Oracle® Database

SODA for Java Developer's Guide

Release 1.1

E85826-05

December 2020

Oracle Database SODA for Java Developer's Guide, Release 1.1

E85826-05

Copyright © 2018, 2020, Oracle and/or its affiliates.

Primary Author: Drew Adams

Contributing Authors: Sheila Moore

Contributors: Douglas McMahon, Maxim Orgiyan, Josh Spiegel

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software" or "commercial computer software documentation" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government’s use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

List of Examples

List of Tables

Preface

This document explains how to use Simple Oracle Document Access (SODA) for Java.

Audience

This document is intended for users of SODA for Java.

Documentation Accessibility

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

Access to Oracle Support

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

Related Documents

For more information, see these Oracle resources:

- SODA for Java on GitHub
- SODA for Java Releases
- SODA for Java Javadoc for detailed information about specific Java methods
- https://docs.oracle.com/en/database/oracle/simple-oracle-document-access/ for complete information about SODA and its implementations
- *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for general information about SODA
- *Oracle as a Document Store* for general information about using JSON data in Oracle Database, including with SODA
- *Oracle Database JSON Developer's Guide* for information about using SQL and PL/SQL with JSON data stored in Oracle Database

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at OTN Registration.

If you already have a user name and password for OTN then you can go directly to the documentation section of the OTN Web site at OTN Documentation.

Conventions

The following text conventions are used in this document:

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

1 SODA for Java Prerequisites

Before you can use SODA for Java you must configure your Java environment.

To use SODA for Java with Oracle Database:

- You must have one of the following Oracle Database releases installed:
  - Oracle Database 12*c* Release 2 (12.2) or later (no patches required)
  - Oracle Database 12*c* Release 1 (12.1.0.2) with Merge Label Request (MLR) bundle patch 20885778 (patch 20885778 obsoletes patch 20080249)

    Obtain this patch from My Oracle Support (My Oracle Support). Select tab **Patches & Updates**. Search for the patch number, 20885778, or access it directly at this URL: `https://support.oracle.com/rs?type=patch&id=20885778`.
- You must have Java Runtime Environment 1.6 (JRE 1.6) or higher.

For information about the minimal versions of the driver and dependencies needed for SODA for Java, see SODA Drivers in *Oracle Database Introduction to Simple Oracle Document Access (SODA)*.

2 SODA for Java Overview

SODA for Java is a Java API that implements **Simple Oracle Document Access** (SODA). You can use it with Java to perform create, read (retrieve), update, and delete (CRUD) operations on documents of any kind, and you can use it to query JSON documents.

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

Oracle relational database management system (RDBMS) supports storing and querying JSON data. To access this functionality, you need structured query language (SQL) with special JSON SQL operators and Java Database Connectivity (JDBC). SODA for Java hides the complexities of SQL/JSON programming.

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

Note:

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

See Also:

- SODA for Java on GitHub
- SODA for Java Releases
- SODA for Java Javadoc for detailed information about specific Java methods
- https://docs.oracle.com/en/database/oracle/simple-oracle-document-access/ for complete information about SODA and its implementations
- *Oracle Database JSON Developer's Guide* for information about using SQL and PL/SQL with JSON data

3

Using SODA for Java

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

(CRUD operations are also called "read and write operations" in this document.)

Getting Started with SODA for Java

How to access SODA for Java 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.

> **Note:**
>
> Don't worry if not everything in this topic is clear to you on first reading. The necessary concepts are developed in detail in other topics. This topic should give you an idea of what is involved overall in using SODA.

Follow these steps to get started with SODA for Java:

1. Ensure that all of the prerequisites have been met for using SODA for Java. See SODA for Java Prerequisites.
2. Identify the database schema (user account) used to store collections, and grant database role `SODA_APP` to that schema. (Replace placeholder *`user`* here by a real account name.)

   ```
   GRANT SODA_APP TO user;
   ```

3. Place all required jar files and file `testSoda.java` (which contains the text in Example 3-1) into a directory.
4. In `testSoda.java`:
   - Use a host name, port number, and service name that are appropriate for your Oracle Database instance. (Replace placeholders *`host_name`*, *`port_number`*, and *`service_name`*, respectively.)
   - Use the name and password for the database schema (user account) identified in step 2. (Replace placeholders *`user`* and *`password`*.)
5. Use the `cd` command to go to the directory that contains the jar files and file `testSoda.java`.
6. Execute these commands:

   ```
   javac -classpath "*" testSoda.java
   java -classpath "*:." testSoda
   ```

Instead of the second of these commands, you can optionally use the following command. It has the additional effect of dropping the collection, cleaning up the database table that is used to store the collection and its metadata.

```
java -classpath "*:." testSoda drop
```

Using argument `drop` here has the effect of invoking method `drop()`, which is the proper way to drop a 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.

To work with SODA for Java you must first open a JDBC connection. This is illustrated in Example 3-1. For details of how to open a JDBC connection, see *Oracle Database JDBC Developer's Guide*.

Example 3-1 testSoda.java

In this example, *replace* placeholders *`host_name`*, *`port_number`*, *`service_name`*, *`user`*, and *`password`* by a host name, port number, service name, database schema (user) name, and password appropriate for your database instance.

```
import java.sql.Connection;
import java.sql.DriverManager;

import oracle.soda.rdbms.OracleRDBMSClient;

import oracle.soda.OracleDatabase;
import oracle.soda.OracleCursor;
import oracle.soda.OracleCollection;
import oracle.soda.OracleDocument;
import oracle.soda.OracleException;

import java.util.Properties;

public class testSoda
{
  public static void main(String[] arg)
  {
      // Set the JDBC connection string, using information appropriate for your Oracle Database instance.
      // (Be sure to replace placeholders host_name, port_number, and service_name in the string.)
      String url = "jdbc:oracle:thin:@//host_name:port_number/service_name";

      // Set properties user and password.
      // (Be sure to replace placeholders user and password with appropriate string values.)
      Properties props = new Properties();
      props.setProperty("user", user);
      props.setProperty("password", password);

      Connection conn = null;

      try
      {
```

```
    // Get a JDBC connection to an Oracle instance.
    conn = DriverManager.getConnection(url, props);

    // Enable JDBC implicit statement caching
    conn.setImplicitCachingEnabled(true);
    conn.setStatementCacheSize(50);

    // Get an OracleRDBMSClient - starting point of SODA for Java application.
    OracleRDBMSClient cl = new OracleRDBMSClient();

    // Get a database.
    OracleDatabase db = cl.getDatabase(conn);

    // Create a collection with the name "MyJSONCollection".
    // This creates a database table, also named "MyJSONCollection", to store the collection.
    OracleCollection col = db.admin().createCollection("MyJSONCollection");

    // Create a JSON document.
    OracleDocument doc =
      db.createDocumentFromString("{ \"name\" : \"Alexander\" }");

    // Insert the document into a collection.
    col.insert(doc);

    // Find all documents in the collection.
    OracleCursor c = null;

    try
    {
      c = col.find().getCursor();
      OracleDocument resultDoc;

      while (c.hasNext())
      {
        // Get the next document.
        resultDoc = c.next();

        // Print document components
        System.out.println ("Key:         " + resultDoc.getKey());
        System.out.println ("Content:     " + resultDoc.getContentAsString());
        System.out.println ("Version:     " + resultDoc.getVersion());
        System.out.println ("Last modified: " + resultDoc.getLastModified());
        System.out.println ("Created on:    " + resultDoc.getCreatedOn());
        System.out.println ("Media:         " + resultDoc.getMediaType());
        System.out.println ("\n");
      }
    }
    finally
    {
      // IMPORTANT: YOU MUST CLOSE THE CURSOR TO RELEASE RESOURCES.
      if (c != null) c.close();
    }

    // Drop the collection, deleting the table underlying it and the collection metadata.
    if (arg.length > 0 && arg[0].equals("drop")) {
      col.admin().drop();
      System.out.println ("\nCollection dropped");
    }
}
// SODA for Java throws a checked OracleException
catch (OracleException e) { e.printStackTrace(); }
catch (Exception e) { e.printStackTrace(); }
finally
{
  try { if (conn != null)  conn.close(); }
  catch (Exception e) { }
}
```

```
  }
}
```

