8

Working with SODA Collections in MLE JavaScript Code

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

SODA APIs exist for different programming languages and include support for MLE JavaScript. SODA APIs are *document-centric*. You can use any SODA implementation to perform create, read, update, and delete (CRUD) operations on documents of nearly any kind (including video, image, sound, and other binary content). You can also use any SODA implementation to query the content of JavaScript Object Notation (JSON) documents using pattern-matching: query-by-example (QBE). CRUD operations can be driven by document keys or by QBEs.

This chapter covers JavaScript in the database, based on Multilingual Engine (MLE) as opposed to the client-side `node-oracledb` driver. Whenever JavaScript is mentioned in this chapter it implicitly refers to MLE JavaScript.

Note:

In order to use the MLE SODA API, the `COMPATIBLE` initialization parameter must be set to `23.0.0`.

See Also:

Oracle Database Introduction to Simple Oracle Document Access (SODA) for a complete introduction to SODA

Topics

- High-Level Introduction to Working with SODA for In-Database JavaScript
  The SODA API is part of the MLE JavaScript SQL driver. Interaction with collections and documents requires you to establish a connection with the database first, before a SODA database object can be obtained.
- SODA Objects
  Objects used with the SODA API.
- Using SODA for In-Database JavaScript
  How to access SODA for In-Database JavaScript is described, as well as how to use it to perform create, read (retrieve), update, and delete (CRUD) operations on collections.

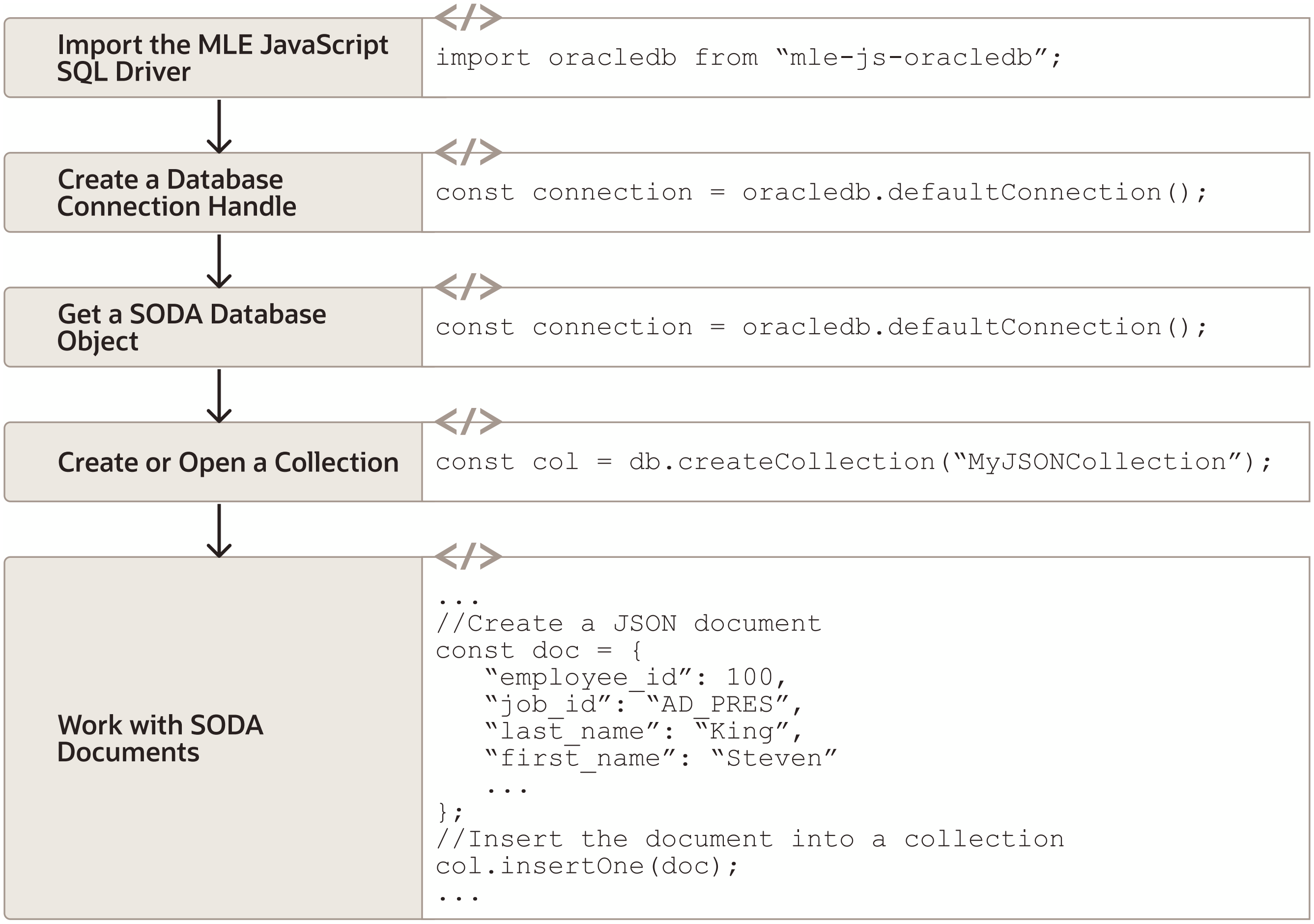
High-Level Introduction to Working with SODA for In-Database JavaScript

The SODA API is part of the MLE JavaScript SQL driver. Interaction with collections and documents requires you to establish a connection with the database first, before a SODA database object can be obtained.

The SODA database is the top-level abstraction object when working with the SODA API.

Figure 8-1 demonstrates the standard control flow.

Figure 8-1 SODA for In-Database JavaScript Basic Workflow


Import the MLE JavaScript SQL Driver
Create a Database Connection Handle
Get a SODA Database Object
Create or Open a Collection
Work with SODA Documents

```
import oracledb from “mle-js-oracledb”;
```

```
const connection = oracledb.defaultConnection();
```

```
const connection = oracledb.defaultConnection();
```

