID of the document be updated.
>
> Content: The field is a read-only field containing the new JSON content. This field is populated by uploading a document from the local file system.

Use the List Collections form to select the collection “MyCustomCollection”. Then use the Get Document form to select and view the current content of the document to be updated.

Switch back to Update Document form and enter the id specified on the Get Document form. Use the Browse button to upload the file poUpdated.json which can be found in the SampleDocuments folder, located under /home/oracle/Desktop/Database_TrackJSON.



Collection: MyCustomCollection

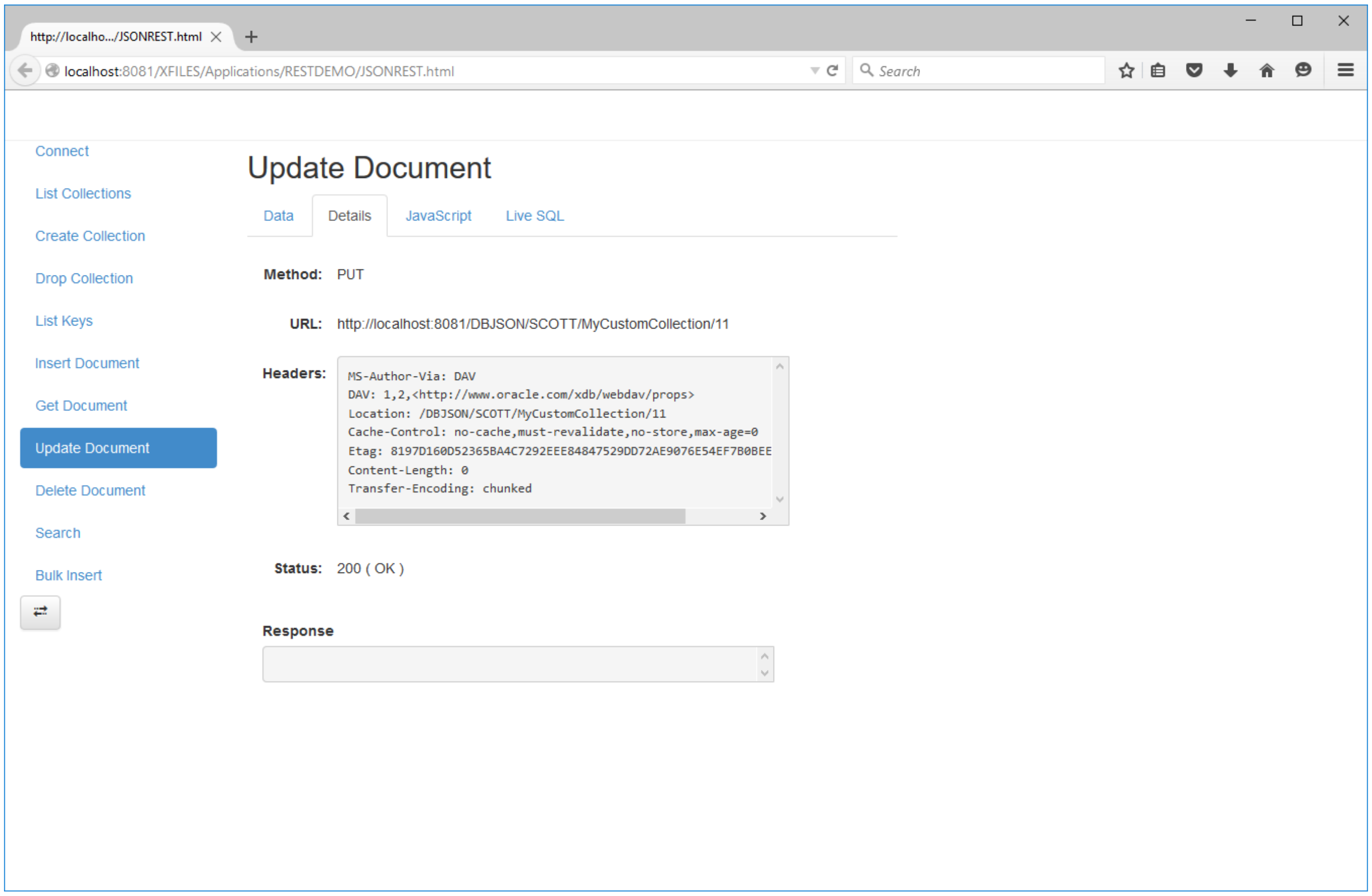
Doc ID: 1

Content:

```
{
  "PONumber" : 1,
  "Content" : "This document has been updated...."
}
```

Click GO to update the document.

Switch to the Details tab to see information about the HTTP request used to invoke the Update Document service.



Update Document

Method: PUT

URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection/11

Headers:

```
MS-Author-Via: DAV
DAV: 1,2,<http://www.oracle.com/xdb/webdav/props>
Location: /DBJSON/SCOTT/MyCustomCollection/11
Cache-Control: no-cache,must-revalidate,no-store,max-age=0
Etag: 8197D160D52365BA4C7292EEE84847529DD72AE9076E54EF7B0BEE
Content-Length: 0
Transfer-Encoding: chunked
```

Status: 200 (OK)

Response

The Update Document service was invoked by performing a PUT operation on the URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection/1.

The PUT operation returned status 200 indicating that the document was updated. There is no response for a successful Update document operation. The header Location contains a URL that can be used to access the document. The header Etag contains an updated ETAG value for the document.

If the collection does not contain a document that matches the supplied id the behavior of the Update Service depends on the key generation method that was specified when the collection was created.

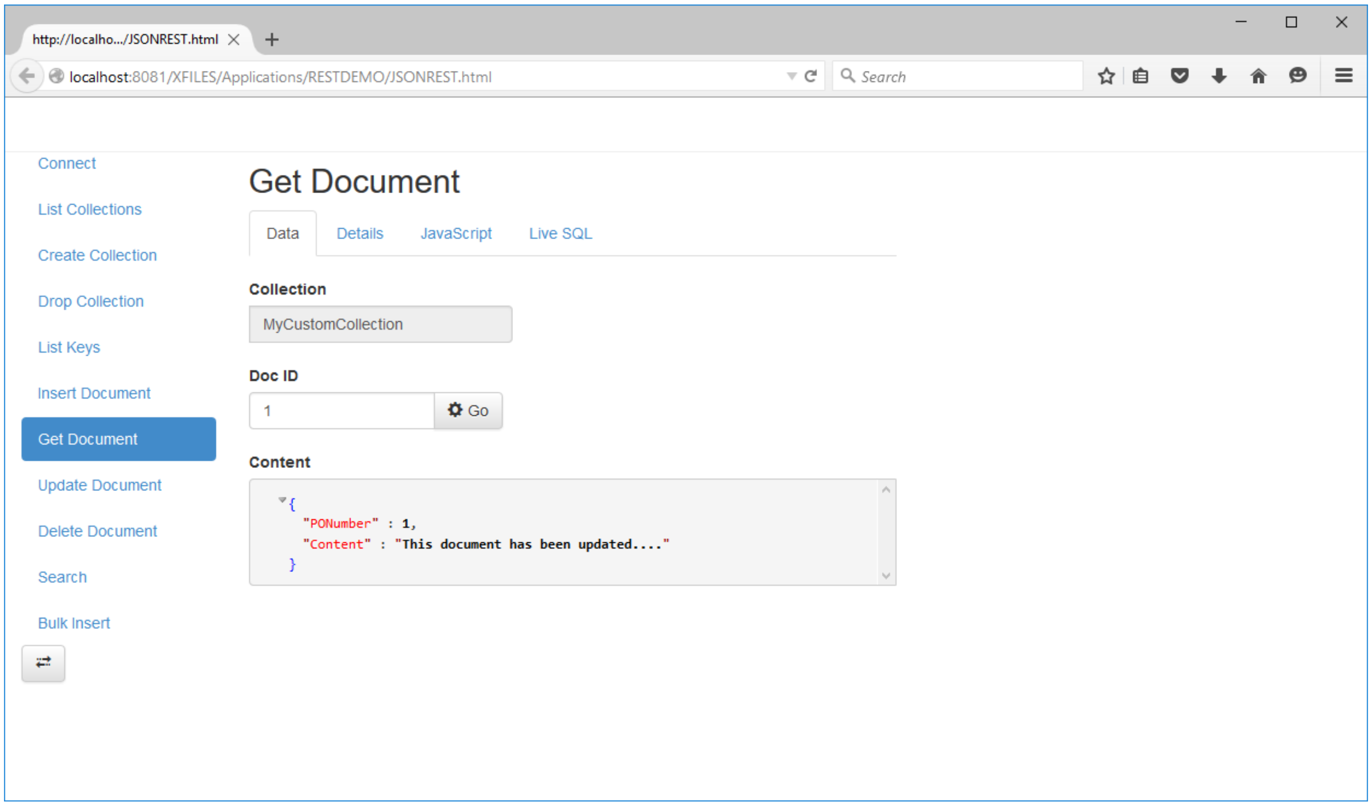
For a collection that uses Server Assigned ids the operation will return status 404. The response will consist of a JSON document containing additional information about the error.

