Oracle® REST Data Services

SODA for REST Developer's Guide

Release 17.2.4

E58123-14

April 2017

Oracle REST Data Services SODA for REST Developer's Guide, Release 17.2.4

E58123-14

Primary Author: Drew Adams

Contributing Authors: Douglas McMahon, Sheila Moore, Maxim Orgiyan, Josh Spiegel

Contents

Preface xi

Audience xi
Documentation Accessibility xi
Related Documents xi
Conventions xi

1 SODA for REST Overview

1.1 Overview of the Representational State Transfer (REST) Architectural Style 1-2

2 Installing SODA for REST 2-1

3 Using SODA for REST

3.1 Creating a New Collection 3-2
3.2 Getting the List of Available Collections 3-3
3.3 Deleting a Collection 3-4
3.4 Inserting a Document into a Collection 3-4
3.5 Retrieving a Document from a Collection 3-5
3.6 Deleting a Document from a Collection 3-6
3.7 Bulk-Inserting Documents from a JSON Array 3-6
3.8 Listing the Documents in a Collection 3-7
3.9 Updating a Document in a Collection 3-9
3.10 Using a Filter Specification to Select Documents From a Collection 3-10
3.10.1 QBE.1.json 3-11
3.10.2 QBE.2.json 3-11
3.10.3 QBE.3.json 3-12
3.10.4 QBE.4.json 3-12

4 SODA for REST HTTP Operations

4.1 SODA for REST HTTP Operation URIs 4-2
4.2 SODA for REST HTTP Operation Response Bodies 4-3
4.3 GET schema 4-4
4.3.1 URL Pattern for GET schema 4-4

4.3.2 Response Codes for GET schema 4-4
4.4 GET collection 4-5
4.4.1 URL Pattern for GET collection 4-5
4.4.2 Response Codes for GET collection 4-6
4.4.3 Links Array for GET collection 4-7
4.5 GET object 4-8
4.5.1 URL Pattern for GET object 4-8
4.5.2 Request Headers for GET object 4-8
4.5.3 Response Codes for GET object 4-8
4.6 PUT collection 4-9
4.6.1 URL Pattern for PUT collection 4-9
4.6.2 Request Body for PUT collection (Optional) 4-9
4.6.3 Response Codes for PUT collection 4-9
4.7 PUT object 4-9
4.7.1 URL Pattern for PUT object 4-10
4.7.2 Request Body for PUT object 4-10
4.7.3 Response Codes for PUT object 4-10
4.8 DELETE collection 4-10
4.8.1 URL Pattern for DELETE collection 4-10
4.8.2 Response Codes for DELETE collection 4-10
4.9 DELETE object 4-11
4.9.1 URL Pattern for DELETE object 4-11
4.9.2 Response Codes for DELETE object 4-11
4.10 POST object 4-11
4.10.1 URL Pattern for POST object 4-12
4.10.2 Request Body for POST object 4-12
4.10.3 Response Codes for POST object 4-12
4.11 POST query 4-12
4.11.1 URL Pattern for POST query 4-13
4.11.2 Request Body for POST query 4-13
4.11.3 Response Codes for POST query 4-13
4.12 POST array insert 4-13
4.12.1 URL Pattern for POST array insert 4-14
4.12.2 Request Body for POST array insert 4-14
4.12.3 Response Codes for POST array insert 4-14
4.13 POST bulk delete 4-15
4.13.1 URL Pattern for POST bulk delete 4-15
4.13.2 Request Body for POST bulk delete (Optional) 4-15
4.13.3 Response Codes for POST bulk delete 4-15

5 Collection Specifications

5.1 Key Assignment Method 5-4
5.2 Versioning Method 5-5

6 Security

6.1 Authentication Mechanisms 6-2
6.2 Security Considerations for Development and Testing 6-2

Index

List of Examples

4-1 Response Body 4-4
5-1 Collection Specification 5-3

List of Tables

4-1	SODA for REST HTTP Operations	4-1
4-2	Fields That Can Appear in Response Bodies	4-3
4-3	Additional Response Body Fields for Operations that Return Objects	4-3
4-4	Relationship of GET collection Parameters to Mode and Links Array	4-7
5-1	Collection Specification Fields	5-1
5-2	Key Assignment Methods	5-5
5-3	Versioning Methods	5-6

Preface

This document explains how to use the Oracle SODA for REST API.

Audience

This document is intended for SODA for REST users.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

None

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

SODA for REST Overview

SODA for REST uses the representational state transfer (REST) architectural style to implement **Simple Oracle Document Access** (SODA). You can use this API to perform create, read, update, and delete (CRUD) operations on documents of any kind, and you can use it to query JSON documents.

Your application can use the API operations to create and manipulate JSON objects that it uses to persist application objects and state. To generate the JSON documents, your application can use JSON serialization techniques. When your application retrieves a document object, a JSON parser converts it to an application object.

SODA is a set of NoSQL-style APIs that let you create and store collections of documents in Oracle Database, retrieve them, and query them, without needing to know Structured Query Language (SQL) or how the data in the documents is stored in the database.

Familiarity with the following can help you take best advantage of the information presented here:

- Oracle Database relational database management system (RDBMS)
- JavaScript Object Notation (JSON)
- Hypertext Transfer Protocol (HTTP)

Note:

Documents used with SODA for REST are limited to approximately 2 gigabytes.

Topics

- Overview of the Representational State Transfer (REST) Architectural Style (page 1-2)

See Also:

- SODA for REST HTTP Operations (page 4-1) for detailed information about the operations defined by SODA for REST
- *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for general information about SODA
- *Oracle Database SODA for Java Developer's Guide*, which explains how to use the Java client API on which SODA for REST is built
- *Oracle Database SODA for C Developer's Guide*
- *Oracle Database SODA for PL/SQL Developer's Guide*
- *Oracle Database JSON Developer's Guide* for information about using SQL with JSON data

1.1 Overview of the Representational State Transfer (REST) Architectural Style

The REST architectural style was used to define HTTP 1.1 and Uniform Resource Identifiers (URIs). A REST-based API strongly resembles the basic functionality provided by an HTTP server, and most REST-based systems are implemented using an HTTP client and an HTTP server.

A typical REST implementation maps create, read, update, and delete (CRUD) operations to HTTP verbs `POST`, `GET`, `PUT`, and `DELETE`, respectively.

A key characteristic of a REST-based system is that it is *stateless*: the server does not track or manage client object state. Each operation performed against a REST-based server is atomic; it is considered a transaction in its own right. In a typical REST-based system, many facilities that are taken for granted in an RDBMS environment, such as locking and concurrency control, are left to the application to manage.

A main advantage of a REST-based system is that its services can be used from almost any modern programming platform, including traditional programming languages (such as C, C#, C++, JAVA, and PL/SQL) and modern scripting languages (such as JavaScript, Perl, Python, and Ruby).

See Also:

Principled Design of the Modern Web Architecture, by Roy T. Fielding and Richard N. Taylor, at:

`http://www.ics.uci.edu/~taylor/documents/2002-REST-TOIT.pdf`

2

Installing SODA for REST

Complete instructions are provided for installing SODA for REST.

To install SODA for REST:

1. Ensure that Oracle Database 12c Release 1 (12.1.0.2) with Merge Label Request (MLR) bundle patch 20885778 is installed. (Patch 20885778 obsoletes patch 20080249.)

   Obtain the patch from My Oracle Support (`https://support.oracle.com`). Select tab **Patches & Updates**. Search for the patch number, 20885778 or access it directly at this URL: `https://support.oracle.com/rs?type=patch&id=20885778`.

2. Start the database.

3. Download Oracle REST Data Services (ORDS), and extract the zip file.

   For instructions, see *Oracle REST Data Services Installation, Configuration, and Development Guide*.

4. Configure ORDS.

   - If the database uses standard port 1521:

     ```
     java -jar ords.war install
     ```

   - If the database uses a nonstandard port (any port except 1521):

     ```
     java -jar ords.war install advanced
     ```

   Note:

   When prompted:

   - Do not skip the step of verifying/installing the Oracle REST Data Services schema.
   - Skip the steps that configure the PL/SQL Gateway.
   - Skip the steps that configure Application Express RESTful Services database users.
   - Decline to start the standalone server.

