FibEmu

A screenshot of a computer

Description automatically generated

Content

[Introduction 3](#_Toc140652909)

[Installation 4](#_Toc140652910)

[Configuration 7](#_Toc140652911)

[Configuration file 7](#_Toc140652912)

[QA header configuration 9](#_Toc140652913)

[Running you QA 13](#_Toc140652914)

[Example files 14](#_Toc140652915)

[Features 24](#_Toc140652916)

[Supported QuickApp functions 24](#_Toc140652917)

[Supported REST APIs 26](#_Toc140652918)

[Web UI 27](#_Toc140652919)

[Event view 28](#_Toc140652920)

[GlobalVariables view 28](#_Toc140652921)

[Configuration view 29](#_Toc140652922)

[Workflow 29](#_Toc140652923)

[Examples & (tips & tricks) 30](#_Toc140652924)

[Example 1 31](#_Toc140652925)

[Example 2 31](#_Toc140652926)

[Implementation details 32](#_Toc140652927)

[Known issues 33](#_Toc140652928)

## Introduction

This is a Visual Studio Code setup for QuickApp development for the Fibaro HC3. Vscode is one of the most popular development environments and is cross-platform. It’s available for Windows, MacOS, and Linux.

Developing QuickApps directly on the HC3 is challenging as the editor is limited and the debugging tools are even more so. Usually, it means residing to using print statements to try and understand what is happening. Many developers do edit their code in an editor on their computer and copy&paste it to the HC3 to try it out.

This setup consists of a Vscode project that provides an HC3 emulator that allow us to edit, run, and debug QA in Vscode. We can use all the productive tools and plugins that Vscode provides; everything from highlighting coding warnings to help us write code using AI like Copilot.

The QA can interact with the HC3 with the provided Fibaro APIs. Like turning on and off light devices on the HC3, call QAs running on the HC3 etc.

This makes it very quick to develop and test your QuickApp, and you are in control of what resources on the HC3 that your QA can interact with.

It gives you a way to develop and run a QuickApp offline on your PC/Mac/Linux and let it interact with the HC3 in a controlled way.

There is also a simple UI to interact with the QAs UI and see events etc.

Please note that this is not a tutorial for Vscode or git/github.

## Installation

Summary of steps to run

1. Install Visual [Studio Code](https://code.visualstudio.com/). Yes, it’s free Microsoft software(!)
2. Use Vscode to clone the fibemu github repository into a folder where you want to do the development. The repository for downloading it is here <https://github.com/jangabrielsson/fibemu>
3. Have python3 installed on your machine. Make sure that you have set up Path environment variable so that the command can be run from any directory.
4. Install pip and do a “pip install” of the needed python libraries listed in the file requirements.txt

pip install -r requirements.txt

1. Create a config.json file with the credentials to access the HC3.   
   See the file config.json.example and the next chapter “Configuration”.
2. Install the vscode extension "[Local Lua Debugger](https://marketplace.visualstudio.com/items?itemName=tomblind.local-lua-debugger-vscode)" by Tom Blind

A screenshot of a black background with white text

Description automatically generatedMost of the emulator files resides inside the .vscode/emufiles/ directory.

* \_\_pycache\_\_/ is just a place where temporary runtime files are placed
* lspaddon/ is a directory to store editor autocompletion and tooltips info. Work in progress.
* lua/ contains the emulator’s lua environment for your QA. Like the QuickApp and fibaro.\* functions.
* static/ contains stylesheets and js for the web interface
* templates/ are the web pages for the emulator Web UIs
* \_\_init\_\_.py is the main/startup file for the emulator
* api.rest is a test file to work with the HC3 REST API. Work in progress
* fibapy.py is the REST API code for the emulator. Tries to mimic the REST API of the real HC3
* fibenv.py is the main “runtime” for the QA environments we run.
* fibnet.py is the HTTPClienet, UDPClient, WebSocketClient etc implementations.
* todo.txt, my to-do list of stuff I would like to implement
* launch.json is the definition of the different way to start debugging. You can launch debugging with HC3 access, or just run it local without HC3 access. More on this later.
* lua.code-snippets contains shortcuts for inserting code snippets, ex. A QA skeleton, or fibemu code headers.
* settings.json contains Vscode project specific settings. Here you also configure some of the Vscode plugins.
* .gitignore is a file defining what files should not be under source control
* config.json is where you setup HC3 credentials etc.
* requirements.txt is a file with the Python libraries that need to be installed to run the emulator

In principle you should not need to change anything in the .vscode/emufiles/ directory.