Creating a Document Collection with SODA for Java

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

In your Java application, first create an `OracleRDBMSClient` object, which is the starting point for any Java application working with SODA for Java:

```
OracleRDBMSClient myClient = new OracleRDBMSClient();
```

Caution:

An `OracleRDBMSClient` object is thread-safe. Other SODA for Java interfaces are *not* thread-safe, however — do not share them among multiple threads.

Next, pass the JDBC connection (`jdbcConnection`, here) to method `OracleClient.getDatabase()`, to obtain an `OracleDatabase` object (`db`, here):

```
OracleDatabase db = myClient.getDatabase(jdbcConnection);
```

Note:

Oracle recommends that you *enable implicit statement caching* for the JDBC connection that you pass to SODA. This can improve the performance of read and write operations. The underlying implementation of read and write operations generates JDBC prepared statements.

If you do not enable implicit caching then each time a read or write operation is created a new JDBC prepared statement is constructed. With implicit caching enabled, a new JDBC prepared statement is created only if it is not already in the cache.

See also: *Oracle Database JDBC Developer's Guide* and *Oracle Universal Connection Pool Developer's Guide*

Collection creation methods are available on interface `OracleDatabaseAdmin`. To access this interface, invoke method `admin()` on an `OracleDatabase` object (`db`, here):

```
OracleDatabaseAdmin dbAdmin = db.admin();
```

You can then create a collection — an `OracleCollection` object (`col`, here), by invoking method `createCollection()` on the object returned from `admin()`, passing it the collection name (`myCollection`, here) as a string:

```
OracleCollection col = dbAdmin.createCollection("myCollection");
```

Method `createCollection()` without a metadata argument creates the following in Oracle Database:

- Persistent default collection metadata.
- A table for storing the collection, in the schema with which the input JDBC connection is configured.

> **Note:**
>
> If the table name used by method `createCollection()` names an *existing* table in the schema with which the JDBC connection is configured, then the method tries to map that table to the collection. This behavior includes the default case, where the table name is derived from the collection name.

The *default collection metadata* has the following characteristics.

- Each document in the collection has these document components:
  - Key
  - Content
  - Creation timestamp
  - Last-modified timestamp
  - Version
- The collection can store only JSON documents.
- Document keys are automatically generated for documents that you add to the collection.

The default collection configuration is recommended in most cases, but collections are highly configurable. When you create a collection you can specify things such as the following:

- Storage details, such as the name of the table that stores the collection and the names and data types of its columns.
- The presence or absence of columns for creation timestamp, last-modified timestamp, and version.
- Whether the collection can store only JSON documents.
- Methods of document key generation, and whether document keys are client-assigned or generated automatically.
- Methods of version generation.

This configurability also lets you map a new collection to an existing table.

To configure a collection in a *non*default way, create a JSON `OracleDocument` instance of custom collection metadata (`collectionMetadata`, here) and pass it, along with the collection-name string (`myCollection`, here) to method `createCollection()`:

```
OracleCollection col2 = dbAdmin.createCollection("myCollection",
collectionMetadata);
```

To build and generate this `OracleDocument` instance easily, you can use `OracleRDBMSMetadataBuilder`.

If you do not care about the details of collection storage and configuration, then use method `createCollection(myCollection)`, as in Example 3-2.

You can search or change a collection only if it is open. A newly created collection is open for the life of your session.

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

Note:

Unless otherwise stated, the remainder of this documentation assumes that a collection has the *default* configuration.

See Also:

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

Opening an Existing Document Collection with SODA for Java

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

Example 3-2 Opening an Existing Document Collection

This example opens the collection named `myCollectionName` and returns the `OracleCollection` object that represents this collection. If the value returned is `null` then there is no existing collection named `myCollectionName`.

```
OracleCollection col = db.openCollection("myCollectionName");
```

Checking Whether a Given Collection Exists with SODA for Java

You can use `OracleDatabase` method `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 3-2, if `myCollectionName` does not name an existing collection then `ocol` is assigned the value `null`.

Discovering Existing Collections with SODA for Java

You can use `OracleDatabaseAdmin` method `getCollectionNames()` to discover existing collections.

Example 3-3 Printing the Names of All Existing Collections

This example prints the names of all existing collections. It uses method `getCollectionNames()` with the simplest signature, which accepts no arguments.

```
List<String> names =  db.admin().getCollectionNames();

for (String name : names)
 System.out.println ("Collection name: " + name);
```

Dropping a Document Collection with SODA for Java

You use `OracleCollectionAdmin` method `drop()` to drop a document 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 method `drop()`. For `drop()` to succeed, all uncommitted writes to the collection must first be committed. Otherwise, an exception is raised.

Example 3-4 Dropping a Document Collection

This example drops collection `col`.

```
col.admin().drop();
```

Creating Documents with SODA for Java

Creation of documents by SODA for Java is described.

SODA for Java represents a document using Java interface `OracleDocument`. This interface is designed primarily to represent JSON documents, but it also supports other content types. An `OracleDocument` object is a carrier of document content and other document components, such as the document key.

To create JSON content for an `OracleDocument` instance, you can use your favorite package — for example, JSR353, the Java API for JSON processing.

