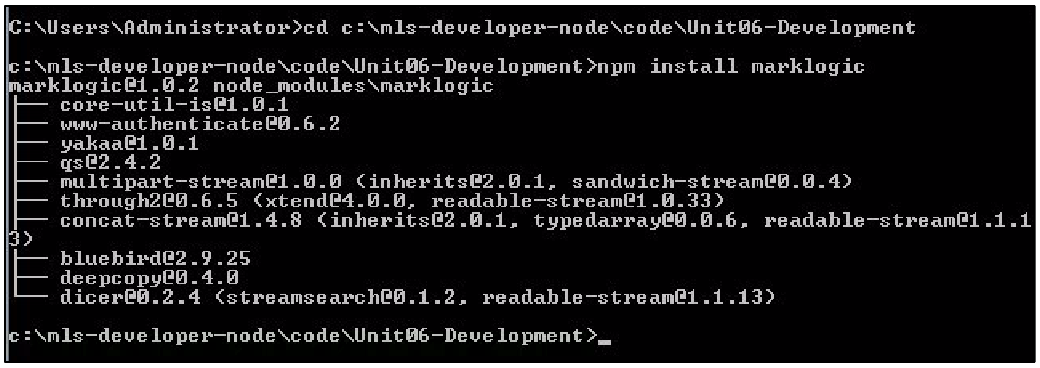
# Unit 6

## Loading Content

## Exercise 1: Load a Document from Memory

In this exercise you will implement code that will load a document from memory into the database, and then read that document back from the database, outputting the information into the console.

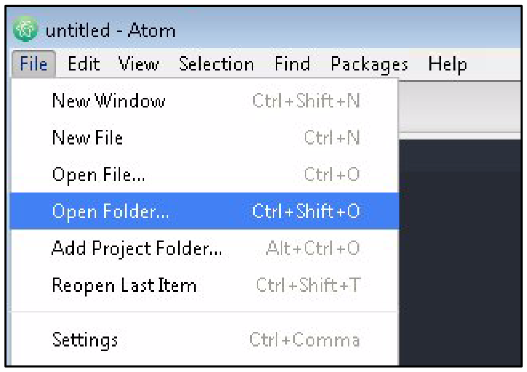
1. From the command line, navigate to the **c:\mls-developer-node\code\Unit06-Development** directory and run **npm install marklogic**. When the install is complete you will see:



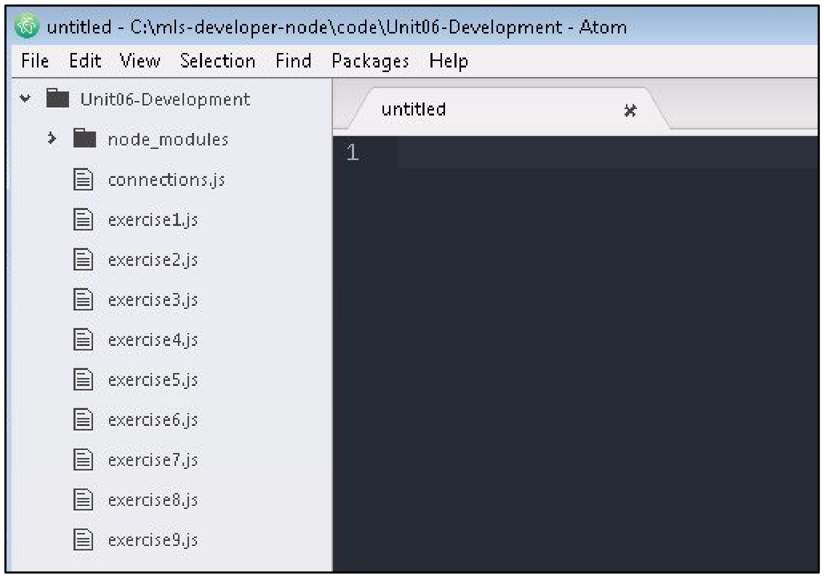
```
C:\Users\Administrator>cd c:\mls-developer-node\code\Unit06-Development

c:\mls-developer-node\code\Unit06-Development>npm install marklogic
marklogic@1.0.2 node_modules\marklogic
├── core-util-is@1.0.1
├── www-authenticate@0.6.2
├── yakaa@1.0.1
├── qs@2.4.2
├── multipart-stream@1.0.0 (inherits@2.0.1, sandwich-stream@0.0.4)
├── through2@0.6.5 (xtend@4.0.0, readable-stream@1.0.33)
├── concat-stream@1.4.8 (inherits@2.0.1, typedarray@0.0.6, readable-stream@1.1.1
3)
├── bluebird@2.9.25
├── deepcopy@0.4.0
└── dicer@0.2.4 (streamsearch@0.1.2, readable-stream@1.1.13)

c:\mls-developer-node\code\Unit06-Development>_
```

2. Open the Atom editor from your Desktop:

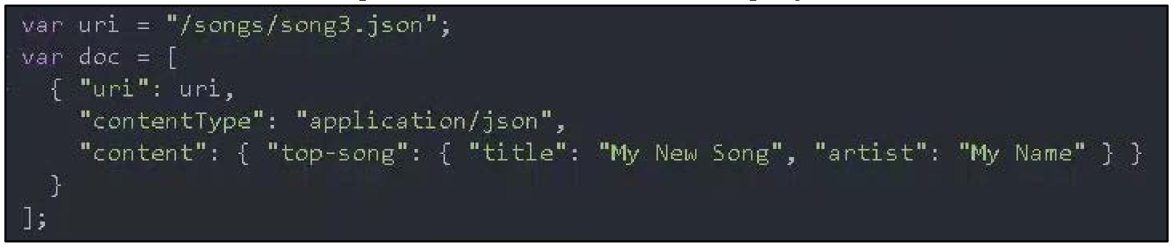
3. In the Atom editor select File→Open Folder…:


untitled - Atom
File Edit View Selection Find Packages Help
New Window Ctrl+Shift+N
New File Ctrl+N
Open File... Ctrl+O
Open Folder... Ctrl+Shift+O
Add Project Folder... Alt+Ctrl+O
Reopen Last Item Ctrl+Shift+T
Settings Ctrl+Comma

4. Open the **Unit06-Development** folder from c**:\mls-developer-node\code\**
5. You should see the following structure in place in your editor:

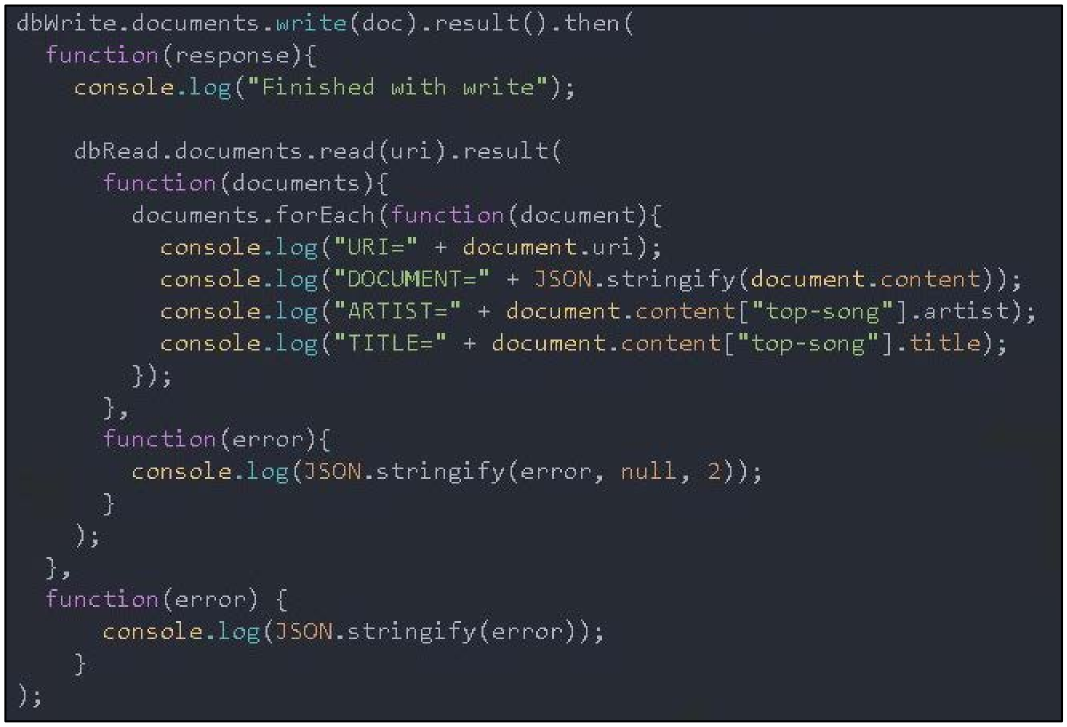


6. Select **exercise1.js**.
7. Take a moment to study the comments and code.
8. Note that the URI is defined and that a document descriptor is built and stored in a variable called doc. The doc variable contains an array of JSON objects, meaning you could have more than 1 document descriptor defined, however our example just has one:



