Declarative Internal DSLs in Lua
A Game-Changing Experience

Alexander Gladysh, ag@logiceditor.com
LogicEditor.com CTO, co-founder

Lua Workshop 2011
Outline

Introduction

Ad-hoc approach

A case study

The ”proper” solution

Where do we use DSLs ourselves?

Why game-changing?

Questions?
namespace:method "title"
{
    data = "here";
}

...Without sugar

_G["namespace"]:method(
    "title"
  ) ( {
    ["data"] = "here";
  })
Naïve implementation

```javascript
namespace = { };

namespace.method = function(self, name) {
  return function(data) {
    -- ...do something
    -- ...with name and data
  }
}
end
end
```
Hypothetical UI description language

```plaintext
ui:dialog "alert"
{
    ui:label "message";
    ui:button "OK"
    {
        on_click = function(self)
            self:close()
        end;
    }
}
```
function ui:label(title)
    return function(data)
        return GUI.Label:new(title, data)
    end
end

function ui:button(title)
    return function(data)
        return GUI.Button:new(title, data)
    end
end
function ui:dialog(title)
    return function(data)
        local dialog = GUI.Dialog:new(title)
        for i = 1, #data do
            dialog:add_child(data)
        end
        return dialog
    end
end
Ad-hoc approach

+ Easy to code simple stuff

But:

- Easily grows out of control
- Difficult to reuse
- Hard to handle errors
- Hard to add new output targets
Practical example: HTTP handler

api:url "/reverse"
{
    doc:description [[String reverser]]
    [[
        Takes a string and reverses it.
    ]]];
    api:input { data:string "text" };  
    api:output
    {
        data:node "result" { data:string "reversed" };  
    };
    handler = function(param)
        return { reversed = param.text:reverse() } 
    end;
}
What do we want to get from that description?

- **HTTP request handler itself**, with:
  - Input validation
  - Multi-format output serialization (JSON, XML, ...)
  - Handler code static checks (globals, ...)

- **Documentation**

- Low-level networking **client code**

- Smoke **tests**
local handler = function(checker, param)
    return
    {
        text = checker:string(param, "text");
    }
end

INPUT_LOADERS["/reverse.xml"] = handler
INPUT_LOADERS["/reverse.json"] = handler
local build_formatter = function(fmt)
    return fmt:node("nil", "result")
    {
        fmt:attribute("reversed");
    }
end

OUTPUT["/reverse.xml"] = build_formatter(
    make_xml_formatter_builder()
):commit()

OUTPUT["/reverse.json"] = build_formatter(
    make_json_formatter_builder()
):commit()
Request handler: the handler itself

-- Handler code is checked for access to illegal globals.
-- Legal globals are aliased to locals at the top.
-- Necessary require() calls are added automatically.

local handler = function(param)
    return
    {
        reversed = param.text:reverse();
    }
end

HANDLERS["/reverse.xml"] = handler;
HANDLERS["/reverse.json"] = handler;
/reverse.{xml, json}: String reverser
Takes a string and reverses it.
IN

?text=STRING

OUT
XML:

<result reversed="STRING" />

JSON:

{ "result": { "reversed": "STRING" } }
Smoke tests

test:case "reverse.xml:smoke.ok" (function()
    local reply = assert(http.GET(
        TEST_HOST .. "/reverse.xml?text=Foo")
    )
    assert(type(reply.result) == "table")
    assert(type(reply.result.reversed) == "string")
end)
Too complicated for ad-hoc solution!
The ”proper” solution?

