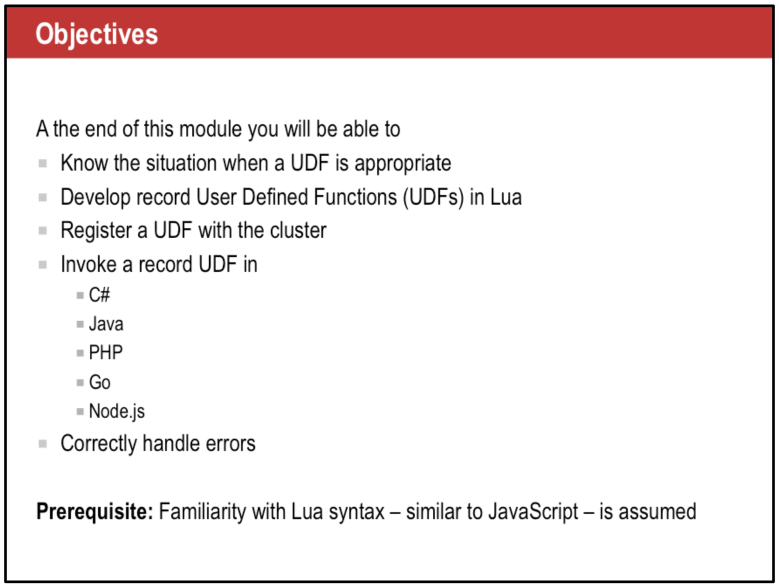
AEROSPIKE

# User Defined Functions

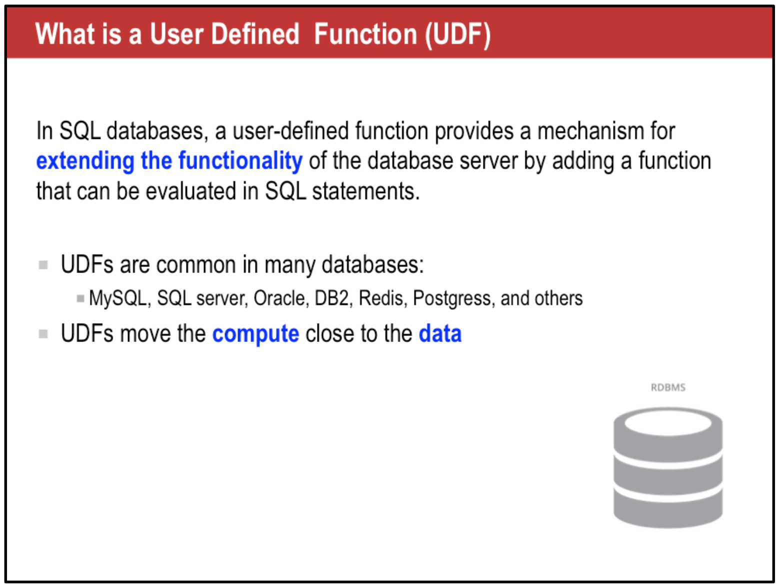


## Objectives

A the end of this module you will be able to

- Know the situation when a UDF is appropriate
- Develop record User Defined Functions (UDFs) in Lua
- Register a UDF with the cluster
- Invoke a record UDF in
  - C#
  - Java
  - PHP
  - Go
  - Node.js
- Correctly handle errors

**Prerequisite:** Familiarity with Lua syntax – similar to JavaScript – is assumed



## What is a User Defined Function (UDF)

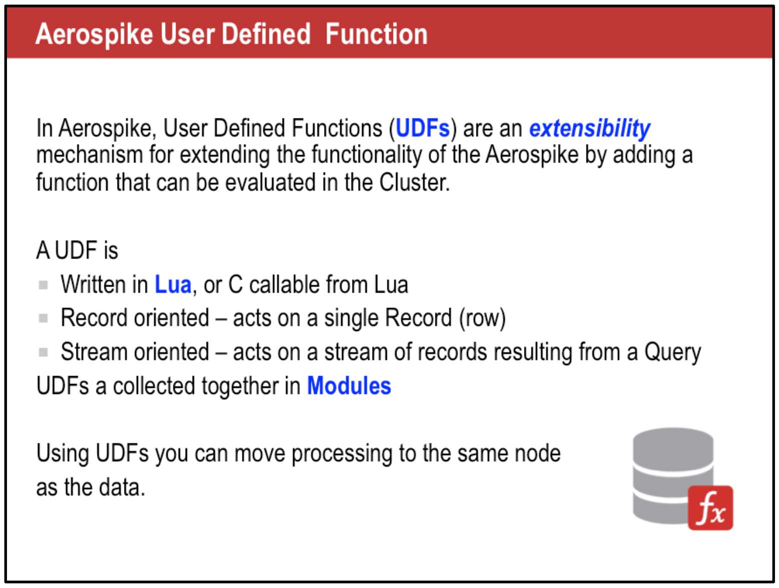
In SQL databases, a user-defined function provides a mechanism for **extending the functionality** of the database server by adding a function that can be evaluated in SQL statements.

- UDFs are common in many databases:
  - MySQL, SQL server, Oracle, DB2, Redis, Postgress, and others
- UDFs move the **compute** close to the **data**

**User Defined Function**

In SQL databases, a user-defined function provides a mechanism for extending the functionality of the database server by adding a function that can be evaluated in SQL statements.

UDFs allow the compute to move close to data. The function is run on the same server where the data resides.



## Aerospike User Defined Function

In Aerospike, User Defined Functions (**UDFs**) are an ***extensibility*** mechanism for extending the functionality of the Aerospike by adding a function that can be evaluated in the Cluster.

A UDF is

- Written in **Lua**, or C callable from Lua
- Record oriented – acts on a single Record (row)
- Stream oriented – acts on a stream of records resulting from a Query

UDFs a collected together in **Modules**

Using UDFs you can move processing to the same node as the data.

**User Defined Function**

A User-Defined Function (UDF) is a piece of code, written by a developer in the Lua programming language. It runs inside the cluster on a server node.

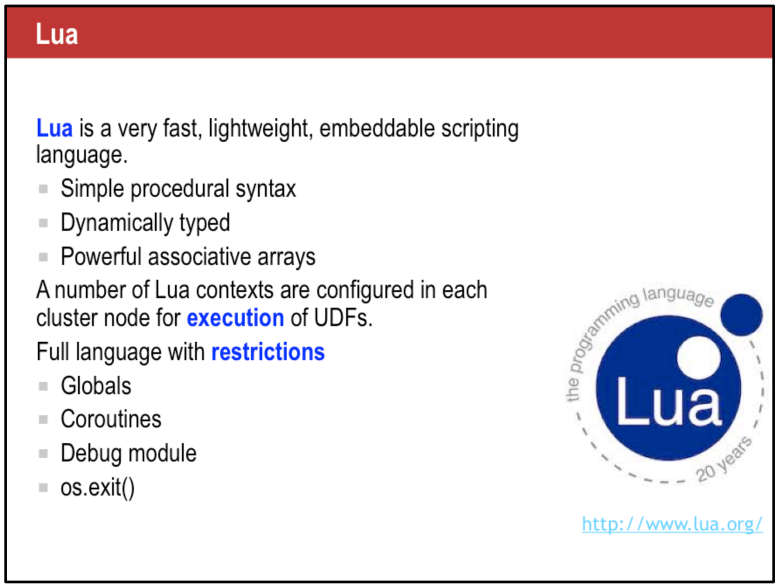
There are two types of UDFs in Aerospike: Record UDFs and Stream UDFs. A Record UDF operates on a single record. A Stream UDF (discussed in a later module) operates on a stream of records, and it can comprise multiple stream operators to perform very complex queries.