For a collection that uses Client Supplied ids a new document will be crated and the operation will return status 201. The response will consist of a JSON document containing the metadata about the new document.

The header Location will contain a URL that can be used to access the document. The header Etag will contain the updated ETAG value for the document.

Switch to the JavaScript tab to see the JavaScript code that was used to execute the HTTP operation.

Click the Get Document option to switch to the Get Document form. Click GO to reload the document. Confirm that the content of the document has been replaced with the document that was supplied to the Update Service.



Get Document

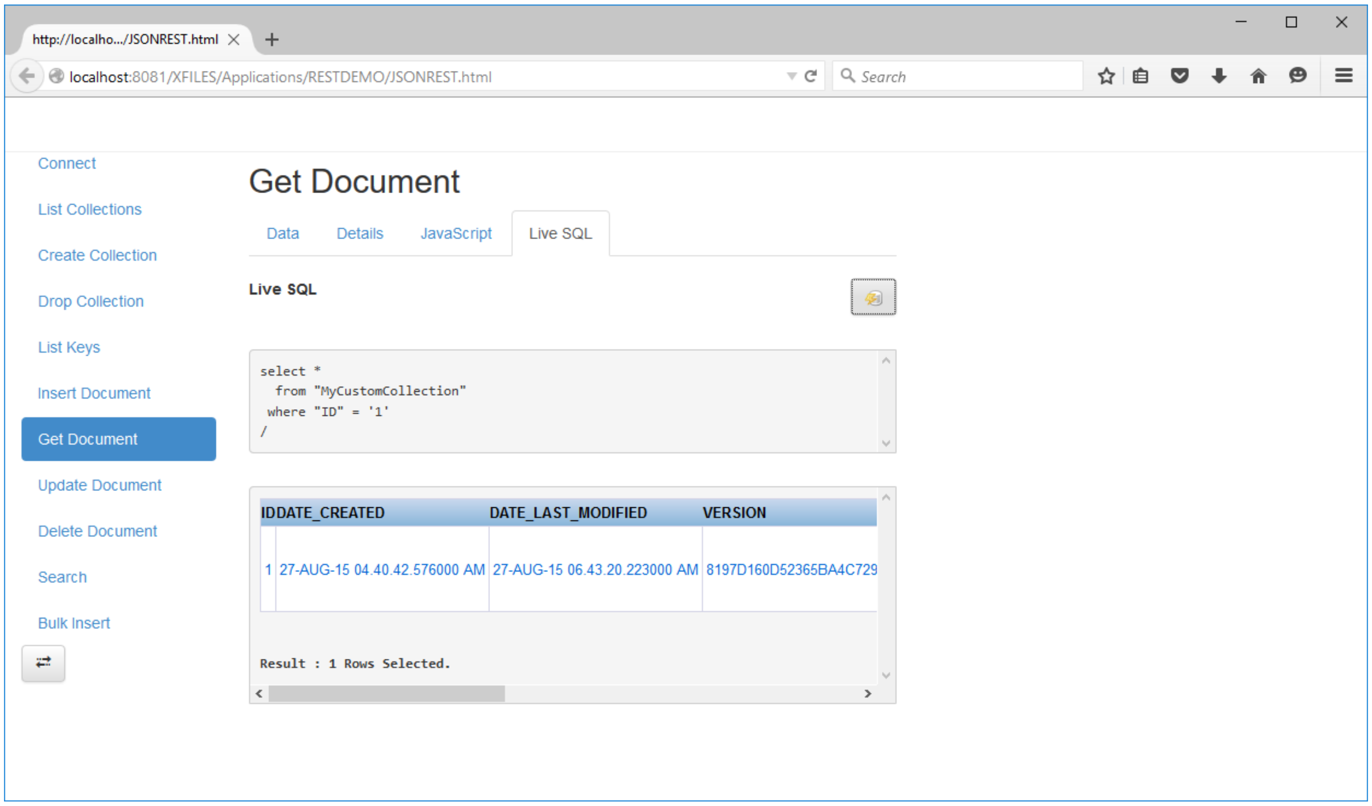
Collection: MyCustomCollection

Doc ID: 1

Content

```
{
  "PONumber" : 1,
  "Content" : "This document has been updated...."
}
```

To verify that the Update Document service successfully updated the document switch to the Live SQL tab and click GO



```
select *
  from "MyCustomCollection"
 where "ID" = '1'
/
```

ID	DATE_CREATED	DATE_LAST_MODIFIED	VERSION
1	27-AUG-15 04.40.42.576000 AM	27-AUG-15 06.43.20.223000 AM	8197D160D52365BA4C729

```
Result : 1 Rows Selected.
```

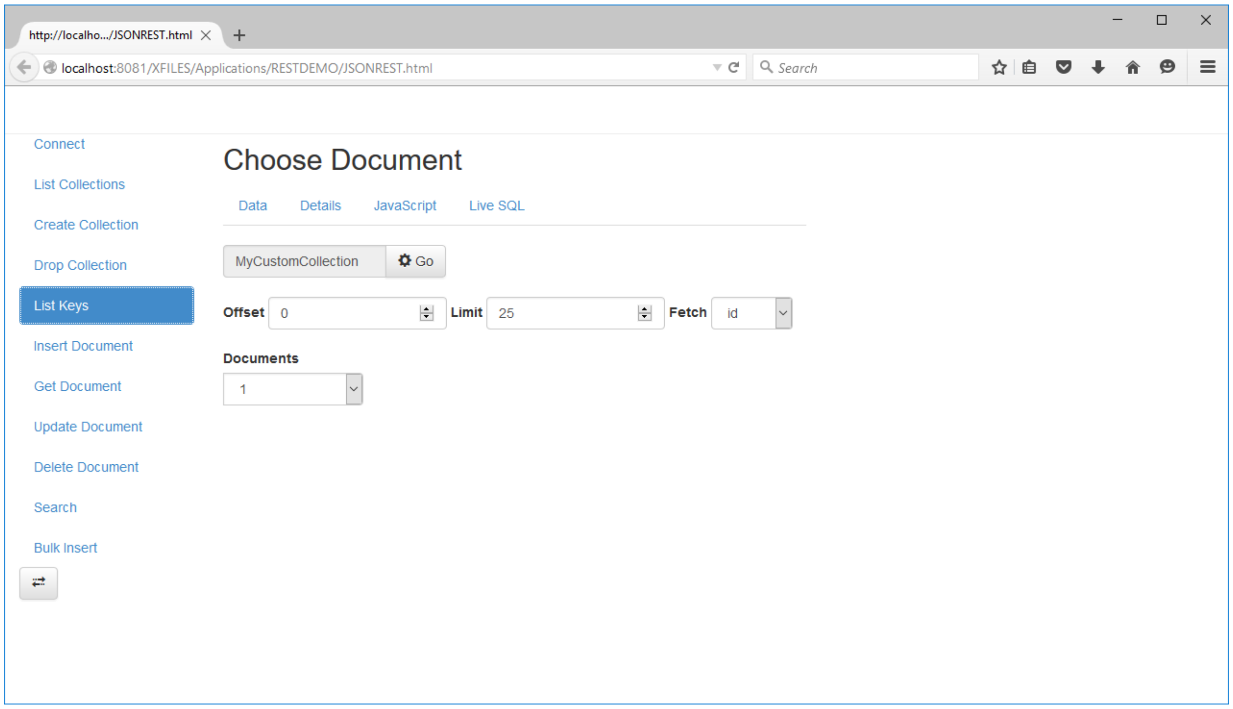
Verify that the modification timestamp has been updated and the content of the document has changed.

