ORACLE

# Working with JSON-Relational Duality Views using Oracle Database API for MongoDB

03/29/23, Version 1.1

## Purpose statement

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

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

```
/*
** Copyright (c) 2023 Oracle and/or its affiliates
** The Universal Permissive License (UPL), Version 1.0
**
** Subject to the condition set forth below, permission is hereby granted to any
** person obtaining a copy of this software, associated documentation and/or data
** (collectively the "Software"), free of charge and under any and all copyright
** rights in the Software, and any and all patent rights owned or freely
** licensable by each licensor hereunder covering either (i) the unmodified
** Software as contributed to or provided by such licensor, or (ii) the Larger
** Works (as defined below), to deal in both
**
** (a) the Software, and
** (b) any piece of software and/or hardware listed in the lrgrwrks.txt file if
** one is included with the Software (each a "Larger Work" to which the Software
** is contributed by such licensors),
**
** without restriction, including without limitation the rights to copy, create
** derivative works of, display, perform, and distribute the Software and make,
** use, sell, offer for sale, import, export, have made, and have sold the
** Software and the Larger Work(s), and to sublicense the foregoing rights on
** either these or other terms.
**
** This license is subject to the following condition:
** The above copyright notice and either this complete permission notice or at
** a minimum a reference to the UPL must be included in all copies or
** substantial portions of the Software.
**
** THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
** IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
** FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
** AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
** LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
** OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
** SOFTWARE.
*/
```

# Table of contents

JSON-Relational Duality View is a revolutionary Oracle Database 23c 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 using Formula-1 car-racing season data. Specifically, it focuses on using the Oracle Database API for MongoDB to work with JSON-Relational duality views. Note that after duality views are created and exposed as collections, you work with such collections *normally*, as you would with *regular collections* (not based on duality views)[1], while reaping all of the additional benefits of duality views. And of course, standard MongoDB drivers and tools can be used. This tutorial uses the Mongo shell, and shows a subset of the full breadth of operations possible, such as inserts, finds, updates, replaces, deletes, projections, and so on.

A prerequisite for this tutorial is Introduction to Car-Racing Duality Views Example chapter, also in in JSON-Relational Duality Developer's Guide. It describes the use-case and duality views you work with in this document.

For more information on Oracle Database API for MongoDB, refer to Oracle Database API for MongoDB. Within that guide, Mongo DB API Collections Supported by JSONRelational Duality Views (section 1.4) is particularly relevant for this tutorial.

## Prerequisites

1. Ensure that you have Oracle database 23c installed and running on a port. Ensure that compatible parameter is set to 23.0.0.0.
2. Download and install the latest MongoDB shell (mongosh).
3. Download ORDS version 23.1 (or higher), and install it with Oracle API for MongoDB enabled, as described in Step 1 of the Getting Started (section 9.1) of Oracle API for MongoDB Support chapter in the ORDS documentation.
4. Create and configure a database user as described in Step 2 of the same Getting Started section. Also grant the user the RESOURCE role:

   ```
   grant resource to foo;
   ```

   The step creates and configures schema `foo`. If you would like to choose a different schema to hold the tables, duality views, and collections used in this tutorial, configure it in the same way.

5. Connect to Oracle API for MongoDB with mongosh, as described in Step 3 of the same Getting Started section. All steps below, beginning with "Step 2", use mongosh.

## Step 1: Create JSON-Relational Duality Views

Connect to the schema `foo` (or whatever schema you picked to host the tables, duality views, and collections) using SQLPlus, SQLcl, SQLDeveloper, or another tool.

[1] Limitations of duality view based collections wrt support for regular collections in Oracle Database API for MongoDB are listed in the appendix

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

Create the team, driver, race, and driver_race_maps tables as follows:

```
CREATE TABLE 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 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 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 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 duality views:

```
-- Create race view, RACE_DV
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW race_dv AS
  SELECT JSON {'_id' 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 {'_id' 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 {'_id'  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;
```

A duality view that is compatible with the MongoDB API always has field _id as its document identifier. Its value specifies the document fields whose values are the primary-key columns of the root table that underlies the duality view. See Mongo DB API Collections Supported by JSONRelational Duality Views for more information on how the _id field should be specified in general, for views other than the ones listed in this tutorial.

Next, create a collection on top of each duality view. This is necessary to be able to access duality views through the Oracle Database API for MongoDB. A collection is an abstraction, typically found in NoSQL document databases - it holds JSON documents, and supports CRUD operations without requiring the application developer to write SQL. In Oracle Database, collections are backed by regular tables or views. Because of this, while performing read and write operations on the collection from Mongo API for Oracle Database does not require any SQL, collection data can be accessed through SQL as well (e.g. for analytics or reporting). In the case of duality view collections, the actual data is stored in the tables underlying the duality view.

Create three collections, each on top of the corresponding duality view, by issuing the following PLSQL:

