Contents

1 Introduction

Hello and welcome to my tutorial for writing your own scripts for *FiveM*. In this tutorial we will be starting from setting up a local Artifacts server for *FiveM*, which will be then later used to write a *base resource script* to handle to give you basic database support to handle the reading and writing of custom user-data; as well as to give you an idea on how to use the *NUI*, which is just a browser overlay over the running *GTAV* game and use the native functions provided to modify the game to your liking. I will assume that you are running a *Microsoft Windows* system.
The development process for the resource will be shown twice, once using Lua as the base scripting language and once using C# as the base scripting language, hence the two chapters in the table of contents.
I expect, that you have already downloaded and installed the *FiveM-Client*[Col18a], if not, do that immediately by downloading from `https://fivem.net`.
Furthermore you need an editor to edit the scripts depending on the main language you want to use. If you are not experienced with any editors that work with the programming languages, I recommend you using *Notepad++* which can be downloaded from `https://notepad-plus-plus.org` or *Atom* which you can download from `https://atom.io`. Furthermore if you do not want to write your resources using Lua, but instead C# you will need to download Visual Studio Community from `https://www.visualstudio.com/downloads`.
While writing scripts there are several sources you should use as reference material.

Google `https://www.google.com`, this is self-explanatory.

Native Reference `https://runtime.fivem.net/doc/reference.html`, this site contains all native functions with ample explanations making sure you use them correctly, if they work.

FiveM docs http://docs.fivem.net, this is probably the best resource you can use, as its content builds the basis of modifing the game.

FiveM Wiki https://wiki.fivem.net, this Wiki is not really completely up-to-date, but it is good enough to be used.

GT-MP Wiki https://wiki.gt-mp.net, this Wiki from a competitive modifying solution, contains an awesome amount of information about game content.

2 Setting up a Development Server

This part of the tutorial just follows the FiveM wiki page about setting up a server[Con17] with a little bit more detail.
The first step before be able to run the server is to make sure to have an up-to-date redistributable for *Microsoft Visual C++* installed. The best way to make sure you have, is just by installing it, possibly again. You can get the executeable from the url https://go.microsoft.com/fwlink/?LinkId=746572.
Now we need the actual server files, which come in two part. The first contains the executable files for the server, which can be downloaded from https://runtime.fivem.net/artifacts/fivem/build_server_windows/master. Please check back at least once a month if there is a more up-to-date version of the server files available and update yours. From there we download the latest version of the Server, which is indicated by the date in the date column and the number starting the directory, which contains the `server.zip` file. We download the `server.zip` file from the most recent directory.
Once we got this file we unpack it to a directory which will contain the server files, e.g. in my case, I unpacked the archive to `D:\myfxserver`. In that directory, if you have successfully unpacked it you should have a bunch of `.dll` files, `components.json`, `FXServer.exe`, `run.cmd`, and a folder `citizen`.
For the next part we need a few basic scripts to run the server, which can be deactivated later if you are sure you do not need them. To download them, you visit https://github.com/citizenfx/cfx-server-data and click on the green *Clone or download* button, select then *Download ZIP*. Unpack your downloaded zip file also to the same directory as your unpacked the `server.zip`, e.g. in my case `D:\myfxserver`. In that folder you should now find another folder, called `cfx-server-data-master` that we will rename to something simpler of your choice; in my case, I will go with `data`.
The next step instead of finalizing the setup of the files first, is to get a key to actually run the server. This key can be obtained from https://keymaster.fivem.net. To use that site you need to be registered on the FiveM Forums. Once you are logged in on that side, you hit the blue *register* link, and fill in the forms. I filled in for the Label *Tutorial Server*, as IP Address I put in *127.0.0.1*, as I will only use it for tutorials or this tutorial. And of course, the server type is *Home hosted*. Now you should have a key displayed for the server in a table, which are some random letters and numbers.
Finally we need to setup the server configuration file and a shortcut to make it easy to start. For that, we open Notepad++ and start a new file. The contents will be just copied & pasted