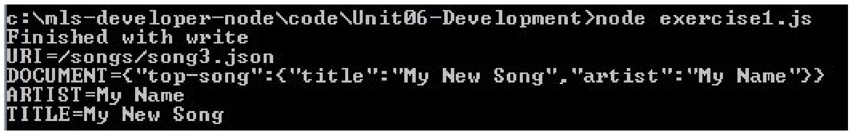
```
var uri = "/songs/song3.json";
var doc = [
  { "uri": uri,
    "contentType": "application/json",
    "content": { "top-song": { "title": "My New Song", "artist": "My Name" } }
  }
];
```

9. Note that when writing the document a promise result handler is used. This is important in that we want the document write to be synchronized with the document read:



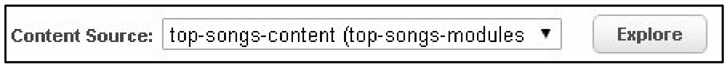
```
dbWrite.documents.write(doc).result().then(
  function(response){
    console.log("Finished with write");

    dbRead.documents.read(uri).result(
      function(documents){
        documents.forEach(function(document){
          console.log("URI=" + document.uri);
          console.log("DOCUMENT=" + JSON.stringify(document.content));
          console.log("ARTIST=" + document.content["top-song"].artist);
          console.log("TITLE=" + document.content["top-song"].title);
        });
      },
      function(error){
        console.log(JSON.stringify(error, null, 2));
      }
    );
  },
  function(error) {
      console.log(JSON.stringify(error));
    }
);
```

10. Now let’s test the code.
11. At the command line, go to the **c:\mls-developer-node\Unit06-Development** directory.
12. Enter **node exercise1.js** and press enter.
13. You should see the following response:



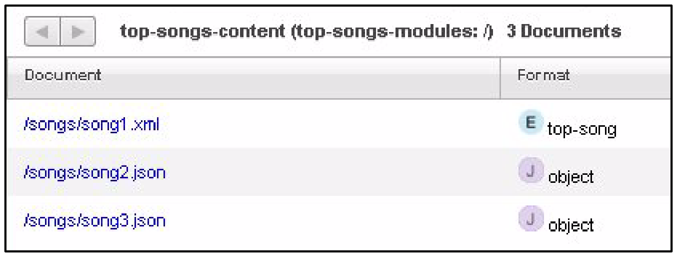
```
c:\mls-developer-node\code\Unit06-Development>node exercise1.js
Finished with write
URI=/songs/song3.json
DOCUMENT={"top-song":{"title":"My New Song","artist":"My Name"}}
ARTIST=My Name
TITLE=My New Song
```

14. In browser, navigate to **http://localhost:8000/qconsole**
15. Select **top-songs-content** from the **content source** dropdown and click explore:



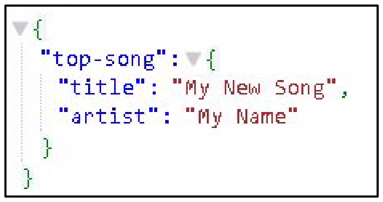
Content Source: top-songs-content (top-songs-modules ▼

16. You should see the new URI (**/songs/song3.json**) in the database:



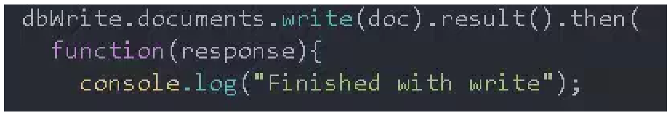
| Document | Format |
|---|---|
| /songs/song1.xml | E top-song |
| /songs/song2.json | J object |
| /songs/song3.json | J object |

17. Click on the URI to view the document contents:



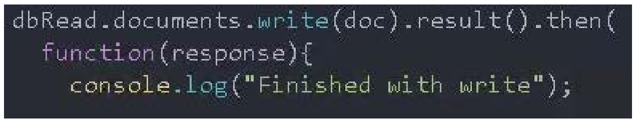
```
{
  "top-song": {
    "title": "My New Song",
    "artist": "My Name"
  }
}
```

18. Next, let's make a change to this code to reinforce our understanding of role based security.
19. Within exercise1.js, locate the bit of code where the document is written to the database:



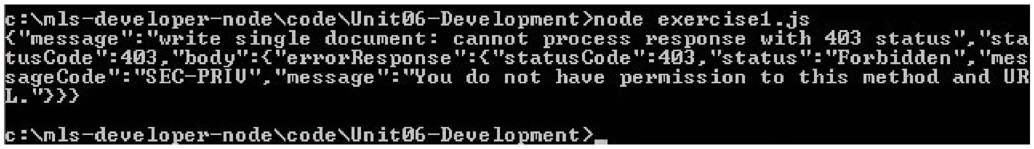
```
dbWrite.documents.write(doc).result().then(
  function(response){
    console.log("Finished with write");
```

20. Change dbWrite to dbRead as shown:



```
dbRead.documents.write(doc).result().then(
  function(response){
    console.log("Finished with write");
```

21. Save your changes and test.
22. At the command line, go to the **c:\mls-developer-node\Unit06-Development** directory.
23. Enter **node exercise1.js** and press enter.
24. Note the error cause by invalid permissions when trying to perform a write using the database client that is authenticating with our rest-reader-user:

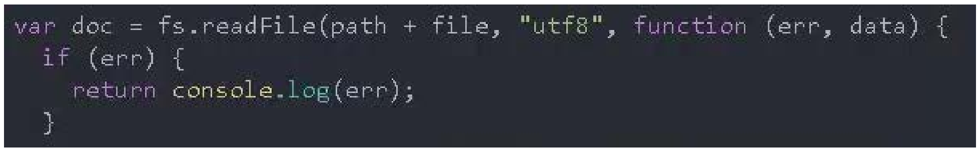


```
c:\mls-developer-node\code\Unit06-Development>node exercise1.js
{"message":"write single document: cannot process response with 403 status","sta
tusCode":403,"body":{"errorResponse":{"statusCode":403,"status":"Forbidden","mes
sageCode":"SEC-PRIV","message":"You do not have permission to this method and UR
L."}}}

c:\mls-developer-node\code\Unit06-Development>_
```

## Exercise 2: Load a Document from the File System

In this exercise you will implement code that will load a document from the file system into the database.

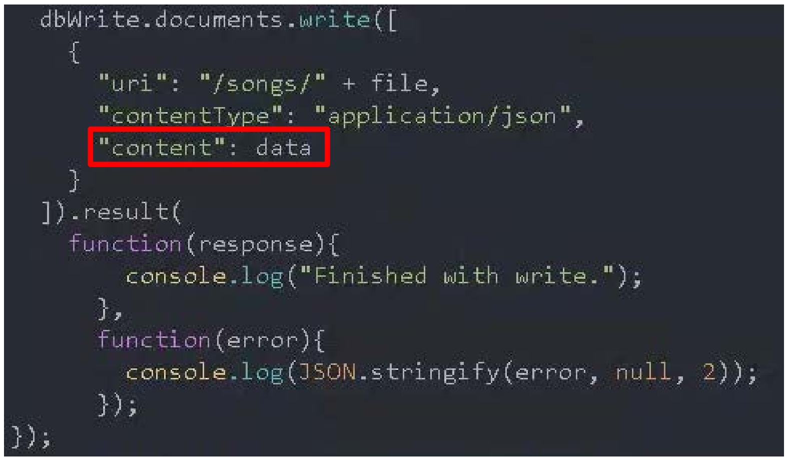
1. In your editor, within the **Unit06-Development** project, select **exercise2.js**.
2. Study the comments and code.
3. Note the use of **fs** to read a document from the file system and store it in the **data** variable in the callback:



```
var doc = fs.readFile(path + file, "utf8", function (err, data) {
  if (err) {
    return console.log(err);
  }
```