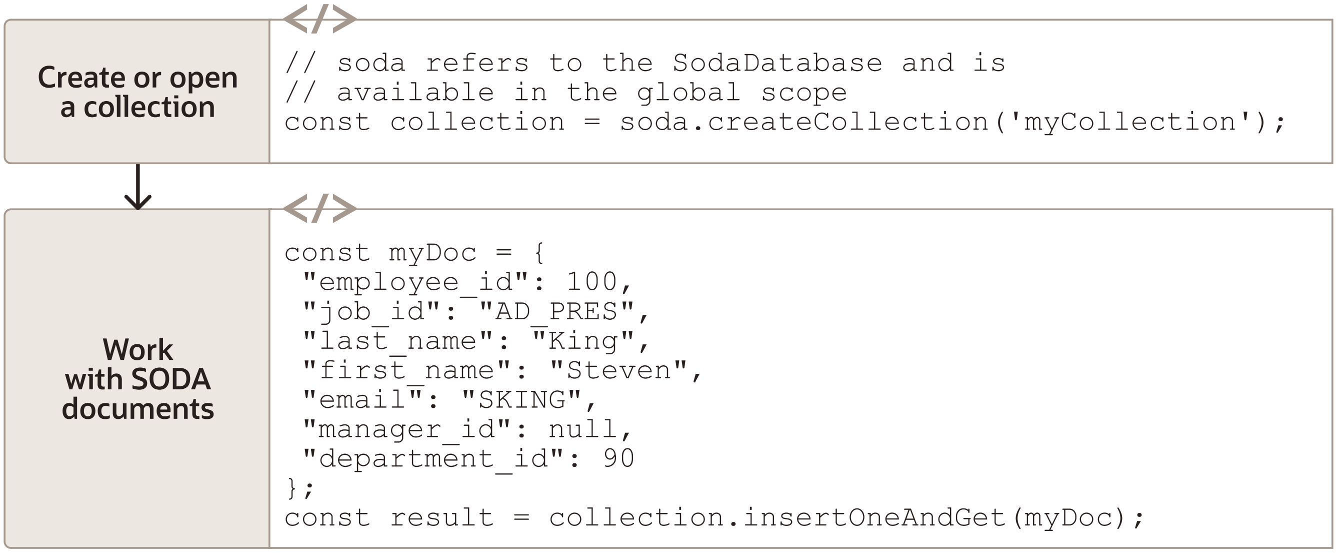
```
const col = db.createCollection(“MyJSONCollection”);
```

```
...
//Create a JSON document
const doc = {
   “employee_id”: 100,
   “job_id”: “AD_PRES”,
   “last_name”: “King”,
   “first_name”: “Steven”
   ...
};
//Insert the document into a collection
col.insertOne(doc);
...
```

Applications that aren't ported from client-side Node.js or Deno can benefit from coding aids available in the MLE JavaScript SQL driver, such as a number of frequently used variables that are available in the global scope. For a complete list of available global variables and types, see Working with the MLE JavaScript Driver.

For SODA applications the most important global variable is the `soda` object, which represents the `SodaDatabase` object. The availability of the `soda` object in the global scope reduces the need for writing boilerplate code. In this case the workflow can be simplified, as in Figure 8-2.

Figure 8-2 SODA for In-Database JavaScript Simplified Workflow



```
// soda refers to the SodaDatabase and is
// available in the global scope
const collection = soda.createCollection('myCollection');
```

```
const myDoc = {
 "employee_id": 100,
 "job_id": "AD_PRES",
 "last_name": "King",
 "first_name": "Steven",
 "email": "SKING",
 "manager_id": null,
 "department_id": 90
};
const result = collection.insertOneAndGet(myDoc);
```

> **Note:**
>
> If you are running your JavaScript code in a restricted execution context, you cannot use the SODA API. For more information about restricted execution contexts, see About Restricted Execution Contexts.

SODA Objects

Objects used with the SODA API.

The following objects are at the core of the SODA API:

- **`SodaDatabase`:** The top-level object for SODA operations. This is acquired from an Oracle Database connection or directly available from the global scope as the `soda` object. A SODA database is an abstraction, allowing access to SODA collections in that SODA database, which then allow access to documents in those collections. A SODA database is analogous to an Oracle Database user or schema. A collection is analogous to a table. A document is analogous to a table row with one column for a unique document key, a column for the document content, and other columns for various document attributes. With the MLE JavaScript SQL driver, the `soda` object is available as a global variable, which represents the `SodaDatabase` object and reduces the need for writing boilerplate code.
- **`SodaCollection`:** Represents a collection of SODA documents. By default, collections allow JSON documents to be stored, and they add a default set of metadata to each document. This is recommended for most users. However, optional metadata can set various details about a collection, such as its database storage, whether it should track version and time stamp document components, how such components are generated, and what document types are supported. Most users do not need to provide custom metadata.
- **`SodaDocument`:** Represents a document. Typically, the document content will be JSON. The document has properties including the content, a key, timestamps, and the media type. By default, document keys are automatically generated.

When working with collections and documents stored therein, you will make use of the following objects:

- **`SodaDocumentCursor`:** A cursor object representing the result of the `getCursor()` method from a `find()` operation. It can be iterated over to access each `SodaDocument`.
- **`SodaOperation`:** An internal object used with `find()` to perform read and write operations on documents. Chained methods set properties on a `SodaOperation` object which is then used by a terminal method to find, count, replace, or remove documents. This is an internal object that should not be directly accessed.

> **See Also:**
>
> Server-Side JavaScript API Documentation for information about using SODA objects with `mle-js-oracledb`

Using SODA for In-Database JavaScript

How to access SODA for In-Database JavaScript is described, as well as how to use it to perform create, read (retrieve), update, and delete (CRUD) operations on collections.

This section describes SODA for MLE JavaScript. Code snippets in this section are sometimes abridged for readability. Care has been taken to ensure that JavaScript functions are listed in their entirety, but they aren't runnable on their own. Embedding the function definition into a JavaScript module and importing the MLE JavaScript SQL driver will convert these code examples to valid JavaScript code for Oracle Database 23ai.

Topics

- Getting Started with SODA for In-Database JavaScript
  How to access SODA for In-Database JavaScript is described, as well as how to use it to create a database collection, insert a document into a collection, and retrieve a document from a collection.
- Creating a Document Collection with SODA for In-Database JavaScript
  How to use SODA for In-Database JavaScript to create a new document collection is explained.
- Opening an Existing Document Collection with SODA for In-Database JavaScript
  You can use the method `SodaDatabase.openCollection()` to open an existing document collection or to test whether a given name names an existing collection.
- Checking Whether a Given Collection Exists with SODA for In-Database JavaScript
  You can use `SodaDatabase.openCollection()` to check for the existence of a given collection. It returns `null` if the collection argument does not name an existing collection; otherwise, it opens the collection having that name.
- Discovering Existing Collections with SODA for In-Database JavaScript
  You can use `SodaDatabase.getCollectionNames()` to fetch the names of all existing collections for a given `SodaDatabase` object.
- Dropping a Document Collection with SODA for In-Database JavaScript
  You use `SodaCollection.drop()` to drop an existing collection.
- Creating Documents with SODA for In-Database JavaScript
  Creation of documents by SODA for In-Database JavaScript is described.

