Contents

Preface ......................................................................................................................................................... ix
  Audience .................................................................................................................................................... ix
  Documentation Accessibility .................................................................................................................... ix
  Related Documents ................................................................................................................................. ix
  Conventions ............................................................................................................................................... ix

1 SODA for REST

SODA for REST Overview ......................................................................................................................... 1-1
  REST ...................................................................................................................................................... 1-2
  Document Collections and Document-Centric APIs ........................................................................ 1-2
  SODA for REST Operations .................................................................................................................. 1-2

SODA for REST Installation ..................................................................................................................... 1-3

Getting Started with SODA for REST ..................................................................................................... 1-4
  Creating a New Collection .................................................................................................................... 1-5
  Getting the List of Available Collections .............................................................................................. 1-6
  Deleting a Collection ............................................................................................................................. 1-6
  Inserting a Document into a Collection ................................................................................................. 1-7
  Retrieving a Document from a Collection ............................................................................................. 1-8
  Deleting a Document from a Collection ................................................................................................ 1-8
  Bulk-Inserting Documents from a JSON Array ...................................................................................... 1-9
  Listing the Documents in a Collection ................................................................................................ 1-10
  Updating a Document in a Collection .................................................................................................. 1-12
  Using a Filter Specification to Select Documents From a Collection .............................................. 1-13
    QBE.1.json ........................................................................................................................................ 1-13
    QBE.2.json ........................................................................................................................................ 1-13
    QBE.3.json ........................................................................................................................................ 1-14
    QBE.4.json ........................................................................................................................................ 1-14

SODA for REST HTTP Operations .......................................................................................................... 1-14
  SODA for REST HTTP Operation URIs ................................................................................................. 1-15
  SODA for REST HTTP Operation Response Bodies ............................................................................. 1-15
  GET schema .......................................................................................................................................... 1-17
    URL Pattern for GET schema .............................................................................................................. 1-17
    Response Codes for GET schema ......................................................................................................... 1-17
  GET collection ....................................................................................................................................... 1-17
    URL Pattern for GET collection ......................................................................................................... 1-18
Filter Specification Details

Filter Specification Overview

Paths

Response Codes for GET collection ......................................................... 1-18
Links Array for GET collection ............................................................... 1-19
GET object .............................................................................................. 1-20
URL Pattern for GET object ................................................................. 1-20
Request Headers for GET object ......................................................... 1-20
Response Codes for GET object ............................................................ 1-20
PUT collection ..................................................................................... 1-21
URL Pattern for PUT collection ......................................................... 1-21
Request Body for PUT collection (Optional) ....................................... 1-21
Response Codes for PUT collection .................................................... 1-21
PUT object ............................................................................................ 1-21
URL Pattern for PUT object ................................................................. 1-21
Request Body for PUT object ............................................................... 1-21
Response Codes for PUT object ............................................................ 1-21
DELETE collection ................................................................................ 1-22
URL Pattern for DELETE collection .................................................. 1-22
Response Codes for DELETE collection .......................................... 1-22
DELETE object ..................................................................................... 1-22
URL Pattern for DELETE object ......................................................... 1-22
Response Codes for DELETE object .................................................... 1-22
POST object .......................................................................................... 1-23
URL Pattern for POST object ............................................................ 1-23
Request Body for POST object ........................................................... 1-23
Response Codes for POST object ....................................................... 1-23
POST query .......................................................................................... 1-24
URL Pattern for POST query ............................................................. 1-24
Request Body for POST query ............................................................ 1-24
Response Codes for POST query ......................................................... 1-24
POST array insert ................................................................................ 1-24
URL Pattern for POST array insert .................................................. 1-24
Request Body for POST array insert ................................................. 1-25
Response Codes for POST array insert ............................................. 1-25
POST bulk delete ................................................................................. 1-25
URL Pattern for POST bulk delete .................................................... 1-25
Request Body for POST bulk delete (Optional) ................................. 1-26
Response Codes for POST bulk delete .............................................. 1-26
Paths ................................................................................................... 1-26
Filter Specification Overview ............................................................ 1-28
Filter Specification Details ................................................................. 1-30
Composite Queries ............................................................................... 1-30
Omitting $query .................................................................................. 1-31
Sorting Selected Objects ..................................................................... 1-31
Filter Conditions ................................................................................ 1-31
Field Clause ......................................................................................... 1-31
Comparison Condition ........................................................................ 1-32
Omitting $eq ....................................................................................... 1-32
Downscoped Field Condition ............................................................ 1-32
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison Conditions and Arrays</td>
<td>1-33</td>
</tr>
<tr>
<td>Conjunction Clause</td>
<td>1-33</td>
</tr>
<tr>
<td>Omitting $and</td>
<td>1-33</td>
</tr>
<tr>
<td>ID Clause</td>
<td>1-34</td>
</tr>
<tr>
<td>Omitting $id for Array of Keys</td>
<td>1-34</td>
</tr>
<tr>
<td>Collection Specifications</td>
<td>1-34</td>
</tr>
<tr>
<td>Key Assignment Method</td>
<td>1-38</td>
</tr>
<tr>
<td>Versioning Method</td>
<td>1-38</td>
</tr>
<tr>
<td>Security</td>
<td>1-40</td>
</tr>
<tr>
<td>Authentication Mechanisms</td>
<td>1-40</td>
</tr>
<tr>
<td>Development and Testing</td>
<td>1-41</td>
</tr>
</tbody>
</table>
List of Examples

1–1 Response Body ................................................................................................................... 1-16
1–2 Filter Specification............................................................................................................ 1-34
1–3 Collection Specification........................................................................................................ 1-37
List of Tables

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1–1</td>
<td>SODA for REST HTTP Operations</td>
</tr>
<tr>
<td>1–2</td>
<td>Fields That Can Appear in Response Bodies</td>
</tr>
<tr>
<td>1–3</td>
<td>Additional Response Body Fields for Operations that Return Objects</td>
</tr>
<tr>
<td>1–4</td>
<td>Relationship of GET collection Parameters to Mode and Links Array</td>
</tr>
<tr>
<td>1–5</td>
<td>Filter Specifications</td>
</tr>
<tr>
<td>1–6</td>
<td>Operands and Operators in Comparison Conditions</td>
</tr>
<tr>
<td>1–7</td>
<td>Collection Specification Fields</td>
</tr>
<tr>
<td>1–8</td>
<td>Key Assignment Methods</td>
</tr>
<tr>
<td>1–9</td>
<td>Versioning Methods</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boldface</strong></td>
<td>Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.</td>
</tr>
<tr>
<td><em>italic</em></td>
<td>Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.</td>
</tr>
<tr>
<td>monospace</td>
<td>Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.</td>
</tr>
</tbody>
</table>
This chapter describes the Oracle SODA for REST API and explains how to install and use it. It assumes that you are familiar with the following:

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

See Also: Oracle Database SODA for Java Developer’s Guide, which explains how to use the Java client API on which SODA for REST is built.