   For more information, see *Oracle REST Data Services Installation, Configuration, and Development Guide*.

5. Connect to the schema that you want ORDS to access.

6. Enable ORDS in the schema by executing this SQL command:

```
EXEC ords.enable_schema;
COMMIT;
```

7. Grant role `SODA_APP` to the database schema (user account) *`schema`* that you enabled in step 6 (page 2-2):

```
GRANT SODA_APP TO schema;
```

8. Only if you are in a development environment:

   a. Remove the default security constraints:

   ```
   BEGIN
     ords.delete_privilege_mapping(
       'oracle.soda.privilege.developer',
       '/soda/*');
     COMMIT;
   END;
   ```

   This enables *anonymous* access to the service and is *not* recommended for production systems. For more information about security, see Security (page 6-1).

   b. Start ORDS in standalone mode:

   ```
   java -jar ords.war standalone
   ```

   For more information, see *Oracle REST Data Services Installation, Configuration, and Development Guide*.

Note:

Disabling security and running ORDS in standalone mode is not recommended in production environments.

9. In a web browser, open:

```
http://localhost:8080/ords/schema/soda/latest/
```

Where *`schema`* is the lowercase name of the schema in which you enabled ORDS in step 6 (page 2-2). If the installation succeeded, you see:

```
{"items":[],"more":false}
```

See Also:

- Security (page 6-1)
- *Oracle REST Data Services Installation, Configuration, and Development Guide*

3

Using SODA for REST

A step-by-step walkthrough is provided for the basic SODA for REST operations, using examples that you can run. The examples use command-line tool cURL to send REST requests to the server.

The examples assume that you started ORDS as instructed in Installing SODA for REST (page 2-1), enabling ORDS in `schema`.

Some examples also use the sample JSON documents included in the zip file that you downloaded in installation step 3 (page 2-1). They are in the directory `ORDS_HOME/examples/soda/getting-started`.

Topics

- Creating a New Collection (page 3-2)
- Getting the List of Available Collections (page 3-3)
- Deleting a Collection (page 3-4)
- Inserting a Document into a Collection (page 3-4)
- Retrieving a Document from a Collection (page 3-5)
- Deleting a Document from a Collection (page 3-6)
- Bulk-Inserting Documents from a JSON Array (page 3-6)
- Listing the Documents in a Collection (page 3-7)
- Updating a Document in a Collection (page 3-9)
- Using a Filter Specification to Select Documents From a Collection (page 3-10)

See Also:

- `http://curl.haxx.se/` for information about command-line tool cURL
- *Oracle REST Data Services Installation, Configuration, and Development Guide*

Steps

1. Creating a New Collection (page 3-2)
2. Getting the List of Available Collections (page 3-3)
3. Deleting a Collection (page 3-4)

4. Inserting a Document into a Collection (page 3-4)
5. Retrieving a Document from a Collection (page 3-5)
6. Deleting a Document from a Collection (page 3-6)
7. Bulk-Inserting Documents from a JSON Array (page 3-6)
8. Listing the Documents in a Collection (page 3-7)
9. Updating a Document in a Collection (page 3-9)
10. Using a Filter Specification to Select Documents From a Collection (page 3-10)

3.1 Creating a New Collection

An example is given of creating a new collection.

To create a new collection, run this command, where *`MyCollection`* is the name of the collection:

```
curl -i -X PUT http://localhost:8080/ords/schema/soda/latest/MyCollection
```

The preceding command sends a `PUT` request with URL `http://localhost:8080/ords/`*`schema`*`/soda/latest/`*`MyCollection`*, to create a collection named *`MyCollection`*. The `-i` command-line option causes cURL to include the HTTP response headers in the output. If the operation succeeds then the output looks similar to this:

```
HTTP/1.1 201 Created
Cache-Control: private,must-revalidate,max-age=0
Location: http://localhost:8080/ords/schema/soda/latest/MyCollection/
Content-Length: 0
```

Response code 201 indicates that the operation succeeded. A `PUT` operation that results in the creation of a new collection—a `PUT collection` operation—returns no response body.

A successful `PUT collection` operation creates a database table to store the new collection. One way to see the details of this table is using SQL*Plus command `describe`:

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

The preceding table reflects the default collection configuration. To create a custom collection configuration, provide a collection specification as the body of the `PUT` operation. For information about collection specifications, see Collection Specifications (page 5-1).

Caution:

To drop a collection, proceed as described in Deleting a Collection (page 3-4). Do *not* use SQL to drop the database table that underlies a collection. Collections have persisted metadata, in addition to the documents that are stored in the collection table.

See Also:

- PUT collection (page 4-9) for more information about this operation
- Getting the List of Available Collections (page 3-3)

3.2 Getting the List of Available Collections

An example is given of listing available collections.

To obtain a list of the collections available in *`schema`*, run this command:

```
curl -X GET http://localhost:8080/ords/schema/soda/latest
```

That sends a `GET` request with the URL `http://localhost:8080/ords/`*`schema`*`/soda/latest` and returns this response body:

```
{
  "items" : [
    {
      "name":"MyCollection",
      "properties": {
          "schemaName":"SCHEMA",
          "tableName":"MyCollection",
          ...
      }
      "links" : [
        {
         "rel" : "canonical",
         "href" :"http://localhost:8080/ords/schema/soda/latest/MyCollection"
        }
      ]
    }
  ],
  "more" : false
}
```

The response body includes all available collections in *`schema`*, which in this case is only collection `MyCollection`.

A successful `GET collection` operation returns response code 200, and the response body is a JSON object that contains an array of available collections and includes the collection specification for each collection.

See Also:

- GET schema (page 4-4)
- Collection Specifications (page 5-1)
- Deleting a Collection (page 3-4)

3.3 Deleting a Collection

An example is given of deleting a collection.

To delete `MyCollection`, run this command:

```
curl -i -X DELETE http://localhost:8080/ords/schema/soda/latest/MyCollection
```

The preceding command sends a `DELETE` request with the URL `http://localhost:8080/ords/`*`schema`*`/soda/latest/MyCollection` and returns:

```
HTTP/1.1 200 OK
Cache-Control: private,must-revalidate,max-age=0
Content-Length: 0
```

Response code 200 indicates that the operation succeeded. A `DELETE` operation that results in the deletion of a collection—a `DELETE collection` operation—returns no response body.

To verify that the collection was deleted, get the list of available collections in *`schema`*:

```
curl -X GET http://localhost:8080/ords/schema/soda/latest
```

If `MyCollection` was deleted, the preceding command returns:

```
{
  "items" : [],
  "more" : false
}
```

Create `MyCollection` again, so that you can use it in the next step:

```
curl -X PUT http://localhost:8080/ords/schema/soda/latest/MyCollection
```

See Also:

- DELETE collection (page 4-10)
- Inserting a Document into a Collection (page 3-4)

3.4 Inserting a Document into a Collection

An example is given of inserting a document into a collection.

The example uses file `po.json`, which was included in the download. The file contains a JSON document that contains a purchase order. To load the JSON document into `MyCollection`, run this command:

```
curl -X POST --data-binary @po.json -H "Content-Type: application/json"
http://localhost:8080/ords/schema/soda/latest/MyCollection
```

The preceding command sends a `POST` request with the URL `http://localhost:8080/ords/`*`schema`*`/soda/latest/MyCollection`. It outputs something like this:

```
{
  "items" : [
    {
      "id" : "2FFD968C531C49B9A7EAC4398DFC02EE",
      "etag" : "C1354F27A5180FF7B828F01CBBC84022DCF5F7209DBF0E6DFFCC626E3B0400C3",
      "lastModified":"2014-09-22T21:25:19.564394Z",
      "created":"2014-09-22T21:25:19.564394Z"
    }
  ],
  "hasMore" : false,
  "count" : 1
}
```

A successful `POST object` operation returns response code 200. The response body is a JSON document that contains the identifier that the server assigned to the document when you inserted it into the collection, as well as the current ETag and last-modified time stamp for the inserted document.