The emufiles comes with Lua and libraries so there is no more to download. There is a python wrapper for the lua runtime ([Lupa](https://pypi.org/project/lupa/" \l "which-lua-version" \t "_blank)) so we solve dependencies on luasocket etc. and we don't need any special headers in the QA lua file to invoke/include the emulator/apis to make the QA being able to execute (we don't even need Lua installed on our machine )

The directory, when cloned, comes with a .gitignore file that tells the source control what files should not be controlled. There are two directories specified there.

./dev/

and

./test/

If you create these directories, you can do your development in them and still pull down new releases of the fibemu repository without it impacting your own files. This is because dec/ and test/ is ignored by the source control.

If you want to develop your own code under source control, I recommend that you make soft links from your own .vscode/emufiles to the .vscode/emufiles in the fibemu directory that you closed. Do also the same with task.json. You should keep your own versions of launch.json, settings.json. and lua.code-snippets

A diagram of a program

Description automatically generated

## Configuration

### Configuration file

A screen shot of a computer screen

Description automatically generated

Configuration parameters are a mix of values that can be set in your config.json file and configurations value that are determined by the emulator.

|  |  |
| --- | --- |
| Param | Value |
| host | **192.168.1.57**, This is the IP address of the HC3 |
| user | **admin**, Username credential for HC3 |
| password | **admin**, Password credential for HC3 |
| secret | **Kaka**, example of an user config parameter that can be supplied and use to initialize quickAppVariables. More on this later. |
| dark | **True**, If set to true will make sure that text in console is white. A necessity for Vscode in dark mode |
| colors | **{'SYS': 'brown', 'SYSERR': 'red', 'DEBUG': 'green', 'TRACE': 'blue', 'WARNING': 'orange', 'ERROR': 'red', 'TEXT': 'black', 'DARKTEXT': 'grey'**}, Mapping of text colors used in the console window |
| local | **False**, set to True if no HC3 access is allowed |
| port | **80**, API port on HC3 |
| wport | **5004**, fibemu’s API port |
| whost | **127.0.0.1,** fibemu’s host address |
| wlog | **Warning**, debug level for the fibemu API server (uvicorn) |
| emulator | **emu.lua** |
| init | **None** |
| break | **False** |
| file1 | **/Users/jangabrielsson/Desktop/dev/fibemu/examples/QA\_ui.lua** |
| file2 | **None** |
| file3 | **None** |
| version | **0.32** |
| server | **False** |
| path | **.vscode/emufiles/** |
| argv | **['/Users/jangabrielsson/Desktop/dev/fibemu/.vscode/emufiles/\_\_init\_\_.py',  '-f', '/Users/jangabrielsson/Desktop/dev/fibemu/examples/QA\_ui.lua']** |
| extra | **[]** |
| nogreet | **False** |
| apiURL | **http://127.0.0.1:5004/api** |
| apiDocURL | <http://127.0.0.1:5004/docs> |
| webURL | http://127.0.0.1:5004/ |

QA header configuration  
To give some hints to the emulator what type of QA we have etc. we can give directive like TQAE in our QA file (but a bit different)

Ex.

--%%name=MyQA

--%%type=com.fibaro.binarySwitch

--%%file=qa3\_1.lua,extra;

--%%remote=devices:788,790

--%%remote=globalVariables:myVar,anotherVar

--%%debug=libraryfiles:false,userfilefiles:false

function QuickApp:onInit()

self:debug(self.name,self.type,self.id)

fibaro.call(788,"turnOn")

end

#### Name of QA

--%%name=MyQA

This will be the name your QA will have

#### Type of QA

--%%type=com.fibaro.binarySwitch

Type type of your QA (see list of types here)

QA deviceID

--%%id=1099

This should normally not be set. Let the emulator choose a deviceId for you. (starting at 5000). In some cases it may be a reason to have the same id as an QA on the HC3.

#### Interfaces

--%%interface=power,energy

List of interfaces the QA should have.

#### UI definition

User elements for you QA can be defined. The elements are added in the order they are listed and elements on the same row needs to be in the same --%%u definition.

