MarkLogic Server

Node.js Application Developer's Guide

MarkLogic 9
June, 2019

Last Revised: 9.0-4, May 2019

Table of Contents

Node.js Application Developer's Guide

1.0 Introduction to the Node.js Client API

The Node.js Client API enables you to create Node.js applications that can read, write, and query documents and semantic data in a MarkLogic database.

The Node.js API is an open source project maintained on GitHub. To access the sources, report or review issues, or contribute to the project, go to http://github.com/marklogic/node-client-api.

1.1 Getting Started

This section demonstrates loading documents into the database, querying the documents, updating a portion of a document, and reading documents from the database. The basic features demonstrated here have many more capabilities. The end of this section contains pointers to resources for exploring the Node.js Client API in more detail.

Before you begin, make sure you have installed the software listed in “Required Software” on page 14. You should also have the `node` and `npm` commands on your path.

> **Note:** If you are working on Microsoft Windows, you should use a DOS command shell rather than a Cygwin shell. Cygwin is not a supported enviroment for `node` and `npm`.

The following procedure walks you through installing the Node.js Client API, loading some simple JSON documents into the database, and then searching and modifying the documents.

1. If you have not already done so, download, install, and start MarkLogic Server from http://developer.marklogic.com.

2. Create or select a project directory from which to exercise the examples in this walk through. The rest of the instructions assume you are in this directory.

3. Download and install the latest version of the Node.js Client API from the public npm repository into your project directory. For example:

```
npm install marklogic
```

4. Configure your MarkLogic connection information: Copy the following code to a file named `my-connection.js`. Modify the MarkLogic Server connection information to match your environment. You must change at least the `user` and `password` values. Select a MarkLogic user that has at least the `rest-reader` and `rest-writer` roles or equivalent privileges; for details, see "Security Requirements" on page 15.

```
module.exports = {
  connInfo: {
    host: 'localhost',
    port: 8000,
    user: 'user',
    password: 'password'
  }
};
```

   The rest of the examples in this guide assume this connection configuration module exists with the path `./my-connection.js`.

5. Load the example documents into the database: Copy the following script to a file and run it using the `node` command. Several JSON documents are inserted into the database using `DatabaseClient.documents.write`.

```
// Load documents into the database.

const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

// Document descriptors to pass to write().
const documents = [
  { uri: '/gs/aardvark.json',
    content: {
      name: 'aardvark',
      kind: 'mammal',
      desc: 'The aardvark is a medium-sized burrowing, nocturnal mammal.'
    }
  },
  { uri: '/gs/bluebird.json',
    content: {
      name: 'bluebird',
      kind: 'bird',
      desc: 'The bluebird is a medium-sized, mostly insectivorous bird.'
    }
  },
  { uri: '/gs/cobra.json',
    content: {
      name: 'cobra',
      kind: 'mammal',
      desc: 'The cobra is a venomous, hooded snake of the family Elapidae.'
    }
  },
];
```

```
// Load the example documents into the database
db.documents.write(documents).result(
  function(response) {
    console.log('Loaded the following documents:');
    response.documents.forEach( function(document) {
      console.log('  ' + document.uri);
    });
  },
  function(error) {
    console.log(JSON.stringify(error, null, 2));
  }
);
```

You should see output similar to the following:

```
Loaded the following documents:
  /gs/aardvark.json
  /gs/bluebird.json
  /gs/cobra.json
```

6. Search the database: Copy the following script to a file and run it using the `node` command. The script retrieves documents from the database that contain the JSON property `kind` with the value `'mammal'`.

```
// Search for documents about mammals, using Query By Example.
// The query returns an array of document descriptors, one per
// matching document. The descriptor includes the URI and the
// contents of each document.

const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const qb = marklogic.queryBuilder;

db.documents.query(
  qb.where(qb.byExample({kind: 'mammal'}))
).result( function(documents) {
    console.log('Matches for kind=mammal:')
    documents.forEach( function(document) {
      console.log('\nURI: ' + document.uri);
      console.log('Name: ' + document.content.name);
    });
}, function(error) {
```

```
    console.log(JSON.stringify(error, null, 2));
});
```

You should see output similar to the following. Notice that cobra is incorrectly labeled as a mammal. The next step will correct this error in the content.

```
Matches for kind=mammal:

URI: /gs/cobra.json
Name: cobra

URI: /gs/aardvark.json
Name: aardvark
```

7. Patch a document: Recall from the previous step that cobra is incorrectly labeled as a mammal. This step changes the `kind` property for `/gs/cobra.json` from `'mammal'` to `'reptile'`. Copy the following script to a file and run it using the `node` command.

```
// Use the patch feature to update just a portion of a document,
// rather than replacing the entire contents.

const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const pb = marklogic.patchBuilder;

db.documents.patch(
  '/gs/cobra.json',
  pb.replace('/kind', 'reptile')
).result( function(response) {
    console.log('Patched ' + response.uri);
}, function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

You should see output similar to the following:

```
Patched /gs/cobra.json
```

8. Confirm the change by re-running the search or retrieving the document by URI. To retrieve `/gs/cobra.json` by URI, copy the following script to a file and run it using the `node` command.

```
// Read documents from the database by URI.

const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read(
```

```
  '/gs/cobra.json'
).result( function(documents) {
  documents.forEach( function(document) {
    console.log(JSON.stringify(document, null, 2) + '\n');
  });
}, function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

You should see output similar to the following:

```
{
  "uri": "/gs/cobra.json",
  "category": "content",
  "format": "json",
  "contentType": "application/json",
  "contentLength": "106",
  "content": {
    "name": "cobra",
    "kind": "reptile",
    "desc": "The cobra is a venomous, hooded snake of the family Elapidae."
  }
}
```

9. Optionally, delete the example documents: Copy the following script to a file and run it using the `node` command. To confirm deletion of the documents, you can re-run the script from Step 8.

```
// Remove the example documents from the database.
// This example removes all the documents in the directory
// /gs/. You can also remove documents by document URI.

const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.removeAll(
  {directory: '/gs/'}
).result( function(response) {
  console.log(response);
});
```

You should see output similar to the following:

```
{ exists: false, directory: '/gs/' }
```

Document removal is an idempotent operation. Running the script again produces the same output.

To explore the API further, see the following resources:

If You Want To	Then See
Explore more examples	The examples and tests that are distributed with the API. Sources are available from http://github.com/marklogic/node-client-api or in your `node_modules/marklogic` directory after you install the API.
Learn about reading and writing documents and metadata	"Manipulating Documents" on page 33.
Learn about searching documents and querying lexicons and indexes	"Querying Documents and Metadata" on page 117. The Search Developer's Guide
Learn about extension points such as content transformations and resource service extensions	"Extensions, Transformations, and Server-Side Code Execution" on page 223
Explore the low level API documentation.	The Node.js API Reference. You can also generate a local copy of the API reference. For details, see the project page on GitHub: http://github.com/marklogic/node-client-api

1.2 Required Software

To use the Node.js Client API, you must have the following software:

- MarkLogic 8 or later. Features in version 2.0.x of the Node.js Client API can only be used with MarkLogic 9 or later.
- Node.js, version 6.3.1 or later. Node.js is available from http://nodejs.org.
- The Node.js Package Manager tool, `npm`. The latest version compatible with a supported Node.js version is recommended.
- If you plan to use Kerberos for authentication, you must have the MIT Kerberos software. For details, see "Using Kerberos Authentication" on page 30.

The examples in this guide assume you have the `node` and `npm` commands on your path.

1.3 Security Requirements

This describes the basic security model used by the Node.js Client API, and some common situations in which you might need to change or extend it. The following topics are covered:

- Basic Security Requirements
- Controlling Document Access
- Evaluating Requests Against a Different Database
- Evaluating or Invoking Server-Side Code

1.3.1 Basic Security Requirements

The user you specify when creating a `DatabaseClient` object must have appropriate URI privileges for the content accessed by the operations performed, such as permission to read or update documents in the target database.

The Node.js Client uses the MarkLogic REST Client API to communicate with MarkLogic Server, so it uses the same security model. In addition to proper URI privileges, the user must have one of the pre-defined roles listed below, or the equivalent privleges. The capabilities of each role in the table is subsumed in the roles below it.

Role	Description
`rest-extension-user`	Enables access to resource service extension methods. This role is implicit in the other pre-defined REST API roles, but you may need to explicitly include it when defining custom roles.
`rest-reader`	Enables read operations, such as retrieving documents and metadata. This role does not grant any other privileges, so the user might still require additional privileges to read content.
`rest-writer`	Enables write operations, such as creating documents, metadata, or configuration information. This role does not grant any other privileges, so the user might still require additional privileges to write content.
`rest-admin`	Enables administrative operations, such as creating an instance and managing instance configuration. This role does not grant any other privileges, so the user might still require additional privileges.

Some operations require additional privileges, such as using a database other than the default database associated with the REST instance and using `eval` or `invoke` methods of `DatabaseClient`. These requirements are detailed below.

1.3.2 Controlling Document Access

Documents you create using the Node.js Client API default roles have a read permission for the `rest-reader` role and an update permission for the `rest-writer` role. By default, users with the `rest-reader` role can read all documents created as `rest-reader` and users with the `rest-writer` role can write all documents created as `rest-writer`. You can override this behavior using document permissions and/or custom roles.

To restrict access to particular users, create custom roles rather than assigning users to the default `rest-*` roles. For example, you can use a custom role to restrict users in one group from seeing documents created by another.

For details, see Controlling Access to Documents and Other Artifacts in the *REST Application Developer's Guide*.

1.3.3 Evaluating Requests Against a Different Database

When you connect to a MarkLogic Server instance by creating a `DatabaseClient`, the REST instance you connect to has a default content database associated with it. You can specify an alternative database when you create the `DatabaseClient`, but to perform operations against an alternative database requires the `http://marklogic.com/xdmp/privileges/xdmp-eval-in` privilege or equivalent.

To enable your application to use a different database:

1. Create a role with the `xdmp:eval-in` execution privilege, in addition to appropriate mix of `rest-*` roles. (You can also add the privileges to an existing role.)

2. Assign the role from Step 1 to a user.

3. Create a `DatabaseClient` with the user from Step 2.

One simple way to achieve this is to inherit from one of the predefined `rest-*` roles and then addin the `eval-in` privileges.

For details about roles and privileges, see the *Security Guide*. To learn more about managing REST API instances, see "Administering REST API Instances" on page 263.

1.3.4 Evaluating or Invoking Server-Side Code

You can use the `DatabaseClient.eval` and `DatabaseClient.invoke` operations to evaluate arbitrary code on MarkLogic Server. These operations require special privileges instead of (or in addition to) the normal REST API roles like `rest-reader` and `rest-writer`.

For details, see "Required Privileges" on page 250.

1.4 Terms and Definitions

This guide uses the following terms and definitions:

Term	Definition
REST Client API	A MarkLogic API for developing applications that communicate with MarkLogic using RESTful HTTP requests. The Node.js Client API is built on top of the REST Client API.
REST API instance	A MarkLogic HTTP App Server specially configured to service REST Client API requests. The Node.js Client API requires a REST API instance. One is available on port 8000 as soon as you install MarkLogic. For details, see “What Is a REST API Instance?” on page 263.
npm	The Node.js package manager. Use npm to download and install the Node.js Client API and its dependencies.
builder	An interface in the Node.js Client API that exposes functions for building potentially complex data structures such as queries (`marklogic.queryBuilder`) and document patches (`marklogic.patchBuilder`).
Promise	A Promise is a JavaScript interface for interacting with the outcome of an asynchronous event. For details, see “Promise Result Handling Pattern” on page 20.
MarkLogic module	The module that encapsulates the Node.js Client API. Include the module in your application using `require()`. For details, see “MarkLogic Namespace” on page 18.
document descriptor	An object that encapsulates document content and metadata as named JavaScript object properties. For details, see “Document Descriptor” on page 19.
database client	A special object that encapsulates your connection to MarkLogic Server through a REST API instance. Almost all Node.js Client API operations take place through a database client object. For details, see “Creating a Database Client” on page 25.
git	A source control management system. You will need a git client if you want to checkout and use the Node.js Client API sources.
GitHub	The open source project repository that hosts the Node.js Client API project. For details, see http://github.com/.

1.5 Key Concepts and Conventions

1.5.1 MarkLogic Namespace

The Node.js Client API library exports a namespace that provides a database client factory method and access to builders such as `queryBuilder` (search), `valuesBuilder` (values queries), and `patchBuilder` (partial document updates).

To include the MarkLogic module in your code, use `require()` and bind the result to a variable. For example, you can include it by the name “marklogic” if you have installed in the module under your own Node.js project:

```
const ml = require('marklogic');
```

You can use any variable name, but the examples in this guide assume `ml`.

1.5.2 Parameter Passing Conventions

Node.js Client API functions that require many input parameter values accept these values as named properties of a call object. For example, you can specify a hostname, port, database name, and several other connection properties when calling the `createDatabaseClient()` method. Do so by encapsulating these values in a single object, such as the following:

```
ml.createDatabaseClient({host: 'some-host', port: 8003, ...});
```

Where a parameter value can have one or more values, the value of the property can be either a single value or an array of values. Some functions support either an array or a list. For example:

```
db.documents.write(docDescriptor)
db.documents.write([docDescriptor1, docDescriptor2, ...])
db.documents.write(docDescriptor1, docDescriptor2, ...)
```

Where a function has a parameter that is frequently used without other parameters, you can pass the parameter directly as a convenient alternative to encapsulating it in a call object. For example, `DatabaseClient.documents.remove` accepts either a call object that can have several properties, or a single URI string:

```
db.documents.remove('/my/doc.json')
db.documents.remove({uri: '/my/doc.json', txid: ...})
```

For details on a particular operation, see the Node.js API Reference.

1.5.3 Document Descriptor

A document descriptor is an object that encapsulates document content and metadata as named JavaScript object properties. Node.js Client API document operations such as `DatabaseClient.documents.read` and `DatabaseClient.documents.write` accept and return document descriptors.

A document descriptor usually includes at least the database URI and properties representing document content, document metadata, or both. For example, the following is a document descriptor for a document with URI `/doc/example.json`. Since the document is a JSON document, its contents can be expressed as a JavaScript object.

```
{ uri : 'example.json', content : {some : 'data'} }
```

Not all properties are always present. For example if you read just the contents of a document, there will be no metadata-related properties in the resulting document descriptor. Similarly, if you insert just content and the `collections` metadata property, the input descriptor will not include `permissions` or `quality` properties.

```
{ uri : 'example.json',
  content : {some : 'data'},
  collections : ['my-collection']
}
```

The `content` property can be an object, string, `Buffer`, or `ReadableStream`.

See `DocumentDescriptor` in the Node.js API Reference for a complete list of property names.

1.5.4 Supported Result Handling Techniques

Most functions in the Node.js Client API support the following ways of processing results returned by MarkLogic Server:

- Callback: Call the `result` function, passing in a success and/or error callback function. Use this pattern when you don't need to synchronize results. For example:

  ```
  db.documents.read(...).result(function(response) {...})
  ```

- Promise: Call the `result` function and process the results through a Promise. Use Promises to chain interactions together, such as writing documents to the database, followed by a search. Your success callback is not invoked until all the requested data has been returned by MarkLogic Server. For example:

  ```
  db.documents.read(...).result().then(function(response) {...})...
  ```

  For details, see "Promise Result Handling Pattern" on page 20.

- Object Mode Streaming: Call the `stream` function and process the results through a `Readable` stream. Your code gets control each time a document or other discrete part is received in full. If you're reading a JSON document, it is converted to a JavaScript object before invoking your callback. For example:

  ```
  db.documents.read(...).stream().pipe(...)
  ```

  For details, see "Stream Result Handling Pattern" on page 21.

- Chunked Mode Streaming: Call the `stream` function with a 'chunked' argument and process the results through a `Readable` stream. Your code gets control each time a sufficient number of bytes are accumulated, and the input to your callback is a byte stream.

  ```
  db.documents.read(...).stream('chunked').pipe(...)
  ```

  For details, see "Stream Result Handling Pattern" on page 21.

When you use the classic callback or promise pattern, your code does not get control until all results are returned by MarkLogic. This is suitable for operations that do not return a large amount of data, such as a read operation that returns a small number of documents or a write. Streaming is better suited to handling large files or a large number documents because it allows you to process results incrementally.

> **Note:** Errors in the user code of a success callback are handled in the next error callback. Therefore, you should include a catch clause to handle such errors. For details, see "Error Handling" on page 24.

1.5.5 Promise Result Handling Pattern

Node.js Client API functions return an object with a `result()` method that returns a Promise object. A Promise is a JavaScript interface for interacting with the outcome of an asynchronous event. A Promise has `then`, `catch`, and `finally` methods. For details, see http://promisesaplus.com/. Promises can be chained together to synchronize multiple operations.

The success callback you pass to the Promise `then` method is not invoked until your interaction with MarkLogic completes and all results are received. The Promise pattern is well suited to synchronizing operations.

For example, you can use a sequence such as the following to insert documents into the database, query them after the insertion completes, and then work with the query results.

```
db.documents.write(...).result().then(
  function(response) {
    // search the documents after insertion
    return db.documents.query(...).result();
  }).then( function(documents) {
    // work with the documents matched by the query
  });
```

For a more complete example, see “Example: Using Promises With a Multi-Statement Transaction” on page 220.

Note: You should include a catch clause in your promise chain to handle errors raised in user code in your success callbacks. For details, see “Error Handling” on page 24.

The Node.js Client API also supports a stream pattern for processing results. A stream is better suited to handling very large amounts of data than a Promise. For details, see “Stream Result Handling Pattern” on page 21.

1.5.6 Stream Result Handling Pattern

Node.js Client API functions return an object with a `stream` method that returns a `Readable` stream on the results from MarkLogic. Streams enable you to process results incrementally. Consider using streaming if you’re reading a large number of documents or if your documents are large.

Streams can provide better throughput at lower memory overhead than the Promises when you’re working with large amounts of data because result data can be processed as it is received from MarkLogic Server.

Two stream modes are supported:

- Object Mode: Your code gets control each time a complete document or other discrete part is received. A Document Descriptor is the unit of interaction. For a JSON document, the content in the descriptor is converted into JavaScript object for ease of use. Object mode is the default streaming mode.
- Chunked mode: Your code gets control each time a certain number of bytes is received. An opaque byte stream is the unit of interaction. Enable chunked mode by passing the value `'chunked'` to the stream method.

Object mode is best when you need to handle each portion of the result as a document or object. For example, if you persist a collection of domain objects in the database as JSON documents and then want to restore them as JavaScript objects in your application. Chunked mode is best for handling large amounts of data opaquely, such as reading a large binary file from the database and saving it out to file.

The following code snippet uses a stream in object mode to process multiple documents as they are fetched from the database. Each time a complete document is received, the stream `on('data')` callback is invoked with a document descriptor. When all documents are received, the `on('end')` callback is called.

```
db.documents.read(uri1, uri2, uriN).stream()
  .on('data', function(document) {
    // process one document
  }).on('end', function() {
    //wrap it up
  }).on('error', function(error) {
    // handle errors
  });
```

The following code snippet uses a stream in chunked mode to stream a large binary file from the database into a file using `pipe`.

```
const fs = require('fs');
const ostrm = fs.createWriteStream(outFilePath);

db.document.read(largeFileUri).stream('chunked').pipe(ostrm);
```

The Promise pattern is usually more convenient if you are not processing a large amount of data. For details, see “Promise Result Handling Pattern” on page 20.

1.5.7 Streaming Into the Database

Most Node.js methods that deal with potentially large input datasets support using a `ReadableStream` to pass in the data. For example , the content property of a document descriptor passed to `DatabaseClient.documents.write` can be an object, a string, a `Buffer`, or a `Readable` stream. If you’re simply streaming data from a source such as a file, this interface is all you need.

For example, the following call uses a `Readable` stream to stream an image from a file into the database:

```
db.documents.write({
  uri: '/my/image.png',
  contentType: 'image/png',
  content: fs.createReadStream(pathToImage)
})
```

If you are assembling the stream on the fly, or otherwise need to have fine grained control, you can use the `createWriteStream` method of the `documents` and `graphs` interfaces. For example, if you use `DatabaseClient.documents.createWriteStream` instead of `DatabaseClient.documents.write`, you can control the calls to `write` so you can assemble the documents yourself, as shown below:

```
const ws = db.documents.createWriteStream({
  uri: '/my/data.json',
  contentType: 'application/json',
```

```
});
// Resulting doc contains {"key":"value"}
ws.write('"{key"', 'utf8');
ws.write(': "value"}', 'utf8');
ws.end();
```

You can use the writeable stream interface to load documents and semantic graphs. For details, see `documents.createWriteStream` and `graphs.createWriteStream` in the Node.js API Reference.

1.5.8 Performing Point-in-Time Operations

If you need to perform read-only operations spanning multiple requests that must all return results based on a consistent snapshot of the database, you can use the "point-in-time query" feature of the Node.js Client API. In this context, "query" means a read-only operation, such as a search or document read.

Most read-only operations accept an optional `Timestamp` object, created by calling `DatabaseClient.createTimestamp`. If no explicit timestamp value is set on the object, then the timestamp is set during execution of the read-only operation.

Alternatively, you can supply an explicit timestamp when creating a timestamp. This must be a timestamp generated by MarkLogic, not an arbitrary value you create. To learn more about point-in-time queries (reads) and timestamps, see Point-In-Time Queries in the *Application Developer's Guide*.

When you pass a `Timestamp` object whose timestamp is set to subsequent supporting operations, these operations see the same snapshot of the database.

For example, suppose you are incrementally fetching search results in a context in which the database is changing and consistency of results is important. If you pass a `Timestamp` object on the search, then the effective query timestamp is captured in the `Timestamp` object and you can pass the object in to subsequent searches.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const qb = marklogic.queryBuilder;
let timestamp = db.createTimestamp();

// First search sets the timestamp value
db.documents.query(
  qb.where(qb.parsedFrom('cat AND dog')).slice(0,5),
  timestamp
).result().then( function(results) {
  console.log(JSON.stringify(results, null, 2));
});

// ...perform subsequent searches, re-using the same timestamp object
```

Another example use case is reading a large number of documents from the database by URI (or search query) in batches. If you need a consistent snapshot of the documents, use the point-in-time feature.

You can use this feature across different kinds of operations. For example you might get the initial timestamp from a search, and then use it to perform a SPARQL query at the same point-in-time.

This capability is supported on any operation that accepts a timestamp parameter, including the following:

- Document read: `DatabaseClient.documents.read`
- Document search: `DatabaseClient.documents.query`
- Values Query: `DatabaseClient.values.read`
- Semantic Search: `DatabaseClient.graphs.sparql`
- Semantic Update: `DatabaseClient.graphs.sparqlUpdate`
- Semantic Graph Access: `DatabaseClient.graphs.read` and `DatabaseClient.graphs.list`
- Rows Query: `DatabaseClient.rows.query`

For more details, see the *Node.js Client API Reference*.

1.5.9 Error Handling

When using the callback or promise pattern, errors in your success callback are handled in the next error callback. If you want to trap such errors, you should include a catch clause at the end of your promise chain (or after your result handler, in the case of the callback pattern). Simply wrapping a try-catch block around your call(s) will not trap such errors.

For example, in the case of the classic callback pattern, if you made a call to `DatabaseClient.documents.write`, you should end with a catch similar to the following. The `onError` function executes if the `onSuccess` callback throws an exception.

```
db.documents.write(...)
  .result(function onSuccess(response) {...})
  .catch(function onError(err) {...});
```

Similarly if you're chaining requests together using thePromise pattern, then you should terminate the chain with a similar handler:

```
db.documents.write(...).result()
  .then(function onSuccess1(response) {...})
  .then(function onSuccess2(response) {...})
  .catch(function onError(err) {...});
```

1.6 Creating a Database Client

All the interactions of your application with MarkLogic Server are through a `marklogic.DatabaseClient` object. Each database client manages a connection by one user to a REST API instance and a particular database. Your application can create multiple database clients for connecting to different REST API instances, connecting to different databases, or connecting as different users.

> **Note:** If you use multi-statement transactions and multiple databases, note that the database context in which you perform an operation as part of a multi-statement transaction must be the same as the database context in which the transaction was created. The same restriction applies to commiting or rolling back a multi-statement transaction.

To create a database client, call `marklogic.createDatabaseClient` with a parameter that describes the connection details. For example, the following code creates a database client attached to the REST API instance listening on the default host and port (`localhost:8000`), using the default database associated with the instance, and digest authentication. The connection authenticates as user “me” with password “mypwd”.

```
const ml = require('marklogic');
const db = ml.createDatabaseClient({user:'me', password:'mypwd'});
```

The connection details must include a username and password if you are not using certificate based authentication or Kerberos. You can include additional properties. The following table lists key properties you can include in the connection object passed to `createDatabaseClient`.

Property Name	DefaultValue	Description
`host`	`localhost`	A MarkLogic Server host with a configured REST API instance.
`port`	`8000`	The port on which the REST API instance listens.
`database`	the default database associated with the REST instance	The database against which document operations and queries are performed. Specifying a database other than the REST API instance default requires the xdmp-eval-in privilege. For details, see “Evaluating Requests Against a Different Database” on page 16.

Property Name	DefaultValue	Description
`authType`	`digest`	The authentication method to use in establishing the connection. Allowed values: `basic`, `digest`, `digestbasic`, `application-level`, or `kerberos-ticket`. This must match the authentication method configured on the REST API instance. For details, see the *Security Guide*.
`ssl`	`false`	Whether or not to establish an SSL connection. For details, see Configuring SSL on App Servers in the *Security Guide*. When set to true, you can include additional SSL properties on the connection object. These are passed through to the agent. For a list of these properties, see http://nodejs.org/api/https.html#https_https_request_options_callback
`agent`	max of 10 free sockets; total of 50 sockets kept alive for 60 seconds	A connection pooling agent.

For details, see `marklogic.createDatabaseClient` in the Node.js API Reference and "Administering REST API Instances" on page 263.

1.7 Authentication and Connection Security

This section provides an overview of how to configure authentication and SSL when creating a database client and establishing a connection to MarkLogic. This section covers the following topics:

- Connecting to MarkLogic with SSL
- Using SAML Authentication
- Using Certificate-Based Authentication
- Using Kerberos Authentication

1.7.1 Connecting to MarkLogic with SSL

You can combine an `ssl` property with any of the authentication methods so that your database client object uses a secure connection to MarkLogic. For example:

```
ml.createDatabaseClient({
  user: 'me',
  password: 'mypassword',
  authType: 'digest',
  ssl: true
})
```

Your App Server must be SSL-enabled. For details, see Configuring SSL on App Servers in the *Security Guide*.

The Node.js Client API must be able to verify that MarkLogic is sending a legitimate certificate when first establishing the connection. If the certificate is signed by an authority other than one of the established authorities like VeriSign, then you must include a certificate from the certification authority in your database client configuration. Use the `ca` property to specify a certificate authority. For example:

```
ml.createDatabaseClient({
  authType: 'certificate',
  ssl: true
  ca: fs.readFileSync('ca.crt')
})
```

For more details, see Procedures for Obtaining a Signed Certificate in the *Security Guide*.

1.7.2 Using SAML Authentication

Your client application is responsible for acquiring a SAML assertions token from the SAML Identity Provider (IDP). You can then use the SAML assertions received from a SAML IDP as well as HTTP access to a MarkLogic cluster configured to verify SAML assertions from the IDP.

Your client application sends the SAML assertions to the MarkLogic enode to invoke MarkLogic operations that are authorized for the user until the SAML assertions expire.

The `MarkLogic.createDatabaseClient` function uses the property values shown in the table below to authenticate using SAML.

Property Name	Purpose
`authType`	Specify the `SAML` as the authentication type.
`token`	The SAML assertions token to make requests to MarkLogic. This is required if the `authType` is `SAML`.

For example, after obtaining an authorization token (base64 encoded) from an IDP, the `MarkLogic.createDatabaseClient` function to create a client might look like the following.

```
const db = marklogic.createDatabaseClient({
            host: appserverHost,
            port: appserverPort,
            authType: 'SAML',
            token: authorizationToken,
          ... other configuration such as SSL ...
});
```

In addition, the database client object uses the `setAuthToken` function that takes a SAML assertions token. Requests made after the token is set use the new SAML assertions token.

Note: Unlike the Java API, the Node.js API doesn't support a reauthorizer or renewer callback. In Node.js, calls run to completion instead of blocking. Consequently, your client application can change the SAML assertions token without affecting requests that are about to be sent to the server.

1.7.3 Using Certificate-Based Authentication

When using certificate-based authentication, your client application obtains a certificate signed by a certificate authority, along with the certificate's private key. The certificate contains a public key and other information required to establish a connection.

See the following topics for details:

- Obtaining a Client Certificate
- Configuring Your App Server
- Examples: Database Client Configuration

Note: You can only use certificate-based authentication with the Node.js Client API when you connect to MarkLogic using SSL. For details, see "Connecting to MarkLogic with SSL" on page 27.

1.7.3.1 Obtaining a Client Certificate

You can use either a client certificate signed by an established certificate authority or a self-signed certificate. Choose one of the following options:

- Obtain a client certificate from an established certificate authority such as Verisign.
- Create a self-signed certificate.

To obtain a client certificate signed by an established certificate authority, create a certificate signing request (CSR) using OpenSSL software or a similar tool, then send the CSR to the certificate authority. For details, see http://openssl.org and man page for the `openssl req` command.

To create a self-signed certificate, install your own certificate authority in MarkLogic, and then use that certificate authority to self-sign your client certificate. For details, see Creating a Certificate Authority in the *Security Guide*.

To obtain a client certificate and the associated key by self-signing, use the `xdmp.x509CertificateGenerate` Server-Side JavaScript function or the `xdmp:x509-certificate-generate` XQuery function. Set the `private-key` parameter to null, and set the `credentialId` option to correspond to your certificate authority. For example:

```
const x509Config = ...;
const cert = xdmp.x509CertificateGenerate(
    x509Config, null, {credentialId: xdmp.credentialId('ca-cred')});
```

1.7.3.2 Configuring Your App Server

Your App Server must also be configured for certificate-based authentication and SSL. For more details, see Configuring an App Server for External Authentication and Procedures for Enabling SSL on App Servers in the *Security Guide*. When configuring the App Server for SSL, include the following steps; for more details, see Enabling SSL for an App Server in the *Security Guide*.

1. Set “ssl require client certificate” to true.

2. Click Show under “SSL Client Certificate Authorities”, and then select the certificate authorities that can be used to sign client certificates for the server

1.7.3.3 Examples: Database Client Configuration

For example, if you have a certificate in a file named “client.crt” and a private key in a file named “clientpriv.pem”, you can use them in your database client configuration as follows,:

```
ml.createDatabaseClient({
  authType: 'certificate',
  cert: fs.readFileSync('client.crt'),
  key: fs.readFileSync('clientpriv.pem'),
  ssl: true
})
```

For enhanced security, the client certificate and private key can also be combined into a single PKCS12 file that can be protected with a passphrase. For details, see http://openssl.org and the man page for the “openssl pkcs12” command.

For example, if you have a PKCS12 file named “credentials.pfx”, then you can use the file and your passphrase in your database client configuration as follows:

```
ml.createDatabaseClient({
  authType: 'certificate',
  pfx: fs.readFileSync('credentials.pfx'),
  key: 'yourPassphrase',
  ssl: true
})
```

You can also use a certificate with basic or digest authentication to enhance the security of these methods. For example, the following code uses a certificate with digest authentication:

```
ml.createDatabaseClient({
  user: 'me',
  password: 'mypassword',
  authType: 'digest',
  cert: fs.readFileSync('client.crt'),
  key: fs.readFileSync('clientpriv.pem'),
  ssl: true
})
```

1.7.4 Using Kerberos Authentication

Use the following steps to configure your MarkLogic installation and client application environment for Kerberos authentication:

- Configuring MarkLogic to Use Kerberos
- Configuring Your Client Host for Kerberos
- Creating a Database Client That Uses Kerberos

1.7.4.1 Configuring MarkLogic to Use Kerberos

Before you can use Kerberos authentication, you must configure MarkLogic to use external security. If your installation is not already configured for Kerberos, you must perform at least the following steps:

1. Create a Kerberos external security configuration object. For details, see Creating an External Authentication Configuration Object in the *Security Guide*.

2. Create a Kerberos keytab file and install it in your MarkLogic installation. For details, see Creating a Kerberos keytab File in the *Security Guide*.

3. Create one or more users associated with an external name. For details, see Assigning an External Name to a User in the *Security Guide*.

4. Configure your App Server to use “kerberos-ticket” authentication. For details, see Configuring an App Server for External Authentication in the *Security Guide*.

For more details, see External Security in the *Security Guide*.

1.7.4.2 Configuring Your Client Host for Kerberos

On the client, the Node.js Client API must be able to access a Ticket-Granting Ticket (TGT) from the Kerberos Key Distribution Center. The API uses the TGT to obtain a Kerberos service ticket.

Follow these steps to make a TGT available to the client application:

1. Install MIT Kerberos in your client environment if it is not already installed. You can download this software from http://www.kerberos.org/software/index.html.

2. If this is a new installation of MIT Kerberos, configure your installation by editing the `krb5.conf` file. On Linux, this file is located in `/etc/` by default. For details, see https://web.mit.edu/kerberos/krb5-1.15/doc/admin/conf_files/krb5_conf.html.

3. Use `kinit` on your client host to create and cache a TGT with the Kerberos Key Distribution Center. The principal supplied to `kinit` must be one you associated with a MarkLogic user when performing the steps in Configuring MarkLogic to Use Kerberos.

For more details, see the following topics:

- https://web.mit.edu/kerberos/krb5-1.15/doc/user/user_commands/kinit.html
- http://web.mit.edu/kerberos/krb5-current/doc/user/tkt_mgmt.html#obtaining-tickets-with-kinit

1.7.4.3 Creating a Database Client That Uses Kerberos

In your client application, set the `authType` property to 'kerberos' when creating a database client.

For example, assuming you're connecting to localhost on port 8000 and therefore don't need to explicitly specify host and port, then the following call creates a database client object that connects to localhost:8000 using kerberos authentication:

```
ml.createDatabaseClient({authType: 'kerberos'});
```

1.8 Using the Examples in This Guide

All requests to MarkLogic Server using the Node.js Client API go through a `DatabaseClient` object. Therefore, all the examples begin by creating such an object. Creating a `DatabaseClient` requires you to specify MarkLogic Server connection information such as host, port, user, and password.

Most of the examples in this guide abstract away the connection details by `require`'ing a module named `my-connection.js` that exports a connection object suitable for use with `marklogic.createDatabaseClient`. This encapsulation is only done for convenience. You are not required to do likewise in your application.

For example, the following statements appear near the top of each example in this guide:

```
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);
```

To use the examples you should first create a file named `my-connection.js` with the following contents. This file should be co-located with any scripts you create by copying the examples in this guide.

```
module.exports = {
  connInfo: {
    host: 'localhost',
    port: 8000,
    user: your-ml-username,
    password: your-ml-user-password
  }
};
```

Modify the connection details to match your environment. You must modify at least the `user` and `password` properties. Most examples require a user with the `rest-reader` and/or `rest-writer` role or equivalent, but some operations require additional privileges. For details, see “Security Requirements” on page 15.

If you do not create `my-connection.js`, modify the calls to `marklogic.createDatabaseClient` in the examples to provide connection details in another way.

2.0 Manipulating Documents

This chapter discusses the following topics related to using the Node.js Client API to create, read, update and delete documents and metadata:

2.1 Introduction to Document Operations

The Node.js Client API exposes functions for creating, reading, updating and deleting documents and document metadata.

Most document manipulation functions are provided through the `DatabaseClient.documents` interface. For example, the following code snippet reads a document by creating a database client object and invoking its `documents.read()` method:

```
const ml = require('marklogic');
const db = ml.createDatabaseClient({'user':'me','password':'mypwd'});

db.documents.read('/doc/example.json'). ...;
```

The `DatabaseClient` interface includes read and write operations for binding JavaScript objects to Database documents, such as `DatabaseClient.read` and `DatabaseClient.createCollection`. Generally, these operations provide a simpler but less powerful capability than the equivalent method of `DatabaseClient.documents`. For example, you cannot specify a transaction id or read document metadata using `DatabaseClient.read`.

Several of the `DatabaseClient.documents` interfaces accept or return document descriptors that encapsulate data such as the URI and document content. For details, see “Input Document Descriptors” on page 37 and “Document Descriptor” on page 19.

When loading data into the database, the `DatabaseClient.documents.write` method provides the most control and richest feature set. However, if you do not need that level control, one of the other interfaces may be simpler to use. For example, if you just want to save JavaScript domain objects in the database, `DatabaseClient.createCollection` enables you to do so without creating document descriptors or constructing document URIs.

By default, each Node.js Client API call that interacts with the database represents a complete transactional operation. For example, if you use a single call to `DatabaseClient.Documents.write` to update multiple documents, then all the updates are applied as part of the same transaction, and the transaction is committed when the operation completes on the server. You can use multi-statement transactions to have multiple client-side operations span a single transaction. For details, see “Managing Transactions” on page 216.

The following table lists some common tasks related to writing to the databaes, along with the method best suited for the completing the task. For a complete list of interfaces, see the Node.js API Reference.

If you want to	Then use
Save a collection of JavaScript objects in the database as JSON documents.	`DatabaseClient.createCollection` For details, see “Managing Collections of Objects and Documents” on page 54.
Update a collection of JavaScript objects created using `DatabaseClient.createCollection`.	`DatabaseClient.writeCollection` For details, see “Managing Collections of Objects and Documents” on page 54.
Insert or update a collection of documents by URI.	`DatabaseClient.writeCollection` For details, see “Managing Collections of Objects and Documents” on page 54.
Insert or update document metadata, with or without accompanying content.	`DatabaseClient.documents.write` For details, see “Inserting or Updating Metadata for One Document” on page 42.
Insert or update documents and/or metadata in the context of a multi-statement transaction.	`DatabaseClient.documents.write` For details, see “Loading Documents into the Database” on page 36.

If you want to	Then use
Apply a content transformation while loading documents.	`DatabaseClient.documents.write` For details, see “Loading Documents into the Database” on page 36 and “Transforming Content During Ingestion” on page 43.
Update a portion of a document or its metadata, rather than replacing the entire document.	`DatabaseClient.documents.patch` For details, see “Patching Document Content or Metadata” on page 70.

The following table lists some common tasks related to reading data from the database, along with the functions best suited for each task. For a complete list of interfaces, see the Node.js API Reference.

If you want to	Then use
Read the contents of one or more documents by URI.	`DatabaseClient.read`
Restore a collection of JavaScript objects previously saved in the in database using `DatabaseClient.createCollection`.	`DatabaseClient.documents.query` For details, see “Querying Documents and Metadata” on page 117 and “Managing Collections of Objects and Documents” on page 54.
Read one or more documents and/or metadata by URI.	`DatabaseClient.documents.read` For details, see “Reading Documents from the Database” on page 44.
Read the contents of one or more documents and/or metadata by URI and apply a read transformation.	`DatabaseClient.documents.read` For details, see “Reading Documents from the Database” on page 44 and “Transforming Content During Retrieval” on page 50.
Read one or more documents and/or metadata by URI in the context of a multi-statement transaction.	`DatabaseClient.documents.read` For details, see “Reading Documents from the Database” on page 44.

If you want to	Then use
Read documents and/or metadata that match a query.	`DatabaseClient.documents.query` For details, see “Querying Documents and Metadata” on page 117.
Query and analyze values in lexicons and range indexes. For details, see “Querying Lexicons and Range Indexes” on page 152.	`DatabaseClient.values.read`
Read a semantic graph from the database. For details, see Node.js API Reference.	`DatabaseClient.graphs.read`

2.2 Loading Documents into the Database

Use the `DatabaseClient.documents.write` or `DatabaseClient.documents.createWriteStream` methods to insert document content and metadata into the database. The stream interface is primarily intended for writing large documents such as binaries.

- Overview
- Input Document Descriptors
- Calling Convention
- Example: Loading A Single Document
- Example: Loading Multiple Documents
- Inserting or Updating Metadata for One Document
- Automatically Generating Document URIs
- Transforming Content During Ingestion

2.2.1 Overview

Use `DatabaseClient.documents.write` to insert or update whole documents and/or metadata. To update only a portion of a document or its metadata, use `DatabaseClient.documents.patch`; for details, see “Patching Document Content or Metadata” on page 70.

The primary input to the `write` function is one or more document descriptors. Each descriptor encapsulates a document URI with the content and/or metadata to be written. For details, see “Input Document Descriptors” on page 37.

For example, the following call writes a single document with the URI `/doc/example.json`:

```
const db = marklogic.createDatabaseClient(...);
db.documents.write(
  { uri: '/doc/example.json',
```

```
    contentType: 'application/json',
    content: { some: 'data' }
  }
);
```

Write multiple documents by passing in multiple desriptors. For example, the following call writes 2 documents:

```
db.documents.write(
    { uri: '/doc/example1.json',
      contentType: 'application/json',
      content: { data: 'one' }
    },
    { uri: '/doc/example2.json',
      contentType: 'application/json',
      content: { data: 'two' }
    },
);
```

Descriptors can be passed as individual parameters, in an array, or in an encapsulating call object. For details, see “Calling Convention” on page 38.

You can take action based on the success or failure of a write operation by calling the `result()` function on the return value. For details, see “Supported Result Handling Techniques” on page 19.

For example, the following code snippet prints an error message to the console if the write fails:

```
db.documents.write(
  { uri: '/doc/example.json',
    contentType: 'application/json',
    content: { some: 'data' }
  }
).result(null, function(error) {
    console.log(
})
```

2.2.2 Input Document Descriptors

Each document to be written is described by a document descriptor. The document descriptor must include a URI and either content, metadata, or both content and metadata. For details, see “Document Descriptor” on page 19.

For example, the following is a document descriptor for a document with URI `/doc/example.json`. The document contents are expressed as a JavaScript object containing a single property.

```
{ uri: '/doc/example.json', content: {'key': 'value'} }
```

The `content` property in a document descriptor can be an object, a string, a `Buffer`, or a `ReadableStream`.

Metadata is expressed as properties of a document descriptor when it applies to a specific document. Metadata is expressed as properties of a call object when it applies to multiple documents; for details, see “Inserting or Updating Metadata for One Document” on page 42.

For example, the following document descriptor includes collection and document quality metadata for the document with URI `/doc/example.json`:

```
{ uri: '/doc/example.json’,
  content: {'key': 'value'},
  collections: [ 'collection1', 'collection2' ],
  quality: 2
}
```

2.2.3 Calling Convention

You must pass at least one document descriptor to `DatabaseClient.documents.write`. You can also include additional properties such as a transform name or a transaction id. The parameters passed to `documents.write` can take one of the following forms:

- One or more document descriptors: `db.documents.write(desc1, desc2,...)`.
- An array of one or more document descriptors: `db.documents.write([desc1, desc2, ...])`.
- A call object that encapsulates a document descriptor array and additional optional properties: `db.documents.write({documents: [desc1, desc2, ...], txid: ..., ...})`.

The following calls are equivalent:

```
// passing document descriptors as parameters
db.documents.write(
  {uri: '/doc/example1.json', content: {...}},
  {uri: '/doc/example2.json', content: {...}}
);

// passing document descriptors in an array
db.documents.write([
  {uri: '/doc/example1.json', content: {...}},
  {uri: '/doc/example1.json', content: {...}}
]);

// passing document descriptors in a call object
db.documents.write({
  documents: [
    {uri: '/doc/example1.json', content: {...}},
    {uri: '/doc/example2.json', content: {...}}
  ],
  additional optional properties
});
```

The additional optional properties can include a transform specification, transaction id, or temporal collection name; for details, see the Node.js API Reference. You can always specify such properties as properties of a call object.

For example, the following call includes a transaction id (`txid`) as an additional property of the call object:

```
// passing a transaction id as a call object property
db.documents.write({
  documents: [
    {uri: '/doc/example1.json', content: {...}},
    {uri: '/doc/example2.json', content: {...}}
  ],
  txid: '1234567890'
});
```

For convenience, if and only if there is a single document descriptor, the additional optional properties can be passed as properties of the document descriptors, as an alternative to using a call object. For example, the following call includes a transaction id inside the single document descriptor:

```
// passing a transaction id as a document descriptor property
db.documents.write(
  { uri: '/doc/example1.json',
    content: {...},
    txid: '1234567890'
  }
);
```

2.2.4 Example: Loading A Single Document

This example inserts a single document into the database using `DatabaseClient.documents.write`.

The document to load is identified by a document descriptor. The following document descriptor describes a JSON document with the URI `/doc/example.json`. The document content is expressed as a JavaScript object here, but it can also be a string, `Buffer`, or `ReadableStream`.

```
{ uri: '/doc/example.json',
  contentType: 'application/json',
  content: { some: 'data' }
})
```

The code below creates a database client and calls `DatabaseClient.documents.write` to load the document. The example checks for a write failure by calling the `result` function and passing in an error handler. In this example, no action is taken if the write succeeds, so `null` is passed as the first parameter to `result()`.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);
```

```
db.documents.write(
    { uri: '/doc/example.json',
      contentType: 'application/json',
      content: { some: 'data' }
    })
  .result(null, function(error) {
      console.log(JSON.stringify(error));
    });
```

For additional examples, see `examples/before-load.js` and `examples/write-remove.js` in the `node-client-api` source directory.

To include metadata, add metadata properties to the document descriptor. For example, to add the document to a collection, you can add a `collections` property to the descriptor:

```
db.documents.write(
    { uri: '/doc/example.json',
      contentType: 'application/json',
      content: { some: 'data' },
      collections: [ 'collection1', 'collection2' ]
    })
```

You can include optional additional parameters such as a transaction id or a write transform by using a call object. For details, see “Calling Convention” on page 38.

2.2.5 Example: Loading Multiple Documents

This example builds on “Example: Loading A Single Document” on page 39 to insert multiple documents into the database using `DatabaseClient.documents.write`.

To insert or update multiple documents in a single request to MarkLogic Server, pass multiple document descriptors to `DatabaseClient.documents.write`.

The following code inserts 2 documents into the database with URIs `/doc/example1.json` and `/doc/example2.json`:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write(
    { uri: '/doc/example1.json',
      contentType: 'application/json',
      content: { data: 'one' }
    },
    { uri: '/doc/example2.json',
      contentType: 'application/json',
      content: { data: 'two' }
    }
).result(null, function(error) {
```

```
    console.log(JSON.stringify(error));
});
```

A multi-document write returns an object that contains a descriptor for each document written. The descriptor includes the URI, the MIME type the contents were interpreted as, and whether the write updated content, metadata, or both.

For example, the return value of the above call is as follows:

```
{ documents: [
  { uri: '/doc/example1.json',
    mime-type: 'application/json',
    category: ['metadata','content']
  },{
    uri: '/doc/example2.json',
    mime-type: 'application/json',
    category: ['metadata','content']
  }
]}
```

Note that the `category` property indicates both content and metadata were updated even though no metadata was explicitly specified. This is because system default metadata values were implicitly assigned to the documents.

To include metadata for a document when you load multiple documents, include document-specific metadata in the descriptor for that document. To specify metadata that applies to multiple documents include a metadata descriptor in the parameter list or documents property.

For example, to add the two documents to the collection “examples”, add a metadata descriptor before the document descriptors, as shown below. The order of the descriptors matters as the set of descriptors is processed in the order it appears. A metadata descriptor only affects document descriptors that appear after it in the parameter list or `documents` array.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write({
  documents: [
    { contentType: 'application/json',
      collections: [ 'examples' ]
    },
    { uri: '/doc/example1.json',
      contentType: 'application/json',
      content: { data: 'one' }
    },
    { uri: '/doc/example2.json',
      contentType: 'application/json',
      content: { data: 'two' }
    }
  ]
```

```
}).result(null, function(error) {
      console.log(JSON.stringify(error));
});
```

2.2.6 Inserting or Updating Metadata for One Document

To insert or update metadata for a specific document, include one or more metadata properties in the document descriptor passed to `DatabaseClient.documents.write`. To insert or update the same metadata for multiple documents, you can include a metadata descriptor in a multi-document write; for details, see "Working with Metadata" on page 64.