The complexity of UDFs can range from a single function that is only a few lines long, to a multi-thousand line module that contains many internal functions and multiple external functions.

When contemplating the construction of a new UDF, it is important to consider the application data model and the interaction between record bins. The general pattern (or life-cycle) for UDF development is:

- Design the application data model
- Design the UDFs to perform desired functions on the data model
- Create/Test the UDFs
- Register UDFs in the Aerospike Servers
- Iterate function test/development cycle
- System Test
- System Deployment

UDF design and development is usually an iterative activity, where the first version is simple and then, potentially, evolves over time to something complex.



## Lua

Lua is a very fast, lightweight, embeddable scripting language.

- Simple procedural syntax
- Dynamically typed
- Powerful associative arrays

A number of Lua contexts are configured in each cluster node for **execution** of UDFs.

Full language with **restrictions**

- Globals
- Coroutines
- Debug module
- os.exit()

http://www.lua.org/

**Lua**

"Lua" means "Moon" in Portuguese.

Lua combines simple procedural syntax with powerful data description constructs based on associative arrays* and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection

**Restrictions**

UDFs can use the full Lua programming language with a few restrictions

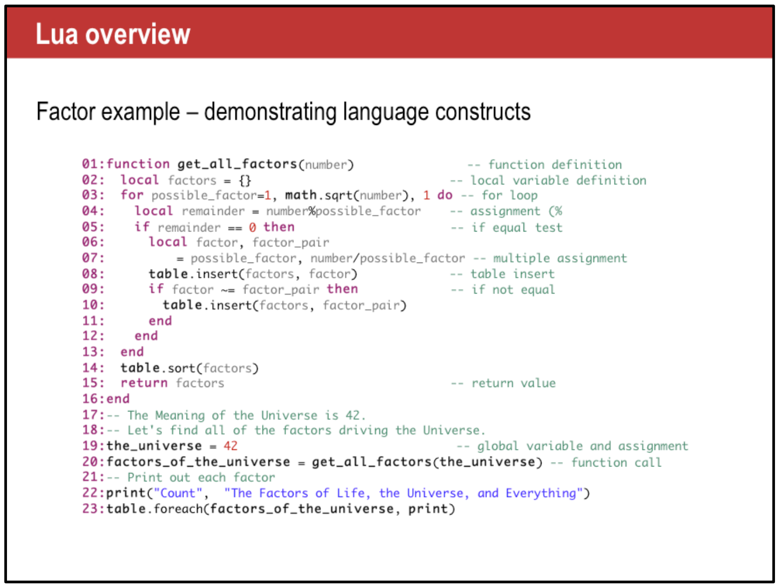
**Globals:** Global variables are not allowed. Global functions can only be called by Aerospike Server, but not by other Lua functions. To call a custom Lua function from another Lua function, you will need to declare it as local function.

**Coroutines** - Lua functions function cooperative multi-tasking.

**Debug module** – Not enabled due to not being able to support the debugging features in Lua.

**os.exit()** – Not enabled because it didn't make sense for a Lua script to cause the database to process to exit.

*associative array, map, symbol table, or dictionary is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.



## Lua overview

Factor example – demonstrating language constructs

```
01:function get_all_factors(number)                -- function definition
02:  local factors = {}                           -- local variable definition
03:  for possible_factor=1, math.sqrt(number), 1 do -- for loop
04:    local remainder = number%possible_factor    -- assignment (%
05:    if remainder == 0 then                     -- if equal test
06:      local factor, factor_pair
07:          = possible_factor, number/possible_factor -- multiple assignment
08:      table.insert(factors, factor)             -- table insert
09:      if factor ~= factor_pair then             -- if not equal
10:        table.insert(factors, factor_pair)
11:      end
12:    end
13:  end
14:  table.sort(factors)
15:  return factors                               -- return value
16:end
17:-- The Meaning of the Universe is 42.
18:-- Let's find all of the factors driving the Universe.
19:the_universe = 42                               -- global variable and assignment
20:factors_of_the_universe = get_all_factors(the_universe) -- function call
21:-- Print out each factor
22:print("Count",  "The Factors of Life, the Universe, and Everything")
23:table.foreach(factors_of_the_universe, print)
```

This is a brief overview of the the typical Lua language constructs that you will encounter.

Line  1 – Global function declaration

Line  2 – local variable, initialized to a Table

Line  3 – For loop

Line  5 – If test

Line  8 – table operation: insert

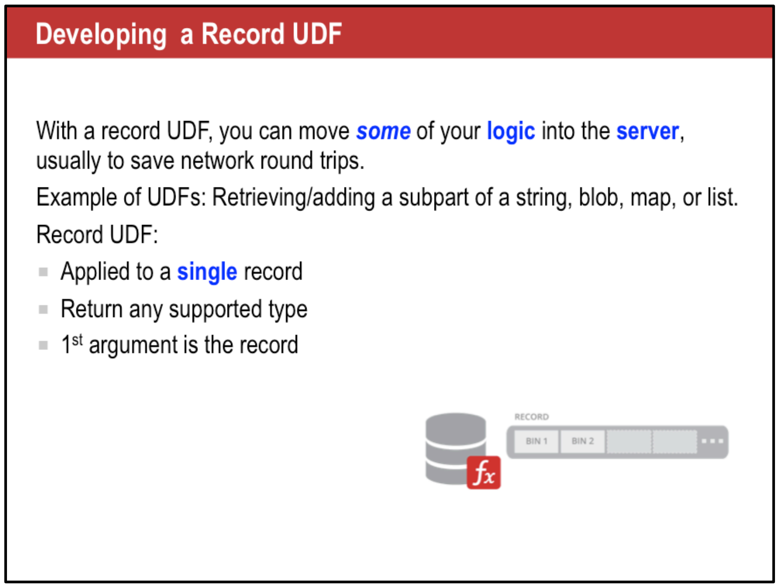
Line 14 – table operation: sort

Line 15 – function return – Lua can return multiple values, UDFs are limited to 1

Line 17 – single line comment

Line 19 – global variable declaration and initialization

Line 20 – function call



## Developing a Record UDF

With a record UDF, you can move ***some*** of your **logic** into the **server**, usually to save network round trips.

Example of UDFs: Retrieving/adding a subpart of a string, blob, map, or list.

Record UDF:

- Applied to a **single** record
- Return any supported type
- 1st argument is the record

**Developing Record UDFs**

With a Record UDF you can move some of your logic into the server, usually to save Network. Don't try to save client CPU, because then you'll just be spending on expensive server CPU.

An example of UDFs would be retrieving/adding a subpart of a string, blob, map, or list. In advertising, and example would be to get the most recent list of search terms. The UDF "get most recent search terms()", could contain more complicated logic like: return last 5, up to 12 hours, and clean up old list elements while you're there.

Record UDFs are applied to a **single record**, UDFs that access "other record" are not 'yet' supported.

A UDF can augment both read and write behavior by:

- Create/delete bins in the record.
- Read any/all bins of the record.
- Modify any/all bins of the record.
- Delete/Create the specified record.
- Access parameters passed in from the client.