--%%u={button=’b1’, text=’My Button’, onReleased=’myButtonFun’}

--%%u={slider=’s1’, text=’My Slider’, onChanged=’mySliderFun’}

--%%u={label=’l1’, text=’My Label’}

If we want to buttons on the same row we do one --%%u directive like:

--%%u={{button=’b1’, text=”Turn On”, onReleased=’turnOn’},={{button=’b2’, text=”Turn Off”, onReleased=’turnOff’}}

#### Debug flags

--%%debug=x:y,z:v. TBD

#### QuickAppVariables

--%%var=test:foo

When defining quickAppVariables with

--%%var=foo:42

--%%var=bar:"Hello"

--%%var=baz:{a = 9, b = 19}

The value, right hand of the ':', is an evaluated lua expressions. The environment where the expression is evaluated is limited so you can't call functions etc. It's for setting up constants.

However, there is one variable available, 'config', that is the config file read in.

This means that if we have the config.json

{

"host": "192.168.1.57",

"user": "admin",

"password": "admin",

"secrets": {"user":"admin","pwd2":"hushhush"},

"dark": true,

}

We can define a quickAppVariable in our QA as

--%%var=password:config.secrets.pwd2

and then access it in our code as

function QuickApp:onInit()

    local pwd = self:getVariable("password")

end

The advantage is of course that we don't have to put the secret in plain text in the QA code, and by accident publish it in the forum or commit it to a repository...

#### File inclusion

--%%file=dev/myfile.lua,lib;

This will include dev/myfile.lua as an QA file named “lib”.

Note that your working directory is the Vscode project folder, so your paths starts from there.

If you have many files in a subdirectory, you can define a file root path that will be prepended to the file path.

--%%root=dev/

--%%file=myfile.lua,lib;

--%%file=myfile2.lua,lib2;

#### Remote access

--%%remote=devices:788,790

--%%remote=globalVariables:myVar,anotherVar

It instructs the emulator that it's ok to call device 788,789 o the HC3. As a default, the emulator treats all resources as local (we can read from HC3 but then treat them as local copies) and we enable resources we want to interact with on the HC3 as 'remote'. This goes for other resources also like 'globalVariables'.

A special case is to set the access to \*

--%%remote=devices:\*

This will allow remote access to all devices

--%%allRemote=true

Will set all resources to remote access allowed.

## Running you QA

A screen shot of a computer

Description automatically generated

When the QA starts up it will log the links to the Web UI and the Web API docs. CMD-click on tese links will open the web

### Example files

#### QA\_ui.lua

This a QA that demonstrates how to declare UI elements in the header of the QA using the

--%%u=…

directive

--[[

Simple QA with UI elements

Open browser at http://127.0.0.1:5004/ to interact with this app

--]]

--%%name=QA0

--%%debug=permissions:false,refresh\_resource:true

--%%u={{button='turnOn', text='On', onReleased='turnOn'},{button='turnOff', text='Off', onReleased='turnOff'}}

--%%u={{button='t1', text='A', onReleased='t1'},{button='t2', text='B', onReleased='t1'},{button='t3', text='C', onReleased='t1'},{button='t4', text='D', onReleased='t1'},{button='t5', text='E', onReleased='t1'}}

--%%u={button='test', text='Test', onReleased='testFun'}

--%%u={{button='test', text='A', onReleased='testA'},{button='test', text='B', onReleased='testB'}}

--%%u={slider="slider", max="80", onChanged='sliderA'}

--%%u={label="lblA", text='This is a text'}

function QuickApp:onInit()

self:debug("Started",self.id)

self:setVariable("test","HELLO")

setTimeout(function() self:updateView("lblA","text","FOO") end, 5000)

end

function QuickApp:testFun()

self:debug("Test pressed")

end

function QuickApp:testA()

self:debug("A pressed")

end

function QuickApp:testB()

self:debug("B pressed")

end

function QuickApp:sliderA(ev)

self:debug("Slide A",ev.values[1])

end

function QuickApp:turnOn()

self:debug("Turned on")

self:updateProperty("value",true)

end

function QuickApp:turnOff()

self:debug("Turned off")

self:updateProperty("value",false)

end

#### QA\_api\_test.lua

This is a quite extensive QA testing various APIs

#### QA\_fibaroExtra.lua

This is a QA including fibaroExtra.lua as an extra QA file and creating a QuickAppChild with the fibaroExtra class QUickerAppChild.