Here is an example of a simple JSON document:

```
{ "name" :    "Alexander",
  "address" : "1234 Main Street",
  "city" :    "Anytown",
  "state" :   "CA",
  "zip" :     "12345"
}
```

> **Note:**
>
> In SODA, JSON content must conform to RFC 4627. In addition, in SODA for *Java*, the encoding of JSON content must be either UTF-8 or UTF-16 (big endian (BE) or little endian (LE)). Although RFC 4627 also allows UTF-32 (BE and LE) encodings, SODA for Java does not support them.

To create an `OracleDocument` instance from content that is represented as a byte array or a `String` instance, use the following methods (which `OracleDatabase` inherits from `OracleDocumentFactory`), respectively:

- `createDocumentFromByteArray()`
- `createDocumentFromString()`

A document has these components:

- Key
- Content
- Creation time stamp
- Last-modified time stamp
- Version
- Media type ("`application/json`" for JSON documents)

When you create a document by invoking method `createDocumentFromString()` or `createDocumentFromByteArray()`:

- You might need to provide the document key as a method argument.

  In a collection, each document must have a key. You must provide the key when you create the document *only* if you expect to insert the document into a collection that does *not* automatically generate keys for inserted documents. By default, collections are configured to automatically generate document keys.
- You can provide the document content as a method argument (the `content` parameter is required, but its value can be `null`).
- The method sets the values of the creation time stamp, last-modified time stamp, and version to `null`.

Methods `createDocumentFromString()` and `createDocumentFromByteArray()` each have multiple variants:

- The simplest variant accepts only document content. The media type defaults to "`application/json`", and the other components default to `null`. This variant is useful for creating documents for insertion into collections that automatically generate document keys.

- Another variant accepts both document key and document content. The media type defaults to `"application/json"`, and the other components default to `null`. This variant is useful for creating documents for insertion into collections that have client-assigned document keys.
- The most flexible (and most verbose) variant accepts key, content, and content type. Because it lets you specify content type, this variant is useful for creating non-JSON documents.

Example 3-5 creates an `OracleDocument` instance with content only. The media type defaults to `"application/json"`, and the other document components default to `null`.

Example 3-6 creates an `OracleDocument` instance with document key and content. The media type defaults to `"application/json"`, and the other document components default to `null`.

You write documents to collections using SODA for Java write operations, and you read documents from collections using SODA for Java read operations.

See Also:

- `OracleDocumentFactory` Javadoc for more information about method`createDocumentFromString()` and `createDocumentFromByteArray()`
- `OracleDocument` Javadoc for information about getter methods, which you use to access document components
- *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for an overview of SODA documents
- *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for restrictions that apply for SODA documents

Example 3-5 Creating a Document with JSON Content

```
OracleDocument doc =
  odb.createDocumentFromString("{ \"name\" : \"Alexander\"}");

// Get the content
String content = doc.getContentAsString();

// Get the content type (it is "application/json")
String contentType = doc.getContentType();
```

Example 3-6 Creating a Document with Document Key and JSON Content

```
OracleDocument doc
  = odb.createDocumentFromString("myKey", "{ \"name\" :
\"Alexander\"}");
```

Related Topics

- Inserting Documents into Collections with SODA for Java
  To insert a document into a collection, you invoke `OracleCollection` method `insert(OracleDocument)` or `insertAndGet(OracleDocument)`. These methods create document keys automatically, unless the collection is configured with client-assigned keys and the input document provides the key.
- Saving Documents into Collections with SODA for Java
  You use `OracleCollection` methods `save(OracleDocument)` and `saveAndGet(OracleDocument)` to save documents into collections.
- Finding Documents in Collections with SODA for Java
  To find documents in a collection, you invoke `OracleCollection` method `find()`, which returns an `OracleOperationBuilder` object that represents a query that finds all documents in the collection.
- Replacing Documents in a Collection with SODA for Java
  To replace the content of one document in a collection with the content of another, you chain together `OracleOperationBuilder` method `key(String)` with either method **`replaceOne`**`(OracleDocument)` or method **`replaceOneAndGet`**`(OracleDocument)`. Method `replaceOne(OracleDocument)` only replaces the document. Method `replaceOneAndGet(OracleDocument)` also returns a result document, which contains all document components except the content.
- Removing Documents from a Collection with SODA for Java
  To remove a document from a collection, you chain together (1) `OracleCollection` method `find()` with these `OracleOperationBuilder` methods: (2) `key()`, `keyLike()`, `keys()`, or `filter()`; (3) `version()` (optional); and (4) `remove()`. Examples are provided.

Inserting Documents into Collections with SODA for Java

To insert a document into a collection, you invoke `OracleCollection` method `insert(OracleDocument)` or `insertAndGet(OracleDocument)`. These methods create document keys automatically, unless the collection is configured with client-assigned keys and the input document provides the key.

Method `insert(OracleDocument)` only inserts the document into the collection. Method `insertAndGet(OracleDocument)` also returns a result document, which contains the document key and any other generated document components (except the content).

Both methods automatically set the values of the creation time stamp, last-modified time stamp, and version (if the collection is configured to include these components and to generate the version automatically, as is the case by default).

Note:

If the collection is configured with client-assigned document keys (which is not the default case), and the input document provides a key that identifies an existing document in the collection, then these methods throw an exception. If you want the input document to *replace* the existing document instead of causing an exception, see Saving Documents into Collections with SODA for Java.

Example 3-7 creates a document and inserts it into a collection using method `insert()`.

Example 3-8 creates a document, inserts it into a collection using method `insertAndGet()`, and then gets each of the generated components from the result document (which contains them).

To efficiently insert a large number of documents into a collection, invoke `OracleCollection` method `insert(Iterator<OracleDocument>)` or `insertAndGet(Iterator<OracleDocument>)`. These methods are analogous to `insert(OracleDocument)` and `insertAndGet(OracleDocument)`, but instead of handling a single document, they handle multiple documents. Parameter `Iterator<oracleDocument>` is an iterator over multiple input documents.

Method `insertAndGet(Iterator<OracleDocument>)` returns a list of result documents — one `OracleDocument` instance for each input document. Each such result document contains the document key and any other generated document components (except the content). The order of the result documents corresponds to the order of input documents, allowing correlation of result and input documents.

There is a variant of method `insertAndGet()` that accepts an optional second argument, `options`, whose value is a Java `Map`.

You can use argument `options` to provide a SQL hint, to turn real-time SQL monitoring of queries on and off. Use method `put()` to add key `"hint"` with value `"MONITOR"` to the map argument. The hint is passed down to the SQL code that underlies SODA.

The string value for key `"hint"` uses the SQL hint syntax (that is, the hint text, without the enclosing SQL comment syntax `/*+...*/`). Use *only* hint `MONITOR` (turn on monitoring) or `NO_MONITOR` (turn off monitoring).

