🔧 The Anatomy of QuickApps – Part 4: QuickAppChildren deep dive

Advanced QuickApp development: understanding child device architecture.

📋 Table of Contents

⚠️ Important disclaimers

- Prerequisite knowledge: Read previous posts on Lua functions/classes and Fibaro's QuickAppChild tutorial: https://manuals.fibaro.com/knowledge-base-browse/hc3-quick-apps-managing-child-devices/
- Advanced topic: This covers complex child device management patterns.
- HC3-specific: Features and limitations specific to the Fibaro HC3 system.

Part 1: Child Device Architecture

Mother-Child Relationship

Child devices are QuickApps that "belong to" and are created by a "mother" QuickApp:

```
"Mother QA"
   |
   +--> Child 1 QA
   |
   +--> Child 2 QA
   |
   +--> Child 3 QA
```

Child Device Characteristics

Key properties:

- ✅ **Regular QuickApp devices** - appear in device lists with unique device IDs
- ✅ **No separate code files** - code resides in mother QA definition
- ✅ **Grouped display** - listed directly under mother QA (breaks deviceID sorting)
- ✅ **External representation** - typically represent external devices/services accessed via HTTP/TCP

Typical Use Cases

External device integration:

```lua
-- Examples:
-- • Philips Hue bulbs as child devices
-- • Temperature sensors from external API
-- • Multiple alarm clocks from one parent
-- • Smart switch endpoints
```

Device Type Selection

Critical consideration: Child QAs cannot have custom UIs (currently), so device type determines available UI controls.
Get available device types:

```
function QuickApp:printTypes()
  local function print(s, p)
    self:debug(p, s.type)
    table.sort(s.children, function(a, b) return a.type < b.type end
    for _, c in ipairs(s.children) do
      print(c, p .. "|")
    end
  end
  print(api.get("/devices/hierarchy"), "")
end
```

⚠️ Note: Not all device types have UIs—testing required.

Part 2: Child Device Structure

Device API access

Child devices are accessible like regular QuickApps:

```
local childDevice = api.get("/devices/77")  -- Get child device stru
```

Parent-child identification

Child device properties:

```
{
  id = 77,
  parentId = 66,  -- ← ID of mother QA
  interfaces = [
    "quickAppChild"  -- ← Identifies as child device
  ],
  -- ... other standard device properties
}
```

Get all children of a parent:

```
local children = api.get("/devices?parentId=66")  -- All children of
```

Event routing mechanism

How child events work:

1. **User interaction** - button press on child device UI
2. **System routing** - HC3 sees `parentId` and routes event to parent
3. **Parent processing** - parent finds child code and calls appropriate method

Event flow:

```
Child Device UI → HC3 System → Parent QA → Child Code
```

Part 3: QuickAppChild class system

Class Hierarchy

Important distinction: `QuickAppChild` is not a subclass of `QuickApp`:

```
-- Class hierarchy:
QuickAppBase  -- (debug, trace, updateView, updateProperty, etc.)
   |
   +--> QuickApp        -- (createChildDevice, removeChildDevice, in
   |
   +--> QuickAppChild   -- (minimal child-specific functionality)
```

Basic Child Class Definition

```
class Child1(QuickAppChild)

function Child1:__init(device)
  QuickAppChild.__init(self, device)  -- Call parent initializer
  self.x = 42                         -- Add custom field
end

function Child1:fun1(x)
  self:debug(x)
end
```

Constructor and initializer patterns

Class definition creates a constructor:

```
class Child1(QuickAppChild)  -- Creates Child1() constructor functio
```

Using the constructor:

```
local c = Child1(device)  -- Create instance
c:fun1(42)                -- Call method
```

The initializer pattern:

```
function Child1:__init(device)
  QuickAppChild.__init(self, device)  -- Must call parent first
  -- Custom initialization here
end
```

Object creation lifecycle

Internal constructor logic (conceptual):

```
function Child1(...)
  local self = createObject()
```

```
  self:__init(...)  -- Call initializer with arguments
  return self
end
```

Custom constructor with additional parameters:

```
class Child2(QuickAppChild)

function Child2:__init(device, nickName)
  QuickAppChild.__init(self, device)
  self.nickName = nickName
end

-- Usage:
local child = Child2(device, 'Bob')
```

⚠️ Note: Additional constructor arguments are limited in practical use—better to use QuickApp variables.

Device parameter source

Where does `device` come from?

