Developing schemaless applications using the Simple Oracle Document Access API's

ORACLE WHITE PAPER | MAY 2017

Disclaimer

Table of Contents

Introduction: Schemaless Application Development

Schema-less development techniques are very popular with today's application developers. The primary characteristic of schema-less development is that application data is stored as a set of self contained documents, rather than being stored as rows and columns in tables. JavaScript Object Notation (JSON) is the format of choice for representing the application data as documents.

Currently, developers adopting schema-less development tend to select a NoSQL-style document store to persist the application data. The products typically provide an application developer with easy to use API's that do not require the application developer to understand SQL in order to develop their applications.

This whitepaper outlines a new abstract API specification, called Simple Oracle Document Access (SODA). The objective of SODA is to enable schemaless application development techniques to be used to create applications that use Oracle Database 12c as the JSON document store. SODA allows a developer to work with an Oracle database without needing to learn SQL and without requiring support from a Database Administrator (DBA). The SODA family of APIs is available starting with Oracle Database 12.1.0.2.0. We will discuss two implementations of the SODA specification, SODA for Java and SODA for REST.

The drivers behind the rise of schema-less application development are well documented. It is generally accepted that schema-less development offers application developers increased flexibility and reduces time to market. This whitepaper assumes that there are solid business reasons why an organization has adopted the schema-less approach and will simply focus on the capabilities of Oracle Database 12c that mean it is still the best choice for managing the application data.

The paper will start by outlining the characteristics of schema-less development and then quickly examine the features and functionality provided by the typical NoSQL style document store. We will discuss SODA and show how it can satisfy the requirements of the application developer while still allowing the organization to use Oracle Database to manage the application data.

Document-based persistence

In traditional application development an application’s data is described using an entity-relationship (E-R) model. The E-R model is then mapped to a set of tables with rows and columns managed by a relational database. When the application data model changes the entity-relationship model has to be revised and the underlying table and column definitions modified. In most cases, it is necessary to take the application and sometimes also other applications offline in order to make changes to the database schema

Schema-less development attempts to avoid this problem by storing application data as documents, rather than as rows and columns in tables. Typically the data is represented as either eXtensible Markup Language (XML) or JavaScript Object Notation (JSON). The primary characteristic of both of these formats is that they are self-describing, allowing individual documents to contain disparate data. Representing application data as documents has the advantage that it separates the details of the application data model from the storage schema. The document encapsulates the complexity of the application data model. The storage schema simply sees a document, and needs no knowledge of its internal structure.

Document based persistence makes it much easier to accommodate changes to the application data model. Since the storage schema is only aware of the existence of the document, as distinct from the structure of the document, the content of the document can be changed without affecting the storage schema. This allows new versions of the application to be deployed without making structural changes to the underlying database. Since no changes to the storage schema are required when the application data model changes, upgrades can be deployed at any time, new versions of an application can be deployed to the production environment as soon as the appropriate acceptance testing is complete. There is no longer any need to wait for a suitable maintenance window, where storage schema changes can be accommodated.

This means that choosing a document based persistence model for application data delivers the extremely flexible mechanism for storing data that today’s developers need.

NoSQL document stores

The documents generated by this type of application are typically managed using some form of document store. Document stores organize the documents they manage into one or more collections. These collections will reflect business objects, or applications. For example, a system may have one collection for Purchase Order documents and another for Invoices, or a collection for User Preferences and another for User History.

Document stores typically use a unique id to identify each document within a collection. The use of the unique id allows the document level operations to be performed extremely efficiently. Most document stores will also track basic metadata for each document, including things like date created, date of last update, owner and document version etc.

The functionality provided by a document store will include:

» Creating a collection, dropping a collection
» Creating, retrieving, updating or deleting a single document based on its ID
» List the documents in a collection
» Querying a collection, typically using some kind of Query By Example (QBE) metaphor
» Creating and dropping indexes

Given the simple level of functionality provided by the document store, the associated application programming interface (API) tends to be quite simple, particularly when compared with traditional SQL based APIs’ such as JDBC.

Oracle Database 12c as a NoSQL Document store

Oracle Database has supported document centric development since Oracle Database 9, with the XML DB functionality providing powerful storage, indexing and query capabilities for XML documents. Oracle Database 12c extends this functionality to JSON, enabling the storage, indexing and querying of JSON documents stored in the Oracle Database. It also introduces the first two implementations of the SODA interface, namely SODA for REST and SODA for Java. When the new JSON capabilities are combined with new the SODA API's, this makes Oracle Database 12c an extremely effective choice for applications developers looking for a NoSQL-style Document store. JSON support and the SODA API's are available with Oracle Database 12c, starting with release Oracle Database 12.1.0.2.0.

Oracle's JSON capabilities are focused on providing full support for schemaless development and document-based storage. Consequently, while Oracle Database understands that a given column contains JSON documents; the documents are stored, indexed and queried without the database having any knowledge of their structure. This leaves developers free to change the structure of their JSON documents as necessary; allowing Oracle Database to delivers the same level of flexibility as a NoSQL document store.

