MarkLogic UNIVERSITY

# Unit 7

## Search Fundamentals

Use String Query

Use String Query and Customize Response Data

Use String Query and Imlement a Custom Grammar

Use Query By Example

Use Query By Example with a Range

Use Structured Query – Part I

Use Structured Query – Part II

Use Structured Query – Part III

Use Structured Query – Part IV

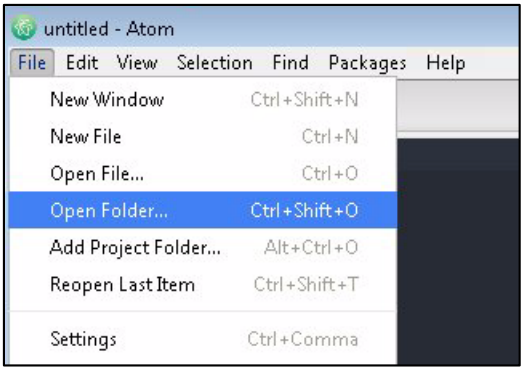
DIY: Search Star Wars Data

## Exercise 1: Use String Query

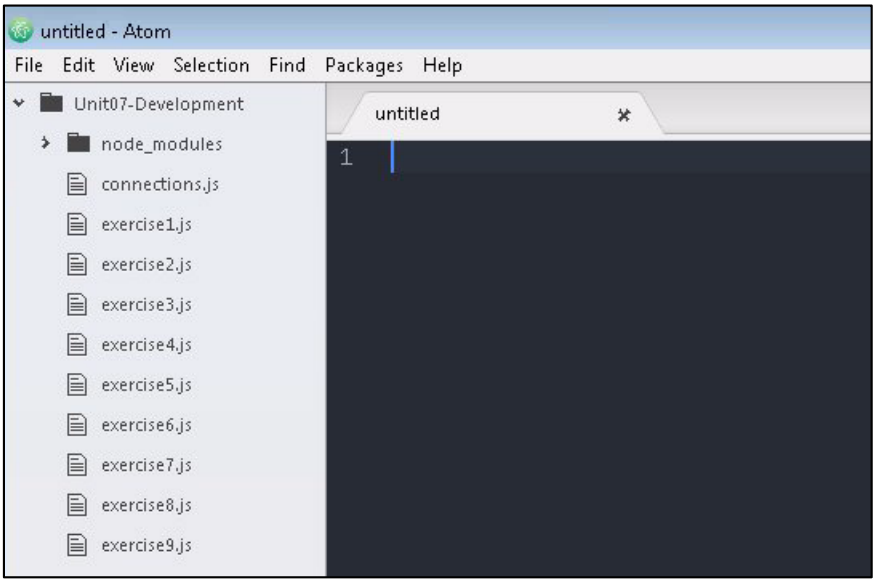
In this exercise you will implement code that will perform a simple string query search. You will study the response data to get familiar with the format and information that is returned to you in the response.

1. From the command line, navigate to the **c:\mls-developer-node\code\Unit07-Development** directory and run **npm install marklogic**.
2. Open the Atom editor from your Desktop:


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




4. Open the **Unit07-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 a query builder is defined:

```
var qb = marklogic.queryBuilder;
```

9. Note that you are defining the search criteria (**parsedFrom**), controlling to result set (**slice**) and executing the query (**results**) using a callback result handler:

```
dbRead.documents.query(
    qb.where(
      qb.parsedFrom(qText)
    ).slice(1, 5)
).result( function(results) {
  console.log(JSON.stringify(results, null, 2));
});
```

10. Now let's test the code.
11. At the command line, go to the **c:\mls-developer-node\Unit07-Development** directory.
12. Enter **node exercise1.js** and press enter.
13. Take some time to look at the response data.
14. Note that you are getting entire document sent back to you – so much information that it won't even all fit in the command prompt buffer.

## Exercise 2: Use String Query and Customize Response Data

In this exercise you will implement code that will perform a simple string query search and bring back to you the title and artist data from the matching documents.

