# Working with JSON-Relational Duality Views via REST APIs

3/29/23, Version 1.1

Jeff Smith Jeff.D.Smith@oracle.com

## Purpose statement

This document provides an overview of features and enhancements included in release Oracle Database 23c.

*Oracle Database 23c Free – Developer Release is the first release of the next-generation Oracle Database, allowing developers a head-start on building applications with innovative 23c features that simplify development of modern data-driven apps. The entire feature set of Oracle Database 23c is planned to be generally available within the next 12 months.*

## License

## Table of contents

JSON-Relational Duality View (DV) is a revolutionary Oracle Database feature that combines the benefits of relational databases and JSON NoSQL document stores. This feature allows you to store the data in relational tables, in a normalized way, and to expose it to applications as JSON documents. Multiple duality views can be created on the same relational data to address different use-cases. In other words, the same relational data can have different JSON representations. This is a major advantage compared to NoSQL document stores, which force you to pick one specific way to structure the JSON data. This also has major security benefits, as the end user or application can be given access only to those duality view(s) that expose relevant parts of the data. Also, through duality views, write access to specific parts of the data can be controlled (or completely disallowed). At the same time, traditional benefits of data normalization such as data integrity and minimization of redundancy are retained. It is also possible to run SQL on the underlying tables, for analytics or reporting.

For a complete introduction to duality views and their benefits see **Overview of JSON-Relational Duality Views** chapter in JSON-Relational Duality Developer's Guide.

This tutorial walks you through examples of working with JSON-Relational duality views via REST APIs using Formula-1 car-racing season data. A prerequisite for this tutorial is **Introduction to Car-Racing Duality Views Example** chapter, also in **JSON-Relational Duality Developer's Guide.** It describes the use-case and the JSON-Relational duality views you work with in the rest of this document.

To access JSON-Relational duality views through REST APIs, Oracle REST Data Services (ORDS) version 23.1 needs to be installed and configured for your 23c Database.

## Prerequisites

1. Ensure that you have Oracle database 23c, with compatible parameter set to 23.0.0.0

2. Ensure you have cURL installed, or an equivalent interface for calling REST APIs

3. Install Oracle REST Data Services, minimally version of 23.1. Configure it for your 23c instance of Oracle and ensure it's running and accessible. `ORDS Downloads` || `ORDS installation Docs`

   ORDS running standalone on your personal machine will be configured to listen for requests at

   ```
   http://localhost:8080/ords
   ```

4. Create the database schema that will host JSON-Relational duality views and their tables. In this tutorial, schema named *JANUS* is used (replace references to *janus* with whatever schema you use, if it is different). Grant the following to this schema:

   ```
   grant create session to janus;
   
   grant RESOURCE to janus;
   
   grant unlimited tablespace to janus;
   ```

5. Connect to the database as the new user  (JANUS) using SQLcl/SQLDeveloper and execute:

```
exec ords.enable_schema;
commit;
```

6. With the schema 'enabled' for REST APIs, ORDS will serve APIs at the following base URI:

```
http://localhost:8080/ords/janus
```

This path includes the following components (similar paths are used throughout this document):

| Component | Description |
|---|---|
| `localhost` | Network name of the machine running ORDS |
| `8080` | HTTP port. 8080 is the default, adjust accordingly. |
| `Janus` | Example schema containing duality views and their tables, adjust accordingly. |

When executing commands in this tutorial, replace the above URL components with values appropriate for your ORDS instance and database schema.

This tutorial shows curl commands to perform REST API requests. Commands may include query parameters with characters requiring escapes to be specified as part of the curl command. Examples with escapes provided in this document are shown for Windows CMD prompt.

Responses are shown formatted for readability and with headers and bodies, however the cURL command examples as written will generally print the bodies, unformatted. Add the verbose flag (-v) to see everything.

If you are not familiar with using shells and command line interfaces, you can use one of many available REST API graphical user interfaces such as Postman or VS Code's Thunder Client Extension.

Note: many examples in this tutorial use files with JSON content. To run these examples with the provided cURL commands, you must create these files with the file names and content provided in this tutorial. Alternatively, the request bodies can be copied into the REST Client of your choice.

## About REST APIs for Oracle Database

Publishing REST APIs for your database objects can be as easy as simply enabling them for access.