- `device` = the table structure from `api.get("/devices/childId")`
- Contains all device properties, variables, and metadata
- Used by `QuickAppChild.__init()` to populate `self.id`, `self.name`, `self.type`, etc.

Part 4: Child device creation

createChildDevice() method

Standard creation pattern:

```
local child = self:createChildDevice({
    name = "My child",
    type = "com.fibaro.binarySwitch",
  }, Child2)
```

Device registration process

Internal implementation (conceptual):

```
function QuickApp:createChildDevice(options, constructor)
  -- 1. Create actual device in HC3 system
  local device = api.post("/plugins/createChildDevice", options)

  -- 2. Create Lua object instance
  local child = constructor(device)

  -- 3. Register in parent's child table
  self.childDevices[device.id] = child
```

```
  -- 4. Set parent reference
  child.parent = self

  return child
end
```

Key process steps

1. **Device creation**: `/plugins/createChildDevice` creates real HC3 device
2. **ID assignment**: HC3 assigns next available device ID (not choosable)
3. **Object instantiation**: Constructor called with device structure
4. **Registration**: Added to `self.childDevices` table
5. **Parent linking**: `child.parent = self` for reverse access

Parent field assignment

Accessing parent from child:

```
function Child2:test(x)
  self.parent:updateView('label', 'text', x)  -- Call parent method
end
```

⚠️ Timing warning: `self.parent` is not available during `__init()`:

```
function Child2:__init(device)
  QuickAppChild.__init(self, device)
  -- self.parent is NIL here!
  -- It's set AFTER __init() completes
end
```

Child onInit() alternative

Two initialization options:

Option 1: Use `__init()` (recommended):

```
function Child2:__init(device)
  QuickAppChild.__init(self, device)
  -- Do initialization here (parent not available)
end
```

Option 2: Use `onInit()` (parent available):

```
function Child2:onInit()
  -- self.parent is available here
  -- Called by QuickAppChild.__init()
end
```

Personal preference: Use `__init()` since it's required anyway.

Part 5: Child device loading

The two-task challenge

Your QuickApp must handle:

1. **Create new children** when needed
2. **Load existing children** at startup

Startup initialization

The persistence problem: When the QA restarts, child devices already exist in HC3 but Lua objects need recreation.

Required steps:

1. Find existing child devices
2. Create corresponding QuickAppChild objects
3. Associate objects with devices
4. Check for missing children and create if needed

initChildDevices() function

Standard Fibaro function:

```
function QuickApp:initChildDevices(map)
  -- map = {["deviceType"] = ConstructorClass}
end
```

Example usage:

```
function QuickApp:onInit()
  self:initChildDevices({
    ["com.fibaro.binarySwitch"] = Child1,
    ["com.fibaro.binarySensor"] = Child2
  })
end
```

Internal implementation (conceptual):

```
function QuickApp:initChildDevices(map)
  local devices = api.get("/devices?parentId=" .. self.id)

  for _, d in ipairs(devices) do
    local constructor = map[d.type]
    if constructor then
      local child = constructor(d)
      self.childDevices[d.id] = child
      child.parent = self
    end
  end
end
```

Limitations of the standard function

Problems:

- ❌ **One type, one constructor** - can't have multiple classes for same type

- ❌ **Type-based only** - limited flexibility in class selection
- ❌ **No return value** - can't count loaded children

Missing device detection

Typical startup pattern:

```
function QuickApp:onInit()
  self:initChildDevices({...})

  -- Count children
  local childCount = 0
  for _, _ in pairs(self.childDevices) do
    childCount = childCount + 1
  end

  if childCount < expectedChildren then
    self:createMissingChildren()
  end
end
```

Why children might be missing:

- User deleted child device in UI
- Incomplete previous initialization
- System error during creation

Part 6: Advanced child management

Custom createChild() helper

Enhanced creation function:

```
function QuickApp:createChild(name, typ, className, variables, prope
  properties = properties or {}
  interfaces = interfaces or {}
  properties.quickAppVariables = properties.quickAppVariables or {}

  -- Add variables as quickAppVariables
  for n, v in pairs(variables or {}) do
    table.insert(properties.quickAppVariables, 1, {name = n, value =
  end

  -- Store class name for later loading
  table.insert(properties.quickAppVariables, 1, {name = 'className',

  local child = self:createChildDevice({
      name = name,
      type = typ,
      initialProperties = properties,
      initialInterfaces = interfaces
    },
```

```
    _G[className]  -- Get constructor from global namespace
  )

  return child
end
```

