Scribunto

An introduction and hands-on tutorial

Wikimedia Hackathon Amsterdam 2013
May 24–26, 2013

An intro to Lua in MediaWiki

- We'll be going through a few examples, so you'll need a web browser open to a wiki with Scribunto installed.
  - https://test.wikipedia.org would be a good choice.
- Choose unique names for your pages, e.g. “Module:Foo/MYUSERNAME”.
- To begin, open
  - Module:Welcome/MYSUERNAME
  - User:MYSUERNAME/Scribunto_testing

Example #1: Module:Welcome

```
local p = {}

function p.everyone()
    return 'Welcome, everyone!'
end

return p
```

A Scribunto module to be used with {{#invoke}} must return a table containing the availble functions. Usually, we declare that table as a local at the top of the module, then add functions below.

A function can be declared directly as a member of a table. Or you could do an assignment:
`p.everyone = function() ... end`

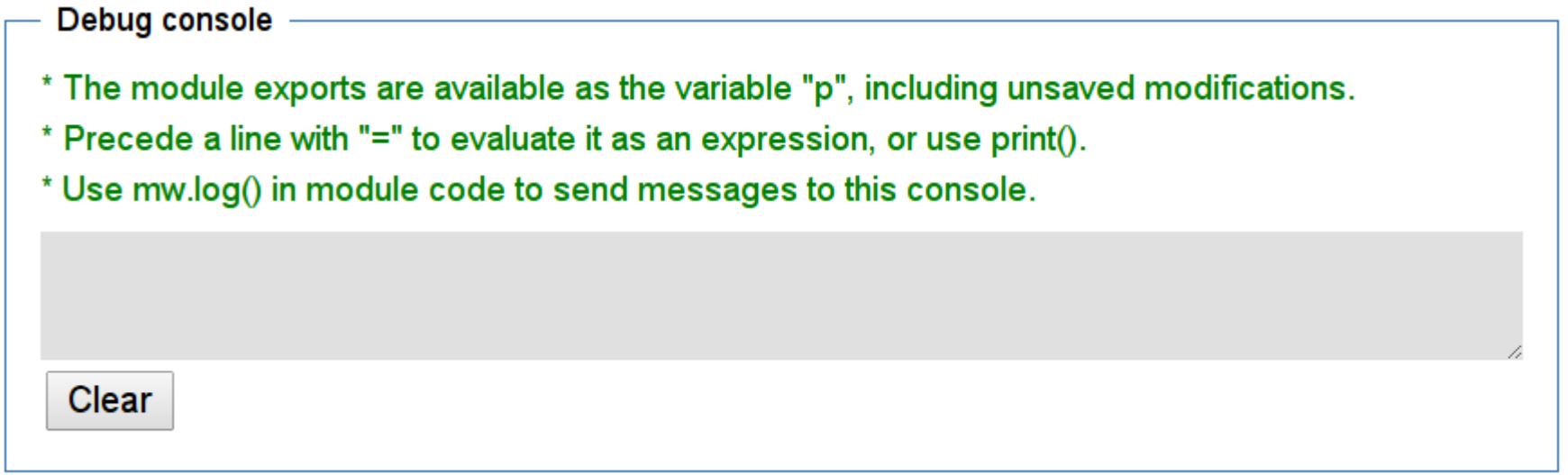
The invoked function returns a string of preprocessed wikitext. That means templates and tag hooks (e.g. <ref>) won't be processed, but other wikimarkup will work as usual.

Use this as `{{#invoke:Welcome|everyone}}`, resulting in “Welcome, everyone!”.

The first parameter to #invoke is the module name (without the prefix). The second is the function name. Anything after is passed to the function.

Debug console

- At the bottom of the edit page for the module, you'll see the debug console:



- Enter “`=p.welcome()`” and see what happens!

Example #2: by name

```
local p = {}

function p.welcome()
    return 'Welcome, everyone!'
end

function p.byname( frame )
    local name = frame.args[1]
    if name == nil then
        name = '<default>'
    end
    return 'Welcome, ' .. name .. '!'
end

return p
```

The `frame` object holds the arguments to the #invoke, and has additional methods to preprocess wikitext, expand templates, and call parser functions.

This accesses the first unnamed paramater passed to #invoke. Lua's 1-based indexing matches MW templates.

Any unset parameter has a nil value, so providing a default is easy.

The concatenation operator in Lua uses two dots. It works on strings and numbers, and errors for any other type

`{{#invoke:Welcome|byname|Hackathon people}}` results in “Welcome, Hackathon people!”

Debug console, again

- Debugging is a little more tricky with frame objects.

```
frame = mw.getCurrentFrame()   -- Get a frame object

newFrame = frame:newChild{     -- Get one with args
    args = { 'User' }
}

=p.byname( newFrame )          -- And use it
```

This is a method call. It's equivalent to a function call like `frame.newChild( frame, ... )`

Note how the parens are omitted in this call. When passing a single table to a function, this is a handy shortcut; it's often used to implement “named args”

Lua tables serve as both objects and arrays. This is equivalent to JSON `{ args: [ 'User' ] }`

Example #3: Named arg

```
local p = {}

function p.welcome()
    return 'Welcome, everyone!'
end

function p.byname( frame )
    local name = frame.args[1]
        or frame.args['name']
        or '<default>'
    return 'Welcome, ' .. name .. '!'
end

return p
```

Named arguments are accessed the same way. If the argument name is an identifier (letter or underscore, followed by letters, numbers, or underscores), you could also reference it as “frame.args.name”.