Note: When setting permissions, at least one update permission must be included.

Metadata is replaced on update, not merged. For example, if your document descriptor includes a `collections` property, then calling `DatabaseClient.documents.write` replaces all existing collection associations for the document.

The following example inserts a document with URI `/doc/example.json` and adds it to the collections "examples" and "metadata-examples". If the document already exists and is part of other collections, it is removed from those collections.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write(
    { uri: '/doc/example.json',
      collections: ['examples', 'metadata-examples'],
      contentType: 'application/json',
      content: { some: 'data' }
    })
  .result(null, function(error) {
      console.log(JSON.stringify(error));
    });
```

To insert or update just metadata for a document, omit the `content` property. For example, the following code sets the quality to 2 and the collections to "some-collection", without changing the document contents:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write(
    { uri: '/doc/example.json',
      collections: ['some-collection'],
      quality: 2,
    })
  .result(null, function(error) {
      console.log(JSON.stringify(error));
    });
```

2.2.7 Automatically Generating Document URIs

You can have document URIs automatically generated during insertion by replacing the `uri` property in your document descriptor with an `extension` property, as described below.

> **Note:** You can only use this feature to create new documents. To update an existing document, you must know its URI.

To use this feature, construct a document descriptor with the following characteristics:

- Omit the `uri` property.
- Include an `extension` property that specifies the generated URI extension, such as “xml” or “json”. Do not include a “dot” (.) prefix. That is, specify “json”, not “.json”.
- Optionally, include a `directory` property that specifies a database directory prefix for the generated URI. The directory prefix must end in a forward slash (/).

The following example inserts a document into the database with a URI of the form `/my/directory/`*`auto-generated`*`.json`.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write(
    { extension: 'json',
      directory: '/my/directory/',
      content: { some: 'data' },
      contentType: 'application/json'
    }
).result(
    function(response) {
      console.log('Loaded ' + response.documents[0].uri);
    },
    function(error) {
      console.log(JSON.stringify(error));
    }
);
```

Running the above script results in output similar to the following upon success:

```
Loaded /my/directory/16764526972136717799.json
```

2.2.8 Transforming Content During Ingestion

You can transform content during ingestion by applying a custom write transform. A transform is server-side XQuery, JavaScript, or XSLT that you install in the modules database associated with your REST API instance. You can install transfoms using the `config.transforms` functions. This topic describes how to apply a transform during ingestion. For more details and examples, see “Working with Content Transformations” on page 233.

To apply a transform when creating or updating documents, call `documents.write` with a call object that includes the `transform` property. The `transform` property encapsulates the transform name and any parameters expected by the transform. The `transform` property has the following form:

```
transform: [transformName, {param1: value, param2: value, ...}]
```

For example, the following code snippet applies a transform installed under the name `my-transform` and passes in values for 2 parameters:

```
db.documents.write({
  documents: [
    {uri: '/doc/example1.json', content: {...}},
    {uri: '/doc/example2.json', content: {...}}
  ],
  transform: [
    'my-transform',
    { my-first-param: 'value',
      my-second-param: 42
    }
  ]
});
```

As a convenience, you can embed the `transform` property in the document descriptor when inserting or updating just one document. For example:

```
db.documents.write({
  uri: '/doc/example1.json',
  content: {...}},
  transform: [
    'my-transform',
    { my-first-param: 'value',
      my-second-param: 42
    }
  ]
});
```

2.3 Reading Documents from the Database

Use `DatabaseClient.documents.read` to read one or more documents and/or metadata from the database. This section covers the following topics:

- Retrieving the Contents of a Document By URI
- Retrieving Metadata About a Document
- Example: Retrieving Content and Metadata
- Transforming Content During Retrieval

2.3.1 Retrieving the Contents of a Document By URI

To retrieve the contents of a document from the database using its URI, use `DatabaseClient.documents.read` or `DatabaseClient.read`. You can also retrieve the contents and metadata of documents that match a query; for details, see "Querying Documents and Metadata" on page 117.

`DatabaseClient.read` and `DatabaseClient.Documents.read` both enable you to read documents by URI, but they differ in power and complexity. If you need to read metadata, use a multi-statement transaction, apply a read transformation, or access information such as the content-type, use `DatabaseClient.Documents.read`. `DatabaseClient.read` accepts only a list of URIs as input and returns just the contents of the requested documents.

The two functions return different output. `DatabaseClient.Documents.read` returns an array of document descriptors, instead of just the content. `DatabaseClient.read` returns an array containing only the content of each document, in the same order as the input URI list. You can process the output of both functions using a callback, Promise, or Stream; for details, see "Supported Result Handling Techniques" on page 19.

For example, the following code uses `DatabaseClient.Documents.read` to read the document with URI `/doc/example1.json` and processes the output using the Promise returned by the `result` method.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read('/doc/example1.json')
   .result( function(documents) {
      documents.forEach(function(document) {
        console.log(JSON.stringify(document));
      });
    }, function(error) {
      console.log(JSON.stringify(error, null, 2));
    });
```

The complete descriptor returned by `result` looks like the following. The returned array contains a document descriptor item for each document returned. In this case, there is only a single document.

```
{
  "partType":"attachment",
  "uri":"/doc/example1.json",
  "category":"content",
  "format":"json",
  "contentType":"application/json",
  "contentLength":"14",
  "content":{"data":"one"}
}
```

If you read the same document using `DatabaseClient.read`, you get the output shown below. Notice that it is just the content, not a descriptor.

```
db.read('/doc/example1.json').result(...);
==>
{"data":"one"}
```

You can read multiple documents by passing in multiple URIs. The following example reads two documents and uses the Stream pattern to process the results. The data handler receives a document descriptor as input.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read('/doc/example1.json', '/doc/example2.json')
  .stream().on('data', function(document) {
    console.log('Read ' + document.uri);
  }).
  on('end', function() {
    console.log('finished');
  }).
  on('error', function(error) {
    console.log(JSON.stringify(error));
    done();
  });
```

You can request metadata by passing a call object parameter to `Documents.read` and including a `categories` property that specifies which document parts to return. For details, see “Retrieving Metadata About a Document” on page 46.

When calling `Documents.read`, you can use a read transform to apply server-side transformations to the content before the response is constructed. For details, see “Transforming Content During Retrieval” on page 50.

To perform reads across multiple operations with a consistent view of the database state, pass a Timestamp object to Documents.read. For more details, see “Performing Point-in-Time Operations” on page 23.

2.3.2 Retrieving Metadata About a Document

To retrieve metadata when reading documents by URI, pass a call object to `DatabaseClient.documents.read` that includes a `categories` property. You can retrieve all metadata (category value `'metadata'`) or a subset of metadata (category values `'collections'`, `'permissions'`, `'properties'`, `'metadataValues'`, and `'quality'`). To retrieve both content and metadata, include the category value `'content'`.

Note: The `metadataValues` metadata category represents simple key-value metadata, sometimes called metadata fields. For more details, see Metadata Fields in the *Administrator's Guide*.

For example, the following code retrieves all metadata about the document `/doc/example.json`, but not the contents.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read({
  uris: ['/doc/example.json'],
  categories: ['metadata']
}).result(
    function(documents) {
      for (const i in documents)
        console.log('Read metadata for ' + documents[i].uri);
    },
    function(error) {
      console.log(JSON.stringify(error));
    }
);
```

The result is a document descriptor that includes all metadata properties for the requested document, as shown below. For details on metadata categories and formats, see "Working with Metadata" on page 64.

```
[{
  "partType":"attachment",
  "uri":"/doc/example.json",
  "category":"metadata",
  "format":"json",
  "contentType":"application/json",
  "contentLength":"168",
  "collections":[],
  "permissions":[
    {"role-name":"rest-writer","capabilities":["update"]},
    {"role-name":"rest-reader","capabilities":["read"]}
  ],
  "properties":{},
  "quality":0
}]
```

The following example retrieves content and collections metadata about 2 JSON documents:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read({
  uris: ['/doc/example1.json', '/doc/example2.json'],
```

```
  categories: ['collections', 'content']
}).stream().
  on('data', function(document) {
    console.log('Collections for ' + document.uri + ': '
      + JSON.stringify(document.collections));
  }).
  on('end', function() {
    console.log('finished');
  }).
  on('error', function(error) {
    console.log(JSON.stringify(error));
  });
```

The result is a document descriptor for each document that includes a `collections` and a `content` property:

```
{
  "partType" : "attachment",
  "uri" : "/doc/example2.json",
  "category" : "content",
  "format" : "json",
  "contentType" : "application/json",
  "contentLength" : "14",
  "collections" : ["collection1", "collection2"],
  "content" : {"data":"two"}
}
```

2.3.3 Example: Retrieving Content and Metadata

The example demonstrates reading content and metadata for a document.

The script below first writes an example document to the database that is in the “examples” collection and has document quality 2. Then, the document and metadata are read back in a single operation by including both `'content'` and `'metadata'` in the `categories` property of the read document descriptor. You can also specify specific metadata properties, such as `'collections'` or `'permissions'`.

To run the example, copy the script below to a file and run it using the node command.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);

// (1) Seed the database with an example document that has custom
metadata
db.documents.write({
  uri: '/read/example.json',
  contentType: 'application/json',
  collections: ['examples'],
  metadataValues: {key1: 'val1', key2: 2},
  quality: 2,
```

```
  content: { some: 'data' }
}).result().then(function(response) {
  // (2) Read back the content and metadata
  return db.documents.read({
    uris: [response.documents[0].uri],
    categories: ['content', 'metadata']
  }).result();
}).then(function(documents) {
  // Emit the read results
  console.log('CONTENT: ' +
              JSON.stringify(documents[0].content));
  console.log('COLLECTIONS: ' +
              JSON.stringify(documents[0].collections));
  console.log('PERMISSIONS: ' +
              JSON.stringify(documents[0].permissions, null, 2));
  console.log('PROPERTIES: ' +
              JSON.stringify(documents[0].properties, null, 2));
  console.log('QUALITY: ' +
              JSON.stringify(documents[0].quality, null, 2));
  console.log("METADATAVALUES: " +
              JSON.stringify(documents[0].metadataValues, null, 2));

});
```

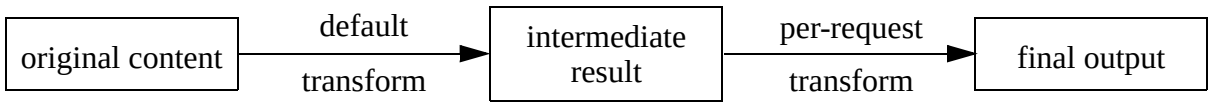
The script produces output similar to the following:

```
CONTENT: {"some": "data"}
COLLECTIONS: ["examples"]
PERMISSIONS: [
  {
    "role-name": "rest-writer",
    "capabilities": [
      "update"
    ]
  },
  {
    "role-name": "rest-reader",
    "capabilities": [
      "read"
    ]
  }
]
PROPERTIES: {}
QUALITY: 2
METADATAVALUES: {
  "key2": 2,
  "key1": "val1"
}
```

2.3.4 Transforming Content During Retrieval

You can apply custom server-side transforms to a document before returning the content to the requestor. A transform is a JavaScript module, XQuery module, or XSLT stylesheet that you install in your REST API instance using the `DatabaseClient.config.transforms.write` function. For details, see "Working with Content Transformations" on page 233.

You can configure a default read transform that is automatically applied whenever a document is retrieved. You can also specify a per-request transform by including a `transform` property in the call object passed to most read operations. If there is both a default transform and a per-request transform, the transforms are chained together, with the default transform running first. Thus, the output of the default transform is the input to the per-request transform, as shown in the following diagram:



To configure a default transform, set the `document-transform-out` configuration parameter of the REST API instance. Instance-wide configuration parameters are set using the `DatabaseClient.config.serverprops.write` function. For details, see "Configuring Instance Properties" on page 264.

To specify a per-request transform, use the call object form of a read operation such as `DatabaseClient.documents.read` and include a `transform` property. The value of the `transform` property is an array where the first item is the transform name and the optional additional items specify the name and value of parameters expected by the transform. That is, the `transform` property has the following form:

```
transform: [transformName, {param1: value, param2: value, ...}]
```

The following example applies a transform installed under the name "example" that expects a single parameter named "reviewer". For a complete example, see "Example: Read, Write, and Query Transforms" on page 237.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read({
    uris: ['/doc/example.json'],
    transform: ['example', {reviewer: 'me'}]
  }).result(
    function(documents) {
      for (const i in documents)
        console.log('Document ' + documents[i].uri + ': ');
        console.log(JSON.stringify(documents[i].content));
```

```
    }
);
```

2.4 Removing Content from the Database

You can use the Node.js Client API to remove documents by URI, collection or directory.

- Removing Documents By URI
- Removing Sets of Documents
- Removing All Documents

2.4.1 Removing Documents By URI

To remove one or more documents by URI, use `DatabaseClient.remove` or `DatabaseClient.documents.remove`. Both functions enable you to remove documents by URI, but `DatabaseClient.remove` offers a simpler, more limited, interface. You can also remove multiple documents by collection or directory; for details, see “Removing Sets of Documents” on page 52.

Removing a document also removes the associated metadata. Removing a binary document with extracted metadata stored in a separate XHTML document also deletes the properties document; for details, see “Working with Binary Documents” on page 61.

The following example removes the documents `/doc/example1.json` and `/docs/example2.json`.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.remove('/doc/example1.json','/doc/example2.json').result(
    function(response) {
      console.log(JSON.stringify(response));
    }
);
```

The response contains the document URIs and a success indicator. For example, the above program produces the following output:

```
{ "uris":["/doc/example1.json", "/doc/example2.json"],
  "removed":true }
```

When you remove documents with `DatabaseClient.remove`, the response only includes the URIs. For example:

```
db.remove('/doc/example1.json')
==> ['/doc/example1.json']

db.remove('/doc/example1.json','/doc/example2.json')
==> ['/doc/example1.json','/doc/example2.json']
```

For additional examples, see the examples and tests in the `node-client-api` sources available on GitHub at http://github.com/marklogic/node-client-api.

Attempting to remove a document that does not exist produces the same output as successfully removing a document that exists.

When removing multiple documents, you can specify the URIs as individual parameters (if the call has no other parameters), an array of URIs, or an object with a `uris` property whose value is the URIs. When removing multiple documents, the whole batch fails if there is an error removing any one of the documents.

You can supply additional parameters to `DatabaseClient.documents.remove`, such as a transaction id and temporal collection information. For details, see the Node.js API Reference.

2.4.2 Removing Sets of Documents

You can use `DatabaseClient.documents.removeAll` to remove all documents in a collection or all documents in a database directory.

To remove documents by collection, use the following form:

```
db.documents.removeAll({collection:..., other-properties...}
```

To remove documents by directory, use the following form:

```
db.documents.removeAll({directory:..., other-properties...}
```

The optional *other-properties* can include a transaction id. For details, see the Node.js API Reference.

Removing all documents in a collection or directory requires the `rest-writer` role or equivalent privileges.

> **Note:** When removing documents by directory, the directory name must include a trailing slash (“/”).

The following example removes all documents in the collection “/countries”:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.removeAll({collection: '/countries'}).result(
    function(response) {
      console.log(JSON.stringify(response));
    }
);

==> {"exists":false,"collection":"/countries"}
```

The following example removes all documents in the directory "/doc/":

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.removeAll({directory: '/doc/'}).result(
    function(response) {
      console.log(JSON.stringify(response));
    }
);

==> {"exists":false,"directory":"/doc/"}
```

You can also include a `txid` property in the call object passed to `DatabaseClient.documents.removeAll` to specify a transaction in which to perform the deletion. For example:

```
db.documents.removeAll({directory: '/doc/', txid: '1234567890'})
```

For additional examples, see the examples and tests in the `node-client-api` sources available on GitHub at http://github.com/marklogic/node-client-api.

2.4.3 Removing All Documents

To remove all documents in the database, call `DatabaseClient.documents.removeAll` and include an all property with value true in the call object. For example:

```
db.documents.removeAll({all: true})
```

Removing all documents in the database requires the `rest-admin` or equivalent privileges.

There is no confirmation or other safety net when you clear the database in this way. Creating a backup is recommended.

The following example removes all documents in the database:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.removeAll({all: true}).result(
    function(response) {
      console.log(JSON.stringify(response));
    }
);

==> {"exists":false,"allDocuments":true}
```

You can also include a `txid` property in the call object passed to `DatabaseClient.documents.removeAll` to specify a transaction in which to perform the deletion. For example:

```
db.documents.removeAll({all: true, txid: '1234567890'})
```

2.5 Managing Collections of Objects and Documents

You can easily manage a collection of JavaScript objects in the database using the following operations. The objects must be serializable as JSON.

- `DatabaseClient.createCollection`: Store a collection of JavaScript objects in the database as JSON documents with auto-generated URIs. The objects must be serializable as JSON.
- `DatabaseClient.read`: Read one or more JavaScript objects from the database by URI. Unlike `DatabaseClient.documents.read`, this method does not return document descriptors. Rather, it returns just the document content. Documents are returned in the same order as the input URIs.
- `DatabaseClient.documents.query`: Restore objects by finding all documents in the collection, or search your collection.
- `DatabaseClient.writeCollection`: Update objects or other documents by URI and collection name. Use this method rather than `creatCollection` to update objects in the collection because `createCollection` always creates a new document for each object.
- `DatabaseClient.removeCollection`: Remove objects or other documents by collection name.

Calling `createCollection` is additive. That is, documents (objects) already in the collection remain in the collection if you call `createCollection` multiple times on the same collection. However, note that `createCollection` generates a new document for each object every time you call it, so calling it twice on the same object creates a new object rather than overwriting the previous one. To update objects/documents in the collection, use `DatabaseClient.writeCollection`.

If you need more control over your documents, use `DatabaseClient.documents.write`. For example, when you use `createCollection` you cannot exercise any control over the document URIs, include metadata such as permissions or document properties, or specify a transaction id.

To learn more about searching documents with `DatabaseClient.documents.query`, see “Querying Documents and Metadata” on page 117.

The example script below does the following:

- Create a collection from a set of objects.
- Read all the objects back.
- Find the just the objects where `kind='cat'`.

- Remove the collection.

To try the example, copy the following script to a file and run it with the `node` command. The database connection information is encapsulated in `my-connection.js`, as described in "Using the Examples in This Guide" on page 31.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const qb = marklogic.queryBuilder;

// The collection of objects to persist
const pets = [
  { name: 'fluffy', kind: 'cat' },
  { name: 'fido', kind: 'dog' },
  { name: 'flipper', kind: 'fish' },
  { name: 'flo', kind: 'rodent' }
];
const collName = 'pets';

// (1) Write the objects to the database
db.createCollection(collName, pets).result()
  .then(function(uris) {
    console.log('Saved ' + uris.length + ' objects with URIs:');
    console.log(uris);

    // (2) Read back all objects in the collection
    return db.documents.query(
      qb.where(qb.collection(collName))
    ).result();
  }, function(error) {
    console.log(JSON.stringify(error));
  }).then( function(documents) {
    console.log('\nFound ' + documents.length + ' documents:');
    documents.forEach( function(document) {
      console.log(document.content);
    });

    // (3) Find the cats in the collection
    return db.documents.query(
      qb.where(qb.collection(collName), qb.value('kind', 'cat'))
    ).result();
  }).then( function(documents) {
    console.log('\nFound the following cats:');
    documents.forEach( function(document) {
      console.log('  ' + document.content.name);
    });

    // (4) Remove the collection from the database
    db.removeCollection(collName);
  });
```

Running the script produces output similar to the following:

```
Saved 4 objects with URIs:
[ '/717293155828968327.json',
  '/5648624202818659648.json',
  '/4552049485172923004.json',
  '/16796864305170577329.json' ]

Found 4 documents:
{ name: 'fido', kind: 'dog' }
{ name: 'flipper', kind: 'fish' }
{ name: 'flo', kind: 'rodent' }
{ name: 'fluffy', kind: 'cat' }

Found the following cats:
  fluffy
```

2.6 Performing a Lightweight Document Check

Use `DatabaseClient.documents.probe` or `DatabaseClient.probe` to test for the existence of a document in the database or retrieve a document identifier without fetching the document (when content versioning is enabled).

The following example probes for the existence of the document `/doc/example.json`:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.probe('/doc/example.json').result(
    function(response) {
      if (response.exists) {
        console.log(response.uri + ' exists');
      } else {
        console.log(response.uri + 'does not exist');
      }
    }
);
```

The return value is a document descriptor that includes a boolean-valued `exists` property. If content versioning is enabled on the REST instance, then the response also includes a `versionId` property. For example, with content versioning enabled, the above example produces the following output:

```
{
  contentType: "application/json",
  versionId: "14115045710437450",
  format: "json",
  uri: "/doc/example.json",
  exists: true
}
```

For more information about content versioning, see “Conditional Updates Using Optimistic Locking” on page 57.

2.7 Conditional Updates Using Optimistic Locking

An application using optimistic locking creates a document only when the document does not exist and updates or deletes a document only when the document has not changed since this application last changed it. However, optimistic locking does not actually involve placing a lock on document.

Optimistic locking is useful in environments where integrity is important, but contention is rare enough that it is useful to minimize server load by avoiding unnecessary multi-statement transactions.

This section covers the following topics:

2.7.1 Understanding Optimistic Locking

Consider an application that reads a document, makes modifications, and then updates the document in the database with the changes. The traditional approach to ensuring document integrity is to perform the read, modification, and update in a multi-statement transaction. This holds a lock on the document from the point when the document is read until the update is committed. However, this pessimistic locking blocks access to the document and incurs more overhead on the App Server.

With *optimistic locking*, the application does not hold a lock on a document between read and update. Instead, the application saves the document state on read, and then checks for changes at the time of update. The update fails if the document has changed between read and update. This is a *conditional update*.

Optimistic locking is useful in environments where integrity is important, but contention is rare enough that it is useful to minimize server load by avoiding unnecessary multi-statement transactions.

The Node.js Client API uses content versioning to implement optimistic locking. When content versioning is enabled on your REST API instance, MarkLogic Server associates an opaque version id with each document when it is created or updated. The version id changes each time you update the document. The version id is returned when you read a document, and you can pass it back in an update or delete operation to test for changes prior to commit.

Note: Enabling content versioning does *not* implement document versioning. MarkLogic Server does not keep multiple versions of a document or track what changes occur. The version id can only be used to detect that a change occurred.

Using optimistic locking in your application requires the following steps:

1. Enable Optimistic Locking in the REST API instance.

2. Obtain a Version Id for documents you wish to conditionally update.

3. Apply a Conditional Update by including the version in your update operations.

You enable optimistic locking by setting the `update-policy` REST API instance property; for details. You send and receive version ids via the `versionId` property in a document descriptor.

2.7.2 Enable Optimistic Locking

To enable optimistic locking, call `DatabaseClient.config.serverprops.write` and set the `update-policy` property to `version-required` or `version-optional`. For example, the following code sets update-policy to version-optional:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.config.serverprops.write({'update-policy': 'version-optional'})
  .result(function(response) {
      console.log(JSON.stringify(response));
  });
```

Set the property to `version-required` if you want every document update or delete operation to use optimistic locking. Set the property to `version-optional` to allow selective use of optimistic locking.

The table below describes how each setting for this property affects document operations.

Setting	Effect
`merge-metadata`	This is the default setting. If you insert, update, or delete a document that does not exist, the operation succeeds. If a version id is provided, it is ignored.
`version-optional`	If you insert, update or delete a document that does not exist, the operation succeeds. If a version id is provided, the operation fails if the document exists and the current version id does not match the supplied version id.
`version-required`	If you update or delete a document without supplying a version id and the document does not exist, then the operation succeeds; if the document exists, the operation fails. If a version id is provided, the operation fails if the document exists and the current version id does not match the version in the header.
`overwrite-metadata`	The behavior is the same as `merge-metadata`, except that metadata in the request overwrites any pre-existing metadata, rather than being merged into it. This setting disables optimistic locking.

2.7.3 Obtain a Version Id

When optimistic locking is enabled, a version id is included in the response when you read documents. You can only obtain a version id if optimistic locking is enabled by setting `update-policy`; for details, see “Enable Optimistic Locking” on page 58.

You can obtain a version id for use in conditional updates in the following ways:

- Call `DatabaseClient.documents.read`. The version id is available through the `versionId` property of the returned document descriptor(s).
- Call `DatabaseClient.documents.probe`. No documents are fetched, but the version id is available through the `versionId` property of the returned document descriptor(s).

You can mix and match these methods of obtaining a version id.

The following example returns the version for multiple documents:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read('/doc/example1.json', '/doc/example2.json').
```

```
  .stream().on('data', function(document) {
    console.log('Read ' + document.uri +
       ' with version ' + document.versionId);
  }).on('end', function() {
    console.log('finished');
  });
```

2.7.4 Apply a Conditional Update

To apply a conditional update, include a `versionId` property in the document descriptor passed to an update or delete operation. The version id is ignored if optimistic locking is not enabled; for details, see "Enable Optimistic Locking" on page 58.

When a document descriptor passed to an update or delete operation includes a version id, MarkLogic Server server checks for a version id match before committing the update or delete. If the input version id does not match a document's current version id, the operation fails. In a multi-document update, the entire batch of updates is rejected if the conditional update of any document in the batch fails.

The following example performs a conditional update of two documents by including the `versionId` property in each input document descriptor.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write({
  documents: [
    { uri: '/doc/example1.json',
      contentType: 'application/json',
      content: { data: 1 },
      versionId: 14115098125553360
    },
    { uri: '/doc/example2.json',
      contentType: 'application/json',
      content: { data: 2 },
      versionId: 14115098125553350
    }
  ]
}).result(
  function(success) {
    console.log('Loaded the following documents:');
    for (const i in success.documents)
      console.log(success.documents[i].uri);
  }
);
```

Similarly, the following example removes the document `/doc/example.json` only if the current version id of the document in the database matches the version id in the input document descriptor:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.remove({
  uris: ['/doc/example.json'],
  versionId: 14115105931044000}).
result(
  function(response) {
    console.log(JSON.stringify(response));
});
```

Note: You cannot use conditional delete when removing multiple documents in a single operation.

If a conditional update or delete fails due to a version id mismatch, MarkLogic Server responds with an error similar to the following. (The object contents have been reformatted for readability.)

```
{message: "remove document: response with invalid 412
           status (on /doc/example1.json)",
  statusCode:412,
  body:{
    error: {
      status-code: "412",
      status: "Precondition Failed",
      message-code: "RESTAPI-CONTENTWRONGVERSION",
      message: "RESTAPI-CONTENTWRONGVERSION: (err:FOER0000)
                Content version mismatch: uri /doc/example.json
                doesn't match if-match: 14115105931044000"
}}}
```

2.8 Working with Binary Documents

This section provides a brief overview of how to use the Node.js API to manipulate binary document data in MarkLogic Server. The following topics are covered:

- Type of Binary Documents
- Streaming Binary Content
- Retrieving Binary Content with Range Requests

2.8.1 Type of Binary Documents

This section provides a brief summary of binary document types. For details, see Working With Binary Documents in the *Application Developer's Guide*.

MarkLogic Server can store binary documents in three representations:

- Small binary documents are stored entirely in the database.

- Large binary documents are stored on disk with a small reference fragment in the database. The on-disk content is managed by MarkLogic Server.
- External binary documents are stored on disk with a small reference fragment in the database. However, the on-disk content is not managed by MarkLogic Server.

MarkLogic automatically determines whether a binary document is a small or large binary document when you insert or update the document, based on the document size and the database configuration.

Though external binary documents cannot be created using the Node.js Client API, you can retrieve them, just like any other document.

2.8.2 Streaming Binary Content

By using streaming techniques to access binary content, you can avoid loading potentially large documents into memory on MarkLogic Server and in your client application.

You can stream binary and other data into MarkLogic using by using a stream as the input source. For details, see “Streaming Into the Database” on page 22.

When you retrieve a large or external binary document from a database, MarkLogic Server automatically streams the content out of the database under the following conditions:

- Your request is for a single document, rather than being a bulk read.
- The size of the binary content returned is over the large binary size threshold. For details, see Working With Binary Documents in the *Application Developer's Guide*.
- The request is for content only. That is, no metadata is requested.
- The MIME type of the content is determinable from the Accept header or the document URI file extension.
- No content transformation is applied.

You can also use range requests to incrementally retrieve pieces of a binary document that meets the above constraints. For details, see “Retrieving Binary Content with Range Requests” on page 62.

You can avoid loading the entire document into memory in your client application by using a streaming result handler, such as the chunked stream pattern described in “Stream Result Handling Pattern” on page 21.

2.8.3 Retrieving Binary Content with Range Requests

When you just use `db.documents.read` to retrieve a binary document, the goal is to eventually retrieve the entire document, even if your application code processes it in chunks. By contrast, using range requests to retrieve parts of a binary document enables retryable, random access to parts of a binary document.

To use range requests, your retrieval operation must meet the conditions described in “Streaming Binary Content” on page 62. Specify the range of bytes to retrieve by including a “range” property in the call object passed to `DatabaseClient.documents.read`.

The following example requests the first 500K of the binary document with URI `/binary/song.m4a`:

```
db.documents.read({
    uris: '/binary/song.m4a',
    range: [0,511999]
  })
```

The document descriptor returned by such a call is similar to the following. The `contentLength` property indicates how many bytes were returned. This value will be smaller than the size of the requested range if you request a range that extends past the end of the source document.

```
result: [{
  content: {
    type': 'Buffer',
    data': [ theData ]
  },
  uri: '/binary/song.m4a',
  category: [ 'content' ],
  format: 'binary',
  contentLength: '10',
  contentType: 'audio/mp4'
}]
```

2.9 Working with Temporal Documents

Most document write operations enable you to work with temporal documents. Temporal-aware document inserts and updates are enabled through the following parameters (or document descriptor properties) on operations such as `documents.create`, `documents.write`, `documents.patch`, and `documents.remove`:

- `temporalCollection`: The URI of the temporal collection into which the new document should be inserted, or the name of the temporal collection that contains the document being updated.
- `temporalDocument`: The “logical” URI of the document in the temporal collection; the temporal collection document URI. This is equivalent to the first parameter of the `temporal:statement-set-document-version-uri` XQuery function or of the `temporal.statementSetDocumentVersionUri` Server-Side JavaScript function.
- `sourceDocument`: The temporal collection document URI of the document being operated on. Only applicable when updating existing documents. This parameter facilitates working with documents with user-maintained version URIs.
- `systemTime`: The system start time for an update or insert.

During an update operation, if you do not specify a `sourceDocument` or `temporalDocument`, then the `uri` descriptor property indicates the source document. If you specify `temporalDocument`, but do not specify `sourceDocument`, then `temporalDocument` identifies the source document.

The `uri` property always refers to the output document URI. When the MarkLogic manages the version URIs, the document URI and temporal document collection URI have the same value. When the user manages version URIs, they can be different.

Use `documents.protect` to protect a temporal document from operations such as update, delete, and wipe for a specified period of time. This method is equivalent to calling the `temporal:document-protect` XQuery function or the `temporal.documentProtect` Server-Side JavaScript function.

Use `documents.advanceLsqt` to advance LSQT on a temporal collection. This method is equivalen to calling the `temporal:advance-lsqt` XQuery function or the `temporal.advanceLsqt` Server-Side JavaScript function.

For more details, see the *Temporal Developer's Guide* and the Node.js Client API JSDoc.

2.10 Working with Metadata

The Node.js Client API enables you to insert, update, retrieve, and query metadata. Metadata is the properties, collections, permissions, and quality of a document. Metadata can be manipulated as either JSON or XML.

This section covers the following topics:

- Metadata Categories
- Metadata Format
- Working with Document Properties
- Disabling Metadata Merging

2.10.1 Metadata Categories

When working with documents and their metadata, metadata is usually expressed as properties of a document descriptor. For example, the following descriptor includes collections metadata for the document `/doc/example.json`, whether the descriptor is input output:

```
{ uri: '/doc/example.json',
  collections: ['collection1','collection2']
```

Some operations support a `categories` property for indicating what parts of a document and its metadata you want to read or write. For example, you can read just the collections and quality associated with `/doc/example.json` by calling `DatabaseClient.documents.read` similar to the following:

```
db.documents.read({
  uris: ['/doc/example.json'],
  categories: ['collections', 'quality']
})
```

The following categories are supported:

- `collections`
- `permissions`
- `properties`
- `quality`
- `metadataValues`
- `metadata`
- `content`

The `metadataValues` category represents simple key-value metadata property, sometimes called "metadata fields". This category can contain both system-managed metadata, such as certain properties of a temporal document, and user-defined metadata. The value of a property in `metadataValues` is always stored as a string in MarkLogic. For more details, see Metadata Fields in the *Administrator's Guide*.

The `metadata` category is shorthand for collections, permissions, properties, and quality. Some operations, such as `DatabaseClient.documents.read`, also support the `content` category as a convenience for retrieving or updating content and metadata together.

2.10.2 Metadata Format

Metadata takes the following form in a document descriptor. You do not need to specify all categories of metadata when including it in a write or query operation. This structure applies to both input and output metadata.

```
{
  "collections" : [ string ],
  "permissions" : [
    {
      "role-name" : string,
      "capabilities" : [ string ]
    }
  ],
  "properties" : {
    property-name : property-value
  },
  "quality" : integer,
  "metadataValues": { key: value, key: value, ... }
}
```

The following example is an output document descriptor from reading the metadata for the document with URI `/doc/example.json`. The document is in two collections and has two permissions, two document properties, and two `metadataValues` key-value pairs:

```
{
  "partType": "attachment",
  "uri": "/doc/example.json",
  "category": "metadata",
  "format": "json",
  "contentType": "application/json",
  "collections": [ "collection1", "collection2" ],
  "permissions": [
    {
      "role-name": "rest-writer",
      "capabilities": [ "update" ]
    },
    {
      "role-name": "rest-reader",
      "capabilities": [ "read" ]
    }
  ],
  "properties": {
    "prop1": "this is my prop",
    "prop2": "this is my other prop"
  },
  "metadataValues": {
    "key1": "value1",
    "key2": 2
  },
  "quality": 0
}
```

The following example shows metadata as XML. All elements are in the namespace `http://marklogic.com/rest-api`. You can have 0 or more `<collection/>`, `<permission/>` or property elements. There can be only one `<quality/>` element. The element name and contents of each property element depends on the property.

```
<metadata xmlns="http://marklogic.com/rest-api">
  <collections>
    <collection>collection-name</collection>
  </collections>
  <permissions>
    <permission>
      <role-name>name</role-name>
      <capability>capability</capability>
    </permission>
  </permissions>
  <properties>
    <property-element/>
  </properties>
  <quality>integer</quality>
  <metadata-values>
```

```
    <metadata-value key="key1">value1</metadata-value>
    <metadata-value key="key2">2</metadata-value>
  <metadata-values>
</metadata>
```

2.10.3 Working with Document Properties

Document properties are a kind of metadata. You can use document properties to add queryable user-defined data to a document without changing the document contents. For details, see Properties Documents and Directories in the *Application Developer's Guide*.

In most cases, your application should work with document properties in a consistent format. That is, if you insert or update properties as JSON, then you should retrieve and query them as JSON. If you insert or update properties as XML, you should retrieve them as XML. If you mix and match XML and JSON formats, you can encounter namespace inconsistencies.

Document properties are always stored in the database as XML. A document property inserted as JSON is converted internally into an XML element in the namespace `http://marklogic.com/xdmp/json/basic`, with an XML local name that matches the JSON property name. For example, a document property expressed in JSON as `{ myProperty : 'value' }` has the following XML representation:

```
<rapi:metadata uri="/doc/example.json" ...>
  <prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
    <myProperty type="string"
        xmlns="http://marklogic.com/xdmp/json/basic">
      value
    </myProperty>
  </prop:properties>
</rapi:metadata>
```

As long as you consistently use a JSON representation for document properties on input and output, this internal representation is transparent to you. However, if you query or read document properties using XML, you must be aware of the namespace and the internal representation. Similarly, you can use XML to insert document properties in no namespace or in your own namespace, but the namespace cannot be reflected in the JSON representation of the property. Therefore, it is best to be consistent in how you work with properties.

The one exception is in code that works with properties on the server, such as resource service extensions and transformations. Such code always accesses document properties as XML.

If you configure an index based on a user-defined document property inserted using JSON, you should use the `http://marklogic.com/xdmp/json/basic` namespace in your configuration.

Protected system properties such as `last-updated` cannot be modified by your application. The JSON representation of such properties wraps them in an object with the key `$ml.prop`. For example:

```
{ "properties": {
    "$ml.prop": {
      "last-updated": "2013-11-06T10:01:11-08:00"
    }
} }
```

2.10.4 Disabling Metadata Merging

If you use the REST Client API to ingest a large number of documents at one time and you find the performance unacceptable, you might see a small performance improvement by disabling metadata merging. This topic explains the tradeoff and how to disable metadata merging.

- When to Consider Disabling Metadata Merging
- How to Disable Metadata Merging

2.10.4.1 When to Consider Disabling Metadata Merging

The performance gain from disabling metadata merging is modest, so you are unlikely to see significant performance improvement from it unless you ingest a large number of documents. You might see a performance gain under one of the following circumstances:

- Ingesting a large number of documents, one at a time.
- Updating a large number of documents that share the same metadata or that use the default metadata.

You cannot disable metadata merging in conjunction with update policies `version-optional` or `version-required`.

Metadata merging is disabled by default for multi-document write requests, as long as the request includes content for a given document. For details, see Understanding When Metadata is Preserved or Replaced in the *REST Application Developer's Guide*.

To learn more about the impact of disabling metadata merging, see Understanding Metadata Merging in the *REST Application Developer's Guide*.

2.10.4.2 How to Disable Metadata Merging

Metadata merging is controlled by the `update-policy` instance configuration property. The default value is `merge-metadata`.

To disable metadata merging, set `update-policy` to `overwrite-metadata` using the procedure described in "Configuring Instance Properties" on page 264. For example:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.config.serverprops.write({'update-policy': 'overwrite-metadata'})
```

```
.result(function(response) {
    console.log(JSON.stringify(response));
});
```

3.0 Patching Document Content or Metadata

This chapter covers the following topics related to updating selected portions of the content or metadata of a document using the `db.documents.patch` function.

3.1 Introduction to Content and Metadata Patching

A *partial update* is an update you apply to a portion of a document or metadata, rather than replacing an entire document or all of its metadata. For example, inserting a JSON property or XML element, or changing the value of a JSON property. You can only apply partial content updates to JSON and XML documents. You can apply partial metadata updates to any document type.

A *patch* is a partial update descriptor, expressed in JSON or XML. A patch tells MarkLogic Server what update to apply and where to apply it. Four operations are available in a patch: insert, replace, replace-insert, and delete. (A replace-insert operation functions as a replace if there is at least one match for the target content; if there are no matches, then the operation functions as an insert.)

Use the partial update feature for the following operations:

- Add or delete a JSON property, property value, or array item in an existing document.
- Add, replace, or delete the value of an array item or JSON property.
- Add, replace, or delete a subset of the metadata of an existing document. For example, modify a permission or insert a document property.
- Dynamically generate replacement content or metadata on MarkLogic Server using builtin or custom, user-supplied functions. For details, see “Constructing Replacement Data on MarkLogic Server” on page 104.

You can apply multiple updates in a single patch, and you can update both content and metadata in the same patch.

Patch operations can target JSON properties, XML elements and attributes, and data values such as JSON array items and the data in an XML element or attribute. You identify the target of an operation using XPath or JSONPath expressions. When inserting new content or metadata, the insertion point is further defined by specifying the position; for details, see How Position Affects the Insertion Point in the *REST Application Developer's Guide*.

When applying a patch to document content, the patch format must match the document format: An XML patch for an XML document, a JSON patch for a JSON document. You cannot patch the content of other document types. You can patch metadata for all document types. A metadata-only patch can be in either XML or JSON. A patch that modifies both content and metadata must match the document content type.

The Node.js Client API provides the following interfaces for building and applying patches:

- `marklogic.patchBuilder`
- `DatabaseClient.documents.patch`

Apply a patch by calling `DatabaseClient.documents.patch` with a URI and one or more patch operations. Build patch operations using `marklogic.patchBuilder`.

The following example patches the JSON document `/patch/example.json` by inserting a property named `child3` in the "last child" position under the property named `theTop`. For a complete example, see "Example: Adding a JSON Property" on page 72.

```
const pb = marklogic.patchBuilder;
db.documents.patch('/patch/example.json',
  pb.insert('/object-node("theTop")', 'last-child',
    {child3: 'INSERTED'})
);
```

For additional examples of building patch operations, see "Patch Examples" on page 86.

If a patch contains multiple operations, they are applied independently to the target document. That is, within the same patch, one operation does not affect the context path or select path results or the content changes of another. Each operation in a patch is applied independently to every matched node. If any operation in a patch fails with an error, the entire patch fails.

Content transformations are not directly supported in a partial update. However, you can implement a custom replacement content generation function to achieve the same effect. For details, see "Constructing Replacement Data on MarkLogic Server" on page 104.

Before patching JSON documents, you should familiarize yourself with the restrictions outlined in Limitations of JSON Path Expressions in the *REST Application Developer's Guide*.

3.2 Example: Adding a JSON Property

The following example inserts a new property into a JSON document using the `DatabaseClient.documents.patch` function and a patch builder. The example program does the following:

1. Insert the example document into the database with the URI `/patch/example.json`.

2. Patch the example document by inserting a new property under `theTop` named `child3` in the "last child" position. The parent node is identified using an XPath expression.

3. Read the patched document from the database and display it on the console.

The `db.documents.patch` function accepts a document URI and one or more operations. In this case, we pass only one operation: A property insertion constructed by calling the patch builder `insert` method.

```
pb.insert('/object-node("theTop")',   // path to parent node
          'last-child',               // insertion position
          {child3: 'INSERTED'})       // content to insert
```

The promise pattern is used to chain together the write, patch, and read operations. For details, see "Promise Result Handling Pattern" on page 20.