Note:

If you intend to retrieve the document then copy the document identifier (the value of field `"id"`), to use for retrieval.

See Also:

- POST object (page 4-11) for more information about this operation
- Retrieving a Document from a Collection (page 3-5)

3.5 Retrieving a Document from a Collection

An example is given of retrieving a document from a collection.

To retrieve the document that was inserted in Inserting a Document into a Collection (page 3-4), run this command, where *`id`* is the document identifier that you copied when inserting the document:

```
curl -X GET http://localhost:8080/ords/schema/soda/latest/MyCollection/id
```

A successful `GET document` operation returns response code 200. The response body contains the retrieved document.

If *`id`* does not exist in `MyCollection` then the response code is 404, as you can see by changing *`id`* to such an identifier:

```
curl -X GET http://localhost:8080/ords/schema/soda/latest/MyCollection/
2FFD968C531C49B9A7EAC4398DFC02EF

{
  "type" : "http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.1",
  "status" : 404,
  "title" : "Key 2FFD968C531C49B9A7EAC4398DFC02EF not found in collection MyCollection.",
  "o:errorCode" : "REST-02001"
}
```

See Also:

- GET object (page 4-8)
- Deleting a Document from a Collection (page 3-6)

3.6 Deleting a Document from a Collection

An example is given of deleting a document from a collection.

To delete, from `MyCollection`, the document that you retrieved in Retrieving a Document from a Collection (page 3-5), run this command (where *`id`* is the document identifier):

```
curl -i -X DELETE http://localhost:8080/ords/schema/soda/latest/MyCollection/id
```

The preceding command sends a `DELETE` request with URL `http://localhost:8080/ords/`*`schema`*`/soda/latest/MyCollection/`*`id`*, and it returns this:

```
HTTP/1.1 200 OK
Cache-Control: private,must-revalidate,max-age=0
Content-Length: 0
```

Response code 200 indicates that the operation succeeded. A `DELETE` operation that results in the deletion of an object from a collection—a `DELETE object` operation—returns no response body.

See Also:

- DELETE object (page 4-11)
- Bulk-Inserting Documents from a JSON Array (page 3-6)

3.7 Bulk-Inserting Documents from a JSON Array

An example is given of bulk-inserting a set of documents into a collection from a JSON array of documents. The bulk insert operation is also called `POST` array insert.

This example uses file `POList.json`, which was included in the download. The file contains a JSON array of 70 purchase orders. To load the purchase orders into collection `MyCollection`, run this command:

```
curl -X POST --data-binary @POList.json -H "Content-Type: application/json"
http://localhost:8080/ords/schema/soda/latest/MyCollection?action=insert
```

Parameter value `action=insert` causes the array to be inserted as a set of documents, rather than as a single document.

The preceding command sends a `POST` request with the URL `http://localhost:8080/ords/`*`schema`*`/soda/latest/MyCollection`, and it outputs something like this:

```
{
  "items" : [
  {
    "id" : "6DEAF8F011FD43249E5F60A93B850AB9",
    "etag" : "49205D7E916EAED914465FCFF029B2795885A1914966E0AE82D4CCDBBE2EAF8E",
    "lastModified" : "2014-09-22T22:39:15.546435Z",
    "created" : "2014-09-22T22:39:15.546435Z"
  },
  {
    "id" : "C9FF7685D48E4E4B8641D8401ED0FB68",
    "etag" : "F3EB514BEDE6A6CC337ADA0F5BE6DEFC5D451E68CE645729224BB6707FBE1F4F",
    "lastModified" : "2014-09-22T22:39:15.546435Z",
    "created":"2014-09-22T22:39:15.546435Z"
  },
  ...
  ],
  "hasMore":false,
  "count":70
}
```

A successful `POST` array insert operation returns response code 200. The response body is a JSON document that contains the identifier, ETag, and last-modified time stamp for each inserted document.

Copy an `"id"` field value returned by your own `POST` array insert operation (not a value from the preceding example). Query the collection using SQL*Plus or SQL Developer, substituting your copied value for *`identifier`*:

```
SELECT json_value(json_document FORMAT JSON, '$.Reference')
  FROM "MyCollection" WHERE id = 'identifier';

JSON_VALUE(JSON_DOCUMENTFORMATJSON,'$.REFERENCE')
--------------------------------------------------------------------------------
MSULLIVA-20141102
```

Note:

In the SQL `SELECT` statement, you must specify the table name `MyCollection` as a quoted identifier, because it is mixed-case (the table name is the same as the collection name).

Because `MyCollection` has the default configuration, which stores the JSON document in a `BLOB` column, you must include `FORMAT JSON` when using the SQL/JSON function `json_value`. You cannot use the simplified JSON syntax.

See Also:

- POST array insert (page 4-13)
- Listing the Documents in a Collection (page 3-7)

3.8 Listing the Documents in a Collection

An example is given of listing the documents in a collection, using a `GET` operation.

You can use parameters to control the result. For example, you can:

- Limit the number of documents returned
- Return only document identifiers (keys), only document contents, or both keys and contents

- Return a range of documents, based on keys or last-modified time stamps
- Specify the order of the list of returned documents

To list the documents in `MyCollection`, returning their keys and other metadata but not their content, run the following command.

```
curl -X GET http://localhost:8080/ords/schema/soda/latest/MyCollection?fields=id
```

The preceding command outputs something like this:

```
{
  "items" : [
  {
    "id" : "023C4A6581D84B71A5C0D5D364CE8484",
    "etag":"3484DFB604DDA3FBC0C681C37972E7DD8C5F4457ACE32BD16960D4388C5A7C0E",
    "lastModified" : "2014-09-22T22:39:15.546435Z",
    "created":"2014-09-22T22:39:15.546435Z"
  },
  {
    "id" : "06DD0319148E40A7B8AA48E39E739184",
    "etag" : "A19A1E9A3A38B1BAE3EE52B93350FBD76309CBFC4072A2BECD95BCA44D4849DD",
    "lastModified" : "2014-09-22T22:39:15.546435Z",
    "created" : "2014-09-22T22:39:15.546435Z"
  },
  ...
  ],
  "hasMore" : false,
  "count" : 70,
  "offset":0,
  "limit":100,
  "totalResults":70
}
```

A successful `GET collection` operation returns response code 200, and the response body is a JSON document that lists the documents in the collection. If the collection is empty, the response body is an empty `items` array.

To list at most 10 documents in `MyCollection`, returning their keys, content, and other metadata, run this command:

```
curl -X GET "http://localhost:8080/ords/schema/soda/latest/MyCollection?fields=all&limit=10"
```

The preceding command outputs something like this:

```
{
  "items": [ ... ],
  "hasMore" : true,
  "count" : 10,
  "offset" : 0,
  "limit" : 10,
  "links" : [{
      "rel" : "next",
      "href" :
"http://localhost:8080/ords/schema/soda/latest/MyCollection?offset=10&limit=10"
   }]
}
```

Note:

Including document content makes the response body much larger. Oracle recommends including the content in the response body only if you will need the content later. Retrieving the content from the response body is more efficient that retrieving it from the server.

The metadata in the response body shows that 10 documents were requested (`"limit" : 10`)) and 10 documents were returned (`"count" : 10)`), and that more documents are available (`"hasMore" : true`). To fetch the next set of documents, you can use the URL in the field `"links"."href"`.

The maximum number of documents returned from a collection by the server is controlled by the following:

- URL parameter `limit`
- Configuration parameters `soda.maxLimit` and `soda.defaultLimit`

Note:

If you intend to update the document then copy the document identifier (value of field `"id"`), to use for updating.

See Also:

- GET collection (page 4-5)
- Updating a Document in a Collection (page 3-9)
- Using a Filter Specification to Select Documents From a Collection (page 3-10), which lets you list documents based on content
- *Oracle REST Data Services Installation, Configuration, and Development Guide* for information about configuration parameters `soda.maxLimit` and `soda.defaultLimit`

3.9 Updating a Document in a Collection

An example is given of updating a document in a collection, that is, replacing it with a newer version. For this, you use a `PUT` operation.

The behavior of the `PUT` operation for a nonexistent document depends on the key-assignment method used by the collection.

