Oracle® REST Data Services

SODA for REST Developer's Guide

Release 18.2
E85825-01
June 2018

Oracle REST Data Services SODA for REST Developer's Guide, Release 18.2

E85825-01

Primary Author: Drew Adams

Contributing Authors: Sheila Moore

Contributors: Douglas McMahon, Maxim Orgiyan, Josh Spiegel

Contents

Preface

Audience ix
Documentation Accessibility ix
Related Documents ix
Conventions ix

1 SODA for REST Overview

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

2 Installing SODA for REST

3 Using SODA for REST

3.1 Creating a Document Collection with SODA for REST 3-2
3.2 Discovering Existing Collections with SODA for REST 3-3
3.3 Dropping a Document Collection with SODA for REST 3-4
3.4 Inserting a Single Document into a Collection with SODA for REST 3-5
3.5 Inserting Multiple Documents into a Collection with SODA for REST 3-6
3.6 Finding Documents in Collections with SODA for REST 3-7
3.7 Replacing Documents in a Collection with SODA for REST 3-8
3.8 Removing a Single Document from a Collection with SODA for REST 3-9
3.9 Removing Multiple Documents from a Collection with SODA for REST 3-10
3.10 Listing the Documents in a Collection with SODA for REST 3-11
3.11 Indexing the Documents in a Collection with SODA for REST 3-13
3.12 Querying Using a Filter Specification with SODA for REST 3-14
3.12.1 QBE.1.json 3-15
3.12.2 QBE.2.json 3-15
3.12.3 QBE.3.json 3-16
3.12.4 QBE.4.json 3-16
3.13 Patching a Single JSON Document with SODA for REST 3-16

3.14 Patching Multiple JSON Documents in a Collection with SODA for REST 3-18

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 catalog 4-5
4.3.1 URL Pattern for GET catalog 4-5
4.3.2 Response Codes for GET catalog 4-6
4.4 GET user collections 4-7
4.4.1 URL Pattern for GET user collections 4-7
4.4.2 Response Codes for GET user collections 4-7
4.5 GET JSON schema for collection 4-8
4.5.1 URL Pattern for GET JSON schema for collection 4-8
4.5.2 Response Codes for GET JSON schema for collection 4-8
4.6 GET actions 4-10
4.6.1 URL Pattern for GET actions 4-11
4.7 GET collection 4-11
4.7.1 URL Pattern for GET collection 4-11
4.7.2 Response Codes for GET collection 4-12
4.7.3 Links Array for GET collection 4-13
4.8 GET object 4-14
4.8.1 URL Pattern for GET object 4-14
4.8.2 Request Headers for GET object 4-14
4.8.3 Response Codes for GET object 4-15
4.9 DELETE collection 4-15
4.9.1 URL Pattern for DELETE collection 4-16
4.9.2 Response Codes for DELETE collection 4-16
4.10 DELETE object 4-16
4.10.1 URL Pattern for DELETE object 4-17
4.10.2 Response Codes for DELETE object 4-17
4.11 PATCH JSON document 4-17
4.11.1 URL Pattern for PATCH JSON document 4-18
4.11.2 Request Headers for PATCH JSON document 4-18
4.11.3 Request Body for PATCH JSON document 4-18
4.11.4 Response Codes for PATCH JSON Document 4-19
4.12 POST object 4-19
4.12.1 URL Pattern for POST object 4-20
4.12.2 Request Body for POST object 4-20
4.12.3 Response Codes for POST object 4-20
4.13 POST query 4-21

4.13.1 URL Pattern for POST query 4-21
4.13.2 Request Body for POST query 4-22
4.13.3 Response Codes for POST query 4-22
4.14 POST bulk insert 4-22
4.14.1 URL Pattern for POST bulk insert 4-23
4.14.2 Request Body for POST bulk insert 4-23
4.14.3 Response Codes for POST bulk insert 4-23
4.15 POST bulk delete 4-24
4.15.1 URL Pattern for POST bulk delete 4-24
4.15.2 Request Body for POST bulk delete (Optional) 4-25
4.15.3 Response Codes for POST bulk delete 4-25
4.16 POST bulk update (patch) 4-26
4.16.1 URL Pattern for POST bulk update (patch) 4-26
4.16.2 Request Body for POST bulk update (patch) 4-27
4.16.3 Response Codes for POST bulk update (patch) 4-27
4.17 POST index 4-27
4.17.1 URL Pattern for POST index 4-28
4.17.2 Request Body for POST index 4-28
4.17.3 Response Codes for POST index 4-29
4.18 POST unindex 4-29
4.18.1 URL Pattern for POST unindex 4-29
4.18.2 Request Body for POST unindex 4-30
4.18.3 Response Codes for POST unindex 4-30
4.19 PUT collection 4-30
4.19.1 URL Pattern for PUT collection 4-31
4.19.2 Request Body for PUT collection (Optional) 4-31
4.19.3 Response Codes for PUT collection 4-31
4.20 PUT object 4-31
4.20.1 URL Pattern for PUT object 4-32
4.20.2 Request Body for PUT object 4-32
4.20.3 Response Codes for PUT object 4-32

5 Collection Specifications

5.1 Key Assignment Method 5-4
5.2 Versioning Method 5-6

6 Security

6.1 Authentication Mechanisms 6-2

6.2 Security Considerations for Development and Testing 6-2

Index

List of Examples

3-1	Bulk-Inserting Documents into a Collection Using a JSON Array of Objects	3-6
3-2	Checking an Inserted Document	3-7
3-3	Bulk-Removing Matching Documents from a Collection	3-10
3-4	Bulk-Removing All Documents from a Collection	3-10
3-5	Indexing a JSON Field with SODA for REST	3-13
3-6	B-Tree Index Specification for Field Requestor (file indexSpec1.json)	3-13
3-7	JSON Patch Specification (File poPatchSpec.json)	3-17
3-8	JSON Document Before Patching	3-17
3-9	JSON Document After Patching	3-18
3-10	QBE for Patching Multiple JSON Documents Using QBE Operator $patch	3-19
3-11	Patching Multiple JSON Documents Using HTTP POST with patch Action	3-19
4-1	Response Body	4-4
5-1	Default Collection Metadata	5-3

List of Tables

4-1	Fields That Can Appear in Response Bodies	4-3
4-2	Additional Response Body Fields for Operations that Return Objects	4-4
4-3	Relationship of GET collection Parameters to Mode and Links Array	4-13
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
- Documentation Accessibility
- Related Documents
- Conventions

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

For more information, see these Oracle resources:

- *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for general information about SODA
- *Oracle as a Document Store* for general information about using JSON data in Oracle Database, including with SODA
- *Oracle Database JSON Developer's Guide* for information about using SQL and PL/SQL with JSON data

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.

In the context of SODA for REST, a **document** in a collection is sometimes called an **object**. Typically it is a *JSON* document, but it can instead be a Multipurpose Internet Mail Extensions (MIME) type — image, audio, or video, for example. An application often uses a given collection to hold instances of a particular type of document. A SODA collection is thus roughly analogous to a table in a relational database: one database column stores document keys, and another column stores document content.

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

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

The remaining topics of this document describe various features of SODA for REST.

> **Note:**
>
> This book provides information about using SODA with REST applications. To use SODA for REST you also need to understand SODA generally. For such general information, please consult *Oracle Database Introduction to Simple Oracle Document Access (SODA)*.

- 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.

Related Topics

- SODA for REST HTTP Operations
  The SODA for REST HTTP operations are described.

> **See Also:**
>
> - *Oracle as a Document Store* for general information about using JSON data in Oracle Database, including with 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 JSON Developer's Guide* for information about using SQL and PL/SQL with JSON data stored in Oracle Database

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

2
Installing SODA for REST

Complete instructions are provided for installing SODA for REST.

1. Ensure that you have one of the following Oracle Database releases installed:
   - Oracle Database 12c Release 2 (12.2) or later
   - Oracle Database 12c Release 1 (12.1.0.2) with Merge Label Request (MLR) bundle patch 20885778 (patch 20885778 obsoletes patch 20080249)

     Obtain this patch from My Oracle Support (My Oracle Support). 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.
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.
5. Connect to the database schema (user account) that you want ORDS to access.
6. Enable ORDS in that database schema by executing this SQL command:

   ```
   EXEC ords.enable_schema;
   COMMIT;
   ```
7. Grant role `SODA_APP` to the database schema (user account) *`database-schema`* that you enabled in step 6:

   ```
   GRANT SODA_APP TO database-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;
```

> **Note:**
>
> This enables *anonymous* access to the service, which is *not* recommended for production systems.

b. Start ORDS in standalone mode:

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

> **Note:**
>
> Running ORDS in standalone mode is *not* recommended for production systems.

9. In a web browser, open:

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

Where *database-schema* is the lowercase name of the database schema in which you enabled ORDS in step 6. If the installation succeeded, you see:

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

Related Topics

- 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.