--[[

QA using fibaroExtra to create child object

Open browser at http://127.0.0.1:5004/ to interact with this app

fibaroExtra.lua is expected to be avaible in ../TQAE/lib/fibaroExtra.lua

relative to this project

--]]

--%%name=QA\_fibaroExtra

--%%debug=refresh\_resource:true

--%%debug=http:true,hc3\_http:true

--%%file=../TQAE/lib/fibaroExtra.lua,fibaroExtra;

--%%u={{button='turnOn', text='On', onReleased='turnOn'},{button='turnOff', text='Off', onReleased='turnOff'}}

--%%u={{button='t1', text='A', onReleased='t1'},{button='t2', text='B', onReleased='t1'},{button='t3', text='C', onReleased='t1'},{button='t4', text='D', onReleased='t1'},{button='t5', text='E', onReleased='t1'}}

--%%u={button='test', text='Test', onReleased='testFun'}

--%%u={{button='test', text='A', onReleased='testA'},{button='test', text='B', onReleased='testB'}}

--%%u={slider="slider", max="80", onChanged='sliderA'}

--%%u={label="lblA", text='This is a text'}

function QuickApp:onInit()

self:debug("Started",self.id)

class 'MyChild'(QuickerAppChild)

function MyChild:\_\_init(args)

QuickerAppChild.\_\_init(self, args)

self:debug("Child init",self.id)

end

function MyChild:turnOn()

self:debug("Child turned on")

end

function MyChild:turnOff()

self:debug("Child turned off")

end

local child = MyChild{

uid = 'x',

name = 'MyChild',

type = 'com.fibaro.binarySwitch',

}

setTimeout(function() fibaro.call(child.id,"turnOn") end, 1000)

self:event({type='device'},function() end)

end

#### QA\_centralSceneEvent.lua

This a QA that generates a centralSceneEvent with the REST api. It also catches the event using the fibaroExtra event mechanism

--%%name=CentralSceneEvent test

--%%type=com.fibaro.genericDevice

--%%file=examples/fibaroExtra.lua,fibaroExtra;

function fibaro.postCentralSceneEvent(keyId,keyAttribute)

local data = {

type = "centralSceneEvent",

source = plugin.mainDeviceId,

data = { keyAttribute = keyAttribute, keyId = keyId }

}

return api.post("/plugins/publishEvent", data)

end

function QuickApp:onInit()

quickApp=self

fibaro.debugFlags.\_allRefreshStates=true

fibaro.event({type="device",property='centralSceneEvent'},function(env)

local ev = env.event

self:debugf("CentralSceneEvent: %s %s",ev.value.keyId,ev.value.keyAttribute)

end)

setTimeout(function() fibaro.postCentralSceneEvent(2,"Pressed") end,0)

end

#### QA\_http\_test.lua

A QA demonstrating the net.HTTPClient by requesting the time for Stockholm Sweden using worldtimeapi.org.

--%%name=HTTP Test

--%%type=com.fibaro.binarySwitch

--%%debug=http:true,hc3\_http:true,dark:true

function QuickApp:onInit()

self:debug("Started",self.id)

net.HTTPClient():request("http://worldtimeapi.org/api/timezone/Europe/Stockholm",{

options = {

method = "GET",

headers = {

["Accept"] = "application/json"

}

},

success = function(response)

self:debug("Response",response.data)

end,

error = function(err)

self:error("Error",err)

end

})

print("HTTP called") -- async, so we get answer later

end

#### QA\_include\_file.lua

Another QA showing how to include another QA file.

--%%name=Include File QA

--%%type=com.fibaro.binarySwitch

--%%file=examples/include\_file.lua,extra;

--%%debug=libraryfiles:false,userfilefiles:true

local function printf(fmt,...) print(string.format(fmt,...)) end

function QuickApp:onInit()

foo()

end

included\_file.lua just declare a global function named foo.

function foo()

print("FOO")

end

#### QA\_tcp\_test.lua

This QA shows the net.TCPClient api.

