





The Anatomy of QuickApps – Part 3: Execution Model Deep Dive

Advanced QuickApp development: understanding the internal Operator model.

Table of Contents

-  The Anatomy of QuickApps – Part 3: Execution Model Deep Dive
 -  Table of Contents
 -  Important disclaimers
 - Part 1: Recap – QuickApp class structure
 - The QuickApp Table Structure
 - Dynamic method extension
 - Runtime method addition
 - Part 2: Method invocation mechanisms
 - Local vs remote calls
 - The fibaro.call() magic
 - Deadlock prevention
 - Part 3: The persistence problem
 - The QuickApp lifecycle
 - Minimal QuickApp Example
 - Why QuickApps don't terminate
 - Answering external requests
 - The disabled QuickApp test
 - Part 4: The Operator model implementation
 - Understanding the QuickApp “Operator”
 - Key implementation details
 - Request processing
 - The table.unpack() pattern
 - Built-in callAction method
 - Part 5: Execution model deep dive
 - The setTimeout pattern
 - Single-threaded nature
 - The blocking problem
 - The fibaro.sleep() problem
 - setTimeout vs busy loops

- [setInterval convenience](#)
- [Key takeaway](#)
- [Simple QuickApp simulator](#)
- [Part 6: Advanced handler patterns](#)
 - [Two categories of incoming requests](#)
 - [Custom action handler](#)
 - [Custom UI handler](#)
 - [Custom dispatch logic](#)
 - [Handler responsibility](#)
 - [When to use custom handlers](#)
 - [Implementation responsibility](#)
- [Summary and best practices](#)
 - [Key takeaways](#)
 - [Practical guidelines](#)
 - [What's next?](#)
 -  [Quick Reference Index](#)
 - [Core Concepts](#)
 - [Essential Functions](#)
 - [Handler Patterns](#)
 - [Best Practices](#)

Important disclaimers

- The "Operator" model presented is a conceptual framework to understand QuickApp execution.
 - Code examples are simplified and may not reflect exact HC3 implementation details.
 - The focus is on practical patterns for robust QuickApp development.
-

Part 1: Recap – QuickApp class structure

The QuickApp Table Structure

In the previous post, we explored how the QuickApp class is fundamentally a Lua table with fields and functions that we can extend:

```
{
  id = 78,                      -- deviceId of QuickApp
  name = "MyApp",               -- Name shown in web UI device li
  type = "com.fibaro.binarySwitch", -- Device type selected during c
  onInit = function(self) end,  -- The onInit function (user-defi
  debug = function(self,...) end, -- Built-in debug function
```

```

trace = function(self,...) end, -- Built-in trace function
warning = function(self,...) end, -- Built-in warning function
error = function(self,...) end, -- Built-in error function
updateProperty = function(self,prop,value) end, -- Property update
updateView = function(self,elm,type,value) end, -- UI update metho
setVariable = function(self,var,value) end, -- Variable setter
getVariable = function(self,var) end, -- Variable getter
-- ... additional methods
}

```

⚠ Note: The `id`, `name`, and `type` fields are not available until the QuickApp object is instantiated.

Dynamic method extension

We can extend the class by adding functions:

```

function QuickApp:turnOn()
    self:debug("TurnOn")
end

```

This adds the field to the class:

```

{
    -- ... existing fields
    turnOn = function(self)
        self:debug("TurnOn")
    end
}

```

Runtime method addition

Can we add methods after object creation?

```

function QuickApp:onInit()
    function self:myNewMethod(x)
        print(x)
    end
end

```

✅ Yes! This dynamically adds:

```

self.myNewMethod = function(self, x)
    print(x)
end

```

This allows runtime extension of QuickApp functionality.

Part 2: Method invocation mechanisms

Local vs remote calls

Local method calls within our code:

```
self:myMethod(...) -- Direct function call
```

We find the key `myMethod` in the `self` object and apply it to the arguments:

```
self = {  
  myMethod = function(self, ...)  
    -- method code  
  end  
  -- ...  
}
```

Remote method calls from other QuickApps:

```
fibaro.call(deviceId, "methodName", arguments...)
```

The fibaro.call() magic

This is where the **second piece of magic** occurs. The HC3 system manages inter-QuickApp communication through a sophisticated request/response mechanism.

Deadlock prevention

In the previous post, we explained the "Operator" analogy and why this could create deadlock:

```
fibaro.call(self.id, "turnOn") -- Self-calling deadlock potential
```

Fibaro introduced **asynchronous behavior by default** to prevent these issues.