```
declare
  col soda_collection_t;
begin
  col := dbms_soda.create_dualv_collection('team_dv', 'TEAM_DV');
  col := dbms_soda.create_dualv_collection('driver_dv', 'DRIVER_DV');
  col := dbms_soda.create_dualv_collection('race_dv', 'RACE_DV');
end;
```

create_dualv_collection function takes a duality view collection name as the first argument, and the duality view name as the second argument.

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

This step as well as subsequent steps in this tutorial will use the Mongo shell (mongosh).

Use find() method to list all documents in the duality view. For example, to list all documents in the driver_dv duality view, execute:

```
db.driver_dv.find();
```

The command does not return any results since duality views (or, more precisely, their underlying tables) have not been populated with data.

## Step 3: Populate duality views

### Single document insert

To insert a single document use `insertOne()` method with the JSON document content as the argument. 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).

To insert a JSON document for the Mercedes team into team_dv, execute:

```
db.team_dv.insertOne(
{
  "_id": 2,
  "name": "Mercedes",
  "points": 0,
  "driver": [
    {
      "driverId": 105,
      "name": "George Russell",
      "points": 0
    },
    {
      "driverId": 106,
      "name": "Lewis Hamilton",
      "points": 0
    }
  ]
}
);
```

The command outputs the following:

```
{ acknowledged: true, insertedId: 2 }
```

When doing inserts into duality views, the value(s) provided for the fields must of course be insertable into the underlying columns, which means that their data types must be compatible with the column types. For example, if the duality view maps the field _id to a column of SQL type NUMBER, the _id value of a document you insert must be numeric or convertible to a numeric vlaue. Otherwise, an error is raised for the insertion attempt.

### Bulk insert

You can also insert multiple documents into a duality view in one shot using `insertMany()`, with the array of documents as the argument.

To insert JSON documents for Redbull and Ferrari teams into team_dv duality view, execute:

```
db.team_dv.insertMany(
[{
  "_id": 301,
  "name": "Red Bull",
  "points": 0,
  "driver": [
    {
      "driverId": 101,
      "name": "Max Verstappen",
      "points": 0
    },
    {
      "driverId": 102,
      "name": "Sergio Perez",
      "points": 0
    }
  ]
},
{
  "_id": 302,
  "name": "Ferrari",
  "points": 0,
  "driver": [
    {
      "driverId": 103,
      "name": "Charles Leclerc",
      "points": 0
    },
    {
      "driverId": 104,
      "name": "Carlos Sainz Jr",
      "points": 0
    }
  ]
}]);
```

`insertMany()` causes the array to be inserted as a set of documents, rather than as a single document.

A successful `insertMany()` operation returns a response with the field acknowledged is set to true and the inserted IDs.

```
{ acknowledged: true, insertedIds: { '0': 301, '1': 302 } }
```

Duality view race_dv that contains information for each auto race is loaded similarly.
(Note that the podium info for each race is empty. It is populated by other operations later in this tutorial.)

```
db.race_dv.insertMany([
{
  "_id": 201,
  "name": "Bahrain Grand Prix",
  "laps": 57,
  "date": "2022-03-20T00:00:00",
  "podium": {}
},
{
  "_id": 202,
  "name": "Saudi Arabian Grand Prix",
  "laps": 50,
  "date": "2022-03-27T00:00:00",
  "podium": {}
},
{
  "_id": 203,
  "name": "Australian Grand Prix",
  "laps": 58,
  "date": "2022-04-09T00:00:00",
  "podium": {}
}]);
```

The command outputs:

```
{ acknowledged: true, insertedIds: { '0': 201, '1': 202, '2': 203 } }
```

Similarly to the preceding step, if you list the contents of the driver_dv, it will additionally show two documents for Mercedes team drivers.

## 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 documents as well.

To list the contents of the driver_dv duality view run:

```
db.driver_dv.find();
```

The command outputs the following:

```
[
  {
    _id: 105,
    name: 'George Russell',
    points: 0,
    teamId: 2,
    team: 'Mercedes',
    race: [],
    _metadata: {
      etag: Binary(Buffer.from("a8bb1825f6218ec0d300671173540597", "hex"), 0),
      asof: Binary(Buffer.from("...", "hex"), 0)
    }
  },
  {
    _id: 106,
    name: 'Lewis Hamilton',
    points: 0,
    teamId: 2,
    team: 'Mercedes',
    race: [],
    _metadata: {
      etag: Binary(Buffer.from("d3ff3213793e306204bb5e5060368e41", "hex"), 0),
      asof: Binary(Buffer.from("…", "hex"), 0)
    }
  },
  {
    _id: 101,
    name: 'Max Verstappen',
    points: 0,
    teamId: 301,
    team: 'Red Bull',
    race: [],
    _metadata: {
      etag: Binary(Buffer.from("f9d9815dff27879f61386cfd1622b065", "hex"), 0),
      asof: Binary(Buffer.from("…", "hex"), 0)
    }
  },
  {
    _id: 102,
    name: 'Sergio Perez',
    points: 0,
    teamId: 301,
    team: 'Red Bull',
    race: [],
    _metadata: {
      etag: Binary(Buffer.from("9865a29dee5f5754674a9b1d2ec58730", "hex"), 0),
      asof: Binary(Buffer.from("...", "hex"), 0)
    }
  },
```

```
  {
    _id: 103,
    name: 'Charles Leclerc',
    points: 0,
    teamId: 302,
    team: 'Ferrari',
    race: [],
    _metadata: {
      etag: Binary(Buffer.from("c47a91f57bf3cf45f0d8284240399a90", "hex"), 0),
      asof: Binary(Buffer.from("...", "hex"), 0)
    }
  },
  {
    _id: 104,
    name: 'Carlos Sainz Jr',
    points: 0,
    teamId: 302,
    team: 'Ferrari',
    race: [],
    _metadata: {
      etag: Binary(Buffer.from("363ec6bbf0fadd913b219482959da39d", "hex"), 0),
      asof: Binary(Buffer.from("...", "hex"), 0)
    }
  }
]
```

Documents 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 |
|---|---|
| _id | Document-identifier, aka _id. |
| etag (under _metadata) | Document eTag. Used for optimistic locking. Automatically computed by the duality view. |
| asof (under _metadata) | System change number (SCN) at the time of fetch. These values are replaced by "..." in the outputs shown in this tutorial, since they will be different depending on the DB instance/time of fetch. |

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 (with optional projection of document fields and sorting).

As usual, documents can be fetched by supplying a filter. Only documents matching the filter will be returned.

For example, the following filter will match the document in race_dv that has the _id field set to 201:

```
db.race_dv.find({_id:201});
```

The command outputs the following:

```
[
  {
    _id: 201,
    name: 'Bahrain Grand Prix',
    laps: 57,
    date: ISODate("2022-03-20T00:00:00.000Z"),
    podium: {},
    result: [],
    _metadata: {
      etag: Binary(Buffer.from("2e8dc09543dd25dc7d588fb9734d962b", "hex"), 0),
      asof: Binary(Buffer.from("...", "hex"), 0)
    }
  }
]
```

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.

Projections can be used to include specific fields or exclude specific fields in the returned documents. For example, to return only the "name" and "team" fields in the documents fetched from driver_dv, run:

```
db.driver_dv.find({}, {"name" : 1, "team" : 1});
```

The command outputs:

```
[
  { _id: 105, name: 'George Russell', team: 'Mercedes' },
  { _id: 106, name: 'Lewis Hamilton', team: 'Mercedes' },
  { _id: 101, name: 'Max Verstappen', team: 'Red Bull' },
  { _id: 102, name: 'Sergio Perez', team: 'Red Bull' },
  { _id: 103, name: 'Charles Leclerc', team: 'Ferrari' },
  { _id: 104, name: 'Carlos Sainz Jr', team: 'Ferrari' }
]
```

As usual, _id can be suppressed from the output by specifying `"_id" : 0` as part of the projection:

```
db.driver_dv.find({}, {"_id" : 0, "name" : 1, "team" : 1});
```

Output is omitted for brevity.

Sorting of the returned documents can be requested. To add ascending sort on team field to the previous command, run:

```
db.driver_dv.find({}, {"_id" : 0, "name" : 1, "team" : 1}).sort({"team" : 1});
```

The command outputs:

```
[
  { name: 'Charles Leclerc', team: 'Ferrari' },
  { name: 'Carlos Sainz Jr', team: 'Ferrari' },
  { name: 'George Russell', team: 'Mercedes' },
  { name: 'Lewis Hamilton', team: 'Mercedes' },
  { name: 'Max Verstappen', team: 'Red Bull' },
  { name: 'Sergio Perez', team: 'Red Bull' }
]
```

## Step 6: Replace a document

To add results information for the "Bahrain Grand Prix" autorace found in the previous step, replace the document for this race with a new document containing results information.

The "Bahrain Grand Prix" document is identified by the `_id : 201,` as highlighted in the result of the preceding step.
This replacement document has "podium" and "result" information.

To replace a specific target document, use `replaceOne()` method (see next page).