- Inserting Documents into Collections with SODA for In-Database JavaScript
  `SodaCollection.insertOne()` or a related call such as `sodaCollection.insertOneAndGet()` offers convenient ways to add documents to a collection. These methods create document keys automatically, unless the collection is configured with client-assigned keys and the input document provides the key, which is not recommended for must users.
- Saving Documents into Collections with SODA for In-Database JavaScript
  You use `SodaCollection.save()` and `saveAndGet()` to save documents into collections.
- SODA for In-Database JavaScript Read and Write Operations
  The primary way you specify read and write operations (other than insert and save) is to use methods provided by the `SodaOperation` class. You can chain together `SodaOperation` methods to specify read or write operations against a collection.
- Finding Documents in Collections with SODA for In-Database JavaScript
  To find documents in a collection, you invoke `SodaCollection.find()`. It creates and returns a `SodaOperation` object which is used via method chaining with nonterminal and terminal methods.
- Replacing Documents in a Collection with SODA for In-Database JavaScript
  To replace the content of one document in a collection with the content of another, you start by looking up the document to be modified using its key. Because `SodaOperation.key()` is a nonterminal operation, the easiest way to replace the contents is to chain `SodaOperation.key()` to `SodaOperation.replaceOne()` or `SodaOperation.replaceOneAndGet()`.
- Removing Documents from a Collection with SODA for In-Database JavaScript
  Removing documents from a collection is similar to replacing. The first step is to perform a lookup operation, usually based on the document's key or by using a search expression in `SodaOperation.filter()`. The call to `SodaOperation.remove()` is a terminal operation, in other words the last operation in the chain.
- Indexing the Documents in a Collection with SODA for In-Database JavaScript
  Indexes can speed up data access, regardless of whether you use the NoSQL style SODA API or a relational approach. You index documents in a SODA collection using `SodaCollection.createIndex()`. Its `IndexSpec` parameter is a textual JSON index specification.
- Getting a Data Guide for a Collection with SODA for In-Database JavaScript
  A data guide is a summary of the structural and type information contained in a set of JSON documents. It records metadata about the fields used in those documents. They provide great insights into JSON documents and are invaluable for getting an overview of a data set.
- Handling Transactions with SODA for In-Database JavaScript
  Unlike the client-side JavaScript SQL driver, the MLE JavaScript SQL driver does not provide an `autoCommit` feature. You need to commit or roll your transactions back, either in the PL/SQL layer in case of module calls, or directly in the JavaScript code by calling `connection.commit()` or `connection.rollback()`.
- Creating Call Specifications Involving the SODA API
  Earlier in this chapter, in the section *Getting Started with SODA for In-Database JavaScript*, an example showing how to invoke the MLE SODA API using an inline call specification is included. The following short example demonstrates how to use SODA in MLE modules.

Getting Started with SODA for In-Database JavaScript

How to access SODA for In-Database JavaScript is described, as well as how to use it to create a database collection, insert a document into a collection, and retrieve a document from a collection.

Before you can get started working with SODA for MLE JavaScript, the account used for storing collections (in this case, `emily`) must be granted the `SODA_APP` roles, either directly or using the `DB_DEVELOPER_ROLE`:

```
grant soda_app to emily
```

Accessing SODA functionality requires the use of the MLE JavaScript SQL driver. Because the database session exists by the time the code is invoked, no additional connection handling is necessary. Example 8-1 demonstrates how to:

- Create a SODA collection,
- Insert a JSON document into it, and
- Iterate over all SODA Documents in the collection, printing their contents on screen

Each concept presented by Example 8-1 - creating collections, adding and modifying documents, and dropping collections - is addressed in more detail later in this chapter.

Example 8-1 SODA with MLE JavaScript General Workflow

This example demonstrates the general workflow using SODA collections with MLE JavaScript. Instead of using an MLE module, the example simplifies the process by implementing an inline call specification.

```
CREATE OR REPLACE PROCEDURE intro_soda(
    "dropCollection" BOOLEAN
) AUTHID CURRENT_USER
AS MLE LANGUAGE JAVASCRIPT
{{

  // use the soda object, available in the global scope instead of importing
  // the mle-js-oracledb driver, getting the default connection and extracting
  // the SodaDatabase from it
  const col = soda.createCollection("MyCollection");

  // create a JSON document (based on the HR.EMPLOYEES table for the employee
with id 100)
  const doc = {
    "_id" : 100,
    "job_id" : "AD_PRES",
    "last_name" : "King",
    "first_name" : "Steven",
    "email" : "SKING",
    "manager_id" : null,
    "department_id" : 90
  };

  // insert the document into collection
  col.insertOne(doc);
  // find all documents in the collection and print them on screen
  // use a cursor to iterate over all documents in the collection
  const c = col.find()
            .getCursor();

  let resultDoc;

  while (resultDoc = c.getNext()){
    const content = resultDoc.getContent();
    console.log(`
      ------------------------------------------
      key:           ${resultDoc.key}
      content (select fields):
      - _id:         ${content._id}
      - job_id:      ${content.job_id}
      - name:        ${content.first_name} ${content.last_name}
      version:       ${resultDoc.version}
      media type:    ${resultDoc.mediaType}`
    );
  }

  // it is very important to close the SODADocumentCursor to free resources
  c.close();

  // optionally drop the collection
  if (dropCollection){
    // there is no auto-commit, the outstanding transaction must be
    // finished before the collection can be dropped
    session.commit();
    col.drop();
  }
}};
/
```

You can try the code by executing the procedure using your favorite IDE. Here is an example of the results of calling the `intro_soda` procedure:

```
BEGIN
  intro_soda(true);
END;
/
```

Result:

```
------------------------------------------
key:           03C202
content (select fields):
- _id:         100
- job_id:      AD_PRES
- name:        Steven King
version:       17EF0F3C102653DDE063DA464664399C
media type:    application/json
```

```
PL/SQL procedure successfully completed.
```

Creating a Document Collection with SODA for In-Database JavaScript

How to use SODA for In-Database JavaScript to create a new document collection is explained.

Collections allow you to logically group documents. Before a collection can be created or accessed, a few more steps must be completed unless you make use of the global `soda` object. Begin by creating a connection object. The connection object is the starting point for all SODA interactions in the MLE JavaScript module:

```
// get a connection handle to the database session
const connection = oracledb.defaultConnection();
```

Once the connection is obtained, you can use it to call `Connection.getSodaDatabase()`, a prerequisite for creating the collection:

```
// get a SODA database
const db = connection.getSodaDatabase();
```

With the SODA database available, the final step is to create the collection. Note that collection names are case-sensitive:

```
// Create a collection with the name "MyCollection".
// This creates a database table, also named "MyCollection",
// to store the collection. If a collection with the same name
// exists, it will be opened
const col = db.createCollection("MyCollection");
```

The preceding statement creates a collection that, by default, allows JSON documents to be stored. If the collection name passed to `SodaDatabase.createCollection()` is that of an existing collection, it will simply be opened. You can alternatively open a known, existing collection using `SodaDatabase.openCollection()`.