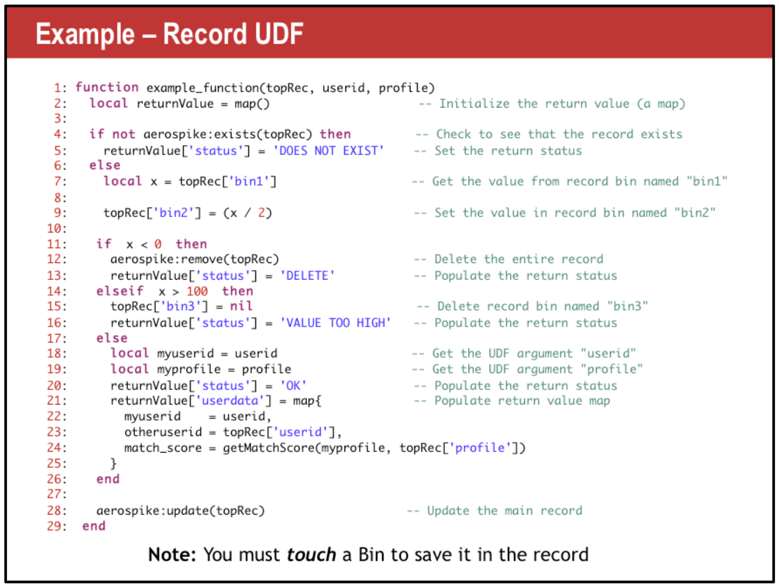
The first argument will always be a record.

A good practice is to use a variable name other than 'record' to reference the parameter passed into the UDF to represent the Aerospike Record. "record" is a Record module keyword to call built-in record functions e.g. record.ttl() .

A Record UDF can have more than one argument. Each additional argument must be of one of the types supported by the database: integer, string, list and map.

If a UDF has N parameters, and only (N-k) parms are passed in, then the last k parms will have a value of nil.

A Record UDF can return a single value of any of the types supported by the database: integer, string, bytes, list and map.



## Example – Record UDF

```
 1: function example_function(topRec, userid, profile)
 2:   local returnValue = map()                         -- Initialize the return value (a map)
 3:
 4:   if not aerospike:exists(topRec) then              -- Check to see that the record exists
 5:     returnValue['status'] = 'DOES NOT EXIST'        -- Set the return status
 6:   else
 7:     local x = topRec['bin1']                        -- Get the value from record bin named "bin1"
 8:
 9:     topRec['bin2'] = (x / 2)                        -- Set the value in record bin named "bin2"
10:
11:      if  x < 0  then
12:        aerospike:remove(topRec)                     -- Delete the entire record
13:        returnValue['status'] = 'DELETE'             -- Populate the return status
14:      elseif  x > 100  then
15:        topRec['bin3'] = nil                          -- Delete record bin named "bin3"
16:        returnValue['status'] = 'VALUE TOO HIGH'     -- Populate the return status
17:      else
18:        local myuserid = userid                      -- Get the UDF argument "userid"
19:        local myprofile = profile                    -- Get the UDF argument "profile"
20:        returnValue['status'] = 'OK'                 -- Populate the return status
21:        returnValue['userdata'] = map{               -- Populate return value map
22:          myuserid    = userid,
23:          otheruserid = topRec['userid'],
24:          match_score = getMatchScore(myprofile, topRec['profile'])
25:        }
26:      end
27:
28:      aerospike:update(topRec)                      -- Update the main record
29:   end
```

**Note:** You must ***touch*** a Bin to save it in the record

Line 1 – The function declaration. The first parameter (rec) of a record UDF is always the record. Subsequent parameters are user defined and are whatever is sent from the client. If the client passes too many args, they are ignored. Too few, and the missing parameters values are 'nil'. If the record exists, a read is done, and the rec object is populated with the current data - before the function is even called.

Line 2 – Return Value. It's convenient to use an Aerospike map as a return value, but **more efficient** to use strings or numbers, if appropriate.

Line 4 – Test the existence of a record. It is best practice to do this test, there might not be a record existing already. You can also create a Record, if it doesn't exist, using **aerospike:create()**.

Line 7 – Get the value from a Bin in a Record. The Record object is accessed, and the data brought into a local variable. The different bins are only converted from internal C representation to Lua when accessed. The highest performance is achieved by not using the record object more than once, but using temporary variables. You can get a value of a bin from a record using either of two different styles of access. You can use index-based access:

local x = rec['bin1']

Or you can also use object-based access:

local x = rec.bin1

Line 9 – Setting the variable does not result in the write, and it does encode the value into Aerospike's format.

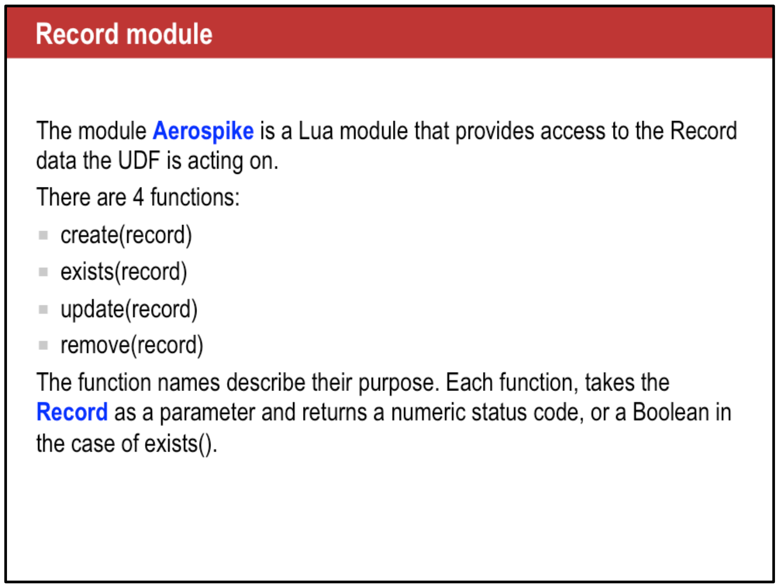
Line 12 – Remove a Record. The **aerospike:remove()** function deletes the record. All **aerospike:xxxx()** functions are consistent. After you call **aerospike:remove()** no bins are accessible

Line 15 – Remove a Bin. You can remove a Bin from a record by setting the value of the Bin to nil.

Lines 21-25 – Creating a Map. This creates a new map and initializes it to the (key, value) pairs specified, you must explicitly create a complex type. Only 1 value can be returned to the client, including simple strings and integers. This example shows using a more complex return type: a map. Lua tables (a combination of list and map type) are not supported in client languages, so you must explicitly create a map() object This will be converted to an equivalent map/dictionary type when returned the client. Returning a String on Integer is more efficient

Line 28 – Update a Record. After modifying the record, to persist the changes, you must call the aerospike.update() function to write to the database. If there is an error in writing, this function will return an error code. If you do not call this function, the record's changes will not be permanent.

Line 31 – Return. We return the map, which gets populated through out the function. A simple mechanism (trick) to return a non-integer status from a UDF is to simply return it as a status bin. This returns an entire string to the client, which can be useful for easily returning different status in a programmer-friendly fashion.


## Record module

The module **Aerospike** is a Lua module that provides access to the Record data the UDF is acting on.

There are 4 functions:

- create(record)
- exists(record)
- update(record)
- remove(record)

The function names describe their purpose. Each function, takes the **Record** as a parameter and returns a numeric status code, or a Boolean in the case of exists().

The Aerospike module provides the following functions:

**aerospike:create()**

Create a new record in the database. If create() is successful, then return 0 (zero) otherwise it is an error.

**aerospike:update()**

Update an existing record in the database. If update() is successful, then return 0 (zero) otherwise it is an error.