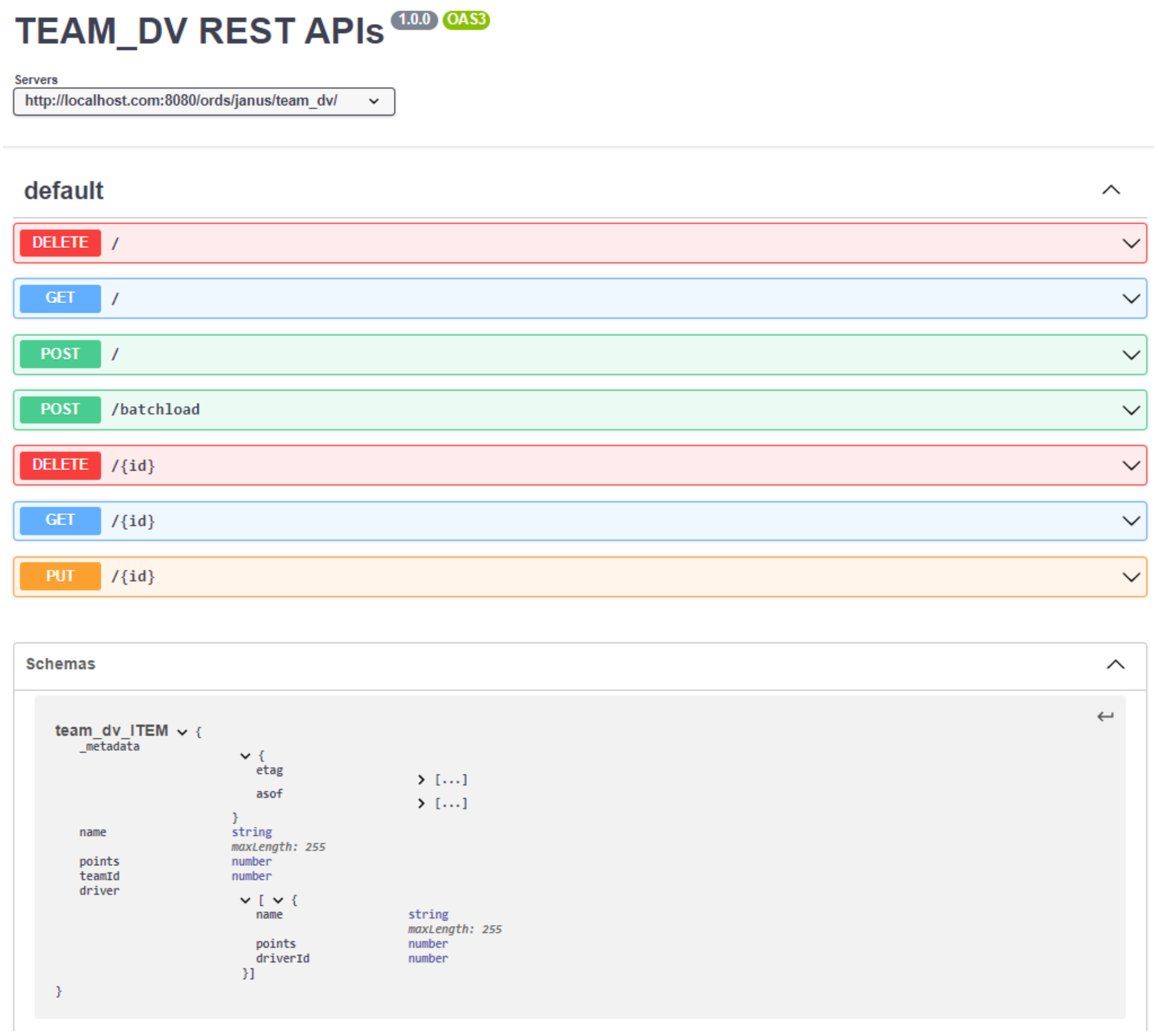
Our REST API technology includes a feature known as 'AutoREST,' where one or more objects are enabled, and REST API endpoints are automatically published. For example, a TABLE can be enabled for GET, PUT, POST, DELETE operations to get one or more rows, insert or update rows, delete rows, or even batchload multiple rows in a single request. This feature has been enhanced for 23c to include similar REST access for JSON-Relational duality views.

This tutorial will walk through the basic use cases for working with a REST Enabled JSON-Relational duality views.

If you are familiar with SQL Developer Web (also known as Database Actions), you may take advantage of its graphical user interface to point and click your way to REST Enabling your DVs, explore their REST APIs, and use the built-in OpenAPI documentation and REST Client to exercise the APIs demonstrated in this document.

The following tutorials include the SQL, PL/SQL, and cURL commands to work with the examples from your Oracle Database tool and Window CMD prompt.

For the sake of simplicity, these REST APIs are unprotected. Oracle Database REST APIs offer performance AND secure access for application developers, and it is highly recommended you protect your endpoints with the proper web privileges and roles (Docs.)


TEAM_DV REST APIs 1.0.0 OAS3

Servers
http://localhost.com:8080/ords/janus/team_dv/

default

DELETE /
GET /
POST /
POST /batchload
DELETE /{id}
GET /{id}
PUT /{id}

Schemas

```
team_dv_ITEM {
    _metadata            {
                            etag             > [...]
                            asof             > [...]
                         }
    name                 string
                         maxLength: 255
    points               number
    teamId               number
    driver               [ {
                            name             string
                                             maxLength: 255
                            points           number
                            driverId         number
                         }]
}
```

Figure 1: OpenAPI Documentation for TEAM_DV JSON-Relational duality view REST APIs

## Step 1: Create JSON-Relational Duality Views

Connect with user *JANUS* using an Oracle utility such as SQLcl, SQL Developer, or SQL*Plus.

If you already have THE TEAM, DRIVER, RACE, and DRIVER_RACE_MAP tables and TEAM_DV, DRIVER_DV, RACE_DV duality views in this schema, first drop them as follows to start with a clean slate:

```
drop view if exists team_dv;
drop view if exists race_dv;
drop view if exists driver_dv;
drop table if exists driver_race_map;
drop table if exists race;
drop table if exists driver;
drop table if exists team;
```