Topics:
- SODA for REST Overview
- SODA for REST Installation
- Getting Started with SODA for REST
- SODA for REST HTTP Operations
- Paths
- Filter Specification Overview
- Filter Specification Details
- Collection Specifications
- Security

SODA for REST Overview

Simple Oracle Document Access (SODA, for short) lets 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.

SODA for REST is a REST API that provides SODA using the representational state transfer (REST) architectural style. You can use it to perform create, retrieve, update, and delete (CRUD) operations on documents of any kind, and you can use it to query JSON documents.

Topics:
- REST
REST

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, retrieve, update and delete (CRUD) operations to the 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—traditional programming languages (such as C, C#, C++, JAVA and PL/SQL) and most modern scripting languages (including JavaScript, Perl, Python, and Ruby).

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


Document Collections and Document-Centric APIs

Typically, you use a document collection to manage the documents that your application uses when it must persist state. In a document collection, each document has a unique identifier. The identifier is typically assigned by the server when the document is created, but client-assigned identifiers can also be used. Document identifiers are metadata—data about the individual documents. Other metadata that a document collection can track for each document include the date and time that it was created and the date and time that it was last modified.

SODA for REST Operations

SODA for REST provides an HTTP-based service that implements the REST architectural style and allows documents to be stored in, and retrieved from, document collections managed by Oracle Database. The API defines a set of simple operations that you can perform, including the following:

- List the set of available collections
- Create a collection
- Drop a collection
- Insert a document into a collection
- Retrieving a document from a collection
- Update a document in a collection
- Delete a document from a collection
- List the contents of a collection
Search a collection

For detailed information about the operations defined by SODA for REST, see “SODA for REST HTTP Operations” on page 1-14.

Your application can use the API operations to create and manipulate the 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 for REST Installation

To install SODA for REST, follow these steps:

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:
grant SODA_APP to schema;

8. Only if you are in a development environment:
1. Remove the default security constraints:

```sql
begin
    ords.delete_privilege_mapping('oracle.soda.privilege.developer',
        '/soda/*');
end;
```

This enables anonymous access to the service and is not recommended for production systems. For more information about security, see "Security" on page 1-40.

2. Start ORDS in standalone mode:

```sh
gnu -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. If the installation succeeded, you see:

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

**See Also:**
- "Security" on page 1-40
- Oracle REST Data Services Installation, Configuration, and Development Guide

### Getting Started with SODA for REST

This introduction walks you through the basic SODA for REST operations step by step, using examples that you can run. The examples assume that you started ORDS as instructed in "SODA for REST Installation" on page 1-3, enabling ORDS in `schema`. The examples in this topic use the command line tool to send REST requests to the server. For information about the command, see:

- [http://curl.haxx.se/](http://curl.haxx.se/)
- Oracle REST Data Services Installation, Configuration, and Development Guide

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

### Steps:

1. **Creating a New Collection**
2. Getting the List of Available Collections

3. Deleting a Collection

4. Inserting a Document into a Collection

5. Retrieving a Document from a Collection

6. Deleting a Document from a Collection

7. Bulk-Inserting Documents from a JSON Array

8. Listing the Documents in a Collection

9. Updating a Document in a Collection

10. Using a Filter Specification to Select Documents From a Collection

Creating a New Collection

To create a new collection, run this command:

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

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

The 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.

The successful PUT collection operation creates an RDBMS table to store the new collection. One way to see the details of this table is with the 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" on page 1-34.

Caution: To drop a collection proceed as described in "Deleting a Collection" on page 1-6. Do not use SQL to drop the database table that underlies the collection. Collections have persisted metadata, in addition to the documents that are stored in the collection table.
Getting Started with SODA for REST

Getting the List of Available Collections

To get a list of available collections in schema, run this command:

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

The preceding command 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
    }
  ],
  "more": false
}
```

The response body includes all available collections in schema, which in this case is only 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. For information about collection specifications, see "Collection Specifications" on page 1-34.

See Also:

- "GET schema" on page 1-17 for more information about this operation
- "Deleting a Collection" on page 1-6

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

The 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
}
```

Recreate MyCollection 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" on page 1-22 for more information about this operation
- "Inserting a Document into a Collection" on page 1-7

Inserting a Document into a Collection

This example uses the file po.json, which was included in the download. The file po.json 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 and outputs something like this:

```
{
    'items': [
        {
            'id': '2FFD968C531C49B9A7EAC4398DFC02EE',
            'etag': 'C1354F27A5180FF7B828F01CBBC84022DCF5F7209DBF0E6DFCC626E3B0400C3',
            '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 and 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 ("id" value), to use for retrieval— see "Retrieving a Document from a Collection" on page 1-8.

See Also:
- "POST object" on page 1-23 for more information about this operation
- "Retrieving a Document from a Collection" on page 1-8

Retrieving a Document from a Collection

To retrieve the document that you inserted in the preceding step, run this command (where id is the document identifier that you copied in the preceding step):

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


If id does not exist in MyCollection, 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" on page 1-20 for more information about this operation
- "Deleting a Document from a Collection" on page 1-8

Deleting a Document from a Collection

To delete from MyCollection the document that you retrieved in the preceding step, 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 the URL http://localhost:8080/ords/schema/soda/latest/MyCollection/id and returns:

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

The 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" on page 1-22 for more information about this operation
- "Bulk-Inserting Documents from a JSON Array" on page 1-9

Bulk-Inserting Documents from a JSON Array

The bulk insert operation inserts a set of documents into a collection. The set is a JSON array of one or more documents. The bulk insert operation is also called POST array insert.