Unless custom metadata is provided to `SodaDatabase.createCollection()` (which is not recommended), *default collection metadata* will be supplied. The default metadata has the following characteristics:

- Each document in the collection has these components:
  - Key
  - Content
  - Version
- The collection can store only JSON documents.
- Document keys and version information are generated automatically.

Optional collection metadata can be provided to the call to `createCollection()`, however, the default collection configuration is recommended in most cases.

If a collection with the same name already exists, it is simply opened and its object is returned. If custom metadata is passed to the method and does not match that of the existing collection, the collection is not opened and an error is raised. To match, all metadata fields must have the same values.

See Also:

Oracle Database Introduction to Simple Oracle Document Access (SODA) for more details about collection metadata, including custom metadata.

Opening an Existing Document Collection with SODA for In-Database JavaScript

You can use the method `SodaDatabase.openCollection()` to open an existing document collection or to test whether a given name names an existing collection.

Example 8-2   Opening an Existing Document Collection

This example opens the collection named `collectionName`. It is very important to check that the collection object returned by `SodaDatabase.openCollection()` is not `null`. Rather than throwing an error, the method will return a `null` value should the requested collection not exist.

```
export function openCollection(collectionName) {

    // perform a lookup. If a connection cannot be found by that
    // name no exception nor error are thrown, but the resulting
    // collection object will be null
    const col = soda.openCollection(collectionName);
    if (col === null) {
        throw new Error(`No such collection ${collectionName}`);
    }

    // do something with the collection
}
```

Checking Whether a Given Collection Exists with SODA for In-Database JavaScript

You can use `SodaDatabase.openCollection()` to check for the existence of a given collection. It returns `null` if the collection argument does not name an existing collection; otherwise, it opens the collection having that name.

In Example 8-2, if `collectionName` does not name an existing collection then `col` is assigned the value `null`.

Discovering Existing Collections with SODA for In-Database JavaScript

You can use `SodaDatabase.getCollectionNames()` to fetch the names of all existing collections for a given `SodaDatabase` object.

If the number of collections is very large, you can limit the number of names returned. Additionally, the lookup can be limited to collections starting with a user-defined string as demonstrated by Example 8-4.

Example 8-3   Fetching All Existing Collection Names

This example prints the names of all existing collections using the method `getCollectionNames()`.

```
export function printCollectionNames(){
  // loop over all collections in the current user's schema
  const allCollections = soda.getCollectionNames();
  for (const col of allCollections){
    console.log(`- ${col}`);
  }
}
```

Example 8-4   Filtering the List of Returned Collections

This example limits the results of `getCollectionNames()` by only printing the names of collections that begin with a user-defined string, `startWith`.

```
export function printSomeCollectionNames(numHits, startWith) {

  // loop over all collections in the current schema, limited
  // to those that start with a specific character sequence and
  // a maximum number of hits returned
  const allCollections = soda.getCollectionNames(
    {
      limit: numHits,
      startsWith: startWith
    }
  );
  for (const col of allCollections){
    console.log(`-${col}`);
  }
}
```

Dropping a Document Collection with SODA for In-Database JavaScript

You use `SodaCollection.drop()` to drop an existing collection.

Caution:

Do *not* use SQL to drop the database *table* that underlies a collection. Dropping a *collection* involves more than just dropping its database table. In addition to the documents that are stored in its table, a collection has *metadata*, which is also persisted in Oracle Database. Dropping the table underlying a collection does *not* also drop the collection metadata.

Note:

Day-to-day use of a typical application that makes use of SODA does not require that you drop and re-create collections. But if you need to do that for any reason then this guideline applies.

Do *not* drop a collection and then re-create it with *different metadata* if there is any application running that uses the collection in any way. Shut down any such applications before re-creating the collection, so that all live SODA objects are released.

There is no problem just dropping a collection. Any read or write operation on a dropped collection raises an error. And there is no problem dropping a collection and then re-creating it with the same metadata. But if you re-create a collection with different metadata, and if there are any live applications using SODA objects, then there is a risk that a stale collection is accessed, and *no error is raised* in this case.

In SODA implementations that allow collection metadata caching, such as SODA for Java, this risk is increased if such caching is enabled. In that case, a (shared or local) cache can return an entry for a stale collection object even if the collection has been dropped.

Note:

Commit all writes to a collection before using `SodaCollection.drop()`. For the method to succeed, all uncommitted writes to the collection must first be committed. Otherwise, an exception is raised.

Example 8-5 Dropping a Collection

This example shows how to drop a collection.

```
export function openAndDropCollection(collectionName) {

    // look the collection up
    const col = soda.openCollection(collectionName);
    if (col === null) {
        throw new Error (`No such collection ${collectionName}`);
    }

    // drop the collection - POTENTIALLY DANGEROUS
    col.drop();
}
```

Creating Documents with SODA for In-Database JavaScript

Creation of documents by SODA for In-Database JavaScript is described.

The `SodaDocument` class represents SODA documents. Although its focus is on JSON documents, it supports other content types as well. A `SodaDocument` stores both the actual document's contents as well as metadata.

JavaScript is especially well-suited to work with JSON by design, giving it an edge over other programming languages.

Here is an example of a simple JSON document:

```
// Create a JSON document (based on the HR.EMPLOYEES table for employee 100)
const doc = {
    "_id": 100,
    "job_id": "AD_PRES",
    "last_name": "King",
    "first_name": "Steven",
    "email": "SKING",
    "manager_id": null,
    "department_id": 90
};
```

> **Note:**
>
> In SODA, JSON content must conform to RFC 4627.

`SodaDocument` objects can be created in three ways:

- As a result of `sodaDatabase.createDocument()`. This is a proto-`SodaDocument` object usable for SODA insert and replace methods. The `SodaDocument` will have content and media type components set.
- As a result of a read operation from the database, such as calling `sodaOperation.getOne()`, or from `sodaDocumentCursor.getNext()` after a `sodaOperation.getCursor()` call. These return complete `SodaDocument` objects containing the document content and attributes, such as media type.
- As a result of `sodaCollection.insertOneAndGet()`, `sodaOperation.replaceOneAndGet()`, or `sodaCollection.insertManyAndGet()` methods. These return `SodaDocuments` that contain all attributes except the document content itself. They are useful for finding document attributes such as system generated keys, and versions of new and updated documents.

A document has these components:

- Key
- Content
- Version
- Media type ("application/json" for JSON documents)

The document's content consists of all the fields representing the information the application needs to store plus an _id field. This field is either provided by the user or injected by Oracle if omitted. If omitted, Oracle adds a random value with a length of 12 bytes.