(You can use this to pass any SQL hints, but `MONITOR` and `NO_MONITOR` are the useful ones for SODA, and an inappropriate hint can cause the optimizer to produce a suboptimal query plan.)

See Also:

- `OracleCollection` Javadoc for more information about the insertion methods:
  - `insert(OracleDocument)`
  - `insert(Iterator<OracleDocument>)`
  - `insertAndGet(OracleDocument)`
  - `insertAndGet(OracleDocument document, Map<String, ?> options)`
  - `insertAndGet(Iterator<OracleDocument>)`
  - `insertAndGet(Iterator<OracleDocument> documents, Map<String, ?> options)`
- Monitoring Database Operations in *Oracle Database SQL Tuning Guide* for complete information about monitoring database operations
- MONITOR and NO_MONITOR Hints in *Oracle Database SQL Tuning Guide* for information about the syntax and behavior of SQL hints `MONITOR` and `NO_MONITOR`

Example 3-7 Inserting a Document into a Collection

```
OracleDocument doc =
  db.createDocumentFromString("{ \"name\" : \"Alexander\"}");

col.insert(doc);
```

Example 3-8 Inserting a Document into a Collection and Getting the Result Document

```
OracleDocument doc =
  db.createDocumentFromString("{ \"name\" : \"Alexander\"}");

OracleDocument insertedDoc = col.insertAndGet(doc);

// Get the generated document key
String key = insertedDoc.getKey();

// Get the generated creation timestamp
String createdOn = insertedDoc.getCreatedOn();

// Get the generated last-modified timestamp
String lastModified = insertedDoc.getLastModified();

// Get the generated version
String version = insertedDoc.getVersion();
```

Saving Documents into Collections with SODA for Java

You use `OracleCollection` methods `save(OracleDocument)` and `saveAndGet(OracleDocument)` to save documents into collections.

These methods are similar to methods `insert(OracleDocument)` and `insertAndGet(OracleDocument)` 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. (Methods `insert(OracleDocument)` and `insertAndGet(OracleDocument)` throw an exception in that case.)

Note:

By default, collections are configured with automatically generated document keys. Therefore, for a default collection, methods `save(OracleDocument)` and `saveAndGet(OracleDocument)` are equivalent to methods `insert(OracleDocument)` and `insertAndGet(OracleDocument)`, respectively.

There is a variant of method `saveAndGet()` that accepts an optional second argument, `options`, whose value is a Java `Map`.

You can use argument `options` to provide a SQL hint, to turn real-time SQL monitoring of queries on and off. Use method `put()` to add key `"hint"` with value `"MONITOR"` to the map argument. The hint is passed down to the SQL code that underlies SODA.

The string value for key `"hint"` uses the SQL hint syntax (that is, the hint text, without the enclosing SQL comment syntax `/*+`...`*/`). Use *only* hint `MONITOR` (turn on monitoring) or `NO_MONITOR` (turn off monitoring).

(You can use this to pass any SQL hints, but `MONITOR` and `NO_MONITOR` are the useful ones for SODA, and an inappropriate hint can cause the optimizer to produce a suboptimal query plan.)

See Also:

- `OracleCollection` Javadoc for more information about methods `save(OracleDocument)`, `saveAndGet(OracleDocument)`, and `saveAndGet(OracleDocument document, Map<String, ?> options)`
- Monitoring Database Operations in *Oracle Database SQL Tuning Guide* for complete information about monitoring database operations
- MONITOR and NO_MONITOR Hints in *Oracle Database SQL Tuning Guide* for information about the syntax and behavior of SQL hints `MONITOR` and `NO_MONITOR`

Example 3-9 Saving a Document into a Collection

This example saves a document into a collection that is configured with client-assigned document keys, using method `saveAndGet()`. It then gets the key and the generated document components (except the content) from the result document (which contains them).

```
OracleRDBMSClient cl = new OracleRDBMSClient();
OracleDatabase db = ...

// Configures the collection with client-assigned document keys
OracleDocument collMeta =

cl.createMetadataBuilder().keyColumnAssignmentMethod("client").build();
OracleCollection clientKeysColl = db.createCollection("collectionName",
                                                     collMeta);

// For a collection configured with client-assigned document keys,
// you must provide the key for the input document.
OracleDocument cKeyDoc =
  db.createDocumentFromString("myKey", "{ \"name\" : \"Alexander\"}");

// If key "myKey" already identifies a document in the collection
// then cKeyDoc replaces the existing doc.
OracleDocument savedDoc = clientKeysColl.saveAndGet(cKeyDoc);

// Get document key ("myKey")
String key = savedDoc.getKey();

// Get the generated creation timestamp
String createdOn = savedDoc.getCreatedOn();

// Get the generated last-modified timestamp
String lastModified = savedDoc.getLastModified();

// Get the generated version
String version = savedDoc.getVersion();
```

SODA for Java Read and Write Operations

The primary way you specify read and write operations (other than insert and save) is to chain together `OracleOperationBuilder` methods.

`OracleOperationBuilder` provides the following nonterminal methods, which you can chain together to specify a read or write operation: `key()`, `keyLike()`, `keys()`, `filter()`, `version()`, `skip()`, `limit()`, and `headerOnly()`.

These are called **nonterminal** methods because they return the same `OracleOperationBuilder` object on which they are invoked, which allows them to be chained together. Nonterminal methods let you specify parts of an operation; they do not create or execute an operation.

`OracleOperationBuilder` also provides terminal methods. A **terminal** method always appears at the end of a method chain, and it creates and executes the operation.

The terminal methods for *read* operations are `getCursor()`, `getOne()`, and `count()`. The terminal methods for *write* operations are `replaceOne()`, `replaceOneAndGet()`, and `remove()`.

Note:

If you use `OracleCursor` method `next()` or `OracleOperationBuilder` method `getOne()`, and if the underlying document is larger than 2 gigabytes, then an exception is thrown.

Unless the Javadoc documentation for a method states 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 3-1 briefly describes `OracleOperationBuilder` nonterminal methods for building operations against a collection.