from https://wiki.fivem.net/wiki/Running_FXServer#server.cfg or from the text below.

```
# you probably don't want to change these!
# only change them if you're using a server with multiple network interfaces
endpoint_add_tcp "0.0.0.0:30120"
endpoint_add_udp "0.0.0.0:30120"

start mapmanager
start chat
start spawnmanager
start sessionmanager
start fivem
start hardcap
start rconlog
start scoreboard
start playernames

sv_scriptHookAllowed 1

# change this
#rcon_password yay

sv_hostname "Tutorial Server"

# nested configs!
#exec server_internal.cfg

# loading a server icon (96x96 PNG file)
#load_server_icon myLogo.png

# convars for use from script
set temp_convar "hey world!"

# disable announcing? clear out the master by uncommenting this
#sv_master1 ""

# want to only allow players authenticated with a third-party provider like
    Steam?
#sv_authMaxVariance 1
#sv_authMinTrust 5

# add system admins
add_ace group.admin command allow # allow all commands
add_ace group.admin command.quit deny # but don't allow quit
add_principal identifier.steam:110000112345678 group.admin # add the admin to
     the group

# remove the # to hide player endpoints in external log output
#sv_endpointprivacy true

# server slots limit (must be between 1 and 31)
sv_maxclients 30

```

```
# license key for server (https://keymaster.fivem.net)
sv_licenseKey changeme
```

Here you still need to change the changeme of the sv_licenseKey changeme to the license key you obtained before. If you have done that save the file in your server data folder with any name you like, e.g. in my case I saved it in D:\myfxserver\data\ as the suggested server.cfg.
All that is missing now is a shortcut to start the server easily. Navigate to do this to the D:\myfxserver folder and right click the file run.cmd and select Create shortcut, move that to somewhere from where you want to start your server, e.g. in my case the Desktop. Then rename it to something you want. Now right click the shortcut and select properties. Currently the line *Start in* should say something like D:\myfxserver, we just change it to D:\myfxserver\data. Now all is left before we can start the server is to change the *Target* to D:\myfxserver\run.cmd +exec server.cfg, if you named your file server.cfg, the same as me.

It is now time to start up your server and join the game. You can start the server by double clicking your shortcut. Start up FiveM, enable *Dev mode* under *Settings*. Now click *Localhost* which you should have in the top menu to join your Vanilla FiveM Server.
Soon after it will be time for you to write you first own resource/script.

3 Your own Base Script with Lua, HTML, and Javascript

The first script you should always add to a server is the one that builds the foundation of your server, so it has to be a script that can load and save data, as well as recognize users, so it allows to recall data based on the user it has stored the data for.
For the database we will be using *CouchDB*, which you probably have to Download from http://couchdb.apache.org and install. After you have installed CouchDB you should visit http://127.0.0.1:5984/_utils and create an admin account and log in; this is your account to do administrative work on the *CouchDB*. Then create a database via a button on the top right, in my case, I called the database tutorial. Then click on the database.
Create another admin account via http://127.0.0.1:5984/_utils which we will use to for the database interaction. I created another admin called *fxserver* with the password *tutorial*.

3.1 __resource.lua

The __resource.lua is very likely the file you usually only write when you start testing your scripts, but since I know where we will be going, we can start and write this important file first. You can orient yourself on the example resource manifest from the *FiveM* wiki found at https://wiki.fivem.net/wiki/Resource_manifest.
First of we will be creating a new folder in our data\resources folder, which will be the name of the resource. Before you go wild, you will need to remember, that the name has to be in all lower case letters, otherwise certain interactions we are going to use will not work.

In my case, I named the folder `mybase`.
We also need to add to the `server.cfg` the line `start mybase` at the appropriate position.
So creating all of our files and folders, I got:

- `D:\myfxserver\data\resources\mybase`
- `__resource.lua`, the file defining what file is which for the resource handler.
- `couchdb.lua`, the script file containing all database interaction stuff.
- `auth.lua`, the script that authenticates players on the server
- `server.lua`, the script that handles all the server interaction with the clients
- `client.lua`, the script that handles all the client interaction with the server and a short interaction with natives.
- `cfg.lua`, the lua file that just holds all the configuration information for our base script. Here it is connection information needed for CouchDB and whether the server is using whitelist mode.
- `scoreboard.html`, the html file which is overlayed over the entire game screen.