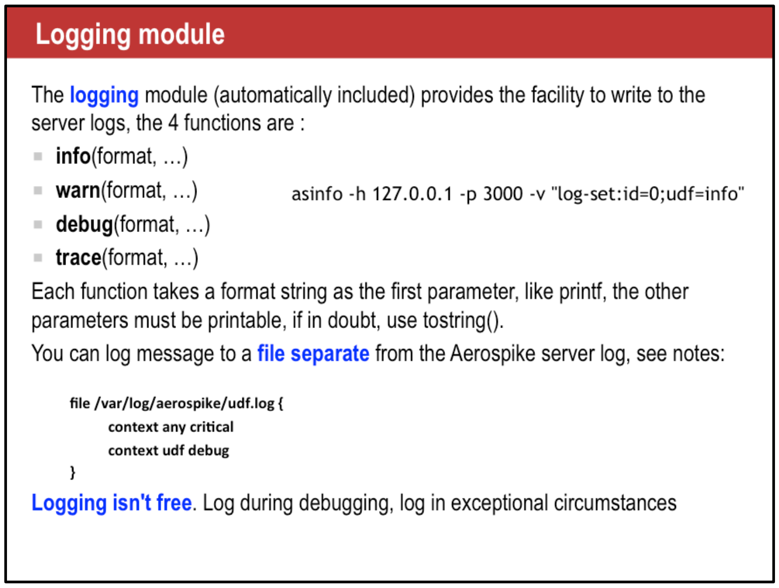
**aerospike:exists()**

Checks for the existence of a record in the database. If the record exists, then true is returned, otherwise false.

**aerospike:remove()**

Remove an existing record from the database. If delete() is success, then return 0 (zero), otherwise it is an error.

**Note:** Lua also offers a special syntax for object-oriented calls, the colon operator. An expression like o:foo(x) is just another way to write o.foo(o, x), that is, to call o.foo adding o as a first extra argument.



## Logging module

The **logging** module (automatically included) provides the facility to write to the server logs, the 4 functions are :

- **info**(format, …)
- **warn**(format, …)
- **debug**(format, …)
- **trace**(format, …)

```
asinfo -h 127.0.0.1 -p 3000 -v "log-set:id=0;udf=info"
```

Each function takes a format string as the first parameter, like printf, the other parameters must be printable, if in doubt, use tostring().

You can log message to a **file separate** from the Aerospike server log, see notes:

```
file /var/log/aerospike/udf.log {
        context any critical
        context udf debug
}
```

**Logging isn't free**. Log during debugging, log in exceptional circumstances

**Logging functions**

Logging functions  send log message to the Aerospike Server's logs. The logging functions all accept a format string as the first parameter with a variable argument list, much like printf in C.

While logging, you will need to ensure the value used as arguments for the format string are valid for the format string parameters. When in doubt, you can use the Lua tostring() function to convert a value to a string, whenever possible. For example:

```
info("Hello %s", tostring(name))
```

There are 4 levels of logging, and the function names reflect this

- info() - Log a message as INFO in the Aerospike Server.
- warn() - Log a message as a WARNING in the Aerospike Server.
- debug() - Log a message as DEBUG in the Aerospike Server.
- trace() - Log a message as DETAIL in the Aerospike Server.

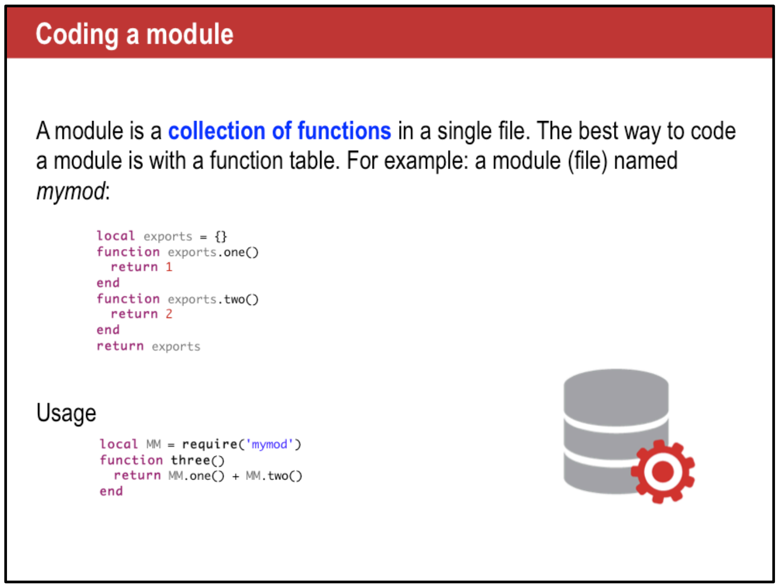
**Logging UDF message to a separate file**

To make it easier to debug UDFs, you can direct the output of the UDF messages to a file separate from the main Aerospike log. To do this, modify the “logging” stanza in the [something] section of  the server configuration file (aerospike.conf). Here is an example:

```
logging {
        # Log file must be an absolute path.
        file /var/log/aerospike/aerospike.log {
                context any info
        }
        file /var/log/aerospike/udf.log {
                context udf debug
        }
}
```

**Logging isn't free.** Log during debugging, log in exceptional circumstances. TO turn it on and off use:

```
asinfo -h 127.0.0.1 -p 3000 -v "log-set:id=0;udf=info"
```



## Coding a module

A module is a **collection of functions** in a single file. The best way to code a module is with a function table. For example: a module (file) named *mymod*:

```lua
local exports = {}
function exports.one()
  return 1
end
function exports.two()
  return 2
end
return exports
```

Usage

```lua
local MM = require('mymod')
function three()
  return MM.one() + MM.two()
end
```

**Lua Module**

A Lua Module is a collection of variables and functions contained in a single file, which can be imported (used) by other Lua Modules. There are various options for how to define a Lua Module, however we found the best method is to use what is called the "local module". A local module is a file that defines all variables and functions as locals, and exports the functions that can be used by other modules by returning a Lua Table.

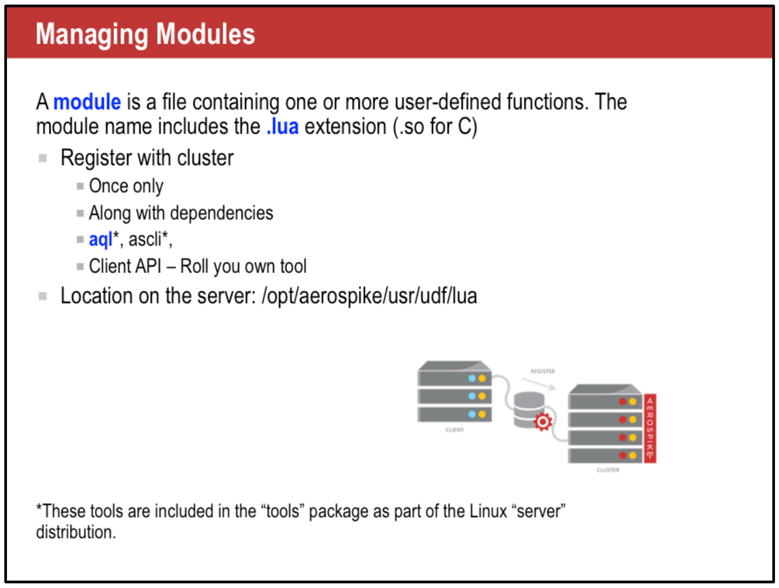
**Exports Object in a function table**

Define a local table that will collect the exported members, this is a function table. In the following example, we define an local table called "exports", which will be populated with the members to be exported.

```lua
local exports = {}
function exports.one()
  return 1
end
function exports.two()
  return 2
end
return exports
```

Another module can then require "mymod.lua" and use the functions of the module:

```lua
local MM = require('mymod')
function three()
  return MM.one() + MM.two()
end
```



## Managing Modules

A **module** is a file containing one or more user-defined functions. The module name includes the **.lua** extension (.so for C)

- Register with cluster
  - Once only
  - Along with dependencies
  - **aql***, ascli*,
  - Client API – Roll you own tool
- Location on the server: /opt/aerospike/usr/udf/lua

REGISTER
CLIENT
AEROSPIKE
CLUSTER

*These tools are included in the "tools" package as part of the Linux "server" distribution.