The example script is shown below. The initial write operation is included in the example only to encapsulate the example document and ensure consistent behavior across runs. Usually, the document targeted by a patch would have been loaded into the database separately.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/example.json',
  contentType: 'application/json',
  content: {
    theTop: {
      child1: { grandchild: 'gc-value' },
      child2: 'c2-value'
  } }
}).result().then(function(response) {
  // (2) Patch the document
  const pb = marklogic.patchBuilder;
  return db.documents.patch(response.documents[0].uri,
    pb.insert('/object-node("theTop")', 'last-child',
              {child3: 'INSERTED'})
  ).result();
}).then(function(response) {
  // (3) Read the patched document
  return db.documents.read(response.uri).result();
```

```
}).then(function(response) {
  console.log(response[0].content);
});
```

The example program results in the following document transformation:

Before Patching	After Patching

{"theTop" : {
"child1" : {
"grandchild" : "gc-value"
},
"child2" : "c2-value"
}}
	{"theTop" : {
	"child1" : {
	"grandchild" : "gc-value"
	},
	"child2" : "c2-value",
	"child3" : "INSERTED"
	}}

On success, `DatabaseClient.documents.patch` returns the URI of the patched document. For example, the result of the db.documents.patch call above is:

```
{ uri: '/patch/example.json' }
```

For more examples, see “Patch Examples” on page 86.

3.3 Patch Reference

You can include one or more of the following operations in a call to `DatabaseClient.documents.patch`: insert, replace, replace-insert, remove. These operations can occur multiple times in a given `patch` call. Use the `marklogic.patchBuilder` interface to construct each operation. For example:

```
db.documents.patch(uri,
  pb.insert(path, position, newContent)
)
```

The patch builder includes special purposes interfaces for constructing patch operations that target metadata: `patchBuilder.collections`, `patchBuilder.permissions`, `patchBuilder.properties`, `patchBuilder.quality`, and `patchBuilder.metadataValues`. These interfaces enable you to patch collections, permissions, quality, document properties, and values metadata without knowing the internal representation of metadata. For an example of how to use these interfaces, see “Example: Patching Metadata” on page 99.

In addition, you can include patch configuration directives, such as `patchBuilder.library` and `patchbuilder.pathLanguage`. You can only include one `library` directive, and it is only needed if you use custom functions to generate replacement content; for details, see “Constructing Replacement Data on MarkLogic Server” on page 104.

The table below summarizes the patch builder methods for constructing operations on content:

Builder Method	Descriptions
insert	Insert a new property or array item.
replace	Replace an existing property or array item.
replaceInsert	Replace a JSON property or array item; if there are no existing matching properties, perform an insertion instead.
remove	Remove a JSON property or array item.
library	Identify a server-side XQuery library module that contains custom replacement content generation functions that can be used in the `apply` operation that is part of a `replace` or `replaceInsert` operation.
apply	Specify a server-side replacement content generation function. The patch operation must include a `library` operation, and the named function must be in that module.
pathLanguage	Configure a patch to parse `select` and `context` expressions as either XPath (default) or JSONPath expressions.

The following table summarizes the patch builder interfaces for constructing operations on metadata.

Builder Method	Descriptions
collections	Construct patch operations for modifying collections metadata.
permissions	Construct patch operations for modifying permissions metadata.
properties	Construct patch operations for modifying document properties metadata.
quality	Construct patch operations for modifying quality metadata.
metadataValues	Construct patch operations for modifying values metadata.

The following table summarizes the patch builder methods for constructing cofiguration directives:

Builder Method	Descriptions
library	Identify a server-side XQuery library module that contains custom replacement content generation functions that can be used in the `apply` operation that is part of a `replace` or `replaceInsert` operation.
apply	Specify a server-side replacement content generation function. The patch operation must include a `library` operation, and the named function must be in that module.
pathLanguage	Configure a patch to parse `select` and `context` expressions as either XPath (default) or JSONPath expressions.

3.3.1 insert

Use `patchbuild.insert` to create an operation that inserts a new JSON property or array item. Build an insert operation with a call of the following form:

```
marklogic.patchBuilder.insert(
  context,
  position,
  content,
  cardinality)
```

The following table summarizes the parameters of the `patchBuilder.insert` function.

Parameter	Req'd	Description
`context`	Y	An XPath or JSONPath expression that selects an existing JSON property or array element on which to operate. The expression can select multiple items. If no matches are found for the `context` expression, the operation is silently ignored. The path expression is restricted to the subset of XPath for which `cts:valid-document-patch-path` (XQuery) or `cts.validDocumentPatchPath` (JavaScript) returns true. For details, see Path Expressions Usable in Patch Operations in the *REST Application Developer's Guide*.
`position`	Y	Where to insert the content, relative to the JSON property or value selected by `context`. The *pos-selector* must be one of `"before"`, `"after"`, or `"last-child"`. For details, see "How Position Affects the Insertion Point" on page 84.
`content`	Y	The new content to be inserted, expressed as a JSON object, array, or atomic value.
`cardinality`	N	The required occurrence of matches to `position`. If the number of matches does not meet the expectation, the operation fails and an error is returned. Allowed values: • Zero or one matches required: `"?"` (question mark) • Exactly one match required: `"."` (period) • Zero or more matches required: "`*`" (asterisk) • One or more matches required: "`+`" (plus) Default: "*" (The occurrence requirement is always met).

3.3.2 replace

Use `patchBuilder.replace` to create an operation that replaces an existing JSON property value or array item. If no matching JSON property or array item exists, the operation is silently ignored. Build a `replace` operation with a call of the following form:

```
marklogic.patchBuilder.replace(
  select,
  content,
```

```
    cardinality,
    apply)
```

You can use `apply` to specify a content generation builtin or custom function for generating dynamic content. For details, see “Constructing Replacement Data on MarkLogic Server” on page 104.

The following table summarizes the parameters of the `patchBuilder.replace` function.

Parameter	Req’d	Description
`select`	Y	An XPath or JSONPath expression that selects the JSON property or array element to replace. If no matches are found for the `select` expression, the operation is silently ignored. The path expression is restricted to the subset of XPath for which `cts:valid-document-patch-path` (XQuery) or `cts.validDocumentPatchPath` (JavaScript) returns true. For details, see Path Expressions Usable in Patch Operations in the *REST Application Developer’s Guide*. The selected item(s) cannot be the target of any other operation in the patch. The ancestor of the selected item may not be modified by a `delete`, `replace`, or `replace-insert` operation in the same patch.
`content`	N	The replacement value. If you omit this parameter, you must specify a content generation function in the `apply` parameter. If `select` targets a property and `content` is an object, the operation replaces the entire target property. Otherwise, the operation replaces just the value.
`cardinality`	N	The required occurrence of matches to `select`. If the number of matches does not meet the expectation, the operation fails and an error is returned. Allowed values: • Zero or one matches required: `"?"` (question mark) • Exactly one match required: `"."` (period) • Zero or more matches required: `"*"` (asterisk) • One or more matches required: `"+"` (plus) Default: `"*"` (The occurrence requirement is always met).

Parameter	Req'd	Description
`apply`	N	The local name of a replacement content generation function. If you do not specify a function, the operation must include a `content` parameter. If you name a custom function, the `patch` must include a `replace-library` operation that describes the XQuery library module containing the function implementation. To construct such an operation, use the `patchBuilder.library` function. For details, see "Constructing Replacement Data on MarkLogic Server" on page 104.

3.3.3 replaceInsert

Use `patchBuilder.replaceInsert` to create an operation that replaces a property value or array item if it exists, or insert a property or array item if does not. Build a `replace-insert` operation with a call of the following form:

```
replaceInsert(
  select,
  context,
  position,
  content,
  cardinality,
  apply)
```

You can omit `content` if you use `apply` to specify a content generation builtin or custom function for generating dynamic content. For details, see "Constructing Replacement Data on MarkLogic Server" on page 104.

The following table summarizes the parts of a `replace-insert` operation.

Parameter	Req'd	Description
`select`	Y	An XPath or JSONPath expression that selects the JSON property or array item to replace. If no matches are found for the `select` expression, `context` and `position` are used to attempt an insert. If no match is found for `context`, the operation does nothing. The path expression is restricted to the subset of XPath for which `cts:valid-document-patch-path` (XQuery) or `cts.validDocumentPatchPath` (JavaScript) returns true.. For details, see Path Expressions Usable in Patch Operations in the *REST Application Developer's Guide*. The selected item(s) cannot be the target of any other operation in the patch. The ancestor of the selected item may not be modified by a `delete`, `replace`, or `replace-insert` operation in the same patch.
`content`	N	The content with which to replace the selected value. If there is no `content`, you must specify a content generation function using `apply`. If `select` targets a property and `content` is an object, the operation replaces the entire target property. Otherwise, the operation replaces just the value.
`context`	Y	An XPath or JSONPath expression that selects an existing property or array element on which to operate. The expression can select multiple items. If no matches are found for the either the `select` or `context` expression, the operation is silently ignored. The path expression is restricted to the subset of XPath for which `cts:valid-document-patch-path` (XQuery) or `cts.validDocumentPatchPath` (JavaScript) returns true. For details, see Path Expressions Usable in Patch Operations in the *REST Application Developer's Guide*. The ancestor of the selected node may not be modified by a `delete`, `replace`, or `replace-insert` operation in the same patch.

Parameter	Req'd	Description
`position`	N	If `select` does not match anything, where to insert the content, relative to the key-value pair or value selected by `context`. The *pos-selector* must be one of `"before"`, `"after"`, or `"last-child"`. For details, see "How Position Affects the Insertion Point" on page 84. Default: `last-child`.
`cardinality`	N	The required occurrence of matches to `position`. If the number of matches does not meet the expectation, the operation fails and an error is returned. Allowed values: • Zero or one matches required: "`?`" (question mark) • Exactly one match required: "`.`" (period) • Zero or more matches required: "`*`" (asterisk) • One or more matches required: "`+`" (plus) Default: * (The occurrence requirement is always met).
`apply`	N	The local name of a replacement content generation function. If you do not specify a function, the operation must include a `content` parameter. If you name a custom function, the `patch` must include a `replace-library` operation that describes the XQuery library module containing the function implementation. Create such an operation using the `patchBuilder.library` function. For details, see "Constructing Replacement Data on MarkLogic Server" on page 104.

3.3.4 remove

Use `patchBuilder.remove` to create an operation that removes a JSON property or array element. A call to the `patchBuilder.remove` function has the following form:

```
marklogic.patchBuilder.remove(
  select,
  cardinality)
```

The following table summarizes the parameters of `patchBuilder.remove`.

Component	Req'd	Description
`select`	Y	An XPath or JSONPath expression that selects the JSON property or array element to remove. If no matches are found for the `select` expression, the operation is silently ignored. The path expression is restricted to the subset of XPath for which `cts:valid-document-patch-path` (XQuery) or `cts.validDocumentPatchPath` (JavaScript) returns true. For details, see Path Expressions Usable in Patch Operations in the *REST Application Developer's Guide*.. The selected item(s) cannot be the target of any other operation in the patch. The ancestor of the selected item may not be modified by a `delete`, `replace`, or `replace-insert` operation in the same patch.
`cardinality`	N	The required occurrence of matches to `select`. If the number of matches does not meet the expectation, the operation fails and an error is returned. Allowed values: • Zero or one matches required: "`?`" (question mark) • Exactly one match required: "`.`" (period) • Zero or more matches required: "`*`" (asterisk) • One or more matches required: "`+`" (plus) Default: `"*"` (The occurrence requirement is always met).

3.3.5 apply

Use `patchBuilder.apply` to create a configuration directive that specifies a server-side function with which to generate replacement conent for a `replace` or `replaceInsert` operation. The named function must be in a server-side module identified by a `library` directive included in the same call to `db.documents.patch`.

A call to the `patchBuilder.apply` function has the following form:

```
marklogic.patchBuilder.apply(functionName)
```

For details, see "Constructing Replacement Data on MarkLogic Server" on page 104.

3.3.6 library

Use `patchBuilder.library` to create a configuration directive that identifies a server-side library module containing one or more functions for generating replacement content. The module must conform to the conventions described in "Writing a Custom Replacement Constructor" on page 108. To make use of the functions in your library, include the generated library directive in your call to `db.documents.patch`, and include the output from `patchBuilder.apply` when calling the `replace` or `replaceInsert` builder functions.

A call to the `patchBuilder.library` function has the following form:

```
marklogic.patchBuilder.library(moduleName)
```

For details, see "Constructing Replacement Data on MarkLogic Server" on page 104.

3.3.7 pathLanguage

By default, all path expressions in patches are expressed using XPath. JSONPath is also supported for compatibility with previous releases, but it is deprecated. You do not need to use a `pathLanguage` directive unless you use JSONPath.

Use `patchBuilder.pathLanguage` to create a configuration directive that specifies the path language in which you express context and select expressions in your patch operations. A call to the `patchBuilder.pathLanguage` function must have one of the following forms:

```
marklogic.patchBuilder.pathLanguage('xpath')

marklogic.patchBuilder.pathLanguage('jsonpath')
```

Include the resulting directive in your call to `db.documents.patch`.

You must use the same path language for all operations in a `DatabaseClient.documents.patch` call.

3.3.8 collections

Use the `patchBuilderCollections` methods to construct a patch operation for collections metadata. Use `patchBuiler.collections` to access these methods. You can add or remove a collection, using the following methods:

```
marklogic.patchBuilder.collections.add(collName)

marklogic.patchBuilder.collections.remove(collName)
```

For an example, see "Example: Patching Metadata" on page 99.

3.3.9 permissions

Use `patchBuilderPermissions` methods to construct a patch operation for permissions metadata. Use `patchBuilder.permissions` to access these methods. You can add or remove a permission or replace the capabilities of a role using this interface.

```
marklogic.patchBuilder.permissions.add(role, capabilities)

marklogic.patchBuilder.permissions.remove(role)

marklogic.patchBuilder.permissions.replace(role, capabilities)
```

Where *role* is a string that contains the role name, and *capabilities* is either a single string or an array of strings with the following possible values: “read”, “insert”, “update”, “execute”.

Note that `replace` replaces all the capabilities associated with a role. You cannot use it to selectively replace a single capability. Rather, you must operate on a role and the associated capabilities as a single unit.

For an example, see “Example: Patching Metadata” on page 99.

3.3.10 properties

Use `patchBuilderProperties` methods to construct a patch operation for document properties metadata. Use `patchBuilder.properties` to access these methods. You can add or remove a property, or replace the value of a property.

```
marklogic.patchBuilder.properties.add(name, value)

marklogic.patchBuilder.properties.remove(name)

marklogic.patchBuilder.properties.replace(name, value)
```

For an example, see “Example: Patching Metadata” on page 99.

3.3.11 quality

Use `patchBuilderQuality` methods to construct a patch operation that sets the value of quality metadata. Use `patchBuilder.quality` to access these methods.

```
marklogic.patchBuilder.quality.set(value)
```

For an example, see “Example: Patching Metadata” on page 99.

3.3.12 metadataValues

Use `patchBuilderMetadataValues` methods to construct a patch operation on values metadata. Use `patchBuilder.metdataValues` to access these methods. You can add or remove a key-value pair, or replace the value of an existing key.

```
marklogic.patchBuilder.metadataValues.add(name, value)

marklogic.patchBuilder.metadataValues.remove(name)

marklogic.patchBuilder.metadataValues.replace(name, value)
```

For an example, see “Example: Patching Metadata” on page 99.

3.4 Defining the Context for a Patch Operation

When you insert, replace, or delete content or metadata, the patch definition must include enough context to tell MarkLogic Server what JSON or XML components to operate on. For example, which json property or XML element to delete, where to insert a new property or element, or which JSON property, XML element, or value to replace.

When you create a patch using a builder, you specify the context through the *contextPath* and *selectPath* parameters of builder methods such as `DocumentPatchBuilder.insertFragment()` or `DocumentPatchBuilder.replaceValue()`. When you create a patch from raw XML or JSON, you specify the operation context through the `context` and `select` XML attributes or JSON property.

Use XPath expressions to define the operation context. For security and performance reasons, your XPath expressions are restricted to the subset of XPath for which `cts:valid-document-patch-path` (XQuery) or `cts.validDocumentPatchPath` (JavaScript) returns true. For details, see Patch Feature of the Client APIs in the *XQuery and XSLT Reference Guide*.

Insertion operations have an additional position parameter/property that defines where to insert new content relative to the context, such as before or after the selected node. For more details, see“How Position Affects the Insertion Point” on page 84.

3.5 How Position Affects the Insertion Point

The `insert` and `replace-insert` patch operations include a position parameter that defines the point of insertion when coupled with a context expression. This section describes the details of how position affects the insertion point.

This topic focuses on patching JSON documents. For details on XML, see Specifying Position in XML in the *REST Application Developer’s Guide*.

The position parameter to a PatchBuilder operation (or the `position` property of a raw patch) specifies where to insert new content relative to the item selected by the context XPath expression. Position can be one of the following values:

- `before`: Insert before the item selected by `context`.
- `after`: Insert after the item selected by `context`.
- `last-child`: Insert into the value of the target property or array, in the position of last child. The value selected by `context` must be an object or array node.

The same usage applies whether inserting on behalf of an `insert` operation or a `replace-insert` operation.

The following table shows example combinations of `context` and `position` and the resulting insertion point for new content. The insertion point is indicated by `***`.

Context	Position	Example Insertion Point

`/theTop/node("child1")`
 	
a property
	`after`	`{"theTop" : {`
		`  "child1" : [ "val1", "val2" ],`
		`  ***`
		`  "child2" : [`
		`  {"one": "val1"},`
		`  {"two": "val2"}`
		`  ]`
		`} }`

`/theTop/child1[1]`
 	
an array item
	`before`	`{"theTop" : {`
		`  "child1" : [ *** "val1", "val2" ],`
		`  "child2" : [`
		`  {"one": "val1"},`
		`  {"two": "val2"}`
		`  ]`
		`} }`

`/theTop/array-node("child1")`
 	
an array
	`last-child`	`{"theTop" : {`
		`  "child1" : [ "val1", "val2" ***],`
		`  "child2" : [`
		`  {"one": "val1"},`
		`  {"two": "val2"}`
		`  ]`
		`} }`

Context	Position	Example Insertion Point

`/node("theTop")`
 	
an object
	`last-child`	`{"theTop" : {`
		`  "child1" : [ "val1", "val2"],`
		`  "child2" : [`
		`  {"one": "val1"},`
		`  {"two": "val2"}`
		`  ]`
		`  ***`
		`} }`

`/theTop/child1`
 	
a value
	`last-child`	Error because the selected value is not an object or array. For example, an error results if the target document has the following contents:
		
		`{"theTop" : {`
		`  "child1" : "val1",`
		`  "child2" : [...]`
		`} }`

For more information about XPath over JSON, see Traversing JSON Documents Using XPath in the *Application Developer's Guide*.

3.6 Patch Examples

This section contains examples of constructing and applying patches to metadata and JSON documents. The following topics are covered:

- Preparing to Run the Examples
- Example: Insert
- Example: Replace
- Example: ReplaceInsert
- Example: Remove
- Example: Patching Metadata

To patch XML documents, you must construct a raw patch. For details, see "Patching XML Documents" on page 103 and "Creating a Patch Without a Builder" on page 102.

3.6.1 Preparing to Run the Examples

The examples are self-contained, except for the database connection information. Each example assumes your connection information is encapsulated in a module named `my-connection.js` that is co-located with the example script.

This module is expected contain contents similar to the following:

```
module.exports = {
  connInfo: {
    host: 'localhost',
    port: 8000,
    user: your-ml-username,
    password: your-ml-user-password
  }
};
```

For details, see “Using the Examples in This Guide” on page 31.

3.6.2 Example: Insert

This example demonstrates the insert operation by making the following changes to the base JSON document:

- Insert a new property of the unnamed root object (`INSERTED1`).
- Insert a new property in a sub-object, relative to a sibling property (`INSERTED2`).
- Insert a new array item relative to an item with a specific value (`INSERTED3`).
- Insert a new array item relative to a specific position in the array (`INSERTED4`).
- Insert a new item at the end of an array (`INSERTED5`).
- Insert a new property in the “last child” position (`INSERTED6`).

The example first inserts the base document into the database, then builds and applies a patch. Finally, the modified document is retrieved from the database and displayed on the console. The example uses a Promise pattern to sequence these operations; for details, see “Promise Result Handling Pattern” on page 20.

To run the example, copy the following code into a file, then supply it to the `node` command. For example: `node insert.js`.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/insert.json',
  contentType: 'application/json',
  content: {
    theTop: {
      child1: { grandchild: 'gcv' },
      child2: [ 'c2v1', 'c2v2' ],
      child3: [ {c3k1: 'c3v1'}, {c3k2: 'c3v2'} ]
  } }
}).result().then(function(response) {
```

```
  // (2) Patch the document
  const pb = marklogic.patchBuilder;
  return db.documents.patch(response.documents[0].uri,
    // insert a sibling of theTop
    pb.insert('/theTop', 'before',
              {INSERTED1: [ 'i1v1', 'i1v2']}),

    // insert a property in child1's value
    pb.insert('/theTop/child1/grandchild', 'before',
              {INSERTED2: 'i2v'}),

    // insert an array item in child2 by value
    pb.insert('/theTop/child2[. = "c2v1"]', 'after',
              'INSERTED3'),

    // insert an array item in child2 by position
    pb.insert('/theTop/child2[2]', 'after',
              'INSERTED4'),

    // insert an object in child3
    pb.insert('/theTop/array-node("child3")', 'last-child',
              {INSERTED5 : 'i5v'}),

    // insert a new child of theTop
    pb.insert('/theTop', 'last-child',
              {INSERTED6: 'i6v'})
  ).result();
}).then(function(response) {
  // (3) Emit the resulting document
  return db.documents.read(response.uri).result();
}).then(function(documents) {
  console.log(JSON.stringify(documents[0].content));
});
```

The following table shows how applying the patch changes the target document.

Before Update	After Update

{"theTop": {
  "child1": {"grandchild": "gcv"},
  "child2": ["c2v1", "c2v2"],
  "child3": [
  {"c3k1": "c3v1"},
  {"c3k2": "c3v2"}
  ]
} }
	{ "INSERTED1": ["i1v1", "i1v2"],
	  "theTop": {
	  "child1": {
	  "INSERTED2": "i2v",
	  "grandchild": "gcv"
	  },
	  "child2": [
	  "c2v1",
	  "INSERTED3",
	  "c2v2",
	  "INSERTED4"
	  ],
	  "child3": [
	  { "c3k1": "c3v1" },
	  { "c3k2": "c3v2" }
	  { "INSERTED5": "i5v" }
	  ],
	  "INSERTED6": "i6v"
	} }

Note that when you insert something using `last-child` for the position, the insertion context XPath expression must refer to the containing object or array. When you construct an XPath expression such as `/a/b`, the expression references a value (or sequence of values), rather than the container. To reference the container, use the `node()` and `array-node()` operators. This is why the operation that creates INSERTED5 under `child3` uses `array-node("child3")`, as shown below:

```
pb.insert('/theTop/array-node("child3")', 'last-child',
              {INSERTED5 : 'i5v'})
```

If you change the insertion context expression to `/theTop/child3`, you're referencing the sequence of values in the array, not the array object to which you want to add a child.

To insert a property if and only if it does not already exist, use a predicate in the context path expression to test for existence. For example, the following patch will insert a property named TARGET only if the object does not already contain such a property:

```
pb.insert('/theTop[fn:empty(TARGET)]', 'last-child',
          {TARGET: 'INSERTED'})
```

This patch must use the `last-child` position because the context selects the node that will contain the new property.

For more information about XPath over JSON, see Traversing JSON Documents Using XPath in the *Application Developer's Guide*.

3.6.3 Example: Replace

This example demonstrates the replace operation by making the following changes to a JSON document:

- Replace the value of a property with a simple value (`child1`), object value (`child2`), or array value (`child3`).
- Replace the value of an array item by position or value (`child4`).
- Replace the value of all items in an array with the same value (`child5`).
- Replace the value of an object in an array by property name (`child6`).
- Replace an array item that is an object by property name (`child6`).
- Replace the value of an array item that is a nested array (child7).
- Replace the value of an item in a nested array (child8).

The example first inserts the base document into the database, then builds and applies a patch. Finally, the modified document is retrieved from the database and displayed on the console. The example uses a Promise pattern to sequence these operations; for details, see “Promise Result Handling Pattern” on page 20.

To run the example, copy the following code into a file, then supply it to the `node` command. For example: `node replace.js`.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/replace.json',
  contentType: 'application/json',
  content: {
    theTop: {
      child1: 'c1v',
      child2: { gc: 'gcv' },
      child3: [ 'c3v1', 'c3v2' ],
      child4: [ 'c4v1', 'c4v2' ],
      child5: [ 'c5v1', 'c5v2' ],
      child6: [ {gc1: 'gc1v'}, {gc2:'gc2v'}],
      child7: [ 'av1', ['nav1', 'nav2'], 'av2'],
      child8: [ 'av1', ['nav1', 'nav2'], 'av2']
  } }
}).result().then(function(response) {
  // (2) Patch the document
  const pb = marklogic.patchBuilder;
  return db.documents.patch(response.documents[0].uri,
    // replace the simple value of a property
    pb.replace('/theTop/child1', 'REPLACED1'),
```

```
    // replace a property value that is an object
    pb.replace('/theTop/child2', {REPLACE2: 'gc2'}),

    // replace the value of a property that is an array
    pb.replace('/theTop/array-node("child3")',
              ['REPLACED3a', 'REPLACED3b']),

    // replace an array item by position
    pb.replace('/theTop/child4[1]', 'REPLACED4a'),

    // replace an array item by value
    pb.replace('/theTop/child4[.="c4v2"]', 'REPLACED4b'),

    // replace the value of all items in an array with the same value
    pb.replace('/theTop/child5', 'REPLACED5'),

    // replace the value of a property in an array item
    pb.replace('/theTop/child6/gc1', 'REPLACED6a'),

    // replace an object-valued array item by property name
    pb.replace('/theTop/child6[gc2]', {REPLACED6b: '6bv'}),

    // replace the value of an array item that is a nested array
    pb.replace('/theTop/array-node("child7")/node()[2]',
               [ 'REPLACED7a', 'REPLACED7b' ]),

    // replace the value of item in a nested array
    pb.replace('/theTop/child8[2]', 'REPLACED8')
).result();
}).then(function(response) {
  // (3) Emit the resulting document
  return db.documents.read(response.uri).result();
}).then(function(documents) {
  console.log(JSON.stringify(documents[0].content, null, 2));
});
```

The following table shows how applying the patch changes the target document.

Before Update	After Update
{"theTop": { "child1": "c1v", "child2": { "gc": "gcv" }, "child3": ["c3v1", "c3v2"], "child4": ["c4v1", "c4v2"], "child5": ["c5v1", "c5v2"], "child6": [{"gc1": "gc1v"}, {"gc2":"gc2v"}], "child7": ["av1", ["nav1", "nav2"], "av2"], "child8": ["av1", ["nav1", "nav2"], "av2"] } }	{"theTop":{ "child1": "REPLACED1", "child2": { "REPLACE2": "gc2" }, "child3": ["REPLACED3a", "REPLACED3b"], "child4": ["REPLACED4a", "REPLACED4b"], "child5": ["REPLACED5", "REPLACED5"], "child6": [{ "gc1": "REPLACED6a" }, { "REPLACED6b": "6bv" }], "child7": ["av1", ["REPLACED7a", "REPLACED7b"], "av2"], "child8": ["av1", ["REPLACED8", "nav2"], "av2"] } }

You should understand the difference between selecting a container and selecting its contents. For example, consider the two replacement operations applied to `child5`. The XPath expression `/theTop/child6/gc1` addresses the value of property with name `gc1`. Therefore, the replacement operation results in the following change:

```
pb.replace('/theTop/child6/gc1', 'REPLACED6a')

{"gc1" : "gc1v"} ==> {"gc1" : "REPLACED6a"}
```

By contrast the XPath expression `/theTop/child6[gc2]` selects object nodes that contain a property named `gc2`. Therefore, the replacement operation replaces the entire object with a new value, resulting in the following change:

```
pb.replace('/theTop/child6[gc2]', {REPLACED6b: '6bv'})

{"gc2" : "gc2v"} ==> {"REPLACED6b": "6bv"}
```

Similarly, consider the replacement operation on `child3`. The XPath expression `/theTop/array-node("child3")` selects the array node named `child3`, so the operation replaces the entire array with a new value. For example:

```
pb.replace('/theTop/array-node("child3")',
              ['REPLACED3a', 'REPLACED3b'])
```

```
"child3": ["c3v1", "c3v2"] ==> "child3": ["REPLACED3a", "REPLACED3b"]
```

By contrast, an XPath expression such as `/theTop/child5` selects the values in the array (`'c5v1'`, `'c5v2'`), so a replacement operation with this select expression replaces each of the array values with the same content. For example:

```
pb.replace('/theTop/child5', 'REPLACED5')

"child5": ["c5v1", "c5v2"] ==> "child5": ["REPLACED5", "REPLACED5"]
```

For more information on XPath over JSON, see Traversing JSON Documents Using XPath in the *Application Developer's Guide*.

3.6.4 Example: ReplaceInsert

This example demonstrates the `replace-insert` patch operation. The example performs the following update operations:

- Replace or insert an array item by position (`child1`).
- Replace or insert an array item by value (`child2`).
- Replace or insert an object-valued array item by contained property name (`child3`).

The example first inserts the base document into the database, then builds and applies a patch. Finally, the modified document is retrieved from the database and displayed on the console. The example uses a Promise pattern to sequence these operations; for details, see "Promise Result Handling Pattern" on page 20.

To run the example, copy the following code into a file, then supply it to the `node` command. For example: `node replace-insert.js`.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/replace-insert.json',
  contentType: 'application/json',
  content: {
    theTop: {
      child1: [ 'c1v1', 'c1v2' ],
      child2: [ 'c2v1', 'c2v2' ],
      child3: [ { c3a: 'c3v1' }, { c3b: 'c3v2' } ]
  } }
}).result().then(function(response) {
  // (2) Patch the document
  const pb = marklogic.patchBuilder;
  return db.documents.patch(response.documents[0].uri,
    // Replace the value of an array item by position, or
    // insert a new one in the target position.
```

```
    pb.replaceInsert('/theTop/child1[1]',
              '/theTop/array-node("child1")', 'last-child',
              'REPLACED1'),
    pb.replaceInsert('/theTop/child1[3]', '/theTop/child1[2]',
'after',
              'INSERTED1'),

    // Replace an array item that has a specific value, or
    // insert a new item with that value at the end of the array
    pb.replaceInsert('/theTop/child2[. = "c2v1"]',
              '/theTop/node("child2")', 'last-child',
              'REPLACED2'),
    pb.replaceInsert('/theTop/child2[. = "INSERTED2"]',
              '/theTop/array-node("child2")', 'last-child',
              'INSERTED2'),

    // Replace the value of an object that is an array item, or
    // insert an equivalent object at the end of the array
    pb.replaceInsert('/theTop/child3[c3a]',
              '/theTop/node("child3")', 'last-child',
              { REPLACED3: 'c3rv'}),
    pb.replaceInsert('/theTop/child3[INSERTED3]',
              '/theTop/node("child3")', 'last-child',
              { INSERTED3: 'c3iv'})
  ).result();
}).then(function(response) {
  // (3) Emit the resulting document
  return db.documents.read(response.uri).result();
}).then(function(documents) {
  console.log(JSON.stringify(documents[0].content, null, 2));
}, function(error) { console.log(error); throw error; }
);
```

The following table shows how applying the patch changes the target document.

Before Update	After Update

{"theTop": {
  "child1": ["c1v1", "c1v2"],
  "child2": ["c2v1", "c2v2"],
  "child3": [
  { "c3a": "c3v1" },
  { "c3b": "c3v2" }
  ]
} }
	{ "theTop": {
	  "child1": [
	  "REPLACED1",
	  "c1v2",
	  "INSERTED1"
	  ],
	  "child2": [
	  "REPLACED2",
	  "c2v2",
	  "INSERTED2"
	  ],
	  "child3": [
	  { "REPLACED3": "c3rv" },
	  { "c3b": "c3v2" },
	  { "INSERTED3": "c3iv" }
	  ]
	} }

Recall that the `select` path identifies the content to replace. When working with an array item, an absolute path is usually required. For example, consider the following patch operation:

```
pb.replaceInsert('/theTop/child1[1]',
                 '/theTop/array-node("child1")', 'last-child',
                 'REPLACED1')
```

The goal is to replace the value of the first item in the array value of `/theTop/child1` if it exists. If the array is empty, insert the new value. That is, one of these two transformations takes place:

```
{"theTop": {"child1": ["c1v1", "c1v2"], ... }
    ==> {"theTop": {"child1": ["REPLACED1", "c1v2"], ... }

{"theTop": {"child1": [], ... }
    ==> {"theTop": {"child1": ["REPLACED1"], ... }
```

The `select` expression, `/theTop/child1[1]`, must target an array item value, while the context expression must target the containing array node by referencing `/theTop/array-node("child1")`. You cannot make the select expression relative to the context expression in this case.

Note that while you can target an entire array item value with `replace-insert`, you cannot target just the value of a property. For example, consider the following array in JSON:

```
"child3": [
  { "c3a": "c3v1" },
  { "c3b": "c3v2" }
]
```

You can use `replace-insert` on the entire object-valued item `{ "c3a": "c3v1" }`, as is done in the example. However, you cannot construct an operation that targets just the value of the property in the object (`"c3v1"`). The replacement of the property value is fundamentally different from inserting a new object in the array. A property (as opposed to the containing object) can only be replaced by deleting it and then inserting a new value.

You cannot use a `replaceInsert` operation to conditionally insert or replace a property because the insertion content and the replacement content requirements differ. However, you can use separate insert and replace operations within the same patch to achieve the same effect.

For example, the following patch inserts a new property named TARGET if it does not already exists, and replaces its value if it does already exist:

```
db.documents.patch('/patch/replace-insert.json',
  pb.insert('/theTop[fn:empty(TARGET)]', 'last-child',
            { TARGET: 'INSERTED' }),
  pb.replace('/theTop/TARGET', 'REPLACED')
)
```

The following table illustrates the effect of applying the patch:

Before Update	After Update

{"parent": {
  "child": "some_value"
}}
	{"parent":{
	  "child": "some_value",
	  "TARGET": "INSERTED"
	}}

{"parent": {
  "child": "some_value",
  "TARGET": "INSERTED"
}}
	{"parent": {
	  "child": "some_value",
	  "TARGET": **"REPLACED"**
	}}

For more details on the JSON document model and traversing JSON documents with XPath, see Working With JSON in the *Application Developer's Guide*.

3.6.5 Example: Remove

This example demonstrates how to use the patch remove (`delete`) operation to make the following changes to a JSON document:

- Remove a property based on type.
- Remove an array item by position, value, or property name.
- Remove all items in an array.

To run the example, copy the following code into a file, then supply it to the `node` command. For example: `node remove.js`.

```
const marklogic = require('marklogic');
const my = require('../my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/remove.json',
  contentType: 'application/json',
  content: {
    props: {
      anyType: [1, 2],
      objOrLiteral: 'anything',
      arrayVal: [3, 4]
    },
    arrayItems: {
      byPos: ['DELETE', 'PRESERVE'],
      byVal: ['DELETE', 'PRESERVE'],
      byPropName: [ {DELETE: 5}, {PRESERVE: 6} ],
      all: ['DELETE1', 'DELETE2']
    }
  }
}).result().then(function(response) {
  // (2) Patch the document
  const pb = marklogic.patchBuilder;
  return db.documents.patch(response.documents[0].uri,
    // Remove a property with any value type
    pb.remove('/props/node("anyType")'),

    // Remove a property with atomic or object value type
    pb.remove('/props/objOrLiteral'),

    // Remove a property with array value type
    pb.remove('/props/array-node("arrayVal")'),

    // Remove all items in an array
    pb.remove('/arrayItems/all'),

    // Remove an array item by position
    pb.remove('/arrayItems/byPos[1]'),

    // Remove an array item by value
    pb.remove('/arrayItems/byVal[. = "DELETE"]'),

    // Remove an object-valued array item by property name
    pb.remove('/arrayItems/byPropName["DELETE"]')
  ).result();
}).then(function(response) {
  // (3) Read the patched document
  return db.documents.read(response.uri).result();
}).then(function(documents) {
```

```
    console.log(JSON.stringify(documents[0].content, null, 2));
  }, function(error) {
    console.log(error); throw error;
  });
```

The following table shows how applying the patch changes the target document.

Before Update	After Update

{ "props": {
"anyType": [1, 2],
"objOrLiteral": "anything",
"arrayVal": [3, 4]
},
"arrayItems": {
"byPos": ["DELETE", "PRESERVE"],
"byVal": ["DELETE", "PRESERVE"],
"byPropName": [
{"DELETE": 5},
{"PRESERVE": 6}
],
"all": ["DELETE1", "DELETE2"]
}
}
	{ "props": { },
	"arrayItems": {
	"byPos": ["PRESERVE"],
	"byVal": ["PRESERVE"],
	"byPropName": [
	{ "PRESERVE": 6 }
	],
	"all": [ ]
	}
	}

Note that when removing properties, you must either use a named node step to identify the target property or be aware of the value type. Consider these 3 operations from the example program:

```
pb.remove('/props/node("anyType")'),
pb.remove('/props/objOrLiteral'),
pb.remove('/props/array-node("arrayVal")')
```

The first operation uses a type agnostic XPath expression, `/props/node("anyType")`. This expression selects any nodes named `anyType`, regardless of the type of value. The second operation uses the XPath expression `/props/objOrLiteral`, which selects the array item values, rather than the containing array node. That is, this operation applied to the original document will delete the contents of the array, not the `arrayVal` property:

```
pb.remove('/props/arrayVal')
==> "arrayVal": [ ]
```

The third form, `/props/array-node("arrayVal")`, deletes the `arrayVal` property, but it will only work on properties with array type. Therefore, if you need to delete a property by name without regard to its type, use an XPath expression of the form `/path/to/parent/node("propName")`.

For more details on the JSON document model and traversing JSON documents with XPath, see Working With JSON in the *Application Developer's Guide*.

3.6.6 Example: Patching Metadata

This example demonstrates patching document metadata. The examples performs the following patch operations:

- Add a document to a collection.
- Remove a document from a collection.
- Modify a permission.
- Add a metadata property.
- Set the quality.
- Add a new values metdata key-value pair.
- Remove a values metadata key-value pair.
- Replace the value of a values metadata key-value pair.

This example uses the metadata patching helper interfaces on `patchBuilder`: `patchBuilderCollections`, `patchBuilderPermissions`, `patchBuilderProperties`, `patchBuilderQuality`, and `patchBuilder.metadataValues`. You can also patch metadata directly, but using the helper interfaces frees you from understanding the layout details of how MarkLogic stores metadata internally.

To run the example, copy the following code into a file, then supply it to the `node` command. For example: `node metadata.js`. Note that the parameter passed to `db.docments.patch` must include a `categories` property that indicates the patch should be applied to metadata rather than content.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

// (1) Ensure the example document doesn't already exist
db.documents.remove('/patch/metadata-example.json')
  .result().then( function() {

// (2) Insert the base document into the database
db.documents.write({
  uri: '/patch/metadata-example.json',
  categories: ['content', 'metadata'],
  contentType: 'application/json',
  content: { key: 'value' },
  collections: [ 'initial1', 'initial2' ],
  permissions: [ {
    'role-name': 'app-user',
    capabilities: ['read']
  } ],
  properties: {
    myProp1: 'some-value',
    myProp2: 'some-other-value'
  },
```

```
    metadataValues: {
      key1: 'value1',
      key2: 2
    }
  }).result().then(function(response) {
    // (3) Patch the document
    const pb = marklogic.patchBuilder;
    return db.documents.patch({
      uri: response.documents[0].uri,
      categories: [ 'metadata' ],
      operations: [
        // Add a collection
        pb.collections.add('INSERTED'),

        // Remove a collection
        pb.collections.remove('initial1'),

        // Add a capability to a role
        pb.permissions.replace(
            {'role-name': 'app-user',
             capabilities: ['read','update']}),

        // Add a property
        pb.properties.add('myProp3', 'INSERTED'),

        // Change the quality
        pb.quality.set(2),

        // Add new values metadata
        pb.metadataValues.add('key3', 'INSERTED'),

        // Remove a values metadata key-value pair
        pb.metadataValues.remove('key2'),

        // Replace the value of a values metadata key
        pb.metadataValues.replace('key1', 'REPLACED')
      ]
    }).result();
  }).then(function(response) {
    // (4) Emit the resulting metadata
    return db.documents.read({
      uris: [ response.uri ],
      categories: [ 'metadata']
    }).result();
  }).then(function(documents) {
    console.log('collections: ' +
                JSON.stringify(documents[0].collections, null, 2));
    console.log('permissions: ' +
                JSON.stringify(documents[0].permissions, null, 2));
    console.log('properties: ' +
                JSON.stringify(documents[0].properties, null, 2));
    console.log('quality: ' +
                JSON.stringify(documents[0].quality, null, 2));
    console.log('metadataValues: ' +
```

```
                JSON.stringify(documents[0].metadataValues, null, 2));
}, function(error) { console.log(error); throw error; });
});
```

The output from the example script should be similar to the following:

```
collections: [
  "initial2",
  "INSERTED"
]
permissions: [
  {
    "role-name": "app-user",
    "capabilities": [
      "read",
      "update"
    ]
  },
  {
    "role-name": "rest-writer",
    "capabilities": [
      "update"
    ]
  },
  {
    "role-name": "rest-reader",
    "capabilities": [
      "read"
    ]
  }
]
properties: {
  "myProp1": "some-value",
  "myProp2": "some-other-value",
  "myProp3": "INSERTED"
}
quality: 2
metadataValues: {
  "key1": "REPLACED",
  "key3": "INSERTED"
}
```

Note that the patch operation that adds the “update” capability to the `app-user` role replaces the entire permission, rather than attempting to insert “update” as a new value in the `capabilities` array. You must operate on each permission as a unit. You cannot modify selected components, such as `role-name` or `capabilities`.

```
// Add a capability to a role
pb.permissions.replace(
          {'role-name': 'app-user',
           capabilities: ['read','update']})
```

Also, notice that the `rest-writer` and `rest-reader` roles are assigned to the document, even though they are never explicitly specified. All documents created using the Node.js, Java, or REST Client APIs have these roles by default. For details, see "Security Requirements" on page 15.

3.7 Creating a Patch Without a Builder

The examples in this chapter use `marklogic.PatchBuilder` to construct a patch using JSON. You can also construct a raw patch without a builder and pass it to `db.documents.patch` as either a JavaScript object or a string.

You must use a raw patch when patching content for XML documents. For details, see "Patching XML Documents" on page 103.

The syntax for raw XML and JSON patches is covered in detail in Partially Updating Document Content or Metadata in the *REST Application Developer's Guide*.

The following call to `db.documents.patch` applies a raw JSON patch that insert an array element:

```
db.documents.patch(response.documents[0].uri,
    { patch: [ {
        insert: [ {
          context: '/theTop/child[2]',
          position: 'after',
          content: 'three'
    } ] } ] }
);
```

In a raw patch, each type of update (insert, replace, replace-insert, remove) is an array of objects, with each object describing one operation of that type. The output from a call to a patch builder operation represents one such operation.

For example, the `insert` operation in the raw patch above is equivalent to the following patch builder call:

```
pb.insert('/theTop/child[2]', 'after', 'three')
```

The patch builder functions correspond to raw patch operations as follows:

PatchBuilder function	Raw patch equivalent

`insert(`
`  context, position,`
`  content, cardinality)`
	`"insert": [ ...,`
	`  { "context": ...,`
	`  "position": ...,`
	`  "content": ...,`
	`  "cardinality": ...`
	`  }, ...`
	`]`

`replace(select, content, cardinality)`	`"replace": [ ...,`
	`  { "select": ...,`
	`  "content": ...`
	`  "cardinality": ...`
	`  }, ...`
	`]`

`replaceInsert(`
`  select, context,`
`  position, content,`
`  cardinality)`
	`"replace-insert": [ ...,`
	`  { "select": ...,`
	`  "context": ...,`
	`  "position": ...,`
	`  "content": ...`
	`  "cardinality": ...`
	`  }, ...`
	`]`

`remove(select, cardinality)`	`"delete": [ ...,`
	`  { "select": ...,`
	`  "cardinality": ...`
	`  }, ...`
	`]`

3.8 Patching XML Documents

You must use a raw XML patch when patching content for XML documents. The patch builder only constructs JSON patch operations, and a JSON patch can only be applied to a JSON document.

You can pass a raw XML patch as a string to `db.documents.patch`. The syntax for raw XML patches is covered in detail in XML Patch Reference in the *REST Application Developer's Guide*.

The following example applies a raw XML patch that inserts a new element as a child of another. The patch is passed as a string in the second parameter of `db.documents.patch`.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);
```

```
// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/raw-patch2.xml',
  contentType: 'application/xml',
  content: '<parent><child>data</child></parent>'
}).result().then(function(response) {
  // (2) Patch the document
  return db.documents.patch(
    response.documents[0].uri,
    '<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">' +
    '  <rapi:insert context="/parent" position="last-child">' +
    '    <new-child>INSERTED</new-child>' +
    '  </rapi:insert>' +
    '</rapi:patch>'
  ).result();
}).then(function(response) {
  // (3) Emit the resulting document
  return db.documents.read(response.uri).result();
}).then(function(documents) {
  console.log(documents[0].content);
}, function(error) { console.log(error); throw error; });
```

The following table shows the document transformation applied by the patch:

Before Update	After Update

`<parent>`
`  <child>data</data>`
`</parent>`
	`<parent>`
	`  <child>data</data>`
	`  <new-child>INSERTED</new-child>`
	`</parent>`

For another Node.js example, see “Example: Custom Replacement Constructors” on page 111. For more details, see “Partially Updating Document Content or Metadata” on page 66 in the *REST Application Developer’s Guide*.

3.9 Constructing Replacement Data on MarkLogic Server

You can use builtin or custom replacement constructor functions to generate the content for a patch operation dynamically on MarkLogic Server. The builtin functions support simple arithmetic and string manipulation. For example, you can use a builtin function to increment the current value of numeric data or concatenate strings. For more complex operations, create and install a custom function.

The following topics are covered:

- Overview of Replacement Constructor Functions

3.9.1 Overview of Replacement Constructor Functions

A *replacement constructor function* is a server-side function that generates content for a patch replace or replace-insert operation.

You can use replacement constructor functions when creating a patch operation using `PatchBuilder.replace` and `PatchBuilder.replaceInsert` functions, or the `replace` and `replace-insert` operations of a raw patch. The replacement constructor function call specification takes the place of replacement content supplied by your application. The replacement constructor function call is evaluated on MarkLogic Server and usually generates new content relative to the current value.

For example, you could use the builtin `PatchBuilder.multiplyBy` operation to increase the current value of a property by 10%. The following replace operations says "For every value selected by the XPath expression `/inventory/price`, multiply the current value by 1.1". Notice that the `multiplyBy` result is passed to `PatchBuilder.replace` instead of new content.

```
pb.replace('/inventory/price', pb.multiplyBy(1.1))
```

The builtin replacement constuctors are available as methods of `PatchBuilder`. For details, see "Using a Builtin Replacement Constructor" on page 106.

You can also use custom replacement constructors by calling `PatchBuilder.library` and `PatchBuilder.apply`. Use `PatchBuilder.library` to identify a server side XQuery library module that contains your replacement constructor functions, then use `PatchBuilder.apply` to create a patch operation that invokes a function in that module.

For example, the following code snippet creates a patch replace operation that doubles the price of every value selected by the XPath expression `/inventory/price`. The custom `dbl` function is implemented by the XQuery library module installed in the modules database with URI `/ext/marklogic/patch/apply/my-lib.xqy`. The `dbl` function does not expect any argument values, so there is no additional content supplied to `PatchBuilder.apply`.

```
        pb.library('my-lib'),
        pb.apply('dbl')
```

For details, see “Writing a Custom Replacement Constructor” on page 108.

For details on using replacement generator functions in a raw patch, see Constructing Replacement Data on the Server in the *REST Application Developer's Guide*.

3.9.2 Using a Builtin Replacement Constructor

The Node.js Client API includes several builtin server-side functions you can use to dynamically generate the content for a `replace` or `replaceInsert` patch operation. For example, you can use a builtin function to increment the current value of a data item.

The builtin arithmetic functions are equivalent to the XQuery `+`, `-`, `*`, and `div` operators, and accept values castable to the same datatypes. That is, numeric, date, dateTime, duration, and Gregorian (`xs:gMonth`, `xs:gYearMonth`, etc.) values. The operand type combinations are as supported by XQuery; for details, see http://www.w3.org/TR/xquery/#mapping. All other functions expect values castable to string.

The `PatchBuilder` interface includes methods corresponding to each builtin function. If you use a raw patch rather than the patch builder, see Constructing Replacement Data on the Server in the *REST Application Developer's Guide*.

The table below lists the available builtin replacement constructor functions. In the table, *`$current`* represents the current value of the target of the replace operation; *`$arg`* and *`$argN`* represent argument values passed in by the patch. For details, see the Node.js API Reference. The Apply Operation Name column lists the name of the equivalent operation for use with `patchBuilder.apply`.

PatchBuilder Function	Apply Operation Name	Num Args	Effect
`add`	`ml.add`	1	`$current + $arg`
`subtract`	`ml.subtract`	1	`$current - $arg`
`multiplyBy`	`ml.multiply`	1	`$current * $arg`
`divideBy`	`ml.divide`	1	`$current div $arg`
`concatBefore`	`ml.concat-before`	1	`fn:concat($arg, $current)`
`concatAfter`	`ml.concat-after`	1	`fn:concat($current, $arg)`

PatchBuilder Function	Apply Operation Name	Num Args	Effect
`concatBetween`	`ml.concat-between`	2	`fn:concat($arg1, $current, $arg2)`
`substringBefore`	`ml.substring-before`	1	`fn:substring-before($current, $arg)`
`substringAfter`	`ml.substring-after`	1	`fn:substring-after($current, $arg)`
`replaceRegex`	`ml.replace-regex`	2 or 3	`fn:replace($current, $arg1, $arg2, $arg3)`

3.9.3 Passing Parameters to a Replacement Constructor

When using a patch builder to construct a call to a builtin or custom replacement constructor, simply pass the expected arguments to the `patchBuilder` method.

For example, `patchBuilder.concatBetween` concatenates each selected value between two strings supplied as input. Therefore, the `concatBetween` method takes the two input string as arguments. The following example concatenates the strings “fore” and “aft” on either side of the current value of a selected data item.

```
pb.replace('/some/path/expr',
           pb.concatBetween('fore', 'aft'))
```

In a raw patch, you supply the input argument values in the `content` property of the operation. For details, see Using a Replacement Constructor Function in the *REST Application Developer's Guide*.

Use `patchBuilder.apply` and `patchBuilder.datatype` to explicitly specify the datatype of an input argument value. You can choose among the types supported by the XML schema; for details, see http://www.w3.org/TR/xmlschema-2/#built-in-datatypes. Omit the namespace prefix on the type name.

For example, the following call explicitly specifies `xs:long` as the datatype for the input value to the raw patch operation equivalent of calling `patchBuilder.multiplyBy`. For a list of the builtin function names usable with apply, see the table in “Using a Builtin Replacement Constructor” on page 106.

```
pb.replace('/some/path/expr',
  p.apply('ml.add', p.datatype('long'), '92233720368547758O7'))
```

3.9.4 Using a Custom Replacement Constructor

Before you can use a custom replacement constructor, the XQuery library module containing your constructor implementation must be installed in the modules database of your REST API instance. For details, see “Installing or Updating a Custom Replace Library” on page 109.

To bring a library of custom replacement constructor functions into scope for a patch operation, include the result of calling `PatchBuilder.library` in your patch. The `library` operation tells the API how to locate your implementation on MarkLogic Server.

For example, the following `library` call indicates that any custom replacement constructor functions used by the patch are in the XQuery module with modules database URI `/ext/marklogic/patch/apply/my-replace-lib.xqy`.

```
pb.patch('/some/doc.json',
  pb.library('my-replace-lib.xqy'),
  ...)
```

You can only use one such library per patch, but you use multiple functions from the same library.

Use `PatchBuilder.apply` to construct a "call" to a function in your library. For example, if `my-replae-lib.xqy` contains a function called `dbl`, you can use by including the result of the following call in your patch:

```
pb.patch('/some/doc.json',
  pb.library('my-replace-lib.xqy'),
  pb.apply('dbl')
  /* additional patch operations */)
```

If the function expects input arguments, include them in the `apply` call:

```
pb.apply('doSomething', 'arg1Val', 'arg2Val')
```

You are responsible for passing in values of the type(s) expected by the named function. No type checking is performed for you.

3.9.5 Writing a Custom Replacement Constructor

This section covers requirements for implementing a custom replacement constructor. For an example implementation, see "Example: Custom Replacement Constructors" on page 111.

You can create your own functions to generate content for the `replace` and `replace-insert` operations using XQuery. A custom replacement generator function has the following XQuery interface:

```
declare function module-ns:func-name(
  $current as node()?,
  $args as item()*
) as node()*
```

The target node of the replace (or replace-insert) operation is provided in `$current`. If the function is invoked as an insert operation on behalf of a replace-insert, then `$current` is empty.

The argument list supplied by the operation is passed through `$args`. You are responsible for validating the argument values. If the arguments supplied by the patch operation is a JSON array or a sequence of XML `<rapi:value/>` elements, then $args is the result of applying the `fn:data` function to each value. If an explicit datatype is specified by the patch operation, the cast is applied before invoking your function.