Part 3: The persistence problem

The QuickApp lifecycle

When your code is loaded, Fibaro follows this sequence:

1. **Code Execution:** Your top-level code runs immediately
2. **Function Definition:** Functions are defined but not executed
3. **Object Creation:** Fibaro creates the QuickApp object instance
4. **onInit Call:** If present, `onInit()` is called

Example execution order:

```
print("Hello") -- ← Executes immediately during load  
function QuickApp:onInit()  
  self:debug("Good bye") -- ← Executes after object creation  
end
```

Output sequence:

1. "Hello" (during code load)
2. "Good bye" (after object creation and onInit call)

Minimal QuickApp Example

This is a **valid QuickApp** with no explicit QuickApp class usage:

```
setInterval(function()  
    fibaro.setGlobalVariable("Time", os.date("%c"))  
end, 60*1000)
```

Why this works:

- Updates a global variable every minute with current time
- No `QuickApp:onInit` needed
- No explicit `QuickApp` references required
- Uses only built-in HC3 functions

Why QuickApps don't terminate

The fundamental question: Why doesn't a QuickApp terminate after executing this simple code?

```
print(42)  -- No onInit, no loops, nothing else
```

The answer: Even with no active Lua code, the QuickApp must remain available to handle:

1. **Remote method calls** from other QuickApps
2. **Scene invocations** via `fibaro.call()`
3. **UI button presses** and slider interactions
4. **System events** and notifications

Someone needs to be "at home" to answer these requests - our **"Operator"**.

Answering external requests

Other QuickApps or Scenes can call your methods:

```
fibaro.call(yourDeviceId, "setVariable", "test", "42")
```


The QuickApp **must remain active** to process these incoming requests.

The disabled QuickApp test

Experiment: Testing QuickApp availability

```
-- Call a QuickApp (deviceId 78)  
fibaro.call(78, "setVariable", "A", "42")  -- Set variable "A" = "42"  
  
-- Disable the QuickApp  
api.put("/devices/78", {enabled = false})  
  
-- Try another call  
fibaro.call(78, "setVariable", "B", "17")  -- Set variable "B" = "17"
```

Result:

-  Variable "A" gets set (QuickApp was active)

- **✗** Variable "B" is NOT set (QuickApp was disabled)

Conclusion: No one was "at home" to handle the second request because the Operator had "left for coffee."

Part 4: The Operator model implementation

Understanding the QuickApp "Operator"

Let's implement our conceptual "Operator" - the main loop of the QuickApp framework:

```
-- Simplified QuickApp runtime implementation
local device = api.get("/devices/78")      -- Get device table de
loadstring(device.properties.mainFunction()) -- Load and execute us
local app = QuickApp(device)               -- Create QuickApp obj
if app.onInit then app:onInit() end        -- Call onInit if it e
quickApp = app                             -- Set global quickApp

-- The Operator main loop
local function Operator()
    local request = getNextRequest()        -- Get next incoming r
    if request then
        local method = request.method      -- Extract method name
        local args = request.args          -- Extract arguments

        if quickApp[method] then           -- Does method exist?
            quickApp[method](quickApp, table.unpack(args)) -- Call it!
        else
            quickApp:warning("Method ", method, " does not exist")
        end
    end
    setTimeout(Operator, 0)                -- Schedule next itera
end

Operator() -- Start the Operator loop
```

Key implementation details

Code storage: User code is stored in `device.properties.mainFunction` as a plain string:

```
-- Example: "print(42)"
-- Or: "function QuickApp:onInit() self:debug('Hello') end"
```

`loadstring()`:

- Built-in Lua function for compiling and loading code from strings
- Not available in the QuickApp Lua environment
- Used internally by the HC3 system

Object creation sequence:

1. Load and execute user code
2. Create QuickApp object instance
3. Call `onInit()` if defined
4. Set global `quickApp` reference
5. Start Operator loop

Request processing

Request structure (conceptual):

```
{
  method = "turnOn",
  args = {}
}
```

Method dispatch:

```
if quickApp['turnOn'] then -- Check if method exists
  quickApp['turnOn'](quickApp, table.unpack({})) -- Call with sel
end
```

The `table.unpack()` pattern

Understanding argument unpacking:

```
function test(a, b, c)
  return a + b + c
end

local rest = {2, 3}
print(test(1, table.unpack(rest))) -- Prints 6
```

How it works:

- `table.unpack(rest)` becomes `2, 3`
- Final call: `test(1, 2, 3)`
- Result: `1 + 2 + 3 = 6`

Built-in `callAction` method

The QuickApp class provides a method for this pattern:

```
function QuickApp:callAction(method, ...)
  -- Internal implementation similar to our Operator dispatch
end
```

Now you understand how to implement this yourself! 🎯

Part 5: Execution model deep dive

The `setTimeout` pattern

Why `setTimeout()` instead of while loops?

