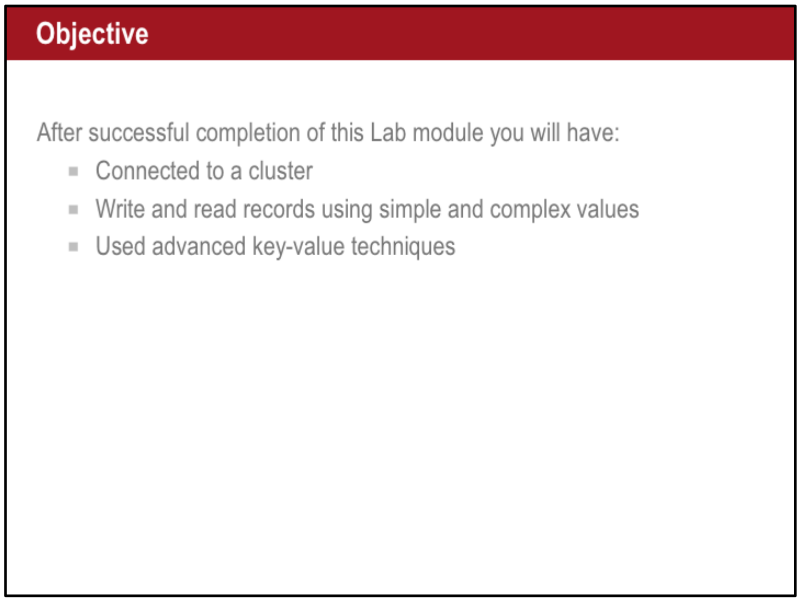
AS101 Node.js Client

Lab Exercises

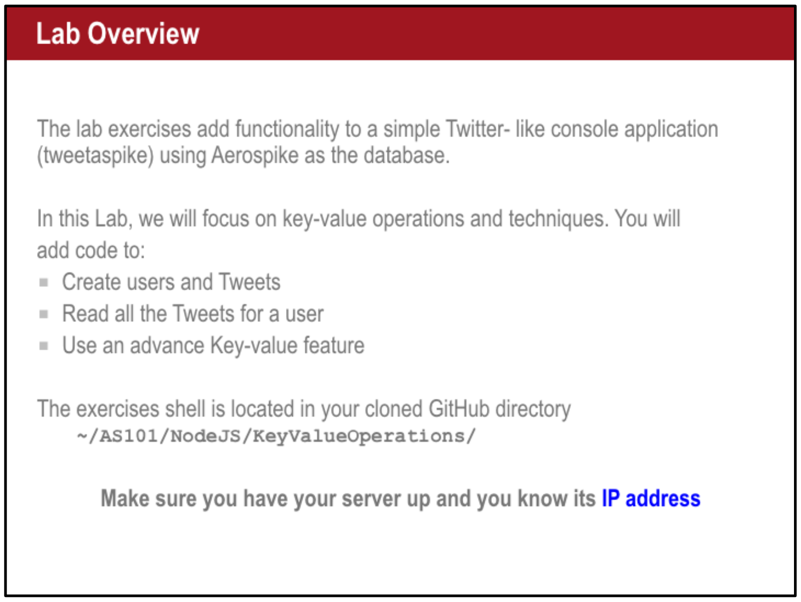
Lab: Key-value Operations



Objective

After successful completion of this Lab module you will have:

- Connected to a cluster
- Write and read records using simple and complex values
- Used advanced key-value techniques



Lab Overview

The lab exercises add functionality to a simple Twitter- like console application (tweetaspike) using Aerospike as the database.

In this Lab, we will focus on key-value operations and techniques. You will add code to:

- Create users and Tweets
- Read all the Tweets for a user
- Use an advance Key-value feature

The exercises shell is located in your cloned GitHub directory

```
~/AS101/NodeJS/KeyValueOperations/
```

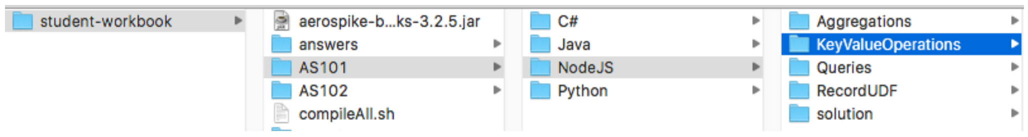
Make sure you have your server up and you know its IP address

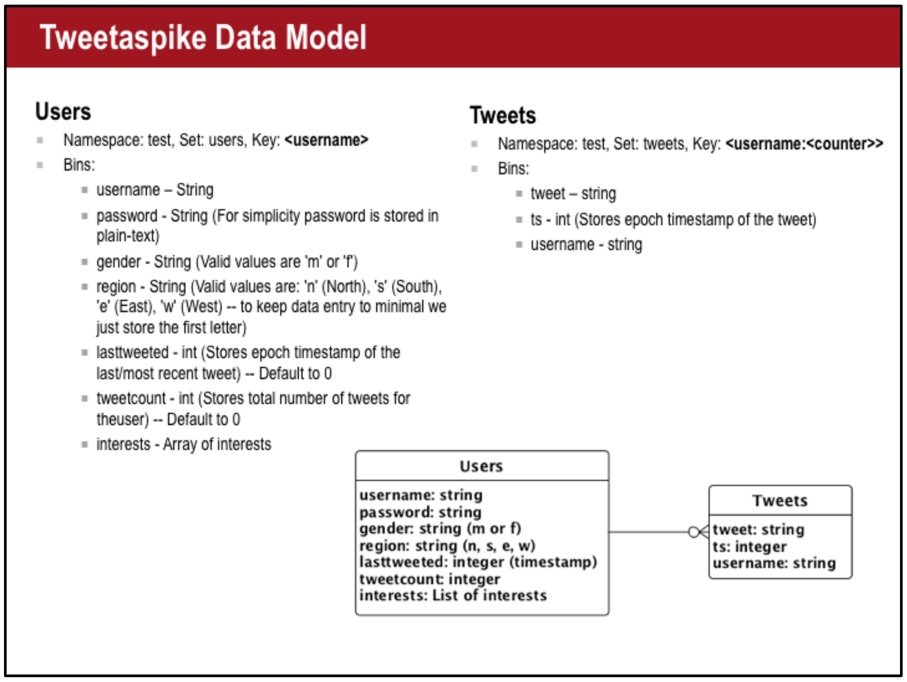
In your cloned or downloaded repository, you will find the following directories:

- AS101
- In the AS101 directory, select the subdirectory for your programming language:
- C#
- Java
- Node.js
- Python

The exercises for this module are in the KeyValueOperations sub-directory and your will find a Project/Solution/Codebase that is partly complete. Your tasks is to complete the code as outlined in each exercise.

Make sure you have your server up and you know its IP address





Tweetaspike Data Model

Users

- Namespace: test, Set: users, Key: **<username>**
- Bins:
  - username – String
  - password - String (For simplicity password is stored in plain-text)
  - gender - String (Valid values are 'm' or 'f')
  - region - String (Valid values are: 'n' (North), 's' (South), 'e' (East), 'w' (West) -- to keep data entry to minimal we just store the first letter)
  - lasttweeted - int (Stores epoch timestamp of the last/most recent tweet) -- Default to 0
  - tweetcount - int (Stores total number of tweets for theuser) -- Default to 0
  - interests - Array of interests

Tweets

- Namespace: test, Set: tweets, Key: **<username:<counter>>**
- Bins:
  - tweet – string
  - ts - int (Stores epoch timestamp of the tweet)
  - username - string

Users

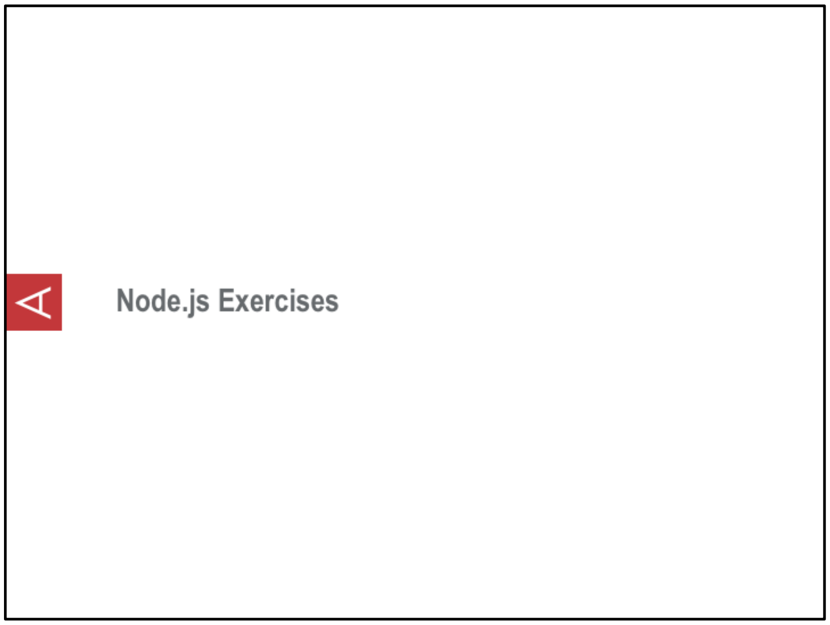
Namespace: test, Set: users, Key: <username>

Bin name	Type	Comment
username	String	
password	String	For simplicity password is stored in plain-text
region	String	Valid values are: 'n' (North), 's' (South), 'e' (East), 'w' (West) -- to keep data entry to minimal we just store the first letter
lasttweeted	Integer	Stores epoch timestamp of the last/most recent tweet -- Default to 0
tweetcount	Integer	Stores total number of tweets for the user – Default 0
Interests	List	A list of interests