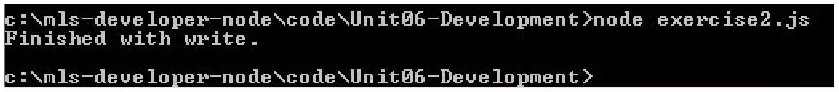
*Note:*
*For more information on fs.readFile please see: https://nodejs.org/api/fs.html*

4. Note the document descriptor that is being created, and passed the information that was read from the file system and stored in the **data** variable:



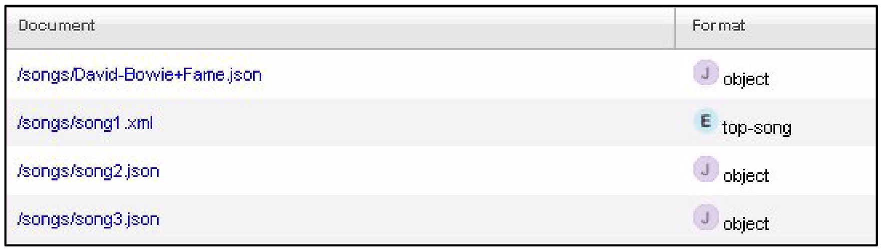
```
  dbWrite.documents.write([
    {
      "uri": "/songs/" + file,
      "contentType": "application/json",
      "content": data
    }
  ]).result(
    function(response){
        console.log("Finished with write.");
      },
      function(error){
        console.log(JSON.stringify(error, null, 2));
      });
});
```

5. Now let's test the code.
6. At the command line, go to the **c:\mls-developer-node\Unit06-Development** directory.
7. Enter **node exercise2.js** and press enter.
8. You should see the following response:



```
c:\mls-developer-node\code\Unit06-Development>node exercise2.js
Finished with write.

c:\mls-developer-node\code\Unit06-Development>
```

9. Validate the document was loaded by exploring your database in Query Console:



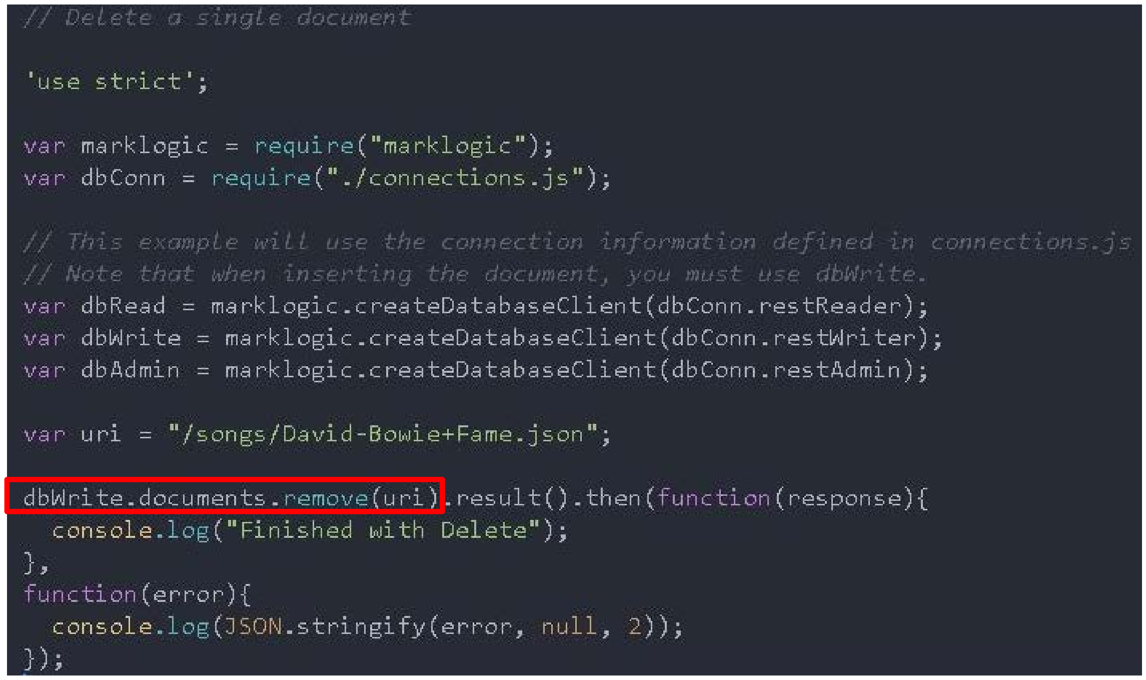
| Document | Format |
|---|---|
| /songs/David-Bowie+Fame.json | J object |
| /songs/song1.xml | E top-song |
| /songs/song2.json | J object |
| /songs/song3.json | J object |

10. Click on the document to view its contents.
11. Note that the song documents stored on the file system have a lot more information in them than the simple song documents we have been working with up to this point. It's this abundance of both structured information and unstructured full text that makes these song documents good for exploring search, which we will cover later.

## Exercise 3: Delete a Doument

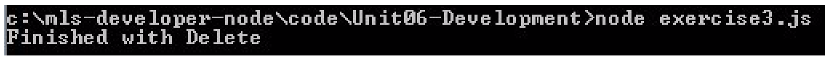
In this exercise you will implement code that will delete a document from the database.

1. In your editor, within the **Unit06-Development** project, select **exercise3.js**.
2. Study the comments and code.
3. Note that the method is called remove so as not to conflict with the standard delete operator in JavaScript. Also note the use of the dbWrite database client. Deleting a document in MarkLogic requires update permissions on that document. We'll go deeper about why that is the case when we discuss transactions and MVCC later in this course:



```
// Delete a single document

'use strict';

var marklogic = require("marklogic");
var dbConn = require("./connections.js");

// This example will use the connection information defined in connections.js
// Note that when inserting the document, you must use dbWrite.
var dbRead = marklogic.createDatabaseClient(dbConn.restReader);
var dbWrite = marklogic.createDatabaseClient(dbConn.restWriter);
var dbAdmin = marklogic.createDatabaseClient(dbConn.restAdmin);

var uri = "/songs/David-Bowie+Fame.json";

dbWrite.documents.remove(uri).result().then(function(response){
  console.log("Finished with Delete");
},
function(error){
  console.log(JSON.stringify(error, null, 2));
});
```

4. Now let's test the code.
5. At the command line, go to the **c:\mls-developer-node\Unit06-Development** directory.
6. Enter **node exercise3.js** and press enter.
7. You should see the following response:



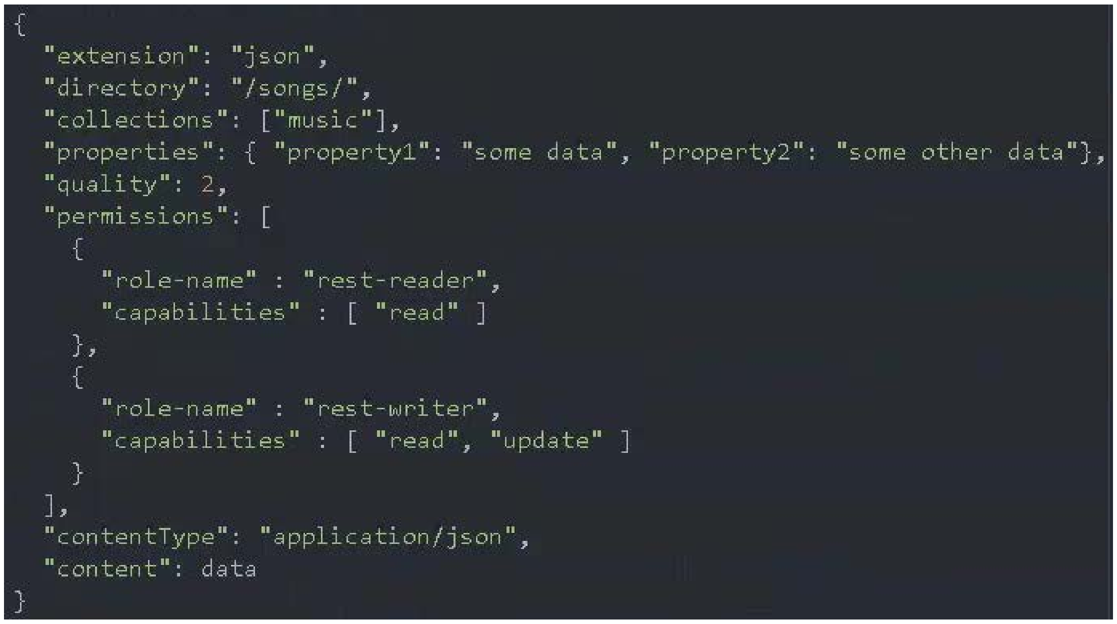
```
c:\mls-developer-node\code\Unit06-Development>node exercise3.js
Finished with Delete
```

8. Explore the database in Query Console to confirm the document was deleted.

## Exercise 4: Load a Document and Metadata

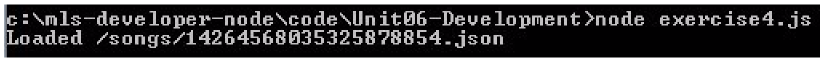
In this exercise you will implement code that will load a document and manage document metadata.