The document's key is a hex-encoded representation of the document's _id column. It is automatically calculated and cannot be changed. The key is often used when building operations such as finds, replaces, and removes, with key() and keys(...) methods. These operations are discussed in later sections.

Example 8-6   Creating SODA Documents

```
export function createJSONDoc() {

  // define the document's contents
  const payload = {
    "_id ": 100,
    "job_id": "AD_PRES",
    "last_name": "King",
    "first_name": "Steven",
    "email": "SKING",
    "manager_id": null,
    "department_id": 90
  };

  // Create a SODA document.
  // Notice that neither key nor version are populated. They will be as soon
  // as the document is inserted into a collection and retrieved.
  const doc = soda.createDocument(payload);
  console.log(`
    ------------------------------------------
    SODA Document using default key
    content (select fields):
    - _id          ${doc.getContent()._id}
    - job_id       ${doc.getContent().job_id}
    - first name   ${doc.getContent().first_name}
    media type:    ${doc.mediaType}
    version   :    ${doc.version}
    key            ${doc.key}`
  );
}
```

Creating SodaDocument instances as shown in this example is the exception rather than the norm. In most cases, developers use SodaCollection.insertOne() or SodeCollection.insertOneAndGet(). The use of SodaCollection.insertOne() is demonstrated in Example 8-7. Multiple documents can be created using sodaCollection.insertMany().

Inserting Documents into Collections with SODA for In-Database JavaScript

`SodaCollection.insertOne()` or a related call such as `sodaCollection.insertOneAndGet()` offers convenient ways to add documents to a collection. These methods create document keys automatically, unless the collection is configured with client-assigned keys and the input document provides the key, which is not recommended for must users.

`SodaCollection.insertOne()` simply inserts the document into the collection, whereas `SodaCollection.insertOneAndGet()` additionally returns a result document. The resulting document contains the document key and any other generated document components, except for the actual document's content (this is done to improve performance).

Both methods automatically set the document's version, unless the collection has been created with custom metadata. Custom metadata might not include all the default metadata. When querying attributes not defined by the collection a null value is returned.

> **Note:**
>
> If you want the input document to *replace* the existing document instead of causing an exception, see Saving Documents into Collections with SODA for In-Database JavaScript.

Example 8-7 Inserting a SODA Document into a Collection

This example demonstrates how to insert a document into a collection using `SodaCollection.insertOne()`.

```
export function insertOneExample() {

  // define the document's contents
  const payload = {
    "_id": 100,
    "job_id": "AD_PRES",
    "last_name": "King",
    "first_name": "Steven",
    "email": "SKING",
    "manager_id": null,
    "department_id": 90
  };

  // create or open the collection to hold the document
  const col = soda.createCollection("MyCollection");

  col.insertOne(payload);
}
```

Example 8-8   Inserting an Array of Documents into a Collection

This example demonstrates the use of `SodaCollection.insertMany()` to insert multiple documents with one command. The example essentially translates the relational table `HR.employees` into a collection.

```
export function insertManyExample() {

  // select all records from the hr.employees table into an array
  // of JavaScript objects in preparation of a call to insertMany
  const result = session.execute(
    `SELECT
      employee_id "_id",
      first_name "firstName",
      last_name "lastName",
      email "email",
      phone_number "phoneNumber",
      hire_date "hireDate",
      job_id "jobId",
      salary "salary",
      commission_pct "commissionPct",
      manager_id "managerId",
      department_id "departmentId"
     FROM
      hr.employees`,
     [],
     { outFormat: oracledb.OUT_FORMAT_OBJECT }
  );

  // create the collection and insert all employee records
  collection = soda.createCollection('employeesCollection');
  collection.insertMany(result.rows);

  // the MLE JavaScript SQL driver does not auto-commit
  session.commit();
}
```

Saving Documents into Collections with SODA for In-Database JavaScript

You use `SodaCollection.save()` and `saveAndGet()` to save documents into collections.

These methods are similar to methods `insertOne()` and `insertOneAndGet()` except that, if the collection is configured with client-assigned document keys, and the input document provides a key that already identifies a document in the collection, then the input document *replaces* the existing document. In contrast, methods `insertOne()` and `insertOneAndGet()` throw an exception in that case.

SODA for In-Database JavaScript Read and Write Operations

The primary way you specify read and write operations (other than insert and save) is to use methods provided by the `SodaOperation` class. You can chain together `SodaOperation` methods to specify read or write operations against a collection.

Nonterminal `SodaOperation` methods return the same object on which they are invoked, allowing them to be chained together.

A *terminal* `SodaOperation` method always appears at the end of a method chain to execute the operation.

> **Note:**
>
> A `SodaOperation` object is an internal object. You should not directly modify its properties.

Unless the SODA documentation for a method says otherwise, you can chain together any nonterminal methods and you can end the chain with any terminal method. However, not all combinations make sense. For example, it does not make sense to chain method `version()` together with a method that does not uniquely identify the document, such as `keys()`.

Table 8-1 Overview of Nonterminal Methods for Read Operations

Method	Description
`key()`	Find a document that has the specified document key.
`keys()`	Find documents that have the specified document keys.
`filter()`	Find documents that match a filter specification (a query-by-example expressed in JSON).
`version()`	Find documents that have the specified version. This is typically used with `key()`.
`headerOnly()`	Exclude document content from the result.
`skip()`	Skip the specified number of documents in the result.
`limit()`	Limit the number of documents in the result to the specified number.

Table 8-2 Overview of Terminal Methods for Read Operations

Method	Description
`getOne()`	Create and execute an operation that returns at most one document. For example, an operation that includes an invocation of nonterminal method `key()`.
`getCursor()`	Get a cursor over read operation results.
`count()`	Count the number of documents found by the operation.
`getDocuments()`	Gets an array of documents matching the query criteria.

Table 8-3 Overview of Terminal Methods for Write Operations

Method	Description
`replaceOne()`	Replace one document.
`replaceOneAndGet()`	Replace one document and return the result document.
`remove()`	Remove documents from a collection.

See Also:

- Node-oracledb Documentation for more details about the `SodaOperations` class.
- SODA Restrictions (Reference) for information about SODA restrictions.

Finding Documents in Collections with SODA for In-Database JavaScript

To find documents in a collection, you invoke `SodaCollection.find()`. It creates and returns a `SodaOperation` object which is used via method chaining with nonterminal and terminal methods.

To execute the query, obtain a cursor for its results by invoking `SodaOperation.getCursor()`. Then use the cursor to visit each document in the result list. This is illustrated by Example 8-1 and other examples. It is important not to forget to close the cursor, to save resources.

However, this is not the typical workflow when searching for documents in a collection. It is more common to chain multiple methods provided by the `SodaOperation` class together.