Create the TEAM, DRIVER, RACE, and DRIVER_RACE_MAPS tables as follows:

```
CREATE TABLE IF NOT EXISTS team
  (team_id INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
   name    VARCHAR2(255) NOT NULL UNIQUE,
   points  INTEGER NOT NULL,
   CONSTRAINT team_pk PRIMARY KEY(team_id));

CREATE TABLE IF NOT EXISTS driver
  (driver_id INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
   name      VARCHAR2(255) NOT NULL UNIQUE,
   points    INTEGER NOT NULL,
   team_id   INTEGER,
   CONSTRAINT driver_pk PRIMARY KEY(driver_id),
   CONSTRAINT driver_fk FOREIGN KEY(team_id) REFERENCES team(team_id));

CREATE TABLE IF NOT EXISTS race
  (race_id   INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
   name      VARCHAR2(255) NOT NULL UNIQUE,
   laps      INTEGER NOT NULL,
   race_date DATE,
   podium  JSON,
   CONSTRAINT   race_pk PRIMARY KEY(race_id));

CREATE TABLE IF NOT EXISTS driver_race_map
  (driver_race_map_id INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,
   race_id            INTEGER NOT NULL,
   driver_id          INTEGER NOT NULL,
   position           INTEGER,
   CONSTRAINT driver_race_map_pk  PRIMARY KEY(driver_race_map_id),
   CONSTRAINT driver_race_map_fk1 FOREIGN KEY(race_id) REFERENCES race(race_id),
   CONSTRAINT driver_race_map_fk2 FOREIGN KEY(driver_id) REFERENCES
driver(driver_id));
```

Create a trigger on the DRIVER_RACE_MAP table to populate the POINTS column in TEAM and DRIVER tables based on race results:

```
CREATE OR REPLACE TRIGGER driver_race_map_trigger
  BEFORE INSERT ON driver_race_map
  FOR EACH ROW
  DECLARE
    v_points  INTEGER;
    v_team_id INTEGER;
BEGIN
  SELECT team_id INTO v_team_id FROM driver WHERE driver_id = :NEW.driver_id;

  IF :NEW.position = 1 THEN
    v_points := 25;
  ELSIF :NEW.position = 2 THEN
    v_points := 18;
  ELSIF :NEW.position = 3 THEN
    v_points := 15;
  ELSIF :NEW.position = 4 THEN
    v_points := 12;
  ELSIF :NEW.position = 5 THEN
    v_points := 10;
  ELSIF :NEW.position = 6 THEN
    v_points := 8;
  ELSIF :NEW.position = 7 THEN
    v_points := 6;
  ELSIF :NEW.position = 8 THEN
    v_points := 4;
  ELSIF :NEW.position = 9 THEN
    v_points := 2;
  ELSIF :NEW.position = 10 THEN
    v_points := 1;
  ELSE
    v_points := 0;
  END IF;

  UPDATE driver SET points = points + v_points
    WHERE driver_id = :NEW.driver_id;
  UPDATE team SET points = points + v_points
    WHERE team_id = v_team_id;
END;
/
```

Create the RACE_DV, DRIVER_DV, and TEAM_DV JSON-Relational duality views:

```
-- Create race view, RACE_DV
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW race_dv AS
  SELECT JSON {'raceId' IS r.race_id,
               'name'   IS r.name,
               'laps'   IS r.laps WITH NOUPDATE,
               'date'   IS r.race_date,
               'podium' IS r.podium WITH NOCHECK,
               'result' IS
                 [ SELECT JSON {'driverRaceMapId' IS drm.driver_race_map_id,
                                'position'        IS drm.position,
                                UNNEST
                                  (SELECT JSON {'driverId' IS d.driver_id,
                                                'name'     IS d.name}
                                     FROM driver d WITH NOINSERT UPDATE NODELETE
                                     WHERE d.driver_id = drm.driver_id)}
                     FROM driver_race_map drm WITH INSERT UPDATE DELETE
                     WHERE drm.race_id = r.race_id ]}
    FROM race r WITH INSERT UPDATE DELETE;

-- Create driver view, DRIVER_DV
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW driver_dv AS
  SELECT JSON {'driverId' IS d.driver_id,
               'name'     IS d.name,
               'points'   IS d.points,
               UNNEST
                 (SELECT JSON {'teamId' IS t.team_id,
                               'team'   IS t.name WITH NOCHECK}
                    FROM team t WITH NOINSERT NOUPDATE NODELETE
                    WHERE t.team_id = d.team_id),
               'race'     IS
                 [ SELECT JSON {'driverRaceMapId' IS drm.driver_race_map_id,
                                UNNEST
                                  (SELECT JSON {'raceId' IS r.race_id,
                                                'name'   IS r.name}
                                     FROM race r WITH NOINSERT NOUPDATE NODELETE
                                     WHERE r.race_id = drm.race_id),
                                'finalPosition'   IS drm.position}
                    FROM driver_race_map drm WITH INSERT UPDATE NODELETE
                    WHERE drm.driver_id = d.driver_id ]}
    FROM driver d WITH INSERT UPDATE DELETE;
```

```
-- Create team view, TEAM_DV
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW team_dv AS
  SELECT JSON {'teamId'  IS t.team_id,
               'name'    IS t.name,
               'points'  IS t.points,
               'driver'  IS
                 [ SELECT JSON {'driverId' IS d.driver_id,
                                'name'     IS d.name,
                                'points'   IS d.points WITH NOCHECK}
                     FROM driver d WITH INSERT UPDATE
                     WHERE d.team_id = t.team_id ]}
    FROM team t WITH INSERT UPDATE DELETE;
```