- Should be easy to add a new target.
- Should be reusable.
- Should have nicer error reporting.
The process

- Load data.
- Validate correctness.
- Generate output.
Let's recap how our data looks like

api:url "/reverse"
{
    doc:description "[[String reverser]]"
    "[[
        Takes a string and reverses it.
    ]]]"
    api:input { data:string "text" };
    api:output
    {
        data:node "result" { data:string "reversed" };
    };
    handler = function(param)
        return { reversed = param.text:reverse() };
    end;
}"
Surprise! It’s a tree!

```javascript
{ id = "api:url", name = "/reverse";
  { id = "doc:description", name = "String reverser";
    text = "Takes a string and reverses it.";
  }
};
{ id = "api:input";
  { id = "data:string", name = "text" };
}
{ id = "api:output";
  { id = "data:node", name = "result";
    { id = "data:string", name = "reversed" };
  }
};
handler = function(param)
  return { reversed = param.text:reverse() }
end;
}
```
We need a loader that does this: (I)

```plaintext
namespace:method "title"
{
    data = "here";
}
⇒
{
    id = "namespace:method";
    name = "title";
    data = "here";
}
```
We need a loader that does this: (II)

```plaintext
namespace:method "title"
⇒
{
  id = "namespace:method";
  name = "title";
}
```
We need a loader that does this: (III)

```javascript
namespace:method {
    data = "here";
}
⇒ {
    id = "namespace:method";
    data = "here";
}
```
We need a loader that does this: (IV)

```plaintext
namespace:method "title"
[ [ text ] ]
⇒
{
    id = "namespace:method";
    name = "title";
    text = [ [ text ] ];
}
```
We need a loader that does this: 

```lua
namespace:method "title" (function()
    -- do something
end)
⇒
{
    id = "namespace:method";
    name = "title";
    handler = function()
        -- do something
    end;
}
```
...And adds some debugging info for nice error messages:

```lua
-- my_dsl.lua:
42: namespace:method "title" =>
43: { id = "namespace:method";
44:     name = "title";
45:     data = "here";
46:     file_ = "my_dsl.lua";
47:     line_ = 42;
48: }
```
Nested nodes should just... nest:

```plaintext
namespace:method "title"
{
  data = "here";
  foo:bar "baz_1";
  foo:bar "baz_2'';
}
⇒
{
  { id = "namespace:method"; name = "title"; data = "here";
    { id = "foo:bar", name = "baz_1" },
    { id = "foo:bar", name = "baz_2" }
  }
}
```
Notes on data structure:

- Use unique objects instead of string keys to avoid name clashes.
- Or you may store user-supplied ”data” in a separate key.
Metatable magic, I

_G["namespace"]:method(
   "title"
 ) ( {
   ["data"] = "here";
 })

setmetatable(
    _G, -- actually, the sandbox
    -- environment for DSL code
    MAGIC_ENV_MT
 )
Metatable magic, II

_G["namespace"]::method(  "title"
 ) ({  
    ["data"] = "here";
})

MAGIC_ENV_MT.__index = function(t, k)
    return setmetatable(  
        { },
        MAGIC_PROXY_MT
    )
end
Metatable magic, III

_G["namespace"]:method(
    "title"
) ( {
    ["data"] = "here";
})

MAGIC_PROXY_MT.__call = function(self, title)
    self.name = title
    return function(data)
        data.name = self.name
        return data
    end
end
Things are somewhat more complex: (I)

- You must detect "syntax sugar" forms (text, handler)...
  - ...just watch out for types, nothing complicated.
- You have to care for single-call forms (name-only, data-only)...
  - ...store all proxies after first call
  - and extract data from what’s left after DSL code is executed.
Things are somewhat more complex: (II)

- Error handling not shown...
  - ...it is mostly argument type validation at this stage,
  - but global environment protection aka strict mode is advisable.
- Debug info gathering not shown...
  - ...just call `debug.getinfo()` in `__call`.
- You *should* keep order of top-level nodes...
  - ...make a list of them at the "name" call stage.
Format-agnostic DSL loader

Loads DSL data to the in-memory tree.

- **Reusability:** Works for any conforming DSL without modifications.
- **Output targets:** N/A.
- **Error reporting:** Does what it can, but mostly that is behind its scope.
namespace:method "title"
  : modifier "text"
  : another { modifier_data = true }
{
  data = "here";
}

On subnodes: DSL vs plain tables, I

What is better?

```plaintext
foo:bar "name"
{
    subnode =
    {
        key = "value";
    };
}

...Or...

foo:bar "name"
{
    foo:subnode
    {
        key = "value";
    };
}
```
On subnodes: DSL vs plain tables, II

It depends on the nature of the data.

