flua — QuickApps for the rest of us

Contents

The flua tutorial — QuickApps for the rest of us

- What you will build
- The chapters
- How to read this tutorial

01 · Install

- What you are installing
- Step 1 — Python
- Step 2 — VS Code
- Step 3 — flua
- Step 4 — the smoke test
- Step 5 — the checkup: `flua doctor`
- Step 6 — a project folder with everything set up
- Step 7 (optional but smart) — your HC3's address
- You're set

02 · Your first QuickApp

- Step 1 — create the file
- Step 2 — run it with F5
- Step 3 — break it, on purpose
- Step 4 — faster tinkering: watch mode
- The two-line takeaway

03 · QuickApp anatomy

- What a QuickApp is
- The Sunset Lamp
- Run it — and click the switch
- What did we just learn?
- Next

04 · The house comes alive

- The lamp that checks the sun
- Run it after sunset
- Fast-forward to the sunset
- What did we just learn?

05 · Make it yours

- The lamp with a control panel
- Run it and flip the switch
- What did we just learn?

06 · On the real HC3
- Before you start
- Step 1 — a dress rehearsal against your real house
- Step 2 — upload the lamp
- Step 3 — fix it, update it
- The other direction: bringing an HC3 QA home
- What did we just learn?

07 · When things go wrong
- Read the log
- Read the error
- Check before you run: `--check`
- Debug like a grown-up: mobdebug
- Shrink the problem
- Ask for help

08 · Proxy mode — the real HC3 UI
- The idea
- What you need
- Turn the lamp into a proxy
- Editing the UI on the HC3 itself
- What did we just learn?

09 · Multi-file QAs — your own libraries
- Declaring extra files
- What libraries are good for
- Deploying multi-file QAs
- The Sunset Lamp, refactored
- What did we just learn?

The flua tutorial — QuickApps for the rest of us

You bought a Fibaro HC3 to automate your home. You discovered that the really good ideas need a *QuickApp* — a small program that lives on the controller. And you have never programmed anything.

This tutorial is written for you. No programming background is assumed. You only need to know how to use a computer and be willing to try things.

What you will build

One project runs through the whole tutorial: **the Sunset Lamp** — a QuickApp that turns a light on when the sun goes down. Along the way you will:

1. install the two tools you need (VS Code and flua),
2. run your first QuickApp on your own computer,
3. learn what a QuickApp *is* — and click its buttons in a browser,
4. teach the lamp when the sun sets,
5. move it onto your real HC3,
6. and teach the lamp to check itself, so future changes can't break it silently.

Each chapter ends with a working result. If a chapter feels like it takes forever, skip to the next one — you can always come back.

The chapters

- **01 · Install** — Python, VS Code, flua, one command to check it all works.
- **02 · Your first QuickApp** — write a dozen lines, run them, see them tick.
- **03 · QuickApp anatomy** — onInit, actions, properties — and the Sunset Lamp gets its On/Off buttons.
- **04 · The house comes alive** — timers, sunset, and flua's virtual clock: a whole evening tested in a second.
- **05 · Make it yours** — your own control panel (a switch and a label) and your first quickApp variable.
- **06 · On the real HC3** — a dress rehearsal against your real house, then uploading, updating and downloading QAs.
- **07 · When things go wrong** — the log, `--check`, breakpoints, and how to ask for help.
- **08 · Proxy mode** — the real HC3 UI driving your code: mirror your QA onto the controller (and leave its HC3-edited UI alone with `noUI`).
- **09 · Multi-file QAs** — your own libraries with `--%%file`, packed and deployed as one device.
- **10 · Debugging multi-QA runs** — several QAs talking to each other, and how to keep the ensemble legible.
- **11 · Let your QuickApp test itself** — `flua-test` + `flua test`: the lamp checks its own behavior, and the suite says pass or fail in one line.

How to read this tutorial

- **Commands** look like this — type them into a terminal, then press Enter:

  ```
  flua --version
  ```

  In VS Code, open a terminal with **Terminal** → **New Terminal** in the menu.

- **Output** shown after a command is what you should actually see on your screen (we removed the colors to keep it readable). Don't worry if the version numbers differ slightly.

- **Checkpoints** mark the moments that matter:

  > ✅ **You should see** — your name in the log. If you don't, re-read the previous steps; the answer is usually one line above.

- Words in `<angle brackets>` are placeholders — replace them with *your* values (your HC3's address, your password, …).

- **PDF edition.** The whole tutorial as one PDF with a chapter index: `./scripts/build-tutorial-pdf.sh` (needs pandoc and Chrome; writes `docs/tutorial/flua-tutorial.pdf`, diagrams included).

- The examples live in the `examples/` directory of the flua repository and are executed by flua's own test suite — the code in this tutorial is the same code that is tested with every release, so it always runs.

- The diagrams are drawn with Mermaid. VS Code renders them in the markdown preview once you accept the recommended **Markdown Preview Mermaid** extension (it's in the `.vscode/extensions.json` that `--tool setup` installs); on GitHub they render on the page automatically.

One promise before we start: **you will not need to understand everything.** QuickApps are small. The whole trick is a handful of ideas — this tutorial introduces them one at a time, exactly when the lamp needs them.

01 · Install

In this chapter you install the two tools you will use from now on, and you run one command that proves everything works. Ten to twenty minutes, no programming.

What you are installing

- **Python** — the engine flua runs on. Your computer may already have it.
- **Visual Studio Code (VS Code)** — a free editor where you write QuickApps.
- **flua** — the program that *runs* QuickApps on your computer, pretending to be your HC3 so you can try everything safely before touching the real thing.

> flua never changes anything on your HC3 until you explicitly tell it to (in a later chapter). Everything in the first chapters happens entirely on your own computer.

Step 1 — Python

Open a terminal and check:

```
python3 --version
```

You want **3.11 or newer**. If you see an error or an older version, install the latest Python from python.org (Windows: tick *"Add Python to PATH"* in the installer).

> ✅ **You should see** something like `Python 3.13.2`.

Step 2 — VS Code

Download and install **Visual Studio Code** from code.visualstudio.com.

Step 3 — flua

In the terminal:

```
python3 -m pip install fibaro-flua
```

(On Windows this is usually `py -3 -m pip install fibaro-flua`.)

Check that it works:

```
flua --version
```

> ✅ **You should see** `flua 0.1.x` (or newer).

If the `flua` command is not found, it still works as a Python module — every `flua` command in this tutorial can also be written `python3 -m flua`.

Step 4 — the smoke test

One line, to prove the whole machine runs:

```
flua --api local -e 'print("Hello HC3!")'
```

```
🚀 flua 0.1.x (Lua 5.4, Python 3.x), mode offline
[04.10.2026][20:11:43][DEBUG  ][QA5000]: Hello HC3!
```

Let's unpack the two lines, because you will type them a lot:

- `--api local` means *"run on my computer, not on my HC3"* — the safe mode for experiments.
- `-e 'print(...)'` is flua's one-liner: run this tiny program and exit.