Table 3-1 OracleOperationBuilder Nonterminal Methods

Method	Description
`key()`	Find a document that has the specified document key.
`keyLike()`	(Supported only for collections with client-assigned keys and a key column of data type `VARCHAR2`.) Find documents that have keys matching a given pattern. The first parameter is the pattern, which can contain the wildcards `_` (underscore), which matches any single character, and `%` (percent), which matches zero or more characters. If the second parameter is non-null then it is a character that, when it immediately precedes either `_` or `%`, escapes that character so that it is matched literally and not as a wildcard. For example, if the pattern is `mykey!_1` and the second parameter is the character `!` then the underscore is matched literally — the only match is the name `mykey_1`.
`keys()`	Find documents that have the specified document keys. The maximum number of keys passed as argument must not exceed 1000, or else a runtime error is raised.
`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()`. For example: `find().key("key1").version("version1")`.
`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 3-1 (Cont.) OracleOperationBuilder Nonterminal Methods

Method	Description
`hint()`	Provide a SQL tuning hint, to turn real-time SQL monitoring of queries on and off. The argument is a string with the SQL hint syntax. Use this *only* with argument `"MONITOR"` (turn on monitoring) or `"NO_MONITOR"` (turn off monitoring). The hint is simply passed down to the SQL code that underlies SODA. (You can use this to pass any SQL hints, but `MONITOR` and `NO_MONITOR` are the useful ones for SODA, and an inappropriate hint can cause the optimizer to produce a suboptimal query plan.) **See Also:** • Monitoring Database Operations in *Oracle Database SQL Tuning Guide* for complete information about monitoring database operations • MONITOR and NO_MONITOR Hints in *Oracle Database SQL Tuning Guide* for information about the syntax and behavior of SQL hints `MONITOR` and `NO_MONITOR`

Table 3-2 briefly describes `OracleOperationBuilder` terminal methods for creating and executing read operations against a collection.

Table 3-2 OracleOperationBuilder 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.

Table 3-3 briefly describes `OracleOperationBuilder` terminal methods for executing write operations against a collection.

Table 3-3 OracleOperationBuilder 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:

- The SODA for Java Javadoc for complete information about `OracleOperationBuilder` methods
- *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for information about SODA restrictions

Finding Documents in Collections with SODA for Java

To find documents in a collection, you invoke `OracleCollection` method `find()`, which returns an `OracleOperationBuilder` object that represents a query that finds all documents in the collection.

To execute the query, obtain a cursor for its results by invoking `OracleOperationBuilder` method `getCursor()`. Then use the cursor to visit each document in the result list. To determine whether the result list has a next document, and to obtain the next document, invoke `OracleCursor` methods `hasNext()` and `next()`, respectively. This is illustrated by Example 3-10 and other examples here.

However, you typically do not work directly with the `OracleOperationBuilder` object. Instead, you *chain together* some of its methods, to specify various find operations. This is illustrated in the other examples here, which find documents by their keys or using query-by-example (QBE) filter specifications.

Note:

Examples here that use method `getContentAsString()` assume that all documents in the collection are JSON documents. If they are not, this method throws an exception.

Example 3-10 Finding All Documents in a Collection

This example first obtains a cursor for a query result list that contains each document in a collection. It then uses the cursor in a `while` statement to get and print the content of each document in the result list, as a string. Finally, it closes the cursor.

Note:

To avoid resource leaks, *close* any cursor that you no longer need.

```
OracleCursor c = col.find().getCursor();

while (c.hasNext()) {
  OracleDocument resultDoc = c.next();
  System.out.println("Document content: " +
                       resultDoc.getContentAsString());
}

// IMPORTANT: You must close the cursor to release resources!
c.close;
```

Example 3-11 Finding the Unique Document That Has a Given Document Key

This example chains together `OracleOperationBuilder` methods to specify an operation that finds the unique document whose key is `"key1"`. It uses nonterminal method `key()` to specify the document. It then uses terminal method `getOne()` to execute the read operation and return the document (or `null` if no such document is found).

```
OracleDocument doc = col.find().key("key1").getOne();
```

Example 3-12 Finding Multiple Documents with Specified Document Keys

This example defines `HashSet` **`myKeys`**, with (string) keys `"key1"`, `"key2"`, and `"key3"`. It then finds the documents that have those keys, and it prints the key and content of each of those documents.

Nonterminal method `keys()` specifies the documents with the given keys. Terminal method `getCursor()` executes the read operation and returns a cursor over the result documents.

Note:

The maximum number of keys in the set supplied to method `keys()` must not exceed 1000, or else a runtime error is raised.

```
Set<String> myKeys = new HashSet<String>();
myKeys.put("key1");
myKeys.put("key2");
myKeys.put("key3");

OracleCursor c = col.find().keys(myKeys).getCursor();

while (c.hasNext()) {
  OracleDocument resultDoc = c.next();

  // Print the document key and document content
  System.out.println ("Document key: " + resultDoc.getKey() + "\n" +
                      " document content: " +
resultDoc.getContentAsString());
}

c.close();
```

Example 3-13   Finding Documents with a Filter Specification

Nonterminal method `filter()` provides a powerful way to filter JSON documents in a collection. Its `OracleDocument` parameter is a JSON query-by-example (QBE, also called a filter specification).

The syntax of filter specifications is an expressive pattern-matching language for JSON documents. This example uses only a very simple QBE, just to indicate how you make use of one in SODA for Java.

This example does the following:

1. Creates a filter specification that looks for all JSON documents whose `name` field has value `"Alexander"`.
2. Uses the filter specification to find the matching documents.
3. Prints the key and content of each document.

```
// Create the filter specification
OracleDocument filterSpec =
  db.createDocumentFromString("{ \"name\" : \"Alexander\"}");

OracleCursor c = col.find().filter(filterSpec).getCursor();

while (c.hasNext()) {
  OracleDocument resultDoc = c.next();

  // Print the document key and document content
  System.out.println ("Document key: " + resultDoc.getKey() + "\n" +
                      " document content: " + resultDoc.getContent());
}

c.close();
```

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 3-14   Specifying Pagination Queries with Methods skip() and limit()

This example uses nonterminal methods `skip()` and `limit()` in a pagination query. (Filter specification `filterSpec` is from Example 3-13.)

```
// Find all documents matching the filterSpec, skip the first 1000,
// and limit the number of returned documents to 100.
OracleCursor c =
  col.find().filter(filterSpec).skip(1000).limit(100).getCursor();

while (c.hasNext()) {
  OracleDocument resultDoc = c.next();

  // Print the document key and document content
  System.out.println ("Document key: " + resultDoc.getKey() + "\n" +
                        " document content: " + resultDoc.getContent());
}

c.close();
```

Example 3-15 Specifying Document Version

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

You typically use `version()` together with method `key()`, which specifies the document. You can also use `version()` with methods `keyLike()` and `filter()`, provided they identify at most one document.

```
// Find a document with key "key1" and version "version1".
OracleDocument doc =
col.find().key("key1").version("version1").getOne();
```

Example 3-16 Finding Documents and Returning Only Their Headers

This example finds all documents with the specified document keys and returns only their headers. (The keys are those in `HashSet` `myKeys`, which is defined in Example 3-12.) Nonterminal method `headerOnly()` specifies the return of document headers only. A document header has all the document components except the content.

```
// Find all documents matching the keys in HashSet myKeys.
// For each document, return all document components except the content.
OracleCursor c = col.find().keys(myKeys).headerOnly().getCursor();
```

Example 3-17 Counting the Number of Documents Found

This example uses terminal method `count()` to get a count of all of the documents in the collection. It then gets a count of all of the documents that are returned by the filter specification `filterSpec` from Example 3-13.

```
// Get a count of all documents in the collection
int numDocs = col.find().count();

// Get a count of all documents in the collection that match a filter
spec
numDocs = col.find().filter(filterSpec).count();
```

Replacing Documents in a Collection with SODA for Java

To replace the content of one document in a collection with the content of another, you chain together `OracleOperationBuilder` method `key(String)` with either method **`replaceOne`**`(OracleDocument)` or method **`replaceOneAndGet`**`(OracleDocument)`.