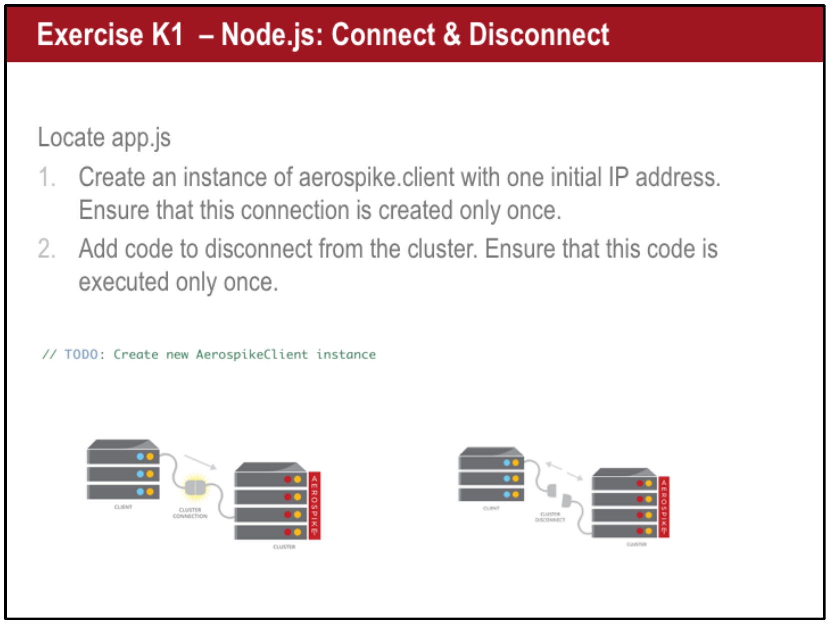
Tweets

Namespace: test, Set: tweets, Key: <username:<counter>>

Bin name	Type	Comment
tweet	String	Tweet text
ts	Integer	Stores epoch timestamp of the tweet
username	String	User name of the tweeter



Node.js Exercises



Exercise K1 – Node.js: Connect & Disconnect

Locate app.js

1. Create an instance of aerospike.client with one initial IP address. Ensure that this connection is created only once.
2. Add code to disconnect from the cluster. Ensure that this code is executed only once.

```
// TODO: Create new AerospikeClient instance
```

CLIENT
CLUSTER CONNECTION
CLUSTER
AEROSPIKE
CLIENT
CLUSTER DISCONNECT
CLUSTER
AEROSPIKE

In this exercise you will connect to a Cluster by creating an Aerospike client instance, passing a single IP address and port to the constructor. The IP address and port should be to a node in your own cluster.

Ensure that you only create one client instance at the start of the program. The Aerospike client is thread safe and creates a pool of worker threads, this means you DO NOT need to create your own connection or thread pool.

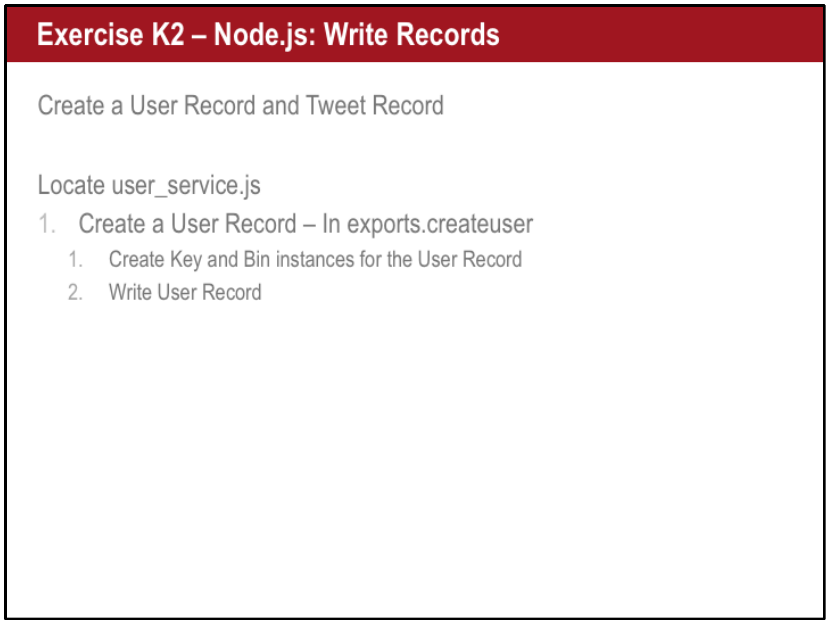
1. In app.js add code similar to this;

```
// Connect to the Aerospike Cluster
var client = aerospike.client({
    hosts: [ { addr: hostaddr, port: 3000 }]
}).connect( function(response) {
    // Check for errors
    // Exercise K1
    if ( response == null ) {
      // Connection succeeded
      console.log("Connection to the Aerospike cluster succeeded!");
    }
    else {
      // Connection failed
      console.log("Connection to the Aerospike cluster failed. Please check cluster IP and Port
settings and try again.");
      process.exit(0);
    }
});
```

Make sure you have your server up and you know its IP address

2. Add a *process.on* exit function and call close() to disconnect from the cluster. This should only be done once. After close() is called, the client instance cannot be used.

```
// Setup tear down
process.on('exit', function() {
  if (client != null) {
    client.close();
    console.log("Connection to Aerospike cluster closed!");
    aerospike.releaseEventLoop();
    console.log("Event loop released.");}
});
```



Exercise K2 – Node.js: Write Records

Create a User Record and Tweet Record

Locate user_service.js

1. Create a User Record – In exports.createuser
   1. Create Key and Bin instances for the User Record
   2. Write User Record

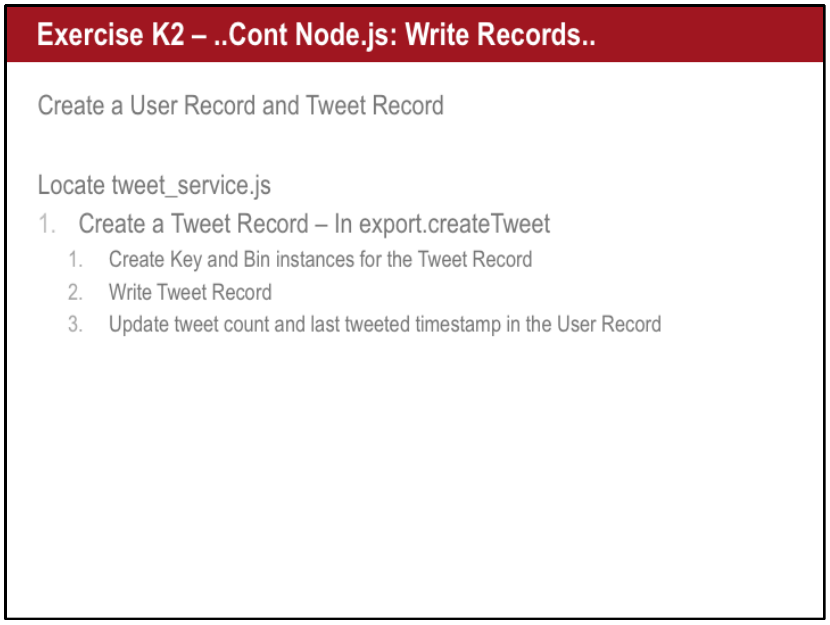
Create a User Record. In exports.createUser, add code similar to this:

1. Create Key and Bin instances

```
var key = {
  ns:  "test",
  set: "users",
  key: answers.username
};

var recBins = {
        username: answers.username,
        password: answers.password,
        gender: answers.gender,
        region: answers.region,
        lasttweeted: 0,
        tweetcount: 0,
        interests: answers.interests.split(",")
      };
```

1. Write a user record using the Key and Bins

```
client.put(key, recBins, function(err, recKey) {
  // Check for errors
  // Exercise K2
  if ( err == null ) {
    console.log("INFO: User "+recKey['key']+" record created!");
  }
  else {
    console.log("ERROR: createUser(): User record not created!");
    console.log(err);
  }
  callback();
});
```



Exercise K2 – ..Cont Node.js: Write Records..

Create a User Record and Tweet Record

Locate tweet_service.js

1. Create a Tweet Record – In export.createTweet
   1. Create Key and Bin instances for the Tweet Record
   2. Write Tweet Record
   3. Update tweet count and last tweeted timestamp in the User Record

Create a Tweet Record. In exports.createTweet add code similar to this:

1. Create Key and Bin instances

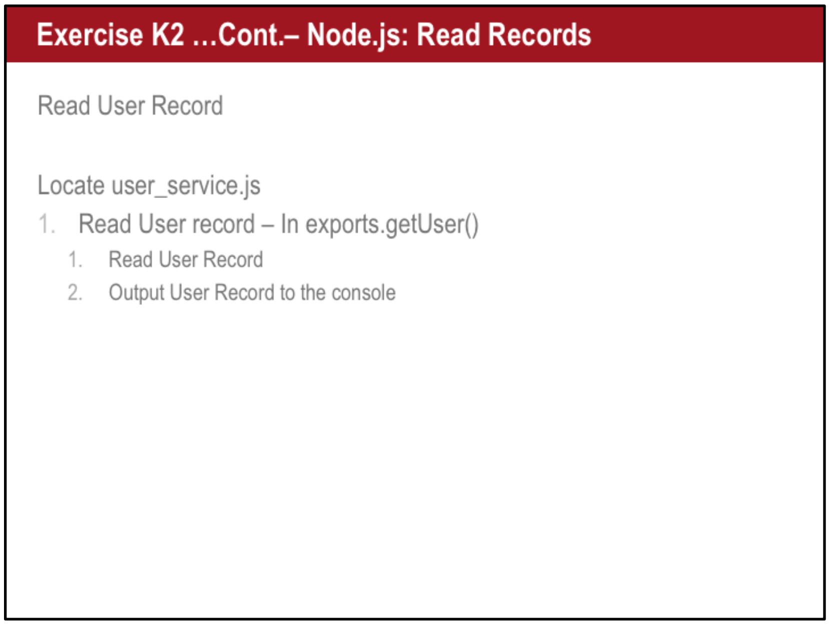
```
// Write Tweet record
var tweet_key = {
  ns:  "test",
  set: "tweets",
  key: userrecord.username + ":" + tweet_count
};

var recBins = {
  username: userrecord.username,
  tweet: answer.tweet,
  ts: ts
};
```

2. Write a tweet record using the Key and Bins

```
client.put(tweet_key, recBins, function(err, recKey) {
  // Check for errors
  // Exercise K2
  if ( err == null ) {
    console.log("INFO: Tweet record created!");
    // Update tweetcount and last tweet'd timestamp in the user record
    // Exercise K2  - add code to updateUser()
    updateUser(client, user_key, ts, tweet_count, callback);
  } else {
    console.log("ERROR: Tweet record not created!");
    console.log("",err);
    callback();}
 });
```

3. Update the user record with tweet count

```
//updateUser()
var bins = {
    lasttweeted: ts,
    tweetcount: tweet_count
  };

client.put(user_key, bins, function(err, recKey) {
    // Check for errors
    if ( err == null ) {
      console.log("INFO: User " + recKey['key'] +" record updated!");
      callback();
    }
    else {
      console.log("ERROR: User record not updated!");
      console.log(err);
      callback();
    }
  });
```