**Managing modules**

A modules is a file containing one or more user-defined functions. A module and all it's non-Aerospike dependencies must be uploaded and registered with the cluster, before the UDF can be invoked. A deployment may have one or many modules.

When a UDF module is registered, it is maintained in the cluster. As nodes are added to the cluster, the UDF modules are synchronized.

**Registration/Deregistration Options**

UDF registration is an administrator operation, and be controlled using normal change control procedures. A UDF module can be registered into a cluster using one of the provided tools:

- **aql*** – A command-line utility with SQL-like commands
- **ascli*** – Aerospike command line interface
- Client API - provides a number of functions that allow you to programmatically manage user-defined functions in a cluster, this is useful of you want to develop your own tool. **Do NOT do this in your production code.**

*These tools are are included the "tools" package as part of the Linux "server" distribution.

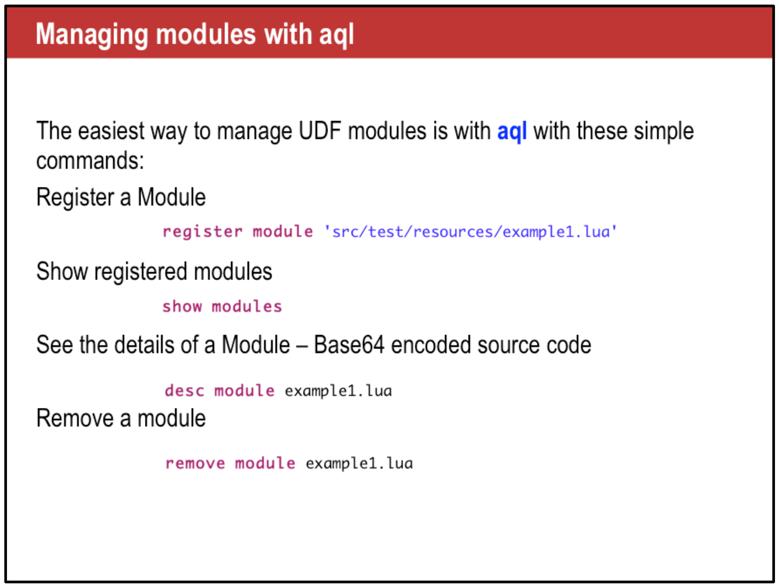
**Lifecycle of a UDF**

When a package is registered, it is immediately compiled into byte-code and made available to subsequent invocations from clients. If the registration is replacing an existing package, currently executing UDFs will use the existing package, all new invocations after the update will only use the most recently updated module. When a modules is removed, it's UDFs are no longer be available.

If a client is in the middle of invoking a UDF when it's module is removed or updated, it will be able to complete the operation without interruption. Once the function completes, any subsequent invocation will either use the updated function, or fail if the module is removed.

**Module Dependencies**

A module's  non-Aerospike dependencies must be manually copied to each node in the cluster, and place in this directory:  /opt/aerospike/usr/udf/lua



## Managing modules with aql

The easiest way to manage UDF modules is with **aql** with these simple commands:

Register a Module

```
register module 'src/test/resources/example1.lua'
```

Show registered modules

```
show modules
```

See the details of a Module – Base64 encoded source code

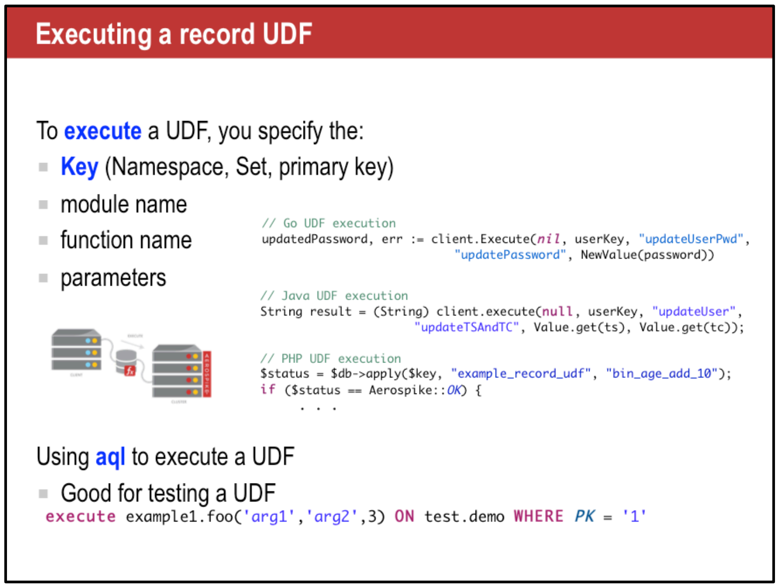
```
desc module example1.lua
```

Remove a module

```
remove module example1.lua
```

Tips:

When you register a module, specify the fully qualified path name the Lua file, including the .lua extension



## Executing a record UDF

To **execute** a UDF, you specify the:

- **Key** (Namespace, Set, primary key)
- module name
- function name
- parameters

```
// Go UDF execution
updatedPassword, err := client.Execute(nil, userKey, "updateUserPwd",
                           "updatePassword", NewValue(password))

// Java UDF execution
String result = (String) client.execute(null, userKey, "updateUser",
                         "updateTSAndTC", Value.get(ts), Value.get(tc));

// PHP UDF execution
$status = $db->apply($key, "example_record_udf", "bin_age_add_10");
if ($status == Aerospike::OK) {
      . . .
```

Using **aql** to execute a UDF

- Good for testing a UDF

```
execute example1.foo('arg1','arg2',3) ON test.demo WHERE PK = '1'
```

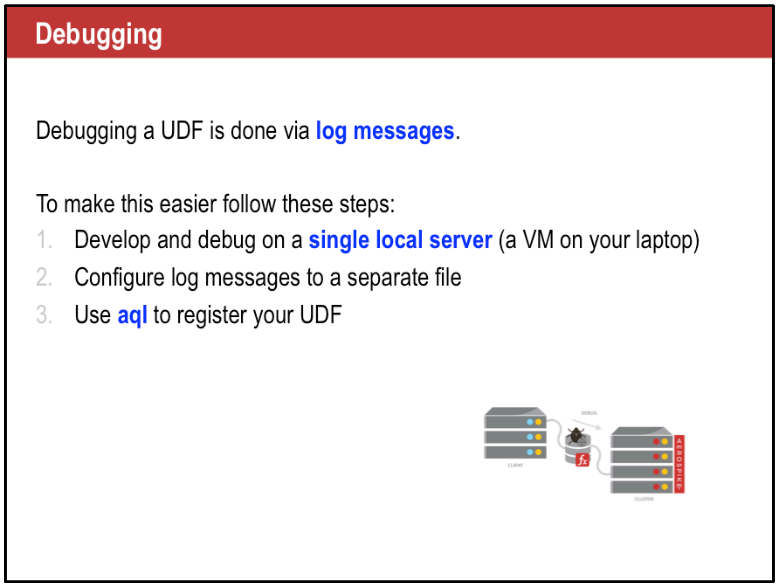
**Execute a record UDF**

To execute a UDF, you specify the:

- Key object containing the Namespace, Set and value of the primary key
- Module name
- Function name within the module
- Arguments (parameters) that will be passed to the function

In your code you will use the Client API to execute UDFs, but for developer and unit testing you can optionally use **aql** to execute a UDF.

```
execute example1.foo('arg1','arg2',3) ON test.demo WHERE PK = '1'
```



## Debugging

Debugging a UDF is done via **log messages**.