Storing and Managing JSON Documents in Oracle Database 12c

In Oracle Database 12c there is no dedicated JSON data type. JSON documents are stored in the database using standard Oracle data types such as VARCHAR2, CLOB and BLOB. VARCHAR2 can be used where the size of the JSON document will never exceed 4K (32K in environments where 12c extended VARCHAR support has been enabled). To store larger documents, the CLOB or BLOB data types should be used. A new constraint, "IS JSON", is used to ensure that the content of a column is valid JSON. This allows the database to understand that the column is being used as a container for JSON documents. This constraint returns TRUE if the content of the column is well formatted JSON and FALSE otherwise (NULL values also return TRUE). Applying the constraint to a column ensures that only valid JSON documents can be stored in that column of a table.

Using a standard data type such as VARCHAR2 for JSON documents allows Oracle Database to provide full JSON support for all of its advanced features, including disaster recovery, replication, compression, and encryption. Additionally, products that support Oracle Database, such as Oracle Golden Gate and Oracle Data Integrator, (as well as third party tools) seamlessly support JSON documents stored in the database.

SODA: A simple programmatic interface to the Oracle JSON Document Store

Application developers using Oracle as NoSQL document store have a choice of using traditional relational API's or the new SODA family of API's. Choosing SODA gives them the ability to work with JSON documents managed by Oracle Database 12c without having to learn SQL. SODA gives developers the freedom to create, deploy and modify applications without requiring any assistance from an Oracle DBA.

SODA provides a very simple set of API that can be used work with documents stored in an Oracle Database. The Database object, which is required to interact with Collections, is instantiated using a database connection obtained using one of Oracle's standard SQL APIs. The current version of SODA adopts an optimistic locking strategy, however pessimistic locking is being considered for a future release.

The SODA specification defines a set of methods that provide the following functionality:

» Establish a connection to an Oracle Database document store.

» Create and drop a collection.

» Create, retrieve, update and delete a document

» List the contents of a collection

» Search a collection for documents that match a Query by Example (QBE) expression

» Bulk Insert operations on a collection

» Create and drop indexes.

Initially, two implementations of SODA will be available:

» SODA for JAVA: A traditional API designed for developers creating applications using Java.

» SODA for REST: designed for developers utilizing the Representational State Transfer (REST) paradigm.

Both of the initial implementations of SODA are Java based. Implementations in other languages, such as Node.js and a Microsoft .NET are being considered for future releases of the database.

SODA for Java

SODA for Java consists of a set of simple classes that represent a database, a document collection and document. Methods on these classes provide all the functionality required to manage and query collections and work with JSON documents stored in an Oracle Database. SODA for Java uses a standard JDBC database connection and SQL*NET to communicate with the database. This means that SODA for Java is transactional and a series of SODA operations can take place within a single database transaction. Since SODA for Java makes use of a JDBC connection, application developer can mix SODA for Java programming with traditional JDBC programming.

The main classes in SODA for Java are outlined below:

Class	Description	Comments
OracleClient	The generic SODA client class. The Entry point to SODA for JSON.	
OracleRDBMSClient	The Clients class for the Oracle Database	Used to get the OracleDatabase object.
OracleDatabase	Represents a document Store. A document store consists of one or more collections	Used to access collections.
OracleDatabaseAdmin	Used to create and drop collections	
OracleCollection	Represents a collection within a Document Store	
OracleCollectionAdmin	Used to create and drop indexes	
OracleDocument	Represents a document with a Document Store	Update (or create) the document with the given id

In order to use any of SODA for Java objects it is necessary to establish a connection to the Document Store. The connection is established by invoking the getDatabase() method on the OracleRDBMSClient class. The getDatabase() method expects to be provided with a JDBC connection object which will provide connectivity to the target Oracle Database.

SODA for REST

Developing using Representational State Transfer (REST) based services is very popular with today's application developers. REST is based on standard Web Architectures and maps HTTP verbs such as PUT, POST, GET and DELETE onto operations on resources or documents, managed by a Web Server. In the case of JSON, the resources in question are JSON documents. Each managed by the system is assigned a unique URL. The URL typically consists of identifier for a collection, followed by a unique identifier for the document.

SODA for REST is a set of REST-based services that provide a document store interface based on the SODA specification. SODA for REST is delivered as part of Oracle Rest Data Services (ORDS) 3.0. It is implemented as a Java servlet and uses SODA for Java to communicate with the backend document store. Applications communicate with the SODA for REST servlet using HTTP. The SODA for REST servlet can be run from inside any Java container supported by ORDS 3.0. It can also be run under the Oracle Database's native HTTP Server. SODA for REST can be invoked from any programming or scripting language that is capable of making HTTP calls which means it can be used with all modern development environments and frameworks.

In SODA for REST, HTTP verbs such as PUT, POST, GET, and DELETE map to CRUD operations on JSON documents. Operations such as LIST, QBE and indexing are performed using POST operations. SODA for REST models the documents and collections within a document store as a set of resources in a simple three level hierarchy. An easy way to think of this is that the document store for a particular schema is represented by a folder, each collection within that schema is represented by a sub-folder, and each document with the collection in represented as a file. The names of subfolders are derived from the names of the collections, and the names of the files are derived from the IDs of the documents. Each object managed by the document store is associated with its own unique URL.