- If the collection uses *server-assigned keys* (as does collection `MyCollection`) then an error is raised if you try to update a nonexistent document (that is, you specify a key that does not belong to any document in the collection).
- If the collection uses *client-assigned keys*, then trying to update a nonexistent document *inserts* into the collection a new document with the specified key.

Retrieve a document from `MyCollection` by running this command, where *`id`* is the document identifier that you copied in Listing the Documents in a Collection (page 3-7):

```
curl -X GET http://localhost:8080/ords/schema/soda/latest/MyCollection/id
```

The preceding command outputs the retrieved document.

To update this document with the content of file `poUpdated.json`, which was included in the download, execute this command:

```
curl -i -X PUT --data-binary @poUpdated.json -H "Content-Type: application/json"
http://localhost:8080/ords/schema/soda/latest/MyCollection/id
```

The preceding command outputs something like this:

```
HTTP/1.1 200 OK
Cache-Control: no-cache,must-revalidate,no-store,max-age=0
ETag: A0B07E0A6D000358C546DC5D8D5059D9CB548A1A5F6F2CAD66E2180B579CCB6D
Last-Modified: Mon, 22 Sep 2014 16:42:35 PDT
Location: http://localhost:8080/ords/schema/soda/latest/MyCollection/
023C4A6581D84B71A5C0D5D364CE8484/
Content-Length: 0
```

The response code 200 indicates that the operation succeeded. A `PUT` operation that results in the successful update of a document in a collection—a `PUT object` operation—returns no response body.

To verify the document has been updated, rerun this command:

```
curl -X GET http://localhost:8080/ords/schema/soda/latest/MyCollection/id
```

The preceding command returns:

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

See Also:

- PUT object (page 4-9)
- Key Assignment Method (page 5-4)
- Using a Filter Specification to Select Documents From a Collection (page 3-10)

3.10 Using a Filter Specification to Select Documents From a Collection

Examples are given of using a filter specification, or query-by-example (QBE), to define query criteria for selecting documents from a collection.

The examples use the `QBE.*.json` files that are included in the zip file that you downloaded in installation step 3 (page 2-1). They are in directory `ORDS_HOME/examples/soda/getting-started`.

See Also:

- POST query (page 4-12)
- *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for information about filter specifications and QBE

3.10.1 QBE.1.json

The query-by-example (QBE) in file `QBE.1.json` returns a list of nine documents, each of which has `"TGATES"` as the value of field `User`.

This is the query in file `QBE.1.json`:

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

To execute the query, run this command:

```
curl -X POST --data-binary @QBE.1.json -H "Content-Type: application/json"
http://localhost:8080/ords/schema/soda/latest/MyCollection?action=query
```

A successful `POST query` operation returns response code 200 and a list of documents that satisfy the query criteria.

Because the command has no `fields` parameter, the default value `fields=all` applies, and the response body contains both the metadata and the content of each document.

Note:

Including document content makes the response body much larger. Oracle recommends including the content in the response body only if you need the content for a subsequent operation. Retrieving the content from the response body is more efficient that retrieving it from the server.

To execute the queries in the other `QBE.*.json` files, run commands similar to the preceding one.

3.10.2 QBE.2.json

The query-by-example (QBE) in file `QBE.2.json` selects documents where the value of field `UPCCode` equals `"13023015692"`. `UPCCode` is a field of object `Part`, which is a field of array `LineItems`. Because no array offset is specified for `LineItems`, the query searches all elements of the array.

This is the query in file `QBE.2.json`:

```
{ "LineItems.Part.UPCCode" : "13023015692" }
```

Note:

Keyword `"$eq"` in the query is implied. See *Oracle Database SODA for Java Developer's Guide* for more information.

3.10.3 QBE.3.json

The query-by-example (QBE) in file `QBE.3.json` selects documents where the value of field `ItemNumber`, in object `LineItems`, is greater than 4. Keyword `"$gt"` is required.

This is the query in file `QBE.3.json`:

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

3.10.4 QBE.4.json

The query-by-example (QBE) in file `QBE.4.json` selects documents where the value of field `UPCCode` equals `"13023015692"` *and* the value of field `ItemNumber` equals `3`. Keyword `$and` is optional.

This is the query in file `QBE.4.json`:

```
{ "$and" : [
     { "LineItems.Part.UPCCode" : "13023015692" },
     { "LineItems.ItemNumber" : 3 }
  ]
}
```

See Also:

Oracle Database Introduction to Simple Oracle Document Access (SODA)

4

SODA for REST HTTP Operations

The SODA for REST HTTP operations are described.

Table 4-1 (page 4-1) summarizes the HTTP operations that SODA for REST provides. For complete descriptions of the operations, click the links in the left column.

Table 4-1 SODA for REST HTTP Operations

Operation	Description
GET schema (page 4-4)	Gets some or all collection names in a schema.
GET collection (page 4-5)	Gets all or a subset of objects from a collection, using parameters to specify the subset. You can page through the return set.
GET object (page 4-8)	Gets a specified object from a collection.
PUT collection (page 4-9)	Creates a collection if it does not exist.
PUT object (page 4-9)	Replaces a specified object with an uploaded object (typically a new version). If the collection has client-assigned keys and the uploaded object is not already in the collection, then `PUT` inserts the uploaded object into the collection.
DELETE collection (page 4-10)	Deletes a collection.
DELETE object (page 4-11)	Deletes a specified object from a collection.
POST object (page 4-11)	Puts an uploaded object in a specified collection, assigning and returning its key. Collection must use server-assigned keys.
POST query (page 4-12)	Gets all or a subset of objects from a collection, using a filter to specify the subset. You cannot page through the return set.
POST array insert (page 4-13)	Inserts an array of objects into a specified collection, assigning and returning their keys.
POST bulk delete (page 4-15)	Deletes all or a subset of objects from a collection.

Topics

- SODA for REST HTTP Operation URIs (page 4-2)

- SODA for REST HTTP Operation Response Bodies (page 4-3)
- GET schema (page 4-4)
- GET collection (page 4-5)
- GET object (page 4-8)
- PUT collection (page 4-9)
- PUT object (page 4-9)
- DELETE collection (page 4-10)
- DELETE object (page 4-11)
- POST object (page 4-11)
- POST query (page 4-12)
- POST array insert (page 4-13)
- POST bulk delete (page 4-15)

4.1 SODA for REST HTTP Operation URIs

A SODA for REST HTTP operation is specified by a Universal Resource Identifier (URI).

The URI has this form:

```
/ords/schema/soda/[version/[collection/[{key/|?action=action}]]]
```

where:

- `ords` is the directory of the Oracle REST Data Services (ORDS) listener, of which SODA for REST is a component.
- *`schema`* is the name of an Oracle Database schema that has been configured as an end point for SODA for REST.
- `soda` is the name given to the Oracle Database JSON service when mapped as a template within ORDS.
- *`version`* is the version number of `soda`.
- *`collection`* is the name of a set of objects stored in *`schema`*.

  Typically, an object is a JSON document, but it can be a Multipurpose Internet Mail Extensions (MIME) type (for example, image, audio, or video).

  A JSON document is represented as textual JSON.

  Typically, an application uses a collection to hold all instances of a particular type of object. Thus, a collection is roughly analogous to a table in a relational database. One column stores keys and another column stores content.
- *`key`* is a string that uniquely identifies an object in *`collection`*.

  A **specified object** is specified by its key.
- *`action`* is either `query`, `index`, `unindex`, `insert`, `update`, or `delete`.

4.2 SODA for REST HTTP Operation Response Bodies

If a SODA for REST HTTP operation returns information or objects, it does so in a response body.

For the operation GET object (page 4-8), the response body is a single object.

Table 4-2 (page 4-3) lists and describes fields that can appear in response bodies.

Table 4-2 Fields That Can Appear in Response Bodies

Field	Description
`key`	String that uniquely identifies an object (typically a JSON document) in a collection.
`etag`	HTTP entity tag (ETag)—checksum or version.
`created`	Created-on time stamp.
`lastModified`	Last-modified time stamp.
`value`	Object contents (applies only to JSON object).
`mediaType`	HTTP Content-Type (applies only to non-JSON object).
`bytes`	HTTP Content-Length (applies only to non-JSON object).
`items`	List of one or more collections or objects that the operation found or created. This field can be followed by the fields in Table 4-3 (page 4-3).