Step 10: Listing the Documents in a Collection

Click the List Keys option to switch to the List Collection form. This form demonstrates the use of the SODA for REST List Collection service. This service returns a list of all the documents in the specified collection. The service supports pagination of the results using offset and limit logic. The service also allows you choose between returning metadata, content or both. Note that searching for documents based on content is not a function of the List Collection operation; it is a function of the Query by Example capability which is covered in final step of the Hands-on-Lab.

The form contains four fields:

> Collection: This field is a read-only field containing the name of the collection that will be the target of the Update operation. This field is automatically pre-populated with the name of the collection selected on the List Collections form.
>
> Offset: Used to specify on offset into the result set. The result set will consist of records beyond the offset. Used when implementing pagination of the results.
>
> Limit: Used to specify the number of documents to be returned. Used when implementing pagination of the results.
>
> Documents: This is a read-only field containing a list of the documents returned by the List Collection service.



Choose Document

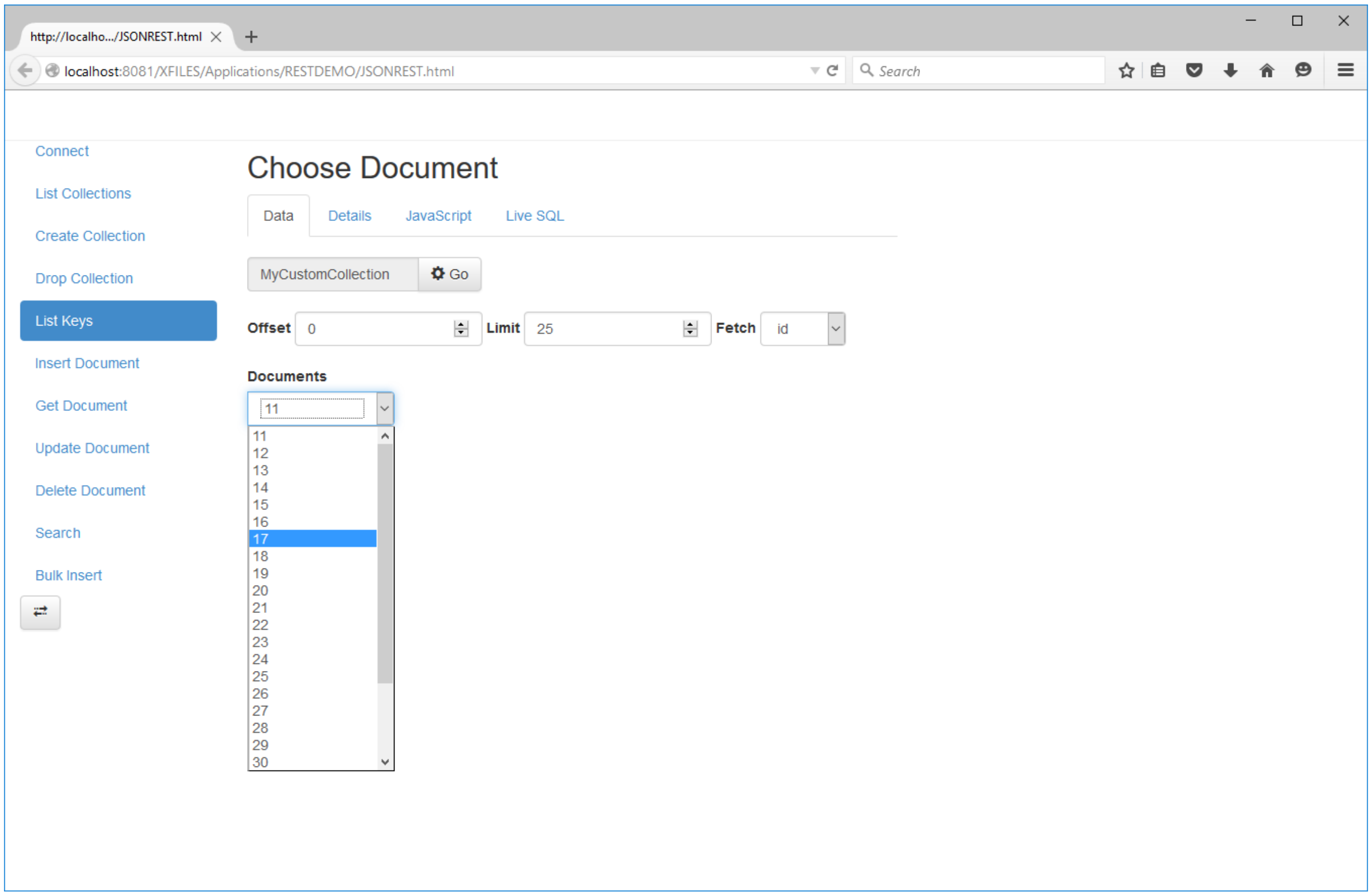
MyCustomCollection

Offset 0 Limit 25 Fetch id

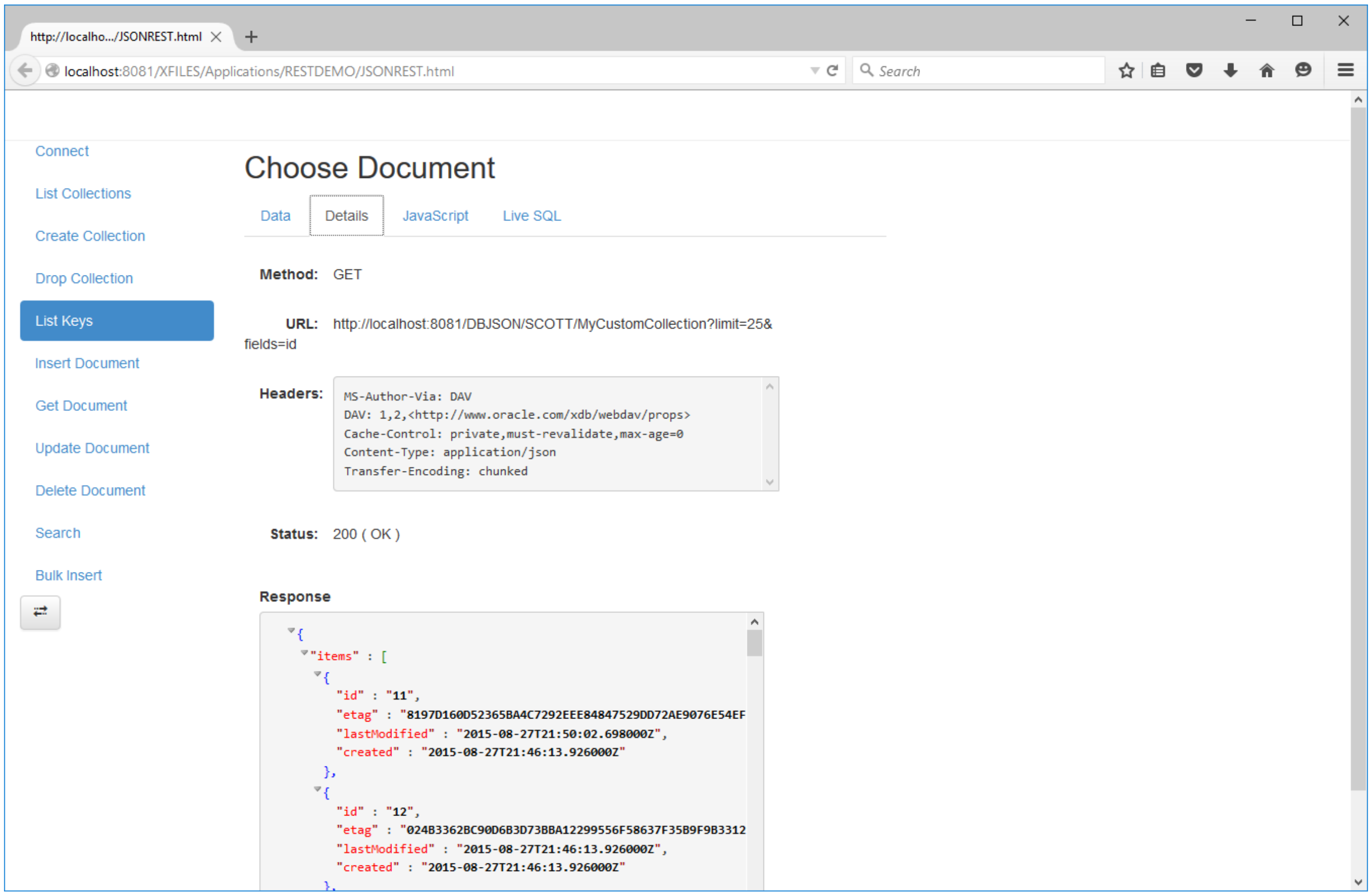
Documents
1

Click GO to fetch the list of documents.

