The Anatomy of QuickApps - Part 2

This deep dive into QuickApp internals provides the foundation for advanced QuickApp development. Understanding these mechanisms helps you write more efficient, robust, and well-architected QuickApps.

> Disclaimer 1: We're venturing into undocumented territory. Fibaro can change how things work at any time (even documented things!). 😉
>
> Disclaimer 2: This explains how I believe some QuickApp functionality is implemented by Fibaro. It's most likely not exactly how Fibaro does it, but it should match the observable behavior. If I'm wrong, please correct me.
>
> Prerequisites: Also see Fibaro's documentation for QuickApp coding: https://docs.fibaro.com

📖 Table of Contents

    - Read-Only vs Persistent Properties
    - Events and Triggers
- Part 6: UI Management
    - `updateView()` Method
    - ViewLayout Structure
    - UI Event Handling
- Part 7: Logging Methods
    - `debug()`, `trace()`, `warning()`, `error()`
    - TAG System Implementation

🌐 Inter-QuickApp Communication

- Part 8: Method Invocation
    - `fibaro.call()` Mechanism
    - Public Method Exposure
    - REST API Access
- Part 9: Execution Model
    - Single-Threaded "Operator" Model
    - Synchronous vs Asynchronous Calls
    - Deadlock Prevention

📋 Summary and What's Next

Let's recap from the previous post:

✅ **Classes** are templates for creating objects
✅ **Objects** are implemented as Lua tables with key-value pairs
✅ **QuickApp** is a class provided by Fibaro for creating QA devices
✅ **Extension Pattern**: We extend QuickApp with our own method definitions
✅ **Lifecycle**: Fibaro loads our code → creates QuickApp object → calls `onInit()`

Key point: `onInit()` is always called after all code has been loaded, regardless of where it's defined in your code.

Part 1: QuickApp Class Extension

Adding Fields to QuickApp Class

Just to prove that QuickApp is a real class:

```
QuickApp.myField = "Hello"  -- Extend the class with a field

function QuickApp:onInit()
    self:debug(self.myField)  -- "Hello" - copied to our object inst
end
```

Behind the scenes: Fibaro's `class` function creates `userdata` objects that we can't easily inspect, but they behave like Lua tables for most practical purposes.

Code vs Object Access

During code loading (top level):

```
-- ✅ Can extend the class
QuickApp.myField = "default value"

-- ❌ Can't use self - object doesn't exist yet
-- self:debug("This will fail!")
```

After object creation (in methods):

```
function QuickApp:onInit()
    -- ✅ Can use self - object now exists
    self:debug("Object is alive!")
end
```

Why this matters: The main purpose of QA code is to extend the QuickApp class with your methods. Fibaro creates the object and calls your `onInit()` after all code has loaded.

Best practices for initialization

```
-- ✅ Good practice - main initialization in onInit()
function QuickApp:onInit()
    -- Code has loaded, object exists, can use self
    self:debug("Starting up...")
end

-- ⚠️ Careful with order if initializing outside onInit()
local myVar = "Hello"  -- This works - runs at load time
```

Global QuickApp access pattern

```
myQuickApp = nil

local function myPrint(str)
    myQuickApp:debug("TEST:", str)
end

function QuickApp:onInit()
    myQuickApp = self  -- Save reference for later use
    myPrint("Hello")   -- Now we can use it
end
```

⚠️ Caution: You can't use `myPrint()` until `onInit()` has run and set up the variable.

Fibaro's global variable

Fibaro actually assigns the QuickApp object to a global variable `quickApp`, but it doesn't get assigned until **after** `:onInit()` exits:

```lua
-- ❌ This will fail:
local function foo()
    quickApp:debug("OK")
end

function QuickApp:onInit()
    foo()  -- Error: "attempt to index a nil value (global 'quickApp
end
```

```lua
-- ✅ This works:
local function foo()
    quickApp:debug("OK")
end

function QuickApp:onInit()
    setTimeout(foo, 0)  -- Delayed execution - prints "OK"
end
```

Why? The `setTimeout` callback runs after `onInit()` has finished and `quickApp` has been assigned.