If an operation creates or returns objects, then its response body can have the additional fields in Table 4-3 (page 4-3). The additional fields appear after field `items`.

Table 4-3 Additional Response Body Fields for Operations that Return Objects

Field	Description
`name`	Name of collection. This field appears only in the response body of GET schema (page 4-4).
`properties`	Properties of collection. This field appears only in the response body of GET schema (page 4-4).
`hasMore`	`true` if `limit` was reached before available objects were exhausted, `false` otherwise. This field is always present.
`limit`	Server-imposed maximum collection (row) limit.
`offset`	Offset of first object returned (if known).
`count`	Number of objects returned. This is the only field that can appear in the response body of POST bulk delete (page 4-15).
`totalResults`	Number of objects in collection (if requested)
`links`	Possible final field for `GET collection` operation. For details, see GET collection (page 4-5).

Example 4-1 (page 4-4) shows the structure of a response body that returns 25 objects. The first object is a JSON object and the second is a `jpeg` image. The collection that contains these objects contains additional objects.

Example 4-1 Response Body

```
{

  "items" : [
    {
      "id"           : "key_of_object_1",
      "etag"         : "etag_of_object_1",
      "lastModified" : "lastmodified_timestamp_of_object_1",
      "value"        : {object_1}
    },
    {
      "id"           : "key_of_object_2",
      "etag"         : "etag_of_object_2",
      "lastModified" : "lastmodified_timestamp_of_object_2",
      "mediaType"    : "image/jpeg",
      "bytes"        : 1234
    },
    ...
  ],
  "hasMore" : true,
  "limit"   : 100,
  "offset"  : 50,
  "count"   : 25
  "links"   : [ ... ]
}
```

4.3 GET schema

`GET` schema gets all or a subset of collection names in a schema.

See Also:

Listing the Documents in a Collection (page 3-7)

4.3.1 URL Pattern for GET schema

The URL pattern for `GET` schema is described.

`/ords/`***`schema`***`/soda/`*`version`*`/`

Without parameters, `GET schema` gets all collection names in *`schema`*.

Parameter	Description
`limit=`*`n`*	Limits number of listed collection names to *`n`*.
`fromID=`*`collection`*	Starts getting with *`collection`* (inclusive).

4.3.2 Response Codes for GET schema

The response codes for `GET` schema are described.

200

Success—response body contains names and properties of collections in schema, ordered by name. For example:

```
{
  "items" : [
    {"name"       : "employees",
     "properties" : {...} },
    {"name"       : "departments",
     "properties" : {...} },
    ...
  ],
  "hasMore" : false
}
```

If `hasMore` is `true`, then to get the next batch of collection names specify `fromID=`*`last_returned_collection`*. (In the preceding example, *`last_returned_collection`* is `"regions"`).

404

Either the schema was not found or access is not authorized.

4.4 GET collection

`GET` collection gets all or a subset of objects from a collection, using parameters to specify the subset. You can page through the set of returned objects.

See Also:

- POST query (page 4-12), which gets all or a subset of objects from a collection, using a filter instead of parameters. You *cannot* page through the set of returned objects.
- Listing the Documents in a Collection (page 3-7)

4.4.1 URL Pattern for GET collection

The URL pattern for `GET` collection is described.

`/ords/`*`schema`*`/soda/`*`version`*`/`***`collection`***`/`

Without parameters, `GET collection` gets all objects (both key and content) from *`collection`* and does not return the number of objects in *`collection`*.

Note:

For non-JSON objects in the collection, `GET collection` returns, instead of content, media type and (if known) size in bytes.

Parameter	Description
`limit=`*`n`*	Limits number of objects to *n*.
`offset=`*`n`*	Skips *n* objects before getting first object.

Parameter	Description
`fields={id|value|all}`	Gets only object `id` fields (keys), only object `value` fields (content), or `all` fields (both key and content). Regardless of the `fields` value, `GET collection` returns the other metadata that the collection stores for each document.
`totalResults=true`	Returns number of objects in collection. **Note:** Inefficient
`fromID=`*`key`*	Starts getting objects after *`key`*, in ascending order.
`toID=`*`key`*	Stops getting objects before *`key`*, in descending order.
`after=`*`key`*	Starts getting objects after *`key`*, in ascending order.
`before=`*`key`*	Stops getting objects before *`key`*, in descending order.
`since=`*`timestamp`*	Gets only objects with time stamp later than *`timestamp`*.
`until=`*`timestamp`*	Gets only objects with time stamp earlier than *`timestamp`*.

4.4.2 Response Codes for GET collection

The response codes for `GET` collection are described.

200

Success—response body contains the specified objects from *`collection`* (or only their keys, if you specified `fields=id`). For example:

```
{
  "items" : [
    {
      "id"           : "key_of_object_1",
      "etag"         : "etag_of_object_1",
      "lastModified" : "lastmodified_timestamp_of_object_1",
      "value"        : {object_1}
    },
    {
      "id"           : "key_of_object_2",
      "etag"         : "etag_of_object_2",
      "lastModified" : "lastmodified_timestamp_of_object_2",
      "value"        : {object_2}
    },
    {
      "id"           : "key_of_object_3",
      "etag"         : "etag_of_object_3",
      "lastModified" : "lastmodified_timestamp_of_object_3",
      "mediaType"    : "image/jpeg",
      "bytes"        : 1234
    },
    ...
  ],
  "hasMore" : true,
  "limit"   : 100,
  "offset"  : 50,
  "count"   : 25
  "links"   : [ ... ]
}
```

If `hasMore` is `true`, then to get the next batch of objects repeat the operation with an appropriate parameter. For example:

- `offset=`*`n`* if the response body includes the offset
- `toID=`*`last_returned_key`* or `before=`*`last_returned_key`* if the response body includes `descending=true`
- `fromID=`*`last_returned_key`* or `after=`*`last_returned_key`* if the response body does not include `descending=true`

For information about `links`, see Links Array for GET collection (page 4-7).

401

Read access to collection is not authorized.

404

Collection was not found.

4.4.3 Links Array for GET collection

The `links` array for `GET` collection is described.

The existence and content of the `links` array depends on the **mode** of the `GET collection` operation, which is determined by its parameters.

When the `links` array exists, it has an element for each returned object. Each element contains links from that object to other objects. The possible links are:

- `first`, which links the object to the first object in the collection
- `prev`, which links the object to the previous object in the collection
- `next`, which links the object to the next object in the collection

Using `prev` and `next` links, you can page through the set of returned objects.

Table 4-4 (page 4-7) shows how `GET collection` parameters determine mode and the existence and content of the `links` array.

Table 4-4 Relationship of GET collection Parameters to Mode and Links Array

Parameter	Mode	Links Array
`fields=id`	Keys-only	Does not exist (regardless of other parameters).
`offset=`*`n`*	Offset	Has an element for each returned object. Each element has these links, except as noted: • `first` (except for first object) • `prev` (except for first object) • `next` (except for last object)

Table 4-4 (Cont.) Relationship of GET collection Parameters to Mode and Links Array

Parameter	Mode	Links Array

`fromID=`*`key`*
`toID=`*`key`*
`after=`*`key`*
`before=`*`key`*
	Keyed	Has an element for each returned object. Each element has these links, except as noted:
		• `prev` (except for first object)
		• `next` (except for last object)

`since=`*`timestamp`*
`until=`*`timestamp`*
	Timestamp	Does not exist.

4.5 GET object

`GET` object gets a specified object from a specified collection.

See Also:

Retrieving a Document from a Collection (page 3-5)

4.5.1 URL Pattern for GET object

The URL pattern for `GET` object is described.

`/ords/`*`schema`*`/soda/`*`version`*`/`***`collection`***`/`***`key`***`/`

No parameters.

4.5.2 Request Headers for GET object

The request headers for `GET` object are described.

Operation `GET object` accepts these optional request headers:

Header	Description
`If-Modified-Since=`*`timestamp`*	Returns response code 304 if object has not changed since *`timestamp`*.
`If-None-Match`	Returns response code 304 if object `etag` matches the current checksum or version.