Example 8-9   Finding a Document by Key

This example shows how to look up a document by its key using the methods `find()`, `key()`, and `getOne()`.

```
export function findDocByKey(searchKey){

  const collectionName = 'MyCollection';

  // open the collection in preparation of a document lookup
  const col = soda.openCollection(collectionName);
  if (col === null){
    throw new Error(`${collectionName} does not exist`);
  }

  try{
    // perform a lookup of a document with the key provided as a
    // parameter to this function. Keys are like primary keys,
    // the lookup therefore can only return 1 document max
    const doc = col.find()
                   .key(searchKey)
                   .getOne();
    console.log(`
      document found for key ${searchKey}
      contents: ${doc.getContentAsString()}`
    );
  } catch(err){
      throw new Error(
        `error retrieving document with key ${searchKey} (${err})`
      );
  }
}
```

> **Note:**
>
> Keys need to be enclosed in quotation marks even if they should be in numeric format.

In case the search for a given key fails, the database throws an ORA-01403 (no data found) exception. It is good practice to handle exceptions properly. In this example, the caller of the function has the responsibility to ensure the error is trapped and dealt with according to the industry's best-known methods.

Example 8-10   Looking up Documents Using Multiple Keys

This example uses the methods `find()`, `keys()`, `getCursor()`, and `getNext()` to search for multiple keys provided in an array.

See Example 8-8 for details about how to create `employeesCollection`, used in this example.

```
export function findDocByKeys(searchKeys){

  if(!Array.isArray(searchKeys)){
    throw new Error('please provide an array of search keys');
  }

  // open a collection in preparation of a document lookup
  const col = soda.openCollection('employeesCollection');
  if (col === null){
    throw new Error('employeesCollection does not exist');
  }

  try{
    // perform a lookup of a set of documents using
    // the "keys" array provided
    const docCursor =
      col.find()
         .keys(searchKeys)
         .getCursor();

    let doc
      while((doc = docCursor.getNext())){
        console.log(`
          document found for key ${doc.key}
          contents: ${doc.getContentAsString()}`
        );
      }
      docCursor.close();
  } catch(err){
      // there is no error thrown if one/all of the keys aren't found
      // this error handler is generic
      throw new Error(
        `error retrieving documents with keys ${searchKeys} (${err})`
      );
  }
}
```

Rather than failing with an error, the `find()` operation simply doesn't return any data for a key not found in a collection. If none of the keys are found, nothing is returned.

Example 8-11 Using a QBE to Filter Documents in a Collection

This example uses `filter()` to locate documents in a collection. The nonterminal `SodaOperation.filter()` method provides a powerful way to filter JSON documents in a collection, allowing for complex document queries and ordering of JSON documents. Filter specifications can include comparisons, regular expressions, logical and spatial operators, among others.

The search expression defined in `filterCondition` matches all employees with an employee ID greater than `110` working in department `30`.

See Example 8-8 for details about how to create `employeesCollection`, used in this example.

```
export function findDocByFiltering(){

  // open a collection in preparation of a document
  // lookup. This particular collection contains all the
  // rows from the HR.employees table converted to SODA
  // documents.
  const col = soda.openCollection('employeesCollection');
  if(col === null){
    throw new Error(`employeesCollection does not exist`);
  }

  // find all employees with an employee_id > 100 and
  // last name beginning with M
  const filterCondition = {
    "$and": [
      { "lastName": { "$upper": { "$startsWith": "M" } } },
      { "_id": { "$gt": 100 } }
    ]
  };

  try{

    // perform the lookup operation using the QBE defined earlier
    const docCursor = col.find()
                         .filter(filterCondition)
                         .getCursor();
    let doc;
    while ((doc = docCursor.getNext())){
      console.log(`
        ------------------------------------
        document found matching the search criteria
        - key:           ${doc.key}
        - _id:           ${doc.getContent()._id}
        - name:          ${doc.getContent().lastName}`
      );
    }

    docCursor.close();
  } catch(err){
      throw new Error(`error looking up documents using a QBE: ${err}`);
  }
}
```

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

Example 8-12 Using skip() and limit() in a Pagination Query

If the number of rows becomes too large, you may choose to paginate and or limit the number of documents returned. This example demonstrates using `skip()` and `limit()` in this type of circumstance.

See Example 8-8 for details about how to create `employeesCollection`, used in this example.

```
export function paginationExample(){

  // open a collection in preparation of a document
  // lookup. This particular collection contains all the
  // rows from the HR.employees table converted to SODA
  // documents.
  const col = soda.openCollection('employeesCollection');
  if(col === null){
    throw new Error ('employeesCollection does not exist, aborting');
  }

  // find all employees with an employee_id > 100 and
  // last name beginning with E
  const filterCondition = {
    "$and": [
      { "lastName": { "$upper": { "$startsWith": "M" } } },
      { "_id": { "$gt": 100 } }
    ]
  };

  try{

    // perform the lookup operation using the QBE, skipping the first
    // 5 documents and limiting the result set to 10 documents
    const docCursor =
      col.find()
         .filter(filterCondition)
         .skip(5)
         .limit(10)
         .getCursor();
    let doc;
    while ((doc = docCursor.getNext())){
      console.log(`
        ------------------------------------
        document found matching the search criteria
        - key:          ${doc.key}
        - employee id:  ${doc.getContent().employeeId}`
      );
    }

    docCursor.close();
  } catch(err){
      throw new Error(
        `error looking up documents by QBE (${err})`
      );
  }
}
```

Example 8-13   Specifying Document Versions

This example uses the nonterminal `version()` method to specify a particular document version. This is useful for implementing optimistic locking, when used with the terminal methods for write operations.

See Example 8-8 for details about how to create `employeesCollection`, used in this example.

```
export function versioningExample(searchKey, version){

  // open a collection in preparation of a document
  // lookup. This particular collection contains all the
  // rows from the HR.employees table converted to SODA
  // documents.
  const col = soda.openCollection("employeesCollection");

  try{
    // perform a lookup of a document using the provided key and version
    const doc = col
      .find()
      .key(searchKey)
      .version(version)
      .getOne();
    console.log(`
      document found for key ${doc.key}
      contents: ${doc.getContentAsString()}`
    );
  } catch(err){
      throw new Error(
        `${err} during lookup. Key: ${searchKey}, version: ${version}`
      );
  }
}
```

If SODA cannot find the document matching the key and version tag, an `ORA-01403: no data found` error is thrown.

Example 8-14   Counting the Number of Documents Found

This example shows how to count the number of documents found in a collection using the `find()`, `filter()`, and `count()` methods. The `filter()` expression limits the result to all employees working in department `30`.