Part 2: Device Table Structure

Common QuickApp methods

The most common predefined methods for the QuickApp class:

```lua
function QuickApp:debug(...)        -- Logging with automatic tag
function QuickApp:trace(...)        -- Same as debug with different
function QuickApp:warning(...)      -- Warning-level logging
function QuickApp:error(...)        -- Error-level logging

function QuickApp:getVariable(varName)           -- Get QuickApp var
function QuickApp:setVariable(varName, value)   -- Set QuickApp vari

function QuickApp:updateProperty(property, value)    -- Update devic
function QuickApp:updateView(element, type, value)   -- Update UI el

function QuickApp:createChildDevice(properties, constructor)  -- Cre
function QuickApp:removeChildDevice(id)                       -- Remo
function QuickApp:initChildDevices(map)                       -- Init
```

Understanding Device Definitions

All devices on the HC3 are represented as Lua table structures accessible via API:

```
device = api.get("/devices/78")
print(device.id)  -- Prints the device ID
```

> Tip: Check the “Swagger” page of your HC3 (button with the {...} icon in the lower-left of the web UI) to see all available API calls.

Typical QuickApp Device Structure

```
{
    name = "MyQuickApp",                -- Device name from Web UI
    id = 78,                            -- Assigned deviceId number
    roomID = 219,                       -- Room assignment (0 = unassi
    type = "com.fibaro.binarySwitch", -- Device type (determines UI/
    baseType = "com.fibaro.actor",    -- Common base type
    enabled = true,                     -- Device enabled state
    visible = true,                     -- Device visibility
    isPlugin = true,                    -- Always true for QuickApps

    interfaces = {                      -- Supported interfaces
        "light",
        "quickApp"
    },

    parentId = 0,                       -- Parent device (for QuickApp

    properties = {                      -- Device properties
        value = true,                   -- Main device value
        dead = false,                   -- Device status
        deviceIcon = 90,                -- UI icon
        categories = { "lights" },    -- UI categories

        quickAppVariables = {         -- Your QuickApp variables
            -- List of {name="varName", value="varValue"} objects
        },

        viewLayout = {                -- UI definition structure
            -- Button, label, slider definitions
        },

        uiCallbacks = {               -- UI event mappings
            {
                name = "mySlider",
                callback = "slider",
                eventType = "onChanged"
            }
        },

        mainFunction = "-- [DEPRECATED] Your Lua code here"
    },
```

```
    actions = {                         -- Standard device actions
        toggle = 0,                     -- Number = parameter count
        turnOff = 0,
        turnOn = 0
    },

    modified = 1590043821,              -- Unix timestamp
    created = 1590043821                -- Unix timestamp
}
```

> Update: The code is no longer stored in `mainFunction`. Modern QuickApps use separate files associated with the device. More on that later.

Device Management via API

Update device properties:

```
api.put("/devices/78", {properties = {value = false}})  -- Update va
api.put("/devices/78", {enabled = false})               -- Disable d
```

> These API calls can also be used from Scenes.

Part 3: QuickApp Object Creation

Class hierarchy

```
Device                    -- Base device class
  ↓
QuickAppBase              -- Base for QuickApp classes
  ↓
QuickApp                  -- Your QuickApp class
QuickAppChild             -- Child device class
```

Object creation process

```
--- Simplified creation process ---
deviceId = 78
deviceTable = api.get("/devices/" .. deviceId)  -- Get device defini
quickApp = QuickApp(deviceTable)                -- Create object wit
if quickApp.onInit then
    quickApp:onInit()                           -- Call initializati
end
```

Note: This is most likely not exactly how Fibaro implements it, but the end result is the same.

Accessing device properties

The QuickApp constructor copies the most important device table values into the object:

```lua
function QuickApp:onInit()
    self:debug("Name:", self.name)                    -- Device name
    self:debug("Device ID:", self.id)                 -- Device ID
    self:debug("Type:", self.type)                    -- Device type
    self:debug("Value:", tostring(self.properties.value))  -- Main v
end
```

Result: Some device table fields become accessible via `self.*`

Part 4: Variable Management

getVariable() and setVariable()

```lua
function QuickApp:getVariable(varName)
function QuickApp:setVariable(varName, value)
```

How getVariable() works

The `self:getVariable(varName)` method searches through the `quickAppVariables` list:

```lua
-- QuickApp variables are stored as:
quickAppVariables = {
    {name = "varName1", value = "value1"},
    {name = "varName2", value = "value2"},
    -- ...
}

-- Implementation (simplified):
function QuickApp:getVariable(varName)
    for _, var in ipairs(self.properties.quickAppVariables) do
        if var.name == varName then
            return var.value
        end
    end
    return ""  -- ⚠️ Returns empty string, not nil!
end
```

Issues with QuickApp variables

❌ Performance issue: Linear search through the list—gets slower with more variables.