```
db.race_dv.replaceOne(
{_id: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"
    }
  ],
  "etag" : Binary(Buffer.from("2e8dc09543dd25dc7d588fb9734d962b", "hex"), 0)});
```

The command returns the following, indicating a successful replace:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

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 see that a replace using an eTag that is not the most recent fails, run the same command again.

Because document content has an "etag" field value that has been obsoleted by the preceding successful replace, the command now returns MongoServerError with the details of the eTag mismatch:

```
MongoServerError: ORA-42699: Cannot update JSON Relational Duality View 'RACE_DV': The ETAG of
document with ID 'FB03C2030200' in the database did not match the ETAG passed in.
```

To fetch the document updated earlier in this step, execute another `find()`:

```
db.race_dv.find({_id:201});
```

The command returns the updated content of this document, which includes "_id" and the updated "etag" (the eTag changed as part of the preceding replace):

```
[
  {
    _id: 201,
    name: 'Bahrain Grand Prix',
    laps: 57,
    date: ISODate("2022-03-20T00:00:00.000Z"),
    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: Binary(Buffer.from("20f7d9f0c69ac5f959dca819f9116848", "hex"), 0),
      asof: Binary(Buffer.from("...", "hex"), 0)
    }
  }
]
```

### Step 7: Update specific fields in the document

As usual, for updating specific document fields, use `updateOne()/updateMany()` method. Various update operators, such as `$set`, can be used.

For example, to add sponsor information to the "Bahrain Grand Prix" race name, run:

```
db.race_dv.updateOne({name : "Bahrain Grand Prix"},{$set : {name : "Blue Air Bahrain Grand Prix"}});
```

The command the following JSON, indicating that the update was successful:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Fetch the updated document by issuing:

```
db.race_dv.find({_id:201});
```

Output is omitted for brevity, but command returns the updated content of this document (the "name" field is now set to "`Blue Air Bahrain Grand Prix`", and the "etag" field is updated).

### Step 8: Re-parenting of sub-objects between two documents

Switching Charles Leclerc's and George Russell's teams can be done by replacing Mercedes and Ferrari documents of team_dv duality view, with new documents that contain the updated driver info.

```
db.team_dv.replaceOne(
{"name": "Mercedes"},
{
  "name": "Mercedes",
  "points": 40,
  "driver": [
    {
      "driverId": 106,
      "name": "Lewis Hamilton",
      "points": 15
    },
    {
      "driverId": 103,
      "name": "Charles Leclerc",
      "points": 25
    }
  ]
});
```

The operation returns the following JSON:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

After this operation, Leclerc appears under Mercedes team and no longer appears under the Ferrari team. This can be verified by running:

```
db.team_dv.find({"name" : "Ferrari"});
```

Output of this command is omitted for brevity.

Then the Ferrari team can be similarly updated:

```
db.team_dv.replaceOne(
{"name": "Ferrari"},
{
  "name": "Ferrari",
  "points": 30,
  "driver": [
    {
      "driverId": 105,
      "name": "George Russell",
      "points": 12
    },
    {
      "driverId": 104,
      "name": "Carlos Sainz Jr",
      "points": 18
    }
  ]
}
);
```

The operation returns similar JSON as previous command.

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

```
db.team_dv.find({"name" : {"$in" : ["Mercedes", "Ferrari"]}});
```

The 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.

```
[
  {
    _id: 2,
    name: 'Mercedes',
    points: 40,
    driver: [
      { driverId: 103, name: 'Charles Leclerc', points: 25 },
      { driverId: 106, name: 'Lewis Hamilton', points: 15 }
    ],
    _metadata: {
      etag: Binary(Buffer.from("9e266cd7554a89663b73b9977b1f967c", "hex"), 0),
      asof: Binary(Buffer.from("...", "hex"), 0)
    }
  },
  {
    _id: 302,
    name: 'Ferrari',
    points: 43,
    driver: [ { driverId: 104, name: 'Carlos Sainz Jr', points: 18 } ],
    _metadata: {
      etag: Binary(Buffer.from("da69dd103e8bae95a0c09811b7ec9628", "hex"), 0),
      asof: Binary(Buffer.from("...", "hex"), 0)
    }
  }
]
```

The drivers duality view has been updated as well. This can be verified by running the following filter:

```
db.driver_dv.find({"$or" : [{"name" : "George Russell"}, {"name" : "Charles Leclerc"}]});
```

The command outputs:

```
[
  {
    _id: 105,
    name: 'George Russell',
    points: 12,
    race: [
      {
        driverRaceMapId: 10,
        raceId: 201,
        name: 'Blue Air Bahrain Grand Prix',
        finalPosition: 4
      }
    ],
    _metadata: {
      etag: Binary(Buffer.from("1a16a5fbcf5a493a9bd4e73412dceb1c", "hex"), 0),
      asof: Binary(Buffer.from("...", "hex"), 0)
    }
  },
  {
    _id: 103,
    name: 'Charles Leclerc',
    points: 25,
    teamId: 2,
    team: 'Mercedes',
    race: [
      {
        driverRaceMapId: 3,
        raceId: 201,
        name: 'Blue Air Bahrain Grand Prix',
        finalPosition: 1
      }
    ],
    _metadata: {
      etag: Binary(Buffer.from("fcd4cec63897f60dea1ec2f64d3ce53a", "hex"), 0),
      asof: Binary(Buffer.from("...", "hex"), 0)
    }
  }
]
```

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

## Step 9: Update a non-updateable field

From the previous command, the _id of Charles Leclerc in driver_dv duality view is 103. The "team" field in the driver_dv duality view is non-updateable because it maps to a column in the team table, which is marked with NOUPDATE annotation.

This update operation will try to set the team of Leclerc back to Ferrari, using the driver_dv duality view. You can attempt to run the update as follows:

```
db.driver_dv.updateOne({"_id” : 103} , {$set : {"team" : "Ferrari"}});
```

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

```
MongoServerError: 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.
```

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 10: Delete documents

As usual, you can also delete documents matching a filter, using deleteOne and deleteMany.

For example, to delete a race document with `_id 202`, run:

```
db.race_dv.deleteOne({_id:202});
```

The command outputs the following, indicating a successful delete. The output shows the number of items deleted (1 in this case. In general can be any number between 0 and the total number of documents in the collection, depending on how many documents match the filter).

```
{"count":1,"itemsDeleted":1}
```

## Conclusion

This tutorial introduced you to using Oracle Database API for MongoDB to work with collections back by JSON-Relational duality views. The following operations were covered:

- Inserting documents
- Listing all documents
- Finding documents with filters
- Projecting document fields
- Sorting documents
- Replacing a document with optional eTag checking (via eTag supplied in the replacement content)
- Updating specific document fields
- Deleting documents

For more information on JSON-Relational duality views, see JSON-Relational Duality Developer's Guide. For more information on Oracle Database API for MongoDB, see Oracle Database API for MongoDB

Also, see a complementary "Working with JSON Relational Duality Views using SQL tutorial". It has steps analogous to the steps in this tutorial, but it is using SQL instead of Oracle Database API for MongoDB.

.

## Appendix

### Limitations

Duality view support with Oracle Database API for MongoDB has the following limitations (in addition to the Oracle Database API for MongoDB general limitations):

- upsert option (in update and findAndModify operations) is not supported
- arrayFilters option (in update and findAndModify operations) is not supported

### Alternative syntax for creating duality views

The SQL below shows creating duality views using an alternative syntax based on GraphQL. This syntax is currently only supported in SQLPlus (not in other tools such as SQLcl, SQLDeveloper, etc).

```
CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW race_dv AS
  race @insert @update @delete
  {
    _id : race_id
    name   : name
    laps   : laps @noUpdate
    date   : race_date
    podium : podium @noCheck
    result : driver_race_map @insert @update @delete
    [
     {
      driverRaceMapId : driver_race_map_id
      position        : position
      driver @noInsert @update @noDelete @unnest
      {
        driverId : driver_id
        name     : name
      }
     }
    ]
  };

CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW driver_dv AS
  driver @insert @update @delete
  {
    _id : driver_id
    name     : name
    points   : points
    team @noInsert @noUpdate @noDelete @unnest
    {
      teamId : team_id
      team   : name @noCheck
    }
    race : driver_race_map @insert @update @noDelete
```

```
    [
     {
      driverRaceMapId : driver_race_map_id
      race @noInsert @noUpdate @noDelete @unnest
      {
        raceId : race_id
        name   : name
      }
      finalPosition   : position
     }]};

CREATE OR REPLACE JSON RELATIONAL DUALITY VIEW team_dv AS
  team @insert @update @delete
  {
    _id : team_id
    name   : name
    points : points
    driver : driver @insert @update
    [
     {
      driverId : driver_id
      name     : name
      points   : points @noCheck
     }
    ]
  };
```