1. In your editor, within the **Unit06-Development** project, select **exercise4.js**.
2. Study the comments and code.
3. Note that the document descriptor in this example is much more robust than in prior examples, as it's in the document descriptor that we can manage all the document metadata. Also note that there is no URI property defined in the document descriptor. This will enable automatic URI generation:



```
{
  "extension": "json",
  "directory": "/songs/",
  "collections": ["music"],
  "properties": { "property1": "some data", "property2": "some other data"},
  "quality": 2,
  "permissions": [
    {
      "role-name" : "rest-reader",
      "capabilities" : [ "read" ]
    },
    {
      "role-name" : "rest-writer",
      "capabilities" : [ "read", "update" ]
    }
  ],
  "contentType": "application/json",
  "content": data
}
```

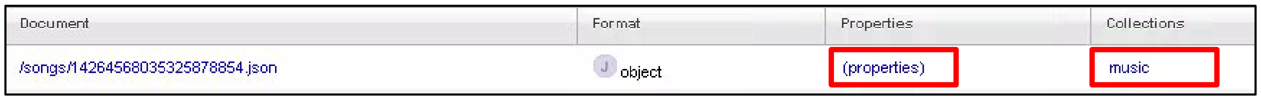
4. Now let's test the code.
5. At the command line, go to the **c:\mls-developer-node\Unit06-Development** directory.
6. Enter **node exercise4.js** and press enter.
7. You should see the following response:



```
c:\mls-developer-node\code\Unit06-Development>node exercise4.js
Loaded /songs/14264568035325878854.json
```

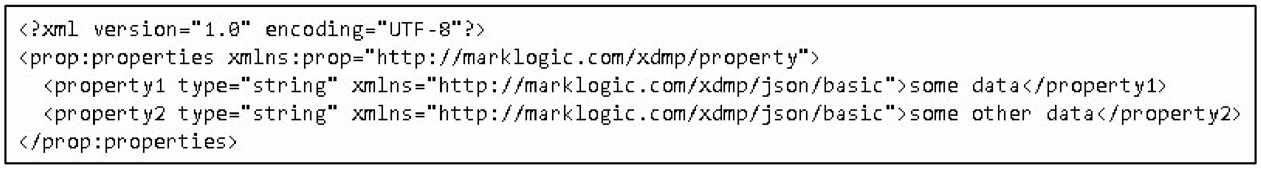
8. Note that the URI name begins with **/songs/**. This was controlled by the **directory** property in the document descriptor.
9. Note that the URI ends with **.json**. This was controlled by the **extension** property in the document descriptor.

10. Explore the database in Query Console, and note that it belongs to a collection called **music** and has a link indicating the document has **properties** metadata. Also note that your URI will be different and that's OK – its because we dynamically generated the URI:



| Document | Format | Properties | Collections |
|---|---|---|---|
| /songs/14264568035325878854.json | J object | (properties) | music |

11. Click on the properties link to view the metadata:

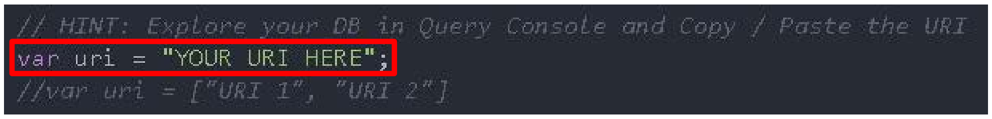


```
<?xml version="1.0" encoding="UTF-8"?>
<prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
  <property1 type="string" xmlns="http://marklogic.com/xdmp/json/basic">some data</property1>
  <property2 type="string" xmlns="http://marklogic.com/xdmp/json/basic">some other data</property2>
</prop:properties>
```

## Exercise 5: Get a Documents Metadata

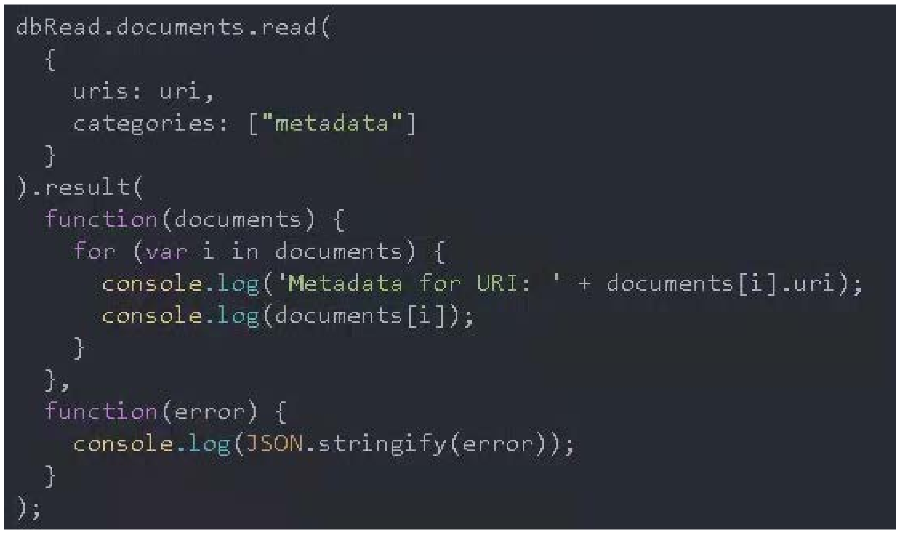
In this exercise you will implement code that will return document metadata from the database.

1. In your editor, within the **Unit06-Development** project, select **exercise5.js**.
2. Study the comments and code.
3. Note that because we used dynamic URI generation in the last exercise, we'll need to copy / paste the URI from Query Console. Explore your database in Query Console and copy the URI of the document that was created in the last exercise.
4. Paste it into the code as highlighted below:



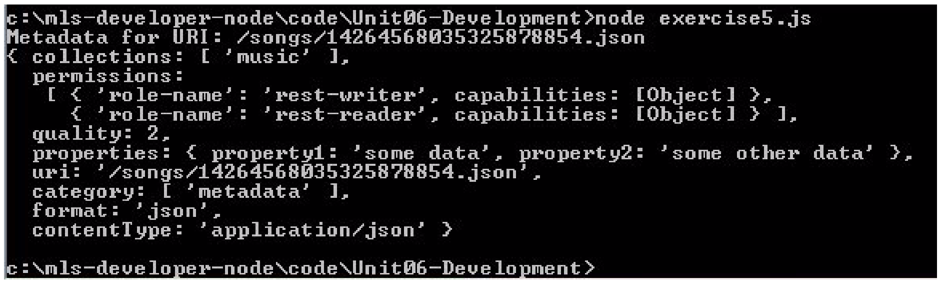
```
// HINT: Explore your DB in Query Console and Copy / Paste the URI
var uri = "YOUR URI HERE";
//var uri = ["URI 1", "URI 2"]
```

5. Study the rest of the code, noting that when performing the read, we pass it a document descriptor that asks for the metadata property to be returned. Also note that we iterate over the results, meaning we could pass in an array of URIs and output the results for each:



```
dbRead.documents.read(
  {
    uris: uri,
    categories: ["metadata"]
  }
).result(
  function(documents) {
    for (var i in documents) {
      console.log('Metadata for URI: ' + documents[i].uri);
      console.log(documents[i]);
    }
  },
  function(error) {
    console.log(JSON.stringify(error));
  }
);
```