Method `replaceOne(OracleDocument)` only replaces the document. Method `replaceOneAndGet(OracleDocument)` also returns a result document, which contains all document components except the content.

Both `replaceOne(OracleDocument)` and `replaceOneAndGet(OracleDocument)` update the values of the last-modified timestamp and the version. Replacement does *not* change the document key or the creation timestamp.

Note:

Some version-generation methods, including the default method, generate hash values of the document content. In such a case, if the document content does not change then neither does the version. For more information about version-generation methods, see SODA Collection Configuration Using Custom Metadata.

See Also:

`OracleOperationBuilder` Javadoc for more information about `replaceOne()` and `replaceOneAndGet()`

Example 3-18 Replacing a Document in a Collection and Getting the Result Document

This example replaces a document in a collection, gets the result document, and gets the generated components from the result document.

```
OracleDocument newDoc = ...
OracleDocument resultDoc =
col.find().key("k1").replaceOneAndGet(newDoc);

if (resultDoc != null)
{
  // Get the generated document key (unchanged by replacement operation)
  String key = resultDoc.getKey();

  // Get the generated version
  String version = resultDoc.getVersion();

  // Get the generated last-modified timestamp
  String lastModified = resultDoc.getLastModified();

  // Get the creation timestamp (unchanged by replacement operation)
  String createdOn = resultDoc.getCreatedOn();
}
```

Example 3-19 Replacing a Particular Version of a Document

To implement optimistic locking when replacing a document, you can chain together methods `key()` and `version()`, as in this example.

```
OracleDocument resultDoc =
  col.find().key("k1").version("v1").replaceOneAndGet(newDoc);
```

Removing Documents from a Collection with SODA for Java

To remove a document from a collection, you chain together (1) `OracleCollection` method `find()` with these `OracleOperationBuilder` methods: (2) `key()`, `keyLike()`, `keys()`, or `filter()`; (3) `version()` (optional); and (4) `remove()`. Examples are provided.

See Also:

`OracleOperationBuilder` Javadoc for more information about `key()`, `keys()`, `filter()`, `version()`, and `remove()`

Example 3-20 Removing a Document from a Collection Using a Document Key

This example removes the document whose document key is `"k1"`. The number of documents removed is returned.

```
// Count is 1, if the document with key "k1" is found in the collection.
// Count is 0, otherwise.
int count = col.find().key("k1").remove();
```

Example 3-21 Removing a Particular Version of a Document

This example implements optimistic locking when removing a document, by specifying the version of the document, as well as its key.

```
col.find().key("k1").version("v1").remove();
```

Example 3-22 Removing Documents from a Collection Using Document Keys

This example removes the documents whose keys are `"k1"` and `"k2"`.

```
Set<String> myKeys = new HashSet<String>();
myKeys.add("k1");
myKeys.add("k2");

// Count is 2 if two documents with keys "k1" and "k2"
// were found in the collection.
int count = col.find().keys(myKeys).remove();
```

Example 3-23 Removing JSON Documents from a Collection Using a Filter

This example uses a filter to remove the JSON documents whose `greeting` field has value `"hello"`. It then prints the number of documents removed.

```
OracleDocument filterSpec =
   db.createDocumentFromString("{ \"greeting\" : \"hello\" }");

int count = col.find().filter(filterSpec).remove();

// Print the number of documents removed
System.out.println ("Removed " + count + " documents"):
```

Indexing the Documents in a Collection with SODA for Java

You index the documents in a SODA collection with `OracleCollectionAdmin` method `createIndex()`. Its `OracleDocument` parameter is a textual JSON index specification. This can specify B-tree, spatial, full-text, or ad hoc indexing, and it can specify support for a JSON data guide.

Note:

To create any kind of index using SODA you need Oracle Database Release 12c (12.2.0.1) or later. But to create a B-tree index for a `DATE` or `TIMESTAMP` value you need Oracle Database Release 18c (18.1) or later.

You drop an index on a SODA collection with method `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.

An Oracle Spatial and Graph index is used for GeoJSON (spatial) data. A JSON search index is used for full-text searching and for persisting 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
- *Oracle Database JSON Developer’s Guide* for information about spatial indexing of GeoJSON data.

Example 3-24 Creating a B-Tree Index for a JSON Field with SODA for Java

This example creates a B-tree non-unique index for numeric field `address.zip` of the JSON documents in collection `myCollection`.

```
OracleDocument indexSpec =
  db.createDocumentFromString(
    "{\"name\" : \"ZIPCODE_IDX\",
      \"fields\" :
        [{\"path\" : \"address.zip\",
          \"datatype\" : \"number\",
          \"order\" : \"asc\"}]}");
col.admin().createIndex(indexSpec);
```

This is the same index specification, pretty-printed for legibility:

```
{"name"   : "ZIPCODE_IDX",
 "fields" : [{"path"     : "address.zip",
              "datatype" : "number",
              "order"    : "asc"}]}
```

Example 3-25 JSON Search Indexing with SODA for Java

This example indexes the documents in collection `myCollection` for ad hoc queries and full-text search (queries using QBE operator `$contains`), and 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`).

```
OracleDocument indexSpec = db.createDocumentFromString(
  "{\"name\" : \"SEARCH_AND_DATA_GUIDE_IDX\"}");
col.admin().createIndex(indexSpec);
```

The simple index specification it uses is equivalent to this one, which makes explicit the default values:

```
{"name"      : "SEARCH_AND_DATA_GUIDE_IDX",
 "dataguide" : "on",
 "search_on" : "text_value"}
```

If you instead wanted *only ad hoc* (search) indexing then you would explicitly specify a value of `"off"` for field `dataguide`. If you instead wanted *only data-guide* support then you would explicitly specify a value of `"none"` for field `search_on`.

Note:

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

Example 3-26 Dropping an Index with SODA for Java

To drop an index on a SODA collection, just pass the index name to `OracleCollectionAdmin` method `dropIndex()`. This example drops index `myIndex`.

```
col.admin().dropIndex("myIndex");
```

Getting a Data Guide for a Collection

You use `OracleCollectionAdmin` method `getDataGuide()` to get a data guide for a collection. A data guide is a JSON document that summarizes the structural and type information of the JSON documents in the collection. It records metadata about the fields used in those documents.