All of these files are empty, and we start writing the `__resource.lua`. First of all we need to pick a resource manifest version. which we will put into our file first with

```
resource_manifest_version '<resource manifest version>'
```

The resource manifests versions which are sensible to choose from are [Col18b]:

44febabe-d386-4d18-afbe-5e627f4af937 This one represents active to the latest insight into the natives/functions from GTAV currently available for *FiveM*.

05cfa83c-a124-4cfa-a768-c24a5811d8f9 This version lets you register network entities, also it registers this script as a network script.

We will use here the **44febabe-d386-4d18-afbe-5e627f4af937** resource manifest version, as it only is a script with a bit of database and interface interaction.
Next we are going to define an `ui_page`, in the same way we added a `resource_manifest_version`.
The `ui_page` of course has to be set to `scoreboard.html`.
To ensure that the `ui_page` is send to the users we need to reference it again with the identifier of `file` or `files{}` and add it there into the curly brackets.
Now all that is left is to define the scripts and which side of the game has to execute those: Client or server. As we have only one script for the client, we define it with `client_script`, the same we did with the resource manifests or the ui page, which in our case is just `client.lua`.
Next we need to define our scripts executed by the server, to do that we add `server_scripts {}` and add all our scripts, namely `couchdb.lua`, `auth.lua`, and `server.lua`, seperated by colons into the curly brackets.

We would be done here now, but we definately want that other resources can interact with our mybase resource. So we need to define exports {} and server_exports {}. As clients interact with the servers via *events*, we will not actually need any exports, but will rather specify some server events called by a client. On the other hand we will export a few function on the server side. They will be called getData, setData, doesDataExist, getUserData, and setUserData.
Before moving on let us check if you have done everything right. If you have done everything right, your __resource.lua should look something like this:

```
resource_manifest_version '44febabe-d386-4d18-afbe-5e627f4af937'

ui_page 'scoreboard.html'
file 'scoreboard.html'

client_script 'client.lua'
server_scripts {
  'couchdb.lua',
  'auth.lua',
  'server.lua',
  'cfg.lua'
}

server_exports {
  'getData',
  'setData',
  'doesDataExists',
  'getUserData',
  'setUserData'
}
```

3.2 couchdb.lua

Now it is time to write our first script file, which will handle the connection to *CouchDB*. We access *CouchDB* via simple http requests, like you do with your webbrowser when browsing on the internet.
Here we will employ chiefly just three methods of doing so: HEAD to return minimal information about a document or database, GET to get data from a document, and PUT to put data into a document.
But first we need to be able to properly dispatch http requests and get a proper result, to do that we have to write a wrapper for the PerformHttpRequest function that comes with *FiveM* lua implementation. For that look at the following address https://wiki.fivem.net/wiki/PerformHttpRequest and try to write a function that wraps the function neatly and awaits it finishing up.
To do this remember, that in lua, you can use variables inside a callback function as long as they count as defined for the function. So you can check if the function has finished by setting a boolean and waiting for it to turn to a certain state you want it to be. This is what I ended up with:

```lua
function HttpRequest(url, method, data, headers)
  local cbStatus, cbContent, cbHeaders = 0, "", {}
  if type(headers) == 'table' then
    if type(data) == 'table' then
      data = json.encode(data)
    end
    local finishedHttpRequest = false
    PerformHttpRequest(url,
      function(status,content,headers)
        cbStatus, cbContent, cbHeaders = status, content, headers
        finishedHttpRequest = true
      end,
      method, data, headers)
    repeat Citizen.Wait(0) until finishedHttpRequest == true
  end
  return cbStatus, cbContent, cbHeaders
end
```