Your function should report errors using `fn:error` and `RESTAPI-SRVEXERR`. For details, see "Error Reporting in Extensions and Transformations" on page 247.

To use a patch builder to construct references to your module, you must adhere to the following namespace and installation URI convention:

- Your module must be in the namespace `http://marklogic.com/patch/apply/`*`yourModuleName`*`.`
- Your module must be installed in the modules database under a URI of the form `/ext/marklogic/patch/apply/`*`yourModuleName`*. This happens automatically if you install your module using `db.config.patch.replace.write`.

For example, if your library module includes the following module namespace declaration:

```
module namespace my-lib = "http://marklogic.com/patch/apply/my-lib";
```

And is installed in the modules database with the following URI:

```
/ext/marklogic/patch/apply/my-lib.xqy
```

Then the following patch successfully applies the function `dbl` in `my-lib.xqy`:

```
pb.patch('/some/doc.json',
  pb.library('my-lib.xqy'),
  pb.replace('/some/path/expr', pb.apply('dbl')),
  ...)
```

This shorthand convention does not apply to raw patches. A raw patch must explicitly specify the complete namespace and module path in the `replace-library` directive, even if you follow the convention naming convention.

3.9.6 Installing or Updating a Custom Replace Library

To install or update custom replacement constructor functions, place your function(s) into an XQuery library module and install the module and any dependent libraries in the modules database associated with your REST API instance.

Use `config.patch.replace.write` to install your module(s). This function ensures your module is installed using the modules database URI convention expected by `PatchBuilder`. You must ensure your module uses the required namespace convention; for details, see "Writing a Custom Replacement Constructor" on page 108.

For example, the following script installs a module into the modules database with the URI `/ext/marklogic/patch/apply/my-lib.xqy`. The implementation is read from a file with pathname `./my-lib.xqy`. The module is executable by users with the `app-user` role.

```
const fs = require('fs');
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.config.patch.replace.write('my-lib.xqy',
  [{'role-name': 'app-user', capabilities: ['execute']}],
  fs.createReadStream('./my-lib.xqy')
).result(function(response) {
  console.log('Installed module ' + response.path);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

Note that only the module name portion of the URI (`'my-lib.xqy'`) is passed in. The remainder of the expected URI is constructed for you.

If you do not specify any permissions when writing the implementation to the modules database, the module is only executable by users with the `rest-admin` role.

For an end-to-end example, see “Example: Custom Replacement Constructors” on page 111.

If your library module requires dependent libraries, you can install them using the `extlibs` interface. The `extlibs` interface allows you to manage modules database assets at both the directory and file level. For details, see “Managing Assets in the Modules Database” on page 257.

Calling `db.config.patch.replace.write` is equivalent to calling `db.config.extlibs.write` and setting the path parameter to a value that conforms to the `PatchBuilder` convention. For example, the following call performs an installation equivalent to the above use of `db.config.patch.replace.write`:

```
db.config.extlibs.write({
  path: '/marklogic/patch/apply/my-lib.xqy',
  permissions: [
    {'role-name': 'app-user', capabilities: ['execute']}
  ],
  contentType: 'application/xquery',
  source: fs.createReadStream('./my-lib.xqy')
})
```

3.9.7 Uninstalling a Custom Replace Library

To remove a module that contains custom replacement constructor functions, use `db.config.patch.replace.remove`. For example, the following call removes the module `my-lib.xqy` that was previously installed using `db.config.patch.replace.write`:

```
db.config.patch.replace.remove('my-lib.xqy');
```

3.9.8 Example: Custom Replacement Constructors

This example walks you through installing and using an XQuery library module that contains custom replacement constructor functions.

The example installs an XQuery library module containing 2 custom replacement constructors, named `dbl` and `min`. The `dbl` function creates a new node whose value is double that of the original input; it accepts no additional arguments. The `min` function creates a new node whose value in the minimum of the current value and the values passed in as additional arguments.

For simplicity, this example skips most of the input data validation that a production implementation should include. For example, the `min` function accepts JSON number nodes and XML elements as input, but it does not allow for boolean, text, or date input. Nor does `min` perform any validation on the additional input args.

After installing the replacement content generators, a patch is applied to double the value of oranges (from 10 to 20) and select the lowest price for pears.

Use the following procedure to set up the files used by the example:

1. Copy the following XQuery module into a file named `my-lib.xqy`. This file contains the implementation of the custom replacement constructors.

```
xquery version "1.0-ml";

module namespace my-lib = "http://marklogic.com/patch/apply/my-lib";

(: Double the value of a node :)
declare function my-lib:dbl(
  $current as node()?,
  $args as item()*
) as node()*
{
  if ($current/data() castable as xs:decimal)
  then
    let $new-value := xs:decimal($current) * 2
    return
      typeswitch($current)
      case number-node()       (: JSON :)
        return number-node {$new-value}
      case element()           (: XML :)
        return element {fn:node-name($current)} {$new-value}
      default return fn:error((), "RESTAPI-SRVEXERR",
         ("400", "Bad Request",
            fn:concat("Not an element or number node: ",
                      xdmp:path($current))
            ))
  else fn:error((), "RESTAPI-SRVEXERR",
```

```
      ("400", "Bad Request", fn:concat("Non-decimal data: ", $current)
    ))
};

(: Find the minimum value in a sequence of value composed of :)
(: the current node and a set of input values. :)
declare function my-lib:min(
  $current as node()?,
  $args as item()*
) as node()*
{
  if ($current/data() castable as xs:decimal)
  then
    let $new-value := fn:min(($current, $args))
    return
      typeswitch($current)
      case element()           (: XML :)
        return element {fn:node-name($current)} {$new-value}
      case number-node()       (: JSON :)
        return number-node {$new-value}
      default return fn:error((), "RESTAPI-SRVEXERR",
        ("400", "Bad Request",
           fn:concat("Not an element or number node: ",
                     xdmp:path($current))
           ))
  else fn:error((), "RESTAPI-SRVEXERR", ("400", "Bad Request",
    fn:concat("Non-decimal data: ", $current)))
};
```

2. Copy the following script into a file named `install-udf.js`. This script installs the above XQuery module. For demonstration purposes, the module is installed such that the role `app-user` has permission to execute the contained functions.

```
const fs = require('fs');
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.config.patch.replace.write('my-lib.xqy',
  [ {'role-name': 'app-user', capabilities: ['execute']} ],
  fs.createReadStream('./my-lib.xqy')
).result(function(response) {
  console.log('Installed module ' + response.path);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

3. Copy the following script into a file named `udf.js`. This script inserts a base document into the database and applies a patch that uses the `dbl` and `min` replacement content constructors.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/udfs.json',
  contentType: 'application/json',
  content: {
    inventory: [
      {name: 'orange', price: 10},
      {name: 'apple', price: 15},
      {name: 'pear', price: 20}
  ] }
}).result().then(function(response) {
  // (2) Patch the document
  const pb = marklogic.patchBuilder;
  return db.documents.patch(response.documents[0].uri,
    pb.library('my-lib.xqy'),
    pb.replace('/inventory[name eq "orange"]/price', pb.apply('dbl')),
    pb.replace('/inventory[name eq "pear"]/price',
               pb.apply('min', 18, 21))
  ).result();
}).then(function(response) {
  // (3) Emit the resulting document
  return db.documents.read(response.uri).result();
}).then(function(documents) {
  console.log(JSON.stringify(documents[0].content, null, 2));
}, function(error) { console.log(error); throw error; });
```

Use the following procedure to run the example. This procedure installs the replacement content constructor module, inserts a base document in the database, patches the document, and displays the update document contents.

1. Install the replacement content constructor module:

```
node install-udf.js
```

2. Insert and patch a document, using the `dbl` and `min` functions.

```
node udf.js
```

You should see output similar to the following:

```
{
  "inventory": [
    {
```

```
      "name": "orange",
      "price": 20
    },
    {
      "name": "apple",
      "price": 15
    },
    {
      "name": "pear",
      "price": 18
    }
  ]
}
```

The price of oranges is doubled, from 10 to 20 by the `dbl` function, due to the following patch operation:

```
pb.replace('/inventory[name eq "orange"]/price', pb.apply('dbl'))
```

The value of pears is lowered from 20 to 18 by the `min` function, due to the following patch operation:

```
pb.replace('/inventory[name eq "pear"]/price',
           pb.apply('min', 18, 21))
```

The XPath expression used in each patch operation selects a number node (`price`) in the target document and the replacement content constructor functions construct a new number node:

```
typeswitch($current)
  case element()            (: XML :)
    return element {fn:node-name($current)} {$new-value}
  case number-node()        (: JSON :)
    return number-node {$new-value}
```

You cannot simply return the new value. You must return a complete replacement node. To learn more about the JSON document model and JSON node constructors, see Working With JSON in the *Application Developer's Guide*.

The typeswitch on the node type of `$current` also enables the example replacement constructors to work with XML input. Run the following script to apply an equivalent to an XML document, using the previously installed `dbl` and `min` replacement content constructors. As described in "Patching XML Documents" on page 103, you must use a raw patch rather than a builder when working with XML documents.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/udf.xml',
```

```
    contentType: 'application/xml',
    content:
      '<inventory>' +
        '<item>' +
          '<name>orange</name>' +
          '<price>10</price>' +
        '</item>' +
        '<item>' +
          '<name>apple</name>' +
          '<price>15</price>' +
        '</item>' +
        '<item>' +
          '<name>pear</name>' +
          '<price>20</price>' +
        '</item>' +
      '</inventory>'

  }).result().then(function(response) {
    // (2) Patch the document
    return db.documents.patch(
      response.documents[0].uri,
      '<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">' +
        '<rapi:replace-library ' +
            'at="/ext/marklogic/patch/apply/my-lib.xqy" ' +
            'ns="http://marklogic.com/patch/apply/my-lib" />' +
        '<rapi:replace ' +
            'select="/inventory/item[name eq \'orange\']/price" ' +
            'apply="dbl" />' +
        '<rapi:replace ' +
            'select="/inventory/item[name eq \'pear\']/price" ' +
            'apply="min">' +
          '<rapi:value>18</rapi:value>' +
          '<rapi:value>21</rapi:value>' +
        '</rapi:replace>' +
      '</rapi:patch>'
    ).result();
  }).then(function(response) {
    // (3) Emit the resulting document
    return db.documents.read(response.uri).result();
  }).then(function(documents) {
    console.log(documents[0].content);
  }, function(error) { console.log(error); throw error; });
```

Notice that in a raw patch, you must explicitly specify the module path and module namespace in the `replace-library` directive:

```
<rapi:replace-library at="/ext/marklogic/patch/apply/my-lib.xqy"
                      ns="http://marklogic.com/patch/apply/my-lib" />'
```

When you use `PatchBuilder` to construct a JSON patch, the call to `PatchBuilder.library` fills these details in for you, assuming you follow the installation path conventions described in “Installing or Updating a Custom Replace Library” on page 109.

3.9.9 Additional Operations

The db.config.patch.replace interface offers additional methods for managing library modules containing replacement content constructor functions, including the following:

- `db.config.patch.replace.read` - Retrieve the implementation of a replace library. This returns a module installed using `db.config.patch.replace.write`.
- `db.config.patch.replace.list` - Retrieve a list of all installed replace library modules.

For details, see *Node.js Client API Reference*.

4.0 Querying Documents and Metadata

This chapter covers the following topics related to querying database content and metadata using the Node.js Client API:

4.1 Query Interface Overview

The Node.js Client API includes interfaces that enable you to search documents and query lexicons using a variety of query types. The following interfaces support query operations:

Method	Description
`marklogic.queryBuilder`	Construct a string query, QBE, or structured query to use with `DatabaseClient.documents.query`. For details, see Understanding the queryBuilder Interface.
`DatabaseClient.documents.query`	Search for documents that match a string query, structured query, combined query, or Query By Example (QBE), returning a search results summary, matching documents, or both. For details, see "Searching with Query By Example" on page 135, "Searching with String Queries" on page 125, or "Searching with Structured Queries" on page 140.

Method	Description
`DatabaseClient.queryCollection`	Search for the persisted JavaScript objects in a collection, and return the matching objects.
`DatabaseClient.valuesBuilder`	Construct a values query to use with `DatabaseClient.values.read`. For details, see "Querying Lexicons and Range Indexes" on page 152.
`DatabaseClient.values.read`	Query the values or tuples (co-occurrences) in lexicons or range indexes. For details, see "Querying Lexicons and Range Indexes" on page 152.
`DatabaseClient.documents.suggest`	Match strings in a lexicon to provide search term completion suggestions. For details, see "Generating Search Term Completion Suggestions" on page 173.
`DatabaseClient.config.query`	Manage query related customizations stored in the modules database, including search result transforms, snippeters, and string query parsers.

4.2 Introduction to Search Concepts

This section provides a brief introduction to search concepts and the capabilities exposed by the Node.js Client API. Search concepts are covered in detail in the *Search Developer's Guide*.

You can query a MarkLogic Server database in two ways: by searching documents contents and metadata, or by querying value and word lexicons created from your content. This topic deals with searching content and metadata. For lexicons, see "Querying Lexicons and Range Indexes" on page 152.

This section covers the following topics:

4.2.1 Search Overview

Performing a search consists of the following basic phases:

1. Build up a set of criteria that defines your desired result set

2. Refine the result set by defining attributes such as the number of results to return or the sort order.

3. Search the database or a lexicon.

The Node.js Client API includes the `marklogic.queryBuilder` interface that abstract away many of the structural details of defining and refining your query. For details, see “Understanding the queryBuilder Interface” on page 122. Use `DatabaseClient.documents.query` to execute your query operation.

MarkLogic Server supports many different kinds of search criteria, such as matching phrases, specific values, ranges of values, and geospatial regions. These and other query types are explored in “Types of Query” on page 120. You can express your search criteria using one of several query styles; for details, see “Query Styles” on page 119.

Query result refinements include whether or not to return entire documents, content snippets, facet information, and/or aggregate results. You can also define your own snippeting algorithm or custom search result tranform. For details, see “Refining Query Results” on page 165.

To perform iterative searches over the database at a fixed point in time, pass a `Timestamp` parameter in your `query` call. For details, see “Performing Point-in-Time Operations” on page 23.

You can also analyze lexicons created from your documents using `marklogic.valuesBuilder` and `DatabaseClient.values.read`. For details, see “Querying Lexicons and Range Indexes” on page 152.

4.2.2 Query Styles

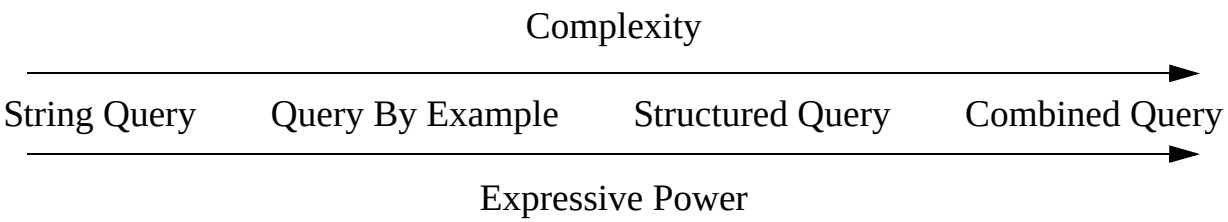
When you search document content and metadata using the Node.js Client API, you can express your search criteria using the following query styles. The syntax of each style is different, and the expressive power varies.

Query Style	Description
Query By Example (QBE)	Search documents by modeling the structure of the documents you want to match. For details, see “Searching with Query By Example” on page 135 and `queryBuilder.byExample`.
String Query	Search documents and metadata using a Google-style query string such as a user enters in a search box. For example, a query of the form “cat AND dog” matches all documents containing the phrases “cat” and “dog”. For details, see “Searching with String Queries” on page 125 and `queryBuilder.parsedFrom`.

Query Style	Description
Structured Query	Search documents and metadata by building up complex queries from a rich set of sub-query types. For details, see “Searching with Structured Queries” on page 140.
Combined Query	Search documents and metadata using a query object that enables you to combine the other query styles plus query options. Combined query is an advanced feature for users who prefer to build queries manually. For details see “Searching with Combined Query” on page 150.

All the query styles support a rich set of search features, but generally, QBE is more expressive than string query, structured query is more expressive than QBE, and combined query is more expressive than any of the others since it is a superset. String query and QBE are designed for ease of use and cover a wide range of search needs. However, they do not provide the same level of control over the search as structured query and combined query do.

The following diagram illustrates this tradeoff, at a high level.



You can combine a string query and structured query criteria in a single query operation. QBE cannot be combined with the other two query styles.

For more details, see Overview of Search Features in MarkLogic Server in the *Search Developer's Guide*.

4.2.3 Types of Query

A query encapsulates your search criteria. No matter what query style you use (string, QBE, or structured), your criteria fall into one or more of the query types described in this section.

The following query types are basic search building blocks that describe the content you want to match.

- Range: Match values that satisfy a relational expression. You can express conditions such as “less than 5” or “not equal to true”. A range query must be backed by a range index.
- Value: Match an entire literal value, such as a string or number, in a specific JSON property or XML element. By default, value queries use exact match semantics. For example, a search for “mark” will not match “Mark Twain”.

- Word: Match a word or phrase in a specific JSON property or XML element or attribute. In contrast to a value query, a word query will match a subset of a text value and does not not use exact match semantics by default. For example, a search for “mark” will match “Mark Twain”, in the specified context.
- Term: Match a word or phrase anywhere it appears. In contrast to a value query, a term query will match a subset of a text value and does not use exact match semantics by default. For example, a search for “mark” will match “Mark Twain”.

Additional query types enable you to build up complex queries by combining the basic content queries with each other and with criteria that add additional constraints. The additional query types fall into the following categories.

- Logical Composers: Express logical relationships between criteria. You can build up compound logical expressions such as “*x* AND (*y* OR *z*)”.
- Document Selectors: Select documents based on collection, directory, or URI. For example, you can express criteria such as “*x* only when it occurs in documents in collection *y*”.
- Location Qualifiers: Further limit results based on where the match appears. For example, “*x* only when contained in JSON property *z*”, or “*x* only when it occurs within *n* words of *y*”, or “*x* only when it occurs in a document property”.

With no additional configuration, string queries support term queries and logical composers. For example, the query string “cat AND dog” is implicitly two term queries, joined by an “and” logical composer.

However, you can easily extend the expressive power of a string query using parse bindings to enable additional query types. For example, if you use a range query binding to tie the identifier “cost” to a specific indexed JSON property, you enable string queries of the form “cost GT 10”. For details, see “Searching with String Queries” on page 125.

In a QBE, content matches are value queries by default. For example, a QBE search criteria of the form `{'my-key': 'desired-value'}` is implicitly a value query for the JSON property `'my-key'` whose value is exactly `'desired-value'`. However, the QBE syntax includes special property names that enable you to construct other types of query. For example, use `$word` to create a word query instead of a value query: `{'my-key': {'$word': 'desired-value'}}`. For details, see “Searching with Query By Example” on page 135.

For structured query, the `queryBuilder` interface includes builders corresponding to all the query types. You can use these builders in combination with each other. Every `queryBuilder` method that return a `queryBuilder.Query` creates a query or sub-query that falls into one of the above query categories. For details, see “Searching with Structured Queries” on page 140.

4.2.4 Indexing

Range queries must be backed by an index. Even queries that do not strictly require a backing index can benefit from indexing by enabling unfiltered searches; for details, see Fast Pagination and Unfiltered Searches in the *Query Performance and Tuning Guide*.

You can create range indexes using the Admin Interface, the XQuery Admin API, and the REST Management API. You can also use the Configuration Manager or REST Packaging API to copy index configurations from one database or host to another. For details, see the following references:

- Range Indexes and Lexicons in the *Administrator's Guide*
- Using the Configuration Manager in the *Administrator's Guide*
- `PUT:/manage/v2/databases/{id|name}/properties` in the *MarkLogic REST API Reference*

Use the element range index interfaces to create indexes on JSON properties. For purposes of index configuration, a JSON property is equivalent to an XML element in no namespace.

You can use the binding feature of the Node.js Client API to bind an index reference to a name that can be used in string queries. For details, see "Using Constraints in a String Query" on page 128 and "Generating Search Term Completion Suggestions" on page 173. Values queries on lexicons and indexes also rely on index references. For details, see "Building an Index Reference" on page 157.

4.3 Understanding the queryBuilder Interface

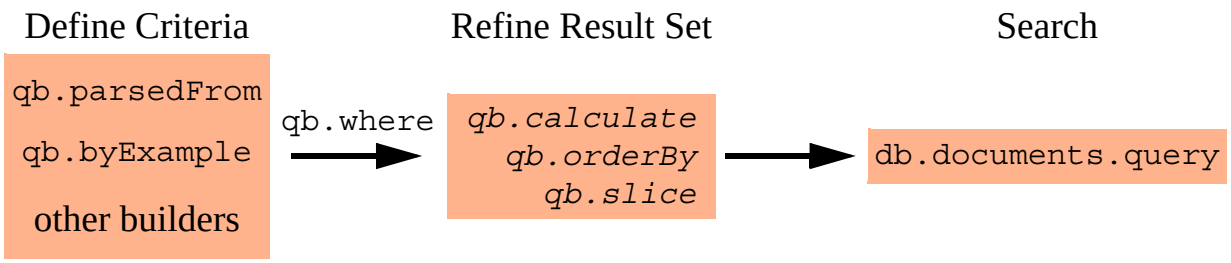
Performing a search using the `marklogic.queryBuilder` interface consists of the following phases:

1. Build up a set of search criteria, creating a query that defines your desired result set.

2. Refine the result set by defining attributes such as the number of results to return or the sort order.

3. Search the database.

The following diagram illustrates using the Node.js Client API to define and execute a search using `queryBuilder` and `DatabaseClient.documents.query`. In the diagram, "qb" represents a `queryBuilder` object, and "db" represents a `DatabaseClient` object. The functions in italics are optional.



The following procedure expresses these steps in more detail:

1. Define your search criteria using string query (`qb.parsedFrom`), QBE (`qb.byExample`), or structured query (other builders, such as `qb.word`, `qb.range`, and `qb.or`). For example:

   ```
   qb.parsedFrom("dog")
   ```

   You can pass a string and one or more structured builders together, in which case they are AND'd together. You cannot combine a QBE with the other query types.

2. Encapsulate your criteria in a query by passing them to `queryBuilder.where`. This produces a `queryBuilder.BuiltQuery` object suitable for passing to `DatabaseClient.documents.query`, with or without further result set refinement.

   ```
   qb.where(qb.parsedFrom("dog"))
   ```

3. Optionally, apply further result set refinements to your query. Any or all of the following steps can be skipped, depending on the results you want.

   a. Use `queryBuilder.slice` to select a subset of documents from the result set and/or specify a server-side transformation to apply to the selected results. The default slice is the first 10 documents, with no transformations.

   b. Use `queryBuilder.orderBy` to specify a sort key and/or sorting direction.

   c. Use `queryBuilder.calculate` to request one or more aggregate calculations on the result set.

4. Optionally, use `queryBuilder.withOptions` to add further refinements to your search, such as specifying low level search options or a transaction id, or requesting query debugging information.

5. Perform the search by passing your final `BuiltQuery` object to the `DatabaseClient.documents.query` function. For example:

   ```
   db.documents.query(qb.where(qb.parsedFrom("dog")))
   ```

The following table contains examples of using `queryBuilder` to construct an equivalent query in each of the available query styles. The queries match documents containing both the phrases “cat” and “dog”. Notice that only the query building portion of the search varies based on the chosen query style.

Query Style	Code Snippet

string	`db.documents.query(`
	`  qb.where(`
	`  qb.parsedFrom('cat AND dog')`
	`  ).orderBy(qb.sort('descending')`
	`  .slice(0,5)`
	`)`

QBE	`db.documents.query(`
	`  qb.where(`
	`  qb.byExample({`
	`  $and:[{$word:'cat'},{$word:'dog'}]`
	`  })`
	`  ).orderBy(qb.sort('descending')`
	`  .slice(0,5)`
	`)`

structured	`db.documents.query(`
	`  qb.where(`
	`  qb.and(qb.term('cat'), qb.term('dog'))`
	`  ).orderBy(qb.sort('descending')`
	`  .slice(0,5)`
	`)`

combined string and structured	`db.documents.query(`
	`  qb.where(`
	`  qb.term('cat'),`
	`  qb.parsedFrom('dog')`
	`  ).orderBy(qb.sort('descending')`
	`  .slice(0,5)`
	`)`

For details, see one of the following topics:

- “Searching with String Queries” on page 125
- “Searching with Query By Example” on page 135
- “Searching with Structured Queries” on page 140

4.4 Searching with String Queries

A string query is a simple, but powerful text string, usually corresponding to query text entered into your application by users via a search box. This section includes the following topics:

4.4.1 Introduction to String Query

The MarkLogic Server Search API default search grammar allows you to quickly construct simple searches such as “cat”, “cat AND dog”, or “cat NEAR dog”. Such a string query often represents query text entered into a search box by a user.

The default grammar supports operators such as AND, OR, NOT, and NEAR, plus grouping. For grammar details, see Searching Using String Queries in the *Search Developer’s Guide*.

The Node.js client supports string queries through the `queryBuilder.parsedFrom` method. For example, to construct a query that matches documents containing the phrases “cat” and “dog”, use the following queryBuilder call:

```
qb.parsedFrom('cat AND dog')
```

For details, see “Example: Basic String Query” on page 126 and the Node.js API Reference.

By default, DatabaseClient.documents.query returns an array of document descriptors, one per matched document, including the document contents. You can further refine the search in various ways, such as controlling which and how many documents, returning snippets and/or facets, and returning a result summary instead of entire documents. For details, see “Refining Query Results” on page 165.

The string grammar also supports the application of search constraints to query terms. For example, you can include a term of the form *`constraintName:value`* or *`constraintName relationalOp value`* to limit matches to cases where the value satisfies the constraint. *`ConstraintName`* is the name of a constraint you configure into your query.

For example, if you define a word constraint named “location” over a JSON property of the same name, then the string query “location:oslo” only matches the term “oslo” when it occurs in the value of the `location` property.

Similarly, if you define a range constraint over a number-valued property, bound to the name "votes", then you can include relational expressions over the value of the property such as "votes GT 5".

The Node.js client supports constraints in string queries through parse bindings that bind a constraint definition to the name usable in a query. Use the `queryBuilder.parseBindings` function to define such bindings. For example:

```
qb.parsedFrom(theQueryString, qb.parseBindings(binding definitions...))
```

For details, see "Using Constraints in a String Query" on page 128 and "Using a Custom Constraint Parser" on page 131.

4.4.2 Example: Basic String Query

The following example script assumes the database is seeded with data "Loading the Example Data" on page 179. The script searches for all documents containing the phrase "oslo".

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const qb = marklogic.queryBuilder;

db.documents.query(
    qb.where(qb.parsedFrom('oslo'))
).result( function(results) {
  console.log(JSON.stringify(results, null, 2));
});
```

The search returns an array of document descriptors, one descriptor per matching document. Each descriptor includes the document contents.

For example, if the file `string-search.js` contains the above script, then the following command produces the results below. The search matches two documents, corresponding to contributors located in Oslo, Norway.

```
$ node string-search.js
[
  {
    "uri": "/contributors/contrib1.json",
    "category": "content",
    "format": "json",
    "contentType": "application/json",
    "contentLength": "230",
    "content": {
      "Contributor": {
        "userName": "souser10002@email.com",
        "reputation": 446,
        "displayName": "Lars Fosdal",
        "originalId": "10002",
```

```
        "location": "Oslo, Norway",
        "aboutMe": "Software Developer since 1987, mainly using
Delphi.",
        "id": "sou10002"
      }
    }
  },
  {
    "uri": "/contributors/contrib2.json",
    "category": "content",
    "format": "json",
    "contentType": "application/json",
    "contentLength": "202",
    "content": {
      "Contributor": {
        "userName": "souser1000634@email.com",
        "reputation": 272,
        "displayName": "petrumo",
        "originalId": "1000634",
        "location": "Oslo, Norway",
        "aboutMe": "Developer at AspiroTV",
        "id": "sou1000634"
      }
    }
  }
]
```

To return a search summary instead of the document contents, use `queryBuilder.withOptions` to set `categories` to `'none'`. For example:

```
db.documents.query(
    qb.where(qb.parsedFrom('oslo')).withOptions({categories: 'none'})
)
```

Now, the result is a search summary that includes a count of the number of matches (2), and snippets of the matching text in each document:

```
[{
  "snippet-format": "snippet",
  "total": 2,
  "start": 1,
  "page-length": 10,
  "results": [...snippets here...],
  "qtext": "oslo",
  "metrics": {
    "query-resolution-time": "PT0.005347S",
    "facet-resolution-time": "PT0.000067S",
    "snippet-resolution-time": "PT0.001523S",
    "total-time": "PT0.007753S"
  }]
```

You can also refine your results in other ways. For details, see “Refining Query Results” on page 165.

4.4.3 Using Constraints in a String Query

The string query interfaces enable you to create parse bindings that define how to interpret parts of the query. You can define a binding between a name and a search constraint so that when a query term is prefixed by the bound name, the associated constraint is applied to search for that term. You can create parse bindings on word, value, range, collection, and scope constraints.

For example, you can define a binding between the name “rep” and a constraint that limits the search to matching values in a JSON property named “reputation”. Then, if a string query includes a term of the form `rep:value`, the constraint is applied to the search for the value. Thus, the following term mean “find all occurrences of the reputation property where the value is 120”:

```
rep:120
```

For details, see Using Relational Operators on Constraints in the *Search Developer's Guide*.

> **Note:** Range constraints, such as the contraint on reputation used here, must be backed by a corresponding range index. For details, see “Indexing” on page 122.

Follow these steps to create and apply parse bindings. For a complete example, see “Example: Using Constraints in a String Query” on page 129.

1. Create a binding name specification by calling `queryBuilder.bind` or `queryBuilder.bindDefault`. For example, the following call creates a bind name specification for the name “rep”:

   ```
   qb.bind('rep')
   ```

2. Create a binding between the name (or default) and a constraint by calling one of the `queryBuilder` binding builder methods (`collection`, `range`, `scope`, `value`, or `word`) and passing in the binding name specification. For example, the following call creates a binding between the name 'rep' and a value constraint on the JSON property name 'reputation'.

   ```
   qb.value('reputation', qb.bind('rep'))
   ```

3. Bundle your bindings into a `queryBuilder.ParseBindings` object using `queryBuilder.parseBindings`. For example:

   ```
   qb.parseBindings(
     qb.value('reputation', qb.bind('rep')), ...more bindings..
   )
   ```

4. Pass the parse bindings as the second parameter of `queryBuilder.parsedFrom` to apply them to a specific query. For example:

```
qb.parsedFrom('rep:120',
  qb.parseBindings(
    qb.value('reputation', qb.bind('rep')), ...more bindings..
  )
)
```

You can also create a binding that defines the behavior when the query string is empty, using `queryBuilder.bindEmptyAs`. You can elect to return all results or none. The default is none. Note that because a query without a slice specifier returns matching documents, setting the empty query binding to `all-results` can cause an empty query to retrieve all documents in the database.

The following example returns all search results because the query text is an empty string and empty query binding specifies `all-results`. Calling `queryBuilder.slice` ensures the query will return at most 5 documents.

```
db.documents.query( qb.where(
  qb.parsedFrom('',
    qb.parseBindings(
      qb.bindEmptyAs('all-results')
  ))
)).slice(0,5)
```

4.4.4 Example: Using Constraints in a String Query

This example defines some custom parse binding rules and applies them to a string query based search. The example illustrates the capability described in “Using Constraints in a String Query” on page 128.

The example uses data derived from the `marklogic-samplestack` application. The seed data includes “contributor” JSON documents of the following form:

```
{ "com.marklogic.samplestack.domain.Contributor": {
    "userName": string,
    "reputation": number,
    "displayName": string,
    "originalId": string,
    "location": string,
    "aboutMe": string,
    "id": string
} }
```

The example script applies the following parse bindings to the search:

- The term “rep” corresponds to the value of the `reputation` JSON property. It is bound to a range constraint, so it can be used with relational expressions such as “rep > 100”. This constraint is expressed by the following binding definition:

  ```
  qb.range('reputation', qb.datatype('int'), qb.bind('rep'))
  ```

- Bare terms that are not covered by another constraint are constrained to match a word query on the `aboutMe` JSON property. This constraint is expressed by the following binding definition:

```
qb.word('aboutMe', qb.bindDefault())
```

The database configuration includes an element range index on the `reputation` JSON property with scalar type “int”. This index is required to support the range constraint on `reputation`.

This combination of bindings and configuration causes the following query text to match documents where “marklogic” occurs in the “aboutMe” property. The term “marklogic” is a bare term because it is not qualified by a constraint name.

```
"marklogic"
```

The following query text matches documents where the value of the “reputation” property is greater than 50:

```
marklogic AND rep GT 50
```

You can use these clauses together to match all documents in which the `aboutMe` property contains “marklogic” and the `reputation` property is greater than 50:

```
marklogic AND rep GT 50
```

Without the bindings, the above query matches documents that contain the phrase “marklogic” anywhere, and the sub-expression “rep GT 50” is meaningless because it compares the word “rep” to “50”.

The following script creates the binding and applies them to the search text shown above.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const qb = marklogic.queryBuilder;

db.documents.query( qb.where(
  qb.parsedFrom('marklogic AND rep GT 50',
    qb.parseBindings(
      qb.word('aboutMe', qb.bindDefault()),
      qb.range('reputation', qb.datatype('int'), qb.bind('rep'))
  ))
)).result(function (documents) {
  console.log(JSON.stringify(documents[0].content, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

When run against the `marklogic-samplestack` seed data, the query matches a single contributor and produces output similar to the following:

```
{
  "Contributor": {
    "userName": "souser1601813@email.com",
    "reputation": 91,
    "displayName": "grechaw",
    "originalId": "1601813",
    "location": "Occidental, CA",
    "aboutMe": "XML (XQuery, Java, XML database) software engineer at
MarkLogic. Hardcore accordion player.",
    "id": "sou1601813"
  }
}
```

4.4.5 Using a Custom Constraint Parser

Support for binding word, value, range, collection, and scope constraint parsing is built into the API. If these constraint types do not meet the needs of your application, you can create a binding to a custom constraint parser. Implement the parser as described in Creating a Custom Constraint in the *Search Developer's Guide*.

To apply a custom constraint parser to a string query with the Node.js Client, follow these steps:

1. Create an XQuery module that implements your custom constraint parser. Use the parser interface for structured queries. For details, see Implementing a Structured Query parse Function in the *Search Developer's Guide*. You must following the naming conventions described below.

2. Install your parser XQuery library module in the modules database associated with your REST API instance using `DatabaseClient.config.query.custom.write`. For details, see "Example: Custom Constraint Parser" on page 132.

3. Use `queryBuilder.parseFunction` to create a parse binding between a constraint name and your custom parser.

The Node.js Client API imposes the following naming conventions on your custom constraint implementation:

- Your parse function must be named `parse`.
- Your start and finish facet functions, if present, must be called `start-facet` and `finish-facet`, respectively.
- Your module namespace must be `http://marklogic.com/query/custom/`*`yourModuleName`*, where *`yourModuleName`* is a name of your choosing.

4.4.6 Example: Custom Constraint Parser

This example demonstrates implementing, installing, and using a custom constraint parser with the Node.js Client API. For details, see “Using a Custom Constraint Parser” on page 131.

This example is based on the `marklogic-samplestack` seed data. The data includes contributor documents, installed in the database directory `/contributors/`, and question documents, installed in the database directory `/questions/`.

The example constraint enables constraining a search to either the contributor or question category by including a term of the form `cat:c` or `cat:q` in your query text. The name “cat” is bound to the custom contraint using the queryBuilder parse bindings. The constraint parser defines the values “c” and “q” as corresponding to contributor and question data, respectively.

The example walks through the following steps:

- Implementing the Constraint Parser
- Installing the Constraint Parser
- Using the Custom Constraint in a String Query

4.4.6.1 Implementing the Constraint Parser

The following XQuery module implements the constraint parser. No facet handling functions are provided. The parser generates a `directory-query` based on the caller-supplied category name. The module maintains a mapping between the category names that can appear in query text and the corresponding database directory in the `categories` variable.

```
xquery version "1.0-ml";

module namespace my = "http://marklogic.com/query/custom/ss-cat";
import module namespace search =
  "http://marklogic.com/appservices/search"
  at "/MarkLogic/appservices/search/search.xqy";

(: The category name to directory name mapping:)
declare variable $my:categories :=
  map:new((
    map:entry("c", "/contributors/"),
    map:entry("q", "/questions/")
  ));

(: parser implementation :)
declare function my:parse(
  $query-elem as element(),
  $options as element(search:options)
) as schema-element(cts:query)
{
let $query :=
  <root>{
```

```
    let $cat := $query-elem/search:text/text()
    let $dir :=
      if (map:contains($my:categories, $cat))
      then map:get($my:categories, $cat)[1]
      else "/"
    return cts:directory-query($dir, "infinity")
  }</root>/*
return
(: add qtextconst attribute so that search:unparse will work -
   required for some search library functions :)
element { fn:node-name($query) }
  { attribute qtextconst {
      fn:concat(
        $query-elem/search:constraint-name, ":",
        $query-elem/search:text/text()) },
    $query/@*,
    $query/node()}
};
```

4.4.6.2 Installing the Constraint Parser

The following script installs the constraint parser module in the modules database, assuming the implementation is saved to a file named `ss-cat.xqy`. Installation is performed by calling `DatabaseClient.config.query.custom.write`. The module name passed as the first parameter must have the same basename as the module name in your module namespace declaration (`ss-cat`).

```
const fs = require('fs');
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.config.query.custom.write(
  'ss-cat.xqy',
  [ {'role-name': 'app-user', capabilities: ['execute']} ],
  fs.createReadStream('./ss-cat.xqy')
).result(function(response) {
  console.log('Installed module ' + response.path);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

If you save the script to a file named `install-parser.js`, then running the script should produce results similar to the following:

```
$ node install-parser.sj
Installed module /marklogic/query/custom/ss-cat.xqy
```

4.4.6.3 Using the Custom Constraint in a String Query

To use this constraint, include a parse binding created by `queryBuilder.parseFunction` in your query. The first parameter must match the module name used when installing the implementation.

For example, the following call binds the name “cat” to the custom constraint parser installed above, enable queries to include terms of the form “cat:c” or “cat:q”.

```
qb.parseFunction('ss-cat.xqy', qb.bind('cat'))
```

Note that the module name (`ss-cat.xqy`) is the same as the module name passed as the first parameter to `config.query.custom.write`.

The following script uses the custom constraint to search for occurrences of “marklogic” in documents in the contributors category (“cat:c”) by specifying query text of the form “marklogic AND cat:c”.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const qb = marklogic.queryBuilder;

db.documents.query( qb.where(
  qb.parsedFrom('marklogic AND cat:c',
    qb.parseBindings(
      qb.parseFunction('ss-cat.xqy', qb.bind('cat'))
  ))
)).result(function (documents) {
  for (const i in documents)
    console.log(JSON.stringify(documents[i].content, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

If you save the script to a file named ss-cat.js and run it, the search returns two contributor documents:

```
$ node ss-cat.js
{
  "Contributor": {
    "userName": "souser1248651@email.com",
    "reputation": 1,
    "displayName": "Nullable",
    "originalId": "1248651",
    "location": "Ogden, UT",
    "aboutMe": "...My current work includes work with MarkLogic
       Application Server (Using XML, Xquery, and Xpath), WPF/C#,
       and Android Development (Using Java)...",
    "id": "sou1248651"
  }
}
{
  "Contributor": {
    "userName": "souser1601813@email.com",
    "reputation": 91,
    "displayName": "grechaw",
```

```
        "originalId": "1601813",
        "location": "Occidental, CA",
        "aboutMe": "XML (XQuery, Java, XML database) software engineer
           at MarkLogic. Hardcore accordion player.",
        "id": "sou1601813"
      }
    }
```

If you remove the “cat:c” term so that the query text is just “marklogic”, the search returns an additional question document.

For more details and examples, see Creating a Custom Constraint in the *Search Developer’s Guide*.

4.4.7 Additional Information

For additional information on creating and using custom constraints, see the following resources:

- The following functions in the Node.js API Reference:
  - `queryBuilder.parsedFrom`
  - `queryBuilder.parseBindings`
  - `queryBuilder.parseFunction`
  - `queryBuilder.binding`
  - any of the query builders that accept a `queryBuilder.BindingParam` argument, such as `queryBuilder.collection`, `queryBuilder.range`, `queryBuilder.scope`, `queryBuilder.value`, and `queryBuilder.word`
- Searching Using String Queries in the *Search Developer’s Guide*
- Creating a Custom Constraint in the *Search Developer’s Guide*
- Constraint Options in the *Search Developer’s Guide*

4.5 Searching with Query By Example

This section covers the following topics related to searching JSON documents using Query By Example (QBE).

- Introduction to QBE
- Creating a QBE with queryBuilder
- Querying XML Content With QBE
- Additional Information

4.5.1 Introduction to QBE

A Query By Example enables rapid prototyping of queries for “documents that look like this” using search criteria that resemble the structure of documents in your database.

For example, if your documents include an `author` property, then the following raw QBE matches documents with an `author` value of “Mark Twain”.

```
{ $query: { author: "Mark Twain" } }
```

Use `queryBuilder.byExample` to construct a QBE with the Node.js Client API. When working with JSON content, this interfaces accepts individual search criteria modeled on the content (`{ author: "Mark Twain" }`) or an entire `$query` object as input. For example:

```
db.documents.query( qb.where(
  qb.byExample( {author: 'Mark Twain'} ))
)
```

When searching XML, you can pass in a serialized XML QBE. For details, see “Querying XML Content With QBE” on page 138.

The subset of the MarkLogic Server Search API exposed by QBE includes value queries, range queries, and word queries. QBE also supports logical and relational operators on values, such as AND, OR, NOT, greater than, less than, and equality tests.

You can only use QBE and the Node.js API to query document content. Metadata search is not supported. Also, you cannot search on fields. To query metadata or search over fields, use the other `queryBuilder` builder functions, such as `queryBuilder.collection`, `queryBuilder.property`, or `queryBuilder.field`. Use a field query to search on the metadataValues metadata category.

This guide provides only a brief introduction to QBE. For details, see Searching Using Query By Example in *Search Developer’s Guide*.

4.5.2 Creating a QBE with queryBuilder

To create a QBE, call `queryBuilder.byExample` and pass in one or more search criteria parameters. When working with XML documents, you can also pass in a fully formed QBE; for details, see “Querying XML Content With QBE” on page 138.

For example, the documents created by “Loading the Example Data” on page 179 include a `location` property. Running the following script against this data enables you to search for all contributors from Oslo, Norway.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const qb = marklogic.queryBuilder;

db.documents.query(
    qb.where(qb.byExample( {location: 'Oslo, Norway'} ))
).result( function(results) {
  console.log(JSON.stringify(results, null, 2));
});
```

The search criteria passed to `qb.byExample` match only those documents that contain a `location` property with a value of 'Oslo, Norway'. A QBE criteria of the form `{propertyName: value}` is a value query, so the value must exactly match 'Oslo, Norway'.

You can construct other query types that model your documents, including word queries and range queries. For example, you can relax the above constraint to be tolerant of variations on the `location` value by using a word query. You can also add a criteria that only matches contributors with a `reputation` value greater than 400. The following table describes the QBE criteria you can use to realize this search:

QBE Criteria	Description
`location: {$word : 'oslo'}`	Match the phrase "oslo" when it appears in the value of `location`. `$word` is a reserved property name that signifies a word query. The use of word query means the match is case insensitive, and the value may or may not include other words. For details, see Word Query in the *Search Developer's Guide*.
`reputation: {$gt : 400}`	Match documents where the value of reputation is greater than 400. `$gt` is a reserved property name that signifies the "greater than" comparison operator. For details, see Range Query in the *Search Developer's Guide*.
`$filtered: true`	Perform a filtered search. QBE uses unfiltered search by default for best performance. However, range queries, such as `{$gt : 400}` require either filtered search or a backing range index, so we must enable filtered search. For details, see How Indexing Affects Your Query in the *Search Developer's Guide*.

The following script combines these criteria into a single QBE:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);
const qb = marklogic.queryBuilder;

db.documents.query( qb.where(
  qb.byExample( {
    location: {$word : 'oslo'},
    reputation: {$gt : 400},
    $filtered: true
  }))
).result( function(results) {
  console.log(JSON.stringify(results, null, 2));
```

```
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

You can pass criteria into `byExample` as individual objects or an array of objects. For example, the following calls are equivalent to the `byExample` call above:

```
// criteria as individual objects
qb.byExample(
  {location: {$word : 'oslo'}},
  {reputation: {$gt : 400}},
  {$filtered: true}
)

// criteria as an array of objects
qb.byExample([
  {location: {$word : 'oslo'}},
  {reputation: {$gt : 400}},
  {$filtered: true}
])
```

The inputs to `queryBuilder.byExample` in these examples correspond to search criteria in the `$query` portion of a raw QBE; for details, see Constructing a QBE with the Node.js QueryBuilder in the *Search Developer's Guide*.

You can also pass the raw `$query` portion of a QBE to `queryBuilder.byExample` by supplying an object that has a `$query` property. For example:

```
// raw QBE $query
qb.byExample(
  { $query: {
      location: {$word : 'oslo'},
      reputation: {$gt : 400},
      $filtered: true
  }}
)
```

4.5.3 Querying XML Content With QBE

Pass JavaScript query criteria to `queryBuilder.byExample`, as described in "Creating a QBE with queryBuilder" on page 136, implicitly creates JSON QBE, which only matches JSON content. By default, a QBE only matches documents with the same content type as the QBE. That is, a QBE expressed in JSON matches JSON documents, and a QBE expressed in XML matches XML documents. You can still search XML content by either using a serialized XML QBE or by setting the `$format` QBE property to 'xml'.

To use a QBE to search XML content, use one of the following techniques:

- Pass a serialized XML QBE as input to `queryBuilder.byExample`. If your query relies on XML namespaces, you must use this technique. For example:

```
qb.byExample(
  '<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">'+
    '<q:query>' +
      '<my:contributor xmlns:my="http://marklogic.com/example">' +
        '<my:location><q:word>oslo</q:word></my:location>' +
      '</my:contributor>' +
      '<my:contributor xmlns:my="http://marklogic.com/example">' +
        '<my:reputation><q:gt>400</q:gt></my:reputation>' +
      '</my:contributor>' +
    '<q:filtered>true</q:filtered>' +
    '</q:query>' +
  '</q:qbe>'
)
```

- Pass a JavaScript object to `queryBuilder.byExample` that represents a fully formed QBE that includes a `$format` property with the value 'xml'. You can only use this technique when working with XML content that is in no namespace. For example:

```
qb.byExample({
  $query: {
    location: {$word : 'oslo'},
    reputation: {$gt : 400},
    $filtered: true
  },
  $format: 'xml'
})
```

In both cases, the data passed in to `queryBuilder.byExample` must be a fully formed QBE (albeit a serialized one, in the XML case), not just the query criteria as when searching JSON documents. For syntax, see Searching Using Query By Example in the *Search Developer's Guide*.

As with any search that matches XML, the XML content returned by the search is serialized and returned as a string.

4.5.4 Additional Information

For additional information on constructing and using QBE, see the following resources:

- `queryBuilder.byExample` in the Node.js API Reference
- Searching Using Query By Example in the *Search Developer's Guide*

4.6 Searching with Structured Queries

The `queryBuilder` functions that return a `queryBuilder.Query` construct sub-queries of a *structured query*. A structured query is an Abstract Syntax Tree representation of a search expression. Use a structured query when the expressiveness of string query or QBE is not sufficient, or when you need to intercept a query and augment or modify it. For details, see Structured Query Overview in the *Search Developer's Guide*.

- Basic Usage
- Example: Using Structured Query
- Builder Methods Taxonomy Reference
- Query Parameter Helper Functions
- Search Result Refiners

4.6.1 Basic Usage

When you pass one or more `queryBuilder.query` objects to a function that creates a `queryBuilder.BuiltQuery`, such as `queryBuilder.where`, the queries are used to build a structured query. A structured query is an Abstract Syntax Tree representation of a search expression. Use a structured query when the expressiveness of string query or QBE is not sufficient, or when you need to intercept a query and augment or modify it. For details, see Structured Query Overview in the *Search Developer's Guide*.

Structured queries are composed of one or more search criteria that you create using the builder methods of `queryBuilder`. For a taxonomy of builders and examples of each, see "Builder Methods Taxonomy Reference" on page 142.

For example, the following code snippet sends your query to MarkLogic Server as a structured query. The query matches documents in the database directory "/contributors/" that also contain the term "marklogic".

