ModulesHowTo

Sergey Ponomarev edited this page on 10 Mar · 2 revisions

Table of Contents

HowTo: Write Modules

Note: If you plan to integrate your module into LuCI, you should read the Module Reference before.

This tutorial describes how to write your own modules for the LuCI WebUI. For this tutorial we refer to your LuCI installation directory as **lucidir** (/usr/lib/lua/luci if you are working with an installed version) and assume your LuCI installation is reachable through your webserver via **/cgi-bin/luci**.

Show me the way (The dispatching process)

To write a module you need to understand the basics of the dispatching process in LuCI. LuCI uses a dispatching tree that will be built by executing the index-Function of every available controller. The CGI-environment variable **PATH_INFO** will be used as the path in this dispatching tree, e.g.: /cgi-bin/luci/foo/bar/baz will be resolved to foo.bar.baz

To register a function in the dispatching tree, you can use the **entry**-function of *luci.dispatcher*. entry takes 4 arguments (2 are optional):

```
entry(path, target, title=nil, order=nil)
```

- **path** is a table that describes the position in the dispatching tree: For example a path of {"foo", "bar", "baz"} would insert your node in foo.bar.baz.
- **target** describes the action that will be taken when a user requests the node. There are several predefined ones of which the 3 most important (call, template, cbi) are described later on on this page
- **title** defines the title that will be visible to the user in the menu (optional)
- **order** is a number with which nodes on the same level will be sorted in the menu (optional)

You can assign more attributes by manipulating the node table returned by the entry-function. A few example attributes:

- **i18n** defines which translation file should be automatically loaded when the page gets requested
- **dependent** protects plugins to be called out of their context if a parent node is missing
- **leaf** stops parsing the request at this node and goes no further in the dispatching tree
- **sysauth** requires the user to authenticate with a given system user account

It's all about names (Naming and the module file)

Now that you know the basics about dispatching, we can start writing modules. But before you have to choose the category and name of your new digital child.

We assume you want to create a new application "myapp" with a module "mymodule".

So you have to create a new subdirectory ***lucidir*/controller/myapp** with a file **mymodule.lua** with the following content:

```
module("luci.controller.myapp.mymodule", package.seeall)

function index()

end
```

The first line is required for Lua to correctly identify the module and create its scope. The index-Function will be used to register actions in the dispatching tree.

Teaching your new child (Actions)

So it is there and has a name but it has no actions.

We assume you want to reuse your module myapp.mymodule that you begun in the last step.

Actions

Reopen ***lucidir*/controller/myapp/mymodule.lua** and just add a function to it so that its content looks like this example:

```
module("luci.controller.myapp.mymodule", package.seeall)

function index()
    entry({"click", "here", "now"}, call("action_tryme"), "Click here", 10).dependent
end

function action_tryme()
    luci.http.prepare_content("text/plain")
    luci.http.write("Haha, rebooting now...")
    luci.sys.reboot()
end
```

And now type **/cgi-bin/luci/click/here/now** (**http://192.168.1.1/luci/click/here/now** on your OpenWrt system) in your browser.

You see these action functions simple have to be added to a dispatching entry.

As you might or might not know: CGI specification requires you to send a Content-Type header before you can send your content. You will find several shortcuts (like the one used above) as well as redirecting functions in the module **luci.http**

Views

If you only want to show the user a text or some interesting familiy photos it may be enough to use a HTML-template. These templates can also include some Lua code but be aware that writing whole office suites by only using these templates might be called "dirty" by other developers.

Now let's create a little template ***lucidir*/view/myapp-mymodule/helloworld.htm** with the content:

```
<%+header%>
<h1><%:Hello World%></h1>
<%+footer%>
```

and add the following line to the index-Function of your module file.

```
entry({"my", "new", "template"}, template("myapp-mymodule/helloworld"), "Hello world"
```

Now type **/cgi-bin/luci/my/new/template** (**http://192.168.1.1/luci/my/new/template** on your OpenWrt system) in your browser.

You may notice those fancy -Tags, these are template markups used by the LuCI template processor. It is always good to include header and footer at the beginning and end of a template as those create the default design and menu.

CBI models

The CBI is one of the uber coolest features of LuCI. It creates a formular based user interface and saves its contents to a specific UCI config file. You only have to describe the structure of the configuration file in a CBI model file and Luci does the rest of the work. This includes generating, parsing and validating a XHTML form and reading and writing the UCI file.

So let's be serious at least for this paragraph and create a real pratical example ***lucidir*/model/cbi/myapp-mymodule/netifaces.lua** with the following contents:

```
m = Map("network", "Network") -- We want to edit the uci config file /etc/config/netw

s = m:section(TypedSection, "interface", "Interfaces") -- Especially the "interface"-
s.addremove = true -- Allow the user to create and remove the interfaces
function s:filter(value)
    return value ~= "loopback" and value -- Don't touch loopback
end
s:depends("proto", "static") -- Only show those with "static"
s:depends("proto", "dhcp") -- or "dhcp" as protocol and leave PPPoE and PPTP alone

p = s:option(ListValue, "proto", "Protocol") -- Creates an element list (select box)
p:value("static", "static") -- Key and value pairs
p:value("dhcp", "DHCP")
p.default = "static"

s:option(Value, "ifname", "interface", "the physical interface to be used") -- This w

s:option(Value, "ipaddr", translate("IP Address")) -- Ja, das ist eine i18n-Funktion

s:option(Value, "netmask", "Netmask"):depends("proto", "static") -- You may remember

mtu = s:option(Value, "mtu", "MTU")
mtu.optional = true -- This one is very optional

dns = s:option(Value, "dns", "DNS-Server")
dns:depends("proto", "static")
dns.optional = true
function dns:validate(value) -- Now, that's nifty, eh?
    return value:match("[0-9]+%.[0-9]+%.[0-9]+%.[0-9]+") -- Returns nil if it doesn't
end

gw = s:option(Value, "gateway", "Gateway")
gw:depends("proto", "static")
gw.rmempty = true -- Remove entry if it is empty

return m -- Returns the map
```

and of course don't forget to add something like this to your module's index-Function.

```
entry({"admin", "network", "interfaces"}, cbi("myapp-mymodule/netifaces"), "Network i
```

There are many more features, see the CBI reference and the modules shipped with LuCI.

Pages 16

Find a Page...

Home
CBI
Datatypes
DevelopmentEnvironmentHowTo
Documentation
Download
i18n
Installation
JsonRpcHowTo
License
LMO
LuCI 0.10
Modules
ModulesHowTo
Templates

Show 1 more pages...