4.5.3 Response Codes for GET object

The response codes for `GET` object are described.

200

Success—response body contains object identified by the URL pattern.

204

Object content is null.

304

Either object was not modified since modification date or checksum matches object `etag` (see Request Headers for GET object (page 4-8)).

401

Read access to collection or object is not authorized.

404

Collection or object was not found.

4.6 PUT collection

PUT collection creates a collection if it does not exist.

See Also:

Creating a New Collection (page 3-2)

4.6.1 URL Pattern for PUT collection

The URL pattern for PUT collection is described.

`/ords/schema/soda/version/collection/`

No parameters.

4.6.2 Request Body for PUT collection (Optional)

The request body for PUT collection is described.

See Collection Specifications (page 5-1).

4.6.3 Response Codes for PUT collection

The response codes for PUT collection are described.

200

Collection with the same name and properties already exists.

201

Success—collection was created.

401

Collection creation is not authorized.

4.7 PUT object

PUT object replaces a specified object in a specified collection with an uploaded object (typically a new version). If the collection has client-assigned keys and the uploaded object is not already in the collection, then PUT inserts the uploaded object into the collection.

See Also:

Updating a Document in a Collection (page 3-9)

4.7.1 URL Pattern for PUT object

The URL pattern for `PUT` object is described.

`/ords/`*`schema`*`/soda/`*`version`*`/`***`collection`***`/`***`key`***`/`

No parameters.

4.7.2 Request Body for PUT object

The request body for `PUT` object is the uploaded object.

4.7.3 Response Codes for PUT object

The response codes for `PUT` object are described.

200

Success—object was replaced.

401

Updating collection is not authorized.

405

Collection is read-only.

4.8 DELETE collection

`DELETE` collection deletes a collection.

To delete all *objects* from a collection, but *not delete the collection* itself, use POST bulk delete (page 4-15).

See Also:

Deleting a Collection (page 3-4)

4.8.1 URL Pattern for DELETE collection

The URL pattern for `DELETE` collection is described.

`/ords/`*`schema`*`/soda/`*`version`*`/`***`collection`***`/`

No parameters.

4.8.2 Response Codes for DELETE collection

The response codes for `DELETE` collection are described.

200

Success—collection was deleted.

401

Deleting collection is not authorized.

404

Collection was not found.

4.9 DELETE object

`DELETE` object deletes a specified object from a specified collection.

See Also:

Deleting a Document from a Collection (page 3-6)

4.9.1 URL Pattern for DELETE object

The URL pattern for `DELETE` object is described.

/ords/*schema*/soda/*version*/***collection***/***key***/

No parameters.

4.9.2 Response Codes for DELETE object

The response codes for `DELETE` object are described.

200

Success—object was deleted.

401

Either deleting from collection or deleting this object is not authorized.

404

Object was not found.

405

Collection is read-only.

4.10 POST object

`POST` object inserts an uploaded object into a specified collection, assigning and returning its key. The collection must use server-assigned keys.

If the collection uses client-assigned keys, use PUT object (page 4-9). For information about key assignment methods, see Key Assignment Method (page 5-4).

See Also:

Inserting a Document into a Collection (page 3-4)

4.10.1 URL Pattern for POST object

The URL pattern for `POST` object is described.

`/ords/schema/soda/version/collection/`

No parameters.

4.10.2 Request Body for POST object

The request body for `POST` object is the uploaded object to be inserted in the collection.

4.10.3 Response Codes for POST object

The response codes for `POST` object are described.

201

Success—object is in collection; response body contains server-assigned key and possibly other information. For example:

```
{
  "items" : [
    {
      "id"           : "key",
      "etag"         : "etag",
      "lastModified" : "timestamp"
      "created"      : "timestamp"
    }
  ],
  "hasMore" : false
}
```

202

Object was accepted and queued for asynchronous insertion; response body contains server-assigned key.

401

Inserting into collection is not authorized.

405

Collection is read-only.

501

Unsupported operation (for example, no server-side key assignment).

4.11 POST query

`POST` query gets all or a subset of objects from a collection, using a filter to specify the subset. You cannot page through the set of returned objects.

See Also:

- GET collection (page 4-5), which gets all objects, or a subset of these, from a collection, using parameters instead of a filter. You can page through the set of returned objects.
- Using a Filter Specification to Select Documents From a Collection (page 3-10)

4.11.1 URL Pattern for POST query

The URL pattern for `POST` query is described.

`/ords/`*`schema`*`/soda/`*`version`*`/`***`collection`***`?action=`**`query`**

Parameters are optional except as noted.

Parameter	Description
`action=query`	**Required.** Specifies kind of action.
`limit=`*`n`*	Limit number of returned objects to *`n`*.
`offset=`*`n`*	Skip *`n`* objects before returning objects.
`fields={id|value|all}`	Return object `id` (key) only, object `value` (content) only, or `all` (object key and content). **Default:** `all`

4.11.2 Request Body for POST query

If you omit the filter specification object from the request body of `POST` query then the operation gets all objects in the collection.

See Also:

Oracle Database Introduction to Simple Oracle Document Access (SODA) for information about SODA filter specifications.

4.11.3 Response Codes for POST query

The response codes for `POST` query are described.

200

Success—object is in collection; response body contains all objects in collection that match filter.

404

Either collection was not found or read access to collection is not authorized.

4.12 POST array insert

`POST` array insert inserts an array of objects into a specified collection, assigning and returning their keys.

See Also:

Bulk-Inserting Documents from a JSON Array (page 3-6)

4.12.1 URL Pattern for POST array insert

The URL pattern for `POST` array insert is described.

`/ords/schema/soda/version/collection?action=insert`

Parameter	Description
`action=insert`	**Required.** Specifies kind of action.

4.12.2 Request Body for POST array insert

The request body for `POST` array insert is an array of objects.

Array of objects.

4.12.3 Response Codes for POST array insert

The response codes for `POST` array insert are described.

200

Success—response body contains an array with the assigned keys for inserted objects. For example:

```
{
  "items" : [
    {
      "id"           : "12345678",
      "etag"         : "...",
      "lastModified" : "..."
      "created"      : "..."
    },
    {
      "id"           : "23456789",
      "etag"         : "...",
      "lastModified" : "..."
      "created"      : "..."
    }
  ],
  "hasMore" : false
}
```

401

Inserting into collection is not authorized.

404

Collection was not found.

405

Collection is read-only.

4.13 POST bulk delete

`POST` bulk delete deletes all or a subset of objects from a specified collection, using a filter to specify the subset.

Note:

If you delete all objects from the collection, the empty collection continues to exist. To delete the collection itself, use DELETE collection (page 4-10).

4.13.1 URL Pattern for POST bulk delete

The URL pattern for `POST` bulk delete is described.

Either of the following:

```
/ords/schema/soda/version/collection?action=delete
```

```
/ords/schema/soda/version/collection?action=truncate
```

Parameter	Description
`action=delete`	**Required.** Specifies the deletion of all or a subset of objects from *`collection`*, using an optional filter to specify the subset. See the following warning.
`action=truncate`	**Required.** Specifies the deletion of all objects from *`collection`*. Does not use a filter.

WARNING:

If you specify `action=delete` and omit the filter specification, or if the filter specification is empty, then the operation deletes all objects from the collection.

4.13.2 Request Body for POST bulk delete (Optional)

See *Oracle Database SODA for Java Developer's Guide* for information about SODA filter specifications.

4.13.3 Response Codes for POST bulk delete

The response codes for `POST` bulk delete are described.

200

Success—response body contains number of deleted collections. For example:

```
{
  "count" : 42
}
```

401

Deleting from collection is not authorized.

405

Collection is read-only.

5 Collection Specifications

A collection specification provides information about the Oracle Database table or view underlying the collection object. The table or view is created when you create the collection.

Note:

In collection specifications, you must use *strict* JSON syntax. That is, you must enclose each nonnumeric value in double quotation marks.

Table 5-1 (page 5-1) describes the collection specification fields and their possible values.