--[[

Simple echo server using net.TCPSocket()

On same machine (Liux/MacOS) run

>nc -l 8986

to start a socket server to interact with this app

--]]

PORT = 8986

--%%name=TCP Test

local Event = {}

local function post(ev,self)

self = self or {}

function self:post(ev) setTimeout(function() Event[ev.type](self,ev) end, 0) end

function self:debug(...) print(string.format(...)) end

setTimeout(function() Event[ev.type](self,ev) end,0)

end

function Event:init(ev)

self.sock = net.TCPSocket()

self:debug("init")

self:post{type='connect'}

end

function Event:connect(ev)

self.sock:connect(self.host,self.port,{

success = function(message)

self:debug("connected",message)

self:post{type='prompt'}

end,

error = function(message)

self:debug("connection error:%s", message)

end,

})

end

function Event:prompt(ev)

self.sock:write("Echo:",{

success = function(n)

self:debug("wrote %s bytes",n)

self:post{type='read'}

end,

error = function(message)

self:debug("send error:%s", message)

end,

})

end

function Event:read(ev)

self.sock:read({

success = function(msg)

self:debug("Echo '%s'",msg)

self.sock:write(string.format("Echo '%s'\n",msg))

self:post{type='prompt'}

end,

error = function(message)

self:debug("read error:%s", message)

end,

})

end

function QuickApp:onInit()

self:debug(self.name,self.id)

post({type='init'}, {host="127.0.0.1",port=PORT})

--post({type='init'}, {host="127.0.0.1",port=PORT+1})

end

#### QA\_udp\_echo\_test.lua

QA that implements a simple UDP echo server

--[[

Simple echo server using net.UDPSocket()

On same machine (Liux/MacOS) run

--]]

--------- Not currently working ------------

ADDR = "192.168.1.129"

PORT = 8986

BROADCAST = true

class "UDPServer"

function UDPServer:\_\_init(port, handler)

self.port = port

self.handler = handler

self.udp = net.UDPSocket({

broadcast = BROADCAST,

--timeout = 10000,

reuseport = true,

reuseaddr = true

})

end

function UDPServer:run()

self.udp:bind(ADDR, self.port)

local cb

cb = {

success = function(data, ip, port)

self.handler(self, data, ip, port)

self.udp:receive(cb) -- will read next datagram

end,

error = function(error)

self:debug("UDP server error:", error)

end

}

print("Server waiting")

self.udp:receive(cb)

print("Running UDP server at port ", self.port)

end

--%%name=TCP Test

function QuickApp:onInit()

local server = UDPServer(PORT, function(self, data, ip, port)

print("Recieved", data, ip, port)

self.udp:sendTo("OK", ip, port, {

success = function()

print("Sent ok")

end,

error = function(error)

print('Error:', error)

end

})

end)

server:run()

self.udp = net.UDPSocket({

broadcast = BROADCAST,

--timeout = 10000,

reuseport = true,

reuseaddr = true

})

local seqNr = 1

local function loop()

local msg = "HELLO-"..seqNr

print("Client sending",msg, BROADCAST and "255.255.255.255" or ADDR, PORT)

self.udp:sendTo(msg, BROADCAST and "255.255.255.255" or ADDR, PORT, {

success = function(n)

print("Sent",n,"bytes")

self.udp:receive({

success = function(data,ip,port)

print("Recieved ",data, ip, port)

setTimeout(loop,2000)

end,

error = function(error)

self:debug("Error:", error)

end

})

end,

error = function(error)

print('UPD Client Error:', error)

end

})

fibaro.sleep(1000)

end

loop()

end

#### QA\_udp\_test.lua

A more extensive UDP test

--[[

Simple echo server using net.UDPSocket()

On same machine (Liux/MacOS) run

>nc -u 8986

>nc -ul 8986

to start a socket server to interact with this app

--]]

PORT = 8986

--%%name=TCP Test

function QuickApp:onInit()

self.udp = net.UDPSocket({

broadcast = true,

timeout = 10000

})

local stat,res = pcall(function()

--self.udp:bind("127.0.0.1",PORT)

end)

if not stat then

self:debug("Error binding",res)

end

local payload = "HELLO"

self.udp:sendTo(payload, '255.255.255.255', PORT, {

success = function()

self:receiveData()

end,

error = function(error)

print('Error:', error)

end

})

end

function QuickApp:receiveData()

print("Waiting for data")

self.udp:receive({

success = function(data, ip, port)

print("Recieved",string.char(table.unpack(data)), ip, port)

self:receiveData() -- will read next datagram

end,

error = function(error)

self:debug("Error:", error)

end})

end

#### QA\_websocket\_test.lua

A QA that connects to echo.websocket.events that echoes messages sent

--[[

Simple websocket test

--]]