Usage example:

```
self:createChild("myChild", "com.fibaro.binarySensor", "Child", {ext
```

Benefits:

- ✅ **Class name storage** - stored as quickAppVariable for loading
- ✅ **Variable initialization** - external_ID and other data stored
- ✅ **Type flexibility** - same type can use different classes

Enhanced loadChildren() method

Improved loading function:

```
function QuickApp:loadChildren()
  local cdevs = api.get("/devices?parentId=" .. self.id) or {}
  local n = 0

  -- Disable default initChildDevices
  function self:initChildDevices() end

  for _, child in ipairs(cdevs) do
    local className = "QuickAppChild"  -- Default

    -- Find stored class name
    for _, v in ipairs(child.properties.quickAppVariables or {}) do
      if v.name == "className" then
        className = v.value
        break
      end
    end

    -- Create child object
    local constructor = _G[className]
    local childObject = constructor and constructor(child) or QuickA

    -- Register child
    self.childDevices[child.id] = childObject
    childObject.parent = self
    n = n + 1
  end

  return n  -- Return count of loaded children
end
```

Key improvements:

- ✅ **Class name retrieval** - reads stored className from quickAppVariables
- ✅ **Flexible construction** - uses stored class or falls back to QuickAppChild
- ✅ **Count return** - allows comparison with expected children
- ✅ **Default override** - prevents double initialization

Device mapping strategies

The external device mapping problem:

- Need to map child device IDs ↔ external device IDs
- Example: HC3 child ID 77 ↔ Hue device ID 3

Storage during creation:

```
self:createChild("Hue Light 3", "com.fibaro.binarySwitch", "HueChild
```

Retrieval during initialization:

```
class HueChild(QuickAppChild)

function HueChild:__init(device)
  QuickAppChild.__init(self, device)
  self.hueID = self:getVariable("hueID")  -- More accessible than qu
end
```

Forward mapping (childID → hueID):

```
local hueID = self.childDevices[childID].hueID
```

Reverse mapping (hueID → childID):

```
-- Option 1: Search (inefficient)
function hueID2childID(hueID)
   for id, c in pairs(self.childDevices) do
     if c.hueID == hueID then return id end
   end
end

-- Option 2: Maintain reverse table (efficient)
local hueIDmap = {}  -- hueID → childID mapping

-- Update during child initialization
hueIDmap[child.hueID] = child.id
```

Child deletion handling

The deletion problem: When users delete children via the UI, `self.childDevices` updates but custom mappings don't.

Solution: Override `removeChildDevice`:

```
do
  local orgRemoveChildDevice = self.removeChildDevice

  function self:removeChildDevice(id)
    -- Update custom mappings
    if self.childDevices[id] then
      hueIDmap[self.childDevices[id].hueID] = nil
    end

    -- Call original function
    orgRemoveChildDevice(self, id)
  end
end
```

Monitoring alternatives:

- Listen to `/refreshStates` for `DeviceRemovedEvent`
- Install custom `onAction` handler
- Patch `removeChildDevice()` (shown above)

Part 7: Communication patterns

Polling strategies

Two main approaches:

Centralized polling (recommended)

Mother QA controls all polling:

```
-- Main QA loop
local function loop()
  for external_ID, QA_ID in pairs(device_map) do
    fetchData(external_ID, function(data)
      fibaro.call(QA_ID, "newData", data)
    end)
  end
  setTimeout(loop, pollInterval)
end
```

Benefits:

- ✅ **Controlled frequency** - single poll rate for all children
- ✅ **Efficient API usage** - batch requests possible
- ✅ **Error handling** - centralized error management

Distributed polling (not recommended)

Each child polls independently:

```
function Child:poll()
  local function loop()
```

```
      fetchData(self.external_ID, function(data)
        self:newData(data)
      end)
      setTimeout(loop, self.pollInterval)
    end
    loop()
  end
```

Problems:

- ❌ **Frequency control** - difficult to manage multiple poll rates
- ❌ **Resource usage** - many concurrent requests
- ❌ **Coordination** - hard to synchronize updates

Optimized batch polling

For many children with expensive API calls:

```
function QuickApp:pollAllDevices()
  -- Single API call gets all device data
  local allData = fetchAllDevicesData()

  -- Distribute to children
  for _, child in pairs(self.childDevices) do
    local childData = allData[child.external_ID]
    if childData then
      child:newData(childData)
    end
  end
end
```