Note:

If you omit one of the optional columns (created-on timestamp, last-modified timestamp, version, or media type) from the collection specification then no such column is created. At a minimum, a collection has a key column and a content column.

Table 5-1 Collection Specification Fields

Field	Description	Possible Values
`schemaName`	SQL name of schema that owns table or view underlying collection object.	—
`tableName` or `viewName`	SQL name of table or view underlying collection object.	—
`keyColumn.name`	Name of key column.	Default: `ID`
`keyColumn.sqlType`	SQL data type of key column.	`VARCHAR2` (default), `NUMBER`, `RAW`
`keyColumn.maxLength`	Maximum length of key column, if not of `NUMBER` data type.	Default: 255
`keyColumn.assignmentMethod`	Key assignment method.	`SEQUENCE`, `GUID`, `UUID` (default), or `CLIENT`
`keyColumn.sequenceName`	If `keyColumn.assignmentMethod` is `SEQUENCE`, then this field must specify the name of a database sequence.	Name of existing database sequence
`contentColumn.name`	Name of content column.	Default: `JSON_DOCUMENT`

Table 5-1 (Cont.) Collection Specification Fields

Field	Description	Possible Values
`contentColumn.sqlType`	SQL data type of content column.	`VARCHAR2`, `BLOB` (default), `CLOB`
`contentColumn.maxLength`	Maximum length of content column, if not of LOB data type.	The default length is 4000 bytes. If `MAX_STRING_SIZE` = `STANDARD` then `maxLength` can be at most 4000 (bytes). If `MAX_STRING_SIZE` = `EXTENDED`, then `maxLength` can be at most 32767 (bytes).
`contentColumn.validation`	Validation level of content column. Corresponds to SQL condition `is json`, which determines the syntax to which JSON content must conform. `STANDARD` validates according to the JSON RFC 4627 standard. (It corresponds to the strict syntax defined for Oracle SQL condition `is json`.) `STRICT` is the same as `STANDARD`, except that it also verifies that the document does not contain duplicate JSON field names. (It corresponds to the strict syntax defined for Oracle SQL condition `is json` when the keywords `WITH UNIQUE KEYS` are also used.) `LAX` validates more loosely. (It corresponds to the lax syntax defined for Oracle SQL condition `is json`.) Some of the relaxations that `LAX` allows include the following: • It does not require JSON field names to be enclosed in double quotation marks (`"`). • It allows uppercase, lowercase, and mixed case versions of `true`, `false`, and `null`. • Numerals can be represented in additional ways.	`STANDARD` (default), `STRICT`, `LAX`
`contentColumn.compress`	Compression level for SecureFiles stored in content column.	`NONE` (default), `HIGH`, `MEDIUM`, `LOW`
`contentColumn.cache`	Caching of SecureFiles stored in content column.	`TRUE`, `FALSE` (default)
`contentColumn.encrypt`	Encryption algorithm for SecureFiles stored in content column.[1]	`NONE` (default), `3DES168`, `AES128`, `AES192`, `AES256`
`creationTimeColumn.name`	Name of optional created-on timestamp column. This column has SQL data type `TIMESTAMP` and default value `SYSTIMESTAMP`.	Default: `CREATED_ON`

Table 5-1 (Cont.) Collection Specification Fields

Field	Description	Possible Values
`lastModifiedColumn.name`	Name of optional last-modified timestamp column. This column has SQL data type `TIMESTAMP` and default value `SYSTIMESTAMP`.	Default: `LAST_MODIFIED`
`lastModifiedColumn.index`	Name of nonunique index on timestamp column. The index is created if a name is specified.	
`versionColumn.name`	Name of optional version (ETag) column. This column has SQL data type `VARCHAR2(255)` unless the method is `SEQUENTIAL` or `TIMESTAMP`, in which case it has data type `NUMBER`. **Note:** If the method is `TIMESTAMP` then the version is stored as an integer representation of the date and time with microsecond precision. It does not store a date/time string or a SQL date/time type.	Default: `VERSION`
`versionColumn.method`	Versioning method.	`SEQUENTIAL`, `TIMESTAMP`, `UUID`, `SHA256` (default), `MD5`, `NONE`
`mediaTypeColumn.name`	Name of optional object media type column. This column has SQL data type `VARCHAR2(255)`.	
`readOnly`	Read/write policy: `TRUE` means read-only.	`TRUE`, `FALSE` (default)

1 Set up Encryption Wallet before creating a collection with SecureFile encryption. For information about the `SET ENCRYPTION WALLET` clause of the `ALTER SYSTEM` statement, see *Oracle Database SQL Language Reference*.

Example 5-1 (page 5-3) is a collection specification for an object whose underlying table is `HR.EMPLOYEES`.

Example 5-1 Collection Specification

```
{
  "schemaName"          : "HR",
  "tableName"           : "EMPLOYEES",
  "contentColumn"       :
  {
    "name"              : "EMP_DOC",
    "sqlType"           : "VARCHAR2",
    "maxLength"         : 4000,
    "validation"        : "STRICT",
    "compress"          : "HIGH",
    "cache"             : true,
    "encrypt"           : "AES128",
  },
  "keyColumn"           :
  {
    "name"              : "EMP_ID",
    "sqlType"           : "NUMBER",
    "assignmentMethod"  : "SEQUENCE",
    "sequenceName"      : "EMPLOYEE_ID_SEQ"
  },
  "creationTimeColumn"  :
  {
    "name"              : "CREATED_ON"
  },
  "lastModifiedColumn"  :
  {
    "name"              : "LAST_UPDATED",
    "index"             : "empLastModIndexName"
  },
  "versionColumn"       :
  {
    "name"              : "VERSION_NUM",
    "method"            : "SEQUENTIAL"
  },
  "mediaTypeColumn"     :
  {
    "name"              : "CONTENT_TYPE"
  },
  "readOnly"            : true
}
```

Topics

- Key Assignment Method (page 5-4)
- Versioning Method (page 5-5)

See Also:

- Key Assignment Method (page 5-4)
- Versioning Method (page 5-5)
- *Oracle Database JSON Developer's Guide* for information about the syntax possibilities used by SQL condition `is json`
- `http://tools.ietf.org/html/rfc4627` for the JSON RFC 4627 standard
- *Oracle Database SecureFiles and Large Objects Developer's Guide* for information about SecureFiles LOB storage

5.1 Key Assignment Method

The key assignment method determines how keys are assigned to objects that are inserted into a collection.

Table 5-2 Key Assignment Methods

Method	**Description**
`SEQUENCE`	Keys are integers generated by a database sequence. You must specify the name of the sequence in the `keyColumn.sequenceName` field.
`GUID`	Keys are generated by the SQL function `SYS_GUID()`, which returns a globally unique `RAW` value (16 bytes). If necessary, the `RAW` value is converted to the SQL data type specified by `keyColumn.sqlType`.
`UUID`	Keys are generated by the built-in `UUID` capability of the Java Virtual Machine (JVM) on which the REST server is running, which returns a universally unique `RAW` value. If necessary, the `RAW` value is converted to the SQL data type specified by `keyColumn.sqlType`.
`CLIENT`	Keys are assigned by the client application (not recommended).

Oracle REST standards strongly recommend using server-assigned keys; that is, avoiding the key assignment method `CLIENT`. If you need simple numeric keys, Oracle recommends `SEQUENCE`. If any unique identifier is sufficient, Oracle recommends `UUID`.

If the key assignment method is `SEQUENCE`, `GUID`, or `UUID`, you insert a object into the collection with the operation POST object (page 4-11). The REST server always interprets `POST` as an insert operation, assigning a key and returning the key in the response body.

If the key assignment method is `CLIENT`, you cannot use `POST` to a insert a object in the collection, because the URL path does not include the necessary key. Instead, you must insert the object into the collection using PUT object (page 4-9). If the object is not already in the collection, then the REST server interprets `PUT` as an insert operation. If the object is already in the collection, then the REST server interprets `PUT` as a replace operation. `PUT` is effectively equivalent to the SQL statement `MERGE`.

Caution:

If client-assigned keys are used and the key column type is `VARCHAR2` then Oracle recommends that the database character set be AL32UTF8. This ensures that conversion of the keys to the database character set is lossless.