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

```
```

The parameter 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 outputs something like this:

```
{
  "items" : [
    {
      "id" : "6DEAF8F011FD43249E5F60A93B850AB9",
      "etag" : "49205D7E916EAED914465FCFF029B2795885A1914966E0AB82D4CCD8E2EAF8E",
      "lastModified" : "2014-09-22T22:39:15.546435Z",
      "created" : "2014-09-22T22:39:15.546435Z"
    },
    {
      "id" : "C9FF7685D48E4E4B8641D8401ED0FB68",
      "etag" : "F3EB514BEDE6A6CC337ADA0F5BE6DEFC5D451E68CE645729224BB6707F8E1F4F",
      "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 timestamp for each inserted document.

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

```
select JSON_VALUE(JSON_DOCUMENT format JSON, '$.Reference')
from "MyCollection"
where ID = 'id';
```

```
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 operators. You cannot use the simplified JSON syntax.

See Also:

■ "POST array insert" on page 1-24 for more information about this operation

■ "Listing the Documents in a Collection" on page 1-10

Listing the Documents in a Collection

To list the documents in a collection, you use 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

See Also: "Using a Filter Specification to Select Documents From a Collection" on page 1-13, which lets you list documents based on content

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


The preceding command outputs something like this:

```
{
  "items" : [
    {
      "id" : "023C4A6581D84B71A5C0D5D364CE8484",
      "etag":"3484DB604DDA3FBC0C681C37972E7DD8C5F4457ACE32BD1696D4388C5A7C0E",
      "lastModified" : "2014-09-22T22:39:15.546435Z",
      "created": "2014-09-22T22:39:15.546435Z"
    },
    {
      "id" : "06DD0319148E40A7B8AA48E39E739184",
      "etag" : "A19A1B9A38B1BB3E52B93350FB076309CFAC0072A2BEC95BCA44D4849DD",
      "lastModified" : "2014-09-22T22:39:15.546435Z",
      "created" : "2014-09-22T22:39:15.546435Z"
    }
  ],
  "hasMore" : false,
  "count" : 70,
  "offset" : 0
}
```
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:

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

The preceding command outputs something like this:

```json
{
  "items": [ ... ],
  "hasMore" : true,
  "count" : 10,
  "offset" : 0,
  "limit" : 10,
  "links" : [{
    "rel" : "next",
  }]
}
```

**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 ("id" value), to use for updating — see "Updating a Document in a Collection" on page 1-12.
Updating a Document in a Collection

To update a document in a collection—that is, to replace it with a newer version—you use a **PUT** operation.

The behavior of the **PUT** operation depends on the key assignment method that the collection uses. If the collection uses server-assigned keys (as **MyCollection** does), it is an error to try to update a nonexistent document (that is, to specify a key that does not belong to a document in the collection). If the collection uses client-assigned keys, then updating a nonexistent document inserts into the collection a new document with the specified key. (For information about key assignment methods, see "Key Assignment Method" on page 1-38.)

Retrieve a document from **MyCollection** by running this command (where **id** is the document identifier that you copied in the preceding step):

```
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: A0B07E0A6D000358C546DC5D0B5D09C5B48A1A5F6F2C6D62180B579CCBC6D
Last-Modified: Mon, 22 Sep 2014 16:42:35 PDT
Location: http://localhost:8080/ords/schema/soda/latest/MyCollection/023C4A6581D871A5C0D364CE8484/
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...."
}
```
Using a Filter Specification to Select Documents From a Collection

You can use a filter specification, or query-by-example (QBE), to define query criteria for selecting documents from a collection. The examples in this topic use the QBE.*.json files that are included in the zip file that you downloaded in installation step 3. They are in the directory ORDS_HOME/examples/soda/getting-started.

QBE.1.json

QBE.1.json has this QBE:

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

To execute the preceding query, run this command:

```
```

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

The preceding command returns a list of nine documents, each of which has "TGATES" as the value of the field "User". Because the command has no fields parameter, the default fields=all applies, and the response body contains both the metadata and 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 will need the content later. Retrieving the content from the response body is more efficient than retrieving it from the server.

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

QBE.2.json

QBE.2.json has this QBE:

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

This query selects documents where the value of the field UPCCode equals "13023015692". UPCCode is a field of the object Part, which is a field of the array LineItems. Because no array offset is specified for LineItems, the query searches all elements of the array.
Note: The keyword "$eq" in the query is implied. For details, see "Comparison Condition" on page 1-32.

QBE.3.json
QBE.3.json has this QBE:

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

This query selects documents where the value of the field ItemNumber, in the object LineItems, is greater than 4. The keyword "$gt" is required.

QBE.4.json
QBE.4.json has this QBE:

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

This query selects documents where both the value of the field UPCCode equals "13023015692" and the value of the field ItemNumber equals 3. The keyword $and is optional—see "Omitting $and" on page 1-33.

SODA for REST HTTP Operations

Table 1–1 summarizes the HTTP operations that SODA for REST provides. For complete descriptions of the operations, click the links in the left column.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET schema</td>
<td>Gets some or all collection names in a schema.</td>
</tr>
<tr>
<td>GET collection</td>
<td>Gets all or a subset of objects from a collection, using parameters to specify the subset. You can page through the return set.</td>
</tr>
<tr>
<td>GET object</td>
<td>Gets a specified object from a collection.</td>
</tr>
<tr>
<td>PUT collection</td>
<td>Creates a collection if it does not exist.</td>
</tr>
<tr>
<td>PUT object</td>
<td>Replaces a specified object with an uploaded object (typically a new version).</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>DELETE collection</td>
<td>Deletes a collection.</td>
</tr>
<tr>
<td>DELETE object</td>
<td>Deletes a specified object from a collection.</td>
</tr>
<tr>
<td>POST object</td>
<td>Puts an uploaded object in a specified collection, assigning and returning its key. Collection must use server-assigned keys.</td>
</tr>
<tr>
<td>POST query</td>
<td>Gets all or a subset of objects from a collection, using a filter to specify the subset. You cannot page through the return set.</td>
</tr>
<tr>
<td>POST array insert</td>
<td>Inserts an array of objects into a specified collection, assigning and returning their keys.</td>
</tr>
</tbody>
</table>
SODA for REST HTTP Operation URIs

A SODA for REST HTTP operation has a Universal Resource Identifier (URI) of 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 RDBMS schema that has been configured as an end point for SODA for REST.
- **soda** is the name given to the RDBMS 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 in either 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.

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" on page 1-20, the response body is a single object.