Exercise K2 …Cont.– Node.js: Read Records

Read User Record

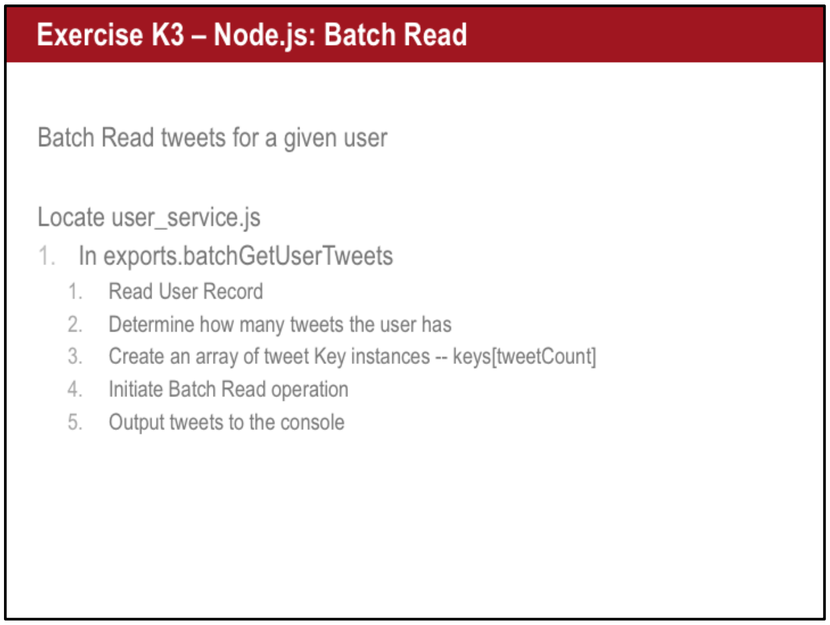
Locate user_service.js

1. Read User record – In exports.getUser()
   1. Read User Record
   2. Output User Record to the console

Read a User Record. In exports.getUser, add code, similar to this, to:

1. Read a User record
2. Output the User record to the console

```
// Read User record
var key = {
  ns:  "test",
  set: "users",
  key: answer.username
};

client.get(key, function(err, rec, meta) {
  // Check for errors (ie does the user record exist?)
  // Exercise K2
  if ( err == null ) {
    // Print user info to console in: "<bin>" : value format
    // Exercise K2
    console.log("INFO: User record read successfully! Here are the details:");
    console.log("username:   " + rec.username);
    console.log("password:   " + rec.password);
    console.log("gender:     " + rec.gender);
    console.log("region:     " + rec.region);
    console.log("tweetcount: " + rec.tweetcount);
    console.log("lasttweeted: " + rec.lasttweeted);
    console.log("interests:  " + rec.interests);
  }
  else {
    console.log("ERROR: User record not found!");
  }
  callback();
});
```



Exercise K3 – Node.js: Batch Read

Batch Read tweets for a given user

Locate user_service.js

1. In exports.batchGetUserTweets
   1. Read User Record
   2. Determine how many tweets the user has
   3. Create an array of tweet Key instances -- keys[tweetCount]
   4. Initiate Batch Read operation
   5. Output tweets to the console

Read all the tweets for a given user. In the function exports.batchGetUserTweets, add code similar to this:

1. Read a user record

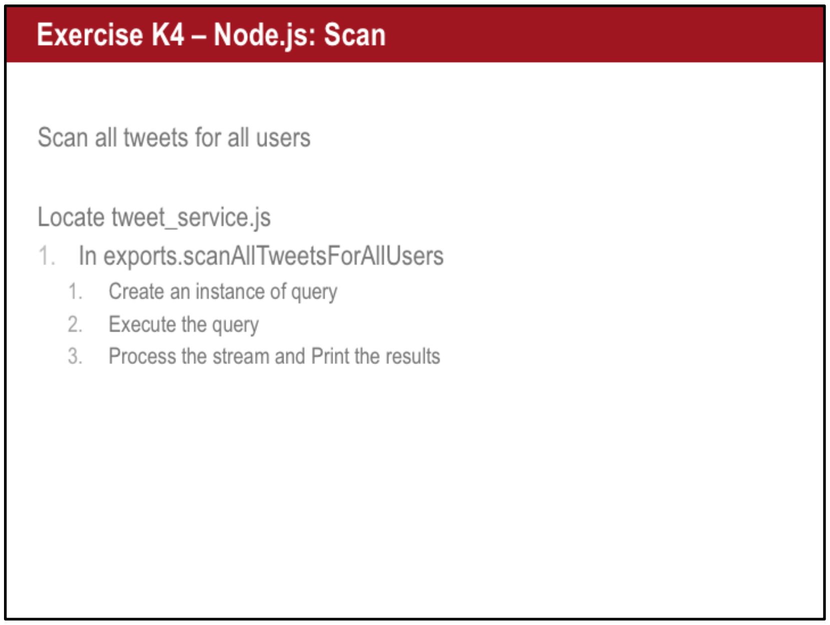
```
// Read User record
var key = {
  ns:  "test",
  set: "users",
  key: answer.username
};
```

2. Get the tweet count
3. Create a “list” of tweet keys

```
client.get(key, function(err, userrecord, meta) {
  // Check for errors
  if ( err == null ) {
    var tweet_count = userrecord.tweetcount;

    var tweet_keys = [];
    for(var i=1;i<=tweet_count;i++)  {
      tweet_keys.push({ns: "test", set: "tweets", key: answer.username + "
    }
```

4. Perform a Batch operation to read all the tweets & print out the tweets

```
    client.batchGet(tweet_keys, function (err, results) {
      // Check for errors
      if ( err == null ) {
        // Print out the tweets retreived
        // Exercise K3
        for(var j=0;j<results.length;j++)  {
          console.log(results[j].record.tweet);
        }
      } else {
        console.log("ERROR: Batch Read Tweets For User failed\n", err);
      }
      callback();  // Batch read returns all records together
    });
```



Exercise K4 – Node.js: Scan

Scan all tweets for all users

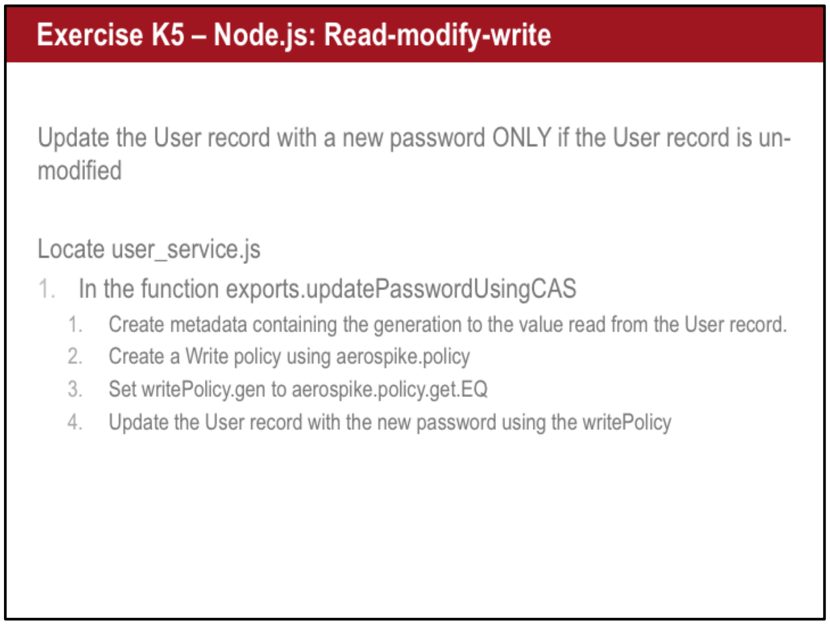
Locate tweet_service.js

1. In exports.scanAllTweetsForAllUsers
   1. Create an instance of query
   2. Execute the query
   3. Process the stream and Print the results

Scan all the tweets for all users – warning – there could be a large result set.

In the function exports.scanAllTweetsForAllUsers, add code similar to this:

1. Create a instance of a query
```
var query = client.query('test', 'tweets');
```
2. Execute the query
```
var stream = query.execute();
```
3. Process the stream and print the results
```
stream.on('data', function(record)  {
    // Handle data event.
    // Exercise K4
    console.log("("+ record['username'] +"):"+record['tweet']);
  });
  stream.on('error', function(err)  {
    // Handle error event.
    // Exercise K4
    console.log('ERROR: Scan All Tweets For All Users failed: ',err);
    callback();
  });
  stream.on('end', function()  {
    // Handle end event.
    // Exercise K4
    console.log('INFO: Scan All Tweets For All Users completed!');
    callback();
  });
```



Exercise K5 – Node.js: Read-modify-write

Update the User record with a new password ONLY if the User record is un-modified

Locate user_service.js

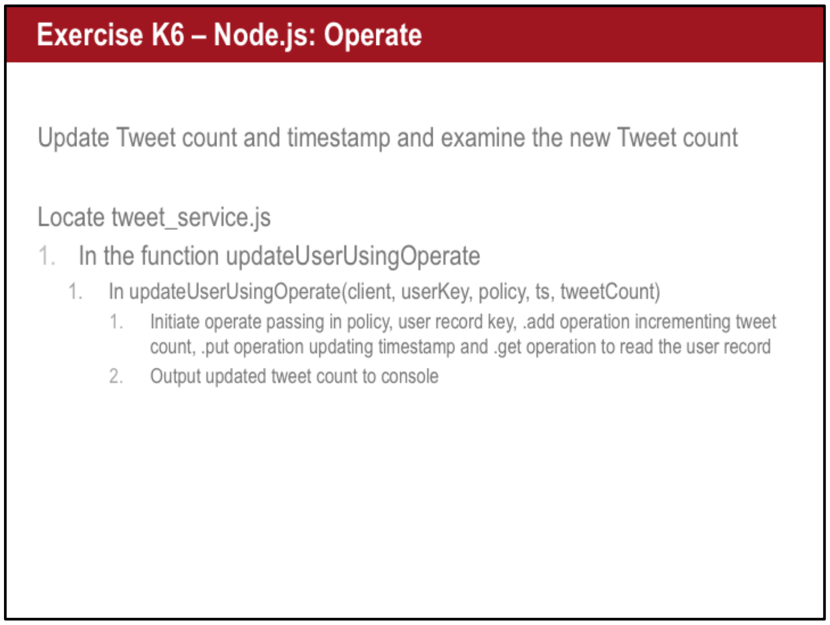
1. In the function exports.updatePasswordUsingCAS
   1. Create metadata containing the generation to the value read from the User record.
   2. Create a Write policy using aerospike.policy
   3. Set writePolicy.gen to aerospike.policy.get.EQ
   4. Update the User record with the new password using the writePolicy