Like all REST based interfaces SODA for REST is stateless. This means that each operation performed using SODA for REST is considered to be an atomic transaction in its own right. Since SODA for REST is based on the HTTP protocol there is no infrastructure available which would allow multiple SODA for REST operations to be considered part of the same transaction.

The following table shows how SODA for REST maps HTTP verbs and URLs to operations on JSON collections that are managed by the Oracle Database.

SERVICE	**VERB**	**URL**	**Action**
List Collection	GET	/DBJSON/SCHEMA	List all collections in a schema
Create Collection	PUT	/DBJSON/SCHEMA/collection	Create a collection if necessary
Insert Document	POST	/DBJSON/SCHEMA/collection	Insert a document into a collection
Get Document	GET	/DBJSON/SCHEMA/collection/id	Get the document with the specified id.
List Collection	GET	/DBJSON/SCHEMA/collection	Get all documents in a collection
Update Document	PUT	/DBJSON/SCHEMA/collection/id	Update (or create) the document with the given id
Delete Document	DELETE	/DBJSON/SCHEMA/collection/id	Delete the document with the given id
Query By Example	POST	/DBJSON/SCHEMA/collection?action=query	Find documents that contain objects matching the specified filter

The URLs in this table and the following examples assume that the SODA for REST servlet is running directly in the database, using the virtual path '/DBJSON'.

When running with the Oracle Database's HTTP server the components of the URL are as follows:

» **"/DBJSON"** is the virtual folder that was mapped to the SODA for REST servlet when it was added to database's xdbconfig.xml document. This value is case sensitive.

» **"/SCHEMA"** is the database schema that is being used to manage the JSON collections. Schema needs to be supplied in uppercase, unless the database schema was created using a quoted identifier.

» **"/collection"** is the name of a collection within the specified schema.

If SODA for REST is running as part of ORDS 3.0 then the URLs required to access the SODA for REST functionality would be in the following format: "/ords/schema/dbjson/version/collection". When running with ORDS 3.0 the components of the URL are as follows:

» **"/ords"** is the virtual folder that has been mapped to the ORDS 3.0 servlet.

» **"/schema"** is the identifier assigned in ORDS to the database schema that manages the collection tables.

» **"/dbjson"** identifies that the request should be routed to the SODA for REST component of ORDS.

» **"/version"** identifies which version of the SODA for REST code should be invoked. You can use "latest" to ensure that you always use the latest version of the SODA for REST code.

» **"/collection"** is the name of the collection within the specified schema

Developing applications with SODA

Let’s look at some simple examples of how to use the SODA functionality to develop an application. This section will address most of the common operations that will be required when building an application and show examples of how to perform the operation in SODA for Java and SODA for REST. The SODA for REST examples will show how to invoke the required service using cURL[1], which is a command line tool for making HTTP requests.

Creating a Collection

SODA allows application developers to create a collection object without requiring any knowledge of SQL. A collection is simply a container for a set of (related) documents. Under the covers, collections are simply relational tables that contain a set of documents. Collection tables have a minimum of two columns. The first column contains the unique ID for each document; the second column contains the document itself. A collection table may also have additional columns to manage metadata such as the date the document was created, the date it was last updated, the document owner, document version etc.

Developers have the option of supplying a “Collection Properties” document to the create collection operation. This is used to override the default set of properties that are used when creating collections. Collection Properties provide the developer with control over a number of aspects of the collection including:

» The method used to assign a unique id for each document in the collection. Developers can choose between client assigned keys and a number of different algorithms for server assigned keys.

» What metadata is maintained for each document

» What indexes should be created on the collection

» Which algorithms should be used to manage versioning

» The names of the columns in the relational table behind a collection.

» Whether or not to apply the IS JSON constraint to the document column.

Collection Properties can also be used to create a collection on top of on an existing database table.

A sample collection properties document is shown below:

*** *Example Collection Properties Document*

*** *Collection tables can be examined using the view*

Creating a collection using SODA for Java

In SODA for Java a collection is created using the createCollection() method provided by the OracleDatabaseAdmin object. This object is obtained by invoking the admin() method on by the OracleDatabase class. The createCollection() method takes two parameters, the name of the collection and, optionally, a collection properties document. The following example shows the code required to create a collection using SODA for JAVA:

```
public OracleCollection createCollection(OracleConnection conn, String collectionName)
throws OracleException {

        OracleRDBMSClient client = new OracleRDBMSClient();
        OracleDatabase database = client.getDatabase(conn);
        OracleCollection collection = database.admin().createCollection(collectionName);
        return collection;

};
```

The method returns the OracleCollection object for the collection.

