Scripting Islands of Battle, Workshop 1

Islands of Battle fully supports and encourages scripting. You can choose to play on script-enabled or script-only islands to see how your programming prowess compares to others across the world.

These workshop documents aim to teach you all you need to know about creating a script for Islands of Battle. Several example scripts are available here: GitHub - Pebsie/IoB-Examples: A selection of example scripts for Islands of Battle.

Don't be put off by the length of this one: it acts a full introduction to both Lua for people who are new to the language and/or programming in general, the idea of HTTP APIs and the way that the Islands of Battle API behaves. Once completed the following workshops cover more ground a lot faster.

Getting started

Lua

Islands of Battle uses Lua (https://lua.org) for scripting. You can run any *.lua file as a script and the game will run it. This documentation is designed to cover everything you need to know about the basic syntax of Lua, so there's no need to have any familiarity with it prior to starting.

Loading scripts

You can load a script from within the game client by typing in the name of the Lua file (including extension) and clicking "Load Script" on the login panel. This should be done after selecting an Island to play on and before logging in to that Island, as your script should handle logging in or registering.

Scripts must be stored in the game's save directory. This varies depending on your system.

- Windows: `%appdata%\battle-client\`
- OS X: `User/*user*/Library/Application Support/LOVE/battle-client/`
- Linux: `~/.local/share/love/battle-client/`
-

Finding these folders may be tricky. On Windows type "Run" into the start menu and type `%appdata` on the box that appears, then press Run. This will take you to your AppData folder. Find battle-client there and place your script file in that folder.

On OS X you'll need to display hidden folders. You can do this by navigating to your User directory and pressing CMD + SHIFT + .
Then click navigate Library ➜ Application Support ➜ LOVE ➜ battle-client and place your script file there.

If you're using Linux it's safe to assume that you know how to access system folders.

Once you've typed in the name of your script on the game client and pressed "Load Script" you'll be given a Play, Pause, Reload and Stop button. Pressing Play will start the script, running the code once every three seconds. Pressing Pause will stop execution. Pressing Reload will attempt to reload the script and pressing Stop will unload the script and log you out.

Registration and Logging In

The Basics

Loaded scripts have access to an `api` object. This can be used to make calls to the API by typing `api.get()`. Within the brackets you enter your *request*. All this is doing is making a call to `servername/api/v1/api.php`. Anything written in the brackets is appended to that URL. This is how requests are made to HTTP APIs.

So, for example, if we wanted to register an account we need to send the register command with `username` and `password` parameters. The APIBattle API (which is used to power Islands of Battle) documentation says that commands are sent with a `?a` parameter.

So: `api.get("?a=register&username=demo&password=demo")` will attempt to register an account on the database. As above, this is the same as typing `servername/api/v1/api.php?a=register&username=demo&password=demo` into a web browser.

Task 1: Try this with your own username and password, then Stop the script and try logging into the account that you just created with the login GUI. If you did everything right then you should be able to play the game as if you'd registered an account normally.

In case of death, alter (variables)

A variable is a name given to a value. For example, if my username is to be demo then I can type `username = "demo"`. From now on whenever I type `username` the client knows to use `"demo"` in its place. It's important to note the use of quotation marks around the text: this tells the client that this isn't a piece of code but is text to be used within the game. In programming this is known as a *string*.

Unlike in other languages you do not need to declare variable types in Lua. Both `val = 5` and `val = “demo"` are perfectly viable.

If we plug these values into our registration script the entire thing becomes

```
username = "demo"
password = "demo"
api.get("?a=register&username="..username.."&password="..password)
```

We’re doing something different in the API call there: *concatenation*, or, joining together strings, is done using the `..` operator. The above will send the exact same call to the API as before we modified the script.

What’s the value in doing this?

It’s unlikely, of course, and probably doesn’t bear thinking about, but in the course of battle you *may* wind up a tiny bit dead. Using a variable for the username and password will allow you to set up a new account when needed changing only one line of code rather than potentially 100s.

Getting results (logic)

Whenever you use `api.get()` the game will *return* a value. This means that you can set a variable to your API call and do something based on the result.

Normally, the API would return a result in JSON format. You don’t need to worry about that, however, as the client handles this for us.

To check the result of our attempt to register we can use an `if` statement. Lua is nice for beginners and is fairly readable right out of the box. Look at our modified script below.

```
username = "demo"
password = "demo"
result = api.get("?a=register&username="..username.."&password="..password)

if result.authcode then
  authcode = result.authcode
end
```

Above we're setting the value of a new variable `result` to the `api.get()` function. The result will be what's known as a table, which is just a list of values contained within a variable.
The `if` statement checks whether or not a value within the `result` variable, `result.authcode`, exists (`if val then` is equivalent to `if val == true then`. If a variable exists then its value is both `true` and whatever value it's set to).

As per the API documentation if a `register` command is successful then it will return the `authcode` of the player (which is used every time we want to do stuff as that player). So, it stands to reason that if `result.authcode` exists we have logged in.

The client acknowledges a login when the value of `authcode` is set. So, the line `authcode = result.authcode` will remove the login UI and allow us to see our units on the battlefield in the correct colour.

Task 2: modify the script above to check whether or not `authcode` has already been set *before* attempting to register.

Using the API documentation

What we've done so far might seem basic but it's the groundwork for scripting *everything*. You don't need to know much more than you already do to start building a full on self-playing script.
Here are the key things to remember:

- The 'command' you need to enter starts the `api.get()` call as `"?a="`
- The 'parameters' extend that. After the command it's `&parameter=value`
- `api.get()` *always* returns a value: a table, with the results of your call.
- You can access the result of a table by setting a value to `api.get()` and placing a `.` then the value name.
-

Final task: The API documentation states that the `login` command takes two parameters: `username` and `password` and returns the authcode of the user if the login is successful. Can you modify your script from Task 2 to attempt login *before* attempting to register?
(The solution can be found here: IoB-Examples/task3.lua at master · Pebsie/IoB-Examples · GitHub)