`Hello HC3!` is your first QuickApp talking. That's the entire loop you will use from now on: write, run, look, adjust.

> ✅ **You should see** `Hello HC3!` in your terminal.

Step 5 — the checkup: `flua doctor`

flua can check its own health — and yours. One command tells you whether Python, flua, your project folder, and the editor setup are all right:

```
flua doctor
```

```
flua doctor
  ✓ flua              0.1.x (Lua 5.4, Python 3.x)
  ✓ virtualenv        virtualenv at ~/sunset-lamp/.venv
  ✓ HC3 credentials   not configured — offline sim only
  ✓ HC3 connection    skipped — no HC3_URL
  ✗ LuaLS setup       no .luarc.json/.luals — run `flua --tool setup` for HC3 autocor
  ✓ VS Code run/debug .vscode/launch.json present
  ✓ VS Code extension installed (version 0.1.84)
  ✓ flua-mcp          flua-mcp on PATH
  ✓ project files     1 QA file(s)
flua doctor: 8 ok, 1 problem(s)
```

Every ✓ is something that works; every ✗ names the fix right there. (The two ✗ above disappear after the next step.) When something breaks later — and it will — this is the first command to run, and the block to paste when you ask for help.

> ✅ **You should see** `flua doctor` with mostly ✓ marks. The ✗ ones are fine for now; the next step fixes them.

Step 6 — a project folder with everything set up

flua can prepare a project folder with all the VS Code settings you need (run buttons, a debugger setup, language help):

```
flua --tool setup my-first-qa
```

```
  wrote .vscode/launch.json
  wrote .vscode/extensions.json
  wrote .luarc.json
  wrote AGENTS.md
  wrote .github/skills
done — VS Code will suggest the Python and Lua MobDebug extensions on the next open;
Read on:
  USAGE.md   https://github.com/jangabrielsson/flua/blob/main/USAGE.md
  Tutorial   https://github.com/jangabrielsson/flua/tree/main/docs/tutorial
```

(No `viewer/index.html` in the list — flua serves the viewer page itself when you run `--ui`; you'll meet it in chapter 03.) The two links at the end open in the browser with a cmd/ctrl-click — the `USAGE.md` manual for when you wonder "can flua do X", and this tutorial.

Now open the folder in VS Code — **File** → **Open Folder…** → `my-first-qa`. A popup asks you to install the recommended extensions (**Python**, **Lua MobDebug**, **Lua Language Server**, and **flua**) — accept. They give you syntax coloring, error squiggles, and later a debugger. The setup also installed a **type library** (the hidden `.luals/` folder): the language server now knows what `QuickApp`, `fibaro` and `api` look like, so from chapter 02 on you'll get autocomplete and parameter hints as you type QA code.

The setup also installed **launch configurations** — small scripts that run your QuickApps when you press `F5`. From the next chapter on, F5 is how you run things; no terminal needed.

Step 7 (optional but smart) — your HC3's address

In a later chapter you will upload the lamp to your real HC3. flua finds it through a small settings file, and setting it up now takes one minute.

In your project folder, create a file named `.env` (in VS Code: **File** → **New File…** → `.env`):

```
HC3_URL=http://192.168.1.10/
HC3_USER=admin
HC3_PASSWORD=your-hc3-password
```

The address is the one you use to open your HC3 in a browser — the same `admin` user and password. This file stays on your computer; flua reads it and sends the credentials only to your own HC3.

(If you prefer not to, skip it — the next chapters run entirely on your computer and don't need it.)

You're set

Everything is installed. Next: your first QuickApp — seven lines that tick.

02 · Your first QuickApp

In this chapter you write a real QuickApp — a dozen lines — and run it. It will count to three, tell you it's done, and stop. You don't need to understand every line yet; chapter 03 explains them. Right now the goal is simpler: **write, press F5, see it live.**

Step 1 — create the file

In VS Code, open your project folder (`my-first-qa` from chapter 01) and create a new file: **File → New File…**, name it `hello.lua`, and type:

```
--%%name:hello
print("Hello from my first QuickApp")

local n = 0
setInterval(function()
  n = n + 1
  print("tick", n)
  if n >= 3 then
    print("done")
    os.exit()
  end
end, 400)
```

A few first impressions, without the theory:

- Lines starting with `--` are *comments* — notes for humans. The two `--%%` lines at the top are the exception: those are notes for **flua** (`--%%name:hello` gives the QuickApp a name).
- `print(...)` writes a line to the log.
- `setInterval(function() … end, 400)` means: *run this little block every 400 milliseconds.* That's a **timer** — the heart of every QuickApp.
- The `if` counts to three: after the third tick the QuickApp says `done` and `os.exit()` ends it — and stops its timer with it.

> `os.exit()` ends the QuickApp that called it. That's the same name real HC3 QuickApps use (there, every QuickApp is its own program — ending it is as simple as leaving a room). In these first experiments it's also what lets flua finish and hand you back the terminal — a QuickApp that never exits just keeps running.

Step 2 — run it with F5

Make sure `hello.lua` is the tab you're looking at, and press **F5** (the first time, choose **Flua: Run Current File (Terminal)** from the menu — chapter 01's setup installed it). VS Code runs the file and shows the log in its terminal panel:

```
🚀 flua 0.1.x (Lua 5.4, Python 3.x), mode offline
[04.10.2026][20:50:10][DEBUG  ][hello5000]: Hello from my first QuickApp
[04.10.2026][20:50:10][DEBUG  ][hello5000]: tick 1
[04.10.2026][20:50:11][DEBUG  ][hello5000]: tick 2
[04.10.2026][20:50:11][DEBUG  ][hello5000]: tick 3
[04.10.2026][20:50:11][DEBUG  ][hello5000]: done
```

> ✅ **You should see** your QuickApp say hello, tick three times, and stop.

That's a complete QuickApp, running. The `hello5000` at the start of each line is the QuickApp's *tag* — its name and its id. The `5000` is the id flua assigned it; on your real HC3 it would get the controller's next free id.

What did F5 do? The launch configuration ran `flua hello.lua` for you — and because your QuickApp doesn't declare a mode, flua ran it in its **default world: the simulated HC3, on your computer** (online is opt-in; chapter 06 shows how). It gave your QuickApp a home, started it, kept time for it, and stopped when the program ended. Your real HC3 was never involved — that's the whole point: experiment safely, then go live later.