❌ No nil distinction: Returns `""` instead of `nil` for missing variables.

```lua
-- ❌ Can't distinguish between missing and empty:
local val = self:getVariable("myVar")
if val ~= "" then
    self.myField = val
end
```

```
-- ✅ Would be better with nil:
-- self.myField = self:getVariable("myVar") or self.myField
```

Variable types

From the web UI: Variables added via the web UI are always stored as strings.

From code: You can store any Lua type:

```
self:setVariable("myTable", {a = 9, b = 19})  -- Stores as table
local tbl = self:getVariable("myTable")       -- Retrieved as table
```

Default value pattern

```
QuickApp.myField = "default value"  -- Class-level default

function QuickApp:onInit()
    local val = self:getVariable("myVar")
    if val ~= "" then
        self.myField = val  -- Override with user setting
    end
    self:debug(self.myField)
end
```

Part 5: Property Management

updateProperty() method

```
function QuickApp:updateProperty(propertyName, value)
```

Read-only vs persistent updates

❌ **Direct property updates (temporary):**

```
self.properties.value = 42  -- Updates locally but doesn't persist
```

The main problem with doing this is that no event is generated to indicate the QA's property changed. Instead, use `self:updateProperty(...)`.

✅ **Persistent property updates:**

```
self:updateProperty("value", 42)  -- Persists and triggers events
```

Why use updateProperty()?

1. **Persistence**: Changes are saved to the device table
2. **Events**: Some properties trigger system events/notifications
3. **Consistency**: Other components (Scenes) are notified of changes

Example:

```
-- Update the main device value
self:updateProperty("value", 42)
```

```
-- This updates self.properties.value AND persists it
```

Variable updates and events

When you call `self:setVariable(varName, value)`:

1. Updates the `self.properties.quickAppVariables` list
2. Fires a `DevicePropertyUpdatedEvent`
3. **Sends the entire variable list** as the event value (not just the changed variable)

Performance consideration: Large variable lists create large events.

Code updates create events too

When you modify QA code and save it:

```
-- Fibaro essentially does:
self:updateProperty("mainFunction", newCode)
```

This triggers a `DevicePropertyUpdatedEvent` with the **entire code** in the event! For large QuickApps (3000+ lines), this creates very large events.

API-Based Updates

You can update device properties via API:

```
api.put("/devices/88", {enabled = false})  -- Disable device (causes
```

⚠️ Warning: API updates often restart the QuickApp. Property updates via `updateProperty()` avoid restarts.

Part 6: UI Management

---

updateView() method

```
function QuickApp:updateView(element, type, value)
```

This function updates the UI elements (buttons, labels, sliders) defined for your QuickApp.

UI element updates

Updating button text:

```
self:updateView("myButton", "text", "New text for this button")
```

Updating slider value:

```
self:updateView("mySlider", "value", "50")  -- Must be string!
```

⚠️ Important: Values must be strings, or the update may not work properly.

viewLayout structure

UI element definitions are stored in the `viewLayout` property. When you update elements, changes are reflected in this structure.

Alternative API method

You can achieve the same result using the API directly:

```
api.post("/plugins/updateView", {
    deviceId = self.id,
    componentName = "myButton",
    propertyName = "text",
    newValue = "New Text"
})
```

UI event handling

UI interactions generate events that trigger QuickApp methods:

Button event structure:

```
{
    eventType = "onReleased",
    elementName = "button1",
    deviceId = 985,
    values = {nil}
}
```

Slider event structure:

```
{
    eventType = "onChanged",
    elementName = "slider",
    deviceId = 985,
    values = {39}  -- Current slider value
}
```

Handling UI events

Define methods with the same name as your UI elements:

```
function QuickApp:button1(event)
    self:debug("Button clicked")
end

function QuickApp:slider(event)
    local value = event.values[1]
    self:debug("Slider value set to", value)

    -- Best practice: Update slider value to prevent drift
    self:updateView("slider", "value", tostring(value))
end
```

Reading UI element values

The challenge: There's no built-in function to read current UI element values.