See Example 8-8 for details about how to create `employeesCollection`, used in this example.

```
export function countingExample(){

  // open a collection in preparation of a document
  // lookup. This particular collection contains all the
  // rows from the HR.employees table converted to SODA
  // documents.
  const col = soda.openCollection("employeesCollection");
  if(col === null){
    throw new Error('employeesCollection does not exist');
  }

  try{

    // perform a lookup operation identifying all employees working
    // in department 30, limiting the result to headers only
    const filterCondition = {"departmentId": 30};
    const numDocs = col.find()
                       .filter(filterCondition)
                       .count();
    console.log(`there are ${numDocs} documents matching the filter`);
  } catch(err){
      throw new Error(
        `No document found in 'employeesCollection' matching the filter`
      );
  }
}
```

Replacing Documents in a Collection with SODA for In-Database JavaScript

To replace the content of one document in a collection with the content of another, you start by looking up the document to be modified using its key. Because `SodaOperation.key()` is a nonterminal operation, the easiest way to replace the contents is to chain `SodaOperation.key()` to `SodaOperation.replaceOne()` or `SodaOperation.replaceOneAndGet()`.

`SodaOperation.replaceOne()` merely replaces the document, whereas `SodaOperation.replaceOneAndGet()` replaces it and provides the resulting new document to the caller.

The difference between `SodaOperation.replace()` and `SodaOperation.save()` is that the latter performs an insert in case the key doesn't already exist in the collection. The replace operation requires an existing document to be found by the lookup via the `SodaOperation.key()` method.

> **Note:**
>
> Some version-generation methods generate hash values of the document content. In such a case, if the document content does not change then neither does the version.

Example 8-15 Replacing a Document in a Collection and Returning the Result Document

This example shows how to replace a document in a collection, returning a reference to the changed document. Let's assume that employee 206 has been given a raise of 100 monetary units. Using the SODA API you can update the salary as follows:

```
export function replaceExample(){

  // open employeesCollection in preparation of the update
  const col = soda.openCollection('employeesCollection');
  if (col === null){
    throw new Error("'employeesCollection does not exist");
  }

  try{
    // look up employeeId 206 using a QBE and get the document.
    // Since the documents are inserted into the collection based
    // on the HR.employees table, it is certain that there is at
    // most 1 document with employeeId 206
    const employeeDoc = col
                        .find()
                        .filter({"_id": 206})
                        .getOne();

    // get the document's actual contents/payload
    employee = employeeDoc.getContent();

    // currently it is not possible to include the _id together with
    // the replacement payload. This means existing _id must be deleted.
    // The document, once replaced in the collection, will have its
    // _id injected from the target document
    delete employee_id;

    // increase the salary
    employee.salary += 100;

    // save the document back to the collection. Note that you need
    // to provide the document's key rather than a QBE or else an
    // ORA-40734: key for the document to replace must be specified
    // using the key attribute error will be thrown
    const resultDoc = col
                      .find()
                      .key(employeeDoc.key)
                      .replaceOneAndGet(employee);

    // print some metadata (note that content is not returned for
    // performance reasons)
    console.log(`Document updated successfully:
    - key:           ${resultDoc.key}
    - version:       ${resultDoc.version}`);

  } catch(err){
      console.log(`error modifying employee 206's salary: ${err}`);
  }
}
```

See Example 8-8 for details about how to create `employeesCollection`, used in this example.

> **Note:**
>
> Trying to read the changed contents will result in an error as the actual document's contents aren't returned, for performance reasons.

Removing Documents from a Collection with SODA for In-Database JavaScript

Removing documents from a collection is similar to replacing. The first step is to perform a lookup operation, usually based on the document's key or by using a search expression in `SodaOperation.filter()`. The call to `SodaOperation.remove()` is a terminal operation, in other words the last operation in the chain.

Example 8-16 Removing a Document from a Collection Using a Document Key

This example removes the document whose document key is `"100"`.

```
export function removeByKey(searchKey){

  // open MyCollection
  const col = soda.openCollection("MyCollection");
  if(col === null){
    throw new Error("'MyCollection' does not exist");
  }

  // perform a lookup of the document about to be removed
  // and ultimately remove it
  const result = col
                 .find()
                 .key(searchKey)
                 .remove();
  if(result.count === 0){
    throw new Error(
      `failed to delete a document with key ${searchKey}`
    );
  }
}
```

Example 8-17 Removing JSON Documents from a Collection Using a Filter

This example uses a filter to remove the JSON documents whose `department_id` is `70`. It then prints the number of documents removed.

```
export function removeByFilter(){

  // open the collection
  const col = soda.openCollection("MyCollection");
  if(col === null){
    throw new Error("'MyCollection' does not exist");
  }
  // perform a lookup based on a filter expression and remove
  // the documents matching the filter
  const result = col
                 .find()
                 .filter({"_id": 100})
                 .remove();

  console.log(`${result.count} documents deleted`);
}
```

Indexing the Documents in a Collection with SODA for In-Database JavaScript

Indexes can speed up data access, regardless of whether you use the NoSQL style SODA API or a relational approach. You index documents in a SODA collection using `SodaCollection.createIndex()`. Its `IndexSpec` parameter is a textual JSON index specification.

Existing indexes can be dropped using `SodaCollection.dropIndex()`.

A JSON search index is used for full-text and ad hoc structural queries, and for persistent recording and automatic updating of JSON data-guide information.

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 8-18 Creating a B-Tree Index for a JSON Field with SODA for In-Database JavaScript

This example creates a B-tree non-unique index for numeric field `department_id` of the JSON documents in collection `employeesCollection` (created in Example 8-8).

```
export function createBTreeIndex(){

  // open the collection
  const col = soda.openCollection('employeesCollection');
  if(col === null){
    throw new Error("'employeesCollection' does not exist");
  }

  // define the index...
  const indexSpec = {
    "name": "DEPARTMENTS_IDX",
    "fields": [
      {
        "path": "departmentId",
        "datatype": "number",
        "order": "asc"
      }
    ]
  };

  //... and create it
  try{
    col.createIndex(indexSpec);
  } catch(err){
      throw new Error(
        `could not create the index: ${err}`
      )
  }
}
```

Example 8-19 Creating a JSON Search Index with SODA for In-Database JavaScript

This example shows how to create a JSON search index for indexing the documents in collection `employeesCollection` (created in Example 8-8). It can be used for ad hoc queries and full-text search (queries using QBE operator `$contains`). It automatically accumulates and updates data-guide information about your JSON documents (aggregate structural and type information). The index specification has only field `name` (no *field* fields unlike the B-tree index in Example 8-18).

```
export function createSearchIndex(){

  // open the collection
  const col = soda.openCollection("employeesCollection");
  if(col === null){
    throw new Error("'employeesCollection' does not exist");
  }

  // define the index properties...
  cost indexSpec = {
    "name": "SEARCH_AND_DATA_GUIDE_IDX",
    "dataguide": "on",
    "search_on": "text_value"
  }

  //...and create it
  try{
    col.createIndex(indexSpec);
  } catch(err){
      throw new Error(
        `could not create the search and Data Guide index: ${err}`
      );
  }
}
```

If you only wanted to speed up ad hoc (search) indexing, you should specify a value of "off" for field dataguide. The dataguide indexing feature can be turned off in the same way if it is not required.