1. In your editor, within the **Unit07-Development** project, select **exercise2.js**.
2. Study the comments and code.
3. Note the use of **qb.extract** to specify that you want the **artist** and **title** properties from the matching documents to be returned to you in the response. Also note that we are going to return the entire response so you can see the information that it contains:

```
dbRead.documents.query(
    qb.where(
      qb.parsedFrom(qText)
    ).slice(1, 5, qb.extract({
        paths: ["//artist", "//title"]
      }))
).result( function(matches) {
  // Study the full response data first to view its structure.
  // Think about how to programmatically iteract with it.
  console.log(matches);
```

4. Now let's test the code.
5. At the command line, go to the **c:\mls-developer-node\Unit07-Development** directory.
6. Enter **node exercise2.js** and press enter.
7. You should see the following response. Take note that you are not being given the entire document back but rather a summary of results returned as an array, including an extracted property:

```
c:\mls-developer-node\code\Unit07-Development>node exercise2.js
[ { uri: '/songs/Coldplay+Viva-la-Vida.json',
    category: 'content',
    format: 'json',
    contentType: 'application/json',
    contentLength: '122',
    content:
     { context: 'fn:doc("/songs/Coldplay+Viva-la-Vida.json")',
       extracted: [Object] } },
  { uri: '/songs/Katy-Perry+I-Kissed-a-Girl.json',
    category: 'content',
    format: 'json',
    contentType: 'application/json',
    contentLength: '132',
    content:
     { context: 'fn:doc("/songs/Katy-Perry+I-Kissed-a-Girl.json")',
       extracted: [Object] } },
```

8. Think about how you would access that extracted data in your code.
9. Next, let's make some changes to **exercise2.js** in your editor.

10. Locate the line that outputs the response data:

```
console.log(matches);
```

11. Comment that line out as follows:

```
//console.log(matches);
```

12. Next, uncomment the block of code that will iterate over each match in the response data array and output the artist and title data for each match:

```
matches.forEach(function(match) {
  console.log('Artist: '+ match.content.extracted[0].artist);
  console.log('Title: ' + match.content.extracted[1].title);
  console.log("--------")
});
```

13. Save your code.
14. Now let's test the code.
15. At the command line, go to the **c:\mls-developer-node\Unit07-Development** directory.
16. Enter **node exercise2.js** and press enter.
17. You should see the following response:

```
c:\mls-developer-node\code\Unit07-Development>node exercise2.js
Artist: Coldplay
Title: Viva la Vida
--------
Artist: Katy Perry
Title: I Kissed a Girl
--------
Artist: Nelly
Title: Hot in Herre
--------
Artist: Jay Sean featuring Lil Wayne
Title: Down
--------
Artist: Michael Jackson
Title: Billie Jean
--------
```

*Question:*
*Why are you getting matches for songs that don't have an artist = coldplay?*

*Answer:*
*The search is looking for the term coldplay anywhere in the document. It doesn't have to be found in the artist property.*

## Exercise 3: Use String Query and Implement a Custom Grammar

In this exercise you will implement code that will perform string query search with a custom grammar binding.

1. In your editor, within the **Unit07-Development** project, select **exercise3.js**.
2. Study the comments and code.
3. Note that you are still performing a string query using **parsedFrom**, but that you are now creating a **value** query on the **artist** property with a binding for **art**. What this means is that anytime **art:term** is found in your string query grammar, that term must now occur within the artist property:

```
dbRead.documents.query(
    qb.where(
      qb.parsedFrom(qText,
        qb.parseBindings(
          qb.value("artist", qb.bind("art"))
        )
      )
    ).slice(1, 5, qb.extract({
        paths: ["//artist", "//title"]
      }))
).result( function(matches) {
  matches.forEach(function(match) {
    console.log('Artist: '+ match.content.extracted[0].artist);
    console.log('Title: ' + match.content.extracted[1].title);
    console.log("--------")
  });
});
```

4. Now let's test the code.
5. At the command line, go to the **c:\mls-developer-node\Unit07-Development** directory.
6. Enter **node exercise3.js** and press enter.
7. You should see the following response:

```
c:\mls-developer-node\code\Unit07-Development>node exercise3.js
Artist: Coldplay
Title: Viva la Vida
--------
```

## Exercise 4: Use Query By Example

In this exercise you will implement code that will perform a search using Query By Example, or QBE.