Update the User record with a new password ONLY if the User record is un-modified

In exports.updatePasswordUsingCAS, add code similar to this:

1. Create metadata containing the generation to the value read from the User record.
2. Create a Write policy using aerospike.policy
3. Set writePolicy.gen to aerospike.policy.get.EQ
4. Update the User record with the new password using the writePolicy

```
// Set generation value to what we just read
// Exercise K5
var metadata = { gen: meta.gen };

// Set write Policy parameters
// Set write policy for gen to be aerospike.policy.gen.EQ
var writePolicy = {
    gen : aerospike.policy.gen.EQ,
    key : aerospike.policy.key.SEND,
    retry : aerospike.policy.retry.NONE,
    exists : aerospike.policy.exists.IGNORE,
    commitLevel : aerospike.policy.commitLevel.ALL
 };
 // Set new password in password bin for the user record
 var recordObj = { password: answer2.password };

// Write the record update. Handle errors.
client.put(key, recordObj, metadata, writePolicy, function(err, recKey) {
  // Check for errors
  if ( err == null ) {
     console.log("INFO: User password updated successfully!");
  } else {
     console.log("ERROR: User password update failed:\n", err);
  }
  callback();
});
```



Exercise K6 – Node.js: Operate

Update Tweet count and timestamp and examine the new Tweet count

Locate tweet_service.js

1. In the function updateUserUsingOperate
   1. In updateUserUsingOperate(client, userKey, policy, ts, tweetCount)
      1. Initiate operate passing in policy, user record key, .add operation incrementing tweet count, .put operation updating timestamp and .get operation to read the user record
      2. Output updated tweet count to console

Aerospike can perform multiple operations on a record in one transaction. Update the tweet count and timestamp in a user record and read the new tweet count.

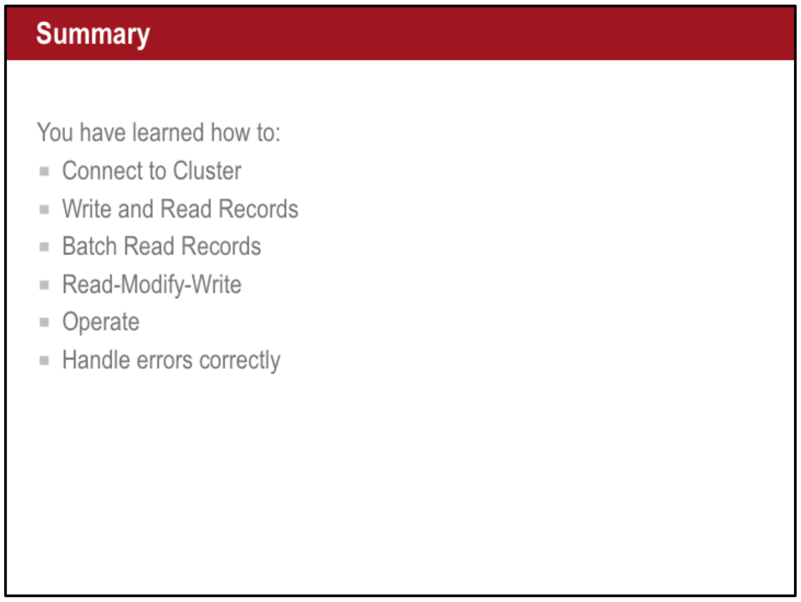
In updateUser

1. Comment out the code added in Exercise K2
2. Uncomment the line:

```
// Exercise K6, uncomment line below
// In Operate, we will use the increment operation, so don't need tweet_count
//updateUserUsingOperate(client, user_key, ts, callback);
```

3. In updateUserUsingOperate , add code similar to this:

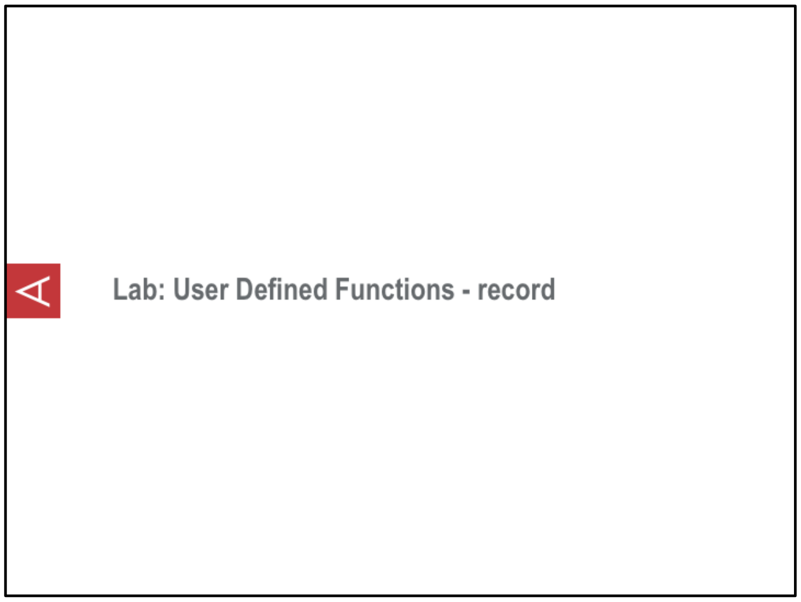
```
// Update User record
// User Operate() to set and get tweetcount
// Exercise K6
var operator = aerospike.operator;
var operations = [operator.incr('tweetcount', 1),operator.write('lasttweeted',
ts),operator.read('tweetcount')];

client.operate(user_key, operations, function(err, record, metadata, key) {
    // Check for errors
    if ( err == null ) {
      console.log("INFO: (Operate) The tweet count now is: " + record.tweetcount);
    }
    else {
      console.log("ERROR: User record not updated!");
      console.log(err);
    }
    callback();
});
```



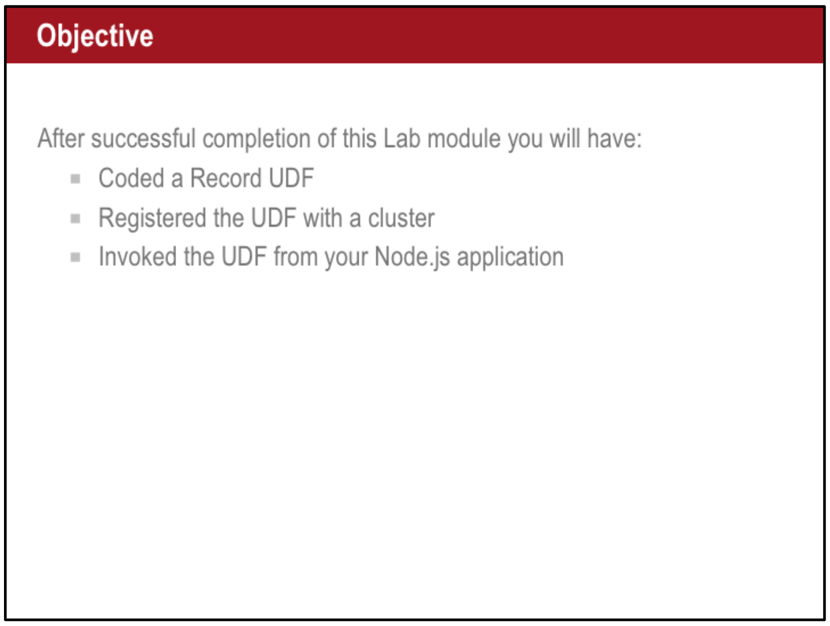
Summary

You have learned how to:

- Connect to Cluster
- Write and Read Records
- Batch Read Records
- Read-Modify-Write
- Operate
- Handle errors correctly



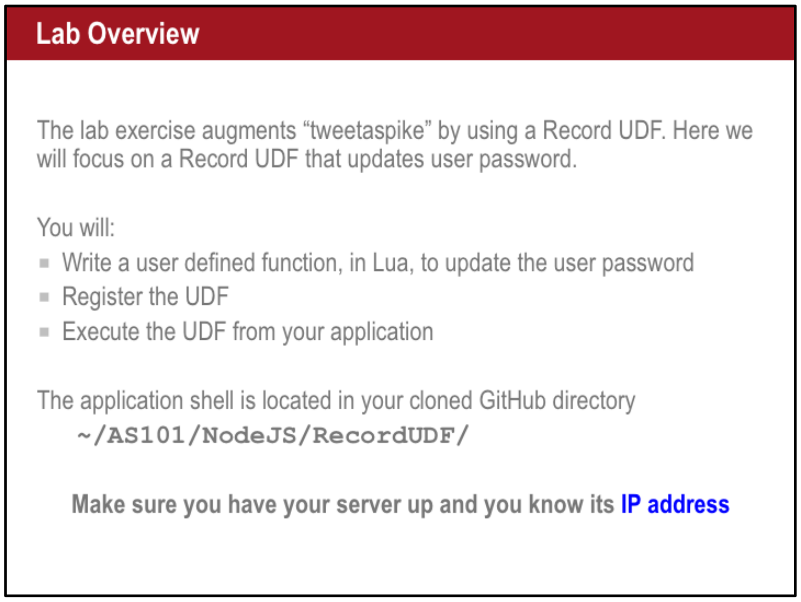
Lab: User Defined Functions - record



Objective

After successful completion of this Lab module you will have:

- Coded a Record UDF
- Registered the UDF with a cluster
- Invoked the UDF from your Node.js application



Lab Overview

The lab exercise augments "tweetaspike" by using a Record UDF. Here we will focus on a Record UDF that updates user password.