> If F5 complains that `flua` was not found: VS Code on some systems doesn't inherit the shell's PATH when started from the dock. Chapter 01's setup normally prevents this — it writes the **absolute path** to flua into the launch configurations. If you wrote the configuration by hand (or reinstalled flua elsewhere), either re-run `flua --tool setup`, or restart VS Code from the terminal once — on macOS run `open -a "Visual Studio Code"` in a terminal, on Windows/Linux start `code` from one. If it says *No module named 'flua'* instead, VS Code is using a Python that doesn't have flua: open the Command Palette (Ctrl/Cmd+Shift+P), choose **Python: Select Interpreter**, and pick the one you installed flua into (chapter 01). You can always run the command by hand instead: open a terminal with **Terminal** → **New Terminal** and type `flua hello.lua` — every F5 run in this tutorial is exactly that command.

Step 3 — break it, on purpose

Programmers don't get code right the first time. They get it *wrong fast* and let the computer tell them where. Try it: change `local n = 0` to `local n = 0 + nn` (a name that doesn't exist), save, and press **F5** again:

```
Error:  hello.lua:4: attempt to perform arithmetic on a nil value (global 'nn')
```

flua tells you the file, the line (4), and the problem: it tried to do arithmetic on `nn`, which is nothing. ("Bootstrap" is flua-speak for "while starting your QuickApp" — mistakes at startup show up like this, mistakes later show up while the QuickApp is running.)

Change `0 + nn` back to `0`, save, F5. **The loop from now on is exactly this: change one thing, save, F5, read the log.**

> QuickApp files run again every time from scratch — there is no hidden state between runs. That makes experiments safe: nothing you break survives the next run.

Step 4 — faster tinkering: watch mode

For a really fast loop, flua can watch your file and restart it on every save — no F5 needed. This one is a terminal command (it has extra flags):

```
flua --api local --watch hello.lua
```

Now edit something — change `"Hello from my first QuickApp"` to `"Good evening, HC3"` — and save. The log shows the QuickApp starting over with the new text. Press `Ctrl-C` in the terminal to stop watching.

The two-line takeaway

- A QuickApp is a small program that **runs on its own**, wakes up on **timers**, and writes to a **log**.
- You develop it **on your computer** (the offline sim is the default) and only later move it to the HC3.

Next chapter: what a QuickApp *is* — and the Sunset Lamp gets its switch.

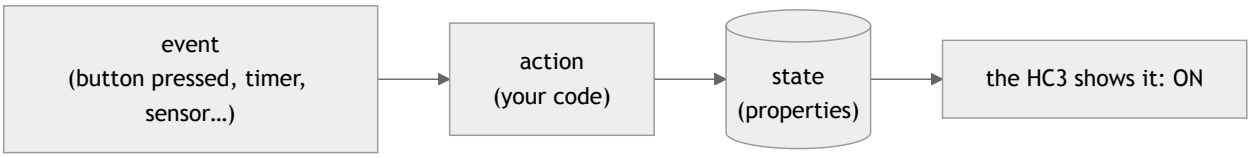
03 · QuickApp anatomy

Now the real thing. In this chapter you meet the Sunset Lamp, learn the three ideas every QuickApp is built from, and click its switch in a browser — while the whole thing still runs safely on your computer.

What a QuickApp is

A QuickApp is a small program that *lives on your HC3*. Most of the time it does nothing. Then something happens — a button is pressed, a timer fires, a sensor reports — and the QuickApp wakes up, runs a few lines of code, and goes back to sleep.

That “something happens” has a name: an **event**. And the code that runs is an **action**. That’s idea number one:



The lamp we’re about to build is tiny, but it has the full skeleton:

- it **starts** once (`onInit`),
- it **reacts** when someone switches it (`turnOn` , `turnOff`),
- it **remembers** whether it’s on — in its *properties*, which the HC3 shows in the app.

The Sunset Lamp

Create `sunset_lamp.lua` in your project (or use the repo’s copy at `examples/sunset_lamp.lua`):

```
--%%name:SunsetLamp
--%%type:com.fibaro.binarySwitch

function QuickApp:onInit()
  self:debug("Sunset lamp ready")
end

function QuickApp:turnOn()
  self:updateProperty("value", true)
  print("Lamp on")
end

function QuickApp:turnOff()
  self:updateProperty("value", false)
  print("Lamp off")
end
```

Line by line:

- `--%%name:SunsetLamp` — the QuickApp's name, as it will appear in the HC3 app. The `--%%` lines at the top are the QuickApp's *header*: instructions to flua (and later, to the HC3) about this QuickApp.
- `--%%type:com.fibaro.binarySwitch` — this QuickApp pretends to be a *binary switch*: a device that is ON or OFF. That's what gives it the On/Off buttons you're about to see.
- `function QuickApp:onInit()` — runs **once**, when the QuickApp starts. (The `self:debug(...)` writes a line to the log, tagged with the lamp's name.)
- `function QuickApp:turnOn()` — an **action**. It runs when someone turns the lamp on — a finger in the HC3 app, a scene, another QuickApp.
- `self:updateProperty("value", true)` — the lamp's state. `value` is the property of a switch: `true` = on. After this line, the HC3 app shows the lamp as ON — the QuickApp *is* the device, and this is how it tells the world.
- `turnOff()` is the mirror image: `value` becomes `false`.

> How does flua know where the header ends? It doesn't need to — only `--%%` lines are header lines, everything else is code. (There *is* an `-- ---------- EOH ----------` marker for the rare case where a QuickApp is embedded inside another file as a long string, so its header lines aren't read as the outer file's. Not needed here.)

That's the whole anatomy: **onInit** (start once), **actions** (react), **properties** (remember and show).

Run it — and click the switch

flua can serve the QuickApp's UI to a small viewer page, so you can click the buttons exactly like in the HC3 app. Press **F5** with the **Flua: Run Current File (UI)** configuration (from the menu, or **Run** → **Run Without Debugging** and pick it). It starts the lamp *and* serves the viewer:

```
🚀 flua 0.1.x (Lua 5.4, Python 3.x), mode offline
flua: UI viewer on http://127.0.0.1:8090/ — open it in a browser
[04.10.2026][20:11:17][DEBUG  ][SunsetLamp5000]: Sunset lamp ready
```

(Under the hood that configuration runs `flua --ui 8090 sunset_lamp.lua` — run it in a terminal if you ever prefer that.)

Now open **http://127.0.0.1:8090/** in a browser — flua serves the viewer page itself, from the flua package, with the address already filled in. Press **Connect**.

> ✅ **You should see** a panel titled `SunsetLamp`, with a red `FALSE` label and **Turn On / Turn Off** buttons.

You didn't write a single line about those buttons. flua added them because your QuickApp declared itself a `binarySwitch` — the HC3 does exactly the same: every switch gets On/Off controls and a state label for free. (That's why `--%%type` matters.)

Click **Turn On**.

> ✅ **You should see** two things: the label flips to a green `TRUE`, and your terminal logs `Lamp on`.

```
[04.10.2026][20:11:19][DEBUG  ][SunsetLamp5000]: Lamp on
```

Walk through what just happened, because it's the whole game:

1. You clicked **Turn On** in the viewer.
2. The viewer called your QuickApp's `turnOn` **action**.
3. Your code wrote `true` into the `value` **property**.
4. The state changed — and the label (the viewer reads the state) flipped.