The local variables inside the function can still be set by the callback, so that they can be returned once they are set. Now that the we can interact via http with websites or with CouchDB for that matter, we will have to have the ability to send the username and password to CouchDB, but we will not be using the crude method of sending it via `http://username:password@127.0.0.1:5984/` but by using the *Authorization* header. For that we need to be able to base64encode data. For that we pick any of the encoding methods from `http://lua-users.org/wiki/BaseSixtyFour`. You should try to understand what is happening in these methods, but if you don't it does not really matter. You should just be aware that nearly all of your problems have been solved already by someone else, and standing on their shoulders by using their code-snippets will be speeding up your workflow, but you should always double check their code.
I picked the last method for Lua 5.3 with binary operators.

```lua
local bs = { [0] =
  'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P',
  'Q','R','S','T','U','V','W','X','Y','Z','a','b','c','d','e','f',
  'g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v',
  'w','x','y','z','0','1','2','3','4','5','6','7','8','9','+','/',
}

local function base64(s)
  local byte, rep = string.byte, string.rep
  local pad = 2 - ((#s-1) % 3)
  s = (s..rep('\0', pad)):gsub("...", function(cs)
    local a, b, c = byte(cs, 1, 3)
    return bs[a>>2] .. bs[(a&3)<<4|b>>4] .. bs[(b&15)<<2|c>>6] .. bs[c&63]
  end)
  return s:sub(1, #s-pad) .. rep('=', pad)
end
```

Now that we can encode our username and password, we can continue. The first thing we should do is write three wrappers for the methods we are going to employ to communicate with CouchDB: HEAD, GET, and PUT. Inside the wrappers for the *HttpRequest* function we set the methods, headers, and content if applicable. The headers are as follows:

Accept = "application/json"

Authorization = someauthorizationvariable

Content-Type = "application/json"; this is only for PUT requests.

While GET and PUT can return the entire HttpRequest variables pack, this is definately not needed for HEAD. As HEAD requests only get very basic information delivered, and can be used mostly only to check if a document exists. Thus if the status is returned as *200* – which means everything is ok – we shall return true on a HEAD request and false otherwise. The code I ended up with looks like this:

```lua
function CouchDB.Get(url)
  if not CouchDB.init then
    repeat Citizen.Wait(0) until CouchDB.init == true
  end
  headers = {}
  headers.Accept = "application/json"
  headers.Authorization = CouchDB.auth
  return HttpRequest(url, "GET", "", headers)
end

function CouchDB.Put(url, content)
  if not CouchDB.init then
    repeat Citizen.Wait(0) until CouchDB.init == true
  end
  headers = {}
  headers.Accept = "application/json"
  headers.Authorization = CouchDB.auth
  headers["Content-Type"] = "application/json"
  return HttpRequest(url, "PUT", content, headers)
end

function CouchDB.Head(url)
  headers = {}
  headers.Authorization = CouchDB.auth
  headers.Accept = "application/json"
  local status, content, headers = HttpRequest(url, "HEAD", "", headers)
  if status == 200 then
    return true
  end
  return false
end
```

It furthermore waits until the connection to CouchDB was properly setup, before beginning to allow queries other than head. Thus we logcally speaking should write the setup of the

connection next. For that we first write a simple setup function that just sets the authorization variable by base64 encoding a string passed to it, same as the server and the databasename, which should be stored locally.
Furthermore that function should be automatically called on start of the resource and be automatically fed the information from the `cfg.lua`. To do that, in C# we would be using an EventHandler, but since they are unneccesary in Lua, we just wrap it into a thread for the server, by using `Citizen.CreateThread`. Here is what I got:

```lua
function CouchDB.Setup(server, dbname, auth)
  CouchDB.auth = "Basic "..base64(auth)
  CouchDB.address = server..dbname.."/"
  if CouchDB.Head(CouchDB.address) then
    return true
  end
  return false
end

Citizen.CreateThread(function()
  CouchDB.init = CouchDB.Setup(cfg.dbserver, cfg.dbname, cfg.auth)
  if CouchDB.init then
    print("Connection to CouchDB established.")
  else
    print("Connection to CouchDB failed.")
  end
end)
```