[1] For more information about cURL, see cURL’s official webite: http://curl.haxx.se/

Creating a collection using SODA for REST

In SODA for REST a collection is created by performing a PUT operation on the URL that represents the collection. Collection properties can be provided as the body to the PUT. The following example shows a cURL command that would create a collection using SODA for REST's "Create Collection" service:

```
curl --digest -X PUT --write-out "%{http_code}\n" -u SCOTT:tiger http://localhost:8080/DBJSON/SCOTT/MyCollection

201
```

In REST, PUT operations are idempotent, meaning that they will return exactly the same result no matter how many times they are invoked. Consequently no error is raised if the collection already exists; the operation simply returns HTTP status code 200 to indicate that the collection already exists. If a new collection is created then the operation returns HTTP status code 201.

Regardless of whether the collection is created using SODA for Java or SODA for REST, the default table for managing collections will be similar to the one shown below:

```
SQL> desc "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

SQL>
```

Creating documents

Creating a document using SODA for Java

The OracleDatabase class provides a set of methods that allow an OracleDocument object to be created from a number of different Java objects. Once the OracleDocument has been created the insertAndGet() method on the OracleCollection class is used to add the document to the collection.

The following example shows the SODA for Java code required to add a document to a collection and return its key. In this example the parameter "is" consists of an input stream on a well formed JSON document.

```
public String createDocument(OracleConnection conn, String collectionName, InputStream is)
throws OracleException {
    OracleRDBMSClient client = new OracleRDBMSClient();
    OracleDatabase database = client.getDatabase(conn);
    OracleCollection collection = database.openCollection(collectionName);
    OracleDocument document = database.createDocumentFromStream(is);
    document = collection.insertAndGet(document);
    return document.getKey();
};
```

The insertAndGet() method returns an updated OracleDocument object that provides methods to access the metadata and content of the new document.

Creating a document using SODA for REST

To add a document to a collection using SODA for REST, perform a POST operation on the URL that represents the collection. The document to be inserted is provided as the body of the POST. The following example shows a cURL command that would add a document to a collection using SODA for REST's "Insert Document" service. In this example file po.json is assumed to contain a well-formed JSON document. Note the use of the "-H" to ensure that the HTTP request's content-type header are set correctly.

```
curl --digest -u SCOTT:tiger -X POST  -H "Accept: application/json" -H "Content-type: application/json" --upload-
file po.json http://localhost:8080/DBJSON/SCOTT/MyCollection
```

The response to the HTTP POST request is shown below

```
{
     "items": [
             {
                     "id": "A450557094D04957B36346F630CDDF9A",
                     "etag": "C1354F27A5180FF7B828F01CBBC84022DCF5F7209DBF0E6DFFCC626E3B0400C3",
                     "lastModified": "2015-02-09T01:03:48.291462",
                     "created": "2015-02-09T01:03:48.291462"
             }
     ],
     "hasMore": false,
     "count": 1
}
```

The response consists of a simple JSON document that provides the calling application with access to the metadata about the new document. The document consists of an array "items" and some additional housekeeping data. The array contains one object for each document created by the POST operation. Each object contains the following keys

» "id": Contains the unique id for this document.

» "etag": A value that can be used to determine if the document has been updated.

» "lastModified": The date and time of the last update of the document.

» "created": The data and time the document was created.

Note that the actual key present in the response object is dependent on the contents of the collection properties. The response also contains a count of the number of documents that were created.

Making "items" an array allows the same response object to be used in conjunction with SODA for REST's "Bulk Insert" option. Bulk insert is triggered by adding the parameter "action=insert" to the URL and passing a JavaScript array as the body of the POST request. In Bulk Insert mode SODA for REST creates a document from each member of the input array and the response's items array contains one object for each of the documents created.

With the above examples we can see how the document store approach differs from that of a traditional SQL-based application. In a SQL-based application, a developer would first have to define all of the data elements in a relational schema. Adding a new record would require the creation of a SQL INSERT statement in which all columns of the new data must match the existing relational schema.

With the document store approach, new documents are added to a collection as JSON objects. There are no restrictions imposed by the database on the keys contained in those documents. And the API calls are simpler for developers who might be accustomed to object-oriented programming environments.

Retrieving Documents

One of the fundamental tenants of document store based application development is that the application will always know the unique ID of the documents it needs to retrieve. Given an ID retrieving the document is very simple.

Retrieving a Document using SODA for Java

A document is retrieved from a collection by calling the findOne() method of the OracleCollection class.

```
    public String retrieveDocumentContent(OracleCollection collection, String ID)
    throws OracleException {
        OracleDocument doc = collection.findOne(ID);
        return doc.getContentAsString();
    };
```

This returns an OracleDocument object, which provides methods to get the content of the document as various different Java Objects.