> **See Also:**
>
> - *Oracle REST Data Services Installation, Configuration, and Development Guide* for complete information about installing and configuring ORDS
> - *Oracle REST Data Services Installation, Configuration, and Development Guide* for instructions about downloading and extracting the ORDS zip archive
> - *Oracle REST Data Services Installation, Configuration, and Development Guide* for information about configuring ORDS
> - *Oracle REST Data Services Installation, Configuration, and Development Guide* for information about starting ORDS in standalone mode

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 Oracle REST Data Services (ORDS) as instructed in Installing SODA for REST, enabling ORDS in *database-schema*.

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

- Creating a Document Collection with SODA for REST
  How to use SODA for REST to create a new document collection is explained.
- Discovering Existing Collections with SODA for REST
  An example is given of listing the existing collections.
- Dropping a Document Collection with SODA for REST
  An example is given of dropping a collection.
- Inserting a Single Document into a Collection with SODA for REST
  An example is given of inserting a document into a collection.
- Inserting Multiple Documents into a Collection with SODA for REST
  You can bulk-insert a set of documents into a collection using a JSON array of objects. Each object corresponds to the content of one of the inserted documents.
- Finding Documents in Collections with SODA for REST
  An example is given of retrieving a document from a collection by providing its key.
- Replacing Documents in a Collection with SODA for REST
  An example is given of replacing a document in a collection with a newer version. For this, you use HTTP operation `PUT`.
- Removing a Single Document from a Collection with SODA for REST
  You can use HTTP operation `DELETE` to remove a single document from a collection.
- Removing Multiple Documents from a Collection with SODA for REST
  You can remove multiple JSON documents from a collection with HTTP operation `POST`, using custom-action `delete` or `truncate` in the request URL. Use `truncate` to remove *all* JSON documents from the collection. Use `delete` together with a QBE to delete only the documents that match that filter.
- Listing the Documents in a Collection with SODA for REST
  An example is given of listing the documents in a collection, using a `GET` operation.
- Indexing the Documents in a Collection with SODA for REST
  You can index the documents in a collection with HTTP operation `POST`, using custom-action `index` in the request URL. The request body contains an index specification. It can specify B-tree, spatial, full-text, or data-guide indexing.

- Querying Using a Filter Specification with SODA for REST
  Examples are given of using a filter specification, or query-by-example (QBE), to define query criteria for selecting documents from a collection.
- Patching a Single JSON Document with SODA for REST
  You can selectively update (patch) parts of a single JSON document using HTTP operation `PATCH`. You specify the update using a JSON Patch specification.
- Patching Multiple JSON Documents in a Collection with SODA for REST
  You can update (patch) multiple JSON documents in a collection by querying the collection to match those documents and specifying the changes to be made. You specify the update with a JSON Patch specification, using QBE operator `$patch`. You use HTTP operation `POST` with custom-action `update` in the request URL.

See Also:

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

3.1 Creating a Document Collection with SODA for REST

How to use SODA for REST to create a new document collection is explained.

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

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

The preceding command sends a `PUT` request with URL `http://localhost:8080/ords/database-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/database-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. The table name was defaulted from the collection name. In this case, the name is mixed-case, so double quotation marks are needed around it. To create a custom collection configuration, provide a collection specification as the body of the `PUT` operation.

If a collection with the same name already exists then it is simply opened. If custom metadata is provided and it does not match the metadata of the existing collection then the collection is not opened and an error is raised. (To match, all metadata fields must have the same values.)

> **Caution:**
>
> To drop a collection, proceed as described in Dropping a Document Collection with SODA for REST. 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.

Related Topics

- Collection Specifications
  A collection specification is a JSON object that provides information about the Oracle Database table or view underlying a collection object. The table or view is created when you create the collection.
- PUT collection
  `PUT` collection creates a collection if it does not exist.
- Discovering Existing Collections with SODA for REST
  An example is given of listing the existing collections.

> **See Also:**
>
> *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for information about the default naming of a collection table

3.2 Discovering Existing Collections with SODA for REST

An example is given of listing the existing collections.

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

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

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

```
{ "items" :
  [ { "name":"MyCollection",
      "properties": { "schemaName":"SCHEMA",
                      "tableName":"MyCollection",
                      ... }
      "links" :
```

```
    [ { "rel"  : "canonical",
        "href" :
        "http://localhost:8080/ords/database-schema/soda/latest/MyCollection" } ] } ],
  "more" : false }
```

The response body includes all available collections in *database-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.

Related Topics

- GET user collections
  `GET` user collections gets all or a subset of the collection names for a given database schema (user account).
- Collection Specifications
  A collection specification is a JSON object that provides information about the Oracle Database table or view underlying a collection object. The table or view is created when you create the collection.
- Dropping a Document Collection with SODA for REST
  An example is given of dropping a collection.

3.3 Dropping a Document Collection with SODA for REST

An example is given of dropping a collection.

To delete `MyCollection`, run this command:

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

The preceding command sends a `DELETE` request with the URL `http://localhost:8080/ords/database-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 *database-schema*:

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

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

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

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

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

Related Topics

- DELETE collection
  `DELETE` collection deletes a collection.
- Inserting a Single Document into a Collection with SODA for REST
  An example is given of inserting a document into a collection.

3.4 Inserting a Single Document into a Collection with SODA for REST

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/database-schema/soda/latest/MyCollection
```

The preceding command sends a `POST` request with the URL `http://localhost:8080/ords/database-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.

Related Topics

- POST object
  `POST` object inserts an uploaded object into a specified collection, assigning and returning its key. The collection must use server-assigned keys.
- Finding Documents in Collections with SODA for REST
  An example is given of retrieving a document from a collection by providing its key.
- Inserting Multiple Documents into a Collection with SODA for REST
  You can bulk-insert a set of documents into a collection using a JSON array of objects. Each object corresponds to the content of one of the inserted documents.

3.5 Inserting Multiple Documents into a Collection with SODA for REST

You can bulk-insert a set of documents into a collection using a JSON array of objects. Each object corresponds to the content of one of the inserted documents.

Example 3-1 inserts a JSON array of purchase-order objects into a collection as a set of documents, each object constituting the content of one document. Example 3-2 checks an inserted document.

A successful `POST` bulk-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.

Example 3-1 Bulk-Inserting Documents into a Collection Using a JSON Array of Objects

This example uses file `POList.json`, which is included in the download. The file contains a JSON array of purchase-order objects. This command loads the purchase orders into collection `MyCollection` as documents.

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

Action `insert` causes the array to be inserted as a set of documents, rather than as a single document.

(You can alternatively use the equivalent URL `http://localhost:8080/ords/database-schema/soda/latest/MyCollection?action=insert`.)

The command sends a `POST` request with the URL `http://localhost:8080/ords/database-schema/soda/latest/MyCollection`. 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
}
```

Example 3-2 Checking an Inserted Document

You can check an inserted document by copying an `id` field value returned by your own `POST` bulk-insert operation (not a value from Example 3-1) and querying the collection for a document that has that `id` value. Using SQL*Plus or SQL Developer, substitute your copied value for placeholder *`identifier`* here:

```
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 here, 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 SQL/JSON function `json_value`. You cannot use the simplified, dot-notation JSON syntax.

Related Topics

- POST bulk insert
  `POST` bulk insert inserts an array of objects into a specified collection, assigning and returning their keys.
- Listing the Documents in a Collection with SODA for REST
  An example is given of listing the documents in a collection, using a `GET` operation.
- Inserting a Single Document into a Collection with SODA for REST
  An example is given of inserting a document into a collection.

> **See Also:**
>
> *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for information about the default naming of a collection table

3.6 Finding Documents in Collections with SODA for REST

An example is given of retrieving a document from a collection by providing its key.

To retrieve the document that was inserted in Inserting a Single Document into a Collection with SODA for REST, run this command, where *`id`* is the document key that you copied when inserting the document:

```
curl -X GET http://localhost:8080/ords/database-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/database-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"
}
```

Related Topics

- GET object
  `GET` object gets a specified object from a specified collection.
- Removing a Single Document from a Collection with SODA for REST
  You can use HTTP operation `DELETE` to remove a single document from a collection.
- Inserting a Single Document into a Collection with SODA for REST
  An example is given of inserting a document into a collection.

3.7 Replacing Documents in a Collection with SODA for REST

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

The behavior of operation `PUT` 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 with SODA for REST:

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

The preceding command outputs the retrieved document.

To replace 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/database-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/database-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 that the document has been updated, rerun this command:

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

The preceding command returns:

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

Related Topics

- 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.
- Key Assignment Method
  The key assignment method determines how keys are assigned to objects that are inserted into a collection.
- Querying Using a Filter Specification with SODA for REST
  Examples are given of using a filter specification, or query-by-example (QBE), to define query criteria for selecting documents from a collection.
- Listing the Documents in a Collection with SODA for REST
  An example is given of listing the documents in a collection, using a `GET` operation.

3.8 Removing a Single Document from a Collection with SODA for REST

You can use HTTP operation `DELETE` to remove a single document from a collection.

To remove, from `MyCollection`, the document that you retrieved in Finding Documents in Collections with SODA for REST, run this command (where *id* is the document identifier):

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

The preceding command sends a `DELETE` request with URL `http://localhost:8080/ords/database-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 removal of an object from a collection—a `DELETE object` operation—returns no response body.

Related Topics

- DELETE object
  `DELETE` object deletes a specified object from a specified collection.
- Removing Multiple Documents from a Collection with SODA for REST
  You can remove multiple JSON documents from a collection with HTTP operation `POST`, using custom-action `delete` or `truncate` in the request URL. Use `truncate` to remove *all* JSON documents from the collection. Use `delete` together with a QBE to delete only the documents that match that filter.

3.9 Removing Multiple Documents from a Collection with SODA for REST

You can remove multiple JSON documents from a collection with HTTP operation `POST`, using custom-action `delete` or `truncate` in the request URL. Use `truncate` to remove *all* JSON documents from the collection. Use `delete` together with a QBE to delete only the documents that match that filter.

Example 3-3 removes the documents where `User` field has value `TGATES` from collection `MyCollection`.Example 3-4 removes all documents from collection `MyCollection`.

Example 3-3 Bulk-Removing Matching Documents from a Collection

This example uses the QBE that is in file QBE.1.json to match the nine documents that have `"TGATES"` as the value of field `User`. It removes (only) those documents from collection `MyCollection`.

```
curl -X POST --data-binary @QBE.1.json -H "Content-Type: application/json"
http://localhost:8080/ords/database-schema/soda/latest/custom-actions/delete/
MyCollection/
```

(You can alternatively use the equivalent URL `http://localhost:8080/ords/database-schema/soda/latest/MyCollection?action=delete`.)

> **WARNING:**
>
> If you specify `delete` as the action, and you use the empty object, `{}`, as the filter specification, then the operation deletes *all* objects from the collection.

Example 3-4 Bulk-Removing All Documents from a Collection

This example removes all documents from collection `MyCollection`.

```
curl -X POST -H "Content-Type: application/json"
http://localhost:8080/ords/database-schema/soda/latest/custom-actions/truncate/
MyCollection/
```

(You can alternatively use the equivalent URL `http://localhost:8080/ords/database-schema/soda/latest/MyCollection?action=truncate`.)

3.10 Listing the Documents in a Collection with SODA for REST

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/database-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/database-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/database-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.

Related Topics

- 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.
- Replacing Documents in a Collection with SODA for REST
  An example is given of replacing a document in a collection with a newer version. For this, you use HTTP operation `PUT`.
- Querying Using a Filter Specification with SODA for REST
  Examples are given of using a filter specification, or query-by-example (QBE), to define query criteria for selecting documents from a collection.

> **See Also:**
>
> *Oracle REST Data Services Installation, Configuration, and Development Guide* for information about configuration parameters `soda.maxLimit` and `soda.defaultLimit`

3.11 Indexing the Documents in a Collection with SODA for REST

You can index the documents in a collection with HTTP operation `POST`, using custom-action `index` in the request URL. The request body contains an index specification. It can specify B-tree, spatial, full-text, or data-guide indexing.

> **Note:**
>
> To create an index with SODA you need Oracle Database Release 12c (12.2.0.1) or later. But to create a B-tree index that for a `DATE` or `TIMESTAMP` value you need Oracle Database Release 18c (18.1) or later.

A JSON search index is used for full-text search and ad hoc structural queries, and for persistent recording and automatic updating of JSON data-guide information. An Oracle Spatial and Graph index is used for GeoJSON (spatial) data.

> **See Also:**
>
> - *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for an overview of using SODA indexing
> - *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for information about SODA index specifications
> - *Oracle Database JSON Developer's Guide* for information about JSON search indexes
> - *Oracle Database JSON Developer's Guide* for information about persistent data-guide information as part of a JSON search index

Example 3-5 Indexing a JSON Field with SODA for REST

This example indexes the documents in collection `MyCollection` according to the index specification in file `indexSpec1.json` (see Example 3-6).

```
curl -i -X POST --data-binary @indexSpec1.json -H "Content-Type: application/json"
http://localhost:8080/ords/database-schema/soda/latest/custom-actions/index/
MyCollection/
```

This request, using the alternative URI syntax, is equivalent:

```
curl -i -X POST --data-binary @indexSpec1.json -H "Content-Type: application/json"
http://localhost:8080/ords/database-schema/soda/latest/MyCollection?action=index
```

Example 3-6 B-Tree Index Specification for Field Requestor (file indexSpec1.json)

This example shows the B-tree index specification in file `indexSpec1.json`.

The index is named `REQUESTOR_IDX`, and it indexes field `Requestor`. The data type is not specified, so it is `VARCHAR2`, the default. Because field `scalarRequired` is specified as `true`, if the collection contains a document that lacks the indexed field then an error is raised when the index creation is attempted.

```
{ "name"           : "REQUESTOR_IDX",
  "scalarRequired" : true,
  "fields"         : [{"path" : "Requestor", "order" : "asc"}] }
```

3.12 Querying Using a Filter Specification with SODA for REST

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. They are in directory `ORDS_HOME/examples/soda/getting-started`.

- 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`.
- 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.
- QBE.3.json
  The query-by-example (QBE) in file `QBE.3.json` selects documents where the value of field `ItemNumber`, in an element of array `LineItems`, is greater than 4. QBE operator field `"$gt"` is required.
- QBE.4.json
  The query-by-example (QBE) in file `QBE.4.json` selects documents where the value of field `UPCCode` equals `"13023015692"` *and* the value of field `ItemNumber` equals `3`. QBE operator field `$and` is optional.

Related Topics

- POST query
  `POST` query gets all or a subset of objects from a collection, using a filter.

> **See Also:**
>
> - *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for an overview of filter specifications and QBE
> - *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for reference information about filter specifications and QBE

3.12.1 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/database-schema/soda/latest/custom-actions/query/
MyCollection/
```

(You can alternatively use the equivalent URL `http://localhost:8080/ords/database-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.

3.12.2 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`. It has an implied use of operator field `"$eq"`.

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

[1] An equivalent composite-filter QBE explicitly uses QBE operator `$query`: `{ $query : { "User" : "TGATES" } }`.
[1] An equivalent composite-filter QBE explicitly uses QBE operator `$query`: `{ $query : { "User" : "TGATES" } }`.

> **See Also:**
>
> *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for more information

3.12.3 QBE.3.json

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

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

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

3.12.4 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`. QBE operator field `$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)*

3.13 Patching a Single JSON Document with SODA for REST

You can selectively update (patch) parts of a single JSON document using HTTP operation `PATCH`. You specify the update using a JSON Patch specification.

> **Note:**
>
> To use operation HTTP operation `PATCH` you need Oracle Database Release 18c or later.

JSON Patch is a format for specifying a sequence of operations to apply to a JSON document. It is identified by media type `application/json-patch+json`, and it is suitable for use with HTTP operation `PATCH`.

Use the QBE that is in file `QBE.5.json` to retrieve the single document from `MyCollection` that has field `PONumber` with a value of `1`:

```
curl -X POST --data-binary @QBE.5.json -H "Content-Type: application/json" http://localhost:8080/ords/database-schema/soda/latest/custom-actions/query/MyCollection/
```

This is the content of file `QBE.5.json`: `{ "PONumber" : 1 }`.

The preceding command outputs the retrieved document.

To update that document according to the JSON Patch specification in file `poPatchSpec.json` (see Example 3-7), execute this command, where *key* is the key of the document returned by the preceding command (`POST` operation for the QBE in file `QBE.5.json`).

```
curl -i -X PATCH --data-binary @poPatchSpec.json -H "Content-Type: application/json-patch+json"
http://localhost:8080/ords/database-schema/soda/latest/MyCollection/key
```

If successful, the preceding command returns a 200 HTTP status code.

If unsuccessful, patching is not performed. In particular, if any step (any operation) fails then patching of that document fails. The document is unchanged from its state before attempting the `PATCH` HTTP operation.

Example 3-8 shows an example document before successful patching with Example 3-7, and Example 3-9 shows the same document after patching (changes are indicated in **bold** type).