6. Now let's test the code.
7. At the command line, go to the **c:\mls-developer-node\Unit06-Development** directory.
8. Enter **node exercise5.js** and press enter.
9. You should see the following response:



```
c:\mls-developer-node\code\Unit06-Development>node exercise5.js
Metadata for URI: /songs/14264568035325878854.json
{ collections: [ 'music' ],
  permissions:
   [ { 'role-name': 'rest-writer', capabilities: [Object] },
     { 'role-name': 'rest-reader', capabilities: [Object] } ],
  quality: 2,
  properties: { property1: 'some data', property2: 'some other data' },
  uri: '/songs/14264568035325878854.json',
  category: [ 'metadata' ],
  format: 'json',
  contentType: 'application/json' }

c:\mls-developer-node\code\Unit06-Development>
```

*Note:*
*If you specify a document URI that does not exist, this is not an error condition. Rather you will get an empty response back.*

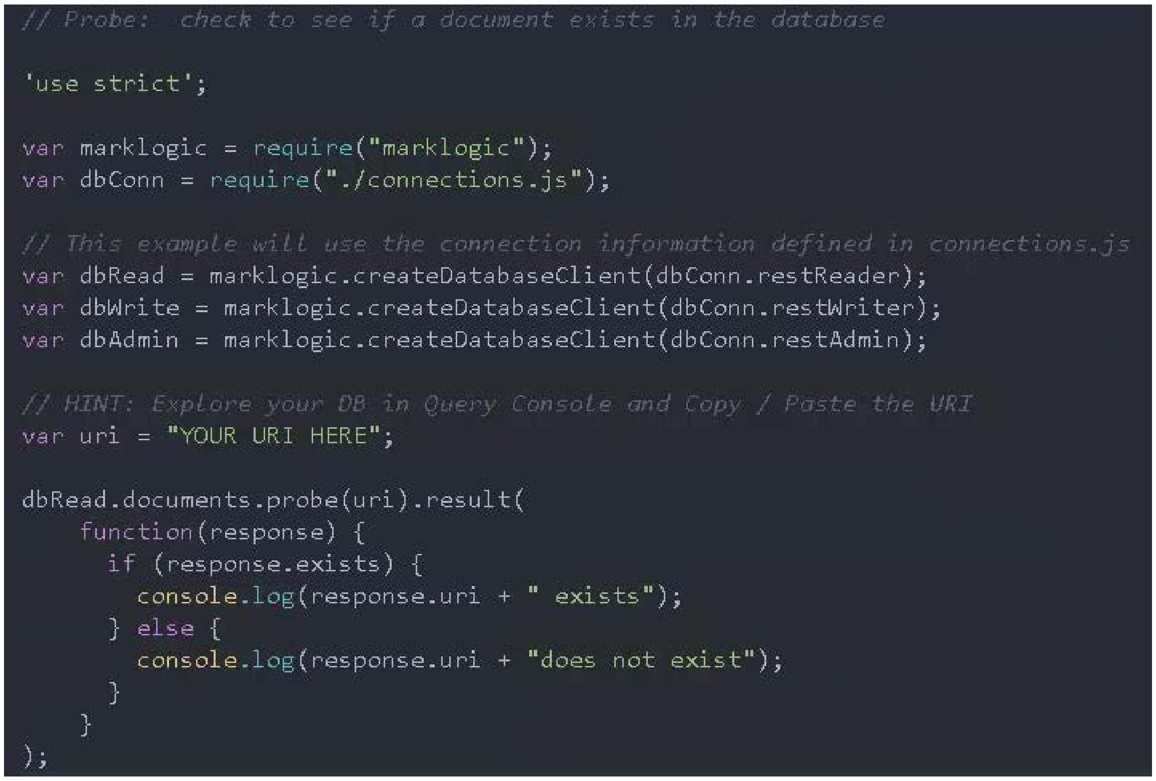
*If you get an empty response, double check that the URI you entered in your code does exist in the database.*

10. Optional: edit the uri variable to pass in an array of URIs from your database and test.

## Exercise 6: Probe for a Document

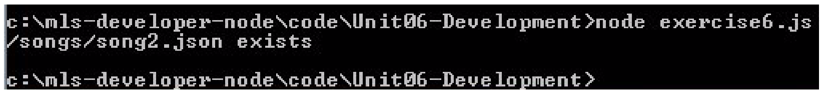
In this exercise you will implement code that will check for the existence of a URI in the database.

1. In your editor, within the **Unit06-Development** project, select **exercise6.js**.
2. Study the comments and code.



```
// Probe:  check to see if a document exists in the database

'use strict';

var marklogic = require("marklogic");
var dbConn = require("./connections.js");

// This example will use the connection information defined in connections.js
var dbRead = marklogic.createDatabaseClient(dbConn.restReader);
var dbWrite = marklogic.createDatabaseClient(dbConn.restWriter);
var dbAdmin = marklogic.createDatabaseClient(dbConn.restAdmin);

// HINT: Explore your DB in Query Console and Copy / Paste the URI
var uri = "YOUR URI HERE";

dbRead.documents.probe(uri).result(
    function(response) {
      if (response.exists) {
        console.log(response.uri + " exists");
      } else {
        console.log(response.uri + "does not exist");
      }
    }
);
```

3. Edit the uri variable, entering a URI from your database.
4. Save your code.
5. Now let’s test the code.
6. At the command line, go to the **c:\mls-developer-node\Unit06-Development** directory.
7. Enter **node exercise6.js** and press enter.
8. You should see the following response:

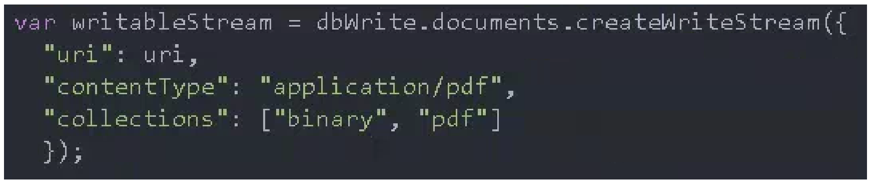


```
c:\mls-developer-node\code\Unit06-Development>node exercise6.js
/songs/song2.json exists

c:\mls-developer-node\code\Unit06-Development>
```

## Exercise 7: Load a Binary Document Using a Stream

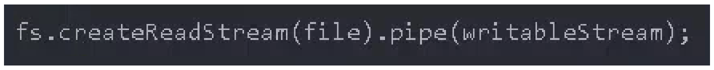
In this exercise you will implement code that will load a binary document using a stream.

1. In your editor, within the **Unit06-Development** project, select **exercise7.js**.
2. Study the comments and code.
3. Note that we are creating a writeable stream which will accept a filestream through the pipe method. We will use this stream the insert. We're also building a document descriptor for our document with the appropriate content type, and we're putting it into some collections to organize our data:



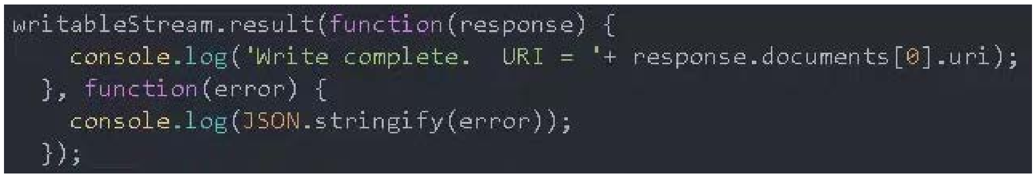
```
var writableStream = dbWrite.documents.createWriteStream({
  "uri": uri,
  "contentType": "application/pdf",
  "collections": ["binary", "pdf"]
  });
```

4. Next, note that we are using fs to stream the read of the document, and piping that into the write stream that we created:



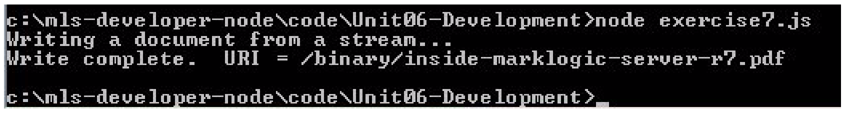
```
fs.createReadStream(file).pipe(writableStream);
```

5. Finally, we stream the write into the database:



```
writableStream.result(function(response) {
    console.log('Write complete.  URI = '+ response.documents[0].uri);
  }, function(error) {
    console.log(JSON.stringify(error));
  });
```

6. Now let's test the code.
7. At the command line, go to the **c:\mls-developer-node\Unit06-Development** directory.
8. Enter **node exercise7.js** and press enter.
9. You should see the following response:



```
c:\mls-developer-node\code\Unit06-Development>node exercise7.js
Writing a document from a stream...
Write complete.  URI = /binary/inside-marklogic-server-r7.pdf

c:\mls-developer-node\code\Unit06-Development>_
```

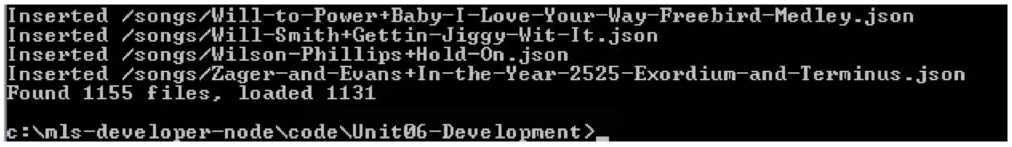
10. Explore your database in Query Console to view the new document.