Retrieving a Document using SODA for REST

With SODA for REST retrieving a document from a collection is as simple as performing a GET operation on the URL that represents the ID of the document. The URL is determined by taking the URL for the collection and appending the unique ID of the document. The following example shows a cURL command that would retrieve a document from a collection using SODA for REST's "Get Document" service:

```
curl --digest -X GET    --write-out "%{http_code}\n" -u SCOTT:tiger
http://localhost:8080/DBJSON/SCOTT/Collection1/14E6656206114A12950ABF745C309CC5
```

The response to the HTTP POST request is the JSON document with the specified ID. If a matching document is found the HTTP status code will be 200 and the body of the response will be the contents of the document. If the document is not found the HTTP status code will be 404. An abbreviated view of the response is shown below:

```
{
     "PONumber": 14,
     "Reference": "SVOLLMAN-20140525",
     "Requestor": "Shanta Vollman",
     "User": "SVOLLMAN",
     "CostCenter": "A50",
     "ShippingInstructions": {
             "name": "Shanta Vollman",
             "Address": {
                     "street": "200 Sporting Green",
                     "city": "South San Francisco",
                     "state": "CA",
                     "zipCode": 99236,
                     "country": "United States of America"
             },
             "Phone": [
                     {
                             "type": "Office",
                             "number": "823-555-9969"
                     }
                     ]
     },
     "Special Instructions": "Counter to Counter",
     "LineItems": […]
}
```

Querying Collections

If the application does not know the ID of a document it is interested in it will have to search for it. Oracle Database 12c supports two types of search operations over collections. The first is an operation that lists all of the documents in the collection. The second is a query-by-example (QBE) based search that filters documents based on their content. SODA provides the developer with control over what information is returned by the server. Developers have the choice of returning metadata, content or both. Developers can also control how to paginate the generated results.

List operations simply provide an unfiltered list of all the documents in a collection.

Query-by-example uses a template document to generate a filtered list of documents that match a set of supplied query predicates. The developer creates a template document which provides an example of the desired keys and values. The following example shows a very basic QBE template:

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

This document would trigger a search for documents that contain a top-level key called User, with the value “TGATES”:

The QBE capability provided by Oracle Database 12c lets you search based on keys that occur at any depth within a document, including keys that are part of an array. A full description of the QBE syntax can be found in the documentation[2]. When searching for documents that satisfy the QBE query criteria, QBE takes full advantage of any indexes that have been created on the collection. When a QBE operation takes place SODA takes the QBE document and translates it into an equivalent set of JSON Path expressions that can be executed using the JSON_EXISTS operator and are optimized by the database.

SODA provides support for paginating the results of List and Search operations. Use a “limit” to control how many documents are included in the result set. Setting a limit of 4 will restrict the results set to the first four matching documents. Use “offset” to control which document will be the first entry in the result set. Setting an offset of 5 will cause the first 5 documents to be omitted from the result set. By combining limit and offset it is possible to paginate a large result set.

Listing Collections using SODA for Java

With SODA for Java, list operations are performed using the OracleOperationBuilder class. The OracleOperationBuilder object is obtained by invoking the OracleCollection object’s find() method. Limit and skip methods on the OraceOperationBuilder class are used to control pagination of the result set. The result of the list operation is returned as an OracleCursor object obtained by invoking the OracleOperationBuilder object’s getCursor method. The following example shows getting the first four documents from a collection.

```
    public void listDocuments(OracleCollection collection)
    throws OracleException {
        OracleCursor results = collection.find().limit(4).getCursor();
        while (results.hasNext()) {
            OracleDocument doc = results.next();
            System.out.println(doc.getKey());
        }
    };
```

The OracleCursor object provides methods for navigating through the result set and returning the matching documents.

Listing Collections using SODA for REST

In SODA for REST a “List Collection” operation is done by performing a GET operation on the URL that represents the collection. Pagination support is provided by adding the parameters “limit” and “offset” in the URL. The following example shows a cURL command that would return the first four documents from a collection using SODA for REST’s “List Collection” service:

```
curl --digest -X GET    --write-out "%{http_code}\n" -u SCOTT:tiger
http://localhost:8080/DBJSON/SCOTT/Collection1?fields=id\&limit=4
```

Query by Example using SODA for Java

[2] https://docs.oracle.com/cd/E56351_01/doc.30/e58123/rest.htm#ADRST256

With SODA for Java, Query-By-Example (QBE) operations are performed by invoking the filter method on the OracleOperationBuilder class. The OracleOperationBuilder object is obtained by invoking the OracleCollection object’s find() method. The QBE is executed by invoking the the OracleOperationBuilder object’s filter method(), passing the QBE specification as an instance of OracleDocument. The filter method returns a new instance of the OracleOperationBuilder class where the conditions specified in the QBE document have been applied to the collection. Limit and skip methods on the OraceOperationBuilder class are used to control pagination of the result set. The result of the QBE operation is returned as an OracleCursor object obtained by invoking the OracleOperationBuilder object’s getCursor method. The following example shows getting the documents which contain a top level key called “User” with the value “TGATES”.

```
    public void listDocuments(OracleConnection conn, String collectionName)
    throws OracleException {
        OracleRDBMSClient client = new OracleRDBMSClient();
        OracleDatabase database = client.getDatabase(conn);
        OracleDocument qbeSpec = database.createDocumentFromString("{ \"User\" : \"TGATES\" }");
        OracleCollection collection = database.openCollection(collectionName);
        OracleCursor results = collection.find().filter(qbeSpec).getCursor();
        while (results.hasNext()) {
            OracleDocument doc = results.next();
            System.out.println(doc.getKey());
        }
    };
```

The OracleCursor object provides methods for navigating through the result set and returning the matching documents.