> **See Also:**
>
> JSON Patch (RFC 6902) for information about the JSON Patch format for describing changes to a JSON document

Example 3-7 JSON Patch Specification (File poPatchSpec.json)

```
[ { "op"    : "test",
    "path"  : "/ShippingInstructions/Address/street",
    "value" : "200 Sporting Green" },
  { "op"    : "replace",
    "path"  : "/ShippingInstructions/Address/street",
    "value" : "Winchester House, Heatley Rd" },
  { "op"    : "copy",
    "from"  : "/ShippingInstructions/Phone/0",
    "path"  : "/ShippingInstructions/Phone/1" },
  { "op"    : "replace",
    "path"  : "/ShippingInstructions/Phone/1/number",
    "value" : "861-555-8765" } ]
```

Example 3-8 JSON Document Before Patching

```
{ "PONumber"             : 1,
  "Reference"            : "MSULLIVA-20141102",
  "Requestor"            : "Martha Sullivan",
  "User"                 : "MSULLIVA",
  "CostCenter"           : "A50",
  "ShippingInstructions" : {
    "name"    : "Martha Sullivan",
    "Address" : { "street"  : "200 Sporting Green",
```

```
                  "city"    : "South San Francisco",
                  "state"   : "CA",
                  "zipCode" : 99236,
                  "country" : "United States of America" },
    "Phone"   : [ { "type"   : "Office",
                    "number" : "979-555-6598" } ] }
  ... }
```

Example 3-9 JSON Document After Patching

```
{ "PONumber"             : 1,
  "Reference"            : "MSULLIVA-20141102",
  "Requestor"            : "Martha Sullivan",
  "User"                 : "MSULLIVA",
  "CostCenter"           : "A50",
  "ShippingInstructions" : {
    "name"    : "Martha Sullivan",
    "Address" : { "city": "South San Francisco",
                  "state": "CA",
                  "zipCode": 99236,
                  "country": "United States of America",
                  "street": "Winchester House, Heatley Rd" },
    "Phone"   : [ { "type"   : "Office",
                    "number" : "979-555-6598" },
                  { "type": "Office",
                    "number": "861-555-8765" } ] }
  ... }
```

Related Topics

- PATCH JSON document
  `PATCH` JSON document replaces a specified object with an patched (edited) copy of it.
- Patching Multiple JSON Documents in a Collection with SODA for REST
  You can update (patch) multiple JSON documents in a collection by querying the collection to match those documents and specifying the changes to be made. You specify the update with a JSON Patch specification, using QBE operator `$patch`. You use HTTP operation `POST` with custom-action `update` in the request URL.

3.14 Patching Multiple JSON Documents in a Collection with SODA for REST

You can update (patch) multiple JSON documents in a collection by querying the collection to match those documents and specifying the changes to be made. You specify the update with a JSON Patch specification, using QBE operator `$patch`. You use HTTP operation `POST` with custom-action `update` in the request URL.

> **Note:**
>
> To use QBE operator `$patch` you need Oracle Database Release 18c or later.

Operator **`$patch`** is specific to SODA for REST; it is not used by other SODA implementations. It is used in a composite filter, at the same level as `$query` and `$orderby`. (If operators `$patch` and `$orderby` are both present in a composite filter then `$orderby` is ignored.)

The operand of operator `$patch` is a JSON Patch specification: a JSON array with object elements that list the patch operations to apply to each document targeted by the query.

JSON Patch is a format for specifying a sequence of operations to apply to a JSON document. It is identified by media type `application/json-patch+json`, and it is suitable for use with HTTP operation `PATCH`.

If any update step (any operation) specified for patching is unsuccessful for a given document then no patching is performed on that document. Patching continues for other targeted documents, however.

Example 3-10 shows a QBE for patching documents where `User` field has value `TGATES`. Example 3-11 shows a command that uses that QBE to perform the update operation.

> **See Also:**
>
> - *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for information about composite filter specifications
> - JSON Patch (RFC 6902) for information about the JSON Patch format for describing changes to a JSON document

Example 3-10 QBE for Patching Multiple JSON Documents Using QBE Operator $patch

This example shows the QBE that is the content of file `qbePatch.json` in the download. The QBE matches the same documents as QBE.1.json. It updates the street address and the first phone number in each document, using the same new values for each document.

Because operator `$patch` is used, the query part of the QBE must be specified using operator `$query`. The value of operator `$patch` is a JSON Patch specification. It replaces street address `"200 Sporting Green"` with `"176 Gateway Blvd"` and the first number in array `Phone` with `999-999-9999`.

```
{ "$query" : {"User" : "TGATES" },
  "$patch" : [ { "op"    : "test",
                 "path"  : "/ShippingInstructions/Address/street",
                 "value" : "200 Sporting Green" },
               { "op"    : "replace",
                 "path"  : "/ShippingInstructions/Address/street",
                 "value" : "176 Gateway Blvd" },
               { "op"    : "replace",
                 "path"  : "/ShippingInstructions/Phone/0/number",
                 "value" : "999-999-9999" } ] }
```

Example 3-11 Patching Multiple JSON Documents Using HTTP POST with patch Action

This command updates documents according to the QBE of Example 3-10. Each document matching the `$query` value is updated.

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

Related Topics

- POST bulk update (patch)
  The `POST` bulk update operation updates (patches) the objects of a specified collection.
- Patching a Single JSON Document with SODA for REST
  You can selectively update (patch) parts of a single JSON document using HTTP operation `PATCH`. You specify the update using a JSON Patch specification.

4
SODA for REST HTTP Operations

The SODA for REST HTTP operations are described.

- SODA for REST HTTP Operation URIs
  A SODA for REST HTTP operation is specified by a Universal Resource Identifier (URI).
- 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.
- GET catalog
  `GET` catalog gets all of the collection names for a given database schema (user account), along with information about each collection.
- GET user collections
  `GET` user collections gets all or a subset of the collection names for a given database schema (user account).
- GET JSON schema for collection
  This operation gets a JSON schema that describes the structure and type information of the JSON documents in a given collection.
- GET actions
  `GET` actions gets all of the available custom actions.
- 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.
- GET object
  `GET` object gets a specified object from a specified collection.
- DELETE collection
  `DELETE` collection deletes a collection.
- DELETE object
  `DELETE` object deletes a specified object from a specified collection.
- PATCH JSON document
  `PATCH` JSON document replaces a specified object with an patched (edited) copy of it.
- POST object
  `POST` object inserts an uploaded object into a specified collection, assigning and returning its key. The collection must use server-assigned keys.
- POST query
  `POST` query gets all or a subset of objects from a collection, using a filter.
- POST bulk insert
  `POST` bulk insert inserts an array of objects into a specified collection, assigning and returning their keys.

- POST bulk delete
  `POST` bulk delete deletes all or a subset of objects from a specified collection, using a filter to specify the subset.
- POST bulk update (patch)
  The `POST` bulk update operation updates (patches) the objects of a specified collection.
- POST index
  `POST` index creates indexes on the documents in a specified collection.
- POST unindex
  `POST` unindex deletes indexes on objects in a specified collection.
- PUT collection
  `PUT` collection creates a collection if it does not exist.
- 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.

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 any of these forms:

`/ords/`*`database-schema`*`/soda/[`*`version`*`/[metadata-catalog/[`*`collection`*`]]]`

`/ords/`*`database-schema`*`/soda/[`*`version`*`/[custom-actions/`*`action`*`/[`*`collection`*`/[`*`key`*`]]]]`

`/ords/`*`database-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.
- *`database-schema`* is the name of an Oracle Database schema (user account) 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`.
- `custom-actions` is the name for the set of possible SODA actions.
- `metadata-catalog` is the name for the catalog of SODA collections.
- *`collection`* is the name of a collection (set) of objects stored in *`database-schema`*.
- *`key`* is a string that uniquely identifies (specifies) an object in *`collection`*.
- *`action`* is either `query`, `index`, `unindex`, `insert`, `update`, `delete`, or `truncate`.

> **Note:**
>
> In the SODA for REST URI syntax, after the version component, you can use `custom-actions`, `metadata-catalog`, *or* a particular collection name. When you use `custom-actions` or `metadata-catalog`, the *next* segment in the URI, if there is one, is a collection name.
>
> Because of this syntax flexibility, you *cannot* have a collection named either `custom-actions` or `metadata-catalog`. An error is raised if you try to create a collection with either of those names using SODA for REST.
>
> In other SODA implementations, besides SODA for REST, nothing prevents you from creating and using a collection named `custom-actions` or `metadata-catalog`. But for possible interoperability, best practice calls for not using these names for collections.

These two syntax possibilities are equivalent:

`/ords/`*`database-schema`*`/soda/`*`version`*`/`**`custom-actions`**`/`***`action`***`/`*`collection`*`/`

`/ords/`*`database-schema`*`/soda/`*`version`*`/`*`collection`*`/`**`?action=`*****`action`**

Actions can only be used with a `POST` HTTP operation. (This applies to both URI syntaxes for performing actions.)

For some SODA for REST operations the path component of the URI syntax can be followed by an optional *query* component, which is preceded by a question mark (`?`). The query component is composed of one or more *parameter–value pairs* separated by ampersand (`&`) query delimiters.

In this URI, for example, the query component (`?action=insert`) is composed of the single parameter–value pair `action=insert`:

`/ords/myUser/soda/v1.0/MyCollection/`**`?`**`action=insert`

And in this URI, the query component is composed of two parameter–value pairs, `fromID=MyCollection` and `limit=2`:

`/ords/myUser/soda/v1.0/metadata-catalog/`**`?`**`fromID=MyCollection`**`&`**`limit=2`

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 operation GET object, the response body is a single object.