```
db.documents.query(
  qb.where(
    qb.and(qb.directory("/contributors/",
           qb.term("marklogic"))
))
```

Use the `queryBuilder` result refinement methods to tailor your results, just as you do when searching with a string query or QBE. For details, see "Search Result Refiners" on page 149.

4.6.2 Example: Using Structured Query

The following example relies on the sample data from "Loading the Example Data" on page 179.

This example demonstrates some of the ways you can use the structured query builders to create complex queries.

The following example finds documents in the `/contributors/` database directory that contain the term “marklogic”. By default, the query returns the matching documents.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const qb = marklogic.queryBuilder;

db.documents.query(
  qb.where(
    qb.and(
      qb.directory('/contributors/'),
      qb.term('marklogic')
    )
  )
).result( function(results) {
  console.log(JSON.stringify(results, null, 2));
});
```

The query returns an array of document descriptors, one for each matching document. The sample data contains 2 documents that match, `/contributors/contrib3.json` and `/contributors/contrib4.json`, so you should see output similar to the following. The `content` property of the document descriptor contains the contents of the matching document.

```
[
  {
    "uri": "/contributors/contrib3.json",
    "category": "content",
    "format": "json",
    "contentType": "application/json",
    "contentLength": "323",
    "content": {
      "Contributor": {
        "userName": "souser1248651@email.com",
        "reputation": 1,
        "displayName": "Nullable",
        "originalId": "1248651",
        "location": "Ogden, UT",
        "aboutMe": "...My current work includes work with MarkLogic
          Application Server (Using XML, Xquery, and Xpath), WPF/C#,
          and Android Development (Using Java)...",
        "id": "sou1248651"
      }
    }
  },
  {
    "uri": "/contributors/contrib4.json",
    "category": "content",
    "format": "json",
    "contentType": "application/json",
    "contentLength": "273",
```

```
    "content": {
      "Contributor": {
        "userName": "souser1601813@email.com",
        "reputation": 91,
        "displayName": "grechaw",
        "originalId": "1601813",
        "location": "Occidental, CA",
        "aboutMe": "XML (XQuery, Java, XML database) software
           engineer at MarkLogic. Hardcore accordion player.",
        "id": "sou1601813"
      }
    }
  }
]
```

You can optionally remove the call to `queryBuilder.and` because `queryBuilder.where` implicitly ANDs together the queries passed to it. For example, you can rewrite the original query as follows and get the same results:

```
db.documents.query(
  qb.where(
    qb.directory('/contributors/'),
    qb.term('marklogic')
  )
```

You can also combine a string query with one or more structured query builder results. For example, you could further limit the results to documents that also contain "java" by adding `qb.parsedFrom('java')` to the query list passed to `qb.where`. The string query is implicitly AND'd with the other query terms. If you change the query to the following, the result set contains only `/contributors/contrib3.json`.

```
db.documents.query(
  qb.where(
    qb.directory('/contributors/'),
    qb.term('marklogic'),
    qb.parsedFrom('java')
  )
```

The queryBuilder interface includes helper functions that make it easy to construct more complex query components, such as index references. For details, see "Query Parameter Helper Functions" on page 147.

As with the other query types, you can refine your result set using `queryBuilder.slice` and `queryBuilder.withOptions`. For details, see "Refining Query Results" on page 165.

4.6.3 Builder Methods Taxonomy Reference

Structured query explicitly exposes all the query types described in "Types of Query" on page 120 through builder methods. This section is a quick reference for locating the builders you need, based on this categorization.

- Basic Content Queries
- Logical Composers
- Location Qualifiers
- Document Selectors

You can use most query types in combination with each other, as indicated by the parameters accepted by the builder functions. For details, see the `queryBuilder` interface in the Node.js API Reference.

The `queryBuilder` interface enables you to build complex structured queries without knowing the underlying structural details of the query. Cross-references into the structured query Syntax Reference in the *Search Developer's Guide* are included here if you require further details about the components of a specific query type.

4.6.3.1 Basic Content Queries

Basic content queries express search criteria about your content, such as "JSON property A contains value B" or "any document containing the phrase 'dog'". These queries function as "leaves" in the structure of a complex, compound query because they never contain sub-queries.

The following table lists the Node.js builder methods that create basic content queries. A link to the corresponding raw JSON structured query type is provided in case you need more detail about a particular aspect of a query. You do not need to construct the raw query; the Node.js API does this for you.

`queryBuilder` Function	Example	Structured Query Sub-Query
`term`	`qb.term('marklogic')`	term-query
`word`	`qb.word('aboutMe','marklogic')`	word-query
`value`	`qb.value('tags','java')`	value-query
`range`	`qb.range('reputation', '>=', 100)`	range-query

`geospatial`	`qb.geospatial(`
	`  qb.geoElement('gElemPoint'),`
	`  qb.latlon(50, 44)`
	`)`
		geo-elem-query
		geo-elem-pair-query
		geo-attr-pair-query
		geo-json-property-query
		geo-json-property-pair-query
		geo-path-query

queryBuilder Function	Example	Structured Query Sub-Query

geospatialRegion	`q.geospatialRegion(`
	`  q.geoPath('/envelope/region'),`
	`  'intersects',`
	`  q.circle(5, q.point(10,20))`
	`)`
		geo-region-path-query

geoElement	`qb.geospatial(`
	`  qb.geoElement('gElemPoint'),`
	`  qb.latlon(50, 44)`
	`)`
		geo-elem-query

geoElementPair	`qb.geospatial(`
	`  qb.geoElementPair(`
	`  'gElemPair',`
	`  'latitude',`
	`  'longitude'),`
	`  qb.latlon(50, 44)`
	`)`
		geo-elem-pair-query

geoAttrPair	`qb.geospatial(`
	`  qb.geoAttributePair(`
	`  'gAttrPair',`
	`  'latitude',`
	`  'longitude'),`
	`  qb.circle(100, 240, 144)`
	`)`
		geo-attr-pair-query

geoProperty	`q.geospatial(`
	`  q.geoProperty('gElemPoint'),`
	`  q.point(34, 88)`
	`)`
		geo-json-property-query

geoPropertyPair	`qb.geospatial(`
	`  qb.geoPropertyPair(`
	`  'gElemPair',`
	`  'latitude',`
	`  'longitude'),`
	`  qb.latlon(12, 5)`
	`)`
		geo-json-property-pair-query

geoPath	`q.geospatial(`
	`  q.geoPath('parent/child'),`
	`  q.latlon(12, 5)`
	`)`
		geo-path-query

4.6.3.2 Logical Composers

Logical composers are queries that join one or more sub-queries into a logical expression. For example, “documents which match both query1 and query2” or “documents which match either query1 or query2 or query3”.

The following table lists the Node.js builder methods for logical composers. A link to the corresponding raw JSON structured query type is provided in case you need more detail about a particular aspect of a query. You do not need to construct the raw query; the Node.js API does this for you.

`queryBuilder` Function	Example	Structured Query Sub-Query

`and`	`qb.and(`
	`  qb.word('text','marklogic'),`
	`  qb.value('tags', 'java')`
	`)`
		and-query

`andNot`	`qb.andNot(`
	`  qb.word('text','marklogic'),`
	`  qb.value('tags', 'java')`
	`)`
		and-not-query

`boost`	`qb.boost(`
	`  qb.word('text','marklogic'),`
	`  qb.word('title', 'json')`
	`)`
		boost-query

`not`	`qb.not(qb.term('marklogic'))`	not-query

`notIn`	`qb.notIn(`
	`  qb.word('text','json'),`
	`  qb.word('text', 'json documents')`
	`)`
		not-in-query

`or`	`qb.or(`
	`  qb.value('tags','marklogic'),`
	`  qb.value('tags', 'nosql')`
	`)`
		or-query

4.6.3.3 Location Qualifiers

Location qualifiers are queries that limit results based on where sub-query matches occur, such as only in content, only in metadata, or only when contained a specified JSON property or XML element. For example, “matches for this sub-query that occur in metadata” or “matches for this sub-query that are contained in JSON Property P”.

The following table lists the Node.js builder methods that create location qualifiers. A link to the corresponding raw JSON structured query type is provided in case you need more detail about a particular aspect of a query. You do not need to construct the raw query; the Node.js API does this for you.

`queryBuilder` Function	Example	Structured Query Sub-Query

`documentFragment`	`qb.documentFragment(`
	`  qb.term('marklogic')`
	`)`
		document-fragment-query

`locksFragment`	`qb.locksFragment(`
	`  qb.term('marklogic')`
	`)`
		locks-fragment-query

`near`	`qb.near(`
	`  qb.term('marklogic'),`
	`  qb.term('xquery'), 5`
	`)`
		near-query

`propertiesFragment`	`qb.propertiesFragment(`
	`  qb.term('marklogic')`
	`)`
		properties-fragment-query

`scope`	`qb.scope(`
	`  'aboutMe',`
	`  qb.term('marklogic')`
	`)`
		container-query

4.6.3.4 Document Selectors

Document selectors are queries that match a group of documents by database attributes such as collection membership, directory, or URI, rather than by contents. For example, “all documents in collections A and B” or “all documents in directory D”.

The following table lists the Node.js builder methods that create document selectors. A link to the corresponding raw JSON structured query type is provided in case you need more detail about a particular aspect of a query. You do not need to construct the raw query; the Node.js API does this for you.

`queryBuilder` Function	Example	Structured Query Sub-Query

`collection`	`qb.and(`
	`  qb.collection('marklogicians'),`
	`  qb.term('java')`
	`)`
		collection-query

`directory`	`qb.and(`
	`  qb.directory('/contributors/'),`
	`  qb.term('java')`
	`)`
		directory-query

`document`	`qb.and(`
	`  qb.document(`
	`  '/contributors/contrib1.json',`
	`  '/contributors/contrib3.json'),`
	`  qb.term('norway')`
	`)`
		document-query

4.6.4 Query Parameter Helper Functions

The `queryBuilder` interface includes helper functions for building sub-query parameters that are structurally non-trivial.

For example, a container query (`queryBuilder.scope`) requires a descriptor that idenfities the container (or scope), such as a JSON property or an XML element. The helper functions `queryBuilder.property` and `queryBuilder.element` enable you to define the container descriptor required by the `scope` function.

The following code snippet constructs a container query that matches the term “marklogic” when it occurs in a JSON property named “aboutMe”. The helper function `queryBuilder.property` builds the JSON property name specification.

```
db.documents.query(
  qb.where(
    qb.scope(qb.property('aboutMe'), qb.term('marklogic'))
```

```
        )
    )
```

Key helper functions provided by `queryBuilder` are listed below. For details, see the Node.js API Reference and the *Search Developer's Guide*.

Helper Function	Purpose
`anchor`	Defines a numeric or dateTime range for the `bucket` helper function. For details, see Constrained Searches and Faceted Navigation in the *Search Developer's Guide*.
`attribute`	Identifies an XML element attribute for use with query builders such as `range`, `word`, `value`, and geospatial query builders.
`bucket`	Defines a numeric or dateTime range bucket for use with the `facet` builder. For details, see Constrained Searches and Faceted Navigation in the *Search Developer's Guide*.
`datatype`	Specifies an index type (int, string, etc.) that can be used with the `range` query builder to disambiguate an index reference. You should only need this if you have multiple indexes of differen types over the same document component.
`element`	Identifies an XML element for use with query builders such as `scope`, `range`, `word`, `value`, and geospatial query builders.
`facet`	Defines a search facet for use with `calculate` result builder. For details, see Constrained Searches and Faceted Navigation in the *Search Developer's Guide*.
`facetOptions`	Specifies additional options for use with the `facet` builder. For details, see Facet Options in the *Search Developer's Guide*.
`field`	Identifies a document or `metadataValues` field for use with the `range`, `word`, and `value` query builders. For details, see Fields Database Settings in the *Administrator's Guide*.
`fragmentScope`	Restrict the scope of a `range`, `scope`, `value`, or `word` query to document content or document properties.
`pathIndex`	Identifies a path range index for query builders such as `range` or `geoPath`. The database configuration must include a corresponding path range index. For details, see Understanding Path Range Indexes in the *Administrator's Guide*. The path expression is limited to a subset of XPath; for details, see Path Field and Path-Based Range Index Configuration in the *XQuery and XSLT Reference Guide*.

Helper Function	Purpose
`property`	Identifies a JSON property name for query builders such as `range`, `scope`, `value`, `word`, `geoProperty`, and `geoPropertyPair`.
`qname`	Identifies an XML element QName (local name and namespace URI) for query builders such as `range`, `scope`, `value`, `word`, `geoElement`, and `geoElementPair`, `geoAttributePair`. Also used in constructing an attribute identifier.
`rangeOptions`	Additional search options available with the `range` query builder. For details, see the Node.js API Reference and Range Options in the *Search Developer's Guide*.
`score`	Specifies a range query relevance scoring algorithm for use with the `orderBy` results builder. For details, see Including a Range or Geospatial Query in Scoring in the *Search Developer's Guide*.
`sort`	Specifies the equivalent of a `sort-order` query option that defines the search result sorting critieria and order for use with the `orderBy` results builder. For details, see sort-order in the *Search Developer's Guide*.
`termOptions`	Specifies the equivalent of a term-option query option for use with the word and value query builders. For details, see Term Options in the *Search Developer's Guide*.
`weight`	Specifies a modified weight to assign to a query. Usable with query builders such as `word` and `value`. For details, see Using Weights to Influence Scores in the *Search Developer's Guide*.

4.6.5 Search Result Refiners

The `queryBuilder` interface includes several functions that enable you to refine the results of a search. For example, you can specify how many results to return, how to sort the results, and whether or not to include search facets.

These refinement functions usually return a `queryBuilder.BuiltQuery` object, in contrast to query builders, which usually return a `queryBuilder.Query` object.

You can chain result modifier calls together. For example:

```
db.documents.query(qb.where(someQuery).slice(0,5).orderBy(...))
```

For details, see "Refining Query Results" on page 165.

The table below summarizes the result modifier functions supported by `queryBuilder`. For details, see Node.js API Reference.

Helper Function	Purpose
`anchor`	Defines a numeric or dateTime range for the `bucket` helper function. For details, see "Generating Search Facets" on page 161 and Constrained Searches and Faceted Navigation in the *Search Developer's Guide*.
`bucket`	Defines a numeric or dateTime range bucket for use with the `facet` builder. For details, see "Generating Search Facets" on page 161 and Constrained Searches and Faceted Navigation in the *Search Developer's Guide*.
`facet`	Defines a search facet for use with `calculate` result builder. For details, see "Generating Search Facets" on page 161 and Constrained Searches and Faceted Navigation in the *Search Developer's Guide*.
`facetOptions`	Specifies additional options for use with the `facet` builder. For details, see "Generating Search Facets" on page 161 and Facet Options in the *Search Developer's Guide*.
`calculate`	Builds a search facet specification. For details, see "Generating Search Facets" on page 161.
`orderBy`	Specifies sort order and sequencing. For example, you can specify a JSON property, XML element, XML element attribute on which to sort. For details, see sort-order in the *Search Developer's Guide*.
`slice`	Defines the slice of documents that should be returned from within the result set and any server-side transformation that should be applied to the results. For details, see "Refining Query Results" on page 165.
`withOptions`	Miscellanious options that can be used to refine and tune you query. For example, use `withOptions` to specify the categories of data to retrieve from the matching documents, such as content or metadata, request query metrics, or specify a transaction id.

4.7 Searching with Combined Query

A combined query is a query object that can contain a combination of different query types plus query options. Most searches can be accomplished without using a combined query. For example, you can combine a string query and a structured query by simply passing the results of `queryBuilder.parsedFrom` and a `queryBuilder.Query` to `queryBuilder.where`.

This feature is best suited for advanced users who are already familiar with the Search API and who have one of the following requirements:

- Your application must use query options previously persisted on MarkLogic Server.
- You require very fine-grained control over query options at query time. (Most query options are already exposed in other parts of the Node.js API, such as the `queryBuilder` methods. You should use those interfaces when possible, rather than relying on combined query.)

In the Node.js Client API, CombinedQueryDefinition encapsulates a combined query. The API provides no builder for `CombinedQueryDefinition`. A `CombinedQueryDefinition` has the following form, where the `search` property contains the combined query, and the remaining properties can optionally be used to customize the results.

```
{ search: {
    query: { structuredQuery },
    qtext: stringQuery,
    options: { queryOptions }
  },
  categories: [ resultCategories ],
  optionsName: persistedOptionsName,
  pageStart: number,
  pageLength: number,
  view: results
}
```

The combined query portion can contain any combination of a structured query, a string query, and Search API query options. If you specify options inside the combined query that conflict with options implied by the settings in the `CombinedQueryDefinition` wrapper, the wrapper option settings override the ones inside the combined query. For example, if `search.options` includes `'page-length':5` and `search.pageLength` is set to 10, then the page length will be 10.

The following table describes the properties of a combined query:

Property Name	Description
query	Optional. A structured query conforming to the syntax described in Searching Using Structured Queries in the *Search Developer's Guide*.
qtext	Optional. A string query conforming to the Search API string query syntax. For details, see "Searching with String Queries" on page 125 and Searching Using String Queries in the *Search Developer's Guide*.
options	Optional. One or more Search API query options. For details, see Appendix: Query Options Reference in the *Search Developer's Guide*.

Use the `categories`, `pageStart`, `pageLength`, and `view` properties to customize your search results, as described in “Refining Query Results” on page 165.

Use the `optionsName` property to name a set of previously persisted query options to apply to the search. If the `CombinedQueryDefinition` contains both options in the combined query and a persistent query options name, then the two sets of options are merged together. Where equivalent options occur in both, the settings in the combined query takes precedence.

> **Note:** You cannot use the Node.js Client API to persist query options. Instead, use the REST or Java Client APIs to do so. For details, see Configuring Query Options in the *REST Application Developer’s Guide* or Query Options in the *Java Application Developer’s Guide*.

The following example uses a `CombinedQueryDefinition` to find documents containing “java” and “marklogic” that are in the database directory “/contributors”. The combined query sets the `return-query` option to true to include the final query structure in the results. The `categories` property is set to “none” so that the search result summary is returned instead of the matching documents; the summary will contain the final query. Results are returned 3 at a time, due to the `pageLength` setting.

```
db.documents.query({
  search: {
    qtext: 'java',
    query: {
      'directory-query' : { uri: '/contributors/' },
      'term-query': { text: ['marklogic'] }
    },
    options: {
      'return-query': true
    }
  },
  categories: [ 'none' ],
  pageLength: 3
})
```

4.8 Searching Values Metadata Fields

Values metadata, sometimes called key-value metadata, can only be searched if you define a metadata field on the keys you want to search. Once you define a field on a metadata key, use the normal field search capabilities to include a metadata field in your search. For example, you can use `queryBuilder.field` and `queryBuilder.word` to create a word query on a metadata field.

For more details, see Metadata Fields in the *Administrator’s Guide*.

4.9 Querying Lexicons and Range Indexes

The Node.js Client API enables you to search and analyze lexicons and range indexes in the following ways:

- Query the values in a single lexicon or range index.
- Find co-occurrences of values in multiple range indexes.
- Analyze range index or lexicon values or value co-occurrences using builtin or user-defined aggregate functions. For details, see "Analyzing Lexicons and Range Indexes with Aggregate Functions" on page 159.

This section covers the following related topics:

- Querying Values in a Lexicon or Range Index
- Finding Value Co-Occurrences in Lexicons
- Building an Index Reference
- Refining the Results of a Values or Co-Occurrence Query
- Analyzing Lexicons and Range Indexes with Aggregate Functions

For related search concepts, see Browsing With Lexicons in the *Search Developer's Guide* and Text Indexes in the *Administrator's Guide*.

4.9.1 Querying Values in a Lexicon or Range Index

Use the `marklogic.valueBuilder` interface to build queries against lexicons and range indexes, then use `DatabaseClient.values.read` to apply your query.

For example, if the database is configured to include a range index on the "reputation" JSON property or XML element, then the following query returns all the values in range index:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const vb = marklogic.valuesBuilder;

db.values.read(
  vb.fromIndexes('reputation')
).result(function (result) {
  console.log(JSON.stringify(result, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

If you save the script to a file and run against the data from "Loading the Example Data" on page 179, you should see results similar to the following. The query returns a `values-response.tuple` item for each distinct value.

```
{ "values-response": {
    "name": "structuredef",
    "types": {
      "type": [ "xs:int" ]
```

```
    },
    "tuple": [
      {
        "frequency": 1,
        "distinct-value": [ "1" ]
      },
      {
        "frequency": 1,
        "distinct-value": [ "91" ]
      },
      {
        "frequency": 1,
        "distinct-value": [ "272" ]
      },
      {
        "frequency": 1,
        "distinct-value": [ "446" ]
      }
    ],
    "metrics": {
      "values-resolution-time": "PT0.000146S",
      "total-time": "PT0.000822S"
    }
  }
}
```

You can use `values.slice` to retrieve a subset of the values. For example, if you modify the above script to so that the query looks like the following, then the query returns 2 values, beginning with the 3rd value:

```
db.values.read(
  vb.fromIndexes('reputation')
    .slice(2,4)
)

==>
{ "values-response": {
    "name": "structuredef",
    "types": {
      "type": [ "xs:int" ]
    },
    "tuple": [
      {
        "frequency": 1,
        "distinct-value": [ "272" ]
      },
      {
        "frequency": 1,
        "distinct-value": [ "446" ]
      }
    ],
    "metrics": {
      "values-resolution-time": "PT0.000174S",
```

```
      "total-time": "PT0.000867S"
    }
  }
}
```

4.9.2 Finding Value Co-Occurrences in Lexicons

A co-occurrence is a set of index or lexicon values occurring in the same document fragment. The Node.js Client API supports queries for *n*-way co-occurrences. That is, tuples of values from multiple lexicons or indexes, occurring in the same fragment.

To find values co-occurrences across multiple range indexes or lexicons, use the `marklogic.valueBuilder` interface to construct a query, then apply it using `DatabaseClient.values.read`. When a values query includes multiple index references, the results are co-occurrence tuples.

For example, the following script find co-occurrences of values in the “tags” and “id” JSON properties or XML elements, assuming the database configuration includes an element range index for “tags” and another for “id”. (Recall that range indexes on JSON properties use the element range index interfaces; for details, see “Indexing” on page 122.)

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const vb = marklogic.valuesBuilder;

db.values.read(
  vb.fromIndexes('tags','id')
).result(function (result) {
  console.log(JSON.stringify(result, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

If you save the script to a file and run it, you should see results similar to the following. The query returns a `values-response.tuple` item for each co-occurrence. The property `values-response.types` can guide you in interpreting the data types of the values in each tuple.

```
{
  "values-response": {
    "name": "structuredef",
    "types": {
      "type": [
        "xs:string",
        "xs:string"
      ]
    },
    "tuple": [
      {
        "frequency": 1,
```

```
            "distinct-value": [
              "dbobject",
              "soq7684223"
            ]
          },
          {
            "frequency": 1,
            "distinct-value": [
              "dbobject",
              "sou69803"
            ]
          },...
        ],
        "metrics": {
          "values-resolution-time": "PT0.000472S",
          "total-time": "PT0.001251S"
        }
      }
    }
```

You can use `values.slice` to retrieve a subset of the values. For example, if you modify the script to so that the query looks like the following, then the query returns two tuples, beginning with the 3rd value:

```
db.values.read(
  vb.fromIndexes('tags','id').slice(2,4)
)

==>
{
  "values-response": {
    "name": "structuredef",
    "types": {
      "type": [
        "xs:string",
        "xs:string"
      ]
    },
    "tuple": [
      {
        "frequency": 1,
        "distinct-value": [
          "java",
          "soq22431350"
        ]
      },
      {
        "frequency": 1,
        "distinct-value": [
          "java",
          "soq7684223"
        ]
      }
```

```
    ],
    "metrics": {
      "values-resolution-time": "PT0.00024S",
      "total-time": "PT0.001018S"
    }
  }
}
```

4.9.3 Building an Index Reference

Use `valuesBuilder.fromIndexes` to create index references for use in your values and co-occurrence queries. For example, a query such as the following includes a reference by name to an index on a JSON property or XML element named “reputation”:

```
db.values.read(vb.fromIndexes('reputation'))
```

You can use an index reference builder method to disambiguate the index reference, use another type of index, or specify a collation. The following interpretation is applied to the inputs to `valuesBuilder.fromIndexes`:

- A simple name identifies a range index on a JSON property. For example, `vb.fromIndexes('reputation')` identifies a range index for the JSON property `reputation`.
- An index reference identifies a range index. For example, `vb.fromIndexes(vb.field('questionId'))` identifies a field range index.
- If you do not explicitly specify the data type of the range index, the API will attempt to look it up server-side during index resolution. Use `valuesBuilder.datatype` to explicitly specify the data type.

For example, all of the following index references identify a JSON property range index for the property named reputation.

```
vb.fromIndexes('reputation')

vb.fromIndexes(vb.range('reputation'))

vb.fromIndexes(vb.range(vb.property('reputation')))

vb.fromIndexes(vb.range(
  vb.property('reputation'), vb.datatype('int')))
```

The following table summarizes the index definition builder methods exposed by `valuesBuilder`:

Lexicon or Index Type	valuesBuilder builder method
uri	`vb.uri`
collection	`vb.collection` (with no arguments)
range	*`name`* `vb.range`
field	`vb.field`

geospatial	`vb.geoAttributePair`
	`vb.geoElement`
	`vb.geoElementPair`
	`vb.geoPath`
	`vb.geoProperty`
	`vb.geoPropertyPair`

The URI and collection lexicons must be enabled on the database in order to use them. For details, see Text Indexes in the *Administrator's Guide*. Use `valuesBuilder.uri` and `valuesBuilder.collection` (with no arguments) to identify these lexicons. For example:

```
db.values.read(
  vb.fromIndexes(
    vb.uri(),           // the URI lexicon
    vb.collection())    // the collection lexicon
```

4.9.4 Refining the Results of a Values or Co-Occurrence Query

You can refine the results of your queries in the following ways:

- Use `valuesBuilder.slice` to select a subset of the results and/or specify a result transform.
- Use `valuesBuilder.BuiltQuery.withOptions` to specify values query options or constrain results to particular forests. For a list of options, see the API documentation for `cts.values` (JavaScript) or `cts:values` (XQuery).
- Use `valuesBuilder.BuiltQuery.where` to limit results to those that match another query.

You can use these refinements singly or in any combination.

For example, the following query returns values from the range index on the JSON property reputation. The `where` clause selects only those values in documents in the collection "myInterestingCollection". The `slice` clause selects two results, beginning with the third value. The `withOptions` clause specifies the results be returned in descending order.

```
db.values.read(
  vb.fromIndexes('reputation').
  where(vb.collection('myInterestingCollection')).
  slice(2,4).
  withOptions({values: ['descending']})
```

4.9.5 Analyzing Lexicons and Range Indexes with Aggregate Functions

You can compute aggregate values over range indexes and lexicons using builtin or user-defined aggregate functions with valuesBuilder.BuiltQuery.aggregates. This section covers the following topics:

- Aggregate Function Overview
- Using Builtin Aggregate Functions
- Using User-Defined Aggregate Functions

4.9.5.1 Aggregate Function Overview

An aggregate function performs an operation over values or value co-occurrences in lexicons and range indexes. For example, you can use an aggregate function to compute the sum of values in a range index.

Use `valuesBuilder.BuiltQuery.aggregates` to apply one or more builtin or user-defined aggregate functions to your values or co-occurrences query. You can combine builtin and user-defined aggregates in the same query.

MarkLogic Server provides builtin aggregate functions for several common analytical functions; for a list of functions, see the Node.js API Reference. For a more detailed description of each builtin, see Using Builtin Aggregate Functions in the *Search Developer's Guide*.

You can also implement aggregate user-defined functions (UDFs) in C++ and deploy them as native plugins. Aggregate UDFs must be installed before you can use them. For details, see Implementing an Aggregate User-Defined Function in the *Application Developer's Guide*. You must install the native plugin that implements your UDF according to the instructions in Using Native Plugins in the *Application Developer's Guide*.

> **Note:** You cannot use the Node.js Client API to apply aggregate UDFs that require additional parameters.

4.9.5.2 Using Builtin Aggregate Functions

To use a builtin aggregate function, pass the name of the function to `valuesBuilder.BuiltQuery.aggregates`. For a list of supported builtin aggregate function names, see the Node.js API Reference.

For example, the following script uses builtin aggregates to calculate the minimum, maximum, and standard deviation of the values in the range index over the JSON property named `reputation`. Use a slice clause of the form `slice(0,0)` to return just the computed aggregates, rather than the aggregates plus values.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const vb = marklogic.valuesBuilder;

db.values.read(
  vb.fromIndexes('reputation')
    .aggregates('min', 'max', 'stddev')
    .slice(0,0)
).result(function (result) {
  console.log(JSON.stringify(result, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

Running the script produces output similar to the following:

```
{ "values-response": {
  "name": "structuredef",
  "aggregate-result": [
    { "name": "min", "_value": "1" },
    { "name": "max", "_value": "446" },
    { "name": "stddev", "_value": "197.616632228498" }
  ],
  "metrics": {
    "aggregate-resolution-time": "PT0.000571S",
    "total-time": "PT0.001279S"
  }
} }
```

4.9.5.3 Using User-Defined Aggregate Functions

An aggregate UDF is identified by the function name and a relative path to the plugin that implements the aggregate, as described in Using Aggregate User-Defined Functions in the *Search Developer's Guide*. You must install your UDF plugin on MarkLogic Server before you can use it in a query. For details on creating and installing aggregate UDFs, see Aggregate User-Defined Functions in the *Application Developer's Guide*.

Once you install your plugin, use `valuesBuilder.udf` to create a reference to your UDF, and pass the reference to `valuesBuilder.builtQuery.aggregates`. For example, the following script uses a native UDF called "count" provided by a plugin installed in the Extensions database under "native/sampleplugin":

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
```

```
const db = marklogic.createDatabaseClient(my.connInfo);
const vb = marklogic.valuesBuilder;

//console.log(vb.fromIndexes(vb.range(vb.pathIndex('/id'))));
db.values.read(
  vb.fromIndexes('reputation')
    .aggregates(vb.udf('native/sampleplugin', 'count')
    .slice(0,0)
).result(function (result) {
  console.log(JSON.stringify(result, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

4.10 Generating Search Facets

You can use the Node.js Client API to include facets in your query results, as described in Constrained Searches and Faceted Navigation in the *Search Developer's Guide*. You define facets using `queryBuilder.facet` and include them in your search using `queryBuilder.calculate`. You can construct facets on JSON properties, XML elements and attributes, fields and paths. A facet must be backed by a range index.

This section includes the following topics:

- Defining a Simple Facet
- Naming a Facet
- Including Facet Options
- Defining Bucket Ranges
- Creating and Using Custom Constraint Facets

For more details, see Constrained Searches and Faceted Navigation in the *Search Developer's Guide*.

4.10.1 Defining a Simple Facet

The following example facets on the reputation JSON property of documents in the database directory "/contributors/". The results include only the facets, rather than the facets plus matching documents, because of the `withOptions` clause; for details, see "Excluding Document Descriptors or Values From Search Results" on page 167.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const qb = marklogic.queryBuilder;

db.documents.query(
  qb.where(qb.directory('/contributors/'))
    .calculate(qb.facet('reputation'))
```

```
    .withOptions({categories: 'none'})
).result( function(results) {
  console.log(JSON.stringify(results, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

If the database includes a range index on “reputation”, and you run the script against the example data from “Loading the Example Data” on page 179, you should see results similar to the following:

```
{ "snippet-format": "empty-snippet",
  "total": 4,
  "start": 1,
  "page-length": 0,
  "results": [],
  "facets": {
    "reputation": {
      "type": "xs:int",
      "facetValues": [
        { "name": "1",
          "count": 1,
          "value": 1 },
        { "name": "91",
          "count": 1,
          "value": 91 },
        { "name": "272",
          "count": 1,
          "value": 272 },
        { "name": "446",
          "count": 1,
          "value": 446 }
      ]
    }
  }
}
```

By default, the facet uses the same name as entity from which the facet is derived, such as an XML element or JSON property, but you can provide a custom name. For details, see “Naming a Facet” on page 163.

The `facets` property of the results includes a set of value buckets for the “reputation” facet, one bucket for each distinct value of `reputation`. Each bucket includes a name (auto-generated from the value by default), the number of matches with that value, and the actual value.

```
"facets": {
    "reputation": {       <-- name of the facet
      "type": "xs:int",
      "facetValues": [
        { "name": "1",    <-- bucket name
          "count": 1,     <-- number of matches with this value
          "value": 1 }    <-- value associated with this bucket
```

4.10.2 Naming a Facet

By default, the name of a facet is derived from the indexed element or property name on which the facet is based. For example, the following facet on the “reputation” property generates a facet with the property name “reputation”:

```
qb.facet('reputation')
==> "facets": { "reputation": {...} }
```

You can override this behavior by passing your own name in as the first argument to `queryBuilder.facet`. For example, the following facet on the “reputation” property generates a facet with the property name “rep”:

```
qb.facet('rep', 'reputation')
==> "facets": { "rep": {... } }
```

4.10.3 Including Facet Options

You can use `queryBuilder.facetOptions` to include options in your facet definition that affect attributes such as sort order and the maximum number of values to return. For details, see Facet Options in the *Search Developer’s Guide* and the detailed API documentation for the query that corresponds to your facet index type, such as `cts.values` (JavaScript) or `cts:values` (XQuery).

For example, the following facet definition requests buckets be ordered by descending values and limits the number of buckets to two. Thus, instead of returning buckets ordered [1, 91, 272, 446], the results are ordered [446, 272, 91, 1] and truncated to the first 2 buckets:

```
qb.facet('rep','reputation', qb.facetOptions('limit=2','descending')))

==>

"facets": {
  "reputation": {
    "type": "xs:int",
    "facetValues": [
      { "name": "446",
        "count": 1,
        "value": 446 },
      { "name": "272",
        "count": 1,
        "value": 272 }
    ]
  }
}
```

4.10.4 Defining Bucket Ranges

By default, a facet is bucketed by distinct values. However, you can define your own buckets on numeric and date values using `queryBuilder.bucket`. A bucket can take on a range of values. The upper and lower bounds of the range of values in a bucket are the bucket *anchors*. You can include both anchor values, or omit the upper or lower anchor.

Buckets over `dateTime` values can use symbolic anchors such as “now” and “start-of-day”. The real values are computed when the query is evaluated. Such definitions describe *computed buckets*. For a list of the supported values, see computed-bucket in the *Search Developer's Guide*.

For example, you can divide the reputation values into buckets of “less than 50”, “50 to 100”, and “greater than 100” using a facet definition such as the following:

```
qb.facet('reputation',
         qb.bucket('less than 50', '<', 50),
         qb.bucket('50 to 100', 50, '<', 101),
         qb.bucket('greater than 100', 101, '<'))
==>

"facets": {
  "reputation": {
    "type": "bucketed",
    "facetValues": [
      { "name": "less than 50",
        "count": 1,
        "value": "less than 50"
      },
      { "name": "50 to 100",
        "count": 1,
        "value": "50 to 100"
      },
      { "name": "greater than 100",
        "count": 2,
        "value": "greater than 100"
      }
    ]
  }
}
```

In the above example, '<' is a constant that serves as a boundary between the upper and lower anchor values. It is not a relational operator, per se. The separator enables the API to handle buckets with no lower bound, with no upper bound, and with both an upper and a lower bound.

For more examples of defining buckets, see Buckets Example in the *Search Developer's Guide* and Computed Buckets Example in the *Search Developer's Guide*.

4.10.5 Creating and Using Custom Constraint Facets

When you define a custom constraint, you can also define facet generators for your constraint, as described in Creating a Custom Constraint in the *Search Developer's Guide*. Use the following procedure to use a custom constraint facet generator.

1. Implement an XQuery module that includes start-facet and finish-facet functions. For details, see Creating a Custom Constraint in the *Search Developer's Guide*.

2. Install your custom constraint module in the modules database associated with your REST API instance using the `DatabaseClient.config.query.custom interface`, as described in “Installing the Constraint Parser” on page 133.

3. Use `queryBilder.CalculateFunction` to create a reference to your facet generator when building your facet definitions.

For example, if your custom constraint module is installed as `ss-cat.xqy`, as shown in “Installing the Constraint Parser” on page 133:

```
db.config.query.write('ss-cat.xqy', ...)
```

Then you can use your facet generator in your facet definitions as follows:

```
qb.facet('categories', qb.calculateFunction('ss-cat.xqy'))
```

4.11 Refining Query Results

This section covers the following features of the Node.js Client API that enable you to customize your search results using `queryBuilder.slice` or `valuesBuilder.BuiltQuery.slice`:

- Available Refinements
- Paginating Query Results
- Returning Metadata
- Excluding Document Descriptors or Values From Search Results
- Generating Search Snippets
- Transforming the Search Results
- Extracting a Portion of Each Matching Document

4.11.1 Available Refinements

By default, when you perform a search using `DatabaseClient.documents.query`, you receive one “page” of matching document descriptors ordered by relevance ranking. Each descriptor includes the content of the matching document.

Some query options always cause a search result summary to be returned, in addition to the matching document descriptors. For example, when you enable options such as `'debug'`, `'metrics'`, or `'queryPlan'`, the additional data requested by the option is returned as part of the search result summary.

The Node.js Client API provides several result refinement `queryBuilder` methods that enable you to customize your results, including the following:

- Change the size and/or starting document for the "page" using `queryBuilder.slice`. For details, see "Paginating Query Results" on page 166
- Change the order of results using `queryBuilder.orderBy`. For details, see the *Node.js Client API Reference*.
- Request metadata in addition to or instead of content using `queryBuilder.withOptions`. For details, see "Returning Metadata" on page 167.
- Exclude the document descriptors from the response using `queryBuilder.withOptions`. This is useful when you just want to fetch snippets, facets, metrics, or other data about the matches. For details, see "Excluding Document Descriptors or Values From Search Results" on page 167
- Request search match snippets in addition to or instead of matching documents using `queryBuilder.snippet` with `queryBuilder.slice`. You can customize your snippets. For details, see "Generating Search Snippets" on page 168
- Request search facets in addition to or instead of matching documents, using `queryBuilder.calculate`. You can customize your facet buckets. For details, see "Generating Search Facets" on page 161.
- Apply a read transform to the matched documents or search results summary. For details, see "Transforming the Search Results" on page 169.

The slice specifies the range of matching documents to include in the result set. If you do not explicitly call `queryBuilder.slice`, a default slice is still defined. The other refinement methods (`calculate`, `orderBy`, `snippet`, `withOptions`) have no effect if the slice is empty, whether the slice is empty because there are no matches for your query or because you defined an empty page range such as `slice(0,0)`.

You can use these features in combination. For example, you can request snippets and facets together, without or without document descriptors.

4.11.2 Paginating Query Results

Use `queryBuilder.slice` and `valuesBuilder.BuiltQuery.slice` to fetch slice (batch) of results. A slice of results is defined by a zero-based starting position and an end position (the value in the end position is not included in the slice), similar to using `Array.prototype.slice`.

For example, the following queries return five results, beginning with the first one:

```
qb.where(qb.parsedFrom('oslo')).slice(0,5)

vb.fromIndexes('reputation').slice(0,5)
```

To return the next 5 results, you would use queries such as the following

```
qb.where(qb.parsedFrom('oslo')).slice(5,10)

vb.fromIndexes('reputation').slice(5,10)
```

The default maximum number of results is 10.

Setting the starting and end positions to zero selects no matches (or values), but returns an abbreviated result summary that includes, for example, estimated total number of matches for a search or computed aggregates for a values query.

4.11.3 Returning Metadata

By default, a query returns document descriptors for each matching documents, and the descriptors include the document content.To return metadata instead of contents, set the `categories` property of `queryBuilder.withOptions` to `'metadata'`. For example:

```
db.documents.query(
  qb.where(qb.parsedFrom('oslo'))
    .withOptions({categories: 'metadata'})
)
```

To return both metadata and documents, set categories to both `'content'` and `'metadata'`. For example:

```
db.documents.query(
  qb.where(qb.parsedFrom('oslo'))
    .withOptions({categories: ['content', 'metadata']})
)
```

4.11.4 Excluding Document Descriptors or Values From Search Results

By default, a query returns document descriptors for each matching documents, and the descriptors include the document content. If you want to retrieve snippets, facets, or other search result data without the matching documents, set the `categories` property of `queryBuilder.withOptions` to `'none'`.

For example, the following query normally returns the contents of two document descriptors:

```
db.documents.query(
  qb.where(qb.parsedFrom('oslo'))
)
```

If you add the following `withOptions` clause, you receive a search result summary that include search snippets, instead of receiving document descriptors:

```
db.documents.query(
  qb.where(qb.parsedFrom('oslo'))
```

```
        .withOptions({categories: 'none'})
)
```

The contents of the search result summary depend on the other refinements you apply to your query, but will never include the document descriptors.

4.11.5 Generating Search Snippets

A search results page typically shows portions of matching documents with the search matches highlighted, perhaps with some text showing the context of the search matches. These search result pieces are known as *snippets*.

Snippets are not included in your query results by default. To request snippets, include a snippet clause in your slice definition using `queryBuilder.snippet`. For example, the following query returns snippets in the default format:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const qb = marklogic.queryBuilder;

db.documents.query(
    qb.where(
      qb.byExample({aboutMe: {$word: 'marklogic'}})
    ).slice(qb.snippet())
).result( function(results) {
  console.log(JSON.stringify(results, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

You can include a snippet clause in a slice that has a start and end position, as well. For example:

```
slice(0, 5, qb.snippet())
```

To retrieve snippets without the matching documents, add a `withOptions({categories: 'none'})` clause. For exaample:

```
...slice(qb.snippet()).withOptions({categories: 'none'})
```

You can use one of several builtin snippet generators or your own custom snippet generator by providing a name to `queryBuilder.snippet`. For example, the following slice definition requests snippets generated by the builtin `metadata-snippet` generator:

```
slice(0, 5, qb.snippet('metadata')
```

Some of the builtin snippeters accept additional options, which you can specify in the second parameter to `queryBuilder.snippet`. For example, the following snippet definition limits the size of the snippeted text 25 characters:

```
qb.snippet('my-snippeter.xqy', {'max-snippet-chars': 25})
```

For details on the supported options, see the Node.js API Reference and Specifying transform-results Options in the *Search Developer's Guide*.

Use the following procedure to use a custom snippeter:

1. Implement your snippet generator in XQuery. Your `snippet` function must conform to the interface specified in Specifying Your Own Code in transform-results in the *Search Developer's Guide*.

2. Install your snippeting module in the modules database of your REST API instance using `DatabaseClient.config.query.snippet.write`.

3. Use the name of your custom snippeting module as the snippeter name provided to `queryBuilder.snippet`. For example:

   ```
   slice(0, 5, qb.snippet('my-snippeter.xqy'))
   ```

You cannot pass options or parameters to a custom snippeter.

For more information on snippet generation, see Modifying Your Snippet Results in the *Search Developer's Guide*.

4.11.6 Transforming the Search Results

You can make arbitrary changes to the response from a search or values query by applying a transformation function. Your transform is applied to each document returned by the query, as well as to the search or values response summary, if any.

Transforms must be installed on MarkLogic Server before you can use them. Use `DatabaseClient.config.transforms` to install and manage transforms.

To use a transform in a query, create a transform descriptor with `queryBuilder.transform` or `valuesBuilder.transform`. You must specify the name of a previously installed transform function. You can also include implementation-specific parameters. For details and examples, see "Working with Content Transformations" on page 233.

For example, the following query applies the transform named "js-query-transform" to the search results. Since no documents are returned (`withOptions`), the query only returns a search results summary and the transform is only applied to the summary. If the query returned documents, the transform would be applied to each matched document as well.

```
db.documents.query(
  qb.where(
    qb.byExample({writeTimestamp: {'$exists': {}}})
  ).slice(qb.transform('js-query-transform'))
   .withOptions({categories: 'none'})
)
```

You can apply a transform to a values query in the same fashion. For example:

```
db.values.read(
  vb.fromIndexes('reputation')
    .slice(0, 5, vb.transform('js-query-transform'))
```

For details, see “Working with Content Transformations” on page 233.

4.11.7 Extracting a Portion of Each Matching Document

Use `queryBuilder.extract` to return a subset of the content in each matching document instead of the complete document. You can return selected properties, selected properties plus their ancestors, or everything except the selected properties. By default, only the selected properties are included.

Selected properties are specified using XPath expressions. You can only use a subset of XPath for these path expressions. For details, see The extract-document-data Query Option in the *XQuery and XSLT Reference Guide*.

