Extensions: Running Your Own Code Inside a Test

Features · https://docs.functionize.com/hc/en-us/articles/43767890012311-Extensions-Running-Your-Own-Code-Inside-a-Test

An extension is a piece of your own code that a test can call mid-run. Most tests never need one. When you do need one, it is usually because a step has to do something a browser cannot: decrypt a token, call an internal API, transform a value, read a file format.

When you actually need one

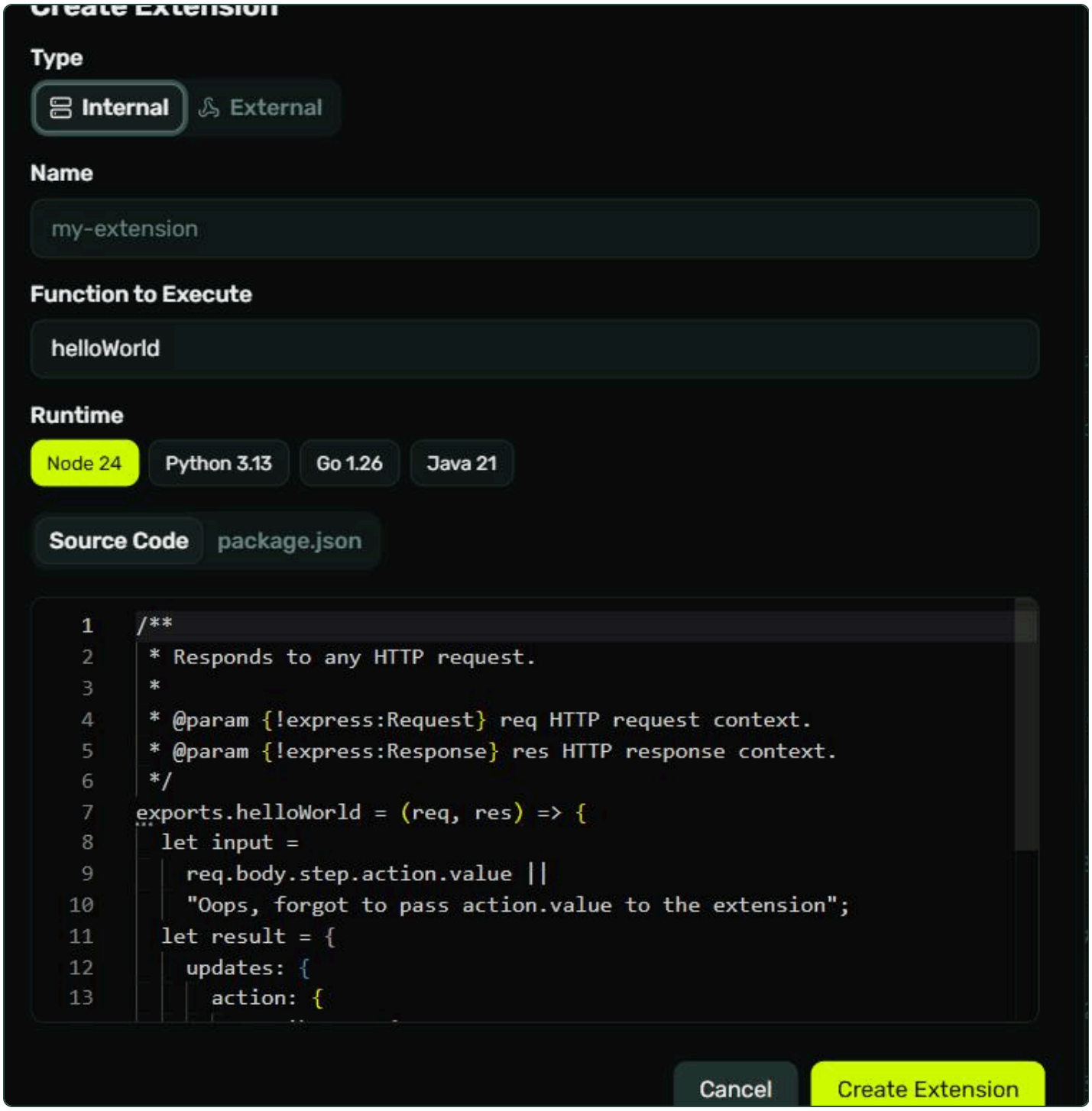
Reach for an extension when a step must:

- **Generate or derive a value** — a one-time passcode, a signed token, a hash.
- **Talk to something outside the browser** — seed data through an internal API, check a queue, read a mailbox.
- **Transform data** — reformat a date, normalize a string, parse a PDF.

Do *not* reach for one to find elements or assert on the page. Describing what you want in plain language is what Studio is for, and a custom selector hard-coded in an extension is exactly the brittleness the product exists to avoid. See Exercise 2 — Write Assertions That Prove Something for the assertions you should be writing instead.

Creating one

Open **Extensions** under **Assets** in the left sidebar and choose **Create Extension**.



The fields:

Type	**Internal** (code you write here, run by us) or **External** (pointing at something you host).
Name	How you will refer to it from a test. Use something descriptive — `lowercaseTransform` beats `test21`.
Function to Execute	The entry point. It says *Auto-detected from code*, so you can usually leave it alone.

Runtime	**Node 24**, **Python 3.13**, **Go 1.26** or **Java 21**.
Source Code	An editor, with a **package.json** tab beside it for dependencies on the Node runtime.

The shape of an extension

Studio starts you with a working skeleton rather than an empty box. On the Node runtime it is an HTTP-style handler:

```
exports.helloWorld = (req, res) => {
  let input = req.body.step.action.value
    || "Oops, forgot to pass action.value to the extension";
  let result = {
    updates: {
      action: { ... }
    }
  };
  ...
}
```

Two things to take from that:

- **Input arrives on the request**, at `req.body.step.action.value` — the value the calling step passes in.
- **You return** `updates`, which is how the extension feeds a result back into the running test rather than merely printing something.

That contract is the part worth understanding before you write anything real. The starter comment even names the common mistake: forgetting to pass `action.value` from the step, so the extension receives nothing.

Who can use it

Each extension has an **Access Level** of **Team** or **Private**. Team makes it available to everyone in your team; Private keeps it to you. The extensions list shows this in its own column, alongside the runtime and when it was last updated.

Prefer **Team** for anything genuinely useful.

An older extension may be on Node 20. The list shows the runtime each extension was built on, and long-lived ones still say `Node 20` although **Create Extension** now only offers Node 24. Nothing is wrong: an existing extension keeps the runtime it was written against. It is worth knowing before you go looking for a Node 20 option that is not there.

A private extension that three people each rewrite is worse than one shared extension with a clear name.

Extensions have their own pass/fail

The extensions list shows a status against each one. An extension that does not compile or errors on its test invocation shows as failed there — which means you can find out it is broken without discovering it through a confusing test failure at three in the morning.

Check that column occasionally. A failing shared extension is a problem for everyone using it.

Calling one from a test

Once an extension exists, a step can invoke it. In a run, that appears in the steps list as a **Customcode** action, and the step detail records that custom code was executed on the page. You will see it beside the ordinary actions in the middle pane described in Understanding a Test Run and Its Results.

The simplest way to wire one in is to ask the agent, naming the extension:

```
Add a step that calls the lowercaseTransform extension with the product name, and verify the result matches the heading.
```

A word of restraint

Every extension is code your team now maintains, in a language that may not be the language your team writes. Extension lists grow quickly and age badly: without naming discipline you end up with dozens of near-duplicates whose purpose nobody remembers, several of them failing.

Write them when you need them, name them for what they do rather than when you made them, share the useful ones at Team level, and delete the experiments.