On your real HC3 it would have been a finger in the Fibaro app, and the lamp would have been a light in your house. Everything else is identical.

Click **Turn Off**, then on again. Each click is one event, one action, one state change.

When you're done, stop the run with the red square (■) in VS Code's Run toolbar — the lamp runs until you stop it.

What did we just learn?

- A QuickApp **starts once** (`onInit`), **reacts to events** (actions like `turnOn`), and **remembers state** (properties like `value`).
- The **type** you choose decides what the HC3 shows: a switch gets On/Off buttons; a sensor gets a reading; a dimmer gets a slider.
- You develop against a **browser UI** on your computer; the same events come from the HC3 app later.

Next

The lamp switches when *you* want it to. Next chapter it learns to switch **by itself** — when the sun goes down. That needs a timer, and a tiny bit of astronomy the HC3 does for us.

04 · The house comes alive

The lamp switches when *you* press the button. This chapter makes it switch **by itself**: every minute it asks “is the sun down yet?” and turns on when the answer changes to yes. Along the way you meet the two ideas that make QuickApps feel alive — **timers** and **time**.

The lamp that checks the sun

Save this as `sunset_lamp_auto.lua` (or use the repo’s copy at `examples/sunset_lamp_auto.lua`):

```
--%%name:SunsetLamp
--%%type:com.fibaro.binarySwitch

function QuickApp:onInit()
  self:debug("Sunset lamp ready")
  self:checkSun()
  setInterval(function() self:checkSun() end, 60 * 1000) -- once a minute
end

function QuickApp:turnOn()
  self:updateProperty("value", true)
  print("Lamp on")
end

function QuickApp:turnOff()
  self:updateProperty("value", false)
  print("Lamp off")
end

function QuickApp:checkSun()
  local now = os.date("*t")
  local h, m = fibaro.getValue(1, "sunsetHour"):match("(%d+):(%d+)")
  local sunsetMin = tonumber(h) * 60 + tonumber(m)
  local nowMin = now.hour * 60 + now.min
  if nowMin >= sunsetMin and not self.properties.value then
    self:turnOn()
  elseif nowMin < sunsetMin and self.properties.value then
    self:turnOff()
  end
end
```

Four new ideas, each worth a moment:

1. Timers. `setInterval(function() … end, 60 * 1000)` runs the block every 60 000 milliseconds — one minute. This is how a QuickApp "does nothing, most of the time": it's just waiting for its timers.

2. The HC3 knows the sun. `fibaro.getValue(1, "sunsetHour")` — device number **1 is the HC3 itself**, and the HC3 can answer questions about your house: when the sun sets (`"18:13"`), when it rises, your location. Your QuickApp asks the controller instead of doing astronomy.

3. The clock. `os.date("*t")` returns the current date and time as a small table: `now.hour`, `now.min`, … The lamp turns both times into "minutes since midnight" (`hour * 60 + min`) so it can compare them with a plain `>=`.

4. Only change things when they must change. The two `if`s make sure the lamp only acts when the state is wrong — otherwise it would call `turnOn` every single minute, forever. QuickApps should be gentle like this; the HC3 logs everything.

The two actions from last chapter are still there, unchanged — the lamp can now be switched **by the sun and by hand**, because both paths go through the same two functions.

Run it after sunset

```
flua --api local --start '2026/10/4 18:30:00' sunset_lamp_auto.lua
```

```
🚀 flua 0.1.x (Lua 5.4, Python 3.x), mode offline
[04.10.2026][18:30:00][DEBUG  ][SunsetLamp5000]: Sunset lamp ready
[04.10.2026][18:30:00][DEBUG  ][SunsetLamp5000]: Lamp on
```

> ✅ **You should see** the lamp turn itself on at boot — 18:30 is after sunset (18:13), so `checkSun` finds the lamp off and switches it on.

Press `Ctrl-C` to stop it. (Without `Ctrl-C` the lamp would run until you stop it — it's waiting for its next minute, after all.)

What's `--start`? flua keeps its own clock — the **virtual clock** — separate from your computer's. `--start` sets it: the QuickApp above truly believes it is October 4th 2026, 18:30, and every log line is stamped with *that* time. That's how you test winter darkness in July, New Year's Eve in March, or — as we're about to do — a whole evening in seconds.

Fast-forward to the sunset

Here's the magic for the impatient. Start the lamp at 18:00, **before** sunset, and let flua fast-forward time for you:

```
flua --api local --start '2026/10/4 18:00:00' --instant --max-hours 1 sunset_lamp_au
```

```
🚀 flua 0.1.x (Lua 5.4, Python 3.x), mode offline
[04.10.2026][18:00:00][DEBUG  ][SunsetLamp5000]: Sunset lamp ready
[04.10.2026][18:13:00][DEBUG  ][SunsetLamp5000]: Lamp on
flua: virtual time limit reached (1.0h)
```

Read the timestamps: the lamp started at 18:00 — correctly decided *not* to turn on — and then time **jumped minute by minute** until 18:13, when it turned itself on at exactly the sunset the HC3 had calculated. A whole evening of lamp logic, tested in under a second.

The two flags:

- `--instant` — **deadline-ordered fast-forward**: the clock jumps to each timer's deadline before its callback runs, one at a time. A one-minute interval becomes "check the sun, jump a minute, check again…" — and `os.time()` inside the callbacks is correct at every step.
- `--max-hours 1` — a safety leash: stop after 1 virtual hour. Without it, an instant lamp would check minutes until the end of time (instantly).

There's also `--speed 60` for "60 times faster than real life" — same idea, but you watch it run — and `--step`, the same deadline-ordered discipline under its plua `speedTime` name. `--instant` and `--step` are equivalent for the QA (`--instant` used to collapse the whole timer heap into one pass, which broke `os.time()`-relative scheduling; that is fixed) — either one suits event engines that compute their next delay from `os.time()`.

> ✅ **You should see** `Lamp on` stamped `18:13:00` — the sunset from the `sunsetHour` calculation, to the minute.

The same knobs as directives — so F5 works too

Every flag from this chapter can live in the QuickApp's **header** instead, so an F5 run (chapter 02's launch configurations pass no flags) behaves the same as the terminal command:

```
--%%time:2026/10/4 18:30:00   -- start the virtual clock at 18:30
--%%speed:60                  -- run 60x faster (optional)
--%%name:SunsetLamp
--%%type:com.fibaro.binarySwitch
```

```
🚀 flua 0.1.x (Lua 5.4, Python 3.x), mode offline
[04.10.2026][18:30:01][DEBUG  ][dtime5000]: NOW 18:30
```

The directives mirror the flags: `--%%time:…` is `--start`, `--%%speed:60` is `--speed`, `--%%instant:true` is `--instant`, `--%%step:true` is `--step`, and `--%%maxhours:1` is `--max-hours`. A terminal flag beats a directive, so the same file can pin the time in its header *and* be overridden from the command line when you want a different date.