Table 4-1 lists and describes fields that can appear in response bodies.

Table 4-1 Fields That Can Appear in Response Bodies

Field	Description
`key`	String that uniquely identifies an object (typically a JSON document) in a collection.

Table 4-1 (Cont.) Fields That Can Appear in Response Bodies

Field	Description
`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 4-2.

If an operation creates or returns objects, then its response body can have the additional fields in Table 4-2. The additional fields appear after field `items`.

Table 4-2 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 user collections.
`properties`	Properties of collection. This field appears only in the response body of GET user collections.
`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.
`totalResults`	Number of objects in collection (if requested)
`links`	Possible final field for `GET collection` operation. For details, see GET collection.

Example 4-1 Response Body

This example 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.

```
{ "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"   : [ ... ] }
```

Related Topics

- GET object
  `GET` object gets a specified object from a specified collection.
- GET user collections
  `GET` user collections gets all or a subset of the collection names for a given database schema (user account).
- POST bulk delete
  `POST` bulk delete deletes all or a subset of objects from a specified collection, using a filter to specify the subset.
- 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.

4.3 GET catalog

`GET` catalog gets all of the collection names for a given database schema (user account), along with information about each collection.

This information includes links to collection descriptions and a link to a JSON schema that describes the structure and type information of the JSON documents in the collection.

> **Note:**
>
> The existence of a JSON schema requires the collection to have a JSON search index with data-guide support, which requires Oracle Database Release 12c (12.2.0.1) or later.

- URL Pattern for GET catalog
  The URL pattern for `GET` catalog is described.
- Response Codes for GET catalog
  The response codes for GET catalog are described.

Related Topics

- Collection Specifications
  A collection specification is a JSON object that provides information about the Oracle Database table or view underlying a collection object. The table or view is created when you create the collection.

4.3.1 URL Pattern for GET catalog

The URL pattern for `GET` catalog is described.

`/ords/`*`database-schema`*`/soda/`*`version`*`/metadata-catalog`

Without parameters, operation `GET` catalog gets catalog information for all collections in *`database-schema`*.

You can include one or more parameter–value pairs at the end of the URL, preceded by a question mark (?) and separated by ampersand (&) query delimiters.

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

4.3.2 Response Codes for GET catalog

The response codes for GET catalog are described.

200

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

```
{ "items": [
  { "name"       : "employees",
    "properties" : { .. .},
    "links"      : [
      { "rel" : "describes",
        "href" :
        "http://host:port/.../database-schema/soda/version/employees" },
      { "rel" : "canonical",
        "href" :
        "http://host:port/.../database-schema/soda/version/metadata-catalog/employees",
        "mediaType" : "application/json" },
      { "rel" : "alternate",
        "href" :
        "http:host:port/.../database-schema/soda/version/metadata-catalog/employees",
        "mediaType":"application/schema+json" } ] },
  { "name"       : "departments",
    "properties" : { ... },
    "links"      : [ ... ] }
  ...
  { "name"       : "regions",
    "properties" : { ... },
    "links"      : [ ... ] } ],
  "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"`).

400

Parameter value is not valid.

401

Access is not authorized.

Related Topics

- 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.

4.4 GET user collections

`GET` user collections gets all or a subset of the collection names for a given database schema (user account).

- URL Pattern for GET user collections
  The URL pattern for `GET` user collections is described.
- Response Codes for GET user collections
  The response codes for GET user collections are described.

Related Topics

- Listing the Documents in a Collection with SODA for REST
  An example is given of listing the documents in a collection, using a `GET` operation.

4.4.1 URL Pattern for GET user collections

The URL pattern for `GET` user collections is described.

`/ords/database-schema/soda/version/`

Without parameters, GET user collections gets all collection names in *`database-schema`*.

You can include one or more parameter–value pairs at the end of the URL, preceded by a question mark (?) and separated by ampersand (&) query delimiters.

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

4.4.2 Response Codes for GET user collections

The response codes for GET user collections are described.

200

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

```
{ "items" : [
  { "name"       : "employees",
    "properties" : {...} },
  { "name"       : "departments",
    "properties" : {...} },
  ...
  { "name"       : "regions",
    "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"`).

400

Parameter value is not valid.

401

Access is not authorized.

404

The database schema (user) was not found.

4.5 GET JSON schema for collection

This operation gets a JSON schema that describes the structure and type information of the JSON documents in a given collection.

> **Note:**
>
> The existence of a JSON schema requires the collection to have a JSON search index with data-guide support, which requires Oracle Database Release 12c (12.2.0.1) or later.

Besides a JSON schema for the collection, the operation also returns the collection metadata, as the value of field `properties`.

- URL Pattern for GET JSON schema for collection
  The URL pattern for getting a JSON schema for a given collection is described.
- Response Codes for GET JSON schema for collection
  The response codes for getting a JSON schema for a given collection are described.

4.5.1 URL Pattern for GET JSON schema for collection

The URL pattern for getting a JSON schema for a given collection is described.

`/ords/database-schema/soda/version/metadata-catalog/collection`

No parameters.

4.5.2 Response Codes for GET JSON schema for collection

The response codes for getting a JSON schema for a given collection are described.

200

Success. The response body contains a JSON schema for the collection, as the value of field `schema`, and the collection metadata, as the value of field `properties`.

For example:

```
{"name"       : "employees",
 "properties" : {
   "schemaName"         : "MYUSER",
   "tableName"          : "EMPLOYEES",
   "keyColumn"          : {"name"             : "ID",
                           "sqlType"          : "VARCHAR2",
                           "maxLength"        : 24,
                           "path"             : "_id",
                           "assignmentMethod" : "MONGO"},
   "contentColumn"      : {"name"       : "DOCUMENT",
                           "sqlType"    : "VARCHAR2",
                           "maxLength"  : 4000,
                           "validation" : "STRICT"},
   "versionColumn"      : {"name"   : "CHECKSUM",
                           "type"   : "String",
                           "method" : "UUID"},
   "lastModifiedColumn" : {"name"  : "LAST_MODIFIED",
                           "index" : "PEOPLE_T1"},
   "readOnly"           : false},
 "schema"     : {
   "type"       : "object",
   "properties" : {
     "dob"      : {"type"                   : "string",
                   "o:length"               : 16,
                   "o:preferred_column_name" : "dob"},
     "name"     : {"type"                   : "string",
                   "o:length"               : 16,
                   "o:preferred_column_name" : "name"},
     "email"    : {"type"                   : "array",
                   "o:length"               : 64,
                   "o:preferred_column_name" : "email",
                   "items"                  : {
                     "type"                   : "string",
                     "o:length"               : 32,
                     "o:preferred_column_name" : "scalar_string"}},
     "empno"    : {"type"                   : "number",
                   "o:length"               : 8,
                   "o:preferred_column_name" : "empno"},
     "title"    : {"type"                   : "string",
                   "o:length"               : 16,
                   "o:preferred_column_name" : "title"},
     "salary"   : {"type"                   : "number",
                   "o:length"               : 8,
                   "o:preferred_column_name" : "salary"},
     "spouse"   : {"type"                   : "null",
                   "o:length"               : 4,
                   "o:preferred_column_name" : "spouse"},
     "address"  : {"type"                   : "object",
                   "o:length"               : 128,
                   "o:preferred_column_name" : "address",
                   "properties"             : {
                     "city"   : {"type"                   : "string",
                                 "o:length"               : 16,
                                 "o:preferred_column_name" : "city"},
                     "state"  : {"type"                   : "string",
                                 "o:length"               : 2,
                                 "o:preferred_column_name" : "state"},
                     "street" : {"type"                   : "string",
                                 "o:length"               : 32,
```

```
                                "o:preferred_column_name" : "street"}}},
    "company"  : {"type"                    : "string",
                  "o:length"                : 16,
                  "o:preferred_column_name" : "company"},
    "location" : {"type"                    : "object",
                  "o:length"                : 64,
                  "o:preferred_column_name" : "location",
                  "properties"              : {
                    "type"        : {
                      "type"                    : "string",
                      "o:length"                : 8,
                      "o:preferred_column_name" : "type"},
                    "coordinates" : {
                      "type"                    : "array",
                      "o:length"                : 32,
                      "o:preferred_column_name" : "coordinates",
                      "items"                   : {
                        "type" : "number",
                        "o:length" : 8,
                        "o:preferred_column_name" : "scalar_number"}}}},
    "department" : {"type"                    : "string",
                    "o:length"                : 16,
                    "o:preferred_column_name" : "department"}}},
 "links" : [
   {"rel"       : "describes",
    "href"      :
    "http : //host:port/.../database-schema/soda/version/employees"},
   {"rel"       : "canonical",
    "href"      :
    "http : //host:port/.../database-schema/soda/version/metadata-catalog/employees",
    "mediaType" : "application/json"},
   {"rel"       : "alternate",
    "href"      :
    "http : //host:port/.../database-schema/soda/version/metadata-catalog/employees",
    "mediaType" : "application/schema+json"}]}
```

401

Access is not authorized.