To make this easier follow these steps:

1. Develop and debug on a **single local server** (a VM on your laptop)
2. Configure log messages to a separate file
3. Use **aql** to register your UDF

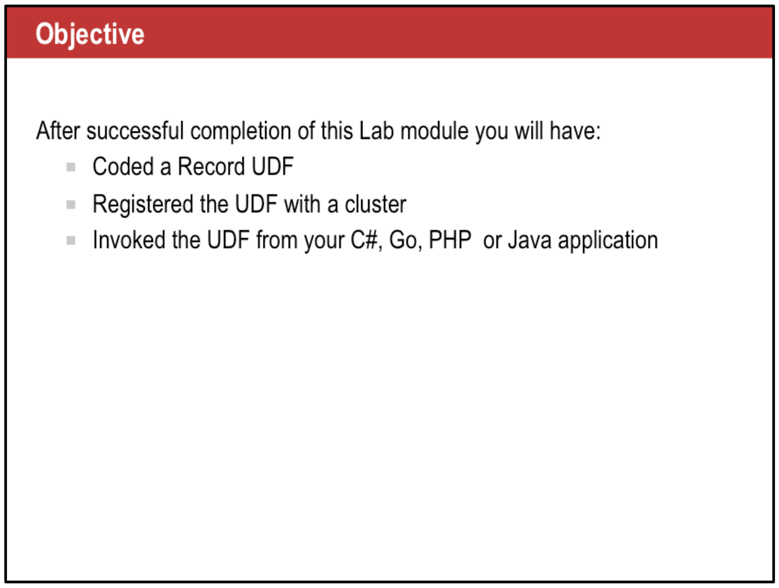
**Debugging**

There is no facility to connect a debugger to the Lua contexts running inside the Aerospike daemon. Debugging is done vial log messages.

**Tips to make debugging easier**

1. **Develop and debug on a single node cluster**, using a VM on your laptop, even if you're using OSX or Windows. In a multi-node cluster, the server that will hold your record is randomly allocated, so where a request is executed can't be predicted. Using a single-node cluster ensures that you will be able to **tail** and **grep** the correct log file – because there is only one log.
2. Configure your UDF log messages to be written to a file separate from the main Aerospike log. It will be easier to **tail** and **grep** this log.
3. Use **aql** to register you UDF. You will probably register and test you UDF many times before you can use it from your application. **aql** provides you a way to load data, register modules, execute UDFs and cleanup data, in a repeatable way.
4. Use a flag to turn on debugging. This is a subtle language feature of Lua that can greatly improve the performance of your UDF without sacrificing your debugging logging.

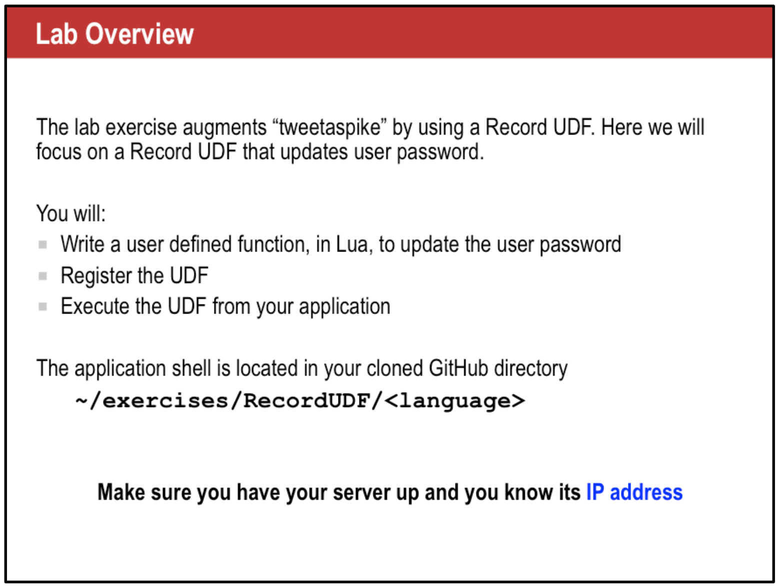
## Lab: User Defined Functions - record



## 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 C#, Go, PHP  or Java 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

```
~/exercises/RecordUDF/<language>
```

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

On your USB stick you will find the following directories:

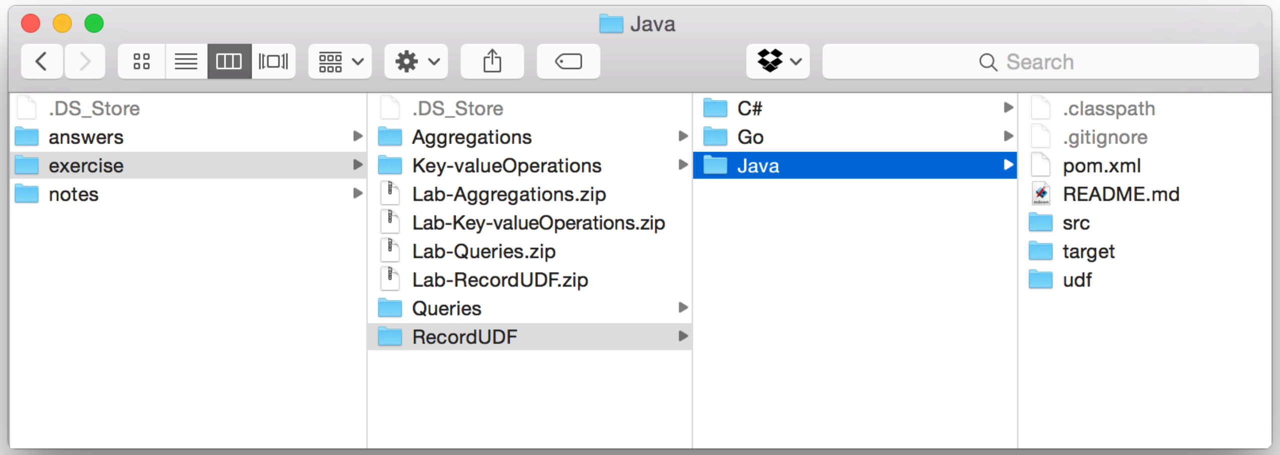
- Answers
- Exercise
- Notes

In the exercise directory, select the subdirectory for your programming language:

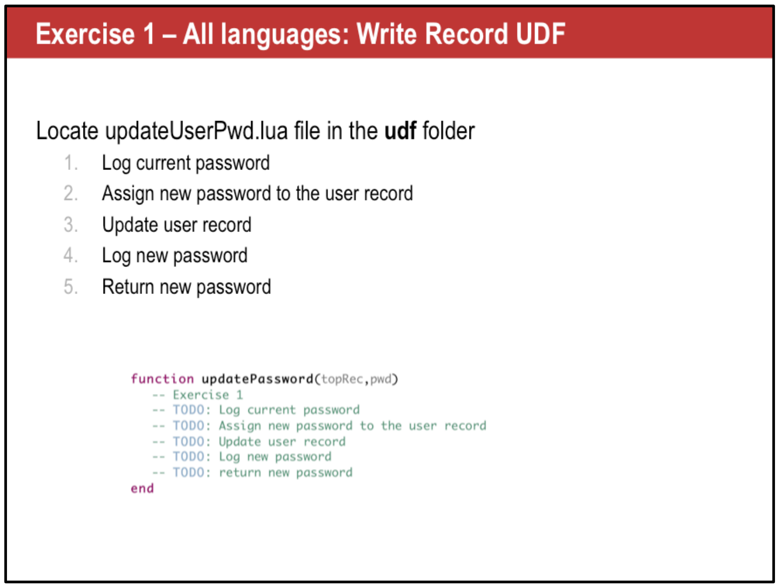
- C#
- Java

The exercises for this module are in the UDF 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**



| | | | |
|---|---|---|---|
| .DS_Store | .DS_Store | C# | .classpath |
| answers | Aggregations | Go | .gitignore |
| exercise | Key-valueOperations | Java | pom.xml |
| notes | Lab-Aggregations.zip | | README.md |
| | Lab-Key-valueOperations.zip | | src |
| | Lab-Queries.zip | | target |
| | Lab-RecordUDF.zip | | udf |
| | Queries | | |
| | RecordUDF | | |



## Exercise 1 – All languages: Write Record UDF

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

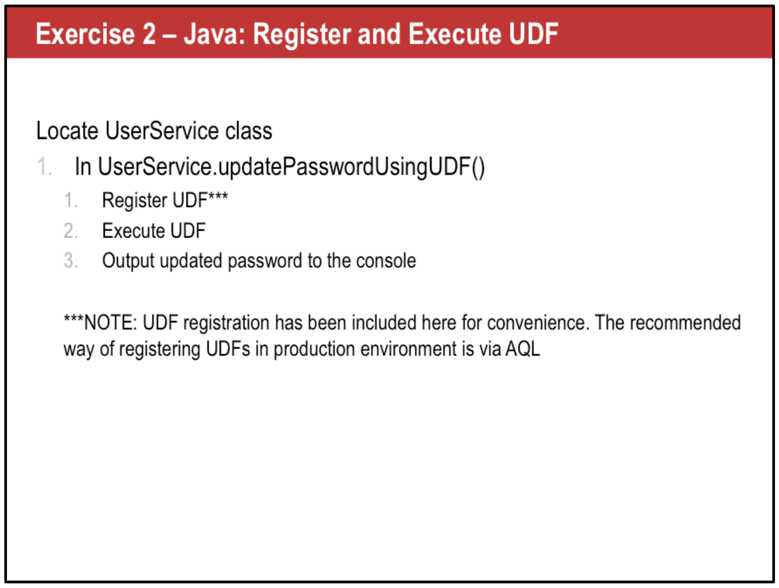
1. Log current password
2. Assign new password to the user record
3. Update user record
4. Log new password
5. Return new password

```
function updatePassword(topRec,pwd)
   -- Exercise 1
   -- 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 2 – Java: Register and Execute UDF

Locate UserService class

1. In UserService.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 1.

We will programmatically register the UDF for convenience.

In UserService.updatePasswordUsing UDF(), locate these comments and add your code:

1. Register the UDF with an API call

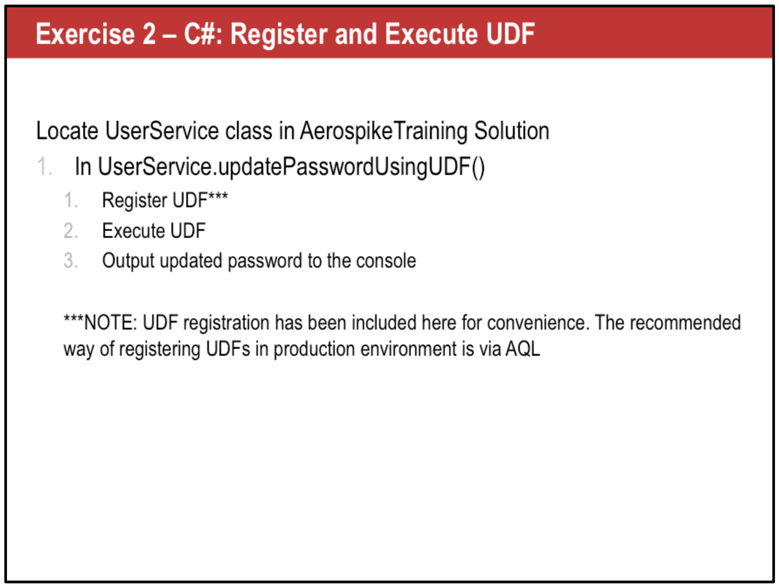
```
LuaConfig.SourceDirectory = "udf";
File udfFile = new File("udf/updateUserPwd.lua");
RegisterTask rt = client.register(null, udfFile.getPath(),
                        udfFile.getName(), Language.LUA);
rt.waitTillComplete(100);
```

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

```
String updatedPassword = (String) client.execute(null, userKey,
            "updateUserPwd", "updatePassword",
                  Value.get(password));
```

3. Output the return from the UDF to the console

```
console.printf("\nINFO: The password has been set to: " + updatedPassword);
```



## Exercise 2 – C#: Register and Execute UDF

Locate UserService class in AerospikeTraining Solution

1. In UserService.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 1.

We will programmatically register the UDF for convenience.

In UserService.updatePasswordUsing UDF(), locate these comments and add your code:

1. Register the UDF with an API call

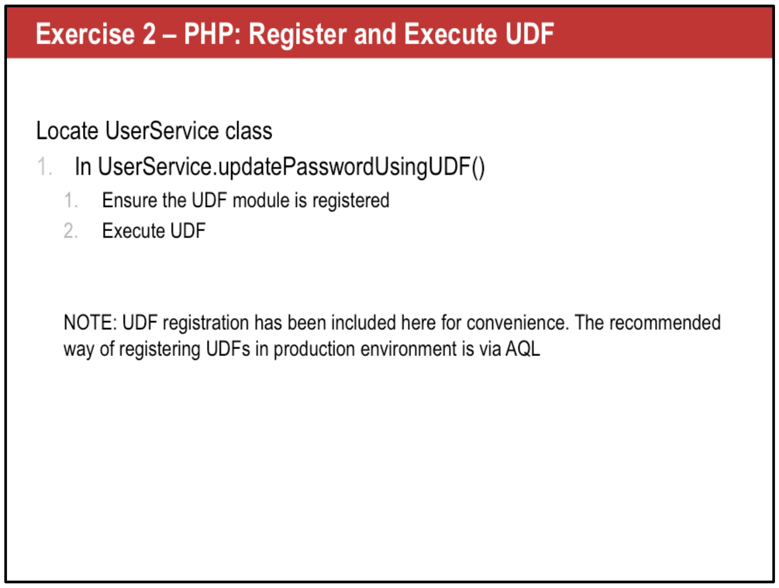
```
string luaDirectory = @"..\..\udf";
LuaConfig.PackagePath = luaDirectory + @"\?.lua";
string filename = "updateUserPwd.lua";
string path = Path.Combine(luaDirectory, filename);
RegisterTask rt = client.Register(null, path, filename, Language.LUA);
rt.Wait();
```

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