▶ If subnode is a *genuine tree node*, use `foo:bar.foo:subnode` DSL subnodes.

▶ But for *parameters* of the tree node, even when they are stored in a sub-table, use plain old `foo:bar.subnode` tables.

▶ When unsure, pick whichever is easier for tree traversal in each case.
One third done

✓ Load data.
   ▶ Validate correctness.
   ▶ Generate output.
Validation and generation

- Trading speed for convenience (but not so much).
- Traversing the tree (or rather forest) once for validation pass and once for each output target.
Tree traversal (hat tip to Metalua)

```lua
namespace:method "title" { -- 3rd
  data = "here";
  foo:bar "baz_1"; -- 1st
  foo:bar "baz_2"; -- 2nd
}
--
local walkers = { up = { }, down = { } }
walkers.down["foo:bar"] = function(walkers, node, parent)
  assert(node.name == "baz_1" or node.name == "baz_2")
end
walkers.down["namespace:method"] = function(walkers, node, parent)
  assert(node.name == "title" and #node.data > 0)
end
--
walk_tree(dsl_data, walkers)
```
Tree traversal process

- Bidirectional, depth-first: down, then up.
- If a handler for a given node.id is not found, it is considered a "do nothing" function. Traversal continues.
- If down handler returns "break" string, traversal of subtree is aborted.
- Knowing a node parent is useful.
Tree traversal hints

- Store state in `walker` object.
- Set metatables on `up` and / or `down` for extra power.
- In complex cases gather data in `down`, act in `up`.
- In even more complex cases break in `down`, and run a custom traversal on the node subtree.
walkers.up["foo:bar"] = function(walkers, node)
    walkers:ensure(
        "check condition A", predicate_A(node),
        node.file_, node.line_
    )
end

walk_tree(dsl_data, walkers)

if not walkers:good() then
    error(
        "data validation failed: ",
        .. walkers:message()
    )
end
Validation notes

▶ *Don’t skip implementing it.* Even poor validation is better than none.

▶ *But don’t overdo as well.* Depending on the nature of the language, overly strict validator may harm usability. Keep optional things optional, and be flexible (just) enough in what input you accept.

▶ *Do validation in a separate pass.* In output generation assume data to be valid and do not clutter the code with redundant checks.

▶ *Accumulate all errors before failing.* This will improve usability. But don’t forget to teach users that errors at the end of the list may be bogus.

▶ *Report full stack of wrong nodes.* From the failed node up to the root.
Almost there

✓ Load data.
✓ Validate correctness.
  ▶ Generate output.
walkers.down["namespace:method"] = function(walkers, node)
    walkers:cat
    [[<method name=]] (xml_escape(node.name)) [[>]]
end

walkers.up["foo:bar"] = function(walkers, node)
    walkers:cat
    [[<bar name=]] (xml_escape(node.name)) [[ />]]
end

walkers.up["namespace:method"] = function(walkers, node)
    walkers:cat [[</method>]]
end
function walkers.cat(walkers, v)
    walkers.buf[#walkers.buf + 1] = tostring(v)
    return walkers.cat
end

function walkers.concat(walkers)
    return table.concat(walkers)
end

walk_tree(dsl_data, walkers)

output:write(walkers:concat())
Output generation notes

- *One tree walker per target.* Otherwise make sure that your trusty old cheese grater is still sharp.
- *Use string ropes or write directly to file.* Or face GC overhead.
- *You may generate run-time objects instead of strings.* But off-line generation is much neater.
- *Think!* A lot of output generation problems are easier than they look.
Validation and output generation

...By means of data tree traversal.

- **Reusability**: High. Almost everything that you have to write is business-logic. No low-level boilerplate code is visible.
- **Output targets**: Conforming targets may be added without changing any of existing code.
- **Error reporting**: You have everything you need to provide good error reports to user.
We’re done with DSL handling

✓ Load data.
✓ Validate correctness.
✓ Generate output.
Where do we use internal DSLs ourselves?