The application will report that 25 documents were retrieved and that more documents are available. The Documents list box will be populated using the ids of the documents that were returned by the List Collection service.



Switch to the Details tab to see information about the HTTP request used to invoke the List Collection service.



Choose Document

Method: GET

URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection?limit=25&fields=id

Headers:

```
MS-Author-Via: DAV
DAV: 1,2,<http://www.oracle.com/xdb/webdav/props>
Cache-Control: private,must-revalidate,max-age=0
Content-Type: application/json
Transfer-Encoding: chunked
```

Status: 200 (OK)

Response

```
{
  "items" : [
    {
      "id" : "11",
      "etag" : "8197D160D52365BA4C7292EEE84847529DD72AE9076E54EF
      "lastModified" : "2015-08-27T21:50:02.698000Z",
      "created" : "2015-08-27T21:46:13.926000Z"
    },
    {
      "id" : "12",
      "etag" : "024B3362BC90D6B3D73BBA12299556F58637F35B9F9B3312
      "lastModified" : "2015-08-27T21:46:13.926000Z",
      "created" : "2015-08-27T21:46:13.926000Z"
    },
```

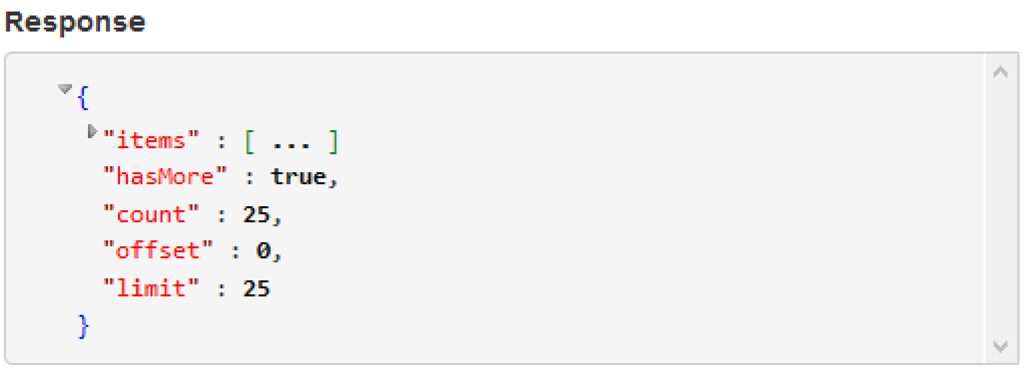
The List Collection service was invoked by performing a GET operation on the URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection?limit=25&fields=id

The GET operation returned status 200 indicating that the list collection operation succeeded. The response document consists of a JSON document that contains the results of the list operation. The results are contained as a set of objects in an array called “items”. There will be one member of the array for each matching document. If the Collection contains no documents the response document will consist of an empty items array.

The condition “fields=id” in the search string means that only the metadata for the matching documents is included in the response. If “fields” is omitted then the default is to return metadata and content. The content of the document can be found under the key “value”

The condition “limit=25” means that only information for the first 25 documents is returned. If “limit” is omitted then the information for all documents in the collection will be returned.

Click on the array next to the “items” key to collapse the items array and see the additional metadata that is returned as part of the response

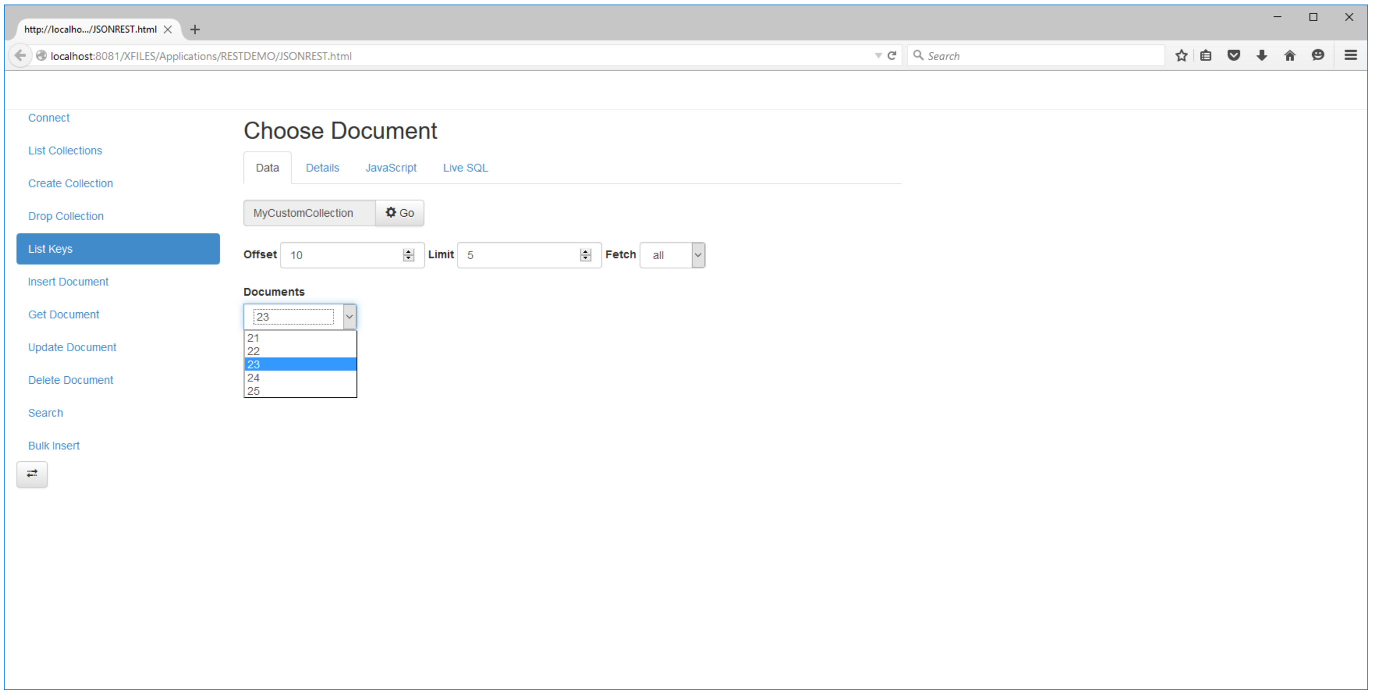


Response

```
{
  "items" : [ ... ]
  "hasMore" : true,
  "count" : 25,
  "offset" : 0,
  "limit" : 25
}
```

As can be seen 25 documents were returned, and there are more documents available that were not included in the List Document response.

Switch to the Data Tab and enter the values 10 for “Offset” and 5 for “Limit”. Set the “Fields” drop down to “all” and click GO.