Query by Example using SODA for REST

In SODA for REST a “Query By Example” operation is done by performing a POST operation on the URL that represents the collection. The parameter “action=query” is used to indicate that the POST contains a QBE request. The body of the request is a JSON document containing the query specification. Pagination support is provided by adding the parameters “limit” and “offset” in the URL. The following example shows a cURL command that would use SODA for REST’s “Query by Example” service to return the first four documents where the key “User” contains the value “TGATES”. The QBE is supplied using cURL’s --data parameter.

```
curl --digest -u SCOTT:tiger -X POST  -H "Accept: application/json" -H "Content-type: application/json" --data '{
"User" : "TGATES" }' http://localhost:8080/DBJSON/SCOTT/Collection1?action=query\&limit=4\&fields=id
```

SODA for REST provides developers with control over whether “List Collection” and “Query by Example” operations return metadata, content, or both. The parameter “fields” is used to control what is returned. Specifying “fields=id” will limit the results to just the ID of the matching documents. Specifying “fields=value” will return the content of the matching documents. Specifying “fields=all” will return both id and content for the matching documents.

In SODA for REST the response to “List Collection” and “Query By Example” operations consists of a JSON document containing an array “items”. The array contains one entry for each document returned by the operation, along with metadata describing how many documents were found, and whether additional results are available.

The response to the HTTP request is shown below.

```
{
     "items": [
              {
                       "id": "09615A98B2B941AF94D84FD44D04AB9C",
                       "etag": "D78FBD68D31C1F7381E944E44693F4B55D5E5A2CB7A5B14EDCD285D0E9AB879E",
                       "lastModified": "2015-02-10T01:15:13.631231",
                       "created": "2015-02-10T01:15:13.631231"
              }, {
                       "id": "09976262769F4E209320E7838C59F5B3",
                       "etag": "8DF0C9CB4C53A7E2C08C1314E97C3277B9686823C7E4622ABAC9F91F502815B9",
                       "lastModified": "2015-02-10T01:15:13.631231",
                       "created": "2015-02-10T01:15:13.631231"
              }, {
                       "id": "14E6656206114A12950ABF745C309CC5",
                       "etag": "7B53F631A4A093738C8281DC7E91FF412232D786A5DB49BBB24F871B745D116A",
                       "lastModified": "2015-02-10T01:15:13.631231",
                       "created": "2015-02-10T01:15:13.631231"
              }, {
                       "id": "1A7828FBCB0647F595B8E075EDFC6E93",
                       "etag": "7486F12F8AF733222A7F4A3C6136E0DCFBD5315EC13E86F86D1A55A78127774D",
                       "lastModified": "2015-02-10T01:15:13.631231",
                       "created": "2015-02-10T01:15:13.631231"
              }
     ],
     "hasMore": true,
     "count": 4,
     "offset": 0,
     "limit": 4
}
```

In this example the parameter “fields=id” was used to restrict the result set to just metadata about the matching documents.

When fetching content, SODA for REST provides a paging mechanism that makes it easy to navigate the entire result set. It does this by including a “links” object in the HTTP response that provides the URLs needed to navigate through pages of the result set. This can be seen in the following snippet.

```
     "hasMore": true,
     "count": 5,
     "offset": 5,
     "limit": 5,
     "links": [
              {
                       "rel": "first",
                       "href": "/DBJSON/SCOTT/Collection1?offset=0&limit=5"
              }, {
                       "rel": "prev",
                       "href": "/DBJSON/SCOTT/Collection1?offset=0&limit=5"
              }, {
                       "rel": "next",
                       "href": "/DBJSON/SCOTT/Collection1?offset=10&limit=5"
              }
     ]
```

In order to minimize the size of the HTTP response the “links” information is omitted when “fields=id” is specified

\

Indexing Collections

Oracle Database 12c includes support for indexing JSON collections. There are two types of indexes that can be created on JSON content. The first is a conventional B-Tree (or bitmap) index that indexes the value of a particular key or, in the case of a compound index, the values of a particular set of keys. (Under the covers, these are simply function-based indexes.)

The second is an inverted list-based index, referred to as a *JSON search index*, which indexes each JSON document as a whole. This provides complete indexing of a JSON document without requiring any prior knowledge of its structure. The Oracle Database optimizer automatically makes use of these indexes when it optimizes queries, regardless of whether the queries consist of conventional SQL statements or SODA QBE expressions .

Both kinds of index can be created using SODA. The definition of what is to be indexed is provided using an Index Specification document. When a create index operation takes place SODA uses the information in the index specification document to generate the SQL statements that will create the required index.