1. In your editor, within the **Unit07-Development** project, select **exercise4.js**.
2. Study the comments and code.
3. Note the prototype, or example document that is being built:

```
var qbeDoc =
  {
    "artist": {$word: "the beatles"},
    "genre": "rock",
    "writer": {$word: "lennon"},
    "album": "Let It Be"
  };
```

*Note:*
*$word means that we desire to perform a "word query" within the specified property. Word queries are case insensitive, and do not require an exact match on the property value.*

*For more information reference: http://docs.marklogic.com/guide/node-dev/search#id_90528*

4. Note that in the query definition we are now using **qb.byExample**:

```
dbRead.documents.query(
    qb.where(
      qb.byExample(qbeDoc)
    ).slice(1, 5, qb.extract({
        paths: ["//artist", "//title"]
      }))
).result( function(matches) {
  matches.forEach(function(match) {
    console.log('Artist: '+ match.content.extracted[0].artist);
    console.log('Title: ' + match.content.extracted[1].title);
    console.log("--------")
  });
});
```

5. Now let's test the code.
6. At the command line, go to the **c:\mls-developer-node\Unit07-Development** directory.

7. Enter **node exercise4.js** and press enter.
8. You should see the following response:

## Exercise 5: Use Query By Example with a Range

In this exercise you will implement code that will perform a search using Query By Example, or QBE, and use a range constraint. You will also specify the search to be filtered, as range queries either must be filtered or have a backing range index.

In our next Unit we'll cover indexes, but since we haven't yet, we will need to perform a filtered search in this example.

1. In your editor, within the **Unit07-Development** project, select **exercise5.js**.
2. Study the comments and code.
3. Note that our example document now defines a date range wrapped inside of AND logic, requiring value of the week property to be within the specified bounds. Also note that because we don't have a range index on the week property, we must specify filtered search:

```
var qbeDoc =
  {
    "artist": {$word: "The Beatles"},
      $and: [
        {"week": { $gt: "1969-01-01" }},
        {"week": { $lt: "1969-12-31" }}
      ],
    $filtered: true
  };
```

4. Now let's test the code.
5. At the command line, go to the **c:\mls-developer-node\Unit07-Development** directory.
6. Enter **node exercise5.js** and press enter.
7. You should see the following response:

```
c:\mls-developer-node\code\Unit07-Development>node exercise5.js
Artist: The Beatles
Title: Come Together" / "Something
--------
Artist: The Beatles with Billy Preston
Title: Get Back
--------
```

## Exercise 6: Use Structured Query – Part I

In this exercise you will implement code that will perform a search using structured query.

1. In your editor, within the **Unit07-Development** project, select **exercise6.js**.
2. Study the comments and code.
3. Note the use of **qb.term**, which is one of many specific query constructors available to you as part of structured query. A term query simply needs to find the term anywhere in the document:

```
var qText = "beatles";

dbRead.documents.query(
    qb.where(qb.term(qText))
      .slice(1, 5, qb.extract({
        paths: ["//artist", "//title"]
      }))
).result( function(matches) {
  matches.forEach(function(match) {
    console.log('Artist: '+ match.content.extracted[0].artist);
    console.log('Title: ' + match.content.extracted[1].title);
    console.log("--------")
  });
});
```

4. Now let's test the code.
5. At the command line, go to the **c:\mls-developer-node\Unit07-Development** directory.
6. Enter **node exercise6.js** and press enter.
7. You should see the following response:

```
c:\mls-developer-node\code\Unit07-Development>node exercise6.js
Artist: The Beatles
Title: She Loves You
--------
Artist: The Beatles
Title: I Want to Hold Your Hand
--------
Artist: The Beatles with Billy Preston
Title: Get Back
--------
Artist: The Beatles
Title: The Long and Winding Road" / "For You Blue
--------
Artist: The Beatles
Title: Love Me Do
--------
```

## Exercise 7: Use Structured Query – Part II

In this exercise you will implement code that will perform a search using structured query.

1. In your editor, within the **Unit07-Development** project, select **exercise7.js**.
2. Study the comments and code.
3. Note that now you are using **qb.word** and specifying the **artist** property. This means that the term "beatles" must be found to be one of the words in the artist property, but that case does not matter:

```
var qText = "beatles";

dbRead.documents.query(
    qb.where(qb.word("artist", qText))
      .slice(1, 5, qb.extract({
        paths: ["//artist", "//title"]
      }))
).result( function(matches) {
  matches.forEach(function(match) {
    console.log('Artist: '+ match.content.extracted[0].artist);
    console.log('Title: ' + match.content.extracted[1].title);
    console.log("--------")
  });
});
```

4. Now let's test the code.
5. At the command line, go to the **c:\mls-developer-node\Unit07-Development** directory.
6. Enter **node exercise7.js** and press enter.
7. You should see the following response:

```
c:\mls-developer-node\code\Unit07-Development>node exercise7.js
Artist: The Beatles
Title: Hello, Goodbye
--------
Artist: The Beatles
Title: Eight Days a Week
--------
Artist: The Beatles
Title: We Can Work It Out
--------
Artist: The Beatles
Title: Ticket to Ride
--------
Artist: The Beatles
Title: Help!
--------
```

## Exercise 8: Use Structured Query – Part III

In this exercise you will implement code that will perform a search using structured query.