Enable the Duality Views for REST APIs:

```
BEGIN
    ORDS.ENABLE_OBJECT(
        P_ENABLED        => TRUE,
        P_SCHEMA         => 'JANUS',
        P_OBJECT         =>  'DRIVER_DV',
        P_OBJECT_TYPE    => 'VIEW',
        P_OBJECT_ALIAS   => 'driver_dv',
        P_AUTO_REST_AUTH => FALSE
    );
    COMMIT;
END;
/

BEGIN
    ORDS.ENABLE_OBJECT(
        P_ENABLED        => TRUE,
        P_SCHEMA         => 'JANUS',
        P_OBJECT         =>  'RACE_DV',
        P_OBJECT_TYPE    => 'VIEW',
        P_OBJECT_ALIAS   => 'race_dv',
        P_AUTO_REST_AUTH => FALSE
    );
    COMMIT;
END;
/
BEGIN
    ORDS.ENABLE_OBJECT(
        P_ENABLED        => TRUE,
        P_SCHEMA         => 'JANUS',
        P_OBJECT         =>  'TEAM_DV',
        P_OBJECT_TYPE    => 'VIEW',
        P_OBJECT_ALIAS   => 'team_dv',
        P_AUTO_REST_AUTH => FALSE
    );
    COMMIT;
END;
/
```

Note the P_OBJECT_ALIAS parameter defines the URI, making the duality views addressable in the REST APIs demonstrated going forward. These aliases are customizable and can be used to hide database implementation details from your REST API consumers.

## Step 2: List all documents in a duality view

To list all documents in a duality view, issue a GET request with the corresponding collection name as the trailing step. For example, to list all documents in the DRIVER_DV duality view, run:

```
curl -X GET http://localhost:8080/ords/janus/driver_dv/
```

The command outputs the following (see next page, output abbreviated for brevity):

```
{
    "items": [],
    "hasMore": false,
    "limit": 25,
    "offset": 0,
    "count": 0,
    "links": […]
}
```

The “items” array contains the JSON documents from the duality view. It is empty, as the duality views (or, more precisely, their underlying tables) have not been loaded.

Oracle Database REST APIs are paginated by default, so "offset" and "limit" fields refer to the offset of the results and the maximum number of results returned per request (by default, this is set to 25 as indicated by the value of "limit" above).

Also included in the response are links to common read and write operations that can be performed on the duality view collection. The contents of this “links” array is not shown in the above output for brevity.

## Step 3: Populate duality views

### Single document insert

To insert a single document use POST, with the document content provided as the body. Note that while the write operations in this tutorial are performed on duality views, the actual data is transparently written to the underlying tables (duality views themselves do not hold any data). This operation uses teamMercedes.json file with the following data for team “Mercedes”:

```
{  "teamId": 2,

  "name": "Mercedes",
  "points": 0,
  "driver": [
    {
      "driverId": 105,
      "name": "George Russell",
      "points": 0
    },
    {
      "driverId": 106,
      "name": "Lewis Hamilton",
      "points": 0
    }] }
```

To insert this data, run:

```
curl -i -X POST --data-binary @teamMercedes.json  -H "Content-Type:
application/json"  http://localhost:8080/ords/janus/team_dv/
```

The command outputs the new JSON document and its corresponding "etag", the document portion omitted for brevity:

```
HTTP/1.1 201 Created
...
],
      "_metadata": {
            "etag": "536001F31A8718819AEEF28EC20D8677",
            "asof": "000000000032C39C"
      },
     "links": [
       {
        "rel": "self",
        "href": " http://localhost:8080/ords/janus/team_dv/2"
       },
…
```

Oracle Database generates an eTag for each document inserted into the duality view. This information is returned under the "_metadata" field in the response body. These eTags are critical for upcoming examples involving updates using PUTs.

In addition, the response includes a "Content-Location"  header redirecting the client to the new resource, in this case the Mercedes team. The "self" link shows where to address the new record. The record locator is derived from the TEAM primary key column, in this case, "teamId."

### Bulk insert

You can also insert multiple records into a duality view in one request, using POST to the REST Enabled view with the /batchload URI and an accompanying array of JSON objects in the request body. This operation uses team.json file. The file contains a JSON array of two objects, each corresponding to a particular team.

The contents of this file are:

```
[  { "teamId": 301,
    "name": "Red Bull",
    "points": 0,
    "driver": [
      {
        "driverId": 101,
        "name": "Max Verstappen",
        "points": 0
      },
      {
        "driverId": 102,
        "name": "Sergio Perez",
        "points": 0
      }
    ]
  },
  { "teamId": 302,
    "name": "Ferrari",
    "points": 0,
    "driver": [
      {
        "driverId": 103,
        "name": "Charles Leclerc",
        "points": 0
      },
      {
        "driverId": 104,
        "name": "Carlos Sainz Jr",
        "points": 0
      }
    ]
  }]
```

The following POST command with “insert” custom action loads these objects into the team_dv duality view:

```
curl -i -X POST --data-binary @team.json -H "Content-Type: application/json"
http://localhost:8080/ords/janus/insert/team_dv/batchload
```

The AutoREST Duality View API includes a POST /batchload endpoint for ‘batch loading’ multiple JSON documents as rows in the view.

A successful POST returns response code 200. The response body is a report showing the number of records processed, errors that failed, and errors successfully committed to the underlying duality views.

```
HTTP/1.1 200 OK
...
#INFO Number of rows processed:  2
#INFO Number of rows in error:  0
#INFO Last row processed in final committed batch: 2
SUCCESS: Processed without errors
```

Duality view that contains information for each auto race is loaded similarly. Use race.json file that contains:

```
[{  "raceId": 201,
    "name": "Bahrain Grand Prix",
    "laps": 57,
    "date": "2022-03-20T00:00:00",
    "podium": {}
  },
  {
     "raceId": 202,
     "name": "Saudi Arabian Grand Prix",
     "laps": 50,
     "date": "2022-03-27T00:00:00",
     "podium": {}
  },
  {
     "raceId": 203,
     "name": "Australian Grand Prix",
     "laps": 58,
     "date": "2022-04-09T00:00:00",
     "podium": {}}]
```

Note that the podium info for each race is empty. It is populated by other operations later in this tutorial.