What is to be indexed is specified using an *index specification document*. Here is a sample index specification that would create a compound B-Tree index on name.surname and city.zip.

```
{
  "name"     : "PERSON_NAME_INDEX",
  "unique"   : false,
  "fields":
  [
     {
      "path"       : "name.surname",
      "datatype"   : "string",
      "maxLength" : 100,
      "order"      : "asc"
    },{
       "path"       : "city.zip",
      "datatype"   : "number",
      "order"      : "desc"
    }
  ]
}
```

The following index specification would be used to create an inverted-list index on a JSON collection.

```
{
  "name"     : "JSON_PATH_INDEX",
  "unique"   : false,
  "language" : "english",

}
```

Creating an Index using SODA for Java

With SODA for Java, search operations are constructed using the OracleCollectionAdmin class. The OracleCollectionAdmin object is obtained using the collection's admin method. The createIndex method provided by OracleCollectionAdmin class is used to create the index. The index specification is passed to the createIndex method as an OracleDocument object. The following example shows creating an index using SODA for Java:

```
    public void createIndex(OracleConnection conn, String collectionName, String indexSpec)
    throws OracleException {
        OracleRDBMSClient client = new OracleRDBMSClient();
        OracleDatabase database = client.getDatabase(conn);
        OracleCollection collection = database.openCollection(collectionName);
        OracleCollectionAdmin admin = collection.admin();
        admin.createIndex(database.createDocumentFromString(indexSpec));
    }
```

Creating an Index using SODA for REST

Using SODA for REST, an index is created by performing a POST operation on the collection and specifying parameter action=index. The index specification document is supplied as the body of the POST operation. The following example shows how to create an index using SODA for REST's "Create Index" service. File indexSpec.json contains the index specification, and forms the body of the POST request.

```
curl --digest -X POST   --write-out "%{http_code}\n" -u SCOTT:tiger -H "Accept: application/json" -H "Content-type:
application/json" --upload-file indexSpec.json http://localhost:8080/DBJSON/SCOTT/Collection1?action=index
```

Invoking SODA for REST from JavaScript.

Since SODA for REST is based on HTTP it is very easy create a JavaScript application that uses SODA for REST to work with JSON documents stored in an Oracle Document store. When a JavaScript application wants to invoke a REST service, such as those provided by SODA for REST, it does so by using the XMLHTTPRequest object that is provided for this purpose. The XMLHTTPRequest object is supported in all modern JavaScript environments and is defined by the W3C recommendation found at http://www.w3.org/TR/XMLHttpRequest.

A well written JavaScript application will take advantage of the asynchronous processing capabilities of the XMLHttpRequest object. Asynchronous processing ensures that the calling application does not have to wait for the request to be processed, freeing it to perform other tasks until the results of the service request are available.

Asynchronous processing is triggered by marking the operation as asynchronous and supplying a callback function via the XMLHTTPRequest object's onreadystatechange property. The callback function is a JavaScript function that will automatically be invoked by the JavaScript engine, once the XMLHTTPRequest object's readyState property is set to 4, indicating that the HTTP response is available for processing.

The role of the callback is to process the HTTP response generated by the server. Program logic that is dependent on the operation having completed needs to be initiated from this function.

A simple JavaScript procedure that would invoke SODA for REST's "Create Collection" service is shown below.

```
function createCollection(collectionName, collectionProperties, callback) {

    var serializedPayload = null;

    if ((typeof collectionProperties === "object") & (collectionProperties != null)) {
      serializedPayload = JSON.stringify(collectionProperties);
    }

    var URL = servletPath + collectionName;

    var XHR = new XMLHttpRequest();
    XHR.open ("PUT", URL, true);
    XHR.onreadystatechange= function() {
                              if (XHR.readyState==4) {
                               callback(XHR,URL,collectionName)
                              }
                            }
    XHR.setRequestHeader("Content-type","application/json");
    XHR.send(serializedPayload);
}
```

The procedure takes 3 arguments, the name of the collection, the collection properties object, and the callback function. The logic of the procedure is as follows

» Check if a Collection Properties object has been supplied, and if so use JavaScript's built in JSON object to create a serialized representation of the collection properties object.

» Uses the supplied collection name to construct the URL that will be the target of the put operation.

» Creates an XMLHTTPRequest object, which will be used to invoke the service.

» Use the XMLHTTPRequest's open method to request an asynchronous PUT operation on the target URL.

» Configure the callback function. In this example the callback expects to be passed 3 arguments, which are the XMLHTTPRequest Object, the URL and the Collection Name.

» Set the "Content-type" header of the request to "application/json"Invoke the XMLHTTPRequest's send method, supplying the serialized representation of the Collection Properties object as the body of the HTTP request.

A simple JavaScript procedure that would perform a “create document” operation is shown below

```
function createDocument(collectionName, object, callback) {
    var jsonDocument = JSON.stringify(object);
    var URL = servletPath + collectionName;
    var XHR = new XMLHttpRequest();
    XHR.open ("POST", URL, true);
    XHR.onreadystatechange= function() {
                              if (XHR.readyState==4) {
                               callback(XHR,URL)
                              }
                            }
    XHR.setRequestHeader("Content-type","application/json");
    XHR.send(jsonDocument);
}
```

The procedure takes 3 arguments, the name of the collection, the object to be persisted, and a callback function. The logic of the procedure is as follows

» Use JavaScript’s built in JSON object to create a serialized representation of the object to be persisted

» Uses the supplied collection name to construct the URL that will be the target of the put operation.