Event handling

Child method for data updates:

```
function Child:newData(data)
  -- Process new data
  -- Update device state
  self:updateProperty("value", data.value)
  self:updateView("label", "text", data.status)
end
```

Parent method calling:

```
function Child:test(x)
  self.parent:updateView('label', 'text', x)  -- Access parent UI
end
```

Data flow management

Variable-based communication:

```lua
-- Store external ID during creation
self:createChild("Sensor", "com.fibaro.binarySensor", "SensorChild",
  external_ID = sensorID,
  poll_interval = 30
})

-- Access in child
function SensorChild:__init(device)
  QuickAppChild.__init(self, device)
  self.external_ID = self:getVariable('external_ID')
  self.poll_interval = tonumber(self:getVariable('poll_interval')) o
end
```

Simplified main loop:

```lua
for _, child in pairs(self.childDevices) do
  child:poll()  -- Each child knows its external_ID
end
```

Summary and best practices

Key takeaways

Child Device Architecture: ✅ Children are regular QuickApps with `parentId` and `quickAppChild` interface

✅ Code resides in mother QA, events route through parent

✅ Device type determines UI (no custom UIs for children currently)

Class System: ✅ `QuickAppChild` is NOT a subclass of `QuickApp`

✅ Both inherit from `QuickAppBase`

✅ `__init()` required, `onInit()` optional

✅ `self.parent` not available during `__init()`

Lifecycle Management: ✅ Must handle both creation and loading of existing children

✅ Store class names as quickAppVariables for flexible loading

✅ Monitor child deletion to maintain custom mappings

Communication Patterns: ✅ Centralized polling preferred over distributed

✅ Use quickAppVariables for external device mapping

✅ Batch API calls when possible for efficiency

Practical guidelines

Child Creation Pattern:

```lua
-- ✅ Enhanced creation with metadata
self:createChild("Device Name", "device.type", "ClassName", {
  external_ID = externalID,
  custom_setting = value
})
```

```
-- ❌ Basic creation without metadata
self:createChildDevice({name = "Device"}, Constructor)
```

Loading Pattern:

```
-- ✅ Count and validate
function QuickApp:onInit()
  local loadedCount = self:loadChildren()
  if loadedCount < self.expectedChildren then
    self:createMissingChildren()
  end
end

-- ❌ Hope everything exists
function QuickApp:onInit()
  self:initChildDevices(map)
end
```

Polling Pattern:

```
-- ✅ Centralized control
function QuickApp:pollLoop()
  for _, child in pairs(self.childDevices) do
    self:fetchAndUpdateChild(child)
  end
  setTimeout(function() self:pollLoop() end, pollInterval)
end

-- ❌ Distributed chaos
function Child:startPolling()
  setInterval(function() self:poll() end, random_interval)
end
```

What's next?

In future posts, we may explore:

- **Non-QuickAppChild implementations** - Managing children without the class system
- **Advanced event handling** - Custom action and UI handlers for children
- **Performance optimization** - Large-scale child device management
- **Error recovery** - Robust patterns for child device failures

📚 Quick Reference Index

Core Concepts

Concept	Description	Key Point

Child Device	QuickApp owned by parent	Has `parentId`, routes events to parent
QuickAppChild Class	Base class for children	NOT subclass of QuickApp
Device Type	Determines child UI	Choose carefully - no custom UIs
External Mapping	Child ↔ external device	Store as quickAppVariables

Essential Methods

Method	Usage	Notes
`createChildDevice(opts, constructor)`	Create child device	Standard Fibaro method
`initChildDevices(map)`	Load existing children	Type → constructor mapping
`removeChildDevice(id)`	Delete child device	Updates `childDevices` table
`loadChildren()`	Enhanced loading	Uses stored class names

Class Patterns

Pattern	Code	Purpose
Class Definition	`class Child(QuickAppChild)`	Define child class
Initializer	`function Child:__init(device)`	Set up child object
Parent Access	`self.parent:method()`	Call parent methods
Variable Access	`self:getVariable('name')`	Get stored data

Best Practices

Practice	Why	Example

Store class names	Flexible loading	`{name='className', value='MyChild'}`
Use quickAppVariables	External device mapping	`{name='hueID', value='3'}`
Centralized polling	Better control	Parent manages all requests
Count loaded children	Validate completeness	`if loadedCount < expected then...`
Override removeChildDevice	Maintain mappings	Update reverse lookup tables

Understanding QuickAppChildren enables you to build sophisticated multi-device QuickApps that can represent and manage complex external systems through intuitive child device interfaces!