To load this data, run:

```
curl -i -X POST --data-binary @race.json -H "Content-Type: application/json"
http://localhost:8080/ords/race_dv/batchload
```

The command outputs the following ("etag" values abbreviated for brevity):

```
HTTP/1.1 200 OK
#INFO Number of rows processed: 3
#INFO Number of rows in error: 0
#INFO Last row processed in final committed batch: 3
SUCCESS: Processed without errors
```

## Step 4: See effects of populating a duality view

Populating a duality view automatically updates data shown in related duality views, by updating their underlying tables. For example, in the previous step documents were inserted into the TEAM_DV duality view. This duality view joins the TEAM table with the DRIVER table, so on insert into this duality view both the TEAM table as well as the DRIVER table are populated. If you now list the contents of the DRIVER_DV duality view, which is based on the DRIVER table, it has records as well.

To list the contents of the DRIVER_DV duality view run:

```
curl -X GET http://localhost:8080/ords/janus/driver_dv/
```

The command outputs the following (some fields, such as “links” to other operations and the “etag” for drivers 2-6 are omitted for brevity):

```
{
      "items": [{
                  "driverId": 105,
                  "name": "George Russell",
                  "points": 0,
                  "teamId": 2,
                  "team": "Mercedes",
                  "race": [],
                  "_metadata": {
                        "etag": "A8BB1825F6218EC0D300671173540597",
                        "asof": "000000000032FAFD"
                  }
            },
            {
                  "driverId": 106,
                  "name": "Lewis Hamilton",
                  "points": 0,
                  "teamId": 2,
                  "team": "Mercedes",
                  "race": [],
                  …
            },
            {
                  "driverId": 101,
                  "name": "Max Verstappen",
                  "points": 0,
                  "teamId": 301,
                  "team": "Red Bull",
                  "race": [].
                  …
            },
            {
                  "driverId": 102,
                  "name": "Sergio Perez",
                  "points": 0,
                  "teamId": 301,
                  "team": "Red Bull",
                  "race": [],
                  …
            },
            {
                  "driverId": 103,
                  "name": "Charles Leclerc",
                  "points": 0,
                  "teamId": 302,
                  "team": "Ferrari",
                  "race": [],
                  …
            },
            {
                  "driverId": 104,
                  "name": "Carlos Sainz Jr",
                  "points": 0,
                  "teamId": 302,
                  "team": "Ferrari",
                  "race": [],
                  …
            }
      ],
      "hasMore": false,
      "limit": 25,
      "offset": 0,
      "count": 6,
      "links": [{
    ...
      ]
}
```

Records corresponding to six drivers now appear in the DRIVER_DV duality view, as shown by the above output. For each document, the following fields are included in the result:

| Field | Description |
|---|---|
| `etag` | Document eTag. Used for optimistic locking. Automatically computed by the duality view. |
| `asof` | Table System Change Number (SCN) – can be used for Flashback queries. |

Note that the "race" field under each driver value is empty – this is because no information connecting the driver to the auto races they participated in has been entered so far. This information is added in later steps of this tutorial.

## Step 5: Find documents matching a filter, Query Parameters

Query Parameters are pattern-like queries for JSON data, also expressed in JSON. Refer to “Filtering in Queries” for an introduction.

Records satisfying specific conditions can be retrieved by supplying query parameters. Let’s saw we want to pull up a particular race, the Bahrain Grand Prix.

This operation uses query parameters with the following content:

```
{"name":{"$eq":"Bahrain Grand Prix"}}
```

This query parameter will find all records having the race name “Bahrain Grand Prix.” To retrieve records matching this filter run:

```
curl -v --location -g
"http://localhost:8080/ords/janus/race_dv/?q=%7B%22name%22%3A%7B%22%24eq%22%3A%22Ba
hrain%20Grand%20Prix%22%7D%7D"
```

The command outputs the following (abbreviated):

```
{
	"items": [{
		"raceId": 201,
		"name": "Bahrain Grand Prix",
		"laps": 57,
		"date": "2022-03-20T00:00:00",
		"podium": {},
		"result": [],
		"_metadata": {
			"etag": "2E8DC09543DD25DC7D588FB9734D962B",
			"asof": "00000000002BECD5"
		},
		"links": [{
			"rel": "self",
			"href": "http:// localhost:8080/ords/janus/race_dv/201"
		}]
	}
```

Note that the podium and result fields in the document are the empty object and the empty array, respectively. This is again because that information for the "Bahrain Grand Prix" has not been entered. In the next step, this information is added.

Note also that the individual race can be retrieved by the “raceId” attribute. Since the underlying RACE table has a primary key defined on RACE_ID, the corresponding AutoREST enabled Duality View’s “raceID’ attribute can be used as a key for working with specific races.

## Step 6: Replace a record identified by the Duality View ID (table Primary Key)

You can replace a document with a given primary key on the underlying table.

To add more information about the "Bahrain Grand Prix" autorace found in the previous step, replace the document for this race with a new document that contains additional information.