You will:

- Write a user defined function, in Lua, to update the user password
- Register the UDF
- Execute the UDF from your application

The application shell is located in your cloned GitHub directory

```
~/AS101/NodeJS/RecordUDF/
```

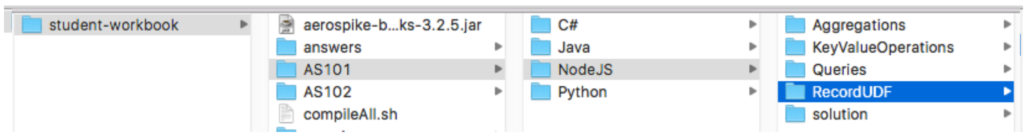
Make sure you have your server up and you know its IP address

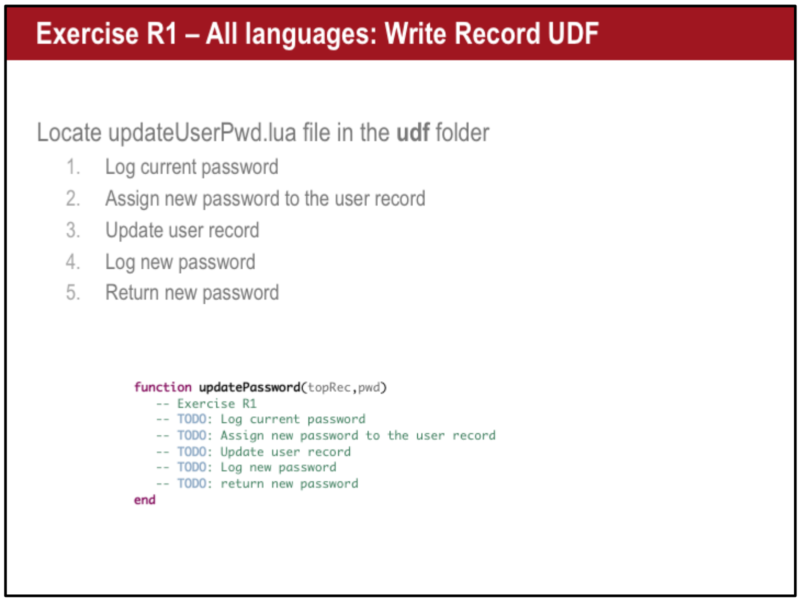
In your cloned or downloaded repository, you will find the following directories:

- AS101
- In the AS101 directory, select the subdirectory for your programming language:
- C#
- Java
- Node.js
- Python

The exercises for this module are in the RecordUDF sub-directory and your will find a Project/Solution/Codebase that is partly complete. Your tasks is to complete the code as outlined in each exercise.

Make sure you have your server up and you know its IP address





Exercise R1 – All languages: Write Record UDF

Locate updateUserPwd.lua file in the **udf** folder

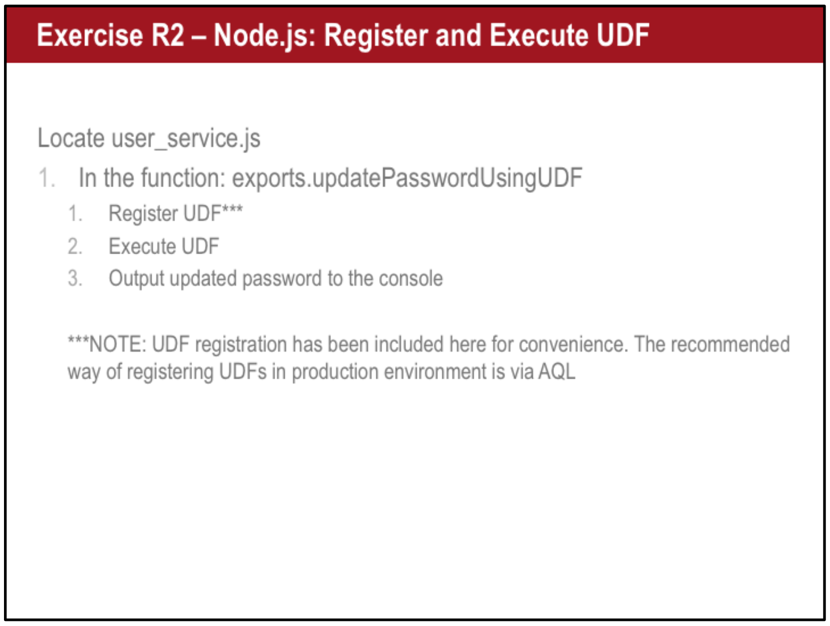
1. Log current password
2. Assign new password to the user record
3. Update user record
4. Log new password
5. Return new password

```
function updatePassword(topRec,pwd)
   -- Exercise R1
   -- TODO: Log current password
   -- TODO: Assign new password to the user record
   -- TODO: Update user record
   -- TODO: Log new password
   -- TODO: return new password
end
```

In this exercise you will create a record UDF that:

1. Logs the current password
2. Assigns a new password to the record, passed in via the pwd parameter
3. Updates the user record by calling **aerospike:update**(topRec)
4. Logs the new password
5. Returns the new password to the client

```
function updatePassword(topRec,pwd)
   -- Log current password
   debug("current password: " .. topRec['password'])
   -- Assign new password to the user record
   topRec['password'] = pwd
   -- Update user record
   aerospike:update(topRec)
   -- Log new password
   debug("new password: " .. topRec['password'])
   -- return new password
   return topRec['password']
end
```



Exercise R2 – Node.js: Register and Execute UDF

Locate user_service.js

1. In the function: exports.updatePasswordUsingUDF
   1. Register UDF***
   2. Execute UDF
   3. Output updated password to the console

***NOTE: UDF registration has been included here for convenience. The recommended way of registering UDFs in production environment is via AQL

In this exercise you will register and invoke the UDF created in Exercise R1.

We will programmatically register the UDF for convenience.

In the function module.updatePasswordUsingUDF add your code:

1. Register the UDF with an API call

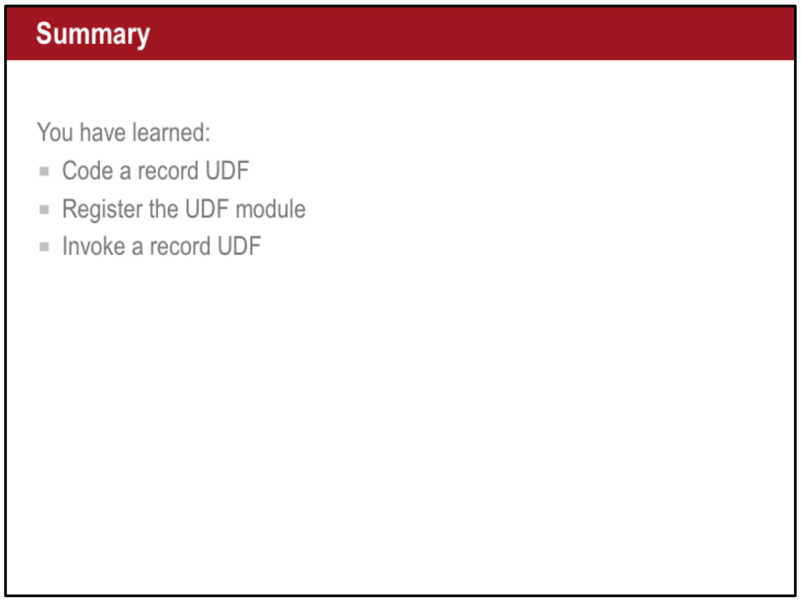
```
// Register UDF
// Exercise R2
client.udfRegister('udf/updateUserPwd.lua', function(err) {
if ( err == null ) {
  // Create UDF object for record udf execution
  // Exercise R2
  var UDF = {module:'updateUserPwd', funcname: 'updatePassword', args: [answers.password]};
```

2. Execute the UDF passing the new password, as a parameter, to the UDF

```
  // Execute Record UDF
  // Exercise R2
  client.execute(key, UDF, function(err) {
     // Check for errors
     if ( err == null ) {
       console.log("INFO: User password updated successfully!");
```

3. Output the return from the UDF to the console

```
       // Print updated password
       client.get(key, (err, rec, meta)=>{
         if(err==null){
           console.log("Password updated to: "+rec.password)
           callback();
         } else {
           console.log("ERROR: User password update failed\n", err);
           callback();
         }
       });
      } else {
        console.log("ERROR: User password update failed\n", err);
        callback(); }
  });
```

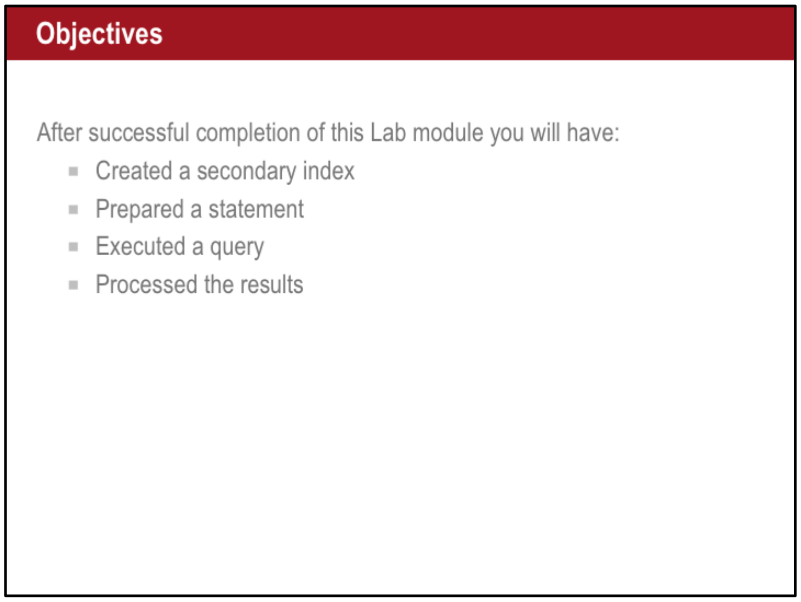


Summary

You have learned:

- Code a record UDF
- Register the UDF module
- Invoke a record UDF

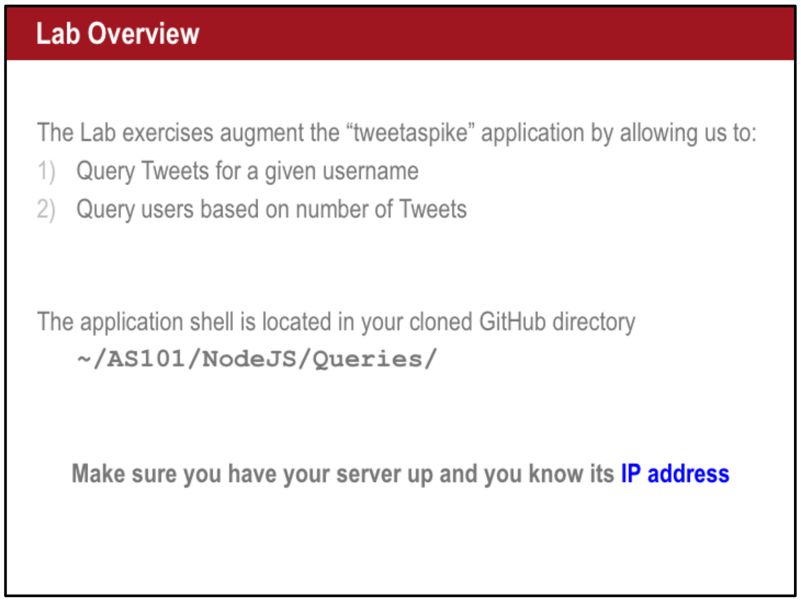
Lab: Queries



Objectives

After successful completion of this Lab module you will have:

- Created a secondary index
- Prepared a statement
- Executed a query
- Processed the results


Lab Overview

The Lab exercises augment the "tweetaspike" application by allowing us to:

1) Query Tweets for a given username
2) Query users based on number of Tweets

The application shell is located in your cloned GitHub directory

`~/AS101/NodeJS/Queries/`

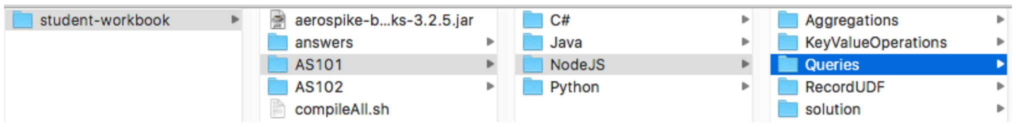
Make sure you have your server up and you know its IP address

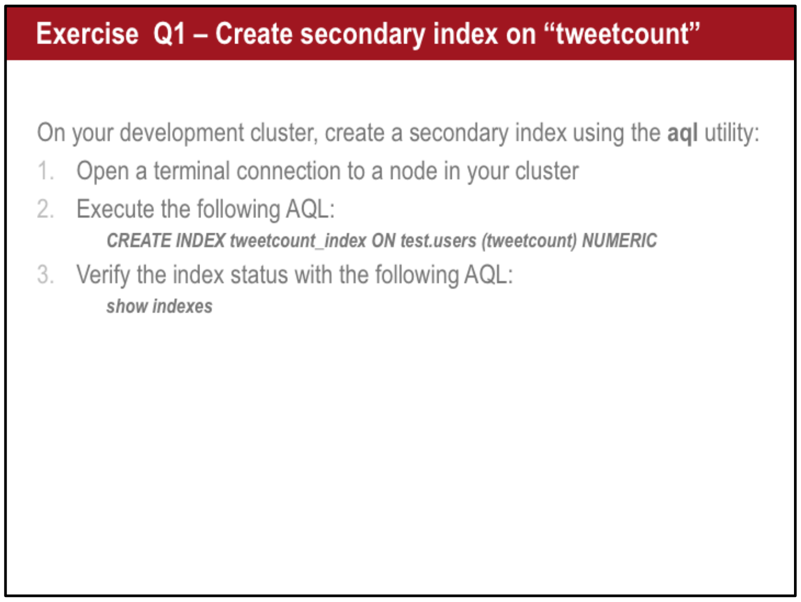
In your cloned or downloaded repository, you will find the following directories:

- AS101
- In the AS101 directory, select the subdirectory for your programming language:
- C#
- Java
- Node.js
- Python

The exercises for this module are in the Queries sub-directory and your will find a Project/Solution/Codebase that is partly complete. Your tasks is to complete the code as outlined in each exercise.

Make sure you have your server up and you know its IP address





Exercise Q1 – Create secondary index on “tweetcount”

On your development cluster, create a secondary index using the **aql** utility:

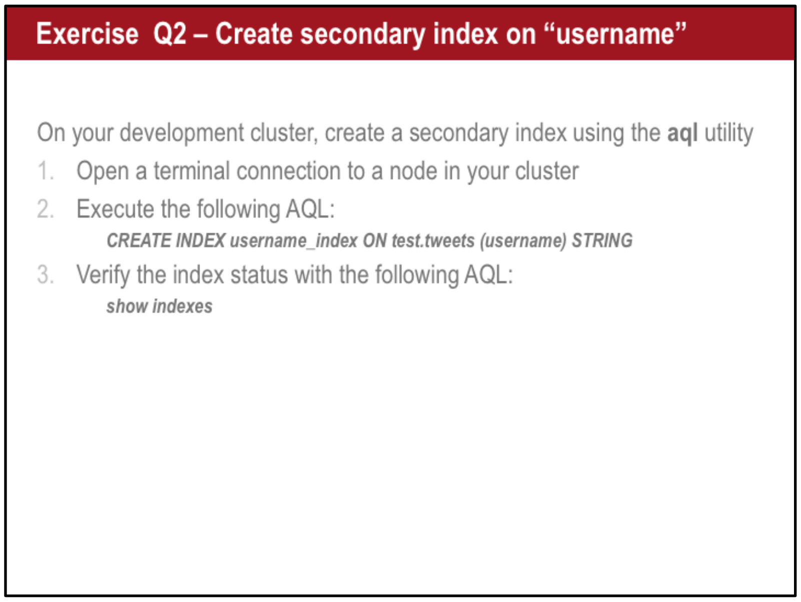
1. Open a terminal connection to a node in your cluster
2. Execute the following AQL:
   CREATE INDEX tweetcount_index ON test.users (tweetcount) NUMERIC
3. Verify the index status with the following AQL:
   show indexes

Logon on to your server instance and run **aql** to create a numeric index on *tweetcount*. At the prompt, enter the command:

CREATE INDEX tweetcount_index ON test.users (tweetcount) NUMERIC

Verify that the index has been created with the command:

show indexes



Exercise Q2 – Create secondary index on “username”

On your development cluster, create a secondary index using the **aql** utility

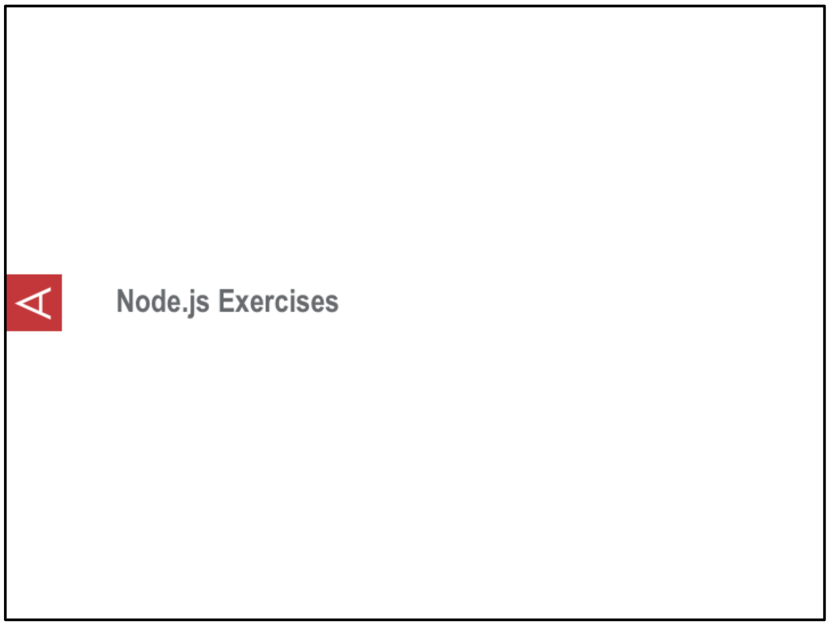
1. Open a terminal connection to a node in your cluster
2. Execute the following AQL:
   CREATE INDEX username_index ON test.tweets (username) STRING
3. Verify the index status with the following AQL:
   show indexes

Logon on to your server instance and run **aql** to create a string index on *username*. At the prompt, enter the command:

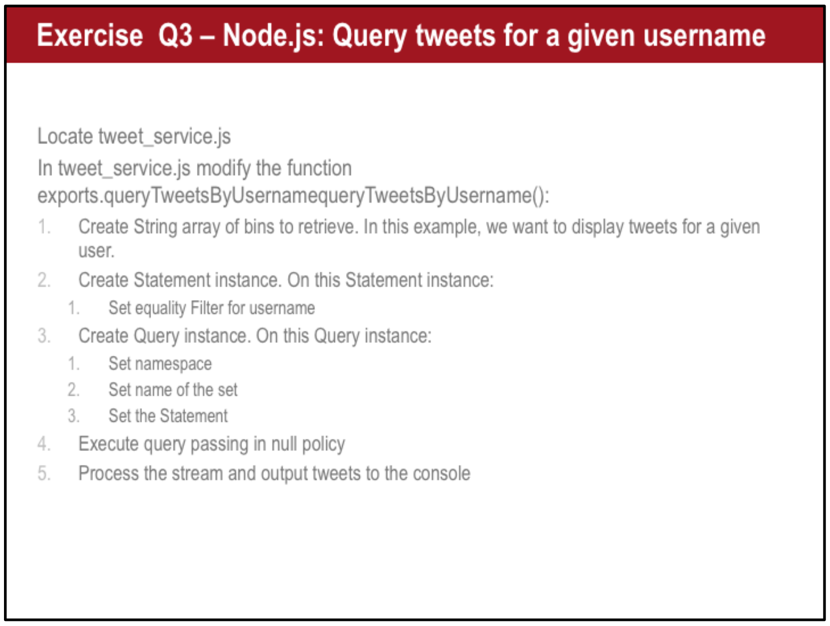
```
CREATE INDEX username_index ON test.tweets (username) STRING
```

Verify that the index has been created with the command:

```
show indexes
```



Node.js Exercises



Exercise Q3 – Node.js: Query tweets for a given username

Locate tweet_service.js

In tweet_service.js modify the function exports.queryTweetsByUsernamequeryTweetsByUsername():

1. Create String array of bins to retrieve. In this example, we want to display tweets for a given user.
2. Create Statement instance. On this Statement instance:
   1. Set equality Filter for username
3. Create Query instance. On this Query instance:
   1. Set namespace
   2. Set name of the set
   3. Set the Statement
4. Execute query passing in null policy
5. Process the stream and output tweets to the console

In the function: queryTweetsByUsername, add your code:

1. Create a statement with equality filter on bin username

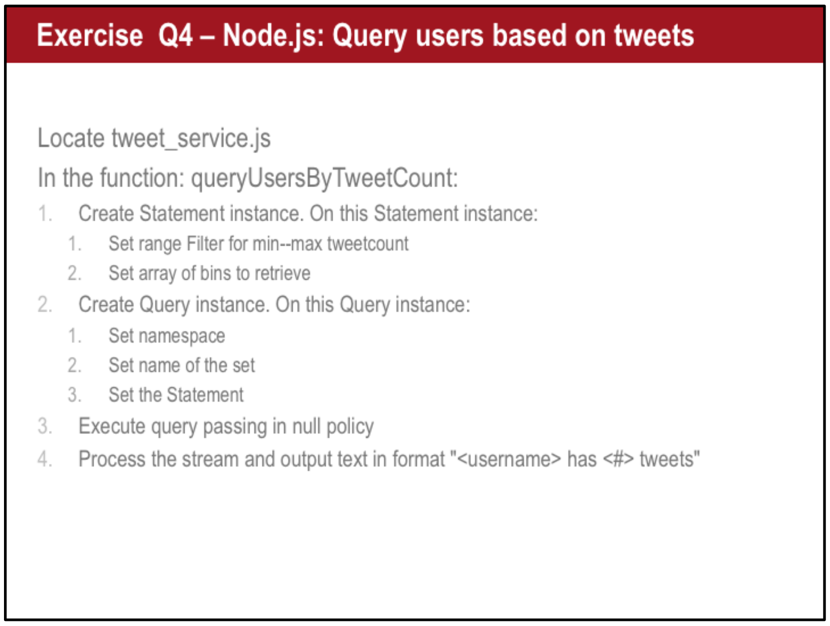
```
// Create Query and Set equality Filter on username
// Exercise Q3
var statement = {filters:[aerospike.filter.equal('username', answer.username)]};
```

2. Create a query with
   1. the Namespace
   2. the Set name
   3. the statement

```
var query = client.query('test', 'tweets', statement);
```

3. Execute the query from your code
4. Iterate through the RecordSet returned from the query

```
// Execute the query
// Exercise Q3
var stream = query.foreach(null);  //Query Policy = null
stream.on('data', function(record)  {
   // Handle 'data' event. Print Tweets for given Username
   // Exercise Q3
   console.log(record.tweet);
 });
 stream.on('error', function(err)  {
   // Handle 'error' event.
   // Exercise Q3
   console.log('ERROR: Query Tweets By Username failed: ',err);
   callback();
 });
 stream.on('end', function()  {
   // Handle 'end' event.
   // Exercise Q3
   console.log('INFO: Query Tweets By Username completed!');
   callback();
 });
```



Exercise Q4 – Node.js: Query users based on tweets

Locate tweet_service.js

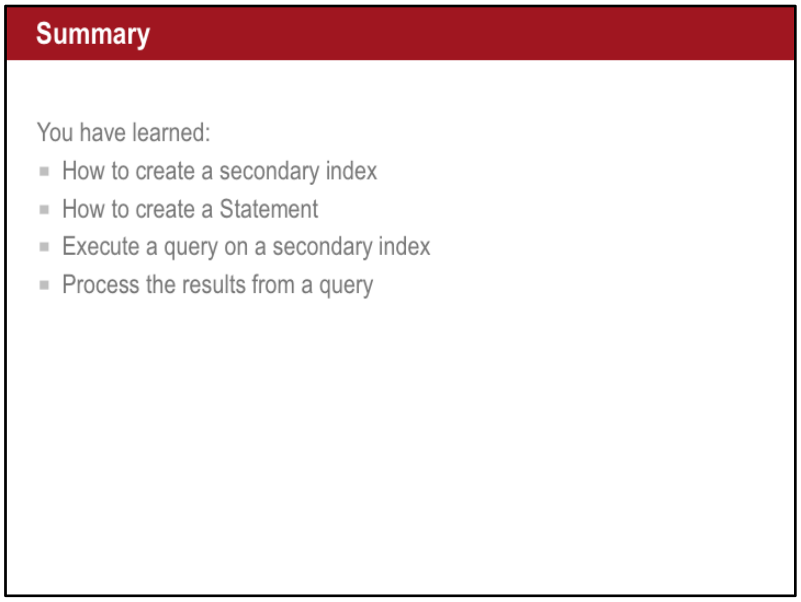
In the function: queryUsersByTweetCount:

1. Create Statement instance. On this Statement instance:
   1. Set range Filter for min--max tweetcount
   2. Set array of bins to retrieve
2. Create Query instance. On this Query instance:
   1. Set namespace
   2. Set name of the set
   3. Set the Statement
3. Execute query passing in null policy
4. Process the stream and output text in format "<username> has <#> tweets"

In the function exports.queryUsersByTweetCount, add your code:

1. Create Statement with:
   1. a range Filter for min--max tweetcount
   2. the bins to retrieve
2. Create querywith:
   1. the namespace
   2. the Set
   3. the statement
3. Execute query passing in null policy and instance of Statement created above
4. Process the stream and output text in format "<username> has <#> tweets"

```
// Prepare query statement - Set range Filter on tweetcount
// Exercise Q4
var statement = {filters:[aerospike.filter.range('tweetcount',
parseInt(answers.min), parseInt(answers.max))]};

// Select bins of interest to retrieve from the query
// Exercise Q4
statement.select = ['username', 'tweetcount'];

// Create query
// Exercise Q4
var query = client.query('test', 'users', statement);

// Execute the query
// Exercise Q4
var stream = query.foreach(null);  //Query Policy = null

// Handle 'data' event returned by the query
// Exercise Q4
stream.on('data', function(record)  {
   console.log(record.username + ' has ' + record.tweetcount +' tweets.');
});

// Handle 'error' event returned by the query
// Exercise Q4
stream.on('error', function(err)  {
  console.log('ERROR: Query Users By Tweet Count Range failed:\n',err);
  callback();
});

// Handle 'end' event returned by the query
// Exercise Q4
stream.on('end', function()  {
  console.log('INFO: Query Users By Tweet Count Range completed!');
  callback();
});
```

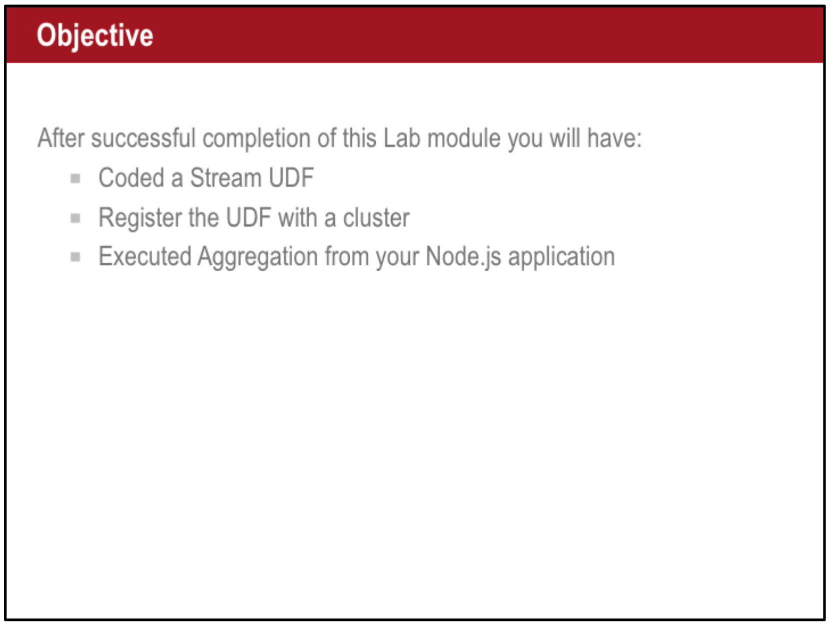


Summary

You have learned:

- How to create a secondary index
- How to create a Statement
- Execute a query on a secondary index
- Process the results from a query

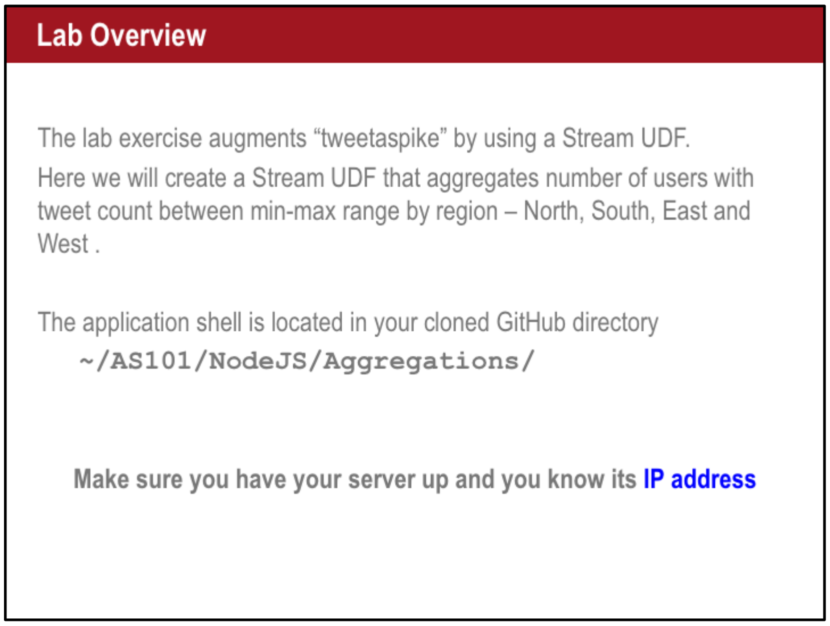
Lab: Aggregations



Objective

After successful completion of this Lab module you will have:

- Coded a Stream UDF
- Register the UDF with a cluster
- Executed Aggregation from your Node.js application


Lab Overview

The lab exercise augments "tweetaspike" by using a Stream UDF.

Here we will create a Stream UDF that aggregates number of users with tweet count between min-max range by region – North, South, East and West .