Table 1–2 lists and describes fields that can appear in response bodies.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>String that uniquely identifies an object (typically a JSON document) in a collection.</td>
</tr>
</tbody>
</table>
If an operation creates or returns objects, then its response body can have the additional fields in Table 1–3. The additional fields appear after the `items` field.

### Table 1–3   Additional Response Body Fields for Operations that Return Objects

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>name</code></td>
<td>Name of collection. This field appears only in the response body of &quot;GET schema&quot; on page 1-17.</td>
</tr>
<tr>
<td><code>properties</code></td>
<td>Properties of collection. This field appears only in the response body of &quot;GET schema&quot; on page 1-17.</td>
</tr>
<tr>
<td><code>hasMore</code></td>
<td>true if limit was reached before available objects were exhausted, false otherwise. This field is always present.</td>
</tr>
<tr>
<td><code>limit</code></td>
<td>Server-imposed maximum collection (row) limit.</td>
</tr>
<tr>
<td><code>offset</code></td>
<td>Offset of first object returned (if known).</td>
</tr>
<tr>
<td><code>count</code></td>
<td>Number of objects returned. This is the only field that can appear in the response body of &quot;POST bulk delete&quot; on page 1-25.</td>
</tr>
<tr>
<td><code>totalResults</code></td>
<td>Number of objects in collection (if requested)</td>
</tr>
<tr>
<td><code>links</code></td>
<td>Possible final field for GET collection operation. For details, see &quot;GET collection&quot; on page 1-17.</td>
</tr>
</tbody>
</table>

Example 1–1 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 1–1 Response Body

```json
{
  "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
    }
  ]
}
```
GET schema

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

See Also: "Listing the Documents in a Collection" on page 1-10

URL Pattern for GET schema
/ords/schema/soda/version/

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>limit=(n)</td>
<td>Limits number of listed collection names to (n).</td>
</tr>
<tr>
<td>fromID=collection</td>
<td>Starts getting with (collection) (inclusive).</td>
</tr>
</tbody>
</table>

Response Codes for GET schema

200
Success—response body contains names and properties of collections in schema, ordered by name. For example:

```json
{
  "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 schema was not found or access is not authorized.

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" on page 1-24, 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" on page 1-10

URL Pattern for GET collection

/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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>limit=n</td>
<td>Limits number of objects to n.</td>
</tr>
<tr>
<td>offset=n</td>
<td>Skips n objects before getting first object.</td>
</tr>
<tr>
<td>fields={id</td>
<td>value</td>
</tr>
<tr>
<td>totalResults=true</td>
<td>Returns number of objects in collection. Note: Inefficient</td>
</tr>
<tr>
<td>fromID=key</td>
<td>Starts getting objects after key, in ascending order.</td>
</tr>
<tr>
<td>toID=key</td>
<td>Stops getting objects before key, in descending order.</td>
</tr>
<tr>
<td>after=key</td>
<td>Starts getting objects after key, in ascending order.</td>
</tr>
<tr>
<td>before=key</td>
<td>Stops getting objects before key, in descending order.</td>
</tr>
<tr>
<td>since=timestamp</td>
<td>Gets only objects with time stamp later than timestamp.</td>
</tr>
<tr>
<td>until=timestamp</td>
<td>Gets only objects with time stamp earlier than timestamp.</td>
</tr>
</tbody>
</table>

Response Codes for GET collection

200
Success—response body contains the specified objects from collection (or only their keys, if you specified fields=id). For example:

```json
{
  "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",
```
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" on page 1-19.

401
Read access to collection is not authorized.

404
Collection was not found.

**Links Array for GET collection**

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 1–4 shows how GET collection parameters determine mode and the existence and content of the `links` array.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Mode</th>
<th>Links Array</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fields=id</code></td>
<td>Keys-only</td>
<td>Does not exist (regardless of other parameters).</td>
</tr>
</tbody>
</table>
### GET object

GET object gets a specified object from a specified collection.

**See Also:** "Retrieving a Document from a Collection" on page 1-8

### URL Pattern for GET object

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

No parameters.

### Request Headers for GET object

This operation accepts these optional request headers:

<table>
<thead>
<tr>
<th>Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>If-Modified-Since=timestamp</td>
<td>Returns response code 304 if object has not changed since timestamp.</td>
</tr>
<tr>
<td>If-None-Match</td>
<td>Returns response code 304 if object etag matches the current checksum or version.</td>
</tr>
</tbody>
</table>

### Response Codes for GET object

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").

401  
Read access to collection or object is not authorized.
404
Collection or object was not found.

**PUT collection**

PUT collection creates a collection if it does not exist.

See Also:  "Creating a New Collection" on page 1-5

**URL Pattern for PUT collection**

/ords/schema/soda/version/collection/

No parameters.

**Request Body for PUT collection (Optional)**
See "Collection Specifications" on page 1-34.

**Response Codes for PUT collection**

200
Collection with the same name and properties already exists.

201
Success—collection was created.

401
Collection creation is not authorized.

**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" on page 1-12

**URL Pattern for PUT object**

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

No parameters.

**Request Body for PUT object**

Uploaded object.

**Response Codes for PUT object**

200
Success—object was replaced.
401
Updating collection is not authorized.

405
Collection is read-only.

DELETE collection

DELETE collection deletes a collection. (To delete all objects from a collection, but not delete the collection itself, use "POST bulk delete" on page 1-25.)

See Also: "Deleting a Collection" on page 1-6

URL Pattern for DELETE collection
/ords/schema/soda/version/collection/

No parameters.

Response Codes for DELETE collection

200
Success—collection was deleted.

401
Deleting collection is not authorized.

404
Collection was not found.

DELETE object

DELETE object deletes a specified object from a specified collection.

See Also: "Deleting a Document from a Collection" on page 1-8

URL Pattern for DELETE object
/ords/schema/soda/version/collection/key/

No parameters.

Response Codes for DELETE object

200
Success—object was deleted.

401
Either deleting from collection or deleting this object is not authorized.

404
Object was not found.
**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" on page 1-21. For information about key assignment methods, see "Key Assignment Method" on page 1-38.

See Also: "Inserting a Document into a Collection" on page 1-7

**URL Pattern for POST object**
/ords/schema/soda/version/collection/

No parameters.

**Request Body for POST object**
Uploaded object to be inserted into collection.

**Response Codes for POST object**

201
Success—object is in collection; response body contains server-assigned key and possibly other information. For example:

```json
{
  "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).
**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" on page 1-17, which gets all or a subset of objects 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" on page 1-13

**URL Pattern for POST query**

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

Parameters are optional except as noted.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action=query</td>
<td>Required. Specifies kind of action.</td>
</tr>
<tr>
<td>limit=n</td>
<td>Limit number of returned objects to n.</td>
</tr>
<tr>
<td>offset=n</td>
<td>Skip n objects before returning objects.</td>
</tr>
<tr>
<td>fields={id</td>
<td>value</td>
</tr>
</tbody>
</table>

**Request Body for POST query**

See "Filter Specification Overview" on page 1-28. If you omit the filter specification object, then the operation gets all objects in the collection.