Before you can obtain a data guide for your collection you must create a data guide-enabled JSON search index on it. Example 3-25 shows how to do that.

Example 3-27 Getting a Data Guide with SODA for Java

This example gets a data guide using `OracleCollectionAdmin` method `getDataGuide()`.

```
OracleDocument dataGuide = col.admin().getDataGuide();
```

This returns a document whose content is a JSON data guide. To obtain this content as a string value, you can use `OracleDocument` method `getContentAsString()`.

```
System.out.println("Dataguide " + dataGuide.getContentAsString());
```

Handling Transactions with SODA for Java

You can cause SODA for Java to treat individual read and write operations, or groups of them, as a single transaction.

The JDBC connection that you pass to method `OracleClient.getDatabase()` has auto-commit mode either on or off.

If auto-commit mode is *on*, then each SODA for Java read operation and write operation is treated as a single transaction. If the operation succeeds, then the transaction automatically commits. If the operation fails, then an `OracleException` or `RuntimeException` is thrown, and the transaction automatically rolls back. SODA for Java itself throws only checked exceptions (`OracleException` and exceptions derived from `OracleException`). However, SODA for Java is built upon JDBC, which can throw a `RuntimeException` that SODA for Java passes through.

If auto-commit mode is *off*, then you can combine multiple SODA for Java read and write operations into one transaction. If the transaction succeeds, then your application must explicitly commit it, by calling method `commit()` on the JDBC connection. If the transaction fails, then an `OracleException` or `RuntimeException`, is thrown. Your application must handle the exception and explicitly roll back the transaction, by invoking method `rollback()` on the JDBC connection. (`RuntimeException` can be thrown only by JDBC, as mentioned in the preceding paragraph.)

Caution:

If auto-commit mode is *off*, an 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).

To facilitate transactional programming, SODA for Java supports optimistic locking: checking, before writing data back, that the data has not been modified (by another transaction) since it was read. With SODA for Java, you do this using replacement or removal by key and version, that is, checking the version value.Example 3-19 illustrates this for replacing a document, and Example 3-21 illustrates it for removing a document. Optimistic locking is especially useful in contexts where there is low data contention.

4

SODA Collection Metadata Caching

SODA collection metadata is stored persistently in the database, just like collection data. It is fetched transparently when needed, to perform collection operations. Fetching metadata from the database carries a performance cost. You can cache collection metadata in clients, to improve performance by avoiding database access to retrieve the metadata.

These are the main use cases for collection metadata caching:

- Listing the existing collections, then opening one or more of the collections listed.
- Creating a collection, then opening it.
- Reopening a collection.

In all of these cases, cached metadata can be used to open the collection.

A collection metadata cache can be *shared* by all of the `OracleDatabase` objects that are obtained from a given `OracleRDBMSClient` object, or it can be *local* to a single `OracleDatabase` object. Both kinds of caching are disabled by default.

If both local and shared caches are enabled for the same `OracleDatabase` object, entry lookup proceeds as follows:

1. The local cache is checked for an entry pertaining to a given collection used by the database object.
2. If not found in the local cache, the shared cache is checked for an entry for the collection.
3. If an entry for the collection is found in neither cache then the database is accessed to try to obtain the its metadata.

Enabling Collection Metadata Caching

Collection metadata caching is disabled by default. You can use constructor `OracleRDBMSClient(Properties *props*)` to enable shared or local collection metadata caching.

Parameter *`props`* here is a `Properties` instance that you initialize with one or both of the following properties:

- Property `oracle.soda.sharedMetadataCache` with value `"true"`: enable the shared cache
- Property `oracle.soda.localMetadataCache` with value `"true"`: enable the local cache

Example 4-1 illustrates this; it enables both shared and local caching.

Example 4-1   Enabling Collection Metadata Caching

```
Properties props = new Properties();
props.put("oracle.soda.sharedMetadataCache", "true");
props.put("oracle.soda.localMetadataCache", "true");
OracleRDBMSClient cl = new OracleRDBMSClient(props);
```

Shared Collection Metadata Cache

Each SODA client (`OracleRDBMSClient` object) is optionally associated with a collection metadata cache that records metadata for all collections (`OracleCollection` objects) that are created for all `OracleDatabase` objects created from that client. The cache is released when its associated client is released.

The number of entries in a shared cache is limited to 10,000 entries (100 database schemas times 100 collections per schema). A shared cache uses a least-recently-used (LRU) replacement policy: the least recently used entry is replaced by the addition of a new entry, when the cache is full (it has 10,000 entries).

A shared metadata cache requires *locking* to avoid access conflict, which can affect performance negatively because it limits concurrency.

Local Collection Metadata Cache

Each `OracleDatabase` object is optionally associated with a local collection metadata cache. It records metadata only for collections that are created for that `OracleDatabase` object. A local cache is released when its associated `OracleDatabase` object is released.

There is no limit on the number of entries for a local cache — entries are never evicted. The number of entries continues to grow as new collections are created for the given database object.

The lack of an eviction policy for local metadata caches means that cached collection metadata is always available; once cached, the database need never be accessed to obtain it.

With local caching, because there is no sharing, using different database objects to access the same collection can result in more round trips and more data replication than is the case for shared caching.

Unlike a shared metadata cache, a local cache requires no locking.

Caution:

Because the number of entries in the local cache is unbounded, Oracle does not recommend using the local cache if a particular Oracle Database object is used to create a large number of collections, as it could result in running out of memory.

5
SODA Collection Configuration Using Custom Metadata

SODA collections are highly configurable. You can customize collection metadata, to obtain different behavior from that provided by default.

Note:

You can customize collection metadata to obtain different behavior from that provided by default. However, changing some components requires familiarity with Oracle Database concepts, such as SQL data types. Oracle recommends that you do *not* change such components unless you have a compelling reason. Because SODA collections are implemented on top of Oracle Database tables (or views), many collection configuration components are related to the underlying table configuration.

For example, if you change the content column type from the default value to `VARCHAR2`, then you must understand the implications: content size for `VARCHAR2` is limited to 32K bytes, character-set conversion can take place, and so on.

See Also:

- *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for general information about SODA document collections and their metadata
- *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for reference information about collection metadata components

Getting the Metadata of an Existing Collection

`OracleCollectionAdmin` method `getMetadata()` returns all of the metadata for a collection, as a JSON document.

```
collectionName.admin().getMetadata();
```

See Also:

Default Collection Metadata example in *Oracle Database Introduction to Simple Oracle Document Access (SODA)*

Creating Custom Metadata for a Collection

Collection metadata is represented as a JSON `OracleDocument` instance. You can create such an instance directly, but Oracle recommends that you instead use `OracleRDBMSMetadataBuilder`, which you obtain by invoking `OracleRDBMSClient` method `createMetadataBuilder()`.