The Operator uses:

```
setTimeout(Operator, 0) -- Schedule next iteration
```

Why not a simple loop?

```
while true do
  -- Process requests
end
```

Single-threaded nature

Critical concept: Nothing runs in parallel inside a QuickApp.

The `setTimeout()` function:

- Built-in HC3 function (not standard Lua)
- Adds the function to a queue to run in the future
- Time in milliseconds—`setTimeout(func, 0)` means “run ASAP”
- Not immediate execution—other queued functions may run first

The blocking problem

Dangerous code example:

```
function QuickApp:onInit()
  while true do
    print("Hello")
    fibaro.sleep(1000) -- ⚠️ BLOCKS EVERYTHING!
  end
end
```

What happens:

1. `onInit()` never exits
2. Operator never starts
3. No `fibaro.call()` requests can be processed
4. QuickApp becomes **unresponsive**

The `fibaro.sleep()` problem

Important limitation: `fibaro.sleep()` does a “busy sleep”:

- ❌ Does not yield time to other functions
- ❌ Does not allow `setTimeout` functions to run
- ❌ Blocks the entire QuickApp

Missed opportunity: Fibaro could have engineered `fibaro.sleep()` to yield time to other “threads,” but they chose not to.

setTimeout vs busy loops

❌ Wrong approach - Blocking loop:


```
function QuickApp:onInit()
  while true do
    -- Do work
    fibaro.sleep(1000) -- BLOCKS everything else
  end
end
```

✓ Correct approach - Non-blocking loop:

```
function QuickApp:onInit()
  local function loop()
    -- Do work
    setTimeout(loop, 1000) -- Schedule next iteration
  end
  loop() -- Start the loop
end
```

Why this works:

- Each iteration completes quickly
- `setTimeout` schedules the next iteration
- Operator can process requests between iterations
- QuickApp remains responsive

setInterval convenience

Even better - use `setInterval`:

```
function QuickApp:onInit()
  setInterval(function()
    -- Do work every second
  end, 1000)
end
```

Benefits:

- Based on `setTimeout` internally
- Automatic recurring execution
- Non-blocking by design

Key takeaway

Always use `setTimeout` or `setInterval` when looping in your QuickApp:

- ✓ Ensures Operator can receive incoming requests
- ✓ Allows `fibaro.call()` processing
- ✓ Enables UI button/slider interactions
- ✓ Maintains QuickApp responsiveness

Remember: Whenever your code is busy running loops, nothing else can happen.

Simple QuickApp simulator

Note: Here's a very simple ["QuickApp simulator"](#) (~20 lines of code):

- Runs in any standard Lua environment with coroutines
- Implements `setTimeout` and `:onInit` logic
- No "Operator" concept (no external events)
- Easy to see how external event handling could be added

Part 6: Advanced handler patterns

Two categories of incoming requests

In reality, the Operator sorts incoming requests into **two categories**:

1. **Actions** - Method calls via `fibaro.call()`
2. **UIEvents** - User interactions with buttons, sliders, etc.

Custom action handler

Override default action handling:

```
function QuickApp:actionHandler(action)
    -- Custom action processing logic
    -- YOU must handle method dispatch yourself
end
```

⚠ Warning: If you define this handler, the Operator will send all incoming requests to your function instead of the default method dispatch.

Example action structure:

```
{
    args = {},
    actionName = "turnOn",
    deviceId = 987
}
```

Log message example:

```
onAction: {"args":[],"actionName":"turnOn","deviceId":987}
```

Custom UI handler

Override default UI event handling:

```
function QuickApp:UIHandler(UIEvent)
    -- Custom UI event processing logic
    -- YOU must handle UI logic yourself
end
```

Example UIEvent structure:

```
{
    values = {nil},
}
```

```

    elementName = "button1",
    eventType = "onReleased",
    deviceId = 987
}

```

Log message example:

```
UIEvent: {"values":[null],"elementName":"button1","eventType":"onRel
```

Custom dispatch logic

Why use custom handlers?