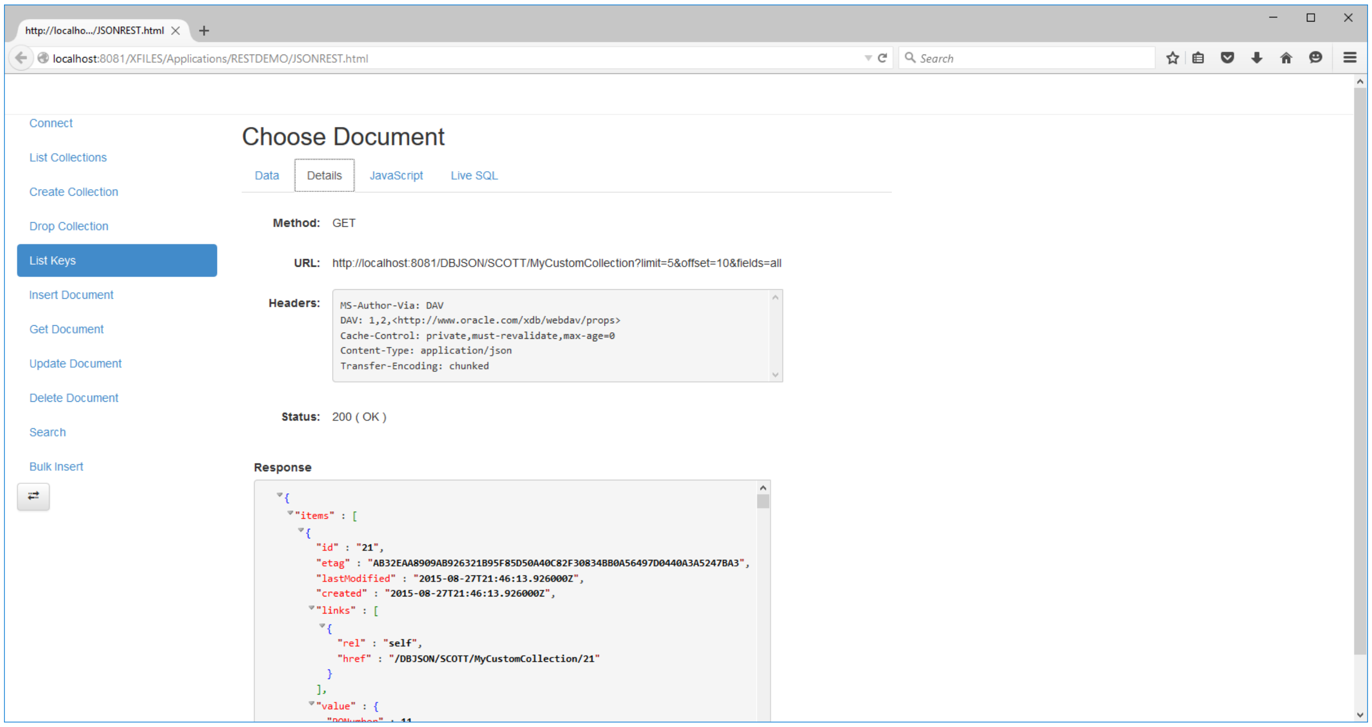


Offset: 10 Limit: 5 Fetch: all

Documents: 23

Open the “Documents” List Box and confirm that it contains 5 values.

Switch to the Details tab to see information about the HTTP request used to invoke the List Collection service.



Method: GET

URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection?limit=5&offset=10&fields=all

Headers:

```
MS-Author-Via: DAV
DAV: 1,2,<http://www.oracle.com/xdb/webdav/props>
Cache-Control: private,must-revalidate,max-age=0
Content-Type: application/json
Transfer-Encoding: chunked
```

Status: 200 (OK)

Response

```
{
  "items" : [
    {
      "id" : "21",
      "etag" : "AB32EAA8909AB926321B95F85D50A40C82F30834BB0A56497D0440A3A5247BA3",
      "lastModified" : "2015-08-27T21:46:13.926000Z",
      "created" : "2015-08-27T21:46:13.926000Z",
      "links" : [
        {
          "rel" : "self",
          "href" : "/DBJSON/SCOTT/MyCustomCollection/21"
        }
      ],
      "value" : {
```

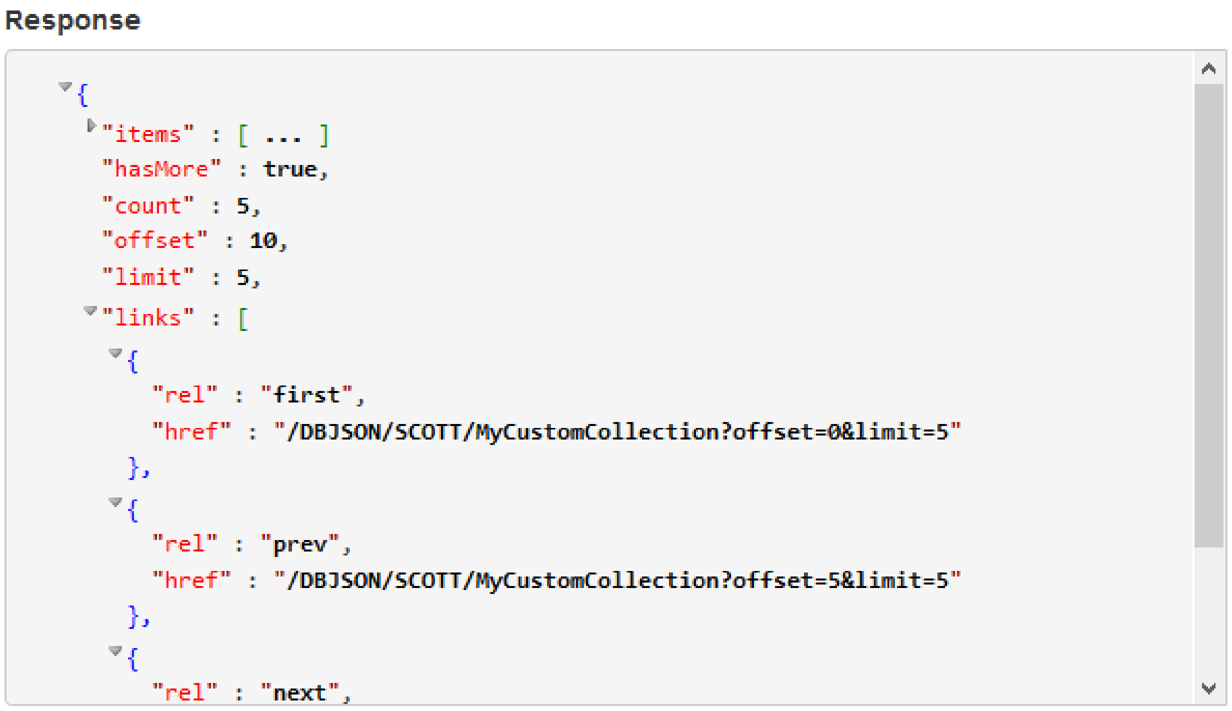
This time the List Collection service was invoked by performing a GET operation on the URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection?limit=5&offset=10&fields=all

The GET operation returned status 200 indicating that the list collection operation succeeded.

Specifying the condition “fields=all” in the search string causes two additional keys to be added to each member of the “items” array. The first key, “links”, contains a key “href” that provides a URL that can be used to access the content of the document via the Get Document service. The second key, “value”, contains the content of the document.

This option should only be used when returning a small number of documents. The use of “fields=all” makes the response document much larger but saves subsequent round trips to the server if the content of the all documents selected needs to be consumed by the application.

Collapse the items array to see the additional metadata that is returned as part of the response.