The "Bahrain Grand Prix" document is identified by the ID `201,` as highlighted in the result of the preceding step. This operation uses updateRace.json file with the following content (the generated “etag” value, also highlighted in the previous step, is supplied as the “etag” field in this content):

```
{ "_metadata":{
      "etag":"2E8DC09543DD25DC7D588FB9734D962B"
   },
  "raceId": 201,
  "name": "Bahrain Grand Prix",
  "laps": 57,
  "date": "2022-03-20T00:00:00",
  "podium": {
    "winner": {
      "name": "Charles Leclerc",
      "time": "01:37:33.584"
    },
    "firstRunnerUp": {
      "name": "Carlos Sainz Jr",
      "time": "01:37:39.182"
    },
    "secondRunnerUp": {
      "name": "Lewis Hamilton",
      "time": "01:37:43.259"
    }
  },
  "result": [
    {
      "driverRaceMapId": 3,
      "position": 1,
      "driverId": 103,
      "name": "Charles Leclerc"
    },
    {
      "driverRaceMapId": 4,
      "position": 2,
      "driverId": 104,
      "name": "Carlos Sainz Jr"
    },
    {
      "driverRaceMapId": 9,
      "position": 3,
      "driverId": 106,
      "name": "Lewis Hamilton"
    },
    {
      "driverRaceMapId": 10,
      "position": 4,
      "driverId": 105,
      "name": "George Russell"
    }]
}
```

This replacement document has "podium" and "result" information.

To replace the target document, use PUT with the ID as the trailing step:

```
curl -i -X PUT --data-binary @updateRace.json  -H "Content-Type: application/json"
http://localhost:8080/ords/janus/race_dv/201
```

The command returns HTTP status code 200 OK, with a Content-Location, of the document. This allows the client to navigate to the race, returning the new results based on the PUT. The race data from the response has been omitted for brevity.

```
HTTP/1.1 200 OK
...
ETag: 20F7D9F0C69AC5F959DCA819F9116848
...
```

The updated eTag is also included in the response header.

Note that the “etag” value supplied in the content is used for “out of the box” optimistic locking, to prevent the well-known “lost update” problem that can occur with concurrent operations. During the replace by ID operation, the database checks that the eTag provided in the replacement document matches the latest eTag of the target duality view document. If the eTags do not match, which can occur if another concurrent operation updated the same document, an error is thrown. In case of such an error, you can reread the updated value (including the updated eTag), and retry the replace operation again, adjusting it (if desired) based on the updated value.

To verify that an update (PUT) with a stale eTag fails, run the same command again:

```
curl -i -X PUT --data-binary @updateRace.json  -H "Content-Type: application/json"
http://localhost:8080/ords/janus/201
```

Because the PUT request body has an “etag” field value that has been obsoleted by the preceding successful replace, the command now outputs 412 (Precondition Failed) with the details of the eTag mismatch error in the response body:

```
HTTP/1.1 412 Precondition Failed
…{
       "code": "PredconditionFailed",
    "message": "Predcondition Failed",
       "type": "tag:oracle.com,2020:error/PredconditionFailed",
   "instance": "tag:oracle.com,2020:ecid/LVm-2DOIAFUkHzscNzznRg"
 }
```

Updating the race again requires a valid etag. This can be retrieved with a HEAD request.

```
curl -i -X HEAD http://localhost:8080/ords/janus/race_dv/201
```

The response is an empty body, but includes the current etag as a response header.

```
HTTP/1.1 200 OK
```

## Step 7: Another query parameter, filter example

We previously replaced the race results of the Bahrain Grand Prix into RACE_DV. When the top 3 finishers were defined, our DRIVER_RACE_MAP TRIGGER fires assigning points to their associated teams.

Let's try  a more interesting filter with our query parameters now. If we only want to return teams with a certain number of points, we can include that in a GET on the TEAMS_DV AUOTREST Enabled Duality View.

```
{"points":{"$gt":40}}
```

This query parameter will match all documents that have the points  field greater than 40. To fetch documents matching this query filter parameter run:

```
curl -v --location -g
"http://localhost:8080/ords/janus/team_dv/?q=%7B%22points%22%3A%7B%22%24gt%22%3A40%
7D%7D"
```

The command outputs the following (abbreviated):

```
HTTP/1.1 200
{
        "items": [{
                "teamId": 302,
                "name": "Ferrari",
                "points": 43,
                "driver": [{
                                "driverId": 103,
                                "name": "Charles Leclerc",
                                "points": 25
                        },
                        {
                                "driverId": 104,
                                "name": "Carlos Sainz Jr",
                                "points": 18
                        }
                ]
…
```

Team Ferrari is the only team with at least 40 points at this time.

## Step 8: Find a record identified by the primary key

You can request a record with a given primary key valud.  To fetch the record updated in the previous step, execute a GET with the ID as the trailing step of the URL:

```
curl http://localhost:8080/ords/janus/race_dv/201
```

The command returns the updated content of our race.

```
{
      "raceId": 201,
      "name": "Bahrain Grand Prix",
      "laps": 57,
      "date": "2022-03-20T00:00:00",
      "podium": {
            "winner": {
                  "name": "Charles Leclerc",
                  "time": "01:37:33.584"
            },
            "firstRunnerUp": {
                  "name": "Carlos Sainz Jr",
                  "time": "01:37:39.182"
            },
            "secondRunnerUp": {
                  "name": "Lewis Hamilton",
                  "time": "01:37:43.259"
            }
      },
      "result": [{
                  "driverRaceMapId": 3,
                  "position": 1,
                  "driverId": 103,
                  "name": "Charles Leclerc"
            },
            {
                  "driverRaceMapId": 4,
                  "position": 2,
                  "driverId": 104,
                  "name": "Carlos Sainz Jr"
            },
            {
                  "driverRaceMapId": 9,
                  "position": 3,
                  "driverId": 106,
                  "name": "Lewis Hamilton"
            },
            {
                  "driverRaceMapId": 10,
                  "position": 4,
                  "driverId": 105,
                  "name": "George Russell"
            }
      ],
      "_metadata": {
            "etag": "20F7D9F0C69AC5F959DCA819F9116848",
            "asof": "000000000033362A"
      }...
```

## Step 9: Re-parenting of sub-objects between two documents

Switching Charles Leclerc's and George Russell's teams can be done by updating the driver arrays in Mercedes and Ferrari documents of TEAM_DV duality view.