What did we just learn?

- **Timers** are how a QuickApp wakes itself up: `setInterval(fn, ms)`.
- **The HC3 is a device too** — device 1 — and it answers questions like "when is sunset?" through `fibaro.getValue(1, …)`.
- **flua's virtual clock** lets you test *any* time instantly: `--start` sets it, `--instant` / `--step` / `--speed` fast-forward it, `--max-hours` bounds it.

Next: the lamp grows a control panel of its own — a switch that turns the automatic mode on and off, right next to the On/Off buttons.

05 · Make it yours

Until now the lamp's interface was the two buttons the HC3 gives every switch. This chapter adds **your own** control panel: a switch that turns the automatic mode on and off, and a label that says what the lamp is doing. You'll also give the lamp its first **quickApp variable** — a value you can change later from the HC3 app without touching the code.

The lamp with a control panel

Save this as `sunset_lamp_ui.lua` (or use the repo's copy at `examples/sunset_lamp_ui.lua`):

```
--%%name:SunsetLamp
--%%type:com.fibaro.binarySwitch
--%%var:auto=true
--%%var:wattage=60

--%%u:{label="status",text="Automatic mode"}
--%%u:{switch="automatic",text="Automatic",value="true",onReleased="toggleAuto"}

function QuickApp:onInit()
  self:debug("Sunset lamp ready,", self:getVariable("wattage"), "W")
  self:checkSun()
  setInterval(function() self:checkSun() end, 60 * 1000)
end

function QuickApp:turnOn()
  self:updateProperty("value", true)
  print("Lamp on")
end

function QuickApp:turnOff()
  self:updateProperty("value", false)
  print("Lamp off")
end

function QuickApp:toggleAuto(e)
  local auto = e.values and e.values[1] == "true"
  self:setVariable("auto", auto)
  self:updateView("status", "text", auto and "Automatic mode" or "Manual mode")
  if auto then self:checkSun() end
end

function QuickApp:checkSun()
  if not self:getVariable("auto") then return end -- manual mode: hands off
  local now = os.date("*t")
  local h, m = fibaro.getValue(1, "sunsetHour"):match("(%d+):(%d+)")
  local sunsetMin = tonumber(h) * 60 + tonumber(m)
  local nowMin = now.hour * 60 + now.min
  if nowMin >= sunsetMin and not self.properties.value then
    self:turnOn()
  elseif nowMin < sunsetMin and self.properties.value then
    self:turnOff()
  end
end
```

Your own UI: the `--%%u` lines

The two new header lines describe the panel:

```
--%%u:{label="status",text="Automatic mode"}
--%%u:{switch="automatic",text="Automatic",value="true",onReleased="toggleAuto"}
```

Each `--%%u` line is one row of your QuickApp's own UI:

- a **label** named `status`, showing "Automatic mode";
- a **switch** named `automatic`, starting ON (`value="true"`). When someone flips it, the QuickApp's `toggleAuto` action runs (`onReleased`).

That's the whole recipe for HC3 interfaces: rows of `label`, `button`, `switch`, `slider`, `select` — each with a name and the action to call. The HC3 app renders the same rows on the real controller later.

Reacting: `toggleAuto`

```
function QuickApp:toggleAuto(e)
  local auto = e.values and e.values[1] == "true"
  self:setVariable("auto", auto)
  self:updateView("status", "text", auto and "Automatic mode" or "Manual mode")
  if auto then self:checkSun() end
end
```

Two new tricks:

- **UI actions receive an event.** `e.values[1]` is what the user did — a switch delivers `"true"` or `"false"`. The lamp stores it in a **variable** named `auto` (see the next section for why a variable and not a property).
- **`self:updateView(...)`** changes a piece of your UI: the `status` label now reads "Manual mode" — or "Automatic mode" again when switched back. This is how QuickApps talk to their own panels.

And the guard at the top of `checkSun`:

```
if not self:getVariable("auto") then return end
```

In manual mode the lamp leaves the light alone — the user has taken over. `return` means "stop this function right here".

Your own icon

A QA's tile icon is the `deviceIcon` property — an icon **id** from the controller's library. While you develop, preview a custom icon set straight from a folder: put the icons next to the QA in the upload

naming (`icon.png` , `icon0.png` + `icon100.png` for a binarySwitch, …) and point at them:

```
--%%icons:icons
```

flua then uses the QA's own id as its `deviceIcon` and the web view renders and switches the folder's icons with `value` / `state` — no HC3, no upload (`examples/local-icons.lua` is the runnable demo). The directive is preview-only and never travels into a packed `.fqa` ; production icons come from the controller (chapter 06).

QuickApp variables: `--%%var`

Both header lines starting with `--%%var` give the lamp **variables** — named values that live *with* the QuickApp:

```
--%%var:auto=true
--%%var:wattage=60
```

- `wattage` is a value you might tweak later — on the real HC3 you can edit variables in the app (device settings) without changing any code.
- `auto` is the lamp's **memory**: whether the automatic mode is on. `toggleAuto` writes it with `self:setVariable("auto", …)` and `checkSun` reads it with `self:getVariable("auto")` .

Why a variable and not a property? A QuickApp's properties are **fixed by its type**: a `binarySwitch` has `value` , and that's the schema — you can't invent an `auto` property for it. (`self:updateProperty("value", …)` from chapter 03 works because `value` is in the schema.) Variables, on the other hand, are free-form, and — the real reason `auto` is one — they **survive restarts**: when the HC3 (or flua) restarts the QuickApp, the variables keep their values, so the user's switch choice isn't lost. The `--%%var:auto=true` line only sets the *starting* value, the first time.

If a value does **not** need to survive a restart, don't give it a variable — a plain Lua variable in the file (or a field on `self` , like `self.lastCheck = os.time()`) is simpler and just as good. Variables earn their keep exactly when the value must outlive the QuickApp's run.

Run it and flip the switch

Press **F5** with the **Flua: Run Current File (UI)** configuration, as in chapter 03, and open **http://127.0.0.1:8090/**.

```
🚀 flua 0.1.x (Lua 5.4, Python 3.x), mode offline
flua: UI viewer on http://127.0.0.1:8090/ — open it in a browser
[04.10.2026][21:00:00][DEBUG  ][SunsetLamp5000]: Sunset lamp ready, 60 W
[04.10.2026][21:00:00][DEBUG  ][SunsetLamp5000]: Lamp on
```

If the lamp *didn't* turn on: it's daytime where your virtual clock thinks it is (F5 starts at the real time, like chapter 04). Pin the sun with chapter 04's trick instead:

```
flua --api local --ui 8090 --start '2026/10/4 18:30:00' sunset_lamp_ui.lua
```

Open the viewer (chapter 03) and press **Connect**.