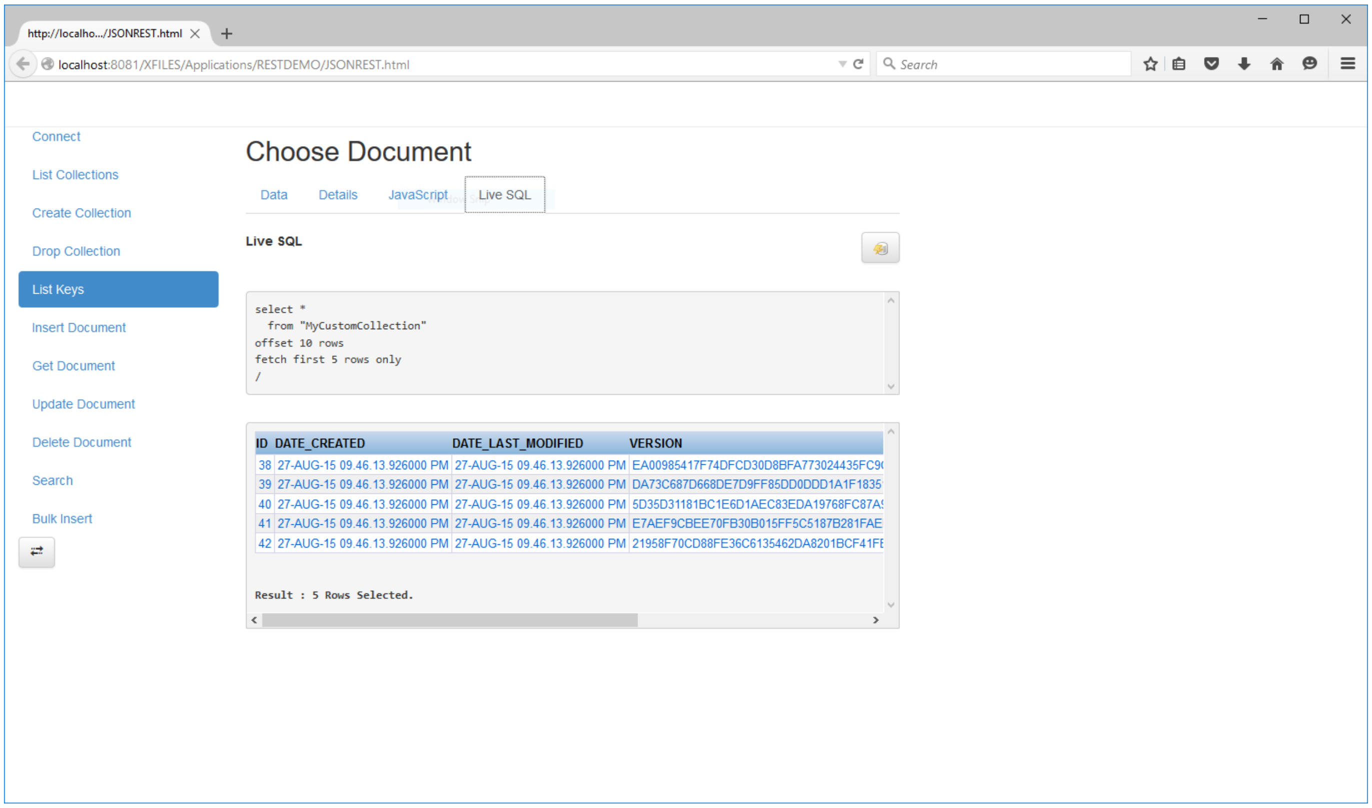


Response

```
{
  "items" : [ ... ]
  "hasMore" : true,
  "count" : 5,
  "offset" : 10,
  "limit" : 5,
  "links" : [
    {
      "rel" : "first",
      "href" : "/DBJSON/SCOTT/MyCustomCollection?offset=0&limit=5"
    },
    {
      "rel" : "prev",
      "href" : "/DBJSON/SCOTT/MyCustomCollection?offset=5&limit=5"
    },
    {
      "rel" : "next",
```

This time the metadata shows that 5 documents were returned, at an offset of 10 and that more documents are available. The metadata also contains a key called “links”, which helps with pagination of the result set by providing URLs that can be used to directly access the first, previous and next set of documents in the collection using the List Collection service.

To see the SQL used to generate the document list switch to the Live SQL tab and click GO.



Choose Document

Live SQL

```
select *
  from "MyCustomCollection"
offset 10 rows
fetch first 5 rows only
/
```

ID	DATE_CREATED	DATE_LAST_MODIFIED	VERSION
38	27-AUG-15 09.46.13.926000 PM	27-AUG-15 09.46.13.926000 PM	EA00985417F74DFCD30D8BFA773024435FC9
39	27-AUG-15 09.46.13.926000 PM	27-AUG-15 09.46.13.926000 PM	DA73C687D668DE7D9FF85DD0DDD1A1F1835
40	27-AUG-15 09.46.13.926000 PM	27-AUG-15 09.46.13.926000 PM	5D35D31181BC1E6D1AEC83EDA19768FC87A
41	27-AUG-15 09.46.13.926000 PM	27-AUG-15 09.46.13.926000 PM	E7AEF9CBEE70FB30B015FF5C5187B281FAE
42	27-AUG-15 09.46.13.926000 PM	27-AUG-15 09.46.13.926000 PM	21958F70CD88FE36C6135462DA8201BCF41F

```
Result : 5 Rows Selected.
```

Step 11: Searching Collections

Click the Search option to switch to the Search Documents form. This form demonstrates the use of the SODA for REST Query-By-Example service. This service returns a list of all the documents in the specified collection that match the provided search criteria.

Search criteria are specified using a Query-By-Example (QBE) metaphor .The QBE service accepts a template document containing one or more predicates and returns a list of documents that match the supplied template. The QBE template is a JSON document.

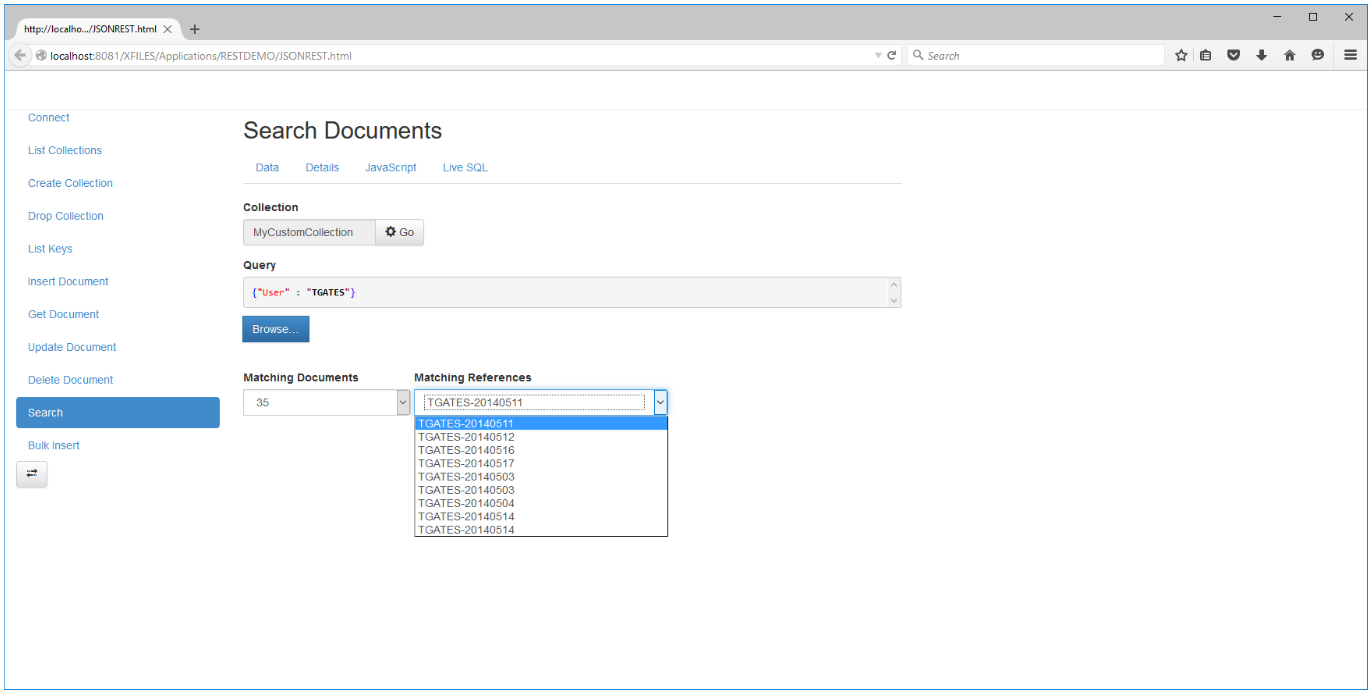
The service supports pagination of the results using offset and limit logic. The service also allows you choose between returning metadata, content or content and metadata.

The form contains four fields:

> Collection: This field is a read-only field containing the name of the collection that will be the target of the Update operation. This field is automatically pre-populated with the name of the collection selected on the List Collections form.
>
> Query: The field is a read-only field containing the QBE template. This field is populated by uploading a document from the local file system.
>
> Documents: This is a List Box containing the ids of the documents returned by QBE service.
>
> References: This is a List Box containing the value of the “Reference” key from each of the matching documents.

Use the Browse button to upload the file QBE.1.json which can be found in the SampleDocuments folder, located under /home/oracle/Desktop/Database_TrackJSON. This document contains a very basic QBE template which will trigger a search for documents that contain a top level key “User” with the value “TBATES”.