» Creates an XMLHTTPRequest object, which will be used to make the HTTP request.

» Use the XMLHTTPRequest’s open method to request an asynchronous POST operation on the target URL.

» Configure the callback function. In this case the callback expects to be passed 2 arguments, which are the XMLHTTPRequest Object and the URL.

» Set the “Content-type” header of the request to “application/json”

» Invoke the XMLHTTPRequest’s send method, supplying the serialized representation of the object as the body of the HTTP request.

A simple callback function that would log the ID of each document to the console is shown below

```
function PostCallback(XHR, URL) {
  var result = JSON.parse(XHR.responseText)
  for (int i=0; i < result.count; i++) {
    console.log("Document [" + i+1 + "]: ID = " + result.items[i].id);
  }
}
```

Support for other API’s and Development Environments

Though this whitepaper has focused on using document store APIs to access JSON documents, these documents are also completely accessible using SQL-based APIs. JSON content can be inserted, accessed, and updated using SQL-based drivers for all popular programming environments, including Java, C, and .NET, as well as from popular scripting frameworks such as PHP, Ruby, Python, and PERL. A developer could use JSON to build a schemaless application while using SQL interfaces — a useful approach for hybrid applications that use both relational and JSON data.

Create Tables for JSON using SQL

JSON documents can also be stored in any normal table. The following SQL statement creaties a table which can be used to store JSON documents.

```
create table J_PURCHASEORDER (
  ID             RAW(16) NOT NULL,
  DATE_LOADED    TIMESTAMP(6) WITH TIME ZONE,
  PO_DOCUMENT    CLOB CHECK (PO_DOCUMENT IS JSON)
)
/
```

This statement creates a very simple table, called **PURCHASEORDER**. The table has a column **PO_DOCUMENT** of type CLOB. The **IS JSON** constraint is applied to the column **PO_DOCUMENT** ensuring that the column can store only well formed JSON documents.

Inserting JSON content using SQL

Since the container data types used for JSON content are standard SQL data types, JSON documents can be inserted just like any other character content. It is interesting to consider the use of the IS JSON constraint when loading data from data sources where all the source documents are not necessarily valid JSON. Using the IS JSON constraint in the where clause makes it possible filter the source data, ensuring that only valid JSON is processed. An example of this is shown below

```
insert into "MyCollection" (ID,CREATED_ON,LAST_MODIFIED,VERSION,JSON_DOCUMENT)
select SYS_GUID(), SYSTIMESTAMP, SYSTIMESTAMP, ‘0’, JSON_DOCUMENT
  from DUMP_FILE_CONTENTS
 where PO_DOCUMENT IS JSON
/
```

This example loads content into the collection table from the external table shown earlier. The IS JSON constraint is used to ensure the operation does not fail if any of source documents are not valid JSON. The built in SQL functions SYS_GUID() and SYSTIMESTAMP are used to provide appropriate values for the columns ID and CREATED and LAST_MODIFIED. Any invalid documents in the source table can be identified using a separate SELECT statement that uses the IS NOT JSON constraint to filter the contents of the table.

Indexing collections using SQL

The following example shows how to create a unique functional B-Tree index on the value of PONumber key using the SQL/JSON operator JSON_VALUE and a JSON Path expression.

```
create unique index PO_NUMBER_IDX
    on J_PURCHASEORDER(
         JSON_VALUE(
           PO_DOCUMENT ,'$.PONumber' returning NUMBER(10) ERROR ON ERROR
         )
      )
/

Index created.
```

The following statement creates an inverted list index on the collection of JSON documents stored in column PO_DOCUMENT in table J_PURCHASEORDER.

```
create index PO_DOCUMENT_INDEX
    on J_PURCHASEORDER(PO_DOCUMENT )
       indextype is ctxsys.context
       parameters('section group CTXSYS.JSON_SECTION_GROUP SYNC (ON COMMIT)')
/

Index created.
```

The index is created using the predefined index preference: CTXSYS.JSON_SECTION_GROUP. This has been optimized for JSON document indexing. The SYNC (ON COMMIT) ensures that inserts and update operations are indexed at commit time.

A New Way to Develop Applications with Oracle Database 12c

With the APIs that are built upon its JSON capabilities, Oracle Database 12c delivers full support for developing document store applications. For those of you familiar with NoSQL style development you can see how the new SODA approach to development delivers all the benefits associated with schema-less development along with all the benefits associated with the robust Oracle Database.

If you are familiar with SQL-based applications then you might see, through the examples above, that Oracle Database 12c enables a different application-development style.

Developers can use Oracle Database to build schemaless applications. When data is stored in JSON documents, developers do not require assistance from a DBA when they develop, deploy or modify their applications. Consequently, they can release new versions of their applications more rapidly than when using a relational schema-based development model.

The addition of document store capabilities to Oracle Database provides a new development model. But new document store applications benefit from the entire set of core Oracle Database capabilities. They can rely on the highly available, secure, and scalable infrastructure of Oracle Database.