**Response Codes for POST query**

- **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.

**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" on page 1-9

**URL Pattern for POST array insert**

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action=insert</td>
<td>Required. Specifies kind of action.</td>
</tr>
</tbody>
</table>
Request Body for POST array insert
Array of objects.

Response Codes for POST array insert

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.

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" on page 1-22.

---

URL Pattern for POST bulk delete

Either of the following:

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

/ords/schema/soda/version/collection?action=truncate
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action=delete</td>
<td><strong>Required.</strong> Specifies the deletion of all or a subset of objects from <em>collection</em>, using an optional filter to specify the subset. See the following warning.</td>
</tr>
<tr>
<td>action=truncate</td>
<td><strong>Required.</strong> Specifies the deletion of all objects from <em>collection</em>. Does not use a filter.</td>
</tr>
</tbody>
</table>

**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.

**Request Body for POST bulk delete (Optional)**


**Response Codes for POST bulk delete**

**200**

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

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

**401**

Deleting from collection is not authorized.

**405**

Collection is read-only.

**Paths**

SODA for REST specifications contain paths, each of which targets a value in a JSON document. A path is composed of a series of steps.

**Note:** In paths, you must use strict JSON syntax. That is, you must enclose every nonnumeric value in double quotation marks. For information about strict and lax JSON syntax, see *Oracle XML DB Developer’s Guide*.

The characters used in path steps are of two kinds: syntactic and allowed. **Syntactic characters** have special syntactic meaning for JSON. They are the following:

- Period (.), which separates a parent-object field name from a child-object field name.
- Brackets ([ and ]), which are array delimiters.
- Comma (,), which separates array elements or index components.
- Wildcard (*), which is a placeholder. It matches any index in an array step and any field name in a field step.
**Allowed characters** are those that are not syntactic.

There are two kinds of steps in a path: field steps and array steps.

A **field step** is one of the following:

- The wildcard character * (by itself)
- A sequence of allowed characters—for example, cat
- A sequence of characters (allowed or syntactic) enclosed in backquote characters (‘)—for example, `dog` and `cat*dog`

Within a field step that is enclosed in backquote characters, a syntactic character does not act syntactically; it is treated literally as an ordinary character. You must enclose any field step that contains a syntactic character in a pair of backquote characters, if you intend for the syntactic character to be treated literally.

Because all of the characters in dog are allowed, backquote characters are optional in `dog`. Because each of the following field steps contains a syntactic character, they must be enclosed in backquote characters:

- `cat.dog`
- `cat[dog]`
- `cat, dog`
- `cat*dog`

In `cat*dog` the asterisk does not act as a wildcard. Because it is escaped by backquotes, it acts as an ordinary character. But in the path `{ "*.b" : 42 }`, the unescaped asterisk acts as a wildcard; it is a placeholder for a field name. Similarly, the unescaped period also acts syntactically.

If a step that you enclose in backquote characters contains a backquote character, then you must represent that character using two consecutive backquote characters. For example: `Customer``s Comment`.

A period (.) must be followed by a field step. After the first step in a path, each field step must be preceded by a period.

An **array step** is delimited by brackets ([ and ]). Inside the brackets can be either:

- The wildcard character * (by itself)
- One or more of these index components:
  - A single index, which is an integer greater than or equal to zero
  - An index range, which has this syntax:
    
    \[ x \text{ to } y \]

    \( x \) and \( y \) are integers greater than or equal to zero, and \( x \) is less than or equal to \( y \). There must be at least one whitespace character between \( x \) and \( \text{ to } \) and between \( \text{ to } \) and \( y \).

Multiple components must be separated by commas (,). In a list of multiple components, indexes must be in ascending order, and ranges cannot overlap.

For example, these are valid array steps:

- [*]
- [1]
- [1, 2, 3]
- [1 to 3]
- [1, 3 to 5]
The following are *not* valid array steps:

- [*, 6]
- [3, 2, 1]
- [3 to 1]
- [1 to 3, 2 to 4]

**Filter Specification Overview**

A filter specification, or **filter**, evaluates to either true or false. Some SODA for REST HTTP operations use filters to select all objects in a collection which satisfy the filter condition (the filter specification evaluates to true for those objects). A filter is also called a **query-by-example (QBE)**.

Table 1–5 gives the syntax of each filter specification and tells when the condition that the filter specifies is true.

---

**Note:**

- Although the syntax descriptions omit them to make reading easier, you must enclose each **field**, **keyword**, **string constant**, and **string key** in double quotation marks. For example:

  ```
  { "nextObject" : { "$exists" : true } }
  ```

- A **field** cannot match a keyword.
- For complete syntax information, see "Filter Specification Details" on page 1-30.

---

**Table 1–5  Filter Specifications**

<table>
<thead>
<tr>
<th>Filter Syntax</th>
<th>True when...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{ field : { $exists : boolean } }</code></td>
<td>One of the following is true:</td>
</tr>
<tr>
<td></td>
<td>- boolean is false, 0 (zero), or null, and field does not exist.</td>
</tr>
<tr>
<td></td>
<td>- boolean is any value except false, 0 (zero), or null, and field exists.</td>
</tr>
<tr>
<td><code>{ field : { $eq : scalar } }</code></td>
<td>field equals scalar.</td>
</tr>
<tr>
<td><code>{ field : { $ne : scalar } }</code></td>
<td>field does not equal scalar, or field does not exist.</td>
</tr>
<tr>
<td><code>{ field : { $gt : string-or-number } }</code></td>
<td>field is greater than string-or-number.</td>
</tr>
<tr>
<td><code>{ field : { $lt : string-or-number } }</code></td>
<td>field is less than string-or-number.</td>
</tr>
<tr>
<td><code>{ field : { $gte : string-or-number } }</code></td>
<td>field is greater than or equal to string-or-number.</td>
</tr>
<tr>
<td><code>{ field : { $lte : string-or-number } }</code></td>
<td>field is less than or equal to string-or-number.</td>
</tr>
<tr>
<td><code>{ field : { $in : [ scalar ... ] } }</code></td>
<td>field equals some scalar in the array.†</td>
</tr>
<tr>
<td><code>{ field : { $nin : [ scalar ... ] } }</code></td>
<td>field does not exist or does not equal any scalar in the array.†</td>
</tr>
<tr>
<td><code>{ field : { $all : [ scalar ... ] } }</code></td>
<td>field is an array that contains each scalar specified.†</td>
</tr>
<tr>
<td><code>{ field : { $startsWith : string } }</code></td>
<td>field starts with string.</td>
</tr>
<tr>
<td><code>{ field : { $regex : regular_expression } }</code></td>
<td>field matches regular_expression (a string).</td>
</tr>
</tbody>
</table>
In Table 1–5:

- **field** is a string that is interpreted as a path from the root of the candidate JSON document.
- **condition** evaluates to true or false.
  - With $not, condition is a single operator/value pair.
  - With $or and $nor, condition is a filter condition that does not include an ID clause.
  - With all other keywords in Table 1–5, condition is a filter condition.
- **scalar** is a JSON scalar: a string, number, null, true, or false.
- **string-or-number** is either a JSON string or a JSON number.
- **string** is a JSON string.
- **regular_expression** is a string whose text is a SQL regular expression. For information about SQL regular expressions, see *Oracle Database SQL Language Reference*.

A syntax error is raised if any of the arguments to a filter-condition operator (keyword) is not of the specified type (for example, if $gt is passed an argument that is not a string or a number).

**See Also:**

- "Paths" on page 1-26 for information about path strings
- "Filter Conditions" on page 1-31
- "Comparison Condition" on page 1-32 for information about $not
- "ID Clause" on page 1-34
Filter Specification Details

Note: In filter specifications, you must use strict JSON syntax. That is, you must enclose every nonnumeric value in double quotation marks. For information about strict and lax JSON syntax, see Oracle XML DB Developer’s Guide.

A query filter specification is a JSON object. There are two types of query filter specifications:

- Composite Queries
- Filter Conditions

A composite query can appear only at the top level, and can be empty. A filter condition cannot be empty.

Composite Queries

Note: A composite query can appear only at the top level. That is, you cannot nest a composite query inside another composite query or inside a filter condition.

A composite query consists of zero, one, or both of these clauses:

- $query condition
  
  condition is a filter condition (see "Filter Conditions" on page 1-31).

- $orderby orderby_clause
  
  For information about orderby_clause, see "Sorting Selected Objects" on page 1-31.

Neither clause can appear more than once.

The following composite query has both possible clauses:

```json
{
  "$query" : { "salary" : { "gt" : 10000 } },
  "$orderby" : { "salary" : 1, "address.city" } }
}
```

The following composite query has no clauses (that is, it is empty):

```json
{}
```

An empty query matches all objects in a collection.

Topics:

- Omitting $query
- Sorting Selected Objects
**Omitting $query**

If you do not include keyword $orderby, then you can omit the keyword $query. For example, these QBEs are equivalent; both select all objects that have a salary value greater than 10,000 and less than 20,000.

```
{ "$query" : { "salary" : { "$gt" : 10000, "$lte" : 20000 } } }
```

```
{ "salary" : { "$gt" : 10000, "$lte" : 20000 } }
```

**Sorting Selected Objects**

This filter specification returns the objects that it selects in sorted order:

```
{ "$query" : condition,
  "$orderby" : { field : n, field : n, ... } }
```

`field` is a string that is interpreted as a path from the root of the candidate object. For information about path strings, see "Paths" on page 1-26.

`n` is either 1 or -1, and sorts field values in ascending or descending order, respectively.

For example, this filter specification selects objects in which the field `salary` has a value greater than 10,000 and less than or equal to 20,000 and sorts the objects first in descending order of `age` and then in ascending order of `zipcode`:

```
{ "$query" : { "salary" : { "$gt" : 10000, "$lte" : 20000 } },
  "$orderby" : { "age" : -1, "zipcode" : 1 } }
```

The equivalent `WHERE` clause of the SQL `SELECT` statement is:

```
WHERE (salary > 10000) AND (salary <= 20000)
ORDER BY age DESC, zipcode ASC
```

**Filter Conditions**

A filter condition consists of one or more of these filter clauses:

- **Field Clause**
- **Conjunction Clause**
- **ID Clause**

A filter condition is true if and only if all of its clauses are true.

A filter condition cannot be empty.

**Field Clause**

A field clause is a field name followed by either a scalar value or a JSON object.

If the field name is followed by a scalar value, then the field clause is treated as an equality test for that value. For example, this field clause tests whether the field `salary` has the value 10000:

```
'salary' : 10000
```

If the field name is followed by a JSON object, then the field name must be the parent field part of a downscoped field condition or the JSON object must be a comparison condition.
Comparison Condition A comparison condition consists of one or more operator/value pairs. The operators appear as field names and the field values are the operands. For example, this comparison condition has two operator/value pairs:

```
{ "$lt" : 20000, "$gte" : 10000 }
```

Table 1–6 lists the operators that can appear in comparison conditions and the operands that they can have.

### Table 1–6 Operands and Operators in Comparison Conditions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operand</th>
</tr>
</thead>
</table>
| $exists | A scalar of any type. If the value of the scalar is 0, false, or null, then the condition is false; otherwise, it is true.  
Example:  
( "$exists" : true ), ( "$exists" : 1 ), ( "$exists" : null ) |
| $eq      | A scalar of any type.  
Example: ( "$eq" : "dog" ), ( "$eq" : false ) |
| $ne      | A scalar of any type.  
Example: ( "$eq" : "dog" ), ( "$eq" : false ) |
| $gt      | An JSON number or a JSON string.  
Example: ( "$gte" : 1234 ) |
| $lt      | An JSON number or a JSON string.  
Example: ( "$lte" : 1234 ) |
| $gte     | Value list—a JSON array of one or more scalar values.  
Example: ( "$in" : [ "cat", "dog", 123, true ] ) |
| $nin     | Value list—a JSON array of one or more scalar values.  
Example: ( "$in" : [ "cat", "dog", 123, true ] ) |
| $all     | Any JSON string.  
Example: ( "$all" : "cat" ) |
| $startsWith | Any string.  
Example: ( "$startsWith" : "cat" ) |
| $regex   | A comparison condition with a single operator/value pair.  
Example: ( "$not" : ( "$gte" : 1234 ) ) |

**Omitting $eq** In a $eq condition, you can usually omit the keyword $eq. For example, these filters are equivalent; both specify objects whose greeting field has the value "hello":

```
{ "greeting" : { "$eq" : "hello" } }
```

```
{ "greeting" : "hello" }
```

Downscoped Field Condition A downscoped field condition consists of the name of a parent field followed by a list of child field/value pairs.