Click GO to fetch the list of matching documents.


http://localho.../JSONREST.html

localhost:8081/XFILES/Applications/RESTDEMO/JSONREST.html

Search Documents

Collection: MyCustomCollection

Query

```
{"User" : "TGATES"}
```

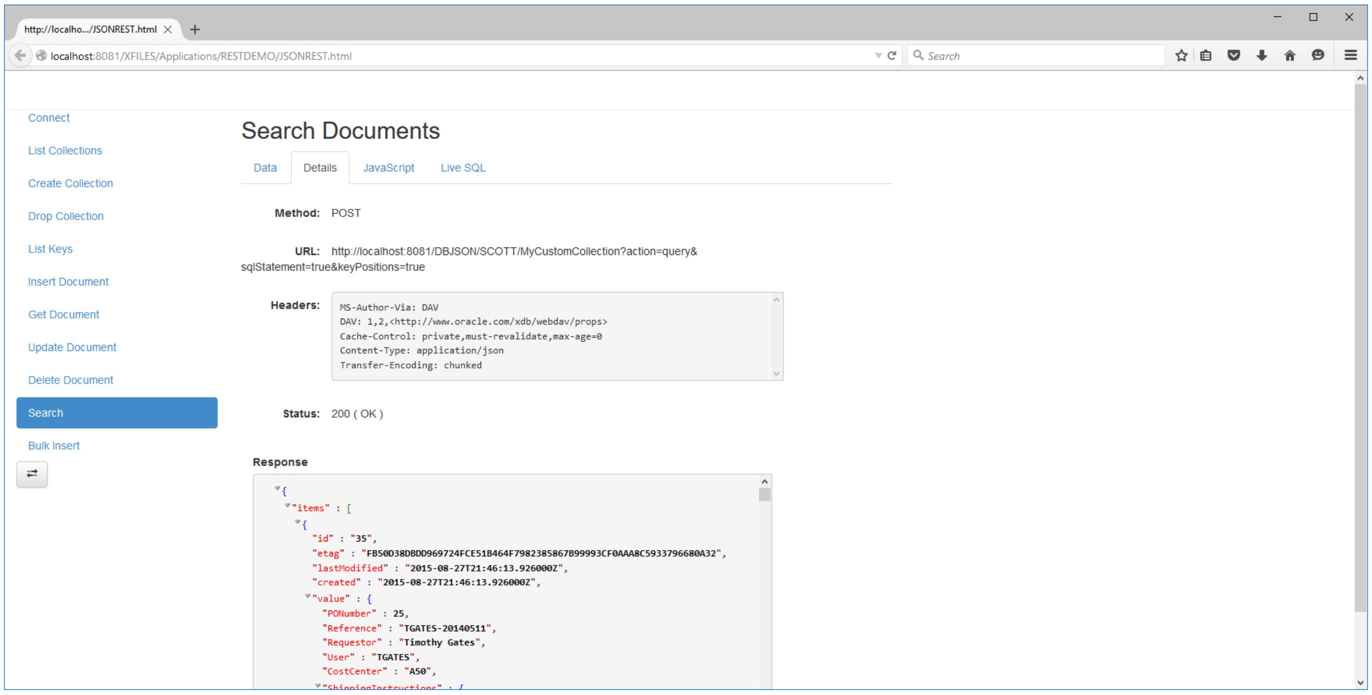
Matching Documents: 35

Matching References: TGATES-20140511

```
TGATES-20140511
TGATES-20140512
TGATES-20140516
TGATES-20140517
TGATES-20140503
TGATES-20140503
TGATES-20140504
TGATES-20140514
TGATES-20140514
```

The application will report that 9 documents matched the predicates specified in the QBE template. Confirm this by opening the “Documents” and “References” List Boxes.

Switch to the Details tab to see information about the HTTP request used to invoke the List Collection service.



Search Documents

Method: POST

URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection?action=query&sqlStatement=true&keyPositions=true

Headers:

```
MS-Author-Via: DAV
DAV: 1,2,<http://www.oracle.com/xdb/webdav/props>
Cache-Control: private,must-revalidate,max-age=0
Content-Type: application/json
Transfer-Encoding: chunked
```

Status: 200 (OK)

Response

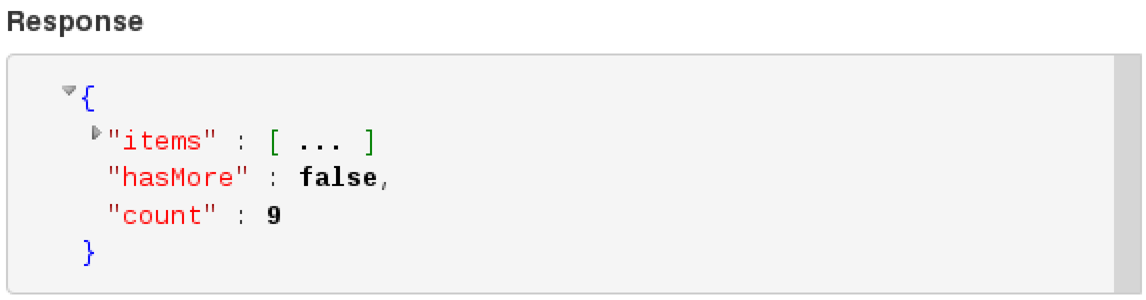
```
{
  "items" : [
    {
      "id" : "35",
      "etag" : "FB50D38DBDD969724FCE51B464F7982385867B99993CF0AAA8C5933796680A32",
      "lastModified" : "2015-08-27T21:46:13.926000Z",
      "created" : "2015-08-27T21:46:13.926000Z",
      "value" : {
        "PONumber" : 25,
        "Reference" : "TGATES-20140511",
        "Requestor" : "Timothy Gates",
        "User" : "TGATES",
        "CostCenter" : "A50",
```

The Query By Example service was invoked by performing a POST operation on the URL: http://localhost:8081/DBJSON/SCOTT/MyCustomCollection?action=query&sqlStatement=true&keyPositions=true. The QBE specification was sent as the body of the request. The POST operation returned status 200 indicating that the list collection operation succeeded.

Specifying “action=query” causes the QBE service to be invoked. Specifying “sqlStatement=true” causes a copy of the SQL Statement generated by the QBE service to included in the respons. Specifying “keyPositions=true” adds an array of key / position mappings to the metadata generated by the service.

Since the “fields” argument was not supplied, the default behavior of “fields=all” is assumed, so each member of the “items” array consists of a JSON object containing the metadata as well as the content of the matching documents.

Collapse the items array to see the additional metadata that is returned as part of the response.



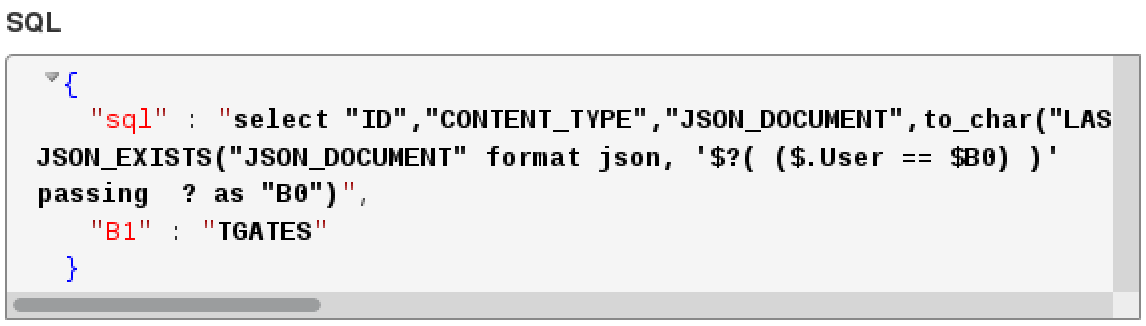
Response

```
{
  "items" : [ ... ]
  "hasMore" : false,
  "count" : 9
}
```

A total of 9 documents were found that match the specified criteria. Since there are no more matching documents the value of the “hasMore” key is false, indicating that there are no further results to send.

Switch to the JavaScript tab to see the JavaScript code that was used to execute the HTTP operation.

To see the SQL that was executed switch to the Live-SQL tab. Note that SQL in this tab is currently not in a format that can be executed using the LiveSQL capability, but is an actual copy of the SQL that was generated by the SODA API.



SQL

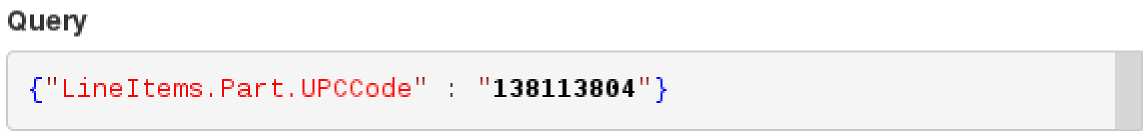
```
{
   "sql" : "select "ID","CONTENT_TYPE","JSON_DOCUMENT",to_char("LAS
JSON_EXISTS("JSON_DOCUMENT" format json, '$?( ($.User == $B0) )'
passing  ? as "B0")",
   "B1" : "TGATES"
}
```

As can be seen the QBE was executed by translating it into a set of JSON Path expressions that can be evaluated using a JSON_EXISTS operator. Values for the predicates are supplied as bind variables.