Example 8-20 Dropping an Index with SODA for In-Database JavaScript

This example shows how you can drop an existing index on a collection using SodaCollection.dropIndex() and the force option.

See Example 8-8 for details about how to create employeesCollection, used in this example.

```
export function dropIndex(indexName){

  // open the collection
  const col = soda.openCollection("employeesCollection");
  if(col === null){
    throw new Error("'employeesCollection' does not exist");
  }

  // drop the index
  const result = col.dropIndex(indexName, {"force": true});
  if(!result.dropped){
    throw `Could not drop SODA index '${indexName}'`;
  }
}
```

SodaCollection.dropIndex() returns a result object containing a single field: dropped. Its value is true if the index has been dropped, otherwise its value is false. The method succeeds either way.

An optional parameter object can be supplied to the method. Setting force to true forces dropping of a JSON index if the underlying Oracle Database domain index does not permit normal dropping.

Getting a Data Guide for a Collection with SODA for In-Database JavaScript

A data guide is a summary of the structural and type information contained in a set of JSON documents. It records metadata about the fields used in those documents. They provide great insights into JSON documents and are invaluable for getting an overview of a data set.

You can create a data guide using SodaCollection.getDataGuide(). To get a data guide in SODA, the collection must be JSON-only and have a JSON search index where the "dataguide" option is "on". Data guides are returned from sodaCollection.getDataGuide() as JSON content in a SodaDocument. The data guide is inferred from the collection as it currently is. As a collection grows and documents change, a new data guide is returned each subsequent time getDataGuide() is called.

Example 8-21 Generating a Data Guide for a Collection

This example gets a data guide for the collection employeesCollection (created in Example 8-8) using the method getDataGuide() and then prints the contents as a string using the method getContentAsString().

```
export function createDataGuide(){

  // open the collection
  const col = soda.openCollection('employeesCollection');
  if(col === null){
    throw new Error("'employeesCollection' does not exist");
  }

  // generate a Data Guide (requires the Data Guide index)
  const doc = col.getDataGuide();
  console.log(doc.getContentAsString());
}
```

The data guide can provide interesting insights into a collection, including all the fields and their data types. Although the Data Guide for `employeesCollection` may already be familiar to readers of this chapter, unknown JSON documents can be analyzed conveniently this way. The previous code block prints the following Data Guide to the screen:

```
{
  "type": "object",
  "o:length": 1,
  "properties": {
    "_id": {
      "type": "id",
      "o:length": 24,
      "o:preferred_column_name": "DATA$_id"
    },
    "email": {
      "type": "string",
      "o:length": 16,
      "o:preferred_column_name": "DATA$email"
    },
    "jobId": {
      "type": "string",
      "o:length": 16,
      "o:preferred_column_name": "DATA$jobId"
    },
    "salary": {
      "type": "number",
      "o:length": 8,
      "o:preferred_column_name": "DATA$salary"
    },
    "hireDate": {
      "type": "string",
      "o:length": 32,
      "o:preferred_column_name": "DATA$hireDate"
    },
    "lastName": {
      "type": "string",
      "o:length": 16,
      "o:preferred_column_name": "DATA$lastName"
    },
    "firstName": {
      "type": "string",
      "o:length": 16,
      "o:preferred_column_name": "DATA$firstName"
    },
    "managerId": {
      "type": "string",
      "o:length": 4,
      "o:preferred_column_name": "DATA$managerId"
    },
    "employeeId": {
      "type": "number",
      "o:length": 4,
      "o:preferred_column_name": "DATA$employeeId"
    },
    "phoneNumber": {
      "type": "string",
      "o:length": 16,
      "o:preferred_column_name": "DATA$phoneNumber"
    },
    "departmentId": {
      "type": "string",
      "o:length": 4,
      "o:preferred_column_name": "DATA$departmentId"
    },
    "commissionPct": {
      "type": "string",
      "o:length": 32,
      "o:preferred_column_name": "DATA$commissionPct"
    }
  }
}
```

Handling Transactions with SODA for In-Database JavaScript

Unlike the client-side JavaScript SQL driver, the MLE JavaScript SQL driver does not provide an `autoCommit` feature. You need to commit or roll your transactions back, either in the PL/SQL layer in case of module calls, or directly in the JavaScript code by calling `connection.commit()` or `connection.rollback()`.

> **Caution:**
>
> If any uncommitted operation raises an error, and you do not explicitly roll back the transaction, the incomplete transaction might leave the relevant data in an inconsistent state (uncommitted, partial results).

Creating Call Specifications Involving the SODA API

Earlier in this chapter, in the section *Getting Started with SODA for In-Database JavaScript*, an example showing how to invoke the MLE SODA API using an inline call specification is included. The following short example demonstrates how to use SODA in MLE modules.

Example 8-22   Use SODA for In-Database JavaScript

See Example 8-8 for details about how to create `employeesCollection`, used in this example.

```
CREATE OR REPLACE MLE MODULE end_to_end_demo
LANGUAGE JAVASCRIPT AS
```

```
/**
 * Example for a private function used to open and return a SodaCollection
 *
 * @param {string} collectionName the name of the collection to open
 * @returns {SodaCollection} the collection handle
 * @throws Error if the collection cannot be opened
 */
function openAndCheckCollection(collectionName){

  const col = soda.openCollection(collectionName);
  if(col === null){
    throw new Error(`invalid collection name: ${collectionName}`);
  }

  return col;
}

/**
 * Top-level (public) function demonstrating how to use a QBE to
 * filter documents in a collection.
 *
 * @param {number} departmentId the numeric department ID
 * @returns {number} the number of employees found in departmentId
 */
export function simpleSodaDemo(departmentId){

  if(departmentId === undefined || isNaN(departmentId)){
    throw new Error('please provide a valid numeric department ID');
  }

  const col = openAndCheckCollection('employeesCollection');

  const numDocs = col.find()
                     .filter({"departmentId": departmentId})
                     .count();

  return numDocs;
}
/
```

After the module has been created you need to create the call specification. The module features a single public function, so a standalone function should suffice:

```
CREATE OR REPLACE FUNCTION simple_soda_demo(
  "departmentId" NUMBER
) RETURN NUMBER
AUTHID current_user
AS MLE MODULE end_to_end_demo
SIGNATURE 'simpleSodaDemo';
/
```

Now everything is in place to call the function:

```
select simple_soda_demo(30);
```

Result:

```
SIMPLE_SODA_DEMO(30)
--------------------
                   6
```