--%%name=TCP Test

--%%var=url:"wss://echo.websocket.events/"

function QuickApp:onInit()

self:debug("onInit")

local url = self:getVariable("url")

self.sock = net.WebSocketClient()

self.sock:addEventListener("connected", function() self:handleConnected() end)

self.sock:addEventListener("disconnected", function() self:handleDisconnected() end)

self.sock:addEventListener("error", function(error) self:handleError(error) end)

self.sock:addEventListener("dataReceived", function(data) self:handleDataReceived(data) end)

self.sock:connect(url)

--setInterval(function() self:debug("interval") end, 1000)

end

function QuickApp:handleConnected()

self:debug("connected")

self.sock:send("Hello from fibemu")

end

function QuickApp:handleDisconnected()

self:warning("handleDisconnected")

end

function QuickApp:handleError(error)

self:error("handleError:", error)

end

function QuickApp:handleDataReceived(data)

self:trace("dataReceived:", data)

end

#### QA.fqa

This a .fqa file that can be loaded into the emulator and run. At the moment there is no way to debug it. It’s just a proof that we can deploy .fqa files too.

## Features

### Supported QuickApp functions

fibaro.debug(tag,str)

fibaro.warning(tag,str)

fibaro.trace(tag,str)

fibaro.error(tag,str)

fibaro.call(deviceID, actionName, ...)

fibaro.getType(deviceID)

fibaro.getValue(deviceID, propertyName)

fibaro.getName(deviceID)

fibaro.get(deviceID,propertyName)

fibaro.getGlobalVariable(varName)

fibaro.setGlobalVariable(varName ,value)

fibaro.getRoomName(roomID)

fibaro.getRoomID(deviceID)

fibaro.getRoomNameByDeviceID(deviceID)

fibaro.getSectionID(deviceID)

fibaro.getIds(devices)

fibaro.getAllDeviceIds()

fibaro.getDevicesID(filter)

fibaro.scene(action, sceneIDs)

fibaro.profile(profile\_id, action)

fibaro.callGroupAction(action,args)

fibaro.alert(alert\_type, user\_ids, notification\_content)

fibaro.alarm(partition\_id, action)

fibaro.setTimeout(ms, func)

fibaro.clearTimeout(ref)

fibaro.setInterval(ms, func)

fibaro.clearInterval(ref)

fibaro.emitCustomEvent(name)

fibaro.wakeUpDeadDevice(deviceID)

fibaro.sleep(ms)

net.HTTPClient()

net.TCPSocket()

net.UDPSocket()

net.WebSocketClient()

net.WebSocketClientTLS()

*mqtt.Client.connect(uri, options) --no yet*

*<mqttclient>:addEventListener(message,handler) --no yet*

*<mqttclient>:subscribe(topic, options) --no yet*

*<mqttclient>:unsubscribe(topics, options) --no yet*

*<mqttclient>:publish(topic, payload, options) --no yet*

*<mqttclient>::disconnect(options) --no yet*

api.get(call)

api.put(call <, data>)

api.post(call <, data>)

api.delete(call <, data>)

setTimeout(func, ms)

clearTimeout(ref)

setInterval(func, ms)

clearInterval(ref)

json.encode(expr)

json.decode(string)

plugin.mainDeviceId

*plugin.deleteDevice(deviceId) --not yet*

plugin.restart(deviceId)

plugin.getProperty(id,prop)

plugin.getChildDevices(id)

plugin.createChildDevice(prop)

class QuickAppBase

class QuickApp

class QuickAppChild

class <name>

property(get,set)

QuickApp:onInit() -- called at startup if defined

QuickApp - self:setVariable(name,value)

QuickApp - self:getVariable(name)

QuickApp - self:debug(...)

QuickApp - self:trace(...)

QuickApp - self:warning(...)

QuickApp - self:error(...)