Switch back to the data tab and use the Browse button to load some other QBE examples. The following screen captures show some simple QBE expressions and the corresponding SQL. These QBE expressions can be loaded from the SampleDocuments folder under /home/oracle/Desktop/Database_TrackJSON.

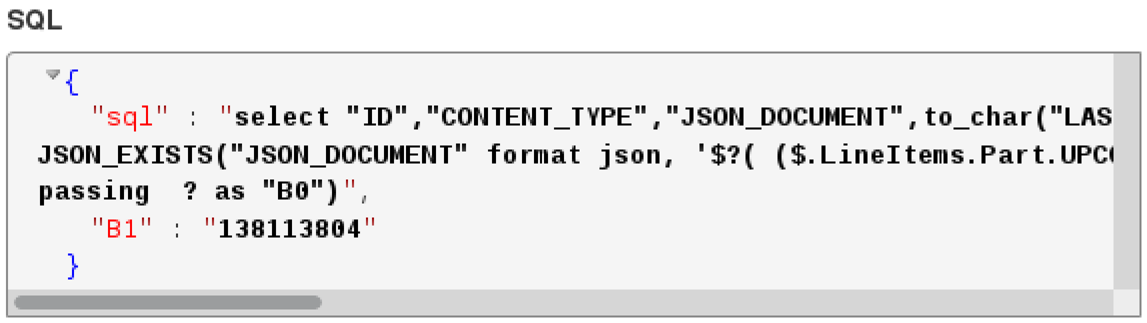
QBE.2.json

This QBE searches for documents where the value of UPCCode key within the Part object of the LineItems object equals “138113804”. Note that since LineItems in an array and no index is specified this will search all members of the array



Query

```
{"LineItems.Part.UPCCode" : "138113804"}
```



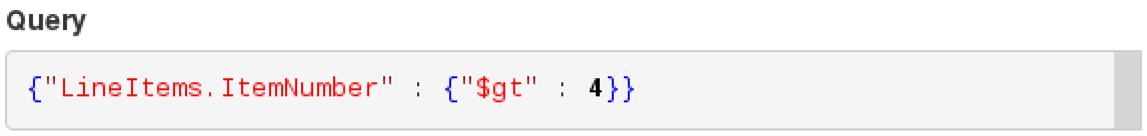
SQL

```
{
  "sql" : "select "ID","CONTENT_TYPE","JSON_DOCUMENT",to_char("LAS
JSON_EXISTS("JSON_DOCUMENT" format json, '$?( ($.LineItems.Part.UPC
passing  ? as "B0")",
  "B1" : "138113804"
}
```

.

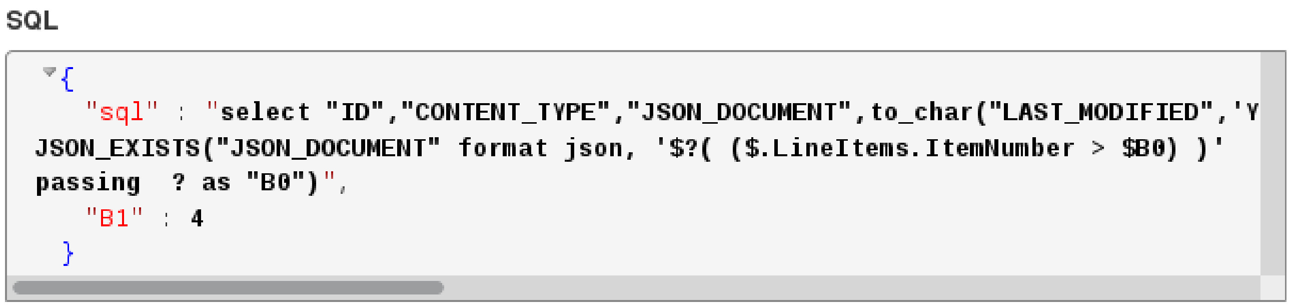
QBE.3.json

This QBE searches for documents where the value of the ItemNumber key of the LineItems object is greater than 4. Note that in this example the predicate is not equality, so a predicate object, which consists of an operator key and a value, is used to provide the search criteria.



Query

```
{"LineItems.ItemNumber" : {"$gt" : 4}}
```

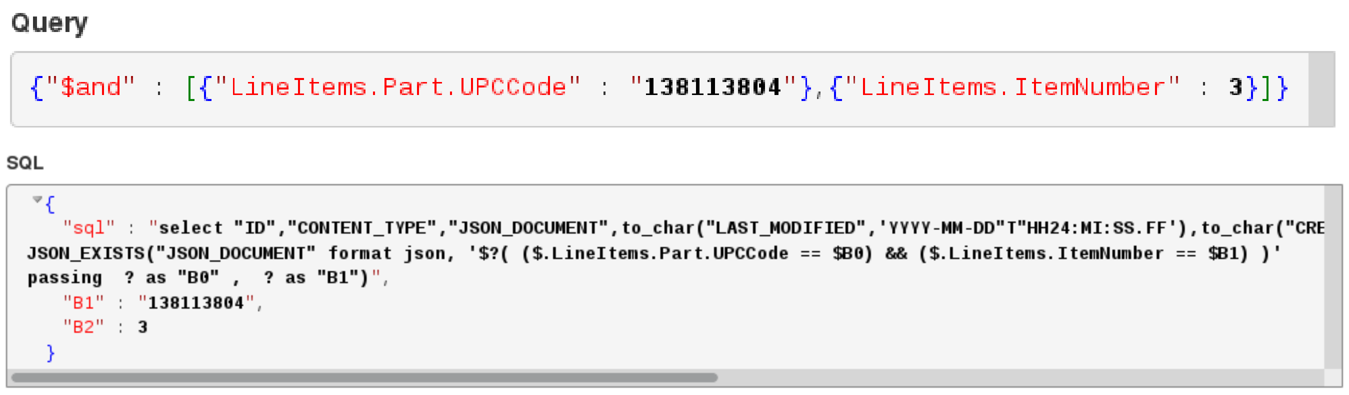


SQL

```
{
  "sql" : "select "ID","CONTENT_TYPE","JSON_DOCUMENT",to_char("LAST_MODIFIED",'Y
JSON_EXISTS("JSON_DOCUMENT" format json, '$?( ($.LineItems.ItemNumber > $B0) )'
passing  ? as "B0")",
  "B1" : 4
}
```

QBE.4.json

This QBE searches for documents where the value of the UPCCode key equals “131138004” and the value of the ItemNumber key = 3. This is done by using an $and object which consists of an array of condition objects.


Query

```
{"$and" : [{"LineItems.Part.UPCCode" : "138113804"},{"LineItems.ItemNumber" : 3}]}
```

SQL

```
{
   "sql" : "select "ID","CONTENT_TYPE","JSON_DOCUMENT",to_char("LAST_MODIFIED",'YYYY-MM-DD"T"HH24:MI:SS.FF'),to_char("CRE
JSON_EXISTS("JSON_DOCUMENT" format json, '$?( ($.LineItems.Part.UPCCode == $B0) && ($.LineItems.ItemNumber == $B1) )'
passing  ? as "B0" ,  ? as "B1")",
   "B1" : "138113804",
   "B2" : 3
}
```

Full details of the QBE language can be found in the documentation.

Conclusion

This Hands-on-Lab has shown how SODA for REST allows developers to create applications that use the Oracle Database as a JSON Document Store. SODA for REST provides full support for using RESTFul techniques to store, query and retrieve JSON documents using an Oracle Database. It allow applications developers to take full advantage of all of the benefits of modern schema-less development paradigms without sacrificing the powerful data management capabilities of the Oracle Database. Using SODA for REST developers can developer and deploy their applications without requiring any support from the Oracle DBA.

The next section of the Hands-on-Lab will show how to Oracle Database 12c allows SQL to be used to perform analytical and reporting operations on JSON content with sacrificing any of the benefits of using JSON as the basis of schema-less development. Instructions for the next section of the Hands-on–Lab can be found here.