Most prominent cases:

- A HTTP webservice API DSL (which we just discussed).
- A config file format DSL family.
- A SQL DB structure DSL.
- Visual Business Logic Editor DSL family.
A config file format DSL family: node description language

types:up "cfg:existing_path" (function(self, info, value)
    local _ =
    self:ensure_equals(
        "unexpected type", type(value), "string"
    ):good()
    and self:ensure(
        "path string must not be empty", value ~= ""
    ):good()
    and self:ensure(
        "path must exist", lfs.attributes(value)
    )
end)
A config file format DSL family: config description language

Controlled by the node description language.

cfg:node "my_tool"
{
  cfg:existing_path "data_path";
}

A config file format DSL family: the config data itself

The data itself is the usual automagic table hierarchy.

my_tool.data_path = "path/to/file.bin"
A config file format DSL family: output targets

- Data loader and validator.
- Documentation.
sql:table "countries"
{
    sql:primary_key "id";
    sql:string "title" { 256 };
    --
    sql:unique_key "title" { "title" };
}
A SQL DB structure DSL: output targets

- Initial DB schema SQL code.
- DB schema patches (semiautomated so far).
- Full-blown backoffice web-UI for data management.
- Documentation.
The Logic Editor DSL family: high-level data schema DSL

Human-friendly concepts, describing data tree structure and transformation rules:

```lang:enum "dow" { -- day of week
  "dow.mon", "dow.tue", "dow.wed", "dow.thu",
  "dow.fri", "dow.sat", "dow.sun";
render:js [[[Weekday]]] {
  { [[[Monday]]] }, { [[[Tuesday]]] }, { [[[Wednesday]]] },
  { [[[Thursday]]] }, { [[[Friday]]] }, { [[[Saturday]]] },
  { [[[Sunday]]] };
};
render:lua { -- Matching os.date() format.
  { [[[2]]] }, { [[[3]]] }, { [[[4]]] }, -- MO, TU, WE
  { [[[5]]] }, { [[[6]]] }, { [[[7]]] }, -- TH, FR, SA
  { [[[1]]] }; -- SU
};
```
The Logic Editor DSL family: low-level data schema DSL

Machine-friendly concepts, generated from high-level DSL:

```javascript
node:variant "dow" {
    "dow.mon", "dow.tue", "dow.wed", "dow.thu",
    "dow.fri", "dow.sat", "dow.sun";
    render:js [[Weekday]] { [[#{1}]] }; 
    render:lua { [[#{1}]] }; }
```

```javascript
node:literal "dow.mon" {
    render:js { [[Monday]] }; 
    render:lua { [[2]] }; }
```

```javascript
node:literal "dow.tue" {
    render:js { [[Tuesday]] }; 
    render:lua { [[3]] }; }
```

```
```
The Logic Editor DSL family: visual DSL

Show banner, only if:

1. Only for: Male
2. Only for marital statuses: In love
3. Only for applications of subject areas: Science, Games, Musics
4. Only for weekdays: Monday, Wednesday, Friday

© 2010–2011 LogicEditor.com
The Logic Editor DSL family: output targets

- From high-level DSL:
  - low-level DSL;
  - schema docs (to be implemented).

- From low-level DSL:
  - schema-specific visual Editor UI;
  - data validators;
  - data-to-code generator;
  - data upgrade code stubs to handle schema changes.
Why game-changing?

- **Before:**
  - DSL is something exotic, hard to maintain.
  - Not much declarative code in codebase except a few special places.
  - All declarative code in code-base totally non-reusable ad-hoc lumps of spaghetti.
  - Code readability suffers, bugs thrive.

- **Now:**
  - DSLs are much easier to write and reuse.
  - At least 2/3 of the new code is written in DSL of one kind or another. (But we heavily combine declarative DSL code with Lua functions embedded in it.)
  - Code much more readable, less bugs in generated code.

- **In future:**
  - A DSL to define DSLs!
Questions?

Alexander Gladyshev, ag@logiceditor.com