QuickApp - self:updateView(elm,type,value)

QuickApp - self:updateProperty(name,value)

QuickApp - self:createChildDevice(props,device)

QuickApp - self:initChildDevices(table)

### Supported REST APIs

…

### Web UI

A screenshot of a computer

Description automatically generated

Event view  
  
A screenshot of a computer

Description automatically generated

### GlobalVariables view

![A screenshot of a computer

Description automatically generated]()

Configuration view  
  
A screenshot of a computer

Description automatically generated

## Workflow

There are some defined vscode Tasks that help in remotely uploading and updating the QA on the HC3 from within the vscode environment

A screenshot of a computer

Description automatically generated

* "QA, download fqa" downloads an QA from the HC3 and saves it as a .fqa file. The task will prompt for deviceID and path where to store. The path/dir needs to exist
* "QA, download and unpack" downloads an QA from the HC3 and saves all QA files as .lua files. It also adds fibemu headers in the main file so it can be opened and run with the emulator . The task will prompt for deviceId and path where to store. The path/dir needs to exist
* "QA, upload" will upload the QA to the HC3. It will prompt for QA file. If '.' is given as argument it will upload the current opened file. This will create a new QA, with a new deviceId on the HC3.
* "QA, update" will try to update QA files, viewLayout, uiCallbacks, and quickAppVariables of an existing QA on the HC3. If '.' is given as argument the file must have set the fibemu header --%%id=<ID> so it knows what QA to update. One can also give the deviceId of the QA on the HC3 that should be updated. This is convenient when developing and avoiding new IDs being "consumed". Sometimes when you update a QA you would not like to update the quickAppVariables. In that case give '-' instead '.' for the current opened file, or -deviceId for an exiting QA on the HC3.

## Examples & (tips & tricks)

* Debug console colors can be setup in config.json (see config.json.example). If you run vscode in dark mode that can be nice to modify to your liking.  
  A quick fix is to set the config parameter dark=true, then the text color will be set to light grey.
* All system logs in the Debug console is tagged as [SYS  ] or [SYSERR] and can thus easily be filtered out in the Debug console's filter field with !SYS
* Break-points in Lua can not be set while the program is running. To add a new break-point, add and restart the QA. This is a limitation of the Lua debugger used. Break-points can be removed while the program is running though.
* The are 3 ways to add configuration. In ~/.fibemu.json. That file, if available, is read first.Can be a good place to store HC3 credentials like ip, user,password.  
  After that, config.json in the vscode project directory is read in and merged with the previous config data.  
  Lastly, config parameters set in the QA source file is added to the config data.
* The vscode launch.json file contains a number of different launch options when debugging. There is usually a "remote" and a "local" version, and the "local" version will run without accessing the HC3. Good to have if developing where there is no access to an HC3, like on the beach...
* The launch option marked with "emu files" include the Lua emulator files in the path for the debugger so that one can step into these files. Normally, you don't want that, instead you only want to step through your own code. Ex. with "emu files" and you step into a 'setTimeout' you will step into the setTimeout implementation...
* There are also 2 python launch options used to run the program with the python debugger ("remote" and "local"). Mainly used for debugging the framework. However, you can't see your Lua code this way.
* There is an option in the launch file to add an Lua init file with the "-I" option. This is a Lua file run before the QAs. It can be a place to setup som "virtual" QAs or other things....

### Example 1

function QuickApp:onInit()  
end

### Example 2

function QuickApp:onInit()  
end

## Implementation details

The environment is a mix of Python and Lua. The blue parts in the picture below is Python and the green is implemented in Lua.

The Lua debugger is started in the green environment when we run the emulator and the vscode IDE can then lets us debug our QA like any other Lua code.

The model is based on that we have a copy of the HC3 resources in our own "Resource DB" and all manipulation of devices, globalvariables, rooms, sections etc are done to towards that database.

If some resource is manipulated in the HC3, we get an event (refreshStates) and update our copy in our database.

If a resource is marked as "remote", e.g. we are allowed to change the resource on the HC3 too, we will "sync" back the change from our database, back to the HC3.

Ex. if we set a globalvariable in the emulator and it is marked "remote" we will sync that back to the HC3. Same for device property changes etc.

The exception is fibaro.call(ID,...) that will call the device on the HC3 if the device is marked "remote".

A diagram of a computer network

Description automatically generated

The picture shows one QA, but the emulator can load several QAs and run them at the "same" time. There will still be only one fibemu thread, and the QAs will share the same timer queue (e.g., they run as Lua coroutines). However, they will have separate fibaro.sleep that don't block each other.

## Known issues

* While the QA is running, break-points can't be added. This is a limitation of the debugger used. Just add the break-point and restart the QA.
* When the emulator crashes, it may leave a process open that keeps the port 5004 in use. The emulator will complain at restart that the port is already bound. You may need to manually kill the process.  
  On Mac:  >kill -9 $(lsof -ti:5004