1. Single entry point for all UI events
2. Custom routing logic for complex UIs
3. Advanced logging and debugging
4. Custom security or validation

Example centralized UI handler:

```

function QuickApp:UIHandler(event)
    local element = event.elementName
    local eventType = event.eventType
    local values = event.values

    self:debug("UI Event:", element, eventType, table.concat(values

    -- Custom routing
    if element == "powerButton" then
        if eventType == "onReleased" then
            self:togglePower()
        end
    elseif element == "brightnessSlider" then
        if eventType == "onChanged" then
            self:setBrightness(values[1])
        end
    end
end
end

```

Handler responsibility

⚠ Critical: If you define these handlers, you become the Operator:

- You must implement method dispatch logic
- You must handle UI event routing
- Default automatic behavior is disabled
- If you don't handle it, nothing works

When to use custom handlers

Normal cases: Usually not needed - default handlers work fine.

Advanced cases:

- Custom dispatch logic requirements
- Single-function UI event handling
- Advanced debugging and logging
- Working around QuickAppChild limitations (next post!)

Implementation responsibility

Default behavior (automatic):

```
-- HC3 automatically calls:
quickApp:buttonPressed() -- For button named "buttonPressed"
quickApp:sliderMoved()   -- For slider named "sliderMoved"
```

Custom handler (manual):

```
function QuickApp:UIHandler(event)
  -- YOU must decide what to call based on event
  if event.elementName == "buttonPressed" then
    self:buttonPressed(event)
  elseif event.elementName == "sliderMoved" then
    self:sliderMoved(event)
  end
end
```


Summary and best practices


Key takeaways

Execution Model Understanding:  QuickApps use a single-threaded "Operator" model

 The Operator processes requests in a main loop

 All method calls are dispatched through this mechanism

Critical Timing Concepts:  Always use `setTimeout` / `setInterval` for loops

 Never use `fibaro.sleep()` in loops (blocks everything)

 QuickApps must remain responsive to incoming requests

Advanced Patterns:  Custom handlers allow override of default dispatch

 You become responsible for method/UI routing

 Useful for advanced scenarios and debugging

Practical guidelines



Loop Implementation:

```
-- ❌ Wrong - blocks everything
while true do
  work()
  fibaro.sleep(1000)
end

-- ✅ Right - non-blocking
```

```
setInterval(function()  
    work()  
end, 1000)
```

Handler Usage:

```
--  Default - automatic dispatch  
function QuickApp:myButton()  
    -- Called automatically  
end  
  
--  Custom - manual dispatch  
function QuickApp:UIHandler(event)  
    if event.elementName == "myButton" then  
        self:myButton(event)  
    end  
end
```

What's next?

In **Part 4**, we'll explore:

- **QuickAppChildren** - Creating and managing child devices with the Operator model
- **Advanced UI patterns** - Complex viewLayout structures and event handling
- **Handler override techniques** - Using custom handlers to solve QuickAppChild limitations
- **Performance optimization** - Best practices for responsive QuickApp development



Quick Reference Index

Core Concepts

Concept	Description	Key Point
Operator Model	Single-threaded request processor	Handles all incoming calls
setTimeout Pattern	Non-blocking loop implementation	Keeps QuickApp responsive
fibaro.sleep Problem	Blocking sleep function	Never use in loops
Custom Handlers	Override default dispatch	You become responsible

Essential Functions

Function	Usage	Notes
<code>setTimeout(func, ms)</code>	Schedule function execution	Non-blocking, preferred
<code>setInterval(func, ms)</code>	Recurring function execution	Built on <code>setTimeout</code>
<code>fibaro.sleep(ms)</code>	Blocking delay	⚠️ Avoid in loops
<code>quickApp.callAction(method, ...)</code>	Manual method dispatch	Built-in dispatch helper

Handler Patterns

Handler	Override	Responsibility
<code>actionHandler(action)</code>	Method calls	Custom method dispatch
<code>UIHandler(event)</code>	UI interactions	Custom UI event routing
Default	Automatic	HC3 handles dispatch

Best Practices

Practice	Why	Example
Use <code>setTimeout</code> for loops	Prevents blocking	<code>setTimeout(loop, 1000)</code>
Avoid <code>fibaro.sleep</code> in loops	Maintains responsiveness	Use <code>setInterval</code> instead
Custom handlers sparingly	Default works fine	Only for advanced cases
Keep methods responsive	Quick execution	No long-running operations

Understanding the Operator model gives you deep insight into QuickApp behavior and enables you to write more efficient, responsive, and well-architected applications.