1. In your editor, within the **Unit07-Development** project, select **exercise8.js**.
2. Study the comments and code.
3. Note that now you are performing a qb.values query against the artist property. Values queries must be an exact match against the value of the property:

```
var qText = "beatles";

dbRead.documents.query(
    qb.where(qb.value("artist", qText))
      .slice(1, 5, qb.extract({
        paths: ["//artist", "//title"]
      }))
).result( function(matches) {
  matches.forEach(function(match) {
    console.log('Artist: '+ match.content.extracted[0].artist);
    console.log('Title: ' + match.content.extracted[1].title);
    console.log("--------")
  });
});
```

4. Now let's test the code.
5. At the command line, go to the **c:\mls-developer-node\Unit07-Development** directory.
6. Enter **node exercise8.js** and press enter.
7. You should see the following empty response, which means no matches were found:

```
c:\mls-developer-node\code\Unit07-Development>node exercise8.js

c:\mls-developer-node\code\Unit07-Development>_
```

8. Change the qText variable to "The Beatles" as shown:

```
var qText = "the beatles";
```

9. Save your code and test it again, noting that you now receive results.

## Exercise 9: Use Structured Query – Part IV

In this exercise you will implement code that will perform a search using structured query.

1. In your editor, within the **Unit07-Development** project, select **exercise9.js**.
2. Study the comments and code.
3. Take note of the variables that are defined that will be used in the structured query:

```
var searchDirectory = "/songs/"
var artistContains = "beatles";
var genreEquals = "rock";
var titleNotContains = "writer";
var docContains1 = "love";
var docContains2 = "night";
var docNearWord1 = "hard";
var docNearWord2 = "day";
var nearDistance = 1;
```

4. Take note of the various query constructors that are used to define the query:

```
dbRead.documents.query(
    qb.where(
      qb.and(
        qb.directory(searchDirectory),
        qb.word("artist", artistContains),
        qb.value("genre", genreEquals),
        qb.not(
          qb.word("title", titleNotContains)
        ),
        qb.or(
          qb.term(docContains1),
          qb.term(docContains2)
        ),
        qb.near(
          qb.term(docNearWord1),
          qb.term(docNearWord2),
          nearDistance
        )
      )
    )
      .slice(1, 5, qb.extract({
        paths: ["//artist", "//title"]
      }))
).result( function(matches) {
```

*Note:*
*There are many query constructors available that enable you to ask some powerful and complex questions about your data.*

*For more information please reference: http://docs.marklogic.com/guide/node-dev/search#id_62247*

1. Now let's test the code.
2. At the command line, go to the **c:\mls-developer-node\Unit07-Development** directory.
3. Enter **node exercise9.js** and press enter.
4. You should see the following response:

```
c:\mls-developer-node\code\Unit07-Development>node exercise9.js
Artist: The Beatles
Title: A Hard Day's Night
--------
Artist: The Beatles
Title: Can't Buy Me Love
--------
Artist: The Beatles
Title: Eight Days a Week
--------
Artist: The Beatles
Title: I Want to Hold Your Hand
--------
Artist: The Beatles
Title: We Can Work It Out
--------
```

5. Now, change the genreEquals variable to "pop":

```
var genreEquals = "pop";
```

6. Save your code and test again, noting that you now don't find any matches to the query.

## DIY: Search Star Wars Data

In this exercise you will build a search against the Star Wars data that you loaded during the last DIY section. Build the search to meet the following requirements:

1. Create a file called **unit07-diy.js** in your **hello** project at **c:\hello**.
2. Study some of the JSON data in your **star-wars-content** database to get familiar with its structure and contents.
3. Write a search that:
   - Returns at most the 5 most relevant results.
   - Only searches character documents.
   - Matches documents where the **alliance** property is equal to "**rebel**" and the **bio** property contains the word "**leia**".
   - Return the value of the **name**, **alliance**, and **bio** property for each matching document.
4. Use the examples and any other supporting materials to help you achieve this goal.