First find out the IDs of the Mercedes and Ferraris team using the following query parameter filter:

```
{"name":{"$in":["Mercedes","Ferrari"]}}
```

Run this query filter parameter as follows:

```
curl -v --location -g
"http://localhost:8080/ords/janus/team_dv/?q=%7B%22name%22%3A%7B%22%24in%22%3A%5B%2
2Mercedes%22%2C%22Ferrari%22%5D%7D%7D"
```

The command returns the following result ("links" omitted for brevity and “etag” values abbreviated):

```
{       "items": [
                {
                        "teamId": 2,
                        "name": "Mercedes",
                        "points": 27,
                        "driver": [
                                {
                                        "driverId": 105,
                                        "name": "George Russell",
                                        "points": 12
                                },
                                {
                                        "driverId": 106,
                                        "name": "Lewis Hamilton",
                                        "points": 15
                                }
                        ],
                        "_metadata": {
                                "etag": "855840B905C8CAFA99FB9CBF813992E5",
                                "asof": "000000000033449D"
                        },...
                },
                {
                        "teamId": 302,
                        "name": "Ferrari",
                        "points": 43,
                        "driver": [
                                {
                                        "driverId": 103,
                                        "name": "Charles Leclerc",
                                        "points": 25
                                },
                                {
                                        "driverId": 104,
                                        "name": "Carlos Sainz Jr",
                                        "points": 18
                                }
                        ],
                        "_metadata": {
                                "etag": "C5DD30F04DA1A6A390BFAB12B7D4F700",
                                "asof": "000000000033449D"
                        },...
                }
        ],
        "hasMore": false,
        "limit": 25,
        "offset": 0,
        "count": 2...
```

The ID for the Mercedes team is 2, and the ID for the Ferrari team is 302 (highlighted above).

updateMercedes.json file contains the replacement document for the Mercedes team:

```
{ "_metadata":{
      "etag":"855840B905C8CAFA99FB9CBF813992E5"
              },
 "teamId": 2,
  "name": "Mercedes",
  "points": 40,
  "driver": [
    {
      "driverId": 106,
      "name": "Lewis Hamilton",
      "points": 15
    },
    {
      "driverId": 103,
      "name": "Charles Leclerc",
      "points": 25
    }
  ]
}
```

updateFerrari.json file contains the replacement document for the Ferrari team:

```
{
       "_metadata": {
              "etag": "DA69DD103E8BAE95A0C09811B7EC9628"
       },
       "teamId": 302,
       "name": "Ferrari",
       "points": 30,
       "driver": [{
                     "driverId": 105,
                     "name": "George Russell",
                     "points": 12
              },
              {
                     "driverId": 104,
                     "name": "Carlos Sainz Jr",
                     "points": 18
              }
       ]
}
```

The ID for the Mercedes team can now be used to update its driver info:

```
curl -i -X PUT --data-binary @updateMercedes.json  -H "Content-Type: application/json"
http://localhost:8080/ords/janus/latest/team_dv/2
```

The operation returns http status code 200:

```
HTTP/1.1 200 OK
...
```

After this operation, Leclerc appears under Mercedes team and no longer appears under the Ferrari team.

Similarly, the ID for the Ferrari team can be used to update its driver info :

```
curl -i -X PUT --data-binary @updateFerrari.json -H "Content-Type: application/json"
http://localhost:8080/ords/janus/team_dv/302
```

The operation returns http status code 200 as well.

To show the Ferrari and Mercedes teams after the update, again execute:

```
curl -v --location -g
"http://localhost:8080/ords/janus/team_dv/?q=%7B%22name%22%3A%7B%22%24in%22%3A%5B%2
2Mercedes%22%2C%22Ferrari%22%5D%7D%7D"
```

The (abbreviated) output shown on the next page confirms that George Russell has been moved to the Ferrari team, and Charles Leclerc has been moved to the Mercedes team.

```
{
       "items": [{
                     "teamId": 2,
                     "name": "Mercedes",
                     "points": 40,
                     "driver": [{
                                   "driverId": 103,
                                   "name": "Charles Leclerc",
                                   "points": 25
                            },
                            {
                                   "driverId": 106,
                                   "name": "Lewis Hamilton",
                                   "points": 15
                            }
                     ]...
              },
              {
                     "teamId": 302,
                     "name": "Ferrari",
                     "points": 30,
                     "driver": [{
                                   "driverId": 104,
                                   "name": "Carlos Sainz Jr",
                                   "points": 18
                            },
                            {
                                   "driverId": 105,
                                   "name": "George Russell",
                                   "points": 12
                            }...
              }
       ],
       "hasMore": false,
       "limit": 25,
       "offset": 0,
       "count": 2...
```

The DRIVER_DV duality view has been updated as well. This can be verified by using the following query parameter filter:

```
{"$or" : [{"name" : {"$like" : "George%"}}, {"name" : {"$like" : "Charles%"}}]}
```