Otherwise, if client-assigned keys contain characters that are not supported in your database character set then conversion of the key into the database character set during a read or write operation is lossy. This can lead to duplicate-key errors during insert operations. More generally, it can lead to unpredictable results. For example, a read operation could return a value that is associated with a different key from the one you expect.

5.2 Versioning Method

The versioning method determines how the REST server computes version values for objects when they are inserted into a collection or replaced.

Table 5-3 Versioning Methods

Method	Description
`MD5`	The REST server computes an MD5 checksum on the bytes of object content. For bytes with character data types (such as `VARCHAR2` and `CLOB`), the computation uses UTF-8 encoding. For bytes with data type `BLOB`, the computation uses the encoding used to transmit the POST body, which can be either UTF-8 or UTF-16. For a bulk insert, the request body is parsed as an array of objects and the bytes of the individual objects are re-serialized with UTF-8 encoding, regardless of the encoding chosen for storage. In all cases, the checksum is computed on the bytes as they would be returned by a `GET` operation for the object.
`SHA256` (default)	The REST server computes a SHA256 checksum on the bytes of object content. For bytes with character data types (such as `VARCHAR2` and `CLOB`), the computation uses UTF-8 encoding. For bytes with data type `BLOB`, the computation uses the encoding used to transmit the POST body, which can be either UTF-8 or UTF-16. For a bulk insert, the request body is parsed as an array of objects and the bytes of the individual objects are re-serialized with UTF-8 encoding, regardless of the encoding chosen for storage. In all cases, the checksum is computed on the bytes as they would be returned by a `GET` operation for the specific object.
`UUID`	Ignoring object content, the REST server generates a universally unique identifier (UUID)—a 32-character hexadecimal value—when the object is inserted and for every replace operation (even if the replace operation does not change the object content).
`TIMESTAMP`	Ignoring object content, the REST server generates an integer value, derived from the value returned by the SQL `SYSTIMESTAMP` function. The integer value changes at the level of accuracy of the system clock (typically microseconds or milliseconds).
`SEQUENTIAL`	Ignoring object content, the REST server assigns version 1 when the object is inserted and increments the version value every time the object is replaced.
`NONE`	The REST server does not assign version values during insert and replace operations. During GET operations, any non-null value stored in the version column is used as an ETag. Your application is responsible for populating the version column (using, for example, a PL/SQL trigger or asynchronous program).

`MD5` and `SHA256` compute checksum values that change when the content itself changes, providing a very accurate way to invalidate client caches. However, they are costly, because the REST server must perform a byte-by-byte computation over the objects as they are inserted or replaced.

`UUID` is most efficient for input operations, because the REST server does not have to examine every byte of input or wait for SQL to return function values. However, replacement operations invalidate cached copies even if they do not change object content.

`TIMESTAMP` is useful when you need integer values or must compare two versions to determine which is more recent. As with `UUID`, replacement operations can invalidate cached copies without changing object content. Because the accuracy of the system clock may be limited, `TIMESTAMP` is not recommended if objects can change at very high frequency (many times per millisecond).

`SEQUENTIAL` is also useful when you need integer values or must compare two versions to determine which is more recent. Version values are easily understood by human users, and the version increases despite system clock limitations. However, the increment operation occurs within SQL; therefore, the new version value is not always available to be returned in the REST response body.

6 Security

ORDS, including SODA for REST, uses role-based access control, to secure services. The roles and privileges you need for SODA for REST are described here.

You should be familiar with the ORDS security features before reading this section. See *Oracle REST Data Services Installation, Configuration, and Development Guide* for the relevant information.

Database role **SODA_APP** must be granted to database users before they can use REST SODA. In addition, when a schema is enabled in ORDS using `ords.enable_schema`, a privilege is created such that only users with the application-server role `SODA Developer` can access the service. Specifically, `ords.enable_schema` creates the following privilege mapping:

```
exec ords.create_role('SODA Developer');
exec ords.create_privilege(p_name => 'oracle.soda.privilege.developer',
                           p_role_name => 'SODA Developer');
exec ords.create_privilege_mapping('oracle.soda.privilege.developer', '/soda/*');
```

This has the effect that, by default, a user must have the application-server role `SODA Developer` to access the JSON document store.

You can also add custom privilege mappings. For example:

```
declare
  l_patterns owa.vc_arr;
begin
  l_patterns(1) := '/soda/latest/employee';
  l_patterns(2) := '/soda/latest/employee/*';
  ords.create_role('EmployeeRole');
  ords.create_privilege(p_name      => 'EmployeePrivilege',
                        p_role_name => 'EmployeeRole');
  ords.create_privilege_mapping(p_privilege_name => 'EmployeePrivilege',
                                p_patterns       => l_patterns);
  commit;
end;
```

This example creates a privilege mapping that specifies that only users with role `EmployeeRole` can access the `employee` collection.

When multiple privilege patterns apply to the same resource, the privilege with the most specific pattern overrides the others. For example, patterns `'/soda/latest/employees/*'` and `'/soda/*'` both match the request URL, `http://example.org/ords/quine/soda/latest/employee/id1`.

Since `'/soda/latest/employees/*'` is more specific than `'/soda/*'`, only privilege `EmployeePrivilege` applies to the request.

Note:

SODA_APP is an Oracle Database role. `SODA Developer` is an application-server role.

Topics

- Authentication Mechanisms (page 6-2)
- Security Considerations for Development and Testing (page 6-2)

6.1 Authentication Mechanisms

ORDS supports many different authentication mechanisms. JSON document store REST services are intended to be used in server-to-server interactions. Therefore, two-legged OAuth (the client-credentials flow) is the recommended authentication mechanism to use with the JSON document store REST services. However, other mechanisms such as HTTP basic authentication, are also supported.

See Also:

Oracle REST Data Services Installation, Configuration, and Development Guide

6.2 Security Considerations for Development and Testing

Security considerations for development and testing are presented.

You can disable security and allow anonymous access by removing the default privilege mapping:

```
exec ords.delete_privilege_mapping('oracle.soda.privilege.developer', '/soda/*')
```

However, Oracle does *not* recommend that you allow anonymous access in production systems. That would allow an unauthenticated user to read, update, or drop any collection.

You can also use command `ords.war user` to create test users that have particular roles. For example (where *new_password* is a placeholder for the password for user `bob`):

```
# Create user bob with role SODA Developer
java -jar ords.war user bob "SODA Developer"

# Access the JSON document store as user bob using basic authentication
curl -u bob:new_password https://example.com/ords/scott/soda/latest/
```

Index

C

collections
- creating, 3-2
- deleting, 3-4
- deleting documents from, 3-6
- listing, 3-3
- listing documents in, 3-7
- specifications for, 5-1

D

database role
- SODA_APP, 6-1

DELETE collection operation, 4-10
DELETE object operation, 4-11
deleting collections, 3-4
deleting documents from collections, 3-6
documents
- deleting from collections, 3-6
- filtering in collections, 3-10
- inserting into collections
  - in bulk from JSON array, 3-6
  - one at a time, 3-4
- listing in collections, 3-7
- retrieving from collections, 3-5
- updating in collections, 3-9

F

filtering documents in collections, 3-10

G

GET collection operation, 4-5
GET object operation, 4-8
GET schema operation, 4-4

I

inserting documents into collections
- in bulk from JSON array, 3-6
- one at a time, 3-4

installing SODA for REST, 2-1

K

key assignment method, 5-4

L

listing collections in schema, 3-3
listing documents in collections, 3-7

O

Oracle REST API HTTP operations
- response bodies, 4-3
- summary of, 4-1
- URI form for, 4-2

P

POST array insert operation, 4-13
POST bulk delete operation, 4-15
POST object operation, 4-11
POST query operation, 4-12
PUT collection operation, 4-9
PUT object operation, 4-9

Q

query-by-example (QBE)
- examples, 3-10

R

REST architectural style, 1-2
retrieving documents from collections, 3-5

S

security, 6-1
SODA_APP database role, 6-1
specifications

specifications *(continued)*
    collection, *5-1*

U

updating documents in collections, *3-9*

V

versioning method, *5-5*