> ✅ **You should see** the panel from before — On/Off and the state label — and *below* it, your own rows: the status label and the **Automatic** switch, sitting ON.

Now flip **Automatic** off.

> ✅ **You should see** the status label change to `Manual mode` — your `toggleAuto` action ran, stored `auto=false`, and updated the label.

And because it's past sunset in the virtual world, flip the lamp off and on with **Turn Off / Turn On** — manual mode means the lamp won't fight you: `checkSun` returns immediately and leaves the light exactly where you put it. Flip **Automatic** back on and watch it take over again.

Press the red square (■) in VS Code's Run toolbar when you're done.

What did we just learn?

- `--%%u` **rows** build your QuickApp's own UI — labels, buttons, switches, sliders, selects — each calling one of your actions.
- **UI actions receive the event** (`e.values`), so they know what the user did.
- `updateView` lets the QuickApp change its own UI live.
- **QuickApp variables** (`--%%var` , `getVariable` / `setVariable`) are free-form values that live with the device — tweakable settings, and state that survives restarts (a QA's properties are fixed by its type).

Next: the lamp has been living in a simulation this whole time. Time to move it into your real HC3.

06 · On the real HC3

The lamp has lived its whole life in a simulation on your computer. This chapter moves it into your real HC3 — the same file, unchanged. That's the flua promise: **what you tested at home is what runs on the controller.**

Before you start

Three small things, all done or known from earlier chapters:

- Your HC3 and your computer are on the same network.
- The `.env` file from chapter 01 (step 6) holds the HC3's address and your credentials.
- You know which light is *safe to play with* — pick a real lamp you can watch, and make sure turning it on and off is harmless (nobody is in the shower on its circuit).

> A note on honesty: everything in this chapter up to the upload is exactly what flua does; the upload commands themselves talk to *your* controller, which this tutorial can't rehearse for you. They are the tool's documented behavior, and they print what they do. If anything looks different, the next chapter teaches you how to read the situation.

Step 1 — a dress rehearsal against your real house

So far the lamp has run in the simulated world (offline is flua's default). For the dress rehearsal, tell the lamp to go **online** with a mode directive in its header — `--%%mode:online` — or pass `--api remote` on the command line. Your QuickApp still runs on your computer, but every question it asks the HC3 — `fibaro.getValue(1, "sunsetHour")`, `api.get(...)`, everything — is answered by your **real** controller:

```
flua --api remote --start '2026/10/4 18:30:00' sunset_lamp_ui.lua
```

```
🚀 flua 0.1.x (Lua 5.4, Python 3.x), mode online
[04.10.2026][18:30:00][DEBUG  ][SunsetLamp5000]: Sunset lamp ready, 60 W
```

The sunset hour now comes from your actual HC3 and location, not the simulation's Stockholm default. Nothing is deployed, nothing is changed on the controller — the lamp is just *consulting* it. Press `Ctrl-C` when you've seen enough.

(If flua refuses to start and complains about credentials, your `.env` is missing or the address/password is wrong — flua exits on the first failed login attempt on purpose, because the HC3 locks itself after four.)

Step 2 — upload the lamp

One command puts the lamp onto the controller:

```
flua --tool uploadQA sunset_lamp_ui.lua
```

```
uploaded SunsetLamp — HC3 device id 123
```

(`123` will be *your* HC3's next free device id.) The tool packages the QA — your `--%%` header becomes the device's name, type, UI and variables — and creates the device on the HC3. Open the Fibaro app on your phone: the SunsetLamp is there, a real switch with your **Automatic** toggle below the On/Off buttons.

The first real test: tap **Turn On** in the app. The light comes on. That's your action running on the HC3, exactly as it ran in the viewer. Welcome to home automation.

Pack first, or verify after. `uploadQA` packages on the fly; when you distribute the QA as a `.fqa`, `flua --tool pack sunset_lamp_ui.lua --verify` proves the artifact came out right (inventory, variables, placeholder and leak scan) — and `flua --tool inspectFQA sunset_lamp_ui.fqa --json` re-checks any package later, machines included.

Custom icons. The HC3 ships the standard icon library; your own icons are `User<id>` sets, managed in the browser with `flua --tool icons ui` (list, preview, upload per device type, delete). A QA can even install its own set on first boot — `examples/iconInstall.lua` is the complete recipe (download, deduplicate by content, upload once, reinstall-safe).

Step 3 — fix it, update it

Found something to improve? Edit the file, then update the QA in place (the device id is the one uploadQA printed):

```
flua --tool updateQA 123 sunset_lamp_ui.lua
```

If you didn't note the id: `flua --tool downloadQA` fetches a QA into a project folder whose file names carry the id —

```
flua --tool downloadQA 123 -d lamp/
# lamp/SunsetLamp_main_123.lua  (the id travels with the file name)
flua --tool updateQA lamp/SunsetLamp_main_123.lua   # updates QA 123
```

so a downloaded QA is always re-uploadable, even weeks later.

New QAs can land directly in a room:

```
flua --tool uploadQA --room 5 sunset_lamp_ui.lua
```

(`5` is the room's id — visible in the HC3 app's room settings.)

The other direction: bringing an HC3 QA home

The HC3 app can export any QuickApp as an `.fqa` file. flua can unpack that file into a normal flua project — so QAs you wrote on the controller can live on your computer too:

```
flua unpack myqa.fqa -d myqa-project/
```

```
unpacked to myqa-project/ (main: myqa-project/<QA-name>_main.lua)
```

```
flua export sunset_lamp_ui.lua -o lamp.fqa
```

```
exported 1 files to lamp.fqa
```

`unpack` regenerates the `--%%` header from the package (name, type, UI rows, variables), so you can read exactly how the device was configured — and `export` produces a package another HC3 owner could install from the app. Both directions are open.

What did we just learn?

- The **same file** runs in the simulation, consults your real HC3, and finally lives **on** it — flua never rewrites your code between worlds.
- `--tool uploadQA` deploys; `updateQA` updates; `downloadQA` / `unpack` / `export` move QAs in both directions.

Next chapter: when it doesn't go as planned — reading the log, checking your work before running it, and debugging like a grown-up.

07 · When things go wrong

Something will go wrong. That's not bad luck — it's the normal state of programming. The difference between people who give up and people with working smart homes is mostly *how they read the situation*. This chapter is that skill: the log, the checker, the debugger, and where to ask.

Read the log

Every QuickApp writes a log, and flua shows it in the terminal. The lamp already logs with `print(...)` — and every line has a shape worth reading:

```
[04.10.2026][18:30:00][DEBUG  ][SunsetLamp5000]: Lamp on
```

- `04.10.2026 18:30:00` — *when* it happened (virtual time — remember `--start` from chapter 04).
- `DEBUG` — the log **level**, how loud the line is.
- `SunsetLamp5000` — *which* QuickApp said it (name + id).
- `Lamp on` — the message.