## Exercise 8: Bulk Load Documents

In this exercise you will implement code that will bulk load documents from the file system.

This bulk load is accomplished by batching documents together and writing them to the database. We batch the documents together rather than trying to load them all at once because Node.js can open up a limited number of files at the same time. While this limit could be changed at the OS level, we instead opted to manage it in the application code.

1. In your editor, within the **Unit06-Development** project, select **exercise8.js**.
2. Study the comments and code.
3. Note that because the amount of files in a directory could be large, this code breaks it up into batches. This adds a bit more complexity to the code and logic.
4. Now let's test the code.
5. At the command line, go to the **c:\mls-developer-node\Unit06-Development** directory.
6. Enter **node exercise8.js** and press enter.
7. After the batch load is complete, which may take a few seconds on your VMs, you should see a similar response indicating the number of files found and loaded:



```
Inserted /songs/Will-to-Power+Baby-I-Love-Your-Way-Freebird-Medley.json
Inserted /songs/Will-Smith+Gettin-Jiggy-Wit-It.json
Inserted /songs/Wilson-Phillips+Hold-On.json
Inserted /songs/Zager-and-Evans+In-the-Year-2525-Exordium-and-Terminus.json
Found 1155 files, loaded 1131

c:\mls-developer-node\code\Unit06-Development>_
```

8. Explore your database in Query Console, noting that you now have a lot more documents that have been loaded.

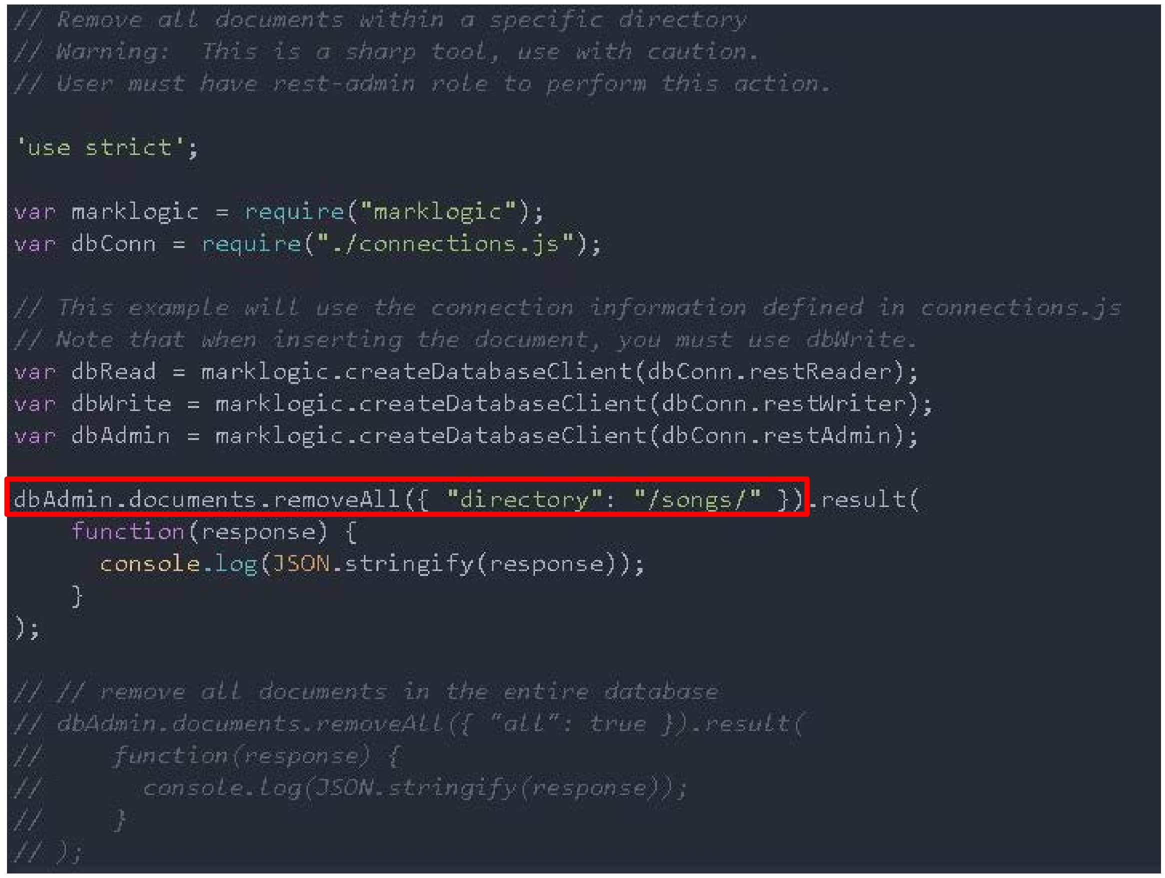
## Exercise 9: Remove All Documents

In this exercise you will implement code that will clear all documents from the database. Performing this operation requires elevated privileges, specifically you must authenticate as a user with the rest-admin-role.

This functionality is considered a sharp tool, meaning you should take precautions and consider taking a database backup before using it in the real world.

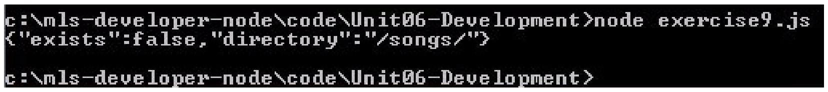
For our objectives in this training, it will be useful for us because we want to clear out everything from our database before we perform the final data ingest using MarkLogic Content Pump (mlcp).

1. In your editor, within the **Unit06-Development** project, select **exercise9.js**.
2. Study the comments and code.
3. Note that we will use the **dbAdmin** database client connection. Also note that for this first example we won't clear the entire database, but rather only those documents that are in the **/songs/** directory:



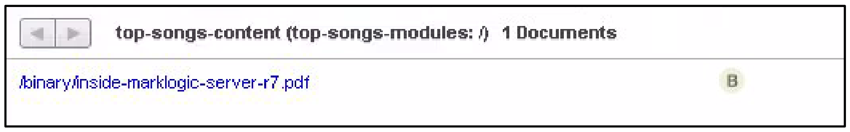
```
// Remove all documents within a specific directory
// Warning:  This is a sharp tool, use with caution.
// User must have rest-admin role to perform this action.

'use strict';

var marklogic = require("marklogic");
var dbConn = require("./connections.js");

// This example will use the connection information defined in connections.js
// Note that when inserting the document, you must use dbWrite.
var dbRead = marklogic.createDatabaseClient(dbConn.restReader);
var dbWrite = marklogic.createDatabaseClient(dbConn.restWriter);
var dbAdmin = marklogic.createDatabaseClient(dbConn.restAdmin);

dbAdmin.documents.removeAll({ "directory": "/songs/" }).result(
    function(response) {
      console.log(JSON.stringify(response));
    }
);

// // remove all documents in the entire database
// dbAdmin.documents.removeAll({ "all": true }).result(
//     function(response) {
//       console.log(JSON.stringify(response));
//     }
// );
```

4. Now let’s test the code.
5. At the command line, go to the **c:\mls-developer-node\Unit06-Development** directory.
6. Enter **node exercise9.js** and press enter.
7. You should see the following response:



```
c:\mls-developer-node\code\Unit06-Development>node exercise9.js
{"exists":false,"directory":"/songs/"}

c:\mls-developer-node\code\Unit06-Development>
```

8. Explore your database in Query Console and validate that the /songs/ documents were removed. The only remaining document should be the binary that we loaded:

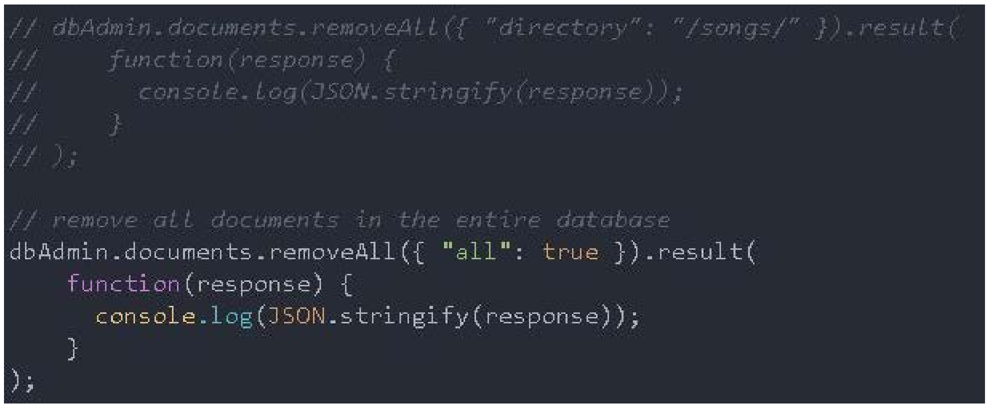


top-songs-content (top-songs-modules: /) 1 Documents

/binary/inside-marklogic-server-r7.pdf B

9. Next, let’s remove this document as well by clearing the whole database.
10. In the exercise9.js code, comment out the section of code that deleted the entire directory, and uncomment out the code that will clear the whole database.

*Hint:*
*You can highlight and unhighlight a block of code in the Atom editor by pressing: CTRL + /*



```
// dbAdmin.documents.removeAll({ "directory": "/songs/" }).result(
//     function(response) {
//       console.log(JSON.stringify(response));
//     }
// );

// remove all documents in the entire database
dbAdmin.documents.removeAll({ "all": true }).result(
    function(response) {
      console.log(JSON.stringify(response));
    }
);
```

11. Now test the code again. When complete, explore the database in Query Console to validate that no more documents remain.

## Exercise 10: Create an XDBC Application Server

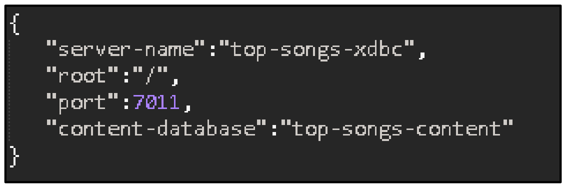
In this exercise you will setup your environment to ingest data using MarkLogic Content Pump.

Since MarkLogic Content Pump is a Java application talking to MarkLogic through the XCC (XML Client Connector) library, it requires an XDBC Application Server to be setup in MarkLogic.

To create the XDBC application server you can use a variety of approaches: By hand using the Admin Interface, by server side JavaScript or XQuery code using the Admin API, or through the MarkLogic Management REST API.

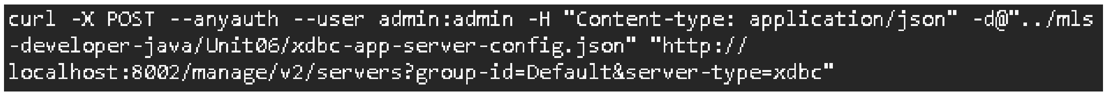
In this exercise we use the Management REST API.

1. Navigate to **c:\mls-developer-node\Unit06** and open **xdbc-app-server-config.json**
2. Take note of the basic configuration parameters defined:



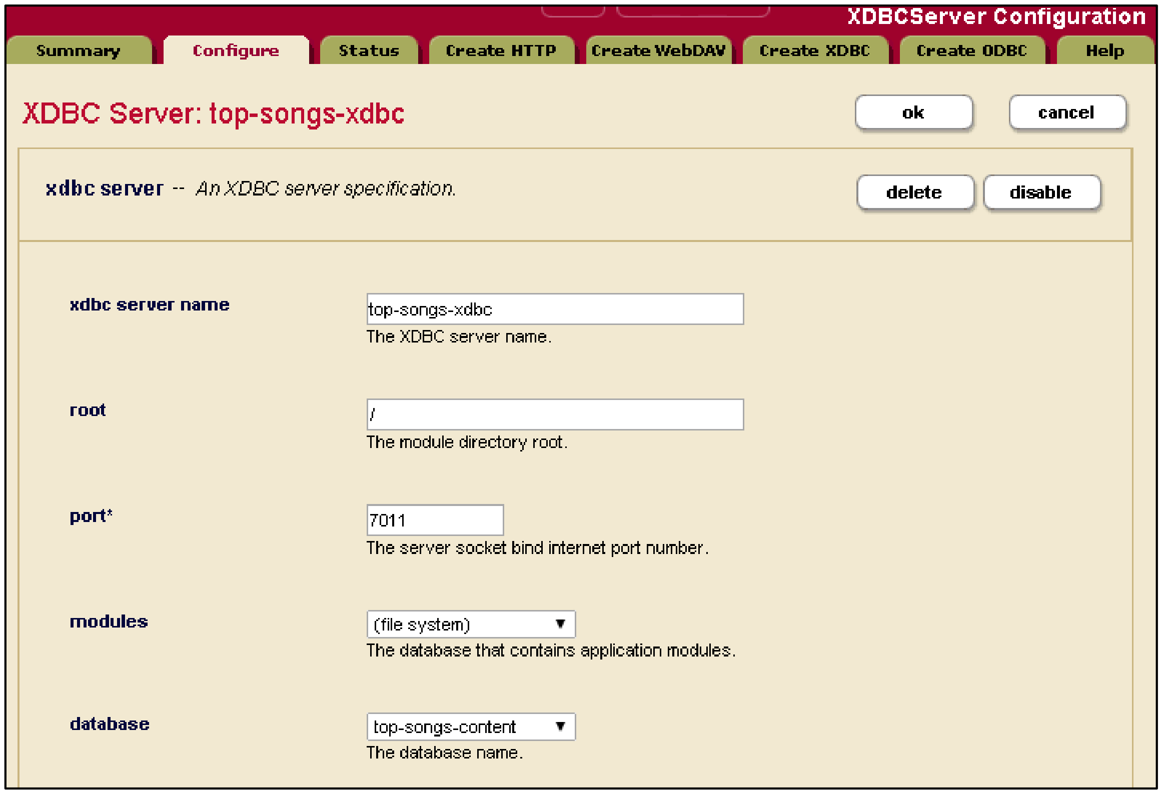
```
{
   "server-name":"top-songs-xdbc",
   "root":"/",
   "port":7011,
   "content-database":"top-songs-content"
}
```

3. Navigate to **c:\mls-developer-node\Unit06** and open **xdbc-app-server-curl.txt**
4. Study the cURL command that builds a POST request to the **/manage/v2/servers** endpoint in order to create our new XDBC app server:



```
curl -X POST --anyauth --user admin:admin -H "Content-type: application/json" -d@"../mls
-developer-java/Unit06/xdbc-app-server-config.json" "http://
localhost:8002/manage/v2/servers?group-id=Default&server-type=xdbc"
```

5. Execute the cURL command from the command prompt at **c:\curl**.
6. In a browser, navigate to the Admin interface at **http://localhost:8001** to view the new app server configuration, taking note that it is setup to point to the **top-songs-content** database:



XDBC Server: top-songs-xdbc

xdbc server -- *An XDBC server specification.*

| | |
|---|---|
| xdbc server name | top-songs-xdbc<br>The XDBC server name. |
| root | /<br>The module directory root. |
| port* | 7011<br>The server socket bind internet port number. |
| modules | (file system)<br>The database that contains application modules. |
| database | top-songs-content<br>The database name. |

## Exercise 11: Load Data with MarkLogic Content Pump

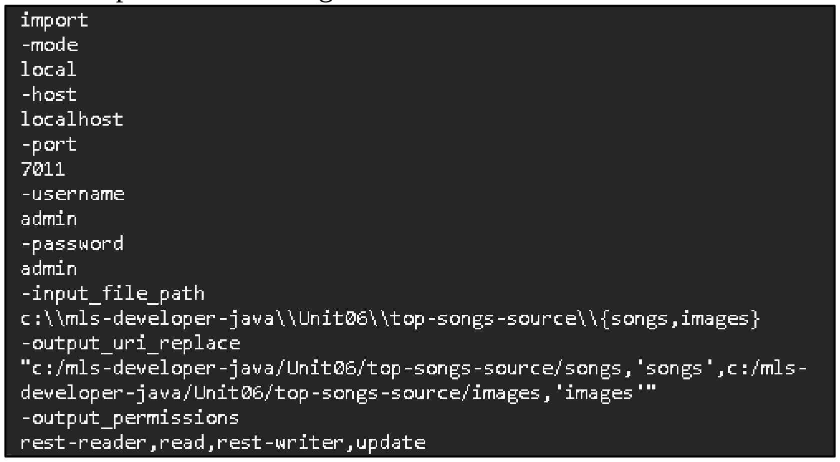
In this exercise we will use MarkLogic Content Pump to ingest the project data (both JSON and binary) into the database.