All that is left, is to write the functions for our 5 server exports, so external resources can also access the database in the same way this resource will. For that we will be first needing to identify a player based on his steamid and license. We ought to take the steamid if it exists, and if it does not exist, we shall use the license to identify the player internally. For you to get that working, you might play around with the function `GetPlayerIdentifiers(source)` where `source` is the serverID of a logged in player, or another temporary ID. If you played around long enough and managed to seperate the table, the result can look similar to this:

```lua
function getIdentifiers(source)
  local steamID64, ip, license = 0, "", ""
  for k,v in pairs(GetPlayerIdentifiers(source)) do
    if string.sub(v,0,6) == 'steam:' then
      steamID64 = tonumber(string.sub(v,7), 16)
    end
    if string.sub(v,0,3) == 'ip:' then
      ip = string.sub(v,4)
    end
    if string.sub(v,0,8) == 'license:' then
      license = string.sub(v,9)
    end
  end
  return steamID64, ip, license
end

function getUserIdentifier(src)
```

```lua
  local steamID64, ip, license = getIdentifiers(src)
  if steamID64 == 0 then
    return license
  end
  return steamID64
end
```

Now that we have the identification cleared, we want to access the user relevant data with `getUserData` just by specifying a document type and the source. Like if you want a character document of a player, you would maybe ask for *char* as the document type, whereas the documentid will be maybe something like *char<steamid>*. Here is what I go for our 5 functions, which concludes our CouchDB connection:

```lua
function getData(documentid)
  result = {}
  result.status, result.content, result.headers = CouchDB.Get(
    CouchDB.address..documentid)
  result.content = json.decode(result.content)
  return result
end

function setData(documentid, documentjson)
  local status, content, headers = CouchDB.Put(CouchDB.address..documentid,
    documentjson)
  if status == 201 then
    return true
  end
  return false
end

function doesDataExist(documentid)
  return CouchDB.Head(CouchDB.address..documentid)
end

function getUserData(src, documentprefix)
  local id = getUserIdentifier(src)
  if id ~= "" and id ~= nil then
    return getData(documentprefix..id)
  else
    return nil
  end
end

function setUserData(src, documentprefix, documentjson)
  local id = getUserIdentifier(src)
  if id ~= "" and id ~= nil then
    return setData(documentprefix..id, documentjson)
  else
    return false
  end
```

```
end
```

3.3 `cfg.lua`

This file will only contain configuration options for other uses who might use the script. Since we used the variables already in the CouchDB connection, writing this file is straight forward. Additionally we are going to add another variable which indicates if this is a whitelist-only server or not.

```
cfg = {}
cfg.dbserver = "http://127.0.0.1:5984/"
cfg.dbname = "tutorial"
cfg.auth = "fxserver:tutorial"
cfg.usewhitelist = true
```

And this was already the entire file, now let us continue to the `auth.lua`, in which the user authenticates himself automatically with our server.

3.4 `auth.lua`

In this file we will make sure when the player is connecting to the server, that he has access to the server, so he is neither banned, nor his ip is banned, and if we use a whitelisting system, he is whitelisted. Furthermore we are going to add a very simple queue system, so that once the server is full, users do not need to spam the connection to get access.

3.5 `server.lua`

3.6 `client.lua`

3.7 `scoreboard.html`

References

[Col18a] CitizenFX Collective. *FiveM - the GTA V multiplayer modification you have dreamt of.* [Online; accessed 20-January-2018]. 2018. URL: https://fivem.net/.

[Col18b] CitizenFX Collective. *Manifest versions :: FiveM Documentation.* [Online; accessed 21-January-2018]. 2018. URL: http://docs.fivem.net/resources/manifest-versions/.

[Con17] FiveM Wiki Contributors. *Running FXServer — FiveM.* [Online; accessed 20-January-2018]. 2017. URL: https://wiki.fivem.net/w/index.php?title=Running_FXServer&oldid=1299.