```
string updatedPassword = client.Execute(null, userKey, "updateUserPwd", "updatePassword",
Value.Get(password)).ToString();
```

3. Output the return from the UDF to the console

```
Console.WriteLine("\nINFO: The password has been set to: " + updatedPassword);
```



## Exercise 2 – PHP: Register and Execute UDF

Locate UserService class

1. In UserService.updatePasswordUsingUDF()
   1. Ensure the UDF module is registered
   2. Execute UDF

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 1.

We will programmatically register the UDF for convenience.

In UserService.updatePasswordUsing UDF(), locate these comments and add your code:

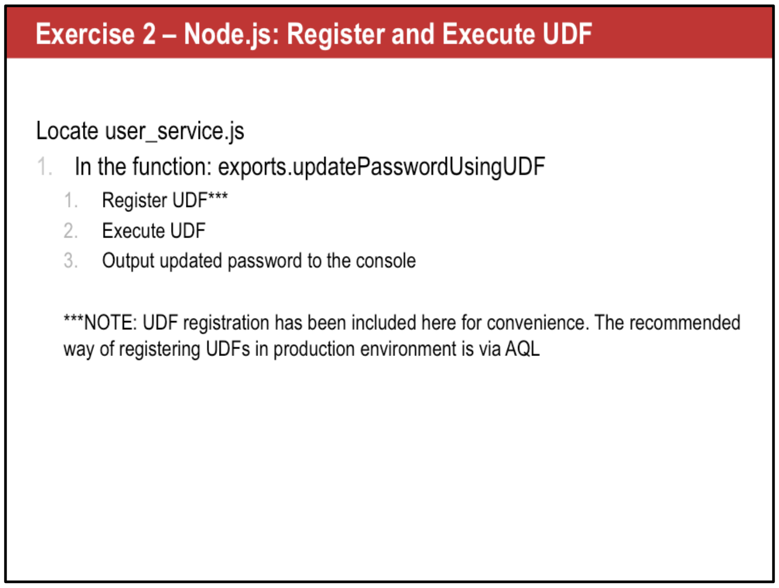
1. Ensure the UDF module is registered

```
$ok = $this->ensureUdfModule('udf/updateUserPwd.lua',
'updateUserPwd.lua');
if ($ok) echo success();
else echo fail();
$this->display_module('udf/updateUserPwd.lua');

echo colorize("Updating the user record >", 'black', true);
$key = $this->getUserKey($username);
if ($this->annotate) display_code(__FILE__, __LINE__, 7);
$args = array($new_password);
```

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

```
$status = $this->client->apply($key, 'updateUserPwd',
'updatePassword', $args);
if ($status !== Aerospike::OK) {
    // throwing an \Aerospike\Training\Exception
    echo fail();
    throw new Exception($this->client, "Failed to update
password for user $username");
}
```



## Exercise 2 – 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 1.

We will programmatically register the UDF for convenience.

In the function module.updatePasswordUsingUDF add your code:

1. Register the UDF with an API call

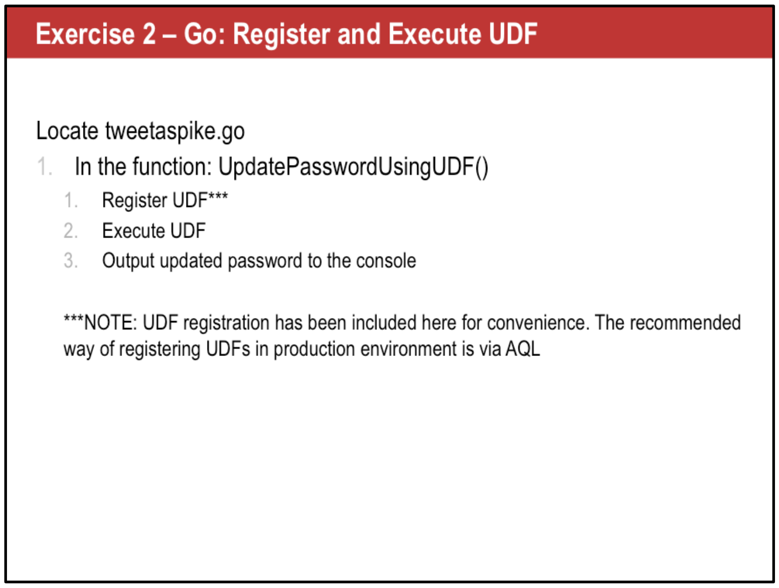
```
// Register UDF
client.udfRegister('udfs/updateUserPwd.lua', function(err) {
  if ( err.code === 0 ) {
    var UDF = {module:'updateUserPwd', funcname:
      'updatePassword', args: [answers.password]};
    var key = {
      ns:  "test",
      set: "users",
      key: answers.username
    };
```

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

```
    client.execute(key, UDF, function(err) {
      // Check for errors
      if ( err.code === 0 ) {
           console.log("INFO: User password updated successfully!");
      }
      else {
           console.log("ERROR: User password update failed\n", err);
      }
    });
```

3. Output the return from the UDF to the console

```
  } else {
    // An error occurred
      console.error("ERROR: updateUserPwd UDF registeration failed:\n", err);
  }
});
```



## Exercise 2 – Go: Register and Execute UDF

Locate tweetaspike.go

1. In the function: 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 1.

We will programmatically register the UDF for convenience.

In UpdatePasswordUsingUDF(), locate these comments and add your code:

1. Register the UDF with an API call

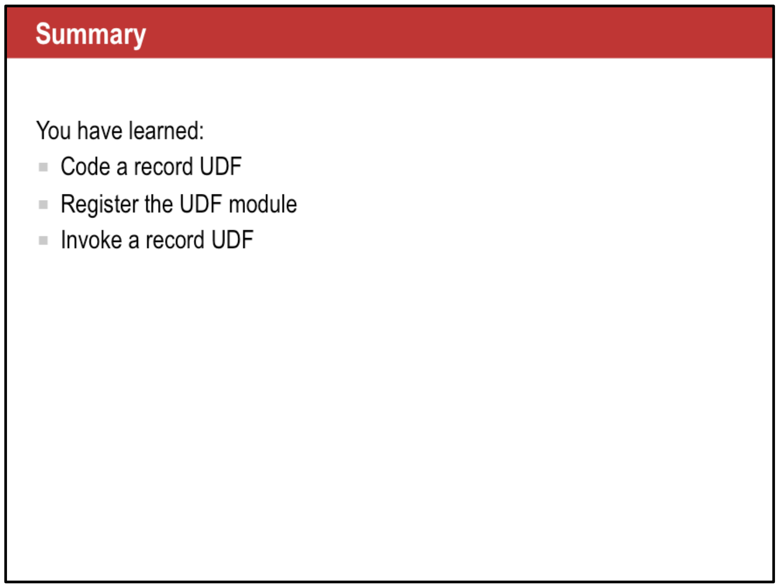
```
regTask, err := client.RegisterUDFFromFile(nil, "udf/updateUserPwd.lua",
                     "updateUserPwd.lua", LUA)
panicOnError(err)
// wait until UDF is created
for {
     if err := <-regTask.OnComplete(); err == nil {
          break
     }
}
```

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

```
updatedPassword, err := client.Execute(nil, userKey, "updateUserPwd",
                                       "updatePassword", NewValue(password))
panicOnError(err)
```

3. Output the return from the UDF to the console

```
fmt.Printf("\nINFO: The password has been set to: %s\n", updatedPassword)
```



## Summary

You have learned:

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

AEROSPIKE