The following example performs the same search as the first query in “Creating a QBE with queryBuilder” on page 136, but refines the results using `queryBuilder.slice` and `queryBuilder.extract` to return just the `displayName` and `location` properties from the matching documents. The search matches two documents when run against the documents created by “Loading the Example Data” on page 179.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const qb = marklogic.queryBuilder;

db.documents.query(
    qb.where(qb.byExample( {location: 'Oslo, Norway'} ))
      .slice(qb.extract({'abc': 'http://marklogic.com/test/abc'}
      selected:’include’,
      paths:['/Contributor/displayName', '/Contributor/location'])
      namespaces:{'abc': 'http://marklogic.com/test/abc'}
      )
).result( function(matches) {
  matches.forEach(function(match) {
    console.log(match.content);
  });
});
```

When you use `queryBuilder.extract` in the manner above, each matching document produces a document descriptor containing content of the following form:

```
{ context: original-document-context,
  extracted: [ obj-from-path1, obj-from-path2, ...] }
```

For example, the above query produces the following output:

```
{ context: 'fn:doc("/contributors/contrib1.json")',
  extracted: [
    { displayName: 'Lars Fosdal' },
    { location: 'Oslo, Norway' } ]
}
{ context: 'fn:doc("/contributors/contrib2.json")',
  extracted: [
    { displayName: 'petrumo' },
    { location: 'Oslo, Norway' } ]
}
```

You can produce a sparse representation of the original matching document instead by passing a `selected` value to `queryBuilder.extract`. You can created a sparse document that includes the selected property(s) plus ancestors, or the whole document exclusive of the selected property(s).

For example, the following query returns the same properties but includes their ancestors:

```
db.documents.query(
    qb.where(qb.byExample( {location: 'Oslo, Norway'} ))
      .slice(qb.extract({
        paths: ['/Contributor/displayName', '/Contributor/location'],
        selected: 'include-with-ancestors',
        namespaces:{'abc': 'http://marklogic.com/test/abc'}
      }))
)
```

The output from this query is the following a sparse version of the original documents:

```
{ Contributor: {
  displayName: 'Lars Fosdal',
  location: 'Oslo, Norway' } }
{ Contributor: {
  displayName: 'petrumo',
  location: 'Oslo, Norway' } }
```

The following table shows the effect of each supported value of the selected parameter of `queryBuilder.extract` on the returned content.

`selected` Value	Output
`include` (default)	`{ context: 'fn:doc("/contributors/contrib1.json")', extracted: [ { displayName: 'Lars Fosdal' }, { location: 'Oslo, Norway' } ] } { context: 'fn:doc("/contributors/contrib2.json")', extracted: [ { displayName: 'petrumo' }, { location: 'Oslo, Norway' } ] }`
`include-with-ancestors`	`{ Contributor: { displayName: 'Lars Fosdal', location: 'Oslo, Norway' } } { Contributor: { displayName: 'petrumo', location: 'Oslo, Norway' } }`
`exclude`	`{ Contributor: { userName: 'souser10002@email.com', reputation: 446, originalId: '10002', aboutMe: 'Software Developer since 1987, ...', id: 'sou10002' } } { Contributor: { userName: 'souser1000634@email.com', reputation: 272, originalId: '1000634', aboutMe: 'Developer at AspiroTV', id: 'sou1000634' } }`
`all`	`{ Contributor: { userName: 'souser10002@email.com', reputation: 446, displayName: 'Lars Fosdal', originalId: '10002', location: 'Oslo, Norway', aboutMe: 'Software Developer since 1987...', id: 'sou10002' } } { Contributor: { userName: 'souser1000634@email.com', reputation: 272, displayName: 'petrumo', originalId: '1000634', location: 'Oslo, Norway', aboutMe: 'Developer at AspiroTV', id: 'sou1000634' } }`

If an extract path does not match any content in a matched document, then the corresponding property is omitted. If no extract paths match, a descriptor for the document is still returned, but it contains an `extracted-none` property instead of an `extracted` property or a sparse document. For example:

```
{ context: 'fn:doc("/contributors/contrib1.json")',
  extracted-none: null
}
```

4.12 Generating Search Term Completion Suggestions

This section describes how to generate search term completion suggestions using the Node.js Client API. The following topics are covered:

- Understanding the Suggestion Interface
- Example: Generating Search Term Suggestions

4.12.1 Understanding the Suggestion Interface

Search applications often offer suggestions for search terms as the user types into the search box. The suggestions are based on terms that are in the database, and are typically used to make the user interface more interactive and to quickly suggest search terms that are appropriate to the application.

Suggestions are drawn from the range indexes and lexicons you specify in your request. For performance reasons, a range or collection index is recommended over a word lexicon; for details, see the Usage Notes for `search:suggest`. Suggestions can be further filtered by additional search criteria.

Use `DatabaseClient.documents.suggest` to generate search term completion suggestions using the Node.js Client API. The simplest suggestion request takes the following form:

```
db.documents.suggest(partialText, qualifyingQuery)
```

Where *partialText* is the query text for which you want to generate suggestions, and *qualifyingQuery* is any additional search criteria, including index and lexicon bindings. Though the qualifying query can be arbitrarily complex, typically at least a portion of it will eventually be "filled in" by the completed phrase.

For example, the following call requests suggestions for the partial phrase "doc". Because the first parameter to `qb.parsedFrom` is an empty string, there are no additional search criteria.

```
db.documents.suggest('doc',
  qb.where(qb.parsedFrom('',
    qb.parseBindings(
      qb.value('prefix', qb.bind('prefix')),
      qb.range('name', qb.bindDefault()))
```

```
  ))
)
```

The parse bindings in the qualifying query include a binding for unqualified terms (`qb.bindDefault()`) to a range query on the JSON property named “name” (`qb.range('name', ...)`). The database must include a matching range index.

Thus, if the database contains documents of the following form, then suggestions for “doc” are drawn only from the values of “name” and never from the values of “prefix” or “alias”:

```
{ "prefix": "xdmp",
  "name": "documentLoad",
  "alias": "document-load" }
```

When the user completes the search term, possibly from the suggestions, the empty string can be replaced by the complete phrase in the document query. Thus if the user completes the term as “documentLoad”, the same query can be used as follows to retrieve matching documents:

```
db.documents.query(
  qb.where(qb.parsedFrom('documentLoad',
    qb.parseBindings(
      qb.value('prefix', qb.bind('prefix')),
      qb.range('name', qb.bindDefault()))
  ))
)
```

The qualifying query can include other search criteria. The following example adds the query “prefix:xdmp”. The bindings associated the “prefix” term with a value query on the JSON property named “prefix”. The “prefix:xdmp” term could be a portion of search box text previously entered by the user.

```
db.documents.suggest('doc',
  qb.where(qb.parsedFrom('prefix:xdmp',
    qb.parseBindings(
      qb.value('prefix', qb.bind('prefix')),
      qb.range('name', qb.bindDefault()))
  ))
)
```

In this case, suggestions are drawn from the “name” property as before, but they are limited to values that occur in documents that satisfy the “prefix:xdmp” query. That is, suggestions are drawn from values in documents that meet both these criteria:

- Contain a JSON property named “name” whose value begins with “doc”, AND
- Contain a JSON property named “prefix” with the exact value “xdmp”

The term to be completed can also use explicit bindings. For example, the following call requests suggestions for “aka:doc”, where “aka” is bound to a range index on the JSON property “alias”. Suggestions are only drawn from values of this property.

```
db.documents.suggest('aka:doc',
  qb.where(qb.parsedFrom('',
    qb.parseBindings(
      qb.range('alias', qb.bind('aka')),
      qb.value('prefix', qb.bind('prefix')),
      qb.range('name', qb.bindDefault()))
  ))
)
```

The suggestions returned in this case include the prefix. For example, one suggestion might be "aka:document-load".

The qualifying query can include both string query and structured query components, but usually will include at least one more index or lexicon bindings with which to constrain the suggestions. For example, the following code adds a directory query that limits suggestions to documents in the database directory `/suggest/`.

```
db.documents.suggest('doc',
  qb.where(qb.parsedFrom('',
    qb.parseBindings(
      qb.value('prefix', qb.bind('prefix')),
      qb.range('name', qb.bindDefault()))
  ), qb.directory('/suggest/', true))
)
```

You can override bindings on a per suggest basis without modifying your qualifying query by including an additional `suggestBinding` parameter.

In cases where you're using a previously constructed qualifying query, but you want to add bindings that limit the scope of suggestions for other reasons (such as performance), you can add override bindings using `queryBuilder.suggestBindings`.

For example, the following code overrides the binding for bare terms in the qualifying query with a binding to a range index on the JSON property alias. Thus, if a document includes a name property with value "documentLoad" and an alias property with value "document-load", then the suggestions would include "documentLoad" without the `suggestBindings` specification, but "document-load" with the override.

```
db.documents.suggest('doc',
  qb.where(qb.parsedFrom('',
    qb.parseBindings(
      qb.value('prefix', qb.bind('prefix')),
      qb.range('name', qb.bindDefault()))
  )),
  qb.suggestBindings(qb.range('alias', qb.bindDefault()))
)
```

Overrides are per binding. In the example above, only the default binding for bare terms is overridden. The binding for "prefix" continues to take effect as long as the suggestBindings do not include a binding for "prefix".

4.12.2 Example: Generating Search Term Suggestions

The example in this section illustrates the use cases described in “Understanding the Suggestion Interface” on page 173.

The script first loads the example documents into the database, and then generates suggestions from the them. To run the example, you must add the following range indexes. You can create them using the Admin Interface or the Admin API. For details, see Range Indexes and Lexicons in the *Administrator's Guide*.

- An element range index of type “string” with local name “name”.
- An element range index of type “string” with local name “alias”.

The example covers the following use cases, which are discussed in more detail in “Understanding the Suggestion Interface” on page 173.

- Case 1: Suggestions for “doc”drawn from the `name` property
- Case 2: Suggestions for “doc” drawn from `name` where `prefix` is “xdmp”
- Case 3: Suggestions for “doc” drawn from `name` where `prefix` is “xdmp” and the suggestion is from a document in the `/suggest/` directory.
- Case 4: Suggestions for “aka:doc” where the “aka” prefix causes suggestions to be drawn from the `alias` property.
- Case 5: Suggestions for “doc” drawn from the `alias` property by virtue of a suggest binding override.

The table below summarizes the property values in the example documents for quick reference.

URI	name	prefix	alias
`/suggest/load.json`	`documentLoad`	`xdmp`	`document-load`
`/suggest/insert.json`	`documentInsert`	`xdmp`	`document-insert`
`/suggest/query.json`	`documentQuery`	`cts`	`document-query`
`/suggest/search.json`	`search`	`cts`	`search`
`/elsewhere/delete.json`	`documentDelete`	`xdmp`	`document-delete`

Running the example produces results similar to the following:

```
1: Suggestions for naked term "doc":
["documentDelete","documentInsert","documentLoad","documentQuery"]

2: Suggestions filtered by prefix:xdmp:
```

```
["documentDelete","documentInsert","documentLoad"]

3: Suggestions filtered by prefix:xdmp and dir /suggest/:
["documentInsert","documentLoad"]

4: Suggestions for "aka:doc":
[
  "aka:document-delete",
  "aka:document-insert",
  "aka:document-load",
  "aka:document-query"
]

5: Suggestions with overriding bindings:
["document-delete","document-insert","document-load"]
```

To run the example, copy the following script into a file, modify the database connection information as needed, and execute the script with the `node` command. The script assumes the connection information is contained in a file named my-connection.js, as described in “Using the Examples in This Guide” on page 31.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);
const qb = marklogic.queryBuilder;

// NOTE: This example requires a database configuration
// that includes two element range index:
// - type string, local name name
// - type string, local name alias

// Initialize the database with the sample documents
db.documents.write([
  { uri: '/suggest/load.json',
    contentType: 'application/json',
    content: {
      prefix: 'xdmp',
      name: 'documentLoad',
      alias: 'document-load'
  } },
  { uri: '/suggest/insert.json',
    contentType: 'application/json',
    content: {
      prefix: 'xdmp',
      name: 'documentInsert',
      alias: 'document-insert'
  } },
  { uri: '/suggest/query.json',
    contentType: 'application/json',
    content: {
      prefix: 'cts',
      name: 'documentQuery',
      alias: 'document-query'
```

```
  } },
  { uri: '/suggest/search.json',
    contentType: 'application/json',
    content: {
      prefix: 'cts',
      name: 'search',
      alias: 'search'
  } },
  { uri: '/elsewhere/delete.json',
    contentType: 'application/json',
    content: {
      prefix: 'xdmp',
      name: 'documentDelete',
      alias: 'document-delete'
  } },
]).result().then(function(response) {
  // (1) Get suggestions for a naked term
  return db.documents.suggest('doc',
    qb.where(qb.parsedFrom('',
      qb.parseBindings(
        qb.range('name', qb.bindDefault()))
    ))
  ).result(null, function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
}).then(function(response) {
  console.log('1: Suggestions for naked term "doc":');
  console.log(JSON.stringify(response));

  // (2) Get suggestions for a qualified term
  return db.documents.suggest('doc',
    qb.where( qb.parsedFrom('prefix:xdmp',
      qb.parseBindings(
        qb.value('prefix', qb.bind('prefix')),
        qb.range('name', qb.bindDefault()))
    ))
  ).result(null, function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
}).then(function(response) {
  console.log('\n2: Suggestions filtered by prefix:xdmp:');
  console.log(JSON.stringify(response));

  // (3) Suggestions limited by directory
  return db.documents.suggest('doc',
    qb.where( qb.parsedFrom('prefix:xdmp',
      qb.parseBindings(
        qb.value('prefix', qb.bind('prefix')),
        qb.range('name', qb.bindDefault()))
      ),
      qb.directory('/suggest/', true))
  ).result(null, function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
```

```
}).then(function(response) {
  console.log('\n3: Suggestions filtered by prefix:xdmp and dir
/suggest/:');
  console.log(JSON.stringify(response));

  // (4) Get suggestions for a term with a binding
  return db.documents.suggest('aka:doc',
    qb.where( qb.parsedFrom('',
      qb.parseBindings(
        qb.range('alias', qb.bind('aka')),
        qb.range('name', qb.bindDefault()))
    ))
  ).result(null, function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
}).then(function(response) {
  console.log('\n4: Suggestions for "aka:doc":');
  console.log(JSON.stringify(response, null, 2));

  // (5) Get suggestions using a binding override
  return db.documents.suggest('doc',
    qb.where( qb.parsedFrom('prefix:xdmp',
      qb.parseBindings(
        qb.value('prefix', qb.bind('prefix')),
        qb.range('name', qb.bindDefault()))
    )),
    qb.suggestBindings(
        qb.range('alias', qb.bindDefault()))
  ).result(null, function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
}).then(function(response) {
  console.log('\n5: Suggestions with overriding bindings:');
  console.log(JSON.stringify(response));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

4.13 Loading the Example Data

Several of the examples in this chapter rely on data derived from the MarkLogic Samplestack seed data. Samplestack is an open-source implementation of the MarkLogic Reference Application architecture; for details, see the *Reference Application Architecture Guide*.

To load the data, copy the following script to a file and run it. The script uses the connection data described in“Using the Examples in This Guide” on page 31.

Some of the examples require range indexes.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
```

```
const documents = [
{ uri: '/contributors/contrib1.json', content:
  {"Contributor":{
    "userName":"souser10002@email.com", "reputation":446,
    "displayName":"Lars Fosdal", "originalId":"10002",
    "location":"Oslo, Norway",
    "aboutMe":"Software Developer since 1987, mainly using Delphi.",
    "id":"sou10002"}}},
{ uri: '/contributors/contrib2.json', content:
  {"Contributor":{
    "userName":"souser1000634@email.com", "reputation":272,
    "displayName":"petrumo", "originalId":"1000634",
    "location":"Oslo, Norway",
    "aboutMe":"Developer at AspiroTV",
    "id":"sou1000634"}}},
{ uri: '/contributors/contrib3.json', content:
  {"Contributor":{
    "userName":"souser1248651@email.com", "reputation":1,
    "displayName":"Nullable", "originalId":"1248651",
    "location":"Ogden, UT",
    "aboutMe":"...My current work includes work with MarkLogic
Application Server (Using XML, Xquery, and Xpath), WPF/C#, and Android
Development (Using Java)...",
    "id":"sou1248651"}}},
{ uri: '/contributors/contrib4.json', content:
  {"Contributor":{
    "userName":"souser1601813@email.com", "reputation":91,
    "displayName":"grechaw", "originalId":"1601813",
    "location":"Occidental, CA",
    "aboutMe":"XML (XQuery, Java, XML database) software engineer at
MarkLogic. Hardcore accordion player.",
    "id":"sou1601813"}}},
{ uri: '/test/query/extraDir/doc6.xml',
  collections: ['http://marklogic.com/test/abc'],
  contentType:'application.xml',
  content:
     <container xmlns:abc="http://marklogic.com/test/abc">
     <target>match</target>
     <abc:elem>word</abc:elem>
</container>'},
{ uri: ''/questions/q1.json'', content:
  { "tags": [ "java", "sql", "json", "nosql", "marklogic" ],
    "owner": {
      "userName": "souser1238625@email.com",
      "displayName": "Raevik",
      "id": "sou1238625"
    },
    "id": "soq22431350",
    "accepted": false,
    "text": "I have a MarkLogic DB instance populated with JSON
documents that interest me. I have executed a basic search and have a
SearchHandle that will give me the URIs that matched. Am I required to
now parse through the flattened JSON string looking for my key?",
    "creationDate": "2014-03-16T00:06:06.497",
```

```
    "title": "MarkLogic basic questions on equivalent of SELECT with
Java API"
 }},
{ uri: '/questions/q2.json', content:
  { "tags": [ "java", "dbobject", "mongodb" ],
    "owner": {
      "userName": "souser69803@email.com",
      "displayName": "Ankur",
      "id": "sou69803"
    },
    "id": "soq7684223",
    "accepted": true,
   "text": "MongoDB seems to return BSON/JSON objects.  I thought that
surely you'd be able to retrieve values as Strings, ints etc. which can
then be saved as POJO.  I have a DBObject (instantiated as a
BasicDBObject) as a result of iterating over a list ... (cur.next()).
Is the only way (other than using some sort of persistence framework)
to get the data into a POJO to use a JSON serlialiser/deserialiser?",
    "creationDate": "2011-10-07T07:27:18.097",
    "title": "Convert DBObject to a POJO using MongoDB Java Driver"
  }},
{ uri: '/questions/q3.json', content:
  { "tags": [ "json", "marklogic" ],
    "owner": {
      "userName": "souser1238625@email.com",
      "displayName": "Raevik",
      "id": "sou1238625"
    },
    "id": "soq22412345",
    "accepted": false,
    "text": "Does marklogic manage JSON documents?",
    "creationDate": "2014-02-10T00:13:03.282",
    "title": "JSON document management in MarkLogic"
 }},
];

const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write(documents)
  .result(null, function(error) {
      console.log(JSON.stringify(error));
    });
```

The corresponing query to extract document ‘`/test/query/extraDir/doc6.xml`’ looks as follows:

```
db.documents.query(
  q.where(
    q.word('target','match')
).
  slice(0, 1, q.extract({
    selected:'include',
    paths:'//abc:elem',
```

```
    namespaces: {'abc': 'http://marklogic.com/test/abc'}
    }))
)
.result(function(response) {
  response.length.should.equal(1);
  var document = response[0];
  document.should.have.property('content');
  var content = document.content;
  var assert = require('assert');

assert(content.includes('<abc:elem
xmlns:abc="http://marklogic.com/test/abc">word</abc:elem>'));
done();
})
```

5.0 Using the Optic API for Relational Operations

This chapter covers the following topics related to performing relational operations on indexed values and documents using the Optic capabilities of the Node.js Client API:

5.1 Introduction to the Optic Interfaces

The Optic API enables you to perform relational operations on indexes and documents. For example, you can extract data in row format, perform joins, and perform relational queries on XML and JSON documents. You can also extract a row view of data from other sources, such as lexicons and semantic triples.

The Optic capabilities of the Node.js Client API closely mirror the server-side Optic API described in Optic API for Multi-Model Data Access in the *Application Developer's Guide*. Refer to that guide for conceptual details.

The usage model for an optic query in Node.js is as follows:

1. Build an Optic execution plan on the client using the `planBuilder` interface. For details, see "Generating a Plan" on page 185.

2. Execute the plan on MarkLogic, resulting in generation of a row set. For details, see "Invoking a Plan" on page 186.

3. Process the results returned by MarkLogic on the client. For details on result formats, see "Configuring Row Set Format" on page 189.

Execution of a plan can yield a row that satisfies any of several common use cases:

- A traditional flat list of atomic values with names and XML Schema atomic datatypes.

- A dynamic JSON or XML document with substructure and leaf atomic values or mixed text.
- An envelope with out-of-band metadata properties and relations for a list of documents.

In addition to executing a query plan, you can also perform the following Optic related operations:

- Generate an execution plan explanation that reflects the logical flow of the plan as a sequence of atomic operations.
- Export a serializable version of the plan for later use.

5.2 Interface Summary

The following are the key Optic interfaces in the Node.js Client API:

Interface or Method	Description
`marklogic.planBuilder`	Use a `planBuilder` to construct an Optic plan to be evaluated using `rows.query`. For details, see “Generating a Plan” on page 185.
`DatabaseClient.rows`	An object that exposes methods for evaluating Optic plans, such as `query`, `queryAsStream`, and `explain`.

`rows.query`
 	
`rows.queryAsStream`
	Execute an Optic plan and return the results specified by the plan, in the form of a row set. For details, see “Invoking a Plan” on page 186 and “Streaming Row Data” on page 193.

`rows.explain`	Generate an Optic API execution plan that expresses the logical dataflow of a plan a sequence of atomic operations. For details, see “Generating an Execution Plan” on page 198.
`planBuilder.export`	Generate a JavaScript object representation of the Abstract Syntax Tree for a plan, which can be serialized for later use. For details, see “Serializing a Plan” on page 199.

5.3 Preparing to Run the Examples

The examples in this chapter use the data, templates, and a plan from SQL on MarkLogic Server Quick Start in the *SQL Data Modeling Guide*. If you want to run the examples, you should use the SQL quick start to set up your environment. See the following topics in the *SQL Data Modeling Guide*:

- Setup MarkLogic Server
- Load the Data
- Create Template Views

If you set up a database named "SQLdata", as directed in the Quick Start, then add a `database` property to your `DatabaseClient` connection information. For example, if your my-connection.js module should be similar to the following when running the Optic examples:

```
module.exports = {
  connInfo: {
    host: 'localhost',
    port: 8000,
    user: your-ml-username,
    password: your-ml-user-password,
    database: 'SQLdata'
  }
};
```

For more details on configuring a `DatabaseClient` for running the examples, see "Using the Examples in This Guide" on page 31.

5.4 Generating a Plan

Use `marklogic.planBuilder` to construct an Optic query plan. Usually, you then use the plan to extract row data from MarkLogic. You can also generate a plan explanation or serialize a built plan.

An Optic plan defines a pipeline of relational operations to be applied to a row set.

1. Select a data source using one of the data accessor methods, `planBuilder.from*`. For example, `planBuilder.fromLiterals`, `planBuilder.fromView`, `planBuilder.fromTriples`, or `planBuilder.fromLexicons`.

2. Refine your plan using modifier and composer operations, such as `select`, `joinInner`, `where`, and `orderBy`.

3. Optionally, specify a mapper or reducer to be applied to each row. See `planBuilder.PreparePlan.map` or `planBuilder.PreparePlan.reduce`. The mapper or reducer function runs on MarkLogic.

For example, the following code snippet builds a plan to list the employee ID, first name, and last name of all employees, in order of employee ID, using the configuration and data from the SQL on MarkLogic Server Quick Start chapter in the *SQL Data Modeling Guide*.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const pb = marklogic.planBuilder;

pb.fromView('main', 'employees')
  .select(['EmployeeID', 'FirstName', 'LastName'])
  .orderBy('EmployeeID')
```

The plan is not executed until you process it with `rows.query` or `rows.queryAsStream`. For details, see "Invoking a Plan" on page 186.

For details on the logical structure of a plan and the available operators, see Objects in an Optic Pipeline in the *Application Developer's Guide*. The Node.js `planBuilder` interface exposes methods with the same names and purpose as the Server-Side JavaScript Optic API.

The `planBuilder` interface includes namespaces that expose proxies for many server-side operations. For example, you can "call" cts query constructors or a server-side function in the `fn`, `xdmp`, or `map` namespaces. The placeholder in the plan translates into an equivalent invocation of a server-side function during plan execution.

For example, the following code constructs a plan that includes cts query constructor proxy calls:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const pb = marklogic.planBuilder;

db.rows.query(
  pb.fromView('main', 'employees')
    .where(pb.cts.andQuery([
             pb.cts.wordQuery('Senior'),
             pb.cts.wordQuery('Researcher')]))
    .select(['FirstName', 'LastName', 'Position']),
  {columnTypes: 'header'}
).then( function(rows) {
    console.log(JSON.stringify(rows, null, 2));
}).catch(
  function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

For more information on the proxy functions, see the planBuilder API Reference and the following topics in the *Application Developer's Guide*:

- Expression Functions For Processing Column Values
- Functions Equivalent to Boolean, Numeric, and String Operators
- Node Constructor Functions

For details on the behavior of a particular server-side function, see the *MarkLogic Server-Side JavaScript Function Reference*.

5.5 Invoking a Plan

Use `rows.query` or `rows.queryAsStream` to execute a plan and generate a row set. You can also pass in an options object to control the output format, specify parameter bindings, or pin the query to a specific point-in-time.

Note: Unlike most other Node.js Client API operations, `query` and `queryAsStream` do not support a `result` method. Instead, the plan is sent to MarkLogic for evaluation when you invoke the `then` or `on` methods. See the examples below.

To perform a query at a specific point in time, pass a `Timestamp` object as the value of the `timestamp` property of the `options` parameter. For more details, see "Performing Point-in-Time Operations" on page 23.

For example, the following code constructs a plan, executes it on MarkLogic, and then processes the resulting row set. The plan and results are based on the configuration and data from the SQL on MarkLogic Server Quick Start chapter in the *SQL Data Modeling Guide*.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const pb = marklogic.planBuilder;

db.rows.query(
  pb.fromView('main', 'employees')
    .select(['EmployeeID', 'FirstName', 'LastName'])
    .orderBy('EmployeeID')
).then( function(rows) {
  console.log(JSON.stringify(rows, null, 2));
}).catch( function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

The following example is the equivalent code using `queryAsStream` to stream the results as JavaScript objects instead of returning them all at once. Several stream modes are available; for details, see "Streaming Row Data" on page 193.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const pb = marklogic.planBuilder;

db.rows.queryAsStream(
  pb.fromView('main', 'employees')
    .select(['EmployeeID', 'FirstName', 'LastName'])
    .orderBy('EmployeeID'),
  'object'
).on( 'data', function(rows) {
    console.log(JSON.stringify(rows, null, 2));
}).on ('end', function() {
  console.log('done');
});
```

The plan returns results similar to the following. You can use the options parameter of the `query` and `queryAsStream` methods to customize the output format; for details, see "Configuring Row Set Format" on page 189.

```
{ "columns": [
    { "name": "main.employees.EmployeeID" },
    { "name": "main.employees.FirstName" },
    { "name": "main.employees.LastName" }
  ],
  "rows": [
    { "main.employees.EmployeeID": {
        "type": "xs:integer",
        "value": 1
      },
      "main.employees.FirstName": {
        "type": "xs:string",
        "value": "John"
      },
      "main.employees.LastName": {
        "type": "xs:string",
        "value": "Widget"
      }
    },
    { "main.employees.EmployeeID": {
        "type": "xs:integer",
        "value": 2
      },
      "main.employees.FirstName": {
        "type": "xs:string",
        "value": "Jane"
      },
      "main.employees.LastName": {
        "type": "xs:string",
        "value": "Lead"
      }
    },
    { "main.employees.EmployeeID": {
        "type": "xs:integer",
        "value": 3
      },
      "main.employees.FirstName": {
        "type": "xs:string",
        "value": "Steve"
      },
      "main.employees.LastName": {
        "type": "xs:string",
        "value": "Manager"
      }
    }, ...
  ]
}
```

5.6 Configuring Row Set Format

When you invoke a query plan as described in “Invoking a Plan” on page 186, the result is a row set. You can use the options parameter of `rows.query` and `rows.queryAsStream` to configuring the layout of the row set.

This section covers the following topics related to how configuration options affect the layout of a row set.

- Configuration Options
- Layout Examples

5.6.1 Configuration Options

You can fetch row data in the form of JSON objects, JSON arrays, XML elements, or CSV (comma separated values).

For JSON and XML, you can also control whether column type information is a part of each row or is only part of the column header data. You should include type information with row unless you know each of your columns contain values of the same type.

Use the following options to configure the row set format and layout:

- `format`: Specify the overal format as `json`, `xml`, or `csv`. Default: `json`.
- `structure`: When the format is JSON, specify whether each row should be represented as an object or an array. Default: `object`.
- `columnTypes`: Specify whether to embed column value type information in each row or only in the column header. Only meaningful when `format` is `json` or `xml`. Default: `rows`.
- `complexValues`: Specify whether to return column values with non-atomic type inline or by reference. Only meaningful with `rows.queryAsStream`. Default: `inline`.

The option settings can yield different layouts when returning a row set as a single document with `query` versus an object stream with `queryAsStream`. The rest of this section explores how various option settings interact for `query` and `queryAsStream`.

5.6.2 Layout Examples

This section illustrates how various configuration option settings affect the data passed to your handler when you use `rows.query` and the promise handling pattern. When you use this pattern, your response handler receives the entire row set at once. For more details on promises, see “Promise Result Handling Pattern” on page 20.

You can also stream a row set using queryAsStream. The output is similar, but your handler receives data in chunks. For details, see “Streaming Row Data” on page 193.

When you fetch rows as a JSON array, the first item in the array is the column header data. When you fetch rows as CSV, the first record is the column header data.

The following table summarizes the output produced by various option combinations when fetching rows using `rows.query`. Each example displays the column header data (where appropriate) and one row.

Options	Example Output

(default)
 	
`{format:'json',`
`structure: 'object',`
`columnTypes: 'rows'}`
	`{ columns: [`
	`{name: 'main.employees.Employee'},`
	`{name: 'main.employees.FirstName'},`
	`{name: 'main.employees.LastName'}`
	`],`
	`rows: [`
	`{ 'main.employees.EmployeeID': {`
	`type: 'xs:integer', value: 1},`
	`'main.employees.FirstName': {`
	`type: 'xs:string', value: 'John'},`
	`'main.employees.LastName': {`
	`type: 'xs:string', value: 'Widget'}`
	`},`
	`{...}, ...`
	`]`
	`}`

`{format:'json',`
`structure: 'object',`
`columnTypes: 'header'}`
	`{ columns: [`
	`{ name: 'main.employees.EmployeeID',`
	`type: 'xs:integer' },`
	`{ name: 'main.employees.FirstName',`
	`type: 'xs:string' },`
	`{ name: 'main.employees.LastName',`
	`type: 'xs:string' }`
	`],`
	`rows: [`
	`{ 'main.employees.EmployeeID': 1,`
	`'main.employees.FirstName': 'John',`
	`'main.employees.LastName': 'Widget'`
	`},`
	`{...}, ...`
	`]`
	`}`

Options	Example Output
`{format:'json', structure: 'array', columnTypes: 'rows'}`	`[ [ { name: 'main.employees.EmployeeID' }, { name: 'main.employees.FirstName' }, { name: 'main.employees.LastName' } ], [ { type: 'xs:integer', value: 1 }, { type: 'xs:string', value: 'John' }, { type: 'xs:string', value: 'Widget' } ], [...], ... ]`
`{format:'json', structure: 'array', columnTypes: 'header'}`	`[ [ { name: 'main.employees.EmployeeID', type: 'xs:integer' }, { name: 'main.employees.FirstName', type: 'xs:string' }, { name: 'main.employees.LastName', type: 'xs:string' } ], [ 1, 'John', 'Widget' ], [...], ... ]`

`{format: 'csv' structure: 'object'}`	A block of text, containing one line per row:
	`main.employees.EmployeeID,main.employees.FirstName,main.employees.LastName`
	`1,John,Widget`
	`...`

`{format: 'csv' structure: 'array'}`	A block of text, containing one line per row:
	`["main.employees.EmployeeID", "main.employees.FirstName", "main.employees.LastName"]`
	`[1, "John", "Widget"]`
	`...`

Options	Example Output

`{format:'xml', columnValues: 'rows'}`	Serialized XML of the following form:
	
	`<t:table xmlns:t="http://marklogic.com/table">`
	`  <t:columns>`
	`  <t:column name="main.employees.EmployeeID"/>`
	`  <t:column name="main.employees.FirstName"/>`
	`  <t:column name="main.employees.LastName"/>`
	`  </t:columns>`
	`  <t:rows>`
	`  <t:row>`
	`  <t:cell name="main.employees.EmployeeID"`
	`  type="xs:integer">1</t:cell>`
	`  <t:cell name="main.employees.FirstName"`
	`  type="xs:string">John</t:cell>`
	`  <t:cell name="main.employees.LastName"`
	`  type="xs:string">Widget</t:cell>`
	`  </t:row>`
	`  ...`
	`  </t:rows>`
	`</t:table`

`{format:'xml', columnValues: 'header'}`	Serialized XML of the following form:
	
	`<t:table xmlns:t="http://marklogic.com/table">`
	`  <t:columns>`
	`  <t:column name="main.employees.EmployeeID"`
	`  type="xs:integer"/>`
	`  <t:column name="main.employees.FirstName"`
	`  type="xs:string"/>`
	`  <t:column name="main.employees.LastName"`
	`  type="xs:string"/>`
	`  </t:columns>`
	`  <t:rows>`
	`  <t:row>`
	`  <t:cell name="main.employees.EmployeeID">1</t:cell>`
	`  <t:cell name="main.employees.FirstName">John</t:cell>`
	`  <t:cell name="main.employees.LastName">Widget</t:cell>`
	`  </t:row>`
	`  <t:row>`
	`  ...`
	`  </t:rows>`
	`</t:table>`

5.7 Streaming Row Data

You can use `rows.queryAsStream` to stream a row set from MarkLogic. Depending on the streaming mode, data is returned to your handler in chunks of bytes, JavaScript objects, or JSON text sequence records.

- Object Mode Streaming
- Chunked Mode Streaming
- Sequence Mode Streaming

Row sets formatted as XML or CSV can only be streamed in chunked mode. Rows sets formatted as JSON can be streamed in any mode.

5.7.1 Object Mode Streaming

When you fetch a row set with `queryAsStream` in object mode, each invocation of your `on('data')` handler receives either a JavaScript object or an array, depending on the value of structure option. Column header data is passed on the first invocation.

> **Note:** You can only use object mode when streaming row data as JSON. To stream XML or CSV row set data, use chunked mode. For details, see “Chunked Mode Streaming” on page 195.

The default streaming mode is chunked. To stream results in object mode, set the `streamType` parameter of `queryAsStream` to `'object'`:

```
db.rows.queryAsStream(plan, 'object', options)
```

The following table illustrates the output produced by various option combinations when fetching rows using `rows.queryAsStream` with the stream in object mode. Each example includes the column header object or array passed in on the first invocation of your data handler, followed by an example data row.

Options	Example Output

`{format:'json', structure: 'object', columnTypes: 'rows'}`	`{ columns: [ { name: 'main.employees.EmployeeID' }, { name: 'main.employees.FirstName' }, { name: 'main.employees.LastName' } ]}`
	
	`{ 'main.employees.EmployeeID': { type: 'xs:integer', value: 1 }, 'main.employees.FirstName': { type: 'xs:string', value: "John" }, 'main.employees.LastName': { type: 'xs:string', value: 'Widget' } }`

`{format:'json', structure: 'object', columnTypes: 'header'}`	`{ columns: [ { name: 'main.employees.EmployeeID', type: 'xs:integer' }, { name: 'main.employees.FirstName', type: 'xs:string' }, { name: 'main.employees.LastName', type: 'xs:string' } ]}`
	
	`{ 'main.employees.EmployeeID': 1, 'main.employees.FirstName': 'John', 'main.employees.LastName': 'Widget' }`

Options	Example Output
{format:'json', structure: 'array', columnTypes: 'rows'}	[{ name: 'main.employees.EmployeeID' }, { name: 'main.employees.FirstName' }, { name: 'main.employees.LastName' }] [{ type: 'xs:integer', value: 1 }, { type: 'xs:string', value: 'John' }, { type: 'xs:string', value: 'Widget' }]
{format:'json', structure: 'array', columnTypes: 'header'}	[{ name: 'main.employees.EmployeeID', type: 'xs:integer' }, { name: 'main.employees.FirstName', type: 'xs:string' }, { name: 'main.employees.LastName', type: 'xs:string' }] [1, 'John', 'Widget']

5.7.2 Chunked Mode Streaming

When you fetch a row set with `queryAsStream` in chunked mode, your handler receives control when some number of bytes are read.

Each chunk provided to your handler is a buffer of bytes, formatted as serialized JSON, XML, or CSV, depending on your option settings. The aggregate byte stream is formatted similar to the layouts shown in “Layout Examples” on page 189.

The default stream mode is chunked. You can also set it explicitly. For example:

```
db.rows.queryAsStream(plan, 'chunked', options)
```

5.7.3 Sequence Mode Streaming

When you fetch a row set with `queryAsStream` in sequence mode, your handler receives control when a record in JSON text sequence format is available. Each record contains the byte representation of either a JSON object or JSON array, depending on your option settings.

> **Note:** You can only use sequence mode when streaming row data as JSON. To stream XML or CSV row set data, use chunked mode. For details, see “Chunked Mode Streaming” on page 195

Each record begins with a record separator (0x1E) in the first byte and ends with an ASCII line feed character (0x0A), as described in https://tools.ietf.org/html/rfc7464.

The default stream mode is chunked. To activate sequence mode, set the streamType parameter of `queryAsStream` to `'sequence'`:

```
db.rows.queryAsStream(plan, 'sequence', options)
```

Each record is formatted like the examples “Layout Examples” on page 189 when the `format` option is set to `'json'` and the `structure` option is set to either `'object'` or `'array'`.

For example, the following code produces records formatted as JSON objects, with column data in the header (first) record. For illustrative purposes, each record is converted to a string and stripped of the record separators before displaying it on the console.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const fs = require('fs');

const db = marklogic.createDatabaseClient(my.connInfo);
const plan = fs.readFileSync('./plan.json', 'utf8');
const options = {
  format: 'json',
  structure: 'object',
  columnTypes: 'header'
};

db.rows.queryAsStream(plan, 'sequence', options)
  .on( 'data', function(record) {
    const asString = record.toString();
    console.log(asString.substring(1,asString.length-2));
  }).on ('end', function() {
    console.log('done');
  });
```

This example returns output similar to the following, where each record contains an object that represents one row. The first record contains the column header data.

```
{"columns":[{"name":"main.employees.EmployeeID","type":"xs:integer"},{
"name":"main.employees.FirstName","type":"xs:string"},{"name":"main.em
ployees.LastName","type":"xs:string"}]}
{"main.employees.EmployeeID":1,"main.employees.FirstName":"John","main
.employees.LastName":"Widget"}
...
```

If you set the structure option to array, then each record contains an array that represents one row. The first record contains the column header data.

```
[{"name":"main.employees.EmployeeID","type":"xs:integer"},{"name":"mai
n.employees.FirstName","type":"xs:string"},{"name"
:"main.employees.LastName","type":"xs:string"}]
[1,"John","Widget"]
...
```

5.8 Passing Parameters into a Plan

If your plan uses placeholder parameters, use the `bindings` option to specify values for the placeholders when you invoke the plan.

The bindings option value is a JavaScript object where the property names correspond to parameters, and the values are either a parameter value or an object that specifies a type or language key and a value. That is, each property must take one of forms shown below:

```
{bindings : {
  paramName1 : value1,
  paramName2 : {value: value2, type: typeNameString},
  paramName3 : {value: value3, lang: languageCode}
}}
```

The type name can be any derivation of `xs:atomicType` other than `xs:QName`. For example, you can use type names such as `'string'`, `'integer'`, and `'decimal'`. If you do not specify a type, the value is interpreted as a string. Use a language code to bind language-tagged strings.

For example, if you defined a placeholder variable named “start” in your plan definition, then you could specify a value for “start” in the `bindings` option in any of the following ways:

```
{bindings:{
  start: 'apple'
}

{bindings:{
  start: {value: 'apple', type: 'string'}
}

{bindings:{
  start: {value: 'apple', lang: 'en'}
}
```

5.9 Handling Complex Column Values

If your row set includes column values with non-atomic type, such as XML elements, JSON arrays, JSON objects, binary content, or text nodes, they are serialized inline by default. For example, the following row contains a column named “complex” whose value is a serialized XML element:

```
{ "id":{
    "type":"xs:integer",
    "value":1
  },
  "complex":{
    "type":"element",
    "value":"<root><A>Detail 1</A><B>2015-12-01</B></root>"
}}
```

You can use the `complexValues` option of `rows.queryAsStream` to specify whether such complex values should be included inline (as shown above) or by reference. You cannot configure the handling of complex values when executing a plan using `rows.query`.

For example, the following option setting specifies complex values should be returned by reference.

```
{complexValues: 'reference'}
```

For example, suppose your row set includes a column named “node” whose value is an XML element. The default behavior (inline) yields a serialized string of the following form for the node value:

```
"node": {
  "type": "element",
  "value": "<alpha><a>true</a></alpha>"
}
```

If you set the complexValues option to “reference”, then the same column value is rendered as follows:

```
"node": {
  "contentType": "application/xml",
  "format": "xml",
  "content": "<?xml version=\"1.0\"
encoding=\"UTF-8\"?>\n<beta><b>false</b></beta>"
}
```

The node value is actually returned as a reference to a complex value by MarkLogic, but the reference is resolved and expanded for you in the results returned from the Node.js Client API.

The details of the format depend on the output formatting options you choose and the type of complex value. For example, if the complex value is a binary document, then it would be returned as a base64 encoded value inline, but as a Node.js `Buffer` when fetched by reference.

5.10 Generating an Execution Plan

Use `rows.explain` to generate an execution plan that expresses the logical flow of an Optic plan as a sequence of atomic operations. For more details, see Optic Execution Plan in the *Application Developer's Guide*.

For example, the following code generates an execution plan based on the data from SQL on MarkLogic Server Quick Start in the *SQL Data Modeling Guide*.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const pb = marklogic.planBuilder;
```

```
db.rows.explain(
  pb.fromView('main', 'employees')
    .select(['EmployeeID', 'FirstName', 'LastName'])
    .orderBy('EmployeeID')
    .limit(3)
).then( function(rows) {
    console.log(JSON.stringify(rows, null, 2));
}).catch(
  function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

5.11 Serializing a Plan

Use the `export` method of a plan object to generate a serializable version of a plan that can be saved to persistent storage for later use. The `export` method produces a JavaScript object representation of a plan, which can be serialized to JSON using, for example, `JSON.stringfy`.

For example, the following code generates a JSON serialization of a plan and logs it to the console:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const pb = marklogic.planBuilder;

console.log( JSON.stringify(
  pb.fromView('main', 'employees')
    .select(['EmployeeID', 'FirstName', 'LastName'])
    .orderBy('EmployeeID')
    .limit(3)
    .export(),
  null,2));
```

To subsequently use a serialized plan, convert it back into a JavaScript object and pass the object to `rows.query` or `rows.queryAsStream`. For example:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const pb = marklogic.planBuilder;

// Read serialized plan from a file or other storage, then...
const serializedPlan = ...;

const thePlan = JSON.parse(serializedPlan);
db.rows.query(thePlan)
  .then( function(rows) {
    console.log(JSON.stringify(rows, null, 2));
```

```
}).catch(
  function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

6.0 Working With Semantic Data

This chapter discusses the following topics related to using the Node.js Client API to load semantic triples, manage semantics graphs, and query semantic data:

This chapter only covers details specific to using the Node.js Client API for semantic operations. For more details, see the *Semantics Developer's Guide*.

6.1 Overview of Common Semantics Tasks

The following table lists some common tasks related to Semantics, along with the method best suited for the completing the task. For a complete list of interfaces, see the Node.js API Reference.

If you want to	Then use
Load semantic triples into a named graph or the default graph without using SPARQL Update.	`DatabaseClient.graphs.write` For details, see "Loading Triples" on page 202.
Manage graphs or graph data with SPARQL Update.	`DatabaseClient.graphs.sparlUpdate` For details, see "Using SPARQL Update to Manage Graphs and Graph Data" on page 211.
Read a semantic graph from the database.	`DatabaseClient.graphs.read` For details, see "Retrieving the Contents, Metadata, or Permissions of a Graph" on page 209.
Query semantic data with SPARQL	`DatabaseClient.graphs.sparql` For details, see "Querying Semantic Triples With SPARQL" on page 204

To read a graph or perform a semantic query at fixed point in time, pass a Timestamp object as a parameter to your call. For more details, see “Performing Point-in-Time Operations” on page 23.

6.2 Loading Triples

This topic covers using `DatabaseClient.graphs.write` or `DatabaseClient.graphs.writeStream` to load semantic triples into the database in several formats. For a list of supported formats, see Supported RDF Triple Formats in *Semantics Developer's Guide*. You can also insert triples with a SPARQL Update request; for details, see “Using SPARQL Update to Manage Graphs and Graph Data” on page 211.

> **Note:** The collection lexicon must be enabled on your database when using the semantics REST services or use the GRAPH '?g' construct in a SPARQL query.

Use `DatabaseClient.graphs.write` to upload a block of triples to MarkLogic Server in a single request. Use `DatabaseClient.graphs.writeStream` to incrementally stream a large number of triples to MarkLogic Server; for details, see “Streaming Into the Database” on page 22. The input semantic data can be expressed as a string, object, or `Readable` stream.

A call to `DatabaseClient.graphs.write` or `DatabaseClient.graphs.writeStream` always includes at least a MIME type parameter indicating the format of the input triples. If you also include a graph URI, the triples are loaded into that graph. If you do not include a graph URI, the triples are loaded into the default graph. That is, you can use one of these two forms, depending on the destination graph:

```
// load into the default graph
db.graphs.write(mimeType, triples)

// load into a named graph
db.graphs.write(graphURI, mimeType, triples)
```

Optionally, you can pass a boolean repair flag. If present and set to true, MarkLogic Server attempts to repair invalid input triples during ingestion. For example:

```
db.graphs.write(graphURI, true, mimeType, triples)
```

You can also call the write functions with a call object instead of with positional parameters. The call object has the following properties:

```
db.graphs.write({
  uri: graphURI,              // optional, omit for default graph
  contentType: mimeType,      // required
  data: triples,              // required
  repair: boolean             // optional
})
```

The output from a call to `DatabaseClient.graphs.write` or `DatabaseClient.graphs.writeStream` is an object that indicates whether the triples were loaded into the default graph or a named graph. If the destination is a named graph, the graph name is returned in the `graph` property. For example:

```
// result of loading into default graph
{ defaultGraph: true, graph: null }

// result of loading into a named graph
{ defaultGraph: false, graph: 'example-graph-uri' }
```

The following example uses `DatabaseClient.graphs.write` to load a set of triples in RDF/JSON format into the default graph.

```
const fs = require('fs');
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);

const triples = {
  'http://dbpedia.org/resource/Joyce_Carol_Oates' : {
    'http://dbpedia.org/property/influences' : [ {
      'type' : 'uri',
      'value' : 'http://dbpedia.org/resource/Ernest_Hemingway'
    } ] ,
    'http://dbpedia.org/ontology/influencedBy' : [ {
      'type' : 'uri',
      'value' : 'http://dbpedia.org/resource/Ernest_Hemingway'
    } ]
  },
  'http://dbpedia.org/resource/Death_in_the_Afternoon' : {
    'http://dbpedia.org/ontology/author' : [ {
      'type' : 'uri',
      'value' : 'http://dbpedia.org/resource/Ernest_Hemingway'
    } ] ,
    'http://dbpedia.org/property/author' : [ {
      'type' : 'uri',
      'value' : 'http://dbpedia.org/resource/Ernest_Hemingway'
    } ]
  }
};

db.graphs.write('application/rdf+json', triples).result(
    function(response) {
      if (response.defaultGraph) {
        console.log('Loaded into default graph');
      } else {
        console.log('Loaded into graph ' + response.graph);
      };
    },
    function(error) { console.log(JSON.stringify(error)); }
);
```

The following example uses `DatabaseClient.graphs.writeStream` to load a set of triples in N-Quads format into a named graph (`example-graph`). The triples are streamed from the file "input.nq" using a `Readable` stream and piped into the `Writable` stream returned by `DatabaseClient.graphs.createWriteStream`. The transaction is not committed until the all the data from the input stream is transmitted.

```
const fs = require('fs');
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
// Load into a named graph using a write stream
const writer =
  db.graphs.createWriteStream('example-graph', 'application/n-quads');
writer.result(
  function(response) {
    if (response.defaultGraph) {
      console.log('Loaded triples into default graph');
    } else {
      console.log('Loaded triples into graph ' + response.graph);
    };
  },
  function(error) { console.log(JSON.stringify(error)); }
);
fs.createReadStream('input.nq').pipe(writer);
```

To read a semantic graph, use `DatabaseClient.graphs.read`; for details, see "Retrieving the Contents, Metadata, or Permissions of a Graph" on page 209. To query semantic data, use `DatabaseClient.graphs.sparql`; for details, see "Querying Semantic Triples With SPARQL" on page 204. For additional operations, see the Node.js API Reference.

6.3 Querying Semantic Triples With SPARQL

Use the `DatabaseClient.graphs.sparql` method to evaluate SPARQL queries against triples in the database. For more details on using SPARQL queries with MarkLogic, see the *Semantics Developer's Guide*.

> **Note:** The collection lexicon must be enabled on your database when using the semantics REST services or use the GRAPH '?g' construct in a SPARQL query.