There are four levels, in order of loudness:

```
[TRACE  ]   quiet details, mostly for your future self
[DEBUG  ]   the normal level — print(...) writes here
[WARNING]   "this is odd, but I'll keep going"
[ERROR  ]   "something failed"
```

`print("Lamp on")` is shorthand for `fibaro.debug` with the QA's own tag — and you can pick a level explicitly: `fibaro.trace`, `fibaro.debug`, `fibaro.warning`, `fibaro.error`. Most QuickApps live on `debug` and reach for `warning` / `error` when something genuinely failed. When in doubt, log *more* — you can delete lines later; you can't read a log you never wrote.

Read the error

Errors come in two flavors. You've met both:

At startup (the file never really ran):

```
Error:  hello.lua:4: attempt to perform arithmetic on a nil value (global 'nn')
```

flua names the **file**, the **line**, and the problem — here, arithmetic on `nn`, which doesn't exist. The fix lives on exactly that line. (When the message says `bootstrap`, it means "while starting your QuickApp".)

At runtime (the QuickApp started, then something broke while it was doing its job). Same shape — file, line, reason — but the QuickApp may keep running. Watch for the last `[DEBUG]` line before the error: that's the last thing that *worked*, and usually the best clue.

A minute of disciplined reading beats an hour of guessing: read the line number, look at that line, read the reason, and only then change something.

Check before you run: `--check`

flua can check your work **without running it** — syntax, `--%%` typos, and APIs the HC3 no longer wants:

```
flua --check sunset_lamp_ui.lua
```

```
ok
```

Three useful outcomes:

- `ok` — nothing wrong that a static check can see. Good, but it doesn't *run* the code; runtime surprises are still possible.
- A syntax error (try deleting one `end`) — named file and line, exit code

    1. Exactly what a missing `end` looks like, before you waste a run:

    ```
    lamp.lua: error: syntax: lamp.lua:12: 'end' expected (to close 'function' at li
    ```

- A warning, like a misspelled directive (`--%%instnat:true`):

    ```
    lamp.lua: warning: unknown directive --%%instnat
    ```

    Warnings don't stop anything — a misspelled directive is silently ignored at runtime, which is precisely why `--check` exists.

Make `flua --check` a habit before long runs; it catches the cheap mistakes for free.

Debug like a grown-up: mobdebug

Logging gets you far. When you need to *watch the code run line by line*, flua has a debugger, and chapter 01's setup already installed the pieces: the **Lua MobDebug** extension and the launch configuration **Flua: Debug Current File (mobdebug)**. Like all the F5 configurations it runs your QA in the offline playground (`--api local`) — the same code you were already running.

The flow:

1. Open your QuickApp in VS Code and press `F5` (choose the **mobdebug** configuration the first time — it starts flua and waits).
2. Click once in the left margin of a line, next to the line number — a red dot appears. That's a **breakpoint**: "pause here".
3. The QuickApp starts and **stops at your dot**. The panel on the left shows every variable and its value.
4. Step through the code with the small toolbar (or `F10`), line by line, watching `n` count up, `sunsetMin` appear, the `if` decide.

The same launch configuration has a sibling, **Flua: Debug Current File (UI+mobdebug)** — same debugger, plus the viewer on port 8090, so you can debug *while clicking* the UI.

If no debugger answers, flua shrugs and runs the QA anyway:

```
Could not connect to localhost:8172: Connection refused
```

so a stray `--debugger` never hangs you.

Shrink the problem

The one trick that beats everything else: when something misbehaves, **make it smaller** until it works, then add pieces back.

- Two hundred lines misbehave? Copy the suspect twenty into a new file.
- A timer never fires? Replace the inside with `print("fired")`.
- A `fibaro.getValue` returns the wrong thing? Print it, and everything around it:

  ```
  self:debug("sunset hour is", fibaro.getValue(1, "sunsetHour"))
  self:debug("auto is", tostring(self.properties.auto))
  ```

Programmers call this "the smallest case that fails" — and half the time, the act of shrinking finds the bug before the bug finds you.

Ask for help

When you've read the log, checked the file, and shrunk the problem — and it's still wrong — ask. Two places work well:

- **An AI agent.** Paste the three things below into Claude Code, Copilot, DeepSeek, ChatGPT — whatever you already have open. AI assistants are surprisingly good at reading QuickApp code, and flua is built to be agent-friendly: the log lines have one fixed shape, `flua --check` reports problems with file and line, and the flua repository ships `AGENTS.md` plus Copilot instructions — so an agent working *inside* the project already knows the house rules. Let the

agent run `flua --check` and the QA itself: it can read the errors faster than you can copy them.
- **The Fibaro forum.** flua has a thread there, and the community of HC3 owners has seen most of it before.

Whatever you ask and wherever — run this first and paste the block:

```
flua doctor
```

It prints your setup as one ready-made block (which flua, which Python, whether your HC3 credentials are found and reachable, which editor pieces are in place). It never prints your passwords — only *where* it found things. “Here is my flua doctor” beats three paragraphs of setup prose, and often the ✗ lines in it *are* the answer.

A question that gets answered — by a human *or* an agent — includes three things:

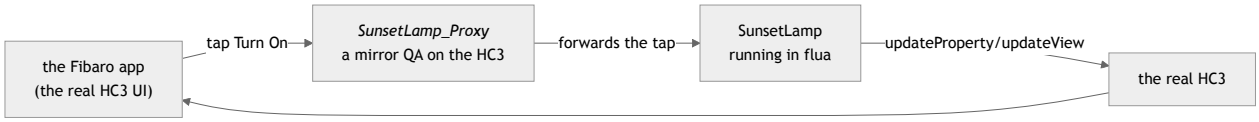
1. **What you expected** — “the lamp should turn on after 18:13”.
2. **What happened** — the exact log lines or error, copied as text.
3. **The smallest version of the code** — the file (or the suspect part), with your real password and address removed.

That’s it. You now have the whole loop: write, run, read, fix — and a lamp that has been on your real HC3 since last chapter. The rest is practice.

08 · Proxy mode — the real HC3 UI

The viewer is a faithful stand-in, but the *real* HC3 UI — the phone app, with its device cards, rooms and history — is what your family will use. Proxy mode gives you exactly that, while the QuickApp itself keeps running in flua on your computer.

The idea



flua deploys a small mirror QuickApp — `<name>_Proxy` — onto your HC3. The mirror has the same type and UI as your QA, so the app renders it like any device. When you tap a button, the proxy forwards the interaction to flua, your code runs on your computer, and every state change is pushed back so the app shows it. **You get the genuine UI, and you keep the development loop** — edit, restart, tap.

What you need

- Online mode with credentials: the `.env` from chapter 01 (proxy mode is online only — without credentials flua logs `proxy disabled` and runs the QA offline as usual).
- Your computer and the HC3 on the **same network**, and your firewall must let the controller reach your machine — flua listens for the proxy's callbacks on port 8080 (set `FLUA_PROXY_PORT` in `.env` to change it).