Two methods for creating collections are available on interface `OracleDatabaseAdmin` (accessed by invoking method `admin()` on an `OracleDatabase` object):

```
createCollection(String collectionName);
createCollection(String collectionName, OracleDocument
collectionMetadata);
```

The first method, which accepts only one argument, creates a collection with the default metadata. The default metadata specifies database schema name, table name (for the table storing the collection), five table columns (key, content, version, last-modified timestamp, and creation timestamp), and the details of these table columns. Each table column is represented by a field with a JSON object as value. That object contains additional details about the column—name, SQL type, and so on.

The default metadata for a collection is presented in Default Collection Metadata in *Oracle Database Introduction to Simple Oracle Document Access (SODA)*.

The second method, which accepts two arguments, lets you provide custom collection metadata in the form of a JSON `OracleDocument` object.

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

Method `createMetadataBuilder()` returns an `OracleRDBMSMetadataBuilder` instance that is preloaded with the default collection metadata. You can modify this preloaded metadata by calling `OracleRDBMSMetadataBuilder` methods that create custom metadata.

These methods correspond to different collection metadata components. You can customize these components by invoking builder methods in a chained manner. At the end of the chain, you invoke method `build()` to create collection metadata as a JSON `OracleDocument` object.

Example 5-1 illustrates this; it uses `OracleRDBMSMetadataBuilder` to create a collection that has custom metadata: a media type column. A media type column lets you store documents that are other than just JSON data, for example images and PDF documents.

The example first uses method `createMetadataBuilder()` to create a metadata builder object. It then invokes builder methods on that object to define the specific metadata to use, and it invokes `build()` to create a `collectionMetadata` object with that metadata. Finally, it creates a new collection that has this metadata.

In this case, the metadata that is specified, and the methods that define it, are as follows:

Method	Metadata
`mediaTypeColumnName()`	The media type column is to be named `MY_MEDIA_TYPE_COLUMN`. By default, there is no media type column.

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

Example 5-1 Creating a Collection That Has Custom Metadata

```
OracleRDBMSClient cl = new OracleRDBMSClient();
OracleRDBMSMetadataBuilder b = cl.createMetadataBuilder();
OracleDatabase db = cl.getDatabase(jdbcConnection);

// Create custom metadata
OracleDocument collectionMetadata =
  b.mediaTypeColumnName("MY_MEDIA_TYPE_COLUMN").
    build();

// Create a new collection with the specified custom metadata
db.admin().createCollection("collectionName", collectionMetadata);
```

SODA for Java Methods for Collection Metadata Components

The `OracleRDBMSMetadataBuilder` methods for selecting collection metadata components are described.

The `OracleRDBMSMetadataBuilder` methods for selecting collection metadata components have names similar to the components they select.

Table 5-1 Java Methods to Select Collection Metadata Components

Component	Method
Schema	`schemaName()`
Table or View	`tableName()` or `viewName()`
Key Column Name	`keyColumnName()`
Key Column Type	`keyColumnType()`
Key Column Max Length	`keyColumnMaxLength()`

Table 5-1 (Cont.) Java Methods to Select Collection Metadata Components

Component	Method
Key Column Assignment Method	`keyColumnAssignmentMethod()`
Key Column Sequence Name	`keyColumnSequenceName()`
Content Column Name	`contentColumnName()`
Content Column Type	`contentColumnType()`
Content Column Max Length	`contentColumnMaxLength()`
Content Column JSON Validation	`contentColumnValidation()`
Content Column SecureFiles LOB Compression	`contentColumnCompress()`
Content Column SecureFiles LOB Cache	`contentColumnCache()`
Content Column SecureFiles LOB Encryption	`contentColumnEncrypt()`
Version Column Name	`versionColumnName()`
Version Column Generation Method	`versionColumnMethod()`
Last-Modified Time Stamp Column Name	`lastModifiedColumnName()`
Last-Modified Column Index Name	`lastModifiedColumnIndex()`
Creation Time Stamp Column Name	`creationTimeColumnName()`
Media Type Column Name	`mediaTypeColumnName()`
Read Only	`readOnly()`

See Also:

- `OracleRDBMSMetadataBuilder` methods Javadoc for more information about collection metadata components
- *Oracle Database Introduction to Simple Oracle Document Access (SODA)*

Note:

The identifiers used for collection metadata components (schema name, table name, view name, database sequence name, and column names) must be valid Oracle quoted identifiers. Some characters and words that are allowed in Oracle quoted identifiers are strongly discouraged. For details, see *Oracle Database SQL Language Reference*.

[1] Reminder: letter case is significant for a quoted SQL identifier; it is interpreted case-sensitively.

A SODA for Java Core Interfaces

The SODA for Java core interfaces are described.

Table A-1 lists and briefly describes these interfaces. For complete information about them, see the SODA Javadoc.

Table A-1 SODA for Java Core Interfaces

Interface	Description
`OracleClient`	SODA for Java entry point (client)
`OracleDocument`	Document Content is typically JSON; possibly a MIME type (for example, image, audio, or video) Provides methods that get document content and metadata.
`OracleDatabase`	Database of collections of documents Provides methods that access `OracleDatabaseAdmin` and open existing collections. Inherits methods that create documents suitable for insertion into collections. Obtained by invoking `OracleClient.getDatabase()`.
`OracleDatabaseAdmin`	Provides methods that create collections and get their metadata. Obtained by invoking `OracleDatabase.admin()`.
`OracleCollection`	Collection of documents Provides methods that access `OracleOperationBuilder` and `OracleCollectionAdmin` and insert and save collection documents. Obtained by invoking `OracleDatabase.admin().createCollection()` or, if it already exists, `OracleDatabase.openCollection()`.
`OracleCollectionAdmin`	Provides methods that index and drop collections and get their metadata. Obtained by invoking `OracleDatabase.admin()`.

Table A-1 (Cont.) SODA for Java Core Interfaces

Interface	Description
`OracleOperationBuilder`	Builder and executor of read and write operations on a collection. Provides nonterminal methods for building operations (for example, `skip()` and `limit()`) and terminal methods for executing operations (for example, `getCursor()`, `count()`, and `remove()`). Obtained by invoking `OracleCollection.find()`, which returns an `OracleOperationBuilder` object that represents a query that finds all documents in the collection.
`OracleCursor`	Cursor for result list of query that `OracleCollection.find()` returns `next()` method returns the next document from the query result list. Obtained by invoking `OracleOperationBuilder.getCursor()`.

Index