404

The collection does not exist.

Related Topics

- 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.

4.6 GET actions

`GET` actions gets all of the available custom actions.

- URL Pattern for GET actions
  The URL pattern for `GET` actions is described.

4.6.1 URL Pattern for GET actions

The URL pattern for `GET` actions is described.

`/ords/database-schema/soda/version/custom-actions/`

No parameters.

4.7 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.

- URL Pattern for GET collection
  The URL pattern for `GET` collection is described.
- Response Codes for GET collection
  The response codes for `GET` collection are described.
- Links Array for GET collection
  The `links` array for `GET` collection is described.

Related Topics

- POST query
  `POST` query gets all or a subset of objects from a collection, using a filter.
- Listing the Documents in a Collection with SODA for REST
  An example is given of listing the documents in a collection, using a `GET` operation.

4.7.1 URL Pattern for GET collection

The URL pattern for `GET` collection is described.

`/ords/database-schema/soda/version/collection/`

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

You can include one or more parameter–value pairs at the end of the URL, preceded by a question mark (?) and separated by ampersand (&) query delimiters.

Parameter	Description
`limit=n`	Limits number of objects returned to a maximum of *n*.
`offset=n`	Skips *n* (default: 0) objects before getting the first of those returned.
`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.

Parameter	Description
`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 a `lastModified` time stamp later than *`timestamp`*.
`until=`*`timestamp`*	Gets only objects with a `lastModified` time stamp earlier than *`timestamp`*.
`q=`*`filter`*	Equivalent to a `POST query` action where *`filter`* is a QBE that is passed in the body of the request.

Related Topics

- Links Array for GET collection
  The `links` array for `GET` collection is described.

4.7.2 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`

400

Parameter value is not valid.

401

Access is not authorized.

404

Collection was not found.

Related Topics

- Links Array for GET collection
  The `links` array for `GET` collection is described.
- 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.

4.7.3 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 4-3 shows how `GET collection` parameters determine mode and the existence and content of the `links` array.

Table 4-3 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 4-3 (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`	`lastModified` Timestamp	Does not exist.
`q=QBE`	Query	Does not exist.

Related Topics

- Response Codes for GET collection
  The response codes for `GET` collection are described.

4.8 GET object

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

- URL Pattern for GET object
  The URL pattern for `GET` object is described.
- Request Headers for GET object
  The request headers for `GET` object are described.
- Response Codes for GET object
  The response codes for `GET` object are described.

Related Topics

- Finding Documents in Collections with SODA for REST
  An example is given of retrieving a document from a collection by providing its key.

4.8.1 URL Pattern for GET object

The URL pattern for `GET` object is described.

`/ords/database-schema/soda/version/collection/key`

4.8.2 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*.

Header	Description
`If-None-Match=etag`	Returns response code 304 if the etag (object version) value you set in the header matches the etag value of the document.

4.8.3 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

The object was not modified.

401

Access is not authorized.

404

Collection or object was not found.

Related Topics

- Request Headers for GET object
  The request headers for `GET` object are described.
- 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.

4.9 DELETE collection

`DELETE` collection deletes a collection.

To delete all *objects* from a collection, but *not delete the collection* itself, use POST bulk delete.

- URL Pattern for DELETE collection
  The URL pattern for `DELETE` collection is described.
- Response Codes for DELETE collection
  The response codes for `DELETE` collection are described.

Related Topics

- Dropping a Document Collection with SODA for REST
  An example is given of dropping a collection.

4.9.1 URL Pattern for DELETE collection

The URL pattern for `DELETE` collection is described.

`/ords/database-schema/soda/version/collection/`

No parameters.

4.9.2 Response Codes for DELETE collection

The response codes for `DELETE` collection are described.

200

Success—collection was deleted.

401

Access is not authorized.

404

Collection was not found.

Related Topics

- 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.

4.10 DELETE object

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

- URL Pattern for DELETE object
  The URL pattern for `DELETE` object is described.
- Response Codes for DELETE object
  The response codes for `DELETE` object are described.

Related Topics

- Removing a Single Document from a Collection with SODA for REST
  You can use HTTP operation `DELETE` to remove a single document from a collection.
- Removing Multiple Documents from a Collection with SODA for REST
  You can remove multiple JSON documents from a collection with HTTP operation `POST`, using custom-action `delete` or `truncate` in the request URL. Use `truncate` to remove *all* JSON documents from the collection. Use `delete` together with a QBE to delete only the documents that match that filter.

4.10.1 URL Pattern for DELETE object

The URL pattern for `DELETE` object is described.

`/ords/database-schema/soda/version/collection/key`

No parameters.

4.10.2 Response Codes for DELETE object

The response codes for `DELETE` object are described.

200

Success—object was deleted.

401

Access is not authorized.

404

Collection or bject was not found.

405

Collection is read-only.

Related Topics

- 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.

4.11 PATCH JSON document

`PATCH` JSON document replaces a specified object with an patched (edited) copy of it.

> **Note:**
>
> To use operation `PATCH` JSON document you need Oracle Database Release 18c or later.

- URL Pattern for PATCH JSON document
  The URL pattern for `PATCH` JSON document is described.
- Request Headers for PATCH JSON document
  Use header `Content-Type=application/json-patch+json` for operation `PATCH` JSON document.

- Request Body for PATCH JSON document
  The request body for `PATCH` JSON document contains a JSON Patch specification, that is, an array of objects, each of which specifies a JSON Patch step (operation). The operations are performed successively in array order.
- Response Codes for PATCH JSON Document
  The response codes for `PATCH` JSON document are described.

Related Topics

- Patching a Single JSON Document with SODA for REST
  You can selectively update (patch) parts of a single JSON document using HTTP operation `PATCH`. You specify the update using a JSON Patch specification.
- Patching Multiple JSON Documents in a Collection with SODA for REST
  You can update (patch) multiple JSON documents in a collection by querying the collection to match those documents and specifying the changes to be made. You specify the update with a JSON Patch specification, using QBE operator `$patch`. You use HTTP operation `POST` with custom-action `update` in the request URL.

4.11.1 URL Pattern for PATCH JSON document

The URL pattern for `PATCH` JSON document is described.

`/ords/database-schema/soda/version/collection/key`

No parameters.

4.11.2 Request Headers for PATCH JSON document

Use header `Content-Type=application/json-patch+json` for operation `PATCH` JSON document.

4.11.3 Request Body for PATCH JSON document

The request body for `PATCH` JSON document contains a JSON Patch specification, that is, an array of objects, each of which specifies a JSON Patch step (operation). The operations are performed successively in array order.

The syntax and meaning of a JSON Patch specification, which describes changes to a JSON document, are specified in the JSON Patch standard, RFC 6902. Paths to parts of a JSON document that are referenced in a JSON Patch specification are specified using the JSON Pointer standard, RFC 6901.

> **See Also:**
>
> - JSON Patch (RFC 6902)
> - JSON Pointer (RFC 6901) for information about JSON Pointer paths

4.11.4 Response Codes for PATCH JSON Document

The response codes for `PATCH` JSON document are described.

200

Success — document was patched (updated).

401

Access is not authorized.

404

Document or collection not found.

405

Collection is read-only.

Related Topics

- 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.

4.12 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.

- URL Pattern for POST object
  The URL pattern for `POST` object is described.
- Request Body for POST object
  The request body for `POST` object is the uploaded object to be inserted in the collection.
- Response Codes for POST object
  The response codes for `POST` object are described.

Related Topics

- 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.
- Key Assignment Method
  The key assignment method determines how keys are assigned to objects that are inserted into a collection.
- Inserting a Single Document into a Collection with SODA for REST
  An example is given of inserting a document into a collection.

4.12.1 URL Pattern for POST object

The URL pattern for `POST` object is described.

`/ords/database-schema/soda/version/collection/`

No parameters.