Run:

```
curl -v --location -g
"http://localhost:8080/ords/janus/driver_dv/?q=%7B%22%24or%22:%5B%7B%22name%22:%7B%22%24lik
e%22:%22George%25%22%7D%7D%2C%7B%22name%22:%7B%22%24like%22:%22Charles%25%22%7D%7D%5D%7D"
```

The command outputs (abbreviated):

```
{
      "items": [
            {
                  "driverId": 105,
                  "name": "George Russell",
                  "points": 12,
                  "teamId": 302,
                  "team": "Ferrari",
                  "race": [
                        {
                              "driverRaceMapId": 10,
                              "raceId": 201,
                              "name": "Bahrain Grand Prix",
                              "finalPosition": 4
                        }...
            },
            {
                  "driverId": 103,
                  "name": "Charles Leclerc",
                  "points": 25,
                  "teamId": 2,
                  "team": "Mercedes",
                  "race": [
                        {
                              "driverRaceMapId": 3,
                              "raceId": 201,
                              "name": "Bahrain Grand Prix",
                              "finalPosition": 1
                        }
                  ],...
            }
      ],
      "hasMore": false,
      "limit": 25,
      "offset": 0,
      "count": 2...
}
```

Charles Leclerc now has "Mercedes" as the team, and "George Russell" has "Ferrari" as the team.

## Step 10: Update a non-updateable field

From the previous command, the primary key of the Charles Leclerc record in the DRIVER_DV duality view is `103`. The "team" field is non-updateable because the team table is marked with NOUPDATE annotation in the DRIVER_DV duality view definition.

updateLeclerc.json file used in this operation contains the following data:

```
{
  "driverId": 103,
  "name": "Charles Leclerc",
  "points": 25,
  "teamId": 2,
  "team": "Ferrari",
  "race": [
    {
      "driverRaceMapId": 3,
      "raceId": 201,
      "name": "Blue Air Bahrain Grand Prix",
      "finalPosition": 1
    }
  ]
}
```

This update document will try to set the team of Leclerc back to Ferrari.

You can attempt to run the update as follows:

```
curl -i -X PUT --data-binary @updateLeclerc.json -H "Content-Type: application/json"
http://localhost:8080/ords/janus/driver_dv/103
```

Because the team is not updateable through the DRIVER_DV view, the command outputs:

```
HTTP/1.1 400 Bad Request
{
      "code": "BadRequest",
      "title": "Bad Request",
      "message": "The request could not be processed for a user defined resource",
      "o:errorCode": "ORDS-25001",
      "cause": "An error occurred when evaluating a SQL statement associated with this
resource. SQL Error Code 40940, Error Message: ORA-40940: Cannot update field 'team'
corresponding to column 'NAME' of table 'TEAM' in JSON Relational Duality View 'DRIVER_DV':
Missing UPDATE annotation or NOUPDATE annotation specified.\nORA-06512: at line 10\nORA-
06512: at line 10\n",
      "action": "Verify that the URI and payload are correctly specified for the requested
operation. If the issue persists then please contact the author of the resource",
      "type": "tag:oracle.com,2020:error/BadRequest",
      "instance": "tag:oracle.com,2020:ecid/P9D3BYJQXKGLUqy_nd-xNA"
}
```

HTTP 400 "Bad Request" is returned as the status code, and the detailed error is shown in the body of a response.

This shows that while the team of the driver can be updated using the TEAM_DV duality view, as described in the preceding step, it cannot be updated through the DRIVER_DV duality view. This illustrates how the same underlying relational data can be made updateable or non-updateable, as needed for the different use-cases, by creating different duality views.

## Step 11: Delete documents

### Target document identified by document key

You can delete a document with a given document key, aka ID. For example, in Step 7 you replaced "Bahrain Grand Prix" document by its primary key value, which is `201`

To delete this document, run:

```
curl --request DELETE --url http://localhost:8080/ords/janus/race_dv/201
```

The command outputs:

```
HTTP/1.1 200 OK
{
      "rowsDeleted": 1
}
```

### Target document(s) identified by query parameter filter

You can also delete records  matching a query parameter filter.

Find the races where RACE_ID=202. The query parameter filter would be:

```
 {"raceId":{"$eq":202}}
```

To delete a race document matching this query filter parameter, run:

```
curl -v --location -g -X DELETE
"http://localhost:8080/ords/janus/race_dv/?q={%22raceId%22:{%22$eq%22:202}}"
```

The operation returns http error code 200, indicating a successful delete. The returned body shows the number of rows deleted (1 in this case.)

```
HTTP/1.1 200 OK
{
      "rowsDeleted": 1
}
```

## Conclusion

This tutorial introduced you to REST APIs for working with JSON-Relational duality views. The following operations were covered:

- Enabling a JSON-Relational duality view for REST API access
- Fetching a record by its primary key
- Fetching documents using query parameter filters
- Fetching all records in a duality view
- Replacing a record in a duality view, with optional eTag checking
- Deleting a document identified by its document key
- Deleting documents identified by a query filter parameter

For more information on JSON-Relational duality views, see JSON-Relational Duality Developer's guide.

For more information on the AutoREST APIs support for Oracle Database objects used for this tutorial, see the ORDS Developer's Guide (Docs.)

For an overview of query parameter filters using JSON syntax see Filtering in Queries (Docs.)

### Connect with us

Call +**1.800.ORACLE1** or visit **oracle.com**. Outside North America, find your local office at: **oracle.com/contact**.

REST APIs for Oracle

twitter.com/oracleords