For example, suppose that the parent field address has the child fields city and state. The following downscoped field condition tests whether the field address.city has the value "Boston" and address.state has the value "MA":

```
"address" : { "city" : "Boston", "state" : "MA" }
```

Suppose that you have this document:
The following query matches each path in the document independently; therefore, it matches the preceding document:

{ "address.city" : "Boston", "address.state" : "CA" }

However, the following downscoped field condition does not match the preceding document:

'address' : { "city" : "Boston", "state" : "CA" }

**Comparison Conditions and Arrays**

When a path that does not end in an array step uses a comparison condition, and the path targets an array, the comparison condition applies to each element of the array.

For example, the QBE { "animal" : { "$eq" : "cat" } } matches the JSON data { "animal" : [ "dog", "cat" ] }, even though "cat" is an array element (at any array index).

**Conjunction Clause**

A conjunction clause is a conjunction—$and, $or, or $nor—followed by an array of one or more filter conditions. For example:

'OR' [ { "name" : "Joe" }, { "salary" : 10000 } ]

**Omitting $and**

In an $and clause, you can usually omit the keyword $and. For example, these filters are equivalent; both specify objects whose salary fields have values greater than 10,000 and less than or equal to 20,000:

{ "$and" : [ { "salary" : { "$gt" : 10000 } }, { "salary" : { "$lte" : 20000 } } ] }

{ "salary" : { "$gt" : 10000, "$lte" : 20000 } }

**Note:** If you omit $and then make sure that no field in the resulting filter appears multiple times at the same level in the same object, as salary does here:

{ "salary" : { "$gt" : 10000 }, "salary" : { "$lt" : 20000 } }

If you do not respect this rule then only one of the conditions is evaluated; the other is ignored silently.

In an object structure, $and is implied; all conditions in the structure must be true.

**Example 1-2** is a filter specification in which $and is implied. The filter specifies objects for which the name starts with "Fred" and the salary is greater than 10,000 and less than or equal to 20,000 and either address.city is "Bedrock" or address.zipcode is 12345.
Example 1–2  Filter Specification

```json
{ "name" : { "$startsWith" : "Fred" },
"salary" : { "$gte" : 10000, "$lte" : 20000 },
"$or" : [ { "address.city" : "Bedrock" },
          { "address.zipcode" : 12345 } ],
"married" : true,
"$exists" : "children[2]" }
```

ID Clause

**Note:** An ID clause is global to the query and must be the outermost filter condition.

An ID clause is $id followed by either a scalar key (document identifier) or an array of scalar keys. The scalar key must be either an integer or a string. The array elements must be either all integers or all strings. For example:

```
"$id" : "USA"
"$id" : [1001,1002,1003]
```

**See Also:** *"Listing the Documents in a Collection"* on page 1-10

Omitting $id for Array of Keys  In an ID clause with an array of keys, you can omit the keyword $id. For example, these filters are equivalent; both specify objects with the same keys:

```
{ "$id" : [ "123456", "123457", "123458" ] }

[ "123456", "123457", "123458" ]
```

Collection Specifications

**Note:** In collection specifications, you must use strict JSON syntax. That is, you must enclose every nonnumeric value in double quotation marks. For information about strict and lax JSON syntax, see *Oracle XML DB Developer’s Guide*.

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.

Table 1–7 lists and describes the collection specification fields and their possible values, if any.