4.12.2 Request Body for POST object

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

4.12.3 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" : "last_modified_timestamp"
                "created"      : "created_timestamp" } ],
  "hasMore" : false }
```

202

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

401

Access is not authorized.

405

Collection is read-only.

501

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

Related Topics

- 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.

4.13 POST query

`POST` query gets all or a subset of objects from a collection, using a filter.

A `links` section is not returned for a `POST` query operation, so you cannot directly do next and previous paging.

> **Note:**
>
> As an alternative to using `POST query` with a filter in the request body you can use `GET collection`, passing the same filter as the value of URL parameter **`q`**. For example, these two commands are equivalent, where the content of file `QBE.1.json` is `{ "User" : "TGATES" }`:
>
> ```
> curl -X POST --data-binary @QBE.1.json -H "Content-Type: application/json"
> http://localhost:8080/ords/database-schema/soda/latest/custom-actions/
> query/MyCollection/
>
> curl -X GET  -H "Content-Type: application/json" http://localhost:8080/
> ords/database-schema/soda/latest/MyCollection/?q={%20%22User
> %22%20:%20%22TGATES%22%20}
> ```

- URL Pattern for POST query
  The URL pattern for `POST` query is described.
- Request Body for POST query
  The request body for a `POST` query action is a QBE (a filter-specification).
- Response Codes for POST query
  The response codes for `POST` query are described.

Related Topics

- 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.
- Querying Using a Filter Specification with SODA for REST
  Examples are given of using a filter specification, or query-by-example (QBE), to define query criteria for selecting documents from a collection.

4.13.1 URL Pattern for POST query

The URL pattern for `POST` query is described.

Query a collection using a filter, with either of these URI patterns:

```
/ords/database-schema/soda/version/custom-actions/query/collection
/ords/database-schema/soda/version/collection?action=query
```

You can include one or more parameter–value pairs at the end of the URL, preceded by a question mark (?) and separated by ampersand (&) query delimiters. Parameters are optional, except as noted.

Parameter	Description
`action=query`	**Required**, if the second syntax form is used. Specifies that the kind of action is a query.
`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`

4.13.2 Request Body for POST query

The request body for a `POST` query action is a QBE (a filter-specification).

The request body cannot be empty, but it can be the empty object, `{}`. If it is `{}` then *all* objects in the collection are returned.

> **See Also:**
>
> *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for information about SODA filter specifications.

4.13.3 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.

401

Access is not authorized.

404

The collection was not found.

Related Topics

- 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.

4.14 POST bulk insert

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

- URL Pattern for POST bulk insert
  The URL pattern for `POST` bulk insert is described.
- Request Body for POST bulk insert
  The request body for `POST` bulk insert is an array of objects.
- Response Codes for POST bulk insert
  The response codes for `POST` bulk insert are described.

Related Topics

- Inserting Multiple Documents into a Collection with SODA for REST
  You can bulk-insert a set of documents into a collection using a JSON array of objects. Each object corresponds to the content of one of the inserted documents.

4.14.1 URL Pattern for POST bulk insert

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

Insert one or more objects into a collection, using either of these URI patterns:

```
/ords/database-schema/soda/version/custom-actions/insert/collection
/ords/database-schema/soda/version/collection?action=insert
```

You can include one or more parameter–value pairs at the end of the URL, preceded by a question mark (?) and separated by ampersand (&) query delimiters.

Parameter	Description
`action=insert`	**Required**, if the second syntax form is used. Specifies that the kind of action is a bulk insert.

4.14.2 Request Body for POST bulk insert

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

4.14.3 Response Codes for POST bulk insert

The response codes for `POST` bulk 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

Access is not authorized.

404

Collection was not found.

405

Collection is read-only.

Related Topics

- 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.

4.15 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 a collection, the empty collection continues to exist. To delete the collection itself, use DELETE collection.

There are two bulk-delete operations, with the HTTP POST actions `delete` and `truncate`, respectively. Action `delete` is more general; you can use it to delete some or all objects in a collection. Action `truncate` always deletes all objects from the collection. Action `delete` is driven by a filter, which selects the objects to delete.

- URL Pattern for POST bulk delete
  The URL pattern for `POST` bulk delete is described.
- Request Body for POST bulk delete (Optional)
- Response Codes for POST bulk delete
  The response codes for `POST` bulk delete are described.

Related Topics

- DELETE collection
  `DELETE` collection deletes a collection.

4.15.1 URL Pattern for POST bulk delete

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

Delete some or all objects from a collection, as determined by a filter using either of these URI patterns:

```
/ords/database-schema/soda/version/custom-actions/delete/collection
```

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

Delete *all* objects from a collection (truncate the collection) using either of these URI patterns:

`/ords/database-schema/soda/version/custom-actions/truncate/collection`

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

You can include one or more parameter–value pairs at the end of the URL, preceded by a question mark (?) and separated by ampersand (&) query delimiters.

Action	Description
`delete`	**Required.** Specifies the deletion of all or a subset of objects from `collection`, using a filter to specify the subset. (The filter must be present, but it can be the empty object, `{}`.)
`truncate`	**Required.** Specifies the deletion of *all* objects from `collection`. Does not use a filter.

> **WARNING:**
>
> If you specify `delete` as the action, and you use the empty object, `{}`, as the filter specification, then the operation deletes *all* objects from the collection.

4.15.2 Request Body for POST bulk delete (Optional)

If the action is `delete` (not `truncate`) then the request body contains the filter (QBE) that specifies which documents to delete from the collection.

> **See Also:**
>
> *Oracle Database SODA for Java Developer's Guide* for information about SODA filter specifications

4.15.3 Response Codes for POST bulk delete

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

200

Success — response body contains the number of deleted objects, as the value of fields `count` and `itemsDeleted`. For example:

```
{ "count"       : 42,
  "itemsDeleted : 42 }
```

401

Access is not authorized.

404

Collection not found.

405

Collection is read-only.

Related Topics

- 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.

4.16 POST bulk update (patch)

The `POST` bulk update operation updates (patches) the objects of a specified collection.

Objects that match a QBE are patched according to a JSON Patch specification.

> **Note:**
>
> To use operation `POST` bulk update you need Oracle Database Release 18c or later.

- URL Pattern for POST bulk update (patch)
  The URL pattern for `POST` bulk update is described.
- Request Body for POST bulk update (patch)
  The request body for `POST` bulk update is an array of objects.
- Response Codes for POST bulk update (patch)
  The response codes for `POST` bulk update are described.

Related Topics

- Patching Multiple JSON Documents in a Collection with SODA for REST
  You can update (patch) multiple JSON documents in a collection by querying the collection to match those documents and specifying the changes to be made. You specify the update with a JSON Patch specification, using QBE operator `$patch`. You use HTTP operation `POST` with custom-action `update` in the request URL.

4.16.1 URL Pattern for POST bulk update (patch)

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

Update one or more objects of a collection, using either of these URI patterns:

```
/ords/database-schema/soda/version/custom-actions/update/collection
/ords/database-schema/soda/version/collection?action=update
```

You can include one or more parameter–value pairs at the end of the URL, preceded by a question mark (?) and separated by ampersand (&) query delimiters.

Parameter	Description
`action=update`	**Required**, if the second syntax form is used. Specifies that the kind of action is a bulk update.

4.16.2 Request Body for POST bulk update (patch)

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

The request body is a QBE that has a `$patch` field whose value is a JSON Patch specification, as in Example 3-10.

4.16.3 Response Codes for POST bulk update (patch)

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

200

Success — response body contains the number of objects updated, as the value of fields `count` and `itemsUpdated`. For example:

```
{ "count"       : 42,
  "itemsUpdated : 42
```

401

Access is not authorized.

405

Not allowed: collection is read-only.

Related Topics

- 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.

4.17 POST index

`POST` index creates indexes on the documents in a specified collection.

> **Note:**
>
> To create an index with SODA you need Oracle Database Release 12c (12.2.0.1) or later. But to create a B-tree index that for a `DATE` or `TIMESTAMP` value you need Oracle Database Release 18c (18.1) or later.

- URL Pattern for POST index
  The URL pattern for `POST` index is described.

- Request Body for POST index
  The request body for `POST` index is a SODA index specification.
- Response Codes for POST index
  The response codes for `POST` index are described.

4.17.1 URL Pattern for POST index

The URL pattern for `POST` index is described.

Index one or more objects of a collection, using either of these URI patterns:

```
/ords/database-schema/soda/version/custom-actions/index/collection
/ords/database-schema/soda/version/collection?action=index
```

You can include one or more parameter–value pairs at the end of the URL, preceded by a question mark (?) and separated by ampersand (&) query delimiters.

Parameter	Description
`action=index`	**Required**, if the second syntax form is used. Specifies that the action is an indexing action.

4.17.2 Request Body for POST index

The request body for `POST` index is a SODA index specification.

A SODA index specification is a JSON object that specifies a particular kind of Oracle Database index, which is used for operations on JSON documents. You can specify these kinds of index:

- B-tree: Used to index scalar JSON values.
- Spatial: Used to index GeoJSON geographic data.
- Search: Used for one or both of the following:
  - Ad hoc structural queries or full-text searches
  - JSON data guide

> **Note:**
>
> To create a data guide-enabled JSON search index, or to data guide-enable an existing JSON search index, you need database privilege `CTXAPP` and Oracle Database Release 12c (12.2.0.1) or later.

> **See Also:**
>
> - *Oracle Database Introduction to Simple Oracle Document Access (SODA)*
>
> for an overview of using SODA indexing
> - *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for information about SODA index specifications

4.17.3 Response Codes for POST index

The response codes for `POST` index are described.

200

Success.

401

Access is not authorized.

404

Collection was not found.

Related Topics

- 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.

4.18 POST unindex

`POST` unindex deletes indexes on objects in a specified collection.

- URL Pattern for POST unindex
  The URL pattern for `POST` unindex is described.
- Request Body for POST unindex
  The request body for `POST` unindex is a SODA index specification. But only the name of the index need be specified — the rest of the index specification is ignored.
- Response Codes for POST unindex
  The response codes for `POST` unindex are described.

4.18.1 URL Pattern for POST unindex

The URL pattern for `POST` unindex is described.

Unindex one or more objects of a collection, using either of these URI patterns:

```
/ords/database-schema/soda/version/custom-actions/unindex/collection
/ords/database-schema/soda/version/collection?action=unindex
```

You can include one or more parameter–value pairs at the end of the URL, preceded by a question mark (?) and separated by ampersand (&) query delimiters.

Parameter	Description
`action=unindex`	**Required**, if the second syntax form is used. Specifies that the action is an unindexing action.

4.18.2 Request Body for POST unindex

The request body for `POST` unindex is a SODA index specification. But only the name of the index need be specified — the rest of the index specification is ignored.

> **See Also:**
>
> *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for information about SODA index specifications

4.18.3 Response Codes for POST unindex

The response codes for `POST` unindex are described.

200

Success.

401

Access is not authorized.

404

Collection was not found.

Related Topics

- 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.

4.19 PUT collection

`PUT` collection creates a collection if it does not exist.

- URL Pattern for PUT collection
  The URL pattern for `PUT` collection is described.
- Request Body for PUT collection (Optional)
  The request body for `PUT` collection optionally contains a collection specification, which defines the metadata of the collection that is created. (If no specification is present then the default metadata is used.)

- Response Codes for PUT collection
  The response codes for `PUT` collection are described.

Related Topics

- Creating a Document Collection with SODA for REST
  How to use SODA for REST to create a new document collection is explained.

4.19.1 URL Pattern for PUT collection

The URL pattern for `PUT` collection is described.

`/ords/database-schema/soda/version/collection`

No parameters.

4.19.2 Request Body for PUT collection (Optional)

The request body for `PUT` collection optionally contains a collection specification, which defines the metadata of the collection that is created. (If no specification is present then the default metadata is used.)

Related Topics

- Collection Specifications
  A collection specification is a JSON object that provides information about the Oracle Database table or view underlying a collection object. The table or view is created when you create the collection.

4.19.3 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

Access is not authorized.

Related Topics

- 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.

4.20 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.

- URL Pattern for PUT object
  The URL pattern for PUT object is described.
- Request Body for PUT object
  The request body for PUT object is the uploaded object.
- Response Codes for PUT object
  The response codes for PUT object are described.

Related Topics

- Replacing Documents in a Collection with SODA for REST
  An example is given of replacing a document in a collection with a newer version. For this, you use HTTP operation PUT.

4.20.1 URL Pattern for PUT object

The URL pattern for PUT object is described.

There are two forms of the URL pattern:

- Pattern for a collection that has client-assigned keys:

  `/ords/database-schema/soda/version/collection/key`

- Pattern for a collection that has system-assigned keys:

  `/ords/database-schema/soda/version/collection/`

No parameters.

4.20.2 Request Body for PUT object

The request body for PUT object is the uploaded object.

4.20.3 Response Codes for PUT object

The response codes for PUT object are described.

200

Success—object was replaced.

401

Access is not authorized.

405

Collection is read-only.

Related Topics

- 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.

5
Collection Specifications

A collection specification is a JSON object that provides information about the Oracle Database table or view underlying a 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.

When you create a collection using `PUT` collection you can set custom metadata for it by providing a collection specification for it as the request body.

Collection metadata for an existing collection can be returned as part of a `GET` catalog operation.

Example 5-1 shows the collection specification that specifies the *default* collection metadata.

Table 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 database schema (user account) 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`

Table 5-1 (Cont.) Collection Specification Fields

Field	Description	Possible Values
`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`
`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`

Table 5-1 (Cont.) Collection Specification Fields

Field	Description	Possible Values
`creationTimeColumn.name`	Name of optional created-on timestamp column. This column has SQL data type `TIMESTAMP` and default value `SYSTIMESTAMP`.	Default: `CREATED_ON`
`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 Default Collection Metadata

```
{
   "schemaName" : "mySchemaName",
   "tableName" : "myTableName",
   "keyColumn" :
   {
      "name" : "ID",
      "sqlType" : "VARCHAR2",
      "maxLength" : 255,
      "assignmentMethod" : "UUID"
   },
   "contentColumn" :
   {
```

```
      "name" : "JSON_DOCUMENT",
      "sqlType" : "BLOB",
      "compress" : "NONE",
      "cache" : true,
      "encrypt" : "NONE",
      "validation" : "STANDARD"
   },
   "versionColumn" :
   {
     "name" : "VERSION",
     "method" : "SHA256"
   },
   "lastModifiedColumn" :
   {
     "name" : "LAST_MODIFIED"
   },
   "creationTimeColumn" :
   {
      "name" : "CREATED_ON"
   },
   "readOnly" : false
}
```

- Key Assignment Method
  The key assignment method determines how keys are assigned to objects that are inserted into a collection.
- Versioning Method
  The versioning method determines how the REST server computes version values for objects when they are inserted into a collection or replaced.

Related Topics

- GET catalog
  `GET` catalog gets all of the collection names for a given database schema (user account), along with information about each collection.

> **See Also:**
>
> - *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 operation POST object. 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. 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.

Related Topics

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

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.

Database role **`SODA_APP`** must be granted to database users before they can use REST SODA. In addition, when a database schema (user account) 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.

- 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.
- Security Considerations for Development and Testing
  Security considerations for development and testing are presented.

> **See Also:**
>
> *Oracle REST Data Services Installation, Configuration, and Development Guide* for information about ORDS security features

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. In this example, replace placeholders <*user_name*> and <*password*> with an appropriate user name and <password>:

```
# Create a user with role SODA Developer.
# (Be sure to replace placeholder <user_name> here.)
java -jar ords.war user <user_name> "SODA Developer"
```

```
# Access the JSON document store using basic authentication.
# (Be sure to replace placeholders <user_name> and <password> here.)
curl -u <user_name>:<password> https://example.com/ords/scott/soda/latest/
```

Index

B

bulk insert of JSON documents, *3-6*
bulk patch (update) of JSON documents, *3-18*
bulk update (patch) of JSON documents, *3-18*

C

collections
  creating, *3-2*
  deleting, *3-4*
  listing, *3-3*
  listing documents in, *3-11*
  removing documents from, *3-9*
  specifications for, *5-1*

D

database role
  SODA_APP, *6-1*
DELETE collection operation, *4-15*
DELETE object operation, *4-16*
deleting a single document from a collection, *3-9*
deleting collections, *3-4*
deleting documents from a collection, *4-24*
documents
  filtering in collections, *3-14*
  inserting into a collection
    in bulk from JSON array, *3-6*
  inserting into collections
    one at a time, *3-5*
  listing in collections, *3-11*
  removing from collections, *3-9*
  replacing in collections, *3-8*
  retrieving from collections, *3-7*

F

filtering documents in collections, *3-14*

G

GET actions operation, *4-10*
GET collection operation, *4-11*
GET object operation, *4-14*
GET user collections operation, *4-7*

I

inserting documents into a collection
  in bulk from JSON array, *3-6*
inserting documents into collections
  one at a time, *3-5*
installing SODA for REST, *2-1*

J

JSON Patch specification, *3-16*, *3-18*, *4-18*
JSON Pointer, *4-18*

K

key assignment method, *5-4*

L

listing collections in a database schema, *3-3*
listing documents in collections, *3-11*

P

PATCH operation for a JSON document, *4-17*
patching (updating) multiple JSON documents, *3-18*
patching a single JSON document, *3-16*
POST bulk delete operation, *4-24*
POST bulk insert operation, *4-22*
POST bulk patch (update) operation, *4-26*
POST bulk truncate (delete all) operation, *4-24*
POST bulk update (patch) operation, *4-26*
POST index operation, *4-27*
POST object operation, *4-19*
POST query operation, *4-21*
POST unindex operation, *4-29*
PUT collection operation, *4-30*
PUT object operation, *4-31*

Q

query-by-example (QBE)
examples, 3-14

R

removing a single document from a collection, 3-9
replacing a document in a collection, 3-8
REST architectural style, 1-2
retrieving documents from collections, 3-7

S

security, 6-1
SODA for REST HTTP operations, 4-1
response bodies, 4-3
SODA for REST HTTP operations *(continued)*
URI forms for, 4-2
SODA_APP database role, 6-1
specifications
collection, 5-1

U

updating (patching) a single JSON document, 3-16
updating (patching) multiple JSON documents, 3-18
URIs used for SODA for REST operations, 4-2

V

versioning method, 5-6