1. Using the Admin interface (http://localhost:8001), **clear** the **top-songs-content** database.

*Note:*
*Remember that clearing a database will remove all the data across all forests attached to that database, while deleting a database removes the actual database configuration.*

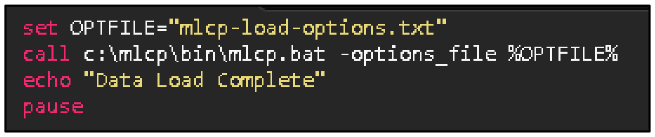
*Our goal is to clear the data but keep the configuration. We will then load the data using mlcp.*

2. Navigate to **c:\mls-developer-node\Unit06** and open **mlcp-load-options.txt**
3. Take a minute to study the options.
4. Note the host, port and authentication information used by mlcp to communicate with the XDBC application server that was just created.
5. Note the locations you are going to load data from on the file system.
6. Note the URI naming scheme being implemented.
7. Note the permissions being defined.



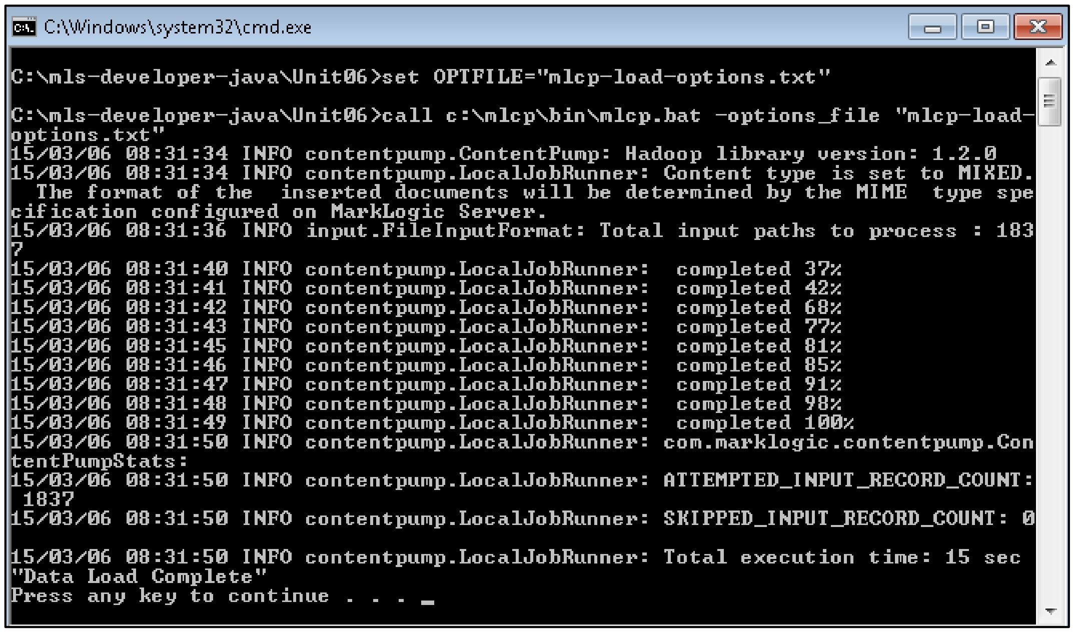
```
import
-mode
local
-host
localhost
-port
7011
-username
admin
-password
admin
-input_file_path
c:\\mls-developer-java\\Unit06\\top-songs-source\\{songs,images}
-output_uri_replace
"c:/mls-developer-java/Unit06/top-songs-source/songs,'songs',c:/mls-
developer-java/Unit06/top-songs-source/images,'images'"
-output_permissions
rest-reader,read,rest-writer,update
```

12. Navigate to **c:\mls-developer-node\Unit06** and *edit* **mlcp-load.bat** in and editor of your choice. Take a minute to study the setup:



```
set OPTFILE="mlcp-load-options.txt"
call c:\mlcp\bin\mlcp.bat -options_file %OPTFILE%
echo "Data Load Complete"
pause
```

13. Note that it defines where to find the load options and that it calls the mlcp.bat file from the location where mlcp has been extracted. This is the directory where mlcp has been placed on your VM; it is not required to be at this location. MarkLogic Content Pump may be downloaded from http://developer.marklogic.com.
14. Double click **mlcp-load.bat** to execute the batch file.
15. View the load statistics. You will have different timings but the amount of data ingested is the same. Remember the source data we are loading. It is a mix of binary (JPEG images) and JSON. The binaries tend to be larger. Also keep in mind that indexing occurs during ingestion, so when the load is complete both the documents and indexes are available.



```
C:\mls-developer-java\Unit06>set OPTFILE="mlcp-load-options.txt"

C:\mls-developer-java\Unit06>call c:\mlcp\bin\mlcp.bat -options_file "mlcp-load-
options.txt"
15/03/06 08:31:34 INFO contentpump.ContentPump: Hadoop library version: 1.2.0
15/03/06 08:31:34 INFO contentpump.LocalJobRunner: Content type is set to MIXED.
  The format of the  inserted documents will be determined by the MIME  type spe
cification configured on MarkLogic Server.
15/03/06 08:31:36 INFO input.FileInputFormat: Total input paths to process : 183
7
15/03/06 08:31:40 INFO contentpump.LocalJobRunner:  completed 37%
15/03/06 08:31:41 INFO contentpump.LocalJobRunner:  completed 42%
15/03/06 08:31:42 INFO contentpump.LocalJobRunner:  completed 68%
15/03/06 08:31:43 INFO contentpump.LocalJobRunner:  completed 77%
15/03/06 08:31:45 INFO contentpump.LocalJobRunner:  completed 81%
15/03/06 08:31:46 INFO contentpump.LocalJobRunner:  completed 85%
15/03/06 08:31:47 INFO contentpump.LocalJobRunner:  completed 91%
15/03/06 08:31:48 INFO contentpump.LocalJobRunner:  completed 98%
15/03/06 08:31:49 INFO contentpump.LocalJobRunner:  completed 100%
15/03/06 08:31:50 INFO contentpump.LocalJobRunner: com.marklogic.contentpump.Con
tentPumpStats:
15/03/06 08:31:50 INFO contentpump.LocalJobRunner: ATTEMPTED_INPUT_RECORD_COUNT:
 1837
15/03/06 08:31:50 INFO contentpump.LocalJobRunner: SKIPPED_INPUT_RECORD_COUNT: 0

15/03/06 08:31:50 INFO contentpump.LocalJobRunner: Total execution time: 15 sec
"Data Load Complete"
Press any key to continue . . . _
```

16. Explore your database in Query Console to validate the load.

MarkLogic UNIVERSITY

## DIY: Load the Star Wars Data

Now you have the chance to apply what you've learned from our discussions and the examples.

The goal is to load all the JSON data and binary images for the Star Wars database. The load must be completed according to these requirements for the final application to be successful:

1. Use **Node.js** code to load all JSON documents found at **c:\mls-developer-node\Unit06\star-wars\json**. All documents should be loaded into the database called **star-wars-content**.
   a. Using the REST instance that you created called **star-wars** on port **5002**.
   b. JSON documents should have URI names in the form of: **/character/filename.json**, where filename is equal to the filename of the document being loaded.
   c. JSON documents should be added to a collection called **character**.
   d. The rest-reader-role should be given read permissions.
   e. The rest-writer-role should be given update permissions.
2. Use **mlcp** to load all binary documents found at **c:\mls-developer-node\Unit06\star-wars\image**. All documents should be loaded into the database called **star-wars-content**.
   a. Create an XDBC app server on port **5001** that point to the **star-wars-content** database.
   b. Image documents should have URI names in the form of: **/image/filename.png**, where filename is equal to the filename of the document being loaded.
   c. Image documents should be added to a collection called **image**.
   d. The **rest-reader-role** should be given **read** permissions.
   e. The **rest-writer-role** should be given **update** permissions.

*Hint:*
*The examples in the lab exercises are there for you to reference and adapt as needed!!*