The solution: Parse the viewLayout structure:

```lua
local function getView(deviceId, name, typ)
    local function find(s)
        if type(s) == 'table' then
            if s.name == name then
                return s[typ]
            else
                for _, v in pairs(s) do
                    local r = find(v)
                    if r then return r end
                end
            end
        end
    end

    local viewData = api.get("/plugins/getView?id=" .. deviceId)
    return find(viewData["$jason"].body.sections)
end

-- Usage:
local buttonText = getView(self.id, "myButton", "text")
local sliderValue = getView(self.id, "mySlider", "value")
```

Part 7: Logging Methods

Logging functions

```lua
function QuickApp:debug(...)
function QuickApp:trace(...)     -- Same as debug with different tag
function QuickApp:warning(...)   -- Warning-level logging
function QuickApp:error(...)     -- Error-level logging
```

How logging works

These methods are similar to `fibaro.debug(tag, ...)` but with automatic tagging:

```lua
-- Simplified implementation:
function QuickApp:debug(...)
    local str = table.concat({...})
    fibaro.debug(__TAG, str)
end
```

The TAG system

- `__TAG` is a global variable set to `"QuickApp" .. self.id` by default
- **You can customize it**: `__TAG = "MyApp"` changes the log prefix
- **Variable arguments**: `debug` accepts any number of arguments via `...`

Usage examples

```lua
function QuickApp:onInit()
    self:debug("Starting QuickApp")           -- "QuickApp123: Start
    self:warning("This is a warning")         -- Warning-level messa
    self:error("Something went wrong")        -- Error-level message

    -- Multiple arguments
    self:debug("Value:", self.properties.value, "Type:", type(self.p
end

-- Custom tag
__TAG = "MyCustomApp"
function QuickApp:onInit()
    self:debug("Custom tagged message")       -- "MyCustomApp: Custo
end
```

Part 8: Method Invocation

fibaro.call() mechanism

All QuickApp methods can be called remotely:

```lua
fibaro.call(deviceId, methodName, arg1, arg2, ...)
```

Example: Set a QuickApp variable on another device:

```lua
fibaro.call(55, "setVariable", "Test", 77)
```

Public method exposure

⚠️ **Important**: All methods you add to the QuickApp class become **publicly accessible**:

1. **From other QuickApps** via `fibaro.call()`
2. **From Scenes** via `fibaro.call()`
3. **From external systems** via REST API

REST API access

External systems can call your methods:

```
POST http://<HC3_IP>/api/devices/<deviceId>/action/<methodName>
Content-Type: application/json

{
    "args": [value1, value2, ...]
}
```

Privacy strategies

Problem: Sometimes you don't want to expose internal logic.

Solution 1: Keep functions outside the QuickApp class:

```lua
local quickApp = nil
local interval = 30

local function loop()  -- Private function
    -- Poll external server and update UI
    fibaro.setGlobalVariable("myValue", value)
    quickApp:updateView("myLabel", "text", tostring(value))
end

function QuickApp:onInit()
    quickApp = self
    setInterval(loop, interval * 1000)
end
```

Solution 2: Pass `self` as a parameter to avoid a global variable:

```lua
local interval = 30

local function loop(self)  -- Private function with self parameter
    -- Poll external server and update UI
    fibaro.setGlobalVariable("myValue", value)
    self:updateView("myLabel", "text", tostring(value))
end

function QuickApp:onInit()
    setInterval(function() loop(self) end, interval * 1000)
end
```

Trade-offs:

- Global variable approach: Simpler, less parameter passing
- Parameter approach: No global state, more explicit

Part 9: Execution Model

Single-threaded “Operator” model

Each QuickApp has **one "Operator"** - QuickApps are single-threaded.