**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 1–7  Collection Specification Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>schemaName</td>
<td>SQL name of schema that owns table or view underlying collection object.</td>
<td></td>
</tr>
<tr>
<td>tableName or viewName</td>
<td>SQL name of table or view underlying collection object.</td>
<td></td>
</tr>
<tr>
<td>keyColumn.name</td>
<td>Name of key column.</td>
<td>Default: ID</td>
</tr>
<tr>
<td>keyColumn.sqlType</td>
<td>SQL data type of key column.</td>
<td>VARCHAR2 (default), NVARCHAR2, NUMBER, RAW</td>
</tr>
<tr>
<td>keyColumn.maxLength</td>
<td>Maximum length of key column, if not of NUMBER data type.</td>
<td>Default: 255</td>
</tr>
<tr>
<td>keyColumn.assignmentMethod</td>
<td>Key assignment method (see &quot;Key Assignment Method&quot; on page 1-38).</td>
<td>SEQUENCE, GUID, UUID (default), or CLIENT</td>
</tr>
<tr>
<td>keyColumn.sequenceName</td>
<td>If keyColumn.assignmentMethod is SEQUENCE, then this field must specify the name of a database sequence.</td>
<td>Name of existing database sequence</td>
</tr>
<tr>
<td>contentColumn.name</td>
<td>Name of content column.</td>
<td>Default: JSON_DOCUMENT</td>
</tr>
<tr>
<td>contentColumn.sqlType</td>
<td>SQL data type of content column.</td>
<td>VARCHAR2, NVARCHAR2, RAW, BLOB (default), CLOB, NCLOB</td>
</tr>
<tr>
<td>contentColumn.maxLength</td>
<td>Maximum length of content column, if not of LOB data type.</td>
<td>Default depends on both initialization parameter MAX_STRING_SIZE and keyColumn.sqlType.</td>
</tr>
<tr>
<td></td>
<td>If MAX_STRING_SIZE = STANDARD (default), then default maxLength is:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 4000 bytes if keyColumn.sqlType is VARCHAR2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 2000 bytes if keyColumn.sqlType is RAW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 2000 characters if keyColumn.sqlType is NVARCHAR2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If MAX_STRING_SIZE = EXTENDED, then default maxLength is:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 32767 bytes if keyColumn.sqlType is VARCHAR2 or RAW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ 16383 characters if keyColumn.sqlType is NVARCHAR2</td>
<td></td>
</tr>
</tbody>
</table>
**Table 1–7 (Cont.) Collection Specification Fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>contentColumn.validation</code></td>
<td>Validation level of content column. corresponds to SQL condition <code>is json</code>, which determines the syntax to which JSON content must conform.</td>
<td>STANDARD (default), STRICT, LAX</td>
</tr>
<tr>
<td></td>
<td><strong>STANDARD</strong> validates according to the JSON RFC 4627 standard. (It corresponds to the strict syntax defined for Oracle SQL condition <code>is json</code>.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>STRICT</strong> is the same as <strong>STANDARD</strong>, 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 <code>is json</code> when the keywords <strong>WITH UNIQUE KEYS</strong> are also used.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>LAX</strong> validates more loosely. (It corresponds to the lax syntax defined for Oracle SQL condition <code>is json</code>.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Some of the relaxations that LAX allows include the following:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- It does not require JSON field names to be enclosed in double quotation marks (<code>&quot;</code>).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- It allows uppercase, lowercase, and mixed case versions of <code>true</code>, <code>false</code>, and <code>null</code>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Numerals can be represented in additional ways.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>For more information about strict and lax syntax, see Oracle XML DB Developer’s Guide. For the JSON RFC 4627 standard, see</td>
<td></td>
</tr>
<tr>
<td><code>contentColumn.compress</code></td>
<td>Compression level for SecureFiles stored in content column. For information about SecureFiles LOB storage, see Oracle Database SecureFiles and Large Objects Developer’s Guide.</td>
<td>NONE (default), HIGH, MEDIUM, LOW</td>
</tr>
<tr>
<td><code>contentColumn.cache</code></td>
<td>Caching of SecureFiles stored in content column.</td>
<td>TRUE, FALSE (default)</td>
</tr>
<tr>
<td><code>contentColumn.encrypt</code></td>
<td>Encryption algorithm for SecureFiles stored in content column.</td>
<td>NONE (default), 3DES168, AES128, AES192, AES256</td>
</tr>
<tr>
<td><code>creationTimeColumn.name</code></td>
<td>Name of optional created-on timestamp column.</td>
<td>Default: CREATED_ON</td>
</tr>
<tr>
<td></td>
<td>This column has SQL data type <code>TIMESTAMP</code> and default value <code>SYSTIMESTAMP</code>.</td>
<td></td>
</tr>
<tr>
<td><code>lastModifiedColumn.name</code></td>
<td>Name of optional last-modified timestamp column.</td>
<td>Default: LAST_MODIFIED</td>
</tr>
<tr>
<td></td>
<td>This column has SQL data type <code>TIMESTAMP</code> and default value <code>SYSTIMESTAMP</code>.</td>
<td></td>
</tr>
<tr>
<td><code>lastModifiedColumn.index</code></td>
<td>Name of nonunique index on timestamp column. The index is created if a name is specified.</td>
<td></td>
</tr>
</tbody>
</table>
Example 1–3 is a collection specification for an object whose underlying table is HR.EMPLOYEES.

**Example 1–3  Collection Specification**

```json
{
  "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"
    }
}
```

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.
Key Assignment Method

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

### Table 1–8 Key Assignment Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQUENCE</td>
<td>Keys are integers generated by a database sequence. You must specify the name of the sequence in the keyColumn.sequenceName field.</td>
</tr>
<tr>
<td>GUID</td>
<td>Keys are generated by the SQL function <code>SYS_GUID()</code>, 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.</td>
</tr>
<tr>
<td>UUID</td>
<td>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.</td>
</tr>
<tr>
<td>CLIENT</td>
<td>Keys are assigned by the client application (not recommended).</td>
</tr>
</tbody>
</table>

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" on page 1-23. 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" on page 1-21. 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.

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>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MD5</strong></td>
<td>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 binary data types (such as BLOB or RAW), 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.</td>
</tr>
<tr>
<td><strong>SHA256</strong> (default)</td>
<td>The REST server computes an 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 binary data types (such as BLOB or RAW), 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.</td>
</tr>
<tr>
<td><strong>UUID</strong></td>
<td>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).</td>
</tr>
<tr>
<td><strong>TIMESTAMP</strong></td>
<td>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).</td>
</tr>
<tr>
<td><strong>SEQUENTIAL</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>NONE</strong></td>
<td>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).</td>
</tr>
</tbody>
</table>

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.

### Security

ORDS uses role-based access control to secure services. 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:

```sql
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:

```sql
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.

### 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

**Development and Testing**

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

```java
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`):

```java
# 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/
```
A
allowed characters, definition, 1-27
array step, definition, 1-27

B
backquotes, 1-27

C
collections
   creating, 1-5
deleting, 1-6
deleting documents from, 1-8
   listing, 1-6
   listing documents in, 1-10
   specifications for, 1-34

D
database role
   SODA_APP, 1-40
   DELETE collection operation, 1-22
   DELETE object operation, 1-22
   deleting collections, 1-6
deleting documents from collections, 1-8
documents
deleting from collections, 1-8
   filtering in collections, 1-13
   inserting into collections
      in bulk from JSON array, 1-9
      one at a time, 1-7
   listing in collections, 1-10
   retrieving from collections, 1-8
   updating in collections, 1-12

F
field step, definition, 1-27
filter specifications
details, 1-30
   overview, 1-28
filtering documents in collections, 1-13

G
GET collection operation, 1-17
GET object operation, 1-20
GET schema operation, 1-17

I
inserting documents into collections
   in bulk from JSON array, 1-9
   one at a time, 1-7

K
key assignment method, 1-38

L
listing collections in schema, 1-6
listing documents in collections, 1-10

O
Oracle REST API, 1-1
Oracle REST API HTTP operations
   response bodies of, 1-15
   summary of, 1-14
   URI form for, 1-15

P
paths, 1-26
POST array insert operation, 1-24
POST bulk delete operation, 1-25
POST object operation, 1-23
POST query operation, 1-24
PUT collection operation, 1-21
PUT object operation, 1-21

Q
query-by-example (QBE)
definition, 1-28
table QBE expressions, 1-13
table query-by-example, 1-13

Index-1
REST architectural style, 1-2
retrieving documents from collections, 1-8

security, 1-40
SODA_APP database role, 1-40
specifications
  collection, 1-34
  filter
details, 1-30
  overview, 1-28
syntactic characters, definition, 1-26

updating documents in collections, 1-12

versioning method, 1-38