You can evaluate a SPARQL query by calling `DatabaseClient.graphs.sparql`. The query is not sent to MarkLogic for evaluation until you call `result()`. You can invoke the `sparql` method in the following ways:

```
// (1) db.graphs.sparql(responseContentType, query)
db.graphs.sparql(
  'application/sparql-results+json',
  'SELECT ?s ?p WHERE {?s ?p Paris }')

// (2) db.graphs.sparql(responseContentType, defaultGraphUris, query)
```

```
db.graphs.sparql(
  'application/sparql-results+json',
  'http://def/graph1', 'http://def/graph2',
  'SELECT ?s ?p WHERE {?s ?p Paris }')

// (3) db.graphs.sparql(callObject)
db.graphs.sparql({
  contentType: 'application/sparql-results+json',
  query: 'SELECT ?s ?p WHERE {?s ?p Paris }',
  start: 0,
  length: 15
})
```

You must include at least a SPARQL query string and a response content type (MIME type) in your call. For a complete list of the accepted response MIME types, see SPARQL Query Types and Output Formats in the *Semantics Developer's Guide*.

Passing in a call object enables you to configure attributes of your query such as named graph URIs, an additional document query, result subset controls, and inference rulesets. For a complete list of the configuration properties, see `graphs.sparql` in the *Node.js Client API Reference*.

Advanced users can supply a `CombinedQueryDefinition` as the docQuery parameter of `db.graphs.sparql`. A combined query enables advanced users finer control over the optional document constraint query you can include in your query. For details, see "Searching with Combined Query" on page 150.

6.4 Example: SPARQL Query

The following example evaluates a SPARQL query expressed as a string literal against the default graph.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);

const query = [
  'PREFIX foaf: <http://xmlns.com/foaf/0.1/>' ,
  'PREFIX ppl:  <http://people.org/>' ,
  'SELECT ?personName1' ,
  'WHERE {' ,
  '    ?personUri1 foaf:name  ?personName1 ;' ,
  '                foaf:knows ppl:person3 .' ,
  '    ?personUri1 foaf:name  ?personName1 .' ,
  '    }'
];
db.graphs.sparql('application/sparql-results+json', query.join('\n')
).result(function (result) {
  console.log(JSON.stringify(result, null, 2));
}, function(error) {
```

```
  console.log(JSON.stringify(error, null, 2));
});
```

Running the script produces output similar to the following, given triples that indicate Person 1 and Person 2 know Person 3.

```
{ "head": {
    "vars": [ "personName1" ]
  },
  "results": {
    "bindings": [
      {
        "personName1": {
          "type": "literal",
          "value": "Person 1",
          "datatype": "http://www.w3.org/2001/XMLSchema#string"
        }
      },
      {
        "personName1": {
          "type": "literal",
          "value": "Person 2",
          "datatype": "http://www.w3.org/2001/XMLSchema#string"
        }
      }
    ]
}}
```

For more examples, see `test-basic/graphs.js` in the `node-client-api` project source directory.

6.5 Managing Graphs

This section covers the following graph management operations:

- Creating or Replacing a Graph
- Adding Triples to an Existing Graph
- Removing a Graph
- Retrieving the Contents, Metadata, or Permissions of a Graph
- Testing for Graph Existence
- Retrieving a List of Graphs

You can also use SPARQL Update to perform similar operations; for details, see “Using SPARQL Update to Manage Graphs and Graph Data” on page 211.

> **Note:** Many graph management operations only apply to managed triples.

6.5.1 Creating or Replacing a Graph

You can use `DatabaseClient.graphs.write` to create or replace a graph without using SPARQL Update. For details and examples, see "Loading Triples" on page 202. You can also use `DatabaseClient.graphs.sparqlUpdate` to create or modify a graph; for details, see "Using SPARQL Update to Manage Graphs and Graph Data" on page 211.

If a graph does not exist when you write to it, the graph is created. If the graph already exists when you write to it, the unmanaged triples in the graph are replaced with the new triples; this is equivalent to removing and recreating the graph.

Any unmanaged triples in the graph are unaffected by this operation.

For more details, see the *Node.js Client API Reference*.

6.5.2 Adding Triples to an Existing Graph

Use `DatabaseClient.graphs.merge` to add triples to an existing graph without replacing the current contents. For other update operations or finer control, use the `DatabaseClient.graphs.sparqlUpdate` to interact with a graph using SPARQL Update; for more details, see "Using SPARQL Update to Manage Graphs and Graph Data" on page 211. To replace the contents of a graph, use `DatabaseClient.graphs.write`; for details, see "Creating or Replacing a Graph" on page 207.

You can invoke `graphs.merge` using the following forms:

```
// Add triples to a named graph
db.graphs.merge(graphUri, contentType, tripleData)

// Add triples to the default graph
db.graphs.merge(null, contentType, tripleData)

// Add triples to a named graph or the default graph using a call object
db.graphs.merge({uri: ..., contentType: ..., data: ..., ...})
```

Use the call object pattern to specify operation parameters such as permissions and a transaction id; for details, see the *Node.js Client API Reference*. When you use the call object pattern, you can omit the `uri` property or set it to null to merge triples into the default graph.

The following example uses the call object pattern to insert two new triples into the graph named "MyGraph", repairing the data as needed. In this example, the triples are passed as a string in Turtle format. You can use other triple formats. You can also supply triple data as a stream, buffer, or object instead of a string.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
```

```
const triples = [
  '@prefix p1: <http://example.org/marklogic/predicate/> .',
  '@prefix p0: <http://example.org/marklogic/people/> .',
  'p0:Julie_Smith  p1:livesIn \"Sterling\" .',
  'p0:Jim_Smith    p1:livesIn \"Bath\" .'
];
db.graphs.merge({
    uri: 'MyGraph',
    contentType: 'text/turtle',
    data: triples.join('\n'),
    repair: true
}).result(
    function(response) {
      console.log(JSON.stringify(response));
    },
    function(error) { console.log(JSON.stringify(error)); }
);
```

If the operation is successful, the script produces output similar to the following:

```
{"defaultGraph":false,"graph":"MyGraph","graphType":"named"}
```

For more details, see the *Node.js Client API Reference*.

6.5.3 Removing a Graph

Use `DatabaseClient.graphs.remove` to remove the triples in a named graph or the default graph. This operation only affects managed triples. If the graph includes unmanaged triples, the embedded triples are unaffected and the graph will continue to exist after this operation.

You can also use a SPARQL Update request to remove a graph; for details, see “Using SPARQL Update to Manage Graphs and Graph Data” on page 211.

If you call `remove` with no parameters, it removes the default graph. If you call `remove` with a graph URI, it removes that named graph.

The following example removes all triples in the graph with URI `example-graph`:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.graphs.remove('example-graph').result(
  function(response) { console.log(JSON.stringify(response)); }
);

// expected output:
// {"defaultGraph":false,"graph":"example-graph"}
```

To remove the default graph, omit the graph URI from your call. The following example removes all triples in the default graph. The value of the `graph` property in the response is `null` because this is the default graph. For a named graph, the `graph` property contains the graph URI.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.graphs.remove().result(
  function(response) { console.log(JSON.stringify(response)); }
);

// expected output:
// {"defaultGraph":true,"graph":null}
```

For more details, see the *Node.js Client API Reference*.

6.5.4 Retrieving the Contents, Metadata, or Permissions of a Graph

Use `DatabaseClient.graphs.read` to retrieve all the triples in a graph, the graph metadata, or graph permissions. You can use this method on the default graph or a named graph.

You can invoke `graphs.read` in the following forms:

```
// (1) Retrieve all triples in the default graph
db.graphs.read(responseContentType)

// (2) Retrieve all triples in a named graph
db.graphs.read(uri, responseContentType)

// (3) Retrieve triples, metadata, or permissions using a call object
db.graphs.read({contentType: ..., ...})
```

When you use the call object pattern, the call object must include at least a `contentType` property. To operate on a named graph, you must also include a `uri` property. If you omit the `uri` property or set it to `null`, the operation applies to the default graph.

You can retrieve triples in several RDF formats; for a list of supported formats, see the API reference for `graphs.read`.

The following example reads all triples from the graph named “MyGraph”. The triples are returned in Turtle format.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);

db.graphs.read('MyGraph','text/turtle')
  .result(
```

```
    function(response) {
      for (const line of response.split('\n')) {
        console.log(line);
      }
    },
    function(error) { console.log(JSON.stringify(error)); }
  );
```

The call object pattern is required when retrieving graph metadata or permissions; a call object can also be used to retrieve triples. Use the `category` property of the call object to specify what to retrieve (content, metadata, or permissions).

The following example retrieves metadata about the graph named “MyGraph”:

```
db.graphs.read({
  uri: 'MyGraph',
  contentType: 'application/json',
  category: 'metadata'
});
```

For more details, see the *Node.js Client API Reference*.

6.5.5 Testing for Graph Existence

Use `DatabaseClient.graphs.probe` to test for the existence of a graph. The following example tests for the existence of a graph with the URI “http://marklogic.com/example/graph”.

```
db.graphs.probe('http://marklogic.com/example/graph')
  .result(
    function(response) {
      console.log(JSON.stringify(response));
    },
    function(error) { console.log(JSON.stringify(error)); }
  );
```

If the graph exists, the call produces output similar to the following:

```
{ "contentType": null,
  "contentLength": null,
  "versionId": null,
  "location": null,
  "systemTime": null,
  "exists": true,
  "defaultGraph": false,
  "graph": "http://marklogic.com/example/graph",
  "graphType": "named"
}
```

If the graph does not exist, the graph produces output similar to the following:

```
{ "exists": false,
  "defaultGraph": false,
  "graph": "NoMyGraph",
  "graphType": "named"
}
```

Probe for the existence of the default graph by omitting the graph URI.

6.5.6 Retrieving a List of Graphs

Use `DatabaseClient.graphs.list` to retrieve a list of graphs stored in MarkLogic. You must specify an expected content MIME type for the response. You can retrieve the list as `text/plain` or `text/html`.

The following example retrieves a list of available graph URI, one URI per line.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);

db.graphs.list('text/uri-list')
  .result(
    function(response) {
      for (const uri of response.split('\n')) {
        console.log(uri);
      }
    },
    function(error) { console.log(JSON.stringify(error)); }
);
```

If the database includes the default graph and a graph with the URI “MyGraph”, then the raw response from the above operation is a string of the following form. Each URI is separated by a newline ('\n').

```
"MyGraph\nhttp://marklogic.com/semantics#graphs\n"
```

For more details, see the *Node.js Client API Reference*.

6.6 Using SPARQL Update to Manage Graphs and Graph Data

You can use a SPARQL Update request to manage graphs and graph data from Node.js by calling `DatabaseClient.graphs.sparqlUpdate`. You can only use this interface to work with managed triples. You can also manage graphs and graph data without using SPARQL Update; for details, see “Managing Graphs” on page 206 and “Loading Triples” on page 202.

You can call `graphs.sparqlUpdate` in the following ways:

```
// (1) pass only the SPARQL Update as a string or ReadableStream
db.graphs.sparqlUpdate(updateOperation)

// (2) pass a call config object that includes the SPARQL Update
db.graphs.sparqlUpdate({data: updateOperation, ...})
```

The call object pattern enables you to control operations details, such as permissions, inferencing rulesets, transaction control, and variable bindings. When you use a call object, the `data` property holds your SPARQL Update request string or stream.

The following example creates a graph, passing in only a SPARQL Update request.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);

db.graphs.sparqlUpdate(
    'CREATE GRAPH <http://marklogic.com/semantics/tutorial/update> ;'
).result(
    function(response) {
      console.log('Graph created.');
      console.log(JSON.stringify(response));
    },
    function(error) { console.log(JSON.stringify(error)); }
);
```

Running this script produces the following output:

```
Graph created.
{"defaultGraph":false,"graph":null,"graphType":"inline"}
```

The following example uses a call object to create a graph and insert triples. The call object defines permissions on the graph, and defines variable bindings used in the SPARQL Update request.

```
const graphURI = 'http://marklogic.com/sparqlupd/example';
db.graphs.sparqlUpdate({
    data:
      'CREATE GRAPH <' + graphURI + '> ;' +
      'INSERT {GRAPH <' + graphURI + '> {?s ?p ?b1 }} '+
      'WHERE  {GRAPH <' + graphURI + '> ' +
          '{?s ?p ?o. filter (?p = ?b2) }}',
    bindings: {
      b1: 'bindval1',
      b2: {value: 'bindval2', type: 'string'}
    },
    permissions: [
      { 'role-name': 'app-user', capabilities: ['read']},
```

```
        { 'role-name': 'admin', capabilities: ['read','update']},
      ]
})
```

For a complete list of configuration properties usable in the call object, see `graphs.sparqlUpdate` in the *Node.js Client API Reference*.

For more details on using SPARQL Update with MarkLogic, see SPARQL Update in the *Semantics Developer's Guide*.

6.7 Applying Inferencing Rules to a SPARQL Query or Update

You can specify one or more inferencing rulesets to apply to a SPARQL query or SPARQL Update request. Inference rules enable you to "discover" new facts about your data at query or update time.

SPARQL inference is discussed in detail in Inference in the *Semantics Developer's Guide*. This section only covers usage details specific to the Node.js Client API.

The following topics are covered:

- Basic Inference Ruleset Usage
- Example: SPARQL Query With Inference Ruleset
- Example: SPARQL Update With Inference Rulesets
- Controlling the Default Database Ruleset

6.7.1 Basic Inference Ruleset Usage

To include one or more inference rulesets when using `graphs.sparql` or `graphs.sparqlUpdate`, pass a call configuration object that includes a `rulesets` property.

For example, to specify one or more rulesets with `graphs.sparql`, construct an input call object that includes at least the following properties.

```
db.graphs.sparql({'content-type': ..., query: ..., rulesets: ...})
```

To specify one or more rulesets with `graphs.sparqlUpdate`, construct an input call object that includes at least the following properties:

```
db.graphs.sparqlUpdate({data: ..., rulesets: ...})
```

The value of the `rulesets` property can be a single string or an array of strings. Each string in `rulesets` must be either the name of a built-in ruleset file or the URI of a custom ruleset installed in the Schemas database.

For more details on built-in rulesets, see Rulesets in the *Semantics Developer's Guide*.

For details on creating custom rulesets, see Creating a New Ruleset in the *Semantics Developer's Guide*. You can install custom rulesets in the Schemas database using standard document operations, such as `DatabaseClient.documents.write`.

To learn more about semantic inferencing with MarkLogic, see Inference in the *Semantics Developer's Guide*.

6.7.2 Example: SPARQL Query With Inference Ruleset

The following example applies the built-in `subPropertyOf.rules` ruleset to a query. This ruleset is automatically installed in the *`MARKLOGIC_INSTALL_DIR`*`/Config` directory when you install MarkLogic.

```
db.graphs.sparql({
  contentType: 'application/sparql-results+json',
  query: 'SELECT ?s ?p WHERE {?s ?p Paris }',
  rulesets: 'subPropertyOf.rules'
})
```

6.7.3 Example: SPARQL Update With Inference Rulesets

The following example applies the built-in `sameAs.rules` ruleset and a custom ruleset installed in the Schemas database with the URI `/my/rules/custom.rules` to a SPARQL Update request:

```
const update = [
  PREFIX exp: <http://example.org/marklogic/people>
  PREFIX pre: <http://example.org/marklogic/predicate>
  INSERT DATA {
    GRAPH <MyGraph>{
      exp:John_Smith pre:livesIn "London" .
      exp:Jane_Smith pre:livesIn "London" .
      exp:Jack_Smith pre:livesIn "Glasgow" .
  }}
];
db.graphs.sparqlUpdate({
  data: update.join('\n'),
  rulesets: ['sameAs.rules', '/my/rules/custom.rules']
})
```

6.7.4 Controlling the Default Database Ruleset

Every database has an implicit, default inferencing ruleset. You can customize the default ruleset for a database, as described in Using the Admin UI to Specify a Default Ruleset for a Database in the *Semantics Developer's Guide*.

The database default ruleset is normally applied to all SPARQL query and update operations. However, you can control whether or not to include the default ruleset for the database in a query or update operation by using the `defaultRulesets` property of the input call object.

Set `defaultRulesets` to “exclude” to exclude the database default ruleset from inferencing. Set `defaultRulesets` to “include” to include the database default ruleset. If you do not explicitly set `defaultRulesets`, the database default ruleset is included in the operation.

The following example excludes the default ruleset from a query operation:

```
db.graphs.sparql({
  contentType: 'application/sparql-results+json',
  query: 'SELECT ?s ?p WHERE {?s ?p Paris }',
  rulesets: 'subPropertyOf.rules',
  defaultRuleSets: 'exclude'
})
```

7.0 Managing Transactions

This chapter covers the following topics related to transaction management using the Node.js Client API.

7.1 Transaction Overview

This section gives a brief introduction to the MarkLogic Server transaction model as it applies to the Node.js Client API. For a full discussion of the MarkLogic transaction model, see Understanding Transactions in MarkLogic Server in the *Application Developer's Guide*..

By default, each operation on the database is equivalent to a single statement transaction. That is, the operation is evaluated as single transaction. For example, when you update one or more documents in the database using `DatabaseClient.documents.write`, the server-side handler effectively creates a new transaction, updates the document(s), commits the transaction, and then sends back a response. The updated documents are visible in the database and available to other operations once the write operation completes successfully. If an error occurs in during the update of one of the documents, the entire operation fails.

The Node.js Client API also enables your application to take direct control of transaction boundaries so that multiple operations can be evaluated in the same transaction context. This is equivalent to the multi-statement transactions described in Multi-Statement Transaction Concept Summary in the *Application Developer's Guide*.

Using multi-statement transactions, you can execute several operations and commit them as a single transaction, ensuring either all or none of the related updates appear in the database. The document manipulation and search capabilities of the Node.js Client API support multi-statement transactions through the `DatabaseClient.transactions` interface, plus the ability to pass a transaction object to most operations.

To use multi-statement transactions:

1. Create a multi-statement transaction using `DatabaseClient.transactions.open`. This operation returns a transaction object. See "Creating a Transaction" on page 217.

2. Perform one or more operations in the context of the transaction by including the transaction object for the `txid` parameter. See "Associating a Transaction with an Operation" on page 218.

3. Commit the transaction using `DatabaseClient.transactions.commit`, or rollback the transaction using `DatabaseClient.transactions.rollback`. See "Committing a Transaction" on page 219 and "Rolling Back a Transaction" on page 219.

If your application interacts with MarkLogic Server through a load balancer, you might need to include a HostId cookie in your requests to preserve session affinity. For details, see "Managing Transactions When Using a Load Balancer" on page 220.

When you explicitly create a transaction, you must explicitly commit it or roll it back. Failure to do so leaves the transaction open until the request or transaction timeout expires. Open transactions can hold locks and consume system resources, so it is important to close transactions when they are complete.

If the request or transaction timeout expires before a transaction is committed, the transaction is automatically rolled back and all updates are discarded. Configure the request timeout of the App Server using the Admin UI. Configure the timeout of a single transaction by setting the `timeLimit` request parameter during transaction creation.

7.2 Creating a Transaction

Use `DatabaseClient.transactions.open` to create a multi-statement transaction. For example:

```
const txObj = null;
db.transactions.open().result()
  .then(function(response) {
    txObj = response
  });
```

This call returns a transaction object that encapsulates state needed to preserve host affinity across the transaction, even in the presence of a load balancer.

Multi-statement transactions must be explicitly committed or rolled back. Failure to commit or rollback the transaction before the request timeout expires causes an automatic rollback. You can assign a shorter time limit to a transaction by supplying a time limit (in seconds) to `open`: For example, the following sets the time limit to `tlimit` and returns a stateful transaction object:

```
db.transactions.open({timeLimit: tlimit})
```

You should not depend on the time limit rolling back your transaction. The limit is only a failsafe. Instead, you should explicitly rollback your transaction when appropriate.

You can also provide a symbolic name when you create a transaction. You must still use the transaction object (or id) in all operations that accept a transaction parameter, but the name can be used with `DatabaseClient.transactions.read` and will show up in the Admin Interface and other transaction status displays.

For example, the following call provides both a time limit and a name, using an input call object with appropriate property names:

```
db.transactions.open({
  timeLimit: 45,
  transactionName: 'mySpecialTxn'
});
```

7.3 Associating a Transaction with an Operation

Once you create a transaction using `DatabaseClient.transactions.open`, you can pass the resulting transaction object (or id) to various operations to perform the operation in the context of a specific transaction.

For example, to update a document in the context of a specific multi-statement transaction, include a transaction id in the `DatabaseClient.documents.write` call:

```
const txnObj = null;
db.transactions.open(true).result()
  .then(function(response) {
     txnObj = response;
     return db.documents.write({
       uri: '/my/documents.json',
       content: {some: 'content'},
       contentType: 'application/json”,
       txid: txnObj
     }).result;
  ...
```

Updates associated with a multi-statement transaction are visible to subsequent operations using the same transaction, but they are not visible outside the transaction until the transaction is committed.

You can have multiple transactions open at the same time, and/or other users can be using the same database concurrently. To prevent conflicts, whenever an update occurs in a transaction, the document is locked until the transaction either commits or rolls back. Therefore, you should commit or roll back your transactions as soon as possible to avoid resource contention.

Note: The database context in which you perform an operation must be the same as the database context in which the transaction was created. Consistency is assured if you’re only using a single `DatabaseClient` configuration.

You can intermix operations that are not part of a transaction with operations that are. Any operation without a `txid` parameter or call object property is not part of a multi-statement transaction. However, you usually group operations in the same transaction together so you can commit or roll back the transaction in a timely fashion.

7.4 Committing a Transaction

Use `DatabaseClient.transactions.commit` to commit a multi-statement transaction. Supply the transaction object (or id) from `DatabaseClient.transactions.open` in your `commit` call. For example:

```
db.transactions.commit(transactionObj);
```

Once a transaction is committed, it cannot be rolled back, and the transaction object (or id) can no longer be used. To perform another transaction, obtain a new transaction by calling `open`.

> **Note:** The database context in which you commit or roll back a transaction must be the same as the database context in which the transaction was created. Consistency is assured if you're only using a single `DatabaseClient` configuration.

7.5 Rolling Back a Transaction

In case of an error or exception, you can roll back an open transaction using `DatabaseClient.transactions.rollback`.

```
db.transactions.rollback(transactionObj);
```

Calling `rollback` cancels the remainder of the transactions and reverts the database to its state prior to the transaction start. It is better to explicitly roll back a transaction than wait for a timeout.

You must have the `rest-writer` or `rest-admin` role or equivalent privileges to roll back a transaction.

> **Note:** The database context in which you commit or roll back a transaction must be the same as the database context in which the transaction was created. Consistency is assured if you're only using a single `DatabaseClient` configuration.

When working with multi-statement transactions, you should ensure your transaction is rolled back expliciting in the event of an error by including a catch clause that calls `rollback`. For example:

```
const txnObj = null;
db.transactions.open(true).result().
  then(function(response) {
    txnObj = response;
    return db.documents.read({uris: oldUri, txid: txnObj}).result();
    }).
  then(...).
```

```
  catch(function() {
    db.transactions.rollback(txnObj);
    });
```

7.6 Example: Using Promises With a Multi-Statement Transaction

The following function demonstrates how you can use the Promise pattern to synchronize operations within a multi-statement transaction. To learn more about Promises, see "Promise Result Handling Pattern" on page 20.

This function "moves" a document by reading the contents from the initial URI, inserting the contents into the database with the new URI, and then removing the original document. The function initially creates a transaction, then executes the read, write, and remove operations in the context of that transaction. When these operations complete, the transaction is committed. If an error occurs, the transaction is rolled back.

```
function transactionalMove(oldUri, newUri) {
  const txnObj = null;
  db.transactions.open(true).result().
    then(function(response) {
      txnObj = response;
      return db.documents.read({uris: oldUri, txid: txnObj}).result();
      }).
    then(function(documents) {
      documents[0].uri = newUri;
      return db.documents.write(
        {documents: documents, txid: txnObj}).result();
      }).
    then(function(response) {
      return db.documents.remove({uri: oldUri, txid: txnObj}).result();
      }).
    then(function(response) {
      return db.transactions.commit(txnObj).result();
      }).
    catch(function(error) {
      console.log('ERROR: ' + JSON.stringify(error));
      db.transactions.rollback(txnObj);
      });
}
```

7.7 Checking Transaction Status

Use `DatabaseClient.transactions.read` to query the status of a transaction. For example:

```
db.transactions.read(transactionObj)
```

7.8 Managing Transactions When Using a Load Balancer

This section applies only to client applications that use multi-statement transactions and interact with a MarkLogic Server cluster through a load balancer.

When you use a load balancer, it is possible for requests from your application to MarkLogic Server to be routed to different hosts, even within the same session. This has no effect on most interactions with MarkLogic Server, but operations that are part of the same multi-statement transaction need to be routed to the same host within your MarkLogic cluster. This consistent routing through a load balancer is called *session affinity*.

Most load balancers provide a mechanism that supports session affinity. This usually takes the form of a session cookie that originates on the load balancer. The client acquires the cookie from the load balancer, and passes it on any requests that belong to the session. The exact steps required to configure a load balancer to generate session cookies depends on the load balancer. Consult your load balancer documentation for details.

To the load balancer, a session corresponds to a browser session, as defined in RFC 2109 (https://www.ietf.org/rfc/rfc2109.txt). However, in the context of a Node.js Client API application using multi-statement transactions, a session corresponds to a single multi-statement transaction.

> **Note:** To properly preserve session affinity, you must call `DatabaseClient.transactions.open` in a way that returns a transaction object, rather than a simple string transaction id. That is, you must ensure the `withState` parameter (or call object property) is not explicitly set to false. The `transactions.open` function returns an object by default.

The Node.js Client API leverages a session cookie to preserve host affinity across operations in a multi-statement transaction in the following way. This process is transparent to your application; the information is provided to illustrate the expected load balancer behavior.

1. When you create a transaction using `DatabaseClient.transactions.open`, the Node.js Client API receives a transaction id from MarkLogic and, if the load balancer is properly configured, a session cookie from the load balancer. This information is cached in the returned `Transaction` object.

2. Each time you perform a Node.js API operation that includes a `Transaction` object, the Node.js Client API attaches the transaction id and the session cookie to the request(s) it sends to MarkLogic. The session cookie causes the load balancer to route the request to the same host in your MarkLogic cluster that created the transaction.

3. When MarkLogic receives a request, it discards the session cookie (if present), but uses the transaction id to ensure the operation is part of the requested transaction. When MarkLogic responds, the load balancer again adds a session cookie, which the Node.js Client API caches on the `Transaction` object.

4. When you commit or roll back a transaction, any cookies returned by the load balancer are discarded since the transaction is no longer valid. This effectively ends the session from the load balancer's perspective because the Node.js Client API will no longer pass the session cookie around.

Any Node.js Client API operation that does not include a `Transaction` object will not include a session cookie (or transaction id) in the request to MarkLogic, so the load balancer is free to route the request to any host in your MarkLogic cluster.

8.0 Extensions, Transformations, and Server-Side Code Execution

This chapter discusses the following topics related to creating and using extensions and transformations, as well as executing arbitrary blocks of code and library modules on MarkLogic Server using the Node.js Client API:

8.1 Ways to Extend and Customize the API

You can extend and customize the behavior of the Node.js Client API through specific extension points or by initiating execution of arbitrary server-side code from your application.

- Content transformations: A user-defined transform function can be applied when documents are written to the database or read from the database; for details, see “Working with Content Transformations” on page 233. You can also define custom replacement content generators for the patch feature; for details, see “Constructing Replacement Data on MarkLogic Server” on page 104.
- Search result transformations: A user-defined transform function can be applied to the search result summary or matching documents when querying documents and values. For details, see “Working with Content Transformations” on page 233.
- Resource service extensions: Define your own REST endpoints, accessible from Node.js using the `DatabaseClient.resources` interface. Resource service extensions are covered in detail in this chapter. To get started, see “Working with Content Transformations” on page 233.
- Ad-hoc query execution: Send an arbitrary block of XQuery or JavaScript code to MarkLogic Server for evaluation. For details, see “Evaluating Ad-Hoc Code and Server-Side Modules” on page 250.
- Server-side module evaluation: Evaluate user-defined XQuery or JavaScript modules after installing them on MarkLogic Server. For details, see “Evaluating Ad-Hoc Code and Server-Side Modules” on page 250

In addition to these features, the API includes other hooks for server-side user-defined code, such as custom constraint parsers, facet and snippet generators, and document patch content generators. All such code, along with resource service extensions and transforms, must be installed in the modules database associated with your REST API instance before you can use them. The Node.js API includes interfaces for installing these special-purpose assets, as well as any dependent libraries and other assets, through the `DatabaseClient.config` interfaces. For details, see "Overview of Asset Management" on page 258.

8.2 Working with Resource Service Extensions

This section covers the concept of a resource service extension and how to create, install, use, and mange them. The following topics are covered:

8.2.1 What is a Resource Service Extension?

Resource service extensions extend the Node.js Client API by creating a RESTful interface to XQuery and server-side JavaScript modules. The server-side extension implements functions to handle GET, PUT, POST, and DELETE HTTP requests received on the extension by the REST Client API. The Node.js Client API enables you to invoke these methods via the `DatabaseClient.resources` interface. You can wrap your own Node.js interface around the `DatabaseClient.resources` operations to expose the service in a domain-specific way.

For example, you can create a dictionary program resource extension that looks up words, checks spelling, and makes suggestions for unknown words on MarkLogic Server. The individual operations an application programmer may call, for example, `lookUpWords()`, `spellCheck()`, and so on, are the domain-specific services that expose the resource extension.

The following are the basic steps to create and use a resource extension using the Node.js Client API:

1. Create an XQuery or JavaScript module that implements the services for the resource.

2. Install the resource service extension implementation in the modules database associated with the REST API instance using `DatabaseClient.config.resources.write`.

3. Access the resource extension methods using `DatabaseClient.resources` operations such as `DatabaseClient.resources.get`.

The `DatabaseClient.config.resources` interface also supports dynamic discovery of installed extensions. When you install an extension, you can specify metadata, including method parameter name and type information to make it easier to use dynamically discovered extensions. The metadata is purely informational.

If your extension depends on other modules or assets, you can install them in the modules database using `DatabaseClient.extlibs` interface. For details, see "Managing Assets in the Modules Database" on page 257.

For a complete example, see "Example: Installing and Using a Resource Service Extension" on page 228.

8.2.2 Creating a Resource Service Extension

You can implement a resource service Extension using server-side JavaScript or XQuery. The interface is shared across multiple MarkLogic client APIs, so you can use the same extensions with the Java Client API, Node.js Client API, and the REST Client API.

You can install an extension with one client API and use it with all them. For example, you can use a resource service extension installed using the REST Client API with an application implemented using the Node.js Client API and the Java Client API.

For the interface definition, authoring guidelines, and example implementations, see Extending the REST API in the *REST Application Developer's Guide*.

8.2.3 Installing a Resource Service Extension

Before you can use a resource extension, you must install the implementation on MarkLogic Server. You must have the `rest-admin` role or equivalent privileges to install a resource service extension.

Use the following procedure to install your extension implementation:

1. If your resource extension depends on additional library modules, install these dependent libraries on MarkLogic Server. For details, see "Managing Assets in the Modules Database" on page 257.

2. Optionally, define metadata for your extension that describes attributes such as provider, description, and version.

3. Call `DatabaseClient.config.resources.write` to install your extension into the modules database of the REST API instance associated with your `DatabaseClient` object. Your call must provide a name for the extension, the implementation language (XQuery or JavaScript), and the implementation source code. You may include optional metadata.

Note: For an XQuery extension, the extension must be installed under the same name as the name in the extension module namespace declaration. For example, an XQuery extension with the following module namespace must be installed as “example”.

```
xquery version "1.0-ml";
module namespace yourNSPrefix =
  "http://marklogic.com/rest-api/resource/example";
...
```

For example, the following code installs a JavaScript extension under the name “js-example”, without metadata. The extension implementation is streamed from the file `js-example.sjs`.

```
const fs = require('fs');
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.config.resources.write(
  'js-example', 'javascript',
  fs.createReadStream('./js-example.sjs')
).result(function(response) {
  console.log('Installed extension: ' + response.name);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

The following code installs the same extension with a full set of metadata. You need not provide all metadata properties. You must include `name`, `format`, and `source`.

```
const fs = require('fs');
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.config.resources.write({
  name: 'js-example',
  format: 'javascript',
  source: fs.createReadStream('./js-example.sjs'),
  // everything below this is optional metadata
  title: 'Example JavaScript Extension',
  description: 'An example of implementing resource extensions in SJS',
  provider: 'MarkLogic',
  version: 1.0
}).result(function(response) {
  console.log('Installed extension: ' + response.name);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

For a complete example, see “Example: Installing and Using a Resource Service Extension” on page 228.

8.2.4 Using a Resource Service Extension

To invoke the HTTP methods of a resource service extension, use the `DatabaseClient.resources` interface. The interface has a function corresponding to each HTTP verb: `get`, `put`, `post`, and `remove` (DELETE). For example, you can invoke the get method of your extension with no parameters as follows:

```
db.resources.get('js-example')
```

You can also pass in an object that encapsulates parameters expected by the implementation, and a transaction id.

The result of the invocation depends on the method. For example, the GET extension interface enables you return one or more documents, so the result of calling `resources.get` is an object whose stream function can be used to incrementally process the documents in the response.

The following example invokes the `get` function of the resource service extension from Example: JavaScript Resource Service Extension in the *REST Application Developer's Guide*. Three parameters are passed to the implementation, named "a", "b", and "c". The GET implementation of this extension simply echos back the supplied parameters as a JSON document, one document per parameter.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.resources.get({
  name: 'js-example',
  params: { a: 1, b: 2, c: 'three'}
}).result(function(response) {
  console.log(response);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

If the call is successful, the output is similar to the following. The value of the `content` property in each array item is a document returned by the extensions GET method implementation.

```
[ { contentType: 'application/json',
    format: 'json',
    contentLength: '29',
    content: { name: 'c', value: 'three' } },
  { contentType: 'application/json',
    format: 'json',
    contentLength: '25',
    content: { name: 'b', value: '2' } },
  { contentType: 'application/json',
    format: 'json',
    contentLength: '25',
    content: { name: 'a', value: '1' } } ]
```

For details, see “Example: Installing and Using a Resource Service Extension” on page 228.

8.2.5 Example: Installing and Using a Resource Service Extension

This example demonstrates how to install and exercise the GET and PUT methods of the JavaScript extension from Example: JavaScript Resource Service Extension in the *REST Application Developer's Guide*. The extension is usable with any of the MarkLogic client APIs (Node.js, Java, REST).

Use the following procedure to install the extension and exercise the GET method. The GET method of this extension accepts one or more caller-defined parameters and returns a JSON document of the following form for each parameter passed in: `{ "name": param-name, "value": param-value }`.

1. Copy the extension implementation to a file named `js-example.sjs`. For the implementation, see JavaScript Extension Implementation in the *REST Application Developer's Guide*.

2. Install the extension under the name “js-example” by running the following script. You must have the `rest-admin` role or equivalent privileges to install an extension. For details, see “Installing a Resource Service Extension” on page 225.

```
const fs = require('fs');
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.config.resources.write({
  name: 'js-example',
  format: 'javascript',
  source: fs.createReadStream('./js-example.sjs'),
  // everything below this is optional metadata
  title: 'Example JavaScript Extension',
  description: 'An example of implementing resource extensions in SJS',
  provider: 'MarkLogic',
  version: 1.0
}).result(function(response) {
  console.log('Installed extension: ' + response.name);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

3. Optionally, retrieve metadata about the extension using the following script. For details, see “Discovering Resource Service Extensions” on page 231.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.config.resources.read('js-example').result(
```

```
  function(response) {
    console.log(response);
  },
  function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
```

4. Exercise the GET method of the extension by running the following script. For details, see "Using a Resource Service Extension" on page 227.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.resources.get({
  name: 'js-example',
  params: { a: 1, b: 2, c: 'three'}
}).result(function(response) {
  console.log(response);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

The GET method generates a JSON document of the form `{ name: pName, value: pValue}` for each parameter passed in. Thus, the invocation above should generate three documents. The expected output from invoking the GET method is similar to the following:

```
[ { contentType: 'application/json',
    format: 'json',
    contentLength: '29',
    content: { name: 'c', value: 'three' } },
  { contentType: 'application/json',
    format: 'json',
    contentLength: '25',
    content: { name: 'b', value: '2' } },
  { contentType: 'application/json',
    format: 'json',
    contentLength: '25',
    content: { name: 'a', value: '1' } } ]
```

5. Exercise the PUT method of the extension by running the following script.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.resources.put({
  name: 'js-example',
  params: {
    basename: ['one', 'two']},
  documents: [
```

```
    { contentType: 'application/json',
    content: {key1:'value1'} },
    { contentType: 'application/json',
    content: {key2:'value2'} },
  ]
}).result(function(response) {
  console.log(JSON.stringify(response, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

The PUT method of this extensions accepts JSON and XML documents as input. For each input JSON document, a `written` property is added to the document before it is inserted into the database. XML documents are inserted into the database unchanged. The document URIs are derived from a "basename" parameter supplied by the caller. The following is the expected output from invoking PUT method.

```
{ "written": [
    "/extensions/one.json",
    "/extensions/two.json"
] }
```

If you examine the two documents created by the PUT exercise, you can see that a written property has been added. For example, /extensions/one.json has contents similar to the following (the timestamp value will vary):

```
{
  "key1": "value1",
  "written": "08:35:54-08:00"
}
```

To report errors from your implementation to the client, you must use the convention described in "Error Reporting in Extensions and Transformations" on page 247. For example, if you do not pass a "basename" parameter value for each input document, the extension reports an error in the following way:

```
if (docs.count > basenames.length) {
  returnErrToClient(400, 'Bad Request',
    'Insufficient number of uri basenames. Expected ' +
    docs.count + ' got ' + basenames.length + '.');
    // unreachable - control does not return from fn.error
}
```

The error reaches the client application in the following form:

```
{
  "message": "js-example: response with invalid 400 status",
  "statusCode": 400,
  "body": {
    "errorResponse": {
      "statusCode": 400,
```

```
      "status": "Bad Request",
      "messageCode": "RESTAPI-SRVEXERR",
      "message": "Insufficient number of uri basenames. Expected 2 got 1."
    }
  }
}
```

8.2.6 Retrieving the Implementation of a Resource Service Extension

Use `DatabaseClient.config.resources.read` to retrieve the implementation of a resource service extension. You must have the `rest-admin` role or equivalent privileges to use this interface.

For example, the following call retrieves the implementation of the resource service extension installed as “js-example”:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.config.resources.read('js-example').result(
  function(response) {
    console.log(response);
  },
  function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
```

8.2.7 Discovering Resource Service Extensions

You can use `DatabaseClient.config.resources.list` to retrieve the name, interface, and other metadata about installed resource extensions. You must have the `rest-reader` role or equivalent privileges to use this interface.

The amount of information available about a given extension depends on the amount of metadata provided during installation of the extension. The name and methods are always available. Details such as provider, version, and method parameter information are optional.

By default, this request rebuilds the extension metadata each time it is called to ensure the metadata is up to date.

The following example retrieves data about the installed extensions:

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.config.resources.list().result(
function(response) {
  console.log('Installed extensions: ');
  console.log(JSON.stringify(response, null, 2));
}, function(error) {
```

```
  console.log(JSON.stringify(error, null, 2));
});
```

If you installed a single extension named “js-example” with metadata, as shown in “Installing a Resource Service Extension” on page 225, then the output of the above script is similar to the following.

```
{ "resources": {
    "resource": [ {
      "name": "js-example",
      "source-format": "javascript",
      "provider-name": "MarkLogic",
      "title": "Example JavaScript Extension",
      "version": "1",
      "description": "An example of implementing resource extensions in SJS",
      "methods": {
        "method": [
          { "method-name": "get" },
          { "method-name": "post" },
          { "method-name": "put" },
          { "method-name": "delete" }
        ]
      },
      "resource-source": "/v1/resources/js-example"
    }
  ] }
}
```

8.2.8 Deleting Resource Service Extensions

Use `DatabaseClient.config.resources.remove` to remove a resource service extension. You must supply the same name that you used in installing the extension. To remove an extension, you must have the `rest-admin` role or the equivalent privileges.

Deleting an extension is an idempotent operations. That is, you will receive the same response whether the named extension exists or not.

The following code snippet removes the resource service extension named “js-example”.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.config.resources.remove('js-example').result(
  function(response) {
    console.log('Removed extension: ', response.name);
  },
  function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
```

8.3 Working with Content Transformations

This section explains the concept of content transformations and describes how to create, install, apply, and manage transforms. The following topics are covered:

8.3.1 What is a Content Transformation?

The Node.js Client API enables you to create custom content transformations and apply them during operations such as document ingestion and retrieval. For example, you can create a write transform that adds or modifies a JSON property or XML element for each document as it is inserted into the database. The API has hooks for applying the following kinds of transform:

- Write transform: Applied before inserting documents into the database.
- Read transform: Applied when reading documents from the database. You can configure both default and per-request read transforms.
- Search result transform: Applied to the search result summary when you make queries that include a summary instead of just returning matching documents and metadata.

You implement a transform as a server-side JavaScript function, XQuery function, or XSLT stylesheet that accepts a document as input and produces documents as output. Your transform must conform to the interface and guidelines described Writing Transformations in the *REST Application Developer's Guide*. Your transforms can accept transform-specific parameters.

Transforms must be installed in the modules database associated with the REST API instance before you can use them. Use the `DatabaseClient.config.transforms` interface to install and manage your transforms using Node.js. For details, see "Installing a Transformation" on page 234.

You apply a transform by passing its name to supporting operations, such as `DatabaseClient.documents.write`, `DatabaseClient.documents.read` and `DatabaseClient.documents.query`. For details, see "Using a Transformation" on page 235.

8.3.2 Creating a Transformation

You can implement a transform function using server-side JavaScript or XQuery. The interface is shared across multiple MarkLogic client APIs, so you can use the same transforms with the Java Client API, Node.js Client API, and the REST Client API.

Your transform module must include an export named `transform`. For example:

```
function insertTimestamp(context, params, content)
{...}
exports.transform = insertTimestamp;
```

You can install a transform with one client API and use it with all of them. For example, you can use transform installed using the Node.js Client API with an application implemented using the Node.js Client API or the Java Client API.

For the interface definition, authoring guidelines, and example implementations, see Writing Transformations in the *REST Application Developer's Guide*. To return errors from your transform to the client, use the conventions described in "Error Reporting in Extensions and Transformations" on page 247.

For a complete Node.js and JavaScript example, see "Example: Read, Write, and Query Transforms" on page 237.

8.3.3 Installing a Transformation

Use `DatabaseClient.config.transforms.write` to install a transform in the modules database associated with your REST API instance. Using this interface ensures your transform is installed according to the conventions expected by the API, enabling you to subsequently apply and manage the transform with the Node.js Client API.

You must have the `rest-admin` role or equivalent privileges to install a transform.

You can include optional metadata about your transform during installation. The metadata is purely informational. You can retrieve it using `DatabaseClient.config.transforms.list`.

The following script installs a transform under the name "js-transform". The transform implementation is read from a file name "transform.sjs". Only `name`, `format`, and `source` are required. Everything else is optional metadata.

```
const fs = require('fs');
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.config.transforms.write({
  name: 'js-transform',
  format: 'javascript',
  source: fs.createReadStream('./transform.sjs'),
```

```
  // everything below this is optional metadata
  title: 'Example JavaScript Transform',
  description: 'An example of an SJS read/write transform',
  provider: 'MarkLogic',
  version: 1.0
}).result(function(response) {
  console.log('Installed transform: ' + response.name);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

For a complete example, see “Example: Read, Write, and Query Transforms” on page 237.

You can retrieve a list of installed transforms and their metadata using `DatabaseClient.transforms.list`. For details, see “Discovering Installed Transforms” on page 246.

For a complete example, see “Example: Read, Write, and Query Transforms” on page 237.

8.3.4 Using a Transformation

You can specify a transform on document read, write, and query operations such as the following:

- `DatabaseClient.documents.read`
- `DatabaseClient.documents.write` and `DatabaseClient.documents.createWriteStream`
- `DatabaseClient.documents.query`, using `queryBuilder.slice`
- `DatabaseClient.values.read`, using `valuesBuilder.slice`

In all cases, you supply the name of a transform previously installed using `DatabaseClient.config.transforms.write` or the equivalent operation through one of the other client APIs.

For a complete example, see “Example: Read, Write, and Query Transforms” on page 237.

You can only specify one transform per operation. The transform applies to all inputs (`write`) or outputs (`read` or `query`).

To specify a transform to `documents.read` or `documents.write`, add a `transform` property to your call object with one of the following forms. The first two forms are equivalent. Use the third form to pass parameters expected by the transform.

```
transform: transformName

transform: [ transformName ]

transform: [ transformName, {paramName: paramValue, ...} ]
```

For read and write, include the `transform` property as an immediate child of the input call object. For example, if you pass a single document descriptor to `documents.write`, you can include the transform in the descriptor:

```
db.documents.write({
  uri: '/doc/example.json',
  contentType: 'application/json',
  content: { some: 'data' },
  transform: ['js-write-transform']
})
```

By contrast, if you use the multi-document form of input to `documents.write`, include the transform descriptor in the top level object, not inside each document descriptor. For example:

```
db.documents.write({
  documents: [
    { uri: '/transforms/example1.json',
      contentType: 'application/json',
      content: { some: 'data' },
    },
    { uri: '/transforms/example2.json',
      contentType: 'application/json',
      content: { some: 'more data' },
    }
  ],
  transform: ['js-write-transform']
})
```

To apply a transform to the results from `DatabaseClient.documents.query` or `DatabaseClient.values.read`, use a `transform` builder to create a descriptor, and then attach the descriptor to the query through the `slice` clause. For example, the following call applies a transform to a content query:

```
db.documents.query(
  qb.where(
    qb.byExample({writeTimestamp: {'$exists': {}}})
  ).slice(qb.transform('js-query-transform', {a: 1, b: 'two'}))
)
```

The following call applies the same transform (without extra parameters) to a values query:

```
db.values.read(
  vb.fromIndexes('reputation')
    .slice(3,5, vb.transform('js-query-transform'))
)
```

8.3.5 Example: Read, Write, and Query Transforms

This example demonstrates installing and using transforms for read, write, and query operations. The example is designed for you to exercise all three types of transform in sequence, as follows:

1. Install the Transforms
2. Use the Write Transform
3. Use the Read Transform
4. Use the Query Transform

The source code for each transform is provided in the following sections:

- Read Transform Source Code
- Write Transform Source Code
- Query Transform Source Code

8.3.5.1 Install the Transforms

Follow this procedure to install the example read, write, and query transforms. For details, see “Installing a Transformation” on page 234.

1. Create a file named `read-transform.sjs` from the code in “Read Transform Source Code” on page 243.
2. Create a file named `write-transform.sjs` from the code in “Write Transform Source Code” on page 244.
3. Create a file named `query-transform.sjs` from the code in “Query Transform Source Code” on page 245.
4. Copy the following code to a file named `install-transform.js`. This script installs all three transforms.