How fibaro.call() works

Local method call:

```lua
self:turnOn()  -- Simple function call: self.turnOn(self)
```

Remote method call:

```lua
fibaro.call(77, "turnOn")  -- Complex inter-process communication
```

The communication process

1. **Operator-55** calls `fibaro.call(77, "turnOn")`
2. **Operator-55** waits for acknowledgment

3. **Operator-77** receives request in "mailbox"
4. **Operator-77** checks if method exists:

```
if self['turnOn'] and type(self['turnOn']) == 'function' then
    self['turnOn']()  -- Call the method
end
```

5. **Operator-77** acknowledges completion back to **Operator-55**

The deadlock problem

Self-calling deadlock:

```
fibaro.call(self.id, "turnOn")  -- QuickApp calls itself
```

What happens:

1. Operator waits for acknowledgment
2. Operator can't check mailbox (busy waiting)
3. Request never gets processed
4. **Deadlock!**

Asynchronous solution

Fibaro introduced async calls (firmware 5.031.33+):

```
fibaro.useAsyncHandler(true)   -- Default: async (recommended)
fibaro.useAsyncHandler(false)  -- Synchronous (old behavior)
```

Async vs sync behavior

Synchronous (old way):

```
fibaro.useAsyncHandler(false)
self:debug("Calling turnOn")
fibaro.call(self.id, "turnOn")  -- Waits 5+ seconds
self:debug("Done")
```

Asynchronous (new way):

```
fibaro.useAsyncHandler(true)   -- Default
self:debug("Calling turnOn")
fibaro.call(self.id, "turnOn")  -- Returns immediately
self:debug("Done")              -- Prints immediately
```

Return values and error handling

Current limitation: `fibaro.call()` doesn't return values or error messages.

REST API advantage: External REST calls do return error messages:

```
HTTP 404: Device not found
HTTP 400: Method does not exist
```

Future possibility: Fibaro may add return values and error handling to `fibaro.call()`.

Best practices

1. **Use async mode** (default) to avoid deadlocks
2. **Keep timeouts in mind** for sync calls (5+ second timeout)
3. **Use REST API** for external integrations with error handling
4. **Design methods carefully** - they become public interfaces

---

Summary and What's Next

What you've learned

QuickApp Architecture: ✅ Class extension patterns and field management
✅ Device table structure and property access
✅ Object creation and initialization lifecycle
Core Methods: ✅ Variable management (`getVariable` / `setVariable`)
✅ Property persistence (`updateProperty`)
✅ UI manipulation (`updateView`)
✅ Logging system with automatic tagging
Communication: ✅ Inter-QuickApp method calls (`fibaro.call`)
✅ Public method exposure and privacy strategies
✅ Single-threaded execution model
✅ Async vs sync call behavior

Key takeaways

Property Management:

- Treat `self.*` values as read-only unless using update methods
- Use `updateProperty()` and `setVariable()` for persistence
- Be aware of event generation and performance implications

Method Design:

- All QuickApp methods become publicly accessible
- Consider keeping private logic outside the class
- Design methods as public interfaces

Execution Model:

- QuickApps are single-threaded with one "Operator"
- Async calls prevent deadlocks (use default async mode)
- No return values from `fibaro.call()` (yet)

What's next?

In **Part 3**, we'll explore:

- **QuickAppChildren** - Creating and managing child devices
- **Advanced UI patterns** - Complex viewLayout structures
- **Event system** - Device events and triggers in detail

- **Performance optimization** - Best practices for resource efficiency
- **Error handling** - Robust QuickApp development patterns

📚 Quick Reference Index

Core Methods

Method	Usage	Notes
`getVariable(name)`	Get QA variable	Returns `""` if not found
`setVariable(name, val)`	Set QA variable	Triggers events
`updateProperty(prop, val)`	Update device property	Persists changes
`updateView(elem, type, val)`	Update UI element	Value must be string
`debug(...)`	Log message	Uses `__TAG` prefix

Communication

Pattern	Usage	Notes
`self:method()`	Local call	Direct function call
`fibaro.call(id, "method", ...)`	Remote call	Async by default
`fibaro.useAsyncHandler(bool)`	Set call mode	`true` = async, `false` = sync

Best Practices

Practice	Reason	Example
Use `updateProperty()`	Persistence + events	`self:updateProperty("value", 42)`
Keep private functions outside class	Avoid public exposure	`local function helper() ... end`
Always use string for `updateView()`	API requirement	`self:updateView("slider", "value", "50")`
Update slider values in handlers	Prevent drift	`self:updateView("slider", "value", tostring(ev.values[1]))`