The application shell is located in your cloned GitHub directory

`~/AS101/NodeJS/Aggregations/`

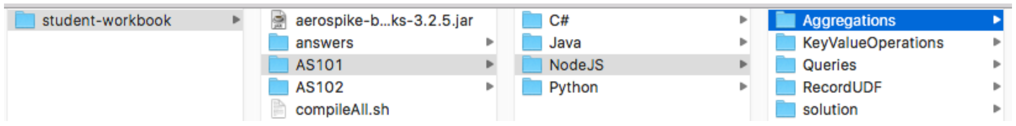
Make sure you have your server up and you know its IP address

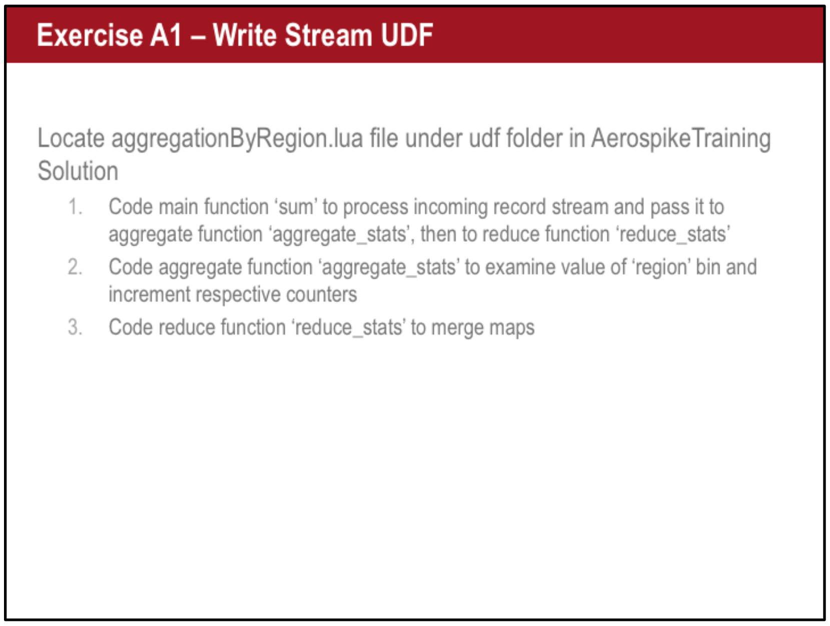
In your cloned or downloaded repository, you will find the following directories:

- AS101
- In the AS101 directory, select the subdirectory for your programming language:
- C#
- Java
- Node.js
- Python

The exercises for this module are in the Aggregations sub-directory and your will find a Project/Solution/Codebase that is partly complete. Your tasks is to complete the code as outlined in each exercise.

Make sure you have your server up and you know its IP address





Exercise A1 – Write Stream UDF

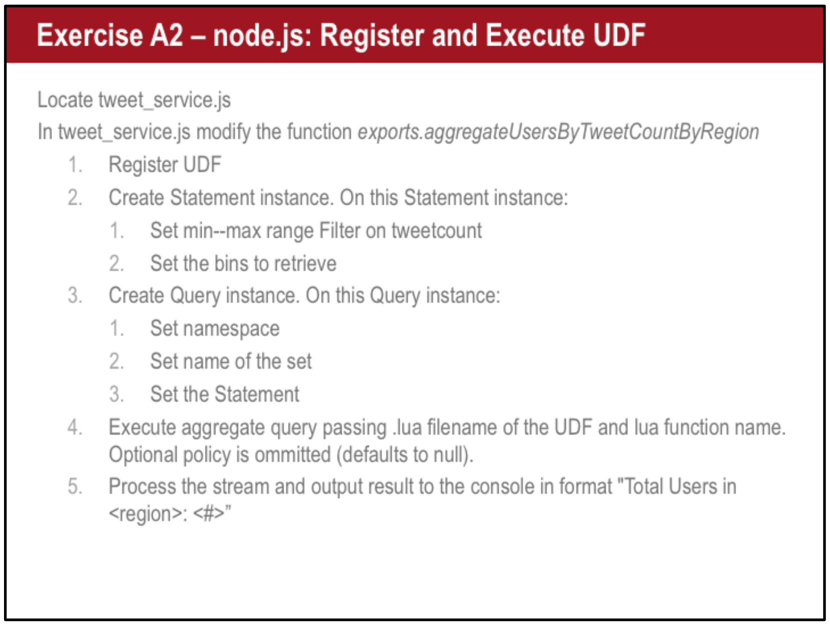
Locate aggregationByRegion.lua file under udf folder in AerospikeTraining Solution

1. Code main function 'sum' to process incoming record stream and pass it to aggregate function 'aggregate_stats', then to reduce function 'reduce_stats'
2. Code aggregate function 'aggregate_stats' to examine value of 'region' bin and increment respective counters
3. Code reduce function 'reduce_stats' to merge maps

In this exercise you will create a Stream UDF module that:

- Aggregates (sums) tweets by region – The aggregate_stats() function is invoked one for each element in the stream.
- Reduces the aggregations into a single Map of values – The reduce_stats() function is invoked once for each data partition, once for each node in the cluster, and finally once on the client.
- The sum() function configures the stream processing, and it is the function invoked by the Client.

```lua
local function aggregate_stats(map,rec)
  -- Examine value of 'region' bin in record rec and increment respective counter in the map
  if rec.region == 'n' then
      map['n'] = map['n'] + 1
  elseif rec.region == 's' then
      map['s'] = map['s'] + 1
  elseif rec.region == 'e' then
      map['e'] = map['e'] + 1
  elseif rec.region == 'w' then
      map['w'] = map['w'] + 1
  end
  -- return updated map
  return map
end
local function reduce_stats(a,b)
  -- Merge values from map b into a
  a.n = a.n + b.n
  a.s = a.s + b.s
  a.e = a.e + b.e
  a.w = a.w + b.w
  -- Return updated map a
  return a
end
function sum(stream)
  -- Process incoming record stream and pass it to aggregate function, then to reduce function
return stream : aggregate(map{n=0,s=0,e=0,w=0},aggregate_stats) : reduce(reduce_stats)
end
```



Exercise A2 – node.js: Register and Execute UDF

Locate tweet_service.js

In tweet_service.js modify the function *exports.aggregateUsersByTweetCountByRegion*

1. Register UDF
2. Create Statement instance. On this Statement instance:
   1. Set min--max range Filter on tweetcount
   2. Set the bins to retrieve
3. Create Query instance. On this Query instance:
   1. Set namespace
   2. Set name of the set
   3. Set the Statement
4. Execute aggregate query passing .lua filename of the UDF and lua function name. Optional policy is ommitted (defaults to null).
5. Process the stream and output result to the console in format "Total Users in <region>: <#>"

In this exercise you will register and invoke the UDF created in Exercise A1.

We will programmatically register the UDF for convenience.

In tweet_service.js modify the function *exports.aggregateUsersByTweetCountByRegion*, add your code to look like this:

1. Register the UDF with an API call

```
// Register UDF, if successful, prepare the aggregation query and execute it.
// Exercise A2
client.udfRegister('udf/aggregationByRegion.lua', function(err1) {
if ( err1 == null ) {
  // Prepare query statement - Set range Filter on tweetcount
  // Exercise A2
```

2. Prepare the Statement

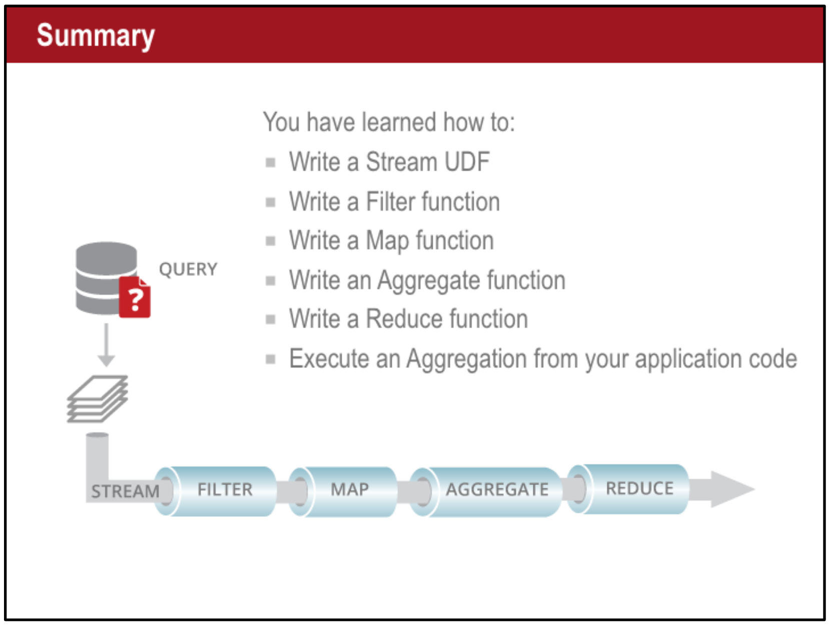
```
  var statement = {filters:[aerospike.filter.range('tweetcount', parseInt(answers.min),
parseInt(answers.max))]};
```

3. Execute the query

```
  // Create query
  // Exercise A2
  var query = client.query('test', 'users', statement);
```

4. Process the stream

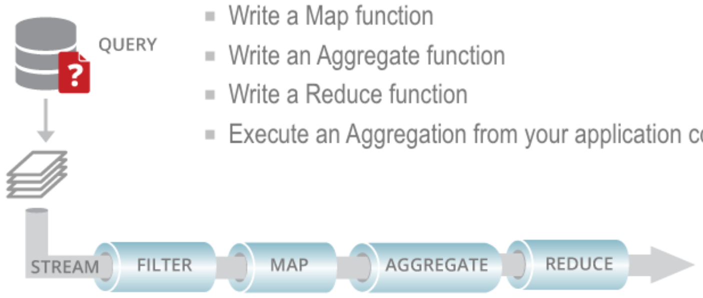
```
  //Or you can also use the construct below using 'where' to create the query object:
  //var query = client.query('test', 'users');
  //query.where(aerospike.filter.range('tweetcount', parseInt(answers.min), parseInt(answers.max)));

  // Execute the query, invoking stream Aggregation UDF on the results of the query
  // UDF returns aggregated result
  // Exercise A2
  query.apply('aggregationByRegion', 'sum', function(err2, result)  {
    if (err2 == null) {
      // Display desired result: "Total Users In <region>: <#>"
      // Exercise A2
      console.log('Total Users In East:  ', result.e);
      console.log('Total Users In West:  ', result.w);
      console.log('Total Users In North: ', result.n);
      console.log('Total Users In South: ', result.s);
      callback();
     } else {
      console.log('ERROR: Aggregation Based on Tweet Count By Region failed: ',err2);
      callback();
     }
  });  //query.apply()
}
```



Summary

You have learned how to:

- Write a Stream UDF
- Write a Filter function
- Write a Map function
- Write an Aggregate function
- Write a Reduce function
- Execute an Aggregation from your application code



AEROSPIKE