Turn the lamp into a proxy

Add one line to the lamp's header:

```
--%%name:SunsetLamp
--%%type:com.fibaro.binarySwitch
--%%mode:proxy
--%%var:auto=true
--%%var:wattage=60

--%%u:{label="status",text="Automatic mode"}
--%%u:{switch="automatic",text="Automatic",value="true",onReleased="toggleAuto"}
```

Run it — no `--api local` this time, the proxy needs the real HC3:

```
flua sunset_lamp_ui.lua
```

```
flua: proxy installed: 123 SunsetLamp_Proxy
flua: proxy connected — SunsetLamp_Proxy (5000) sends actions and UI events to http:/
🚀 flua 0.1.x (Lua 5.4, Python 3.x), mode proxy:5000
[05.10.2026][21:00:00][DEBUG  ][SunsetLamp5000]: Sunset lamp ready, 60 W
```

> ✅ **You should see** a new device in the Fibaro app — `SunsetLamp_Proxy`, with your custom UI. Open it and tap **Turn On**: the log on your computer shows `Lamp on`, and the app shows the lamp ON.

What just happened: the tap went from the app to the proxy to flua, ran `turnOn()` on your computer, and the state update travelled back to the HC3 — one round trip, one log line. The `5000` after `mode proxy:` is the emulator's id; the proxy's HC3 id (here `123`) is the device id in the app. They're the same QA.

Your QA now works like a real HC3 citizen: scenes and other QAs can call its actions, the app shows its state and history, and `self:updateView` / `updateProperty` / `setVariable` updates appear live in the app. Child devices (`self:createChildDevice`) are created on the HC3 as the proxy's children and mirrored back.

Press `Ctrl-C` when you're done — the proxy stays on the HC3 (it's a device now), and the next run reuses it: flua finds the existing `SunsetLamp_Proxy`, reconnects it to your computer, and refreshes its UI from your current `--%%u` directives. The QA file remains the single source of truth.

Editing the UI on the HC3 itself

The HC3 has a rudimentary UI editor, and some developers prefer shaping the proxy's interface there. To make flua **not** clobber those edits on every connect:

```
--%%mode:proxy,noUI
```

The proxy is still reused and connected, but its UI is left exactly as you edited it — flua logs `proxy UI left untouched`. Without `noUI`, every connect pushes the UI from your `--%%u` directives (the backward-compatible behavior).

> A note on honesty, as in chapter 06: this chapter's live steps are flua's documented behavior — deploying, CONNECTing and mirroring — but they talk to your controller, so rehearse with your scratch QA and watch the log the first time. Anything unexpected lands in the next chapter's toolkit.

What did we just learn?

- **Proxy mode** mirrors your QA onto the HC3 so the *real* app UI drives your code in the emulator — actions in, state out, one round trip.
- The proxy is **reused** across runs (UI refreshed from your directives), unless you declare `--%%mode:proxy,noUI` to protect HC3-side UI edits.
- Offline/sim first, proxy when you want the genuine UI — the same file, one header line apart.

Next: growing out of one file — your own libraries, shared between QAs.

09 · Multi-file QAs — your own libraries

The lamp has grown, and so will the next QuickApp. Before copy-pasting the same helper into every QA, learn the HC3's answer: a QuickApp can consist of **several files** — one main file plus your own libraries, packed and deployed together as one device.

Declaring extra files

The header directive `--%%file:path,name` adds a file. The files load **in declaration order, before the main file**, and share the same environment — a function defined in a library is simply there when the main file runs:

```
--%%name:multifile-demo
--%%file:multifile/lib.lua,lib
-- --------------- EOH ---------------
function QuickApp:onInit()
  print("helper says:", helper())
end
```

`multifile/lib.lua`:

```
function helper()
  return "loaded in order"
end
```

Run it:

```
flua --api local examples/multifile.lua
```

```
🚀 flua 0.1.x (Lua 5.4, Python 3.x), mode offline
[05.10.2026][10:33:55][DEBUG  ][multifile-demo5000]: helper says: loaded in order
```

That's the whole feature. The `,lib` part is the file's *name* on the HC3 (the path is where it lives on your disk — relative to the main file); flua reports both when you ask:

```
local files = api.get("/quickApp/"..self.id.."/files")
for _, f in ipairs(files) do print("file:", f.name) end
```

```
file: main
file: lib
```

What libraries are good for

- **Shared helpers** — parsing, formatting, time math, the `findDevice` pattern from chapter 10 — written once, used by every QA that declares the file.
- **Big QAs, organized** — the lamp's sun logic, its UI callbacks and its actions can live in separate files instead of one long scroll.
- **Versioned code** — swap one library file to upgrade a behavior across QAs.

Two ground rules:

- The files are **one program**, not several QAs — a library has no `--%%name`, no `onInit` of its own (well, its top-level code *does* run, in declaration order — but one QA, one life).
- `--%%` lines in a library file are just comments to flua — directives are read from the **main file only**.

Deploying multi-file QAs

`--tool uploadQA` packs the main file and every `--%%file` into one package (the `.fqa` format the HC3 app understands), so the whole QA — libraries included — travels as one device:

```
flua --tool uploadQA main.lua     # packs + uploads all declared files
flua --tool updateQA 123 main.lua # same, into an existing QA
```

And the other direction, `downloadQA`, writes the files out with names that carry the HC3 id — `My_QA_main_123.lua`, `My_QA_lib_123.lua` — so a downloaded QA is a working flua project you can edit and re-upload.

The Sunset Lamp, refactored

A taste of the payoff. Move the sun logic into `sun.lua`:

```
-- sun.lua — the lamp's library: everything about the sun
function sunMinutes()
  local h, m = fibaro.getValue(1, "sunsetHour"):match("(%d+):(%d+)")
  return tonumber(h) * 60 + tonumber(m)
end

function isAfterSunset()
  local now = os.date("*t")
  return now.hour * 60 + now.min >= sunMinutes()
end
```

…and the lamp's main file becomes smaller, with the library declared in the header:

```
--%%name:SunsetLamp
--%%type:com.fibaro.binarySwitch
--%%file:sun.lua,sun

function QuickApp:checkSun()
  if not self:getVariable("auto") then return end
  if isAfterSunset() and not self.properties.value then
    self:turnOn()
  elseif not isAfterSunset() and self.properties.value then
    self:turnOff()
  end
end
```

Same behavior, easier to read — and `sun.lua` is ready for the next sun-driven QA you write.

What did we just learn?

- `--%%file:path,name` adds library files; they load **before the main file**, in declaration order, sharing its environment.
- One QA, several files — directives live in the main file only.
- The tools pack and unpack the whole set as one device.

Next: the finale of this tutorial — several QAs running *together*, finding each other, and how to debug the ensemble.