```
const fs = require('fs');
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

// Descriptors for 3 transforms: Read, write, and query.
const transforms = [
  {
    name: 'js-read-transform',
    format: 'javascript',
    source: fs.createReadStream('./read-transform.sjs'),
      // everything below this is optional metadata
```

```
    title: 'Example JavaScript Read Transform',
    description: 'An example of an SJS read transform',
    provider: 'MarkLogic',
    version: 1.0
  },
  { name: 'js-write-transform',
    format: 'javascript',
    source: fs.createReadStream('./write-transform.sjs')
  },
  {
    name: 'js-query-transform',
    format: 'javascript',
    source: fs.createReadStream('./query-transform.sjs')
  }
];

// Install the transforms
transforms.forEach( function installTransform(transform) {
  db.config.transforms.write(transform).result(
    function(response) {
      console.log('Installed transform: ' + response.name);
    },
    function(error) {
      console.log(JSON.stringify(error, null, 2));
    }
  );
});
```

If installation is successful, you should see results similar to the following:

```
$ node install-transform.js
Installed transform: js-write-transform
Installed transform: js-query-transform
Installed transform: js-read-transform
```

8.3.5.2 Use the Write Transform

The following script writes documents to the database using a write transform. This script demonstrates the usage guidelines from “Using a Transformation” on page 235.

You should already have installed the transform using the instructions in “Install the Transforms” on page 237.

The example transform adds a `writeTimestamp` to the input documents; for details see “Write Transform Source Code” on page 244.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write({
```

```
  documents: [
    { uri: '/transforms/example1.json',
      contentType: 'application/json',
      content: { some: 'data' },
    },
    { uri: '/transforms/example2.json',
      contentType: 'application/json',
      content: { some: 'more data' },
    }
  ],
  transform: ['js-write-transform']
}).result(function(response) {
  response.documents.forEach(function(document) {
    console.log(document.uri);
  });
}, function(error) {
  console.log(JSON.stringify(error));
});
```

If you run the above script, you should see output similar to the following:

```
$ node write.js
/transforms/example1.json
/transforms/example2.json
```

If you use Query Console to inspect the documents in the database, you can see that a `writeTimestamp` property has been added to the content of each one. For example, `/transform/example1.json` should have contents similar to the following:

```
{
  "some": "data",
  "writeTimestamp": "2015-01-02T10:33:39.330483-08:00"
}
```

A transform applies to every document in a write operation. You cannot specify different transforms for each document. As a convenience, if you're only inserting a single document, you can include the transform in the single document descriptor rather than having to build up a `documents` array. For example:

```
db.documents.write({
  uri: '/transforms/example3.json',
  contentType: 'application/json',
  content: { some: 'even more data' },
  transform: ['js-write-transform']
})...
```

You can pass parameters to a transform. For an example, see "Use the Read Transform" on page 240.

8.3.5.3 Use the Read Transform

This section demonstrates applying a read transform, following the usage guidelines from “Using a Transformation” on page 235.

Before running this script, you should have installed the example transforms and run the write transform example. For details, see “Install the Transforms” on page 237 and “Use the Write Transform” on page 238.

The script reads back the documents inserted in “Use the Write Transform” on page 238. A read transform is applied to each document. The transform adds a `readTimestamp` property to the returned documents. The transform also supports adding properties to the output by accepting property name-value pairs as input parameters. The example adds two extra properties to the output documents, `extra1` and `extra2` by specifying the following parameters in the transform descriptor:

```
transform: ['js-read-transform', {extra1: 1, extra2: 'two'}]
```

Copy the following script to a file and run it with the `node` command to exercise the example read transform. To review the transform implementation, see “Read Transform Source Code” on page 243.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read({
  uris: ['/transforms/example1.json', '/transforms/example2.json'],
  transform: ['js-read-transform', {extra1: 1, extra2: 'two'}]
}).stream().on('data', function(document) {
  console.log("URI: " + document.uri);
  console.log(JSON.stringify(document.content, null, 2) + '\n');
}).on('end', function() {
  console.log('Finished');
})
```

If you run the script, you should see output similar to the following.

```
URI: /transforms/example1.json
{
  "some": "data",
  "writeTimestamp": "2015-01-02T10:33:39.330483-08:00",
  "readTimestamp": "2015-01-02T10:44:18.538343-08:00",
  "extra2": "two",
  "extra1": "1"
}

URI: /transforms/example2.json
{
  "some": "more data",
```

```
  "writeTimestamp": "2015-01-02T10:33:39.355913-08:00",
  "readTimestamp": "2015-01-02T10:44:18.5632-08:00",
  "extra2": "two",
  "extra1": "1"
}
```

The `readTimestamp`, `extra1`, and `extra2` properties are added by the read transform. These properties are only part of the read output. The documents in the database are unchanaged. The `writeTimestamp` property was added to the document by the write transform during ingestion; for details see “Use the Write Transform” on page 238.

8.3.5.4 Use the Query Transform

The following script applies a transform to a query operation following the usage guidelines from “Using a Transformation” on page 235.

Before running this script, you should have installed the example transforms and run the write transform example. For details, see “Install the Transforms” on page 237 and “Use the Write Transform” on page 238.

The script below uses a QBE to read back all the documents with a `writeTimestamp` JSON property. This property was previously added to some documents by the write transform in “Use the Write Transform” on page 238.

The script makes two queries, one that returns matching documents and one that just returns a search result summary. When retrieving documents, the transform behaves exactly like the read transform in “Read Transform Source Code” on page 243. That is, it adds a `readTimestamp` property to each document and, optionally, properties corresponding to each input parameter. When retrieving a search result summary as JSON, a `queryTimestamp` property is added to the summary.

Copy the following script to a file and run it using the `node` command in order to demonstrate applying a transform at query time. To review the transform implementation, see “Query Transform Source Code” on page 245.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);
const qb = marklogic.queryBuilder;

// Retrieve just search result summary by setting slice to 0
db.documents.query(
  qb.where(
    qb.byExample({writeTimestamp: {'$exists': {}}})
  ).slice(qb.transform('js-query-transform'))
   .withOptions({categories: 'none'})
).result(function(response) {
  console.log(JSON.stringify(response, null, 2));
```

```
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});

// Retrieve matching documents instead of summary
db.documents.query(
  qb.where(
    qb.byExample({writeTimestamp: {'$exists': {}}})
  ).slice(qb.transform('js-query-transform', {a: 1, b: 'two'}))
).stream().on('data', function(document) {
  console.log("URI: " + document.uri);
  console.log(JSON.stringify(document.content, null, 2) + '\n');
}).on('end', function() {
  console.log('Finished');
});
```

If you run the script, you should see output similar to the following. The bolded properties were added by the transform.

```
$ node query.js
URI: /transforms/example1.json
{
  "some": "data",
  "writeTimestamp": "2015-01-02T10:33:39.330483-08:00",
  "readTimestamp": "2015-01-02T11:09:58.410351-08:00",
  "b": "two",
  "a": "1"
}

URI: /transforms/example2.json
{
  "some": "more data",
  "writeTimestamp": "2015-01-02T10:33:39.355913-08:00",
  "readTimestamp": "2015-01-02T11:09:58.431676-08:00",
  "b": "two",
  "a": "1"
}

Finished
[
  {
    "snippet-format": "snippet",
    "total": 2,
    "start": 1,
    "page-length": 0,
    "results": [],
    "metrics": {
      "query-resolution-time": "PT0.001552S",
      "facet-resolution-time": "PT0.000141S",
      "snippet-resolution-time": "PT0S",
      "total-time": "PT0.165583S"
    },
    "queryTimestamp": "2015-01-02T11:10:00.208708-08:00"
```

```
  }
]
```

Note that the transform specification is part of the `slice` result refinement clause and that a builder (`qb.transform`) is used to construct the transform specification. For example:

```
slice(qb.transform('js-query-transform', {a: 1, b: 'two'}))
```

The syntax for a transform specification is not the same in a query context as for `documents.read` and `documents.write`, so it is best to use the builder.

On query operations, your transform is invoked for all output, whether it is a matched document or a result summary. When you query using the Node.js Client API, the search result summary is always JSON, so you can only distinguish it from matched documents by probing the properties. For example, the query transform does the following to identify the search summary:

```
if (result.hasOwnProperty('snippet-format')) {
  // search result summary
  result.queryTimestamp = fn.currentDateTime();
}
```

If your transform is invoked on behalf of another client API, such as the Java Client API, the results summary can be in XML, and the query can retrieve both documents and a search summary.

8.3.5.5 Read Transform Source Code

The following server-side JavaScript module is meant to be used as a transform on read operations such as `DatabaseClient.documents.read`. This transform adds properties to the output document when you read JSON documents; see the comments in the code for details.

Copy the following code to a file named `read-transform.sjs`. You can use a different filename, but the installation script elsewhere in this section assumes this name.

```
// Example Read Transform
//
// If the input is a JSON document:
// - Add a readTimestamp to the result document.
// - For each parameter passed in by the client, add a
//   property of the form: propName: propValue.
// Other document types are unchanged.
function readTimestamp(context, params, content)
{
  //if (context.inputType.search('json') >= 0) {
  if (context.inputType.search('json') >= 0) {
    const result = content.toObject();

    result.readTimestamp = fn.currentDateTime();

    // Add a property for each caller-supplied request param
```

```
    for (const pname in params) {
      if (params.hasOwnProperty(pname)) {
        result[pname] = params[pname];
      }
    }
    return result;
  } else {
    // Pass thru for non-JSON documents
    return content;
  }
};

exports.transform = readTimestamp;
```

8.3.5.6 Write Transform Source Code

The following server-side JavaScript module is meant to be used as a transform on write operations such as `DatabaseClient.documents.write`. This transform adds properties to any JSON documents you ingest; see the comments in the code for details.

Copy the following code to a file named `write-transform.sjs`. You can use a different filename, but the installation script elsewhere in this section assumes this name.

```
// Example Write Transform
//
// If the input is a JSON document:
// - Add a writeTimestamp to the document.
// - For each parameter passed in by the client, add a property
//   of the form "propName: propValue" to the document.
// Non-JSON documents are returned unmodified.
function writeTimestamp(context, params, content)
{
  if (context.inputType.search('json') >= 0) {
    const result = content.toObject();
    result.writeTimestamp = fn.currentDateTime();

    // Add a property for each caller-supplied request param
    for (const pname in params) {
      if (params.hasOwnProperty(pname)) {
        result[pname] = params[pname];
      }
    }
    return result;
  } else {
    // Pass thru for non-JSON documents
    return content;
  }
};

exports.transform = writeTimestamp;
```

8.3.5.7 Query Transform Source Code

The following server-side JavaScript module is meant to be used as a transform on query operations such as `DatabaseClient.documents.query`. You can also use the read transform in "Read Transform Source Code" on page 243 for this purpose. However, in a query context, your transform is applied to both the matching documents and the generated search summary. For demonstration purposes, this transform distinguishes between the two cases.

This transform adds properties to JSON output, but it distinguishes between the search result summary and matched documents. The transform assumes that JSON input that contains a `snippet-format` property is a search summary, and any other JSON input is a document matching the query.

Copy the following code to a file named `query-transform.sjs`. You can use a different filename, but the installation script elsewhere in this section assumes this name.

```
// When applied to a query operation, a transform is invoked on
// both the search result summary and the matching documents (when
// used as a multi-document read).
//
// The transform does the following:
// - For a JSON search result summary (determined by the presence
//   of a search-snippet property), add a queryTimestamp property.
// - For a JSON document, add a readTimestamp property.
// - For all other input, pass it through unchanged.
function queryTimestamp(context, params, content)
{
  if (context.inputType.search('json') >= 0) {
    const result = content.toObject();
    if (result.hasOwnProperty('snippet-format')) {
      // search result summary
      result.queryTimestamp = fn.currentDateTime();
    } else {
      // JSON document. Add readTimestamp property plus a property
      // for each param passed in by the client.
      result.readTimestamp = fn.currentDateTime();
      for (const pname in params) {
        if (params.hasOwnProperty(pname)) {
          result[pname] = params[pname];
        }
      }
    }
    return result;
  } else {
    // Pass thru for non-JSON documents or XML search summary
    return content;
  }
};

exports.transform = queryTimestamp;
```

8.3.6 Discovering Installed Transforms

You can retrieve the names and metadata for installed transforms using `DatabaseClient.transforms.list`. You must have the `read-reader` role or equivalent privileges to retrieve the list of installed transforms.

The following example retrieves the list of installed transforms and displays the response on the console.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.config.transforms.list().result(
function(response) {
  console.log('Installed transforms: ');
  console.log(JSON.stringify(response, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

If you have installed the transform from “Installing a Transformation” on page 234, then running the above script produces output similar to the following:

```
{ "transforms": {
    "transform": [ {
        "name": "js-transform",
        "source-format": "javascript",
        "title": "Example JavaScript Transform",
        "version": "1",
        "provider-name": "MarkLogic",
        "description": "An example of an SJS read/write transform",
        "transform-parameters": "",
        "transform-source": "/v1/config/transforms/js-transform"
    } ]
} }
```

For additional examples, see `test-basic/documents-transform.js` in the Node.js Client API GitHub project.

8.3.7 Deleting a Transformation

Use `DatabaseClient.transforms.remove` to uninstall a transform on MarkLogic Server. The uninstall operation is idempotent. That is, the results are the same whether or not the named transform is installed.

You must have the `rest-admin` role or equivalent privileges to uninstall a transform.

The following script uninstalls a transform named “js-transform”.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.config.transforms.remove('js-transform').result(
  function(response) {
    console.log('Removed transform: ', response.name);
  },
  function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
```

8.4 Error Reporting in Extensions and Transformations

Extensions and transforms use the same mechanism to report errors to the calling application:

Use `fn.error` (JavaScript) or `fn:error` (XQuery) to raise `RESTAPI-SRVEXERR` and provide additional information in the data parameter. You can control the response status code, status message, and provide an additional error reporting response payload.

If you raise an error in any other way, it is returned to the client application as a 500 Internal Server Error.

See the following topics for examples:

- Example: Reporting Errors in JavaScript
- Example: Reporting Errors in XQuery

8.4.1 Example: Reporting Errors in JavaScript

To return an error to the client application from a JavaScript extension or transform, use `fn.error` to report a `RESTAPI-SRVEXERR` error and provide additional information in the `data` parameter of `fn.error`. You can control the response status code and status message, and provide an additional error reporting response payload. For example, you can return an error to the client in the following way:

```
fn.error(null, 'RESTAPI-SRVEXERR',
  Sequence.from([400, 'Bad Request',
                 'Insufficient number of uri basenames.']));
  // unreachable - control does not return from fn.error
```

The 3rd parameter to `fn.error` should be a sequence of the form `(`*`status-code`*`, '`*`status-message`*`', '`*`payload-format`*`', '`*`response-payload`*`')`. That is, when using `fn.error` to raise `RESTAPI-SRVEXERR`, the `data` parameter to `fn.error` is sequence containing the following items, all optional:

- HTTP status code. Default: 400.
- HTTP status message. Default: Bad Request.

- Response payload. It is best to restrict this to text as the payload may be in JSON or XML, depending on the REST API instance configuration.

  Note: Best practice is to use `RESTAPI-SRVEXERR`. If you report any other error or raise any other exception, it is reported to the calling application as a 500 Server Internal Error.

You can use `xdmp.arrayValues` or `Sequence.from` to construct a sequence from a JavaScript array.

Control does not return from `fn.error`. You should perform any necessary cleanup or other tasks prior to calling it.

You can use a utility function similar to the following to abstract most of the details away from your extension implementation:

```
function returnErrToClient(statusCode, statusMsg, body)
{
  fn.error(null, 'RESTAPI-SRVEXERR',
           Sequence.from([statusCode, statusMsg, body]));
  // unreachable
};
```

The following is an example of using this function:

```
returnErrToClient(400, 'Bad Request',
  'Insufficient number of uri basenames.');
```

If errors from an extension invocation are trapped as follows using the Node.js API:

```
db.resources.put({
  ...
}).result(function(response) {
  console.log(JSON.stringify(response, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

Then the output is similar to the following:

```
{
  "message": "js-example: response with invalid 400 status",
  "statusCode": 400,
  "body": {
    "errorResponse": {
      "statusCode": 400,
      "status": "Bad Request",
      "messageCode": "RESTAPI-SRVEXERR",
      "message": "Insufficient number of uri basenames."
    }
```

```
  }
}
```

For a working example, see “Example: Installing and Using a Resource Service Extension” on page 228.

8.4.2 Example: Reporting Errors in XQuery

Use `fn:error` to report a `RESTAPI-SRVEXERR` error, and provide additional information in the `$data` parameter of `fn:error`. You can control the response status code, status message, and provide an additional error reporting response payload. For example, you can return an error to the client in the following way:

```
fn:error((),"RESTAPI-SRVEXERR",
  (415, "Unsupported Input Type",
   "Only application/xml is supported"))
```

The 3rd parameter to `fn:error` should be a sequence of the form `("`*`status-code`*`", "`*`status-message`*`", "`*`response-payload`*`")`. That is, when using `fn:error` to raise `RESTAPI-SRVEXERR`, the `$data` parameter to `fn:error` is a sequence with the following members, all optional:

- HTTP status code. Default: 400.
- HTTP status message. Default: Bad Request.
- Response payload. It best to limit this to text as the payload can be either JSON or XML, depending on the REST API instance configuration.

  Note: Best practice is to use `RESTAPI-SRVEXERR`. If you report any other error or raise any other exception, it is reported to the calling application as a 500 Server Internal Error.

For example, this resource extension function raises `RESTAPI-SRVEXERR` if the input content type is not as expected:

```
declare function example:put(
    $context as map:map,
    $params  as map:map,
    $input   as document-node()
) as document-node()
{
    (: get 'input-types' to use in content negotiation :)
    let $input-types := map:get($context,"input-types")
    let $negotiate :=
        if ($input-types = "application/xml")
        then () (: process, insert/update :)
        else fn:error((),"RESTAPI-SRVEXERR",
          ("415", "Raven", "nevermore"))
    return document { "Done"}  (: may return a document node :)
};
```

If a PUT request is made to the extension with an unexpected content type, the `fn:error` call causes the request to fail with a status 415 and to include the additional error description in the response body:

```
HTTP/1.1 415 Raven
Content-type: application/xml
Server: MarkLogic
Set-Cookie: SessionID=714070bdf4076536; path=/
Content-Length: 62
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<word>nevermore</word>
```

8.5 Evaluating Ad-Hoc Code and Server-Side Modules

You can use `DatabaseClient.eval` or `DatabaseClient.xqueryEval` to evaluate ad-hoc blocks of XQuery or server-side JavaScript code on MarkLogic Server. The code blocks originate in your client application. You can use `DatabaseClient.invoke` to evaluate previously installed XQuery or server-side JavaScript modules on MarkLogic Server.

This section covers the following topics related to using eval and invoke:

- Required Privileges
- Evaluating a Ad-Hoc Query
- Invoking a Module Installed on MarkLogic Server
- Interpreting the Results of Eval or Invoke
- Specifying External Variable Values

8.5.1 Required Privileges

Using `DatabaseClient.eval`, `DatabaseClient.xqueryEval`, and `DatabaseClient.invoke` requires additional privileges, beyond those required for normal read/write/query operations using the Node.js Client API.

To use `DatabaseClient.eval` or `DatabaseClient.xqueryEval`, you must have at least the following privileges or their equivalent:

- `http://marklogic.com/xdmp/privileges/xdmp-eval`
- `http://marklogic.com/xdmp/privileges/xdmp-eval-in`
- `http://marklogic.com/xdmp/privileges/xdbc-eval`
- `http://marklogic.com/xdmp/privileges/xdbc-eval-in`

To use `DatabaseClient.invoke`, you must have at least the following privileges or their equivalent:

- `http://marklogic.com/xdmp/privileges/xdmp-invoke`
- `http://marklogic.com/xdmp/privileges/xdmp-invoke-in`
- `http://marklogic.com/xdmp/privileges/xdbc-invoke`
- `http://marklogic.com/xdmp/privileges/xdbc-invoke-in`

The privileges listed above merely make it possible to eval/invoke server-side code. The operations performed by that code may require additional privileges.

8.5.2 Evaluating a Ad-Hoc Query

Use `DatabaseClient.eval` to evaluate an ad-hoc block of JavaScript on MarkLogic Server. You must use the MarkLogic server-side JavaScript dialect described in the *JavaScript Reference Guide*. To evaluate an ad-hoc block of XQuery, use `DatabaseClient.xqueryEval`. The calling and response conventions are the same for both eval and `xqueryEval`. These operations are equivalent to using the `xdmp.eval` (JavaScript) or `xdmp:eval` (XQuery) builtin function. The code is evaluated in the context of the database associated with the `DatabaseClient` object.

Using `eval` or `xqueryEval` requires extra security privileges; for details, see "Required Privileges" on page 250.

You can call `eval` and `xqueryEval` using one of the following forms. The code is the only required parameter/property.

```
db.eval(codeAsString, externalVarsObj)
db.eval({source: codeAsString, variables: externalVarsObj, txid:...})

db.xqueryEval(codeAsString, externalVarsObj)
db.xqueryEval({
  source: codeAsString,
  variables: externalVarsObj,
  txid:...})
```

External variables enable you to pass variable values to MarkLogic Server, where they're substituted into your ad-hoc code. For details, see "Specifying External Variable Values" on page 256.

For example, suppose you want to evaluate the following JavaScript code, where `word1` and `word2` are external variable values supplied by your application:

```
word1 + " " + word2
```

Then the following call evaluates the code on MarkLogic Server. The values for `word1` and `word2` are passed to MarkLogic Server through the second parameter.

```
db.eval('word1 + " " + word2', {word1: 'hello', word2: 'world'})
```

The response from calling eval is an array containing an item for each value returned by the code block. Each item contains the returned value, plus type information to help you interpret the value. For details, see “Interpreting the Results of Eval or Invoke” on page 255.

For example, the above call returns the following response.

```
[{
  "format": "text",
  "datatype": "string",
  "value": "hello world"
}]
```

You can return documents, objects, and arrays as well as atomic values. To return multiple items, you must return either a `Sequence` (JavaScript only) or a sequence. You can construct a `Sequence` from an array-like or generator, and many builtin functions return multiple values return a `Sequence`. To construct a sequence in server-side JavaScript, apply `xdmp.arrayValues` or `Sequence.from` to a JavaScript array.

For example, to extend the previous example to return the combined lenght of the two input values as well as the concatenated string, accumulate the results in an array and then apply `Sequence.from` to the array:

```
Sequence.from([word1.length + word2.length, word1 + " " + word2])
```

The following script evaluates the above code. The response contains 2 array items: One for the length and one for concatenated string.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.eval(
   'Sequence.from([word1.length + word2.length, word1 + " " + word2])',
   {word1: 'hello', word2: 'world'}
).result(function(response) {
    console.log(JSON.stringify(response, null, 2));
}, function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

Running the script produces the following output:

```
[
  {
    "format": "text",
    "datatype": "integer",
    "value": 10
  },
  {
    "format": "text",
```

```
    "datatype": "string",
    "value": "hello world"
  }
]
```

The following script uses `DatabaseClient.xqueryEval` to evaluates a block XQuery that performs the same operations as the previous JavaScript eval. The output is exactly as before. Note that in XQuery you must explicitly declare the external variables in your ad-hoc code.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.xqueryEval(
  'xquery version "1.0-ml";' +
  'declare variable $word1 as xs:string external;' +
  'declare variable $word2 as xs:string external;' +
  '(fn:string-length($word1) + fn:string-length($word2),' +
  ' concat($word1, " ", $word2))',
  {word1: 'hello', word2: 'world'}
).result(function(response) {
    console.log(JSON.stringify(response, null, 2));
}, function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

For more examples, see `test-basic/server-exec.js` in the Node.js Client API source project on GitHub.

8.5.3 Invoking a Module Installed on MarkLogic Server

You can use `DatabaseClient.invoke` to an XQuery or server-side JavaScript module installed on MarkLogic Server. This is equivalent to calling the builtin server function `xdmp.invoke` (JavaScript) or `xdmp:invoke` (XQuery). Using invoke requires extra security privileges; for details, see “Required Privileges” on page 250.

The module you invoke must already be installed on MarkLogic Server. You can install your module in the modules database associated with your REST API instance using `DatabaseClient.config.extlibs.write` or an equivalent operation. For details, see “Managing Assets in the Modules Database” on page 257.

> **Note:** Installing a module using `DatabaseClient.config.extlibs.write` adds a `/ext/`. prefix to the path. Omit the prefix when using the `config.extlibs` interface, but include it in your module path when calling `invoke`.

When installing the module, you must include the module path, content type, and source code. For a JavaScript module, set the content type to `application/vnd.marklogic-javascript` and set the file extension in your module path to `.sjs`. For an XQuery module, set the content type to `application/xquery` and set the file extension in your module path to `.xqy`. See the example below.

You can use external variables to pass arbitrary values to your module at runtime. For details, see “Specifying External Variable Values” on page 256.

The response to invoke is an array containing one item for each value returned by the invoked module. For details, see “Interpreting the Results of Eval or Invoke” on page 255.

The following example installs a JavaScript module on MarkLogic Server and then uses `DatabaseClient.invoke` to evaluate it.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

// (1) Install the module in the modules database
//     Note: You do not need to install on every invocation.
//     It is included here to make the example self-contained.
db.config.extlibs.write({
  path: '/invoke/example.sjs',
  contentType: 'application/vnd.marklogic-javascript',
  source: 'Sequence.from([word1, word2, word1 + " " + word2])'
}).result().then(function(response) {
  console.log('Installed module: ' + response.path);

  // (2) Invoke the module
  return db.invoke({
    path: '/ext/' + response.path,
    variables: {word1: 'hello', word2: 'world'}
  }).result(function(response) {
    console.log(JSON.stringify(response, null, 2));
  }, function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
}, function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

If you save the script to a file and run it, you should see results similar to the following:

```
[
  {
    "format": "text",
    "datatype": "string",
    "value": "hello"
  },
  {
    "format": "text",
    "datatype": "string",
    "value": "world"
  },
  {
    "format": "text",
    "datatype": "string",
```

```
    "value": "hello world"
  }
]
```

To install an equivalent XQuery module, use a call similar to the following:

```
db.config.extlibs.write({
  path: '/invoke/example.xqy',
  contentType: 'application/xquery',
  source:
    'xquery version "1.0-ml";' +
    'declare variable $word1 as xs:string external;' +
    'declare variable $word2 as xs:string external;' +
    '($word1, $word2, fn:concat($word1, " ", $word2))'
})
```

8.5.4 Interpreting the Results of Eval or Invoke

When you evaluate or invoke server-side code using `DatabaseClient.eval`, `DatabaseClient.xqueryEval`, or `DatabaseClient.invoke`, the response is always an array containing an item for each value returned by the server.

Each item contains information that helps your application interpret the value. Each item has the following form, where `format` and `value` are always present, but `datatype` is not.

```
{
  format: 'text' | 'json' | 'xml' | 'binary'
  datatype: string
  value: ...
}
```

The datatype property can be a node type, an XSD datatype, or any other server type, such as `cts:query`. The reported type may be more general than the actual type. Types derived from `anyAtomicType` include `anyURI`, `boolean`, `dateTime`, `double`, and `string`. For details, see http://www.w3.org/TR/xpath-functions/#datatypes.

The table below summarizes how the representation of the data in the value property is determined.

format	datatype	value Representation
json	node()	A parsed JavaScript object or array
text	any atomic type	A JavaScript boolean, number, or null value, if datatype permits conversion from string; otherwise, a string value. For example, if datatype is integer, then value is a number.
xml	node()	string
binary		a Buffer object

For example, an atomic value (anyAtomicType, a type derived from anyAtomicType, or an equivalent JavaScript type) has a datatype property that can specify an explicit type such as integer, string, or date.

If your code or module returns JSON (or a Javascript object or array), then value is a parsed JavaScript object or array. For example:

```
db.eval('const result = {number: 42, phrase: "hello"}; result;')

==>
[ { format: 'json',
    datatype: 'node()',
    value: { number: 42, phrase: 'hello' }
} ]
```

8.5.5 Specifying External Variable Values

You can pass values to an ad-hoc query (or invoked module) at runtime using external variables. Specify external variables to your eval and invoke calls using a JavaScript object of the following form. The values must be JavaScript primitives.

```
{ varName1: varValue1, varName2: varValue2, ... }
```

For example, the following object supplies values for two external variables, named word1 and word2:

```
{ word1: 'hello', word2: 'world' }
```

If you're evaluating or invoking XQuery code, you must declare the variables explicitly in the ad-hoc query or module. For example, the folloiwng prolog declares two external variables whose values can be supplied by the above parameter object:

```
xquery version "1.0-ml";
declare variable $word1 as xs:string external;
declare variable $word2 as xs:string external;
...
```

If you're evaluating or invoking XQuery code that depends on variables in a namespace, use Clark notation on the variable name. That is, specify the name using notation of the form `{namespaceURI}name`.

For example, the following script uses a namspace qualified external variable, `$my:who`. The external variable input parameter uses the fully qualified variable in Clark notation: `{'{http://example.com}who':'world'}`.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');
const db = marklogic.createDatabaseClient(my.connInfo);

db.xqueryEval(
  'xquery version "1.0-ml";' +
  'declare namespace my = "http://example.com";' +
  'declare variable $my:who as xs:string external;' +
  'fn:concat("hello ", $my:who)',
  {'{http://example.com}who' : 'world'}
).result(function(response) {
    console.log(JSON.stringify(response, null, 2));
}, function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

8.6 Managing Assets in the Modules Database

Use the DatabaseClient.config.extlibs interface to install and manage server-side assets required by your application, such as XQuery and JavaScript modules usable with `DatabaseClient.eval` and dependent libraries used by resource service extensions and tranforms.

This section covers the following topics:

- Overview of Asset Management
- Installing or Updating an Asset
- Referencing an Asset from Server-Side Code
- Removing an Asset
- Retrieving an Asset List
- Retrieving an Asset

8.6.1 Overview of Asset Management

Your Node.js Client API application can use several kinds of user-defined code that is stored in the modules database associated with your REST API instance, including transforms, resource service extension implementations, constraint binding parsers, custom snippet generators, and patch content generators.

Most of these asset classes have specialized management interfaces, such as `DatabaseClient.config.resources` and `DatabaseClient.config.query.snippet`. These interfaces abstract away the details of where and how the API manages the assets. You generally should not manage such assets through another, more general interface. Assets which do not have a specialized interface can be managed using the `DatabaseClient.config.extlibs` interface.

The table below summarizes the asset management interfaces available through the Node.js Client API.

Interface	Used to Manage
`DatabaseClient.config.extlibs`	XQuery and JavaScript modules that can be invoked using `DatabaseClient.invoke`. For details, see “Invoking a Module Installed on MarkLogic Server” on page 253. Dependent libraries and other assets needed by your resource service extensions, transforms. For details, see “Working with Resource Service Extensions” on page 224.
`DatabaseClient.config.patch.replace`	Replacement content generators for DatabaseClient.documents.patch. For details, see “Constructing Replacement Data on MarkLogic Server” on page 104.

Interface	Used to Manage
`DatabaseClient.config.query.custom`	Custom query binding and facet generators. For details, see “Using a Custom Constraint Parser” on page 131 and “Generating Search Facets” on page 161.
`DatabaseClient.config.query.snippet`	Custom snippet generators. For details, see “Generating Search Snippets” on page 168.
`DatabaseClient.config.resources`	Resource service extensions. For details, see “Working with Resource Service Extensions” on page 224.
`DatabaseClient.config.transforms`	Read, write and query transforms. For details, see “Working with Content Transformations” on page 233.

All the asset management interfaces offer the same basic set of methods, customized to suit a given asset classet:

- `write`: Install an asset in the modules database.
- `read`: Retrieve an asset from the modules database.
- `list`: Retrieve a list of all assets of a given class from the modules database, such as all resource service extensions or all facet generators.
- `remove`: Remove an asset from the modules database.

You should not mix and match interfaces among asset classes. For example, you should not install a snippeter using `DatabaseClient.config.query.snippet.write` and then delete it using `DatabaseClient.config.extlibs.remove`. You can manage assets through the equivalent interfaces of the other client APIs, such as the Java Client API and the REST Client API.

When you install or update an asset in the modules database, the asset is replicated across your cluster automatically. There can be a delay of up to one minute between update and availability.

MarkLogic Server does not automatically remove dependent assets when you delete the related extension or transform.

Since dependent assets are installed in the modules database, they are removed when you remove the REST API instance if you include the modules database in the instance teardown. For details, see Removing an Instance in the *REST Application Developer's Guide*.

8.6.2 Installing or Updating an Asset

This section describes how to install or update an asset that is not covered by a specialized asset management interface, such as a dependent library or a module to be invoked using `DatabaseClient.invoke`. For other asset classes, use the `write` method of the specialized interface. For a list of the specialized interfaces, see "Overview of Asset Management" on page 258.

Use `DatabaseClient.config.extlibs.write` to install or update an asset in the modules database associated with your REST API instance. You must provide a module path, content type, and the asset contents. You can insert assets into the modules database as JSON, XML, text, or binary documents. MarkLogic Server determines the document format. The document type is determined by the content type or the module path URI file extension and the server MIME type mappings.

The module path you provide is prepended with `/ext/` during installation. You can omit the prefix when manipulating the asset using the `extlibs` interface, but you should include when you reference the module elsewhere, such as in a resource service extension `require` statement that uses an absolute path or when invoking a module with using `DatabaseClient.invoke`.

The following example installs a module whose contents are read in from a file. The module is installed in the modules database with the URI `/ext/extlibs/example.sjs`.

```
const fs = require('fs');
const marklogic = require('marklogic');
const db = marklogic.createDatabaseClient(my.connInfo);

...
db.config.extlibs.write({
  path: '/extlibs/example.sjs',
  contentType: 'application/vnd.marklogic-javascript',
  source: fs.createReadStream('./example.sjs')
})...
```

For additional examples, see "Invoking a Module Installed on MarkLogic Server" on page 253 or `test-basic/extlibs.js` in the `marklogic/node-client-api` project on GitHub.

8.6.3 Referencing an Asset from Server-Side Code

To use a dependent library installed with `DatabaseClient.extlibs.write` from an extension, transform, or invoked module, use the same URI under which you installed the dependent library, including the `/ext/` prefix.

For example, if a dependent asset is installed with using `db.config.extlibs.write({path: '/my/domain/lib/`*`myasset`*`', ...})`, then its URI in the modules database is `/ext/my/domain/`*`myasset`*.

A JavaScript extension, transform, or invoked module using this asset can refer to it as follows:

```
const myDep = require('/ext/my/domain/lib/myasset');
```

An XQuery extension, transform, or invoked module using this library can include an import of the following form:

```
import module namespace dep="mylib" at "/ext/my/domain/lib/myasset";
```

8.6.4 Removing an Asset

Use `DatabaseClient.config.extlibs.remove` to delete an asset from the modules database if it was installed using `DatabaseClient.config.extlibs.write`. For assets with specialized interfaces, such as extensions and tranforms, use the `remove` method of the specialized interface, such as `DatabaseClient.config.resources.remove`.

Removing an asset is an idempotent operation. That is, it returns the same response whether the asset exists or not.

To remove all the assets in a given directory, supply the containing directory name instead of a specific asset path.

For example, if an asset is installed as follows:

```
db.config.extlibs.write({
  path: '/invoke/example.sjs',
  contentType: 'application/vnd.marklogic-javascript',
  source: ...
})
```

Then you can remove that single asset with a call similar to the following:

```
db.config.extlibs.remove('/invoke/example.sjs');
```

To remove all the assets installed under /ext/invoke/ instead, use a call similar to the following:

```
db.config.extlibs.remove('/invoke/');
```

8.6.5 Retrieving an Asset List

Use `DatabaseClient.config.extlibs.list` to retrieve a list of assets installed using `DatabaseClient.config.extlibs.write`. For assets with specialized interfaces, such as extensions and tranforms, use the `list` method of the specialized interface, such as `DatabaseClient.config.transforms.list`.

The response has the following format:

```
{ "assets": [
    { "asset": "/ext/invoke/example.sjs" },
    { "asset": "/ext/util/dep.sjs" },
    { "asset": assetModulePath }, ...
]}
```

8.6.6 Retrieving an Asset

Use `DatabaseClient.config.extlibs.read` to retrieve an asset installed using `DatabaseClient.config.extlibs.write`. For assets with specialized interfaces, such as extensions and tranforms, use the `read` method of the specialized interface, such as `DatabaseClient.config.transforms.read`.

Retrieve the asset using the same module path you used to install it. For example:

```
db.config.extlibs.read('/invoke/example.sjs')
```

9.0 Administering REST API Instances

The Node.js Client API requires a REST Client API instance on MarkLogic Server in order to communicate with the server and access the database. This chapter describes how to create and manage an instance.

The following topics are covered:

9.1 What Is a REST API Instance?

The Node.js Client API implementation communicates with MarkLogic Server using the REST Client API described in *REST Application Developer's Guide*. Therefore, requests to MarkLogic Server through the Node.js Client API require the presence of a *REST API instance*. A REST API instance consists of an HTTP App Server specially configured to handle REST Client API requests, a default content database, and a modules database.

> **Note:** Each REST API instance can host a single application. If you have multiple REST API applications, you must create an instance for each one, and each one must have its own modules database.

When you install MarkLogic Server, a pre-configured REST API instance is available on port 8000. This instance is available as soon as you install MarkLogic Server. No further setup is required. This instance uses the Documents database as the default content database and the Modules database as the modules database.

The instance on port 8000 is convenient for getting started, but you will usually create a dedicated instance for production purposes. This chapter covers creating and managing your own instance.

When you use `marklogic.createDatabaseClient` to create a `DatabaseClient` object, you're creating a connection to a REST API instance. When you create the `DatabaseClient`, you can specify a content database other than the default content database associated with the instance. Using an alternative database requires extra security privileges. For details, see "Evaluating Requests Against a Different Database" on page 16.

The default content database associated with a REST API instance can be created for you when the instance is created, or you can create it separately before making the instance. You can associate any content database with an instance. Administer your content database as usual, using the Admin Interface, XQuery or JavaScript Admin API, or REST Management API.

The REST instance modules database can be created for you during instance creation, or you can create it separately before making the instance. If you choose to pre-create the modules database, it must not be shared across instances. Special code is inserted into the modules database during instance creation. The modules database also holds any persistent query options, extensions, content transformations, custom parsers, and other assets installed using the `DatabaseClient.config` interfaces.

Aside from the instance properties described in this chapter, you cannot pre-configure the App Server associated with an instance. However, once the instance is created, you can further customize properties such as request timeouts using the Admin Interface, XQuery or JavaScript Admin API, or REST Management API.

9.2 Creating an Instance

When you install MarkLogic Server, a pre-configured REST API instance is available on port 8000. However, you can create your own instance using the REST Client API.

To create a new REST instance, send a POST request to the `/rest-apis` service on port 8002 with a URL of the form:

```
http://host:8002/version/rest-apis
```

You can use either the keyword LATEST or the current version for *version*. The POST body should contain a JSON or XML instance configuration. The configuration must include at least a name, but can also include a port number, content and modules database name, and other instance properties.

For example, the following command uses the cURL command line tool to create an instance named “RESTstop” using the defaults for port, databases, and properties.

```
curl --anyauth --user user:password -X POST -i \
  -d '{"rest-api": {"name":"RESTstop" }}' \
  -H "Content-type: application/json" \
  http://localhost:8002/LATEST/rest-apis
```

For details and examples, see Creating an Instance in the *REST Application Developer’s Guide*. For an example of creating an instance using Node.js libraries, see `etc/test-setup.js` in the Node.js Client API source code project on GitHub.

9.3 Configuring Instance Properties

Several instance properties can be examined and modified after you create an instance. For example, you can use the `document-transform-out` property to specify a default read transform.

Use `DatabaseClient.config.serverprops` interface to read and write instance properties. You must have the `rest-admin` role or equivalent privileges to use this interface. For a description of the available properties, see Instance Configuration Properties in the *REST Application Developer’s Guide*.

To retrieve the current configuration, use the `read` method. The response is an object that contains all the properties. For example:

```
db.config.serverprops.read()
```

To set properties, use the write method and pass in an object that contains an object property for each instance property you want to change. The object returned by read is suitable as input to write.

```
db.config.serverprops.write(props)
```

For example, the following script reads the current instance properties, uses the result to toggle the value of the `debug` property, then sets it back to its original value using a property descriptor that only contains the `debug` setting.

```
const marklogic = require('marklogic');
const my = require('./my-connection.js');

const db = marklogic.createDatabaseClient(my.connInfo);

db.config.serverprops.read().result()
  .then(function(props) {
    console.log("Current instance properties:");
    console.log(props);
    // flip the debug property setting
    props.debug = !props.debug;
    return db.config.serverprops.write(props).result();
  }).then(function(response) {
    console.log("Props updated: " + response);
    // demonstrate the setting changed
    return db.config.serverprops.read().result();
  }).then(function(props) {
    console.log("Debug setting is now: " + props.debug);
    // flip the setting back using sparse properties
    const newProps = {};
    newProps.debug = !props.debug;
    return db.config.serverprops.write(newProps).result();
  }).then(function(response) {
    return db.config.serverprops.read().result();
  }).then(function(props) {
    console.log("Debug setting is now: " + props.debug);
  });
```

If you run the script, you should see output similar to the following. Your instance property values may differ.

```
{ 'content-versions': 'none',
  'validate-options': true,
  'document-transform-out': '',
  debug: false,
  'document-transform-all': true,
```

```
  'update-policy': 'merge-metadata',
  'validate-queries': false }
Props updated: true
Debug setting is now: true
Debug setting is now: false
```

9.4 Retrieving Configuration Information

You can use a GET request on the `/rest-apis` service on port 8002 to retrieve configuration information about all REST API instances on a host, or about a specific instance that you identify by instance name or content database.

For details, see Retrieving Configuration Information in the *REST Application Developer's Guide*.

9.5 Removing an Instance

To remove an instance of the REST Client API, send a DELETE request to the `/rest-apis` service on port 8002. You can choose wehther or not to leave the content database intact.

Warning You usually should not apply this procedure to the pre-configured REST API instance on port 8000. Doing so can disable other services on that port, including XDBC, Query Console, and the REST Management API.

For details and examples, see Removing an Instance in the *REST Application Developer's Guide*.

10.0 Creating Data Services and Developer Actions in Node.js

Data Services is a convenient way to integrate MarkLogic into an existing enterprise environment. A data service is a fixed interface over the data managed in MarkLogic expressed in terms of the consuming application. Data services can run queries ("Find eligible insurance plans for an applicant"), updates ("Flag this claim as fraudulent"), or both ("Adjust the rates of plans that haven't made claims in the last year"). A MarkLogic cluster can support dozens or even hundreds of different data services operating over the data and metadata managed in a data hub.

As with Java generalists, a Node.js generalist and a MarkLogic Server expert need the ability to collaborate based on contractual data services that are implemented close to the data as service endpoints and are callable as functions in client programs. In particular:

Table 1:

Given	A proxy service directory with paired function declarations and SJS, MJS, or XQuery endpoint implementations that has been loaded into the modules database on the enode.
When	A developer uses MarkLogic tooling to generate a Node.js module.
Then	The developer can call the methods provided by the module to execute the enode endpoints.

Note: The initial release doesn't provide a way to generate stubs for endpoint modules or to load the proxy service directory into the modules database. The MarkLogic expert can use the Gradle tasks provided by the Java API for those tasks.

To learn how to create and deploy a Data Services proxy service, follow the instructions in the Creating the Proxy Service Directory section of our Introduction to the Java API. Once you have created the Data Service Proxy, you may return to this chapter to learn how you use Gulp to generate modules as well as how to call them from your Node.js client application.

The following topics are covered:

10.1 Node.js Annotations for Declarations

The service and function declarations can take the following annotations to affect the generated module:

Table 2:

Declaration	Annotation	Purpose
service.json	$jsModule	Supplies a name for the generated Node.js module to use instead of the default (the directory name).
*.api	$jsOutputMode	Specifies whether to get return values as a Promise (the default) or as a Stream of data objects.
*.api	$jsType	Specifies a mapping to a JavaScript data type other than the default for a return value

The $jsType annotation supports the mappings listed below for return values.

Table 3:

Declared data type of return value	Mappable JavaScript types
boolean	boolean (default) string
dateTime	Date string (default)
float, int, or unsignedInt	number (default) string

The $jsType annotation is not supported for other server data types or for parameters.

Parameters accept and implicitly convert JavaScript input values to server data types as follows:

Table 4:

Declared data types of parameter	Convertible JavaScript types for values
boolean	boolean or string
dateTime	Date or string
numeric atomic data types (such as double or int)	number or string
all other atomic data types	string
array	array, Buffer, Set, Readable stream, or string

Table 4:

binaryDocument	Buffer or Readable stream
jsonDocument	array, Buffer, Map, object, Set, Readable stream, or string
object	Buffer, Map, object literals, Readable stream, or string
textDocument	Buffer, Readable stream, or string
xmlDocument	Buffer, Readable stream, or string

The declarations can also contain $java* annotations, which are ignored in the Node.js environment. Similarly, the $js* annotations are ignored in the Java environment.

10.2 Using Gulp to Generate Models

Gulp provides a tool for build task execution in Node.js environments that's comparable Gradle in Java environments.

The Node.js API makes it easy to write Gulp pipelines that generate proxy modules. The proxy-generator.js module provides a generate() factory function, which return a Node.js Transform that takes the Gulp Vinyl descriptor for a proxy service directory as input and produces the Gulp Vinyl wrapper for a generated proxy module as output.

Following the Gradle project convention (where ml-modules/root/ds contains proxy service directories), the example below of a gulpfile.js pipes proxy service directories to the proxy module generator and then pipes the generated proxy modules to the lib directory for the project:

```
'use strict';

const gulp  = require('gulp');

const proxy = require('marklogic/lib/proxy-generator.js');


function proxygen() {

  return gulp.src('ml-modules/root/ds/*')

      .pipe(proxy.generate())

      .pipe(gulp.dest('lib/'));

}

exports.proxygen = proxygen;
```

The following Gulp command command is used to process this gulfile.js:

```
gulp proxygen
```

The Node.js client modules can use require() to import the generated proxy modules in the usual way and then construct proxy objects and call the methods of the proxy objects to invoke the server endpoints.

10.3 Generated Modules

Similar to the Interface generated for Java, the JavaScript module generated for Node.js defines and exports a JavaScript class with the following characteristics:

- A constructor that takes a database client with the host, port, and authentication for the server as well as an optional service declaration (where this optional service declaration specifies a directory in the modules database with custom endpoints that implement the interface for the service).
- A static on() method that takes the database client and optional service declaration parameters of the constructor and provides a convenience for instantiating the class.
- A method for each function declared in the proxy service directory.
- If any of those functions takes a session parameter, a createSession() method for constructing a ServiceState argument.
- JSDoc documentation comments for the class and each method for generating reference documentation where the content of the JSDoc comments uses the desc properties from the service declaration and *.api function declarations.

10.4 Expected Pattern of Use

The following listings show a generic example of how a developer might write code to use a Data Service method available on a service you have defined:

```
// Import the marklocig utilities

const marklogic = require('marklogic');



// import the generated class

const GENERATED_CLASS = require("./lib/GENERATED_CLASS.js");



// create a client for the host and port as the user

const client =
```

```
marklogic.createDatabaseClient({host:THE_HOST, port:THE_PORT,
user:THE_USER, password:THE_PASSWORD});

// construct an instance of the class

const theInstance = GENERATED_CLASS.on(client);



// call a method of the instance to execute an endpoint on the
server

const theOutput = theInstance.theMethod(...parameters...);
```

The method returns either a JavaScript Promise or a stream for the output depending on the $jsOutputMode annotation (as described earlier).

Note: By providing the class instead of providing only a factory for instances, we make it possible to use the class in instanceof tests.

11.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement.

We invite you to visit our support website at http://help.marklogic.com to access information on known and fixed issues, knowledge base articles, and more. For licensed customers with an active maintenance contract, see the Support Handbook for instructions on registering support contacts and on working with the MarkLogic Technical Support team.

Complete product documentation, the latest product release downloads, and other useful information is available for all developers at http://developer.marklogic.com. For technical questions, we encourage you to ask your question on Stack Overflow.

12.0 Copyright

MarkLogic Server 10.0 and supporting products.
Last updated: January 8, 2020

COPYRIGHT

This technology is protected by U.S. Patent No. 7,127,469B2, U.S. Patent No. 7,171,404B2, U.S. Patent No. 7,756,858 B2, and U.S. Patent No 7,962,474 B2, US 8,892,599, and US 8,935,267.

The MarkLogic software is protected by United States and international copyright laws, and incorporates certain third party libraries and components which are subject to the attributions, terms, conditions and disclaimers set forth below.

For all copyright notices, including third-party copyright notices, see the Combined Product Notices for your version of MarkLogic.