This use of the “or” logical operator is common for specifying default parameters.

What is Lua?

(much of this summary is taken from lua.org)

- Lua is a powerful, light-weight, interpreted scripting language with procedural syntax and dynamic typing.
- Designed, implemented, and maintained by a team at PUC-Rio in Brazil.
- First public release (1.1) in 1994; 5.1 in 2006, and 5.2 in December 2011.
- Available under the MIT license.

Similarities to other languages

- Syntax not entirely unlike JavaScript, C, PHP, etc.
  - Procedural, functional, object-oriented, and data-driven programming
- First-class functions as closures with lexical scoping, like JavaScript
- Strings delimited with single or double quotes
- Table syntax not unlike JS object literals
- Automatic garbage collection

Differences from other languages

- Blocks delimited with keywords (“do ... end”) rather than braces
- Comments are “--” or “--[[...]]”, not “//” and “/* .. */”
- Inequality is “~=”, not “!=”
- Concatenation is “..”, not “.” or “+”
- “Arrays” (tables with integer keys) are indexed starting at 1, not 0
- Only nil and false are boolean “false”; the empty string and 0 are both “true”

Example #4: I18n

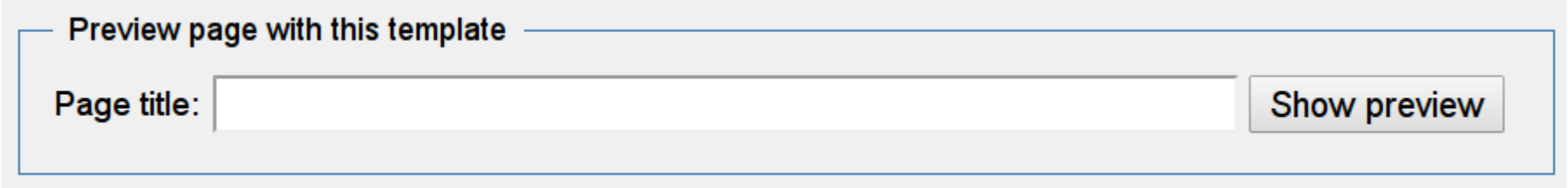
```
local p = {}

function p.byname( frame )
    local name = frame.args[1]
        or frame.args['name']
        or '<default>'

    local msg = mw.message( 'welcomeuser' )
    return msg:params( name ):text()
end

return p
```

Scribunto provides easy access to MediaWiki's Message class, which uses the pages in the MediaWiki namespace for i18n. Interface to MW's Language class is also available.

The interface is much like that of the Message class, too. Note the method call syntax again (with colons) and the method chaining.

Digression: TemplateSandbox

- An extension that goes well with Scribunto is TemplateSandbox.
- Just under the edit buttons, you should see a box like this:



- You can use this to preview your changes to the module *without saving it first*.
- Try it!

Example: Cite web
(simplified)

- Looking at the real Scribunto module for enwiki's {{cite web}} would be too much for this presentation. So let's look at a simplified version.
- But we can look at the code for the calling template, it's that simple:

```
<includeonly>{{#invoke:citation/CS1|citation
|CitationClass=web
}}</includeonly><noinclude>
{{documentation}}
</noinclude>
```

- But why don't we have to pass in the title, author, and so on?

Example: Cite web (simplified)

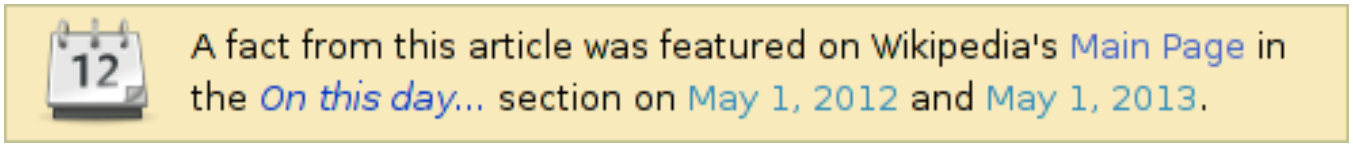
```
local p = {}

function p.citation( frame )
   local class = frame.args.CitationClass
    local pargs = frame:getParent().args
    local title = pargs.title
    local url = pargs.url
    local author = pargs.author
    local accessdate = pargs.accessdate
    local lang = mw.getContentLanguage()

    return author .. '. "[' .. url .. ' ' .. title .. ']." ..
           lang:formatDate( 'Y-m-d', accessdate )
end

return p
```

frame:getParent() gets the frame for the template #invoking this function, so we don't have to pass all possible parameters explcitily. **Very** useful.

This gets the mw.language object for the wiki's content language.

Date formatting, just like the {{#time:}} parser function.

Example #5: On this day




```
{{On this day|oldid1=12345|date1=2012-05-01|oldid2=67890|date2=2013-05-01}}
```

- {{On this day}} is a template on the English Wikipedia used to display a box on an article's talk page saying when the article was mentioned in the main page's “On this day...” section.
- Some articles are featured a lot: Teachers' Day is featured 13 times per year, on the days it is celebrated in various countries. So it has been featured over 100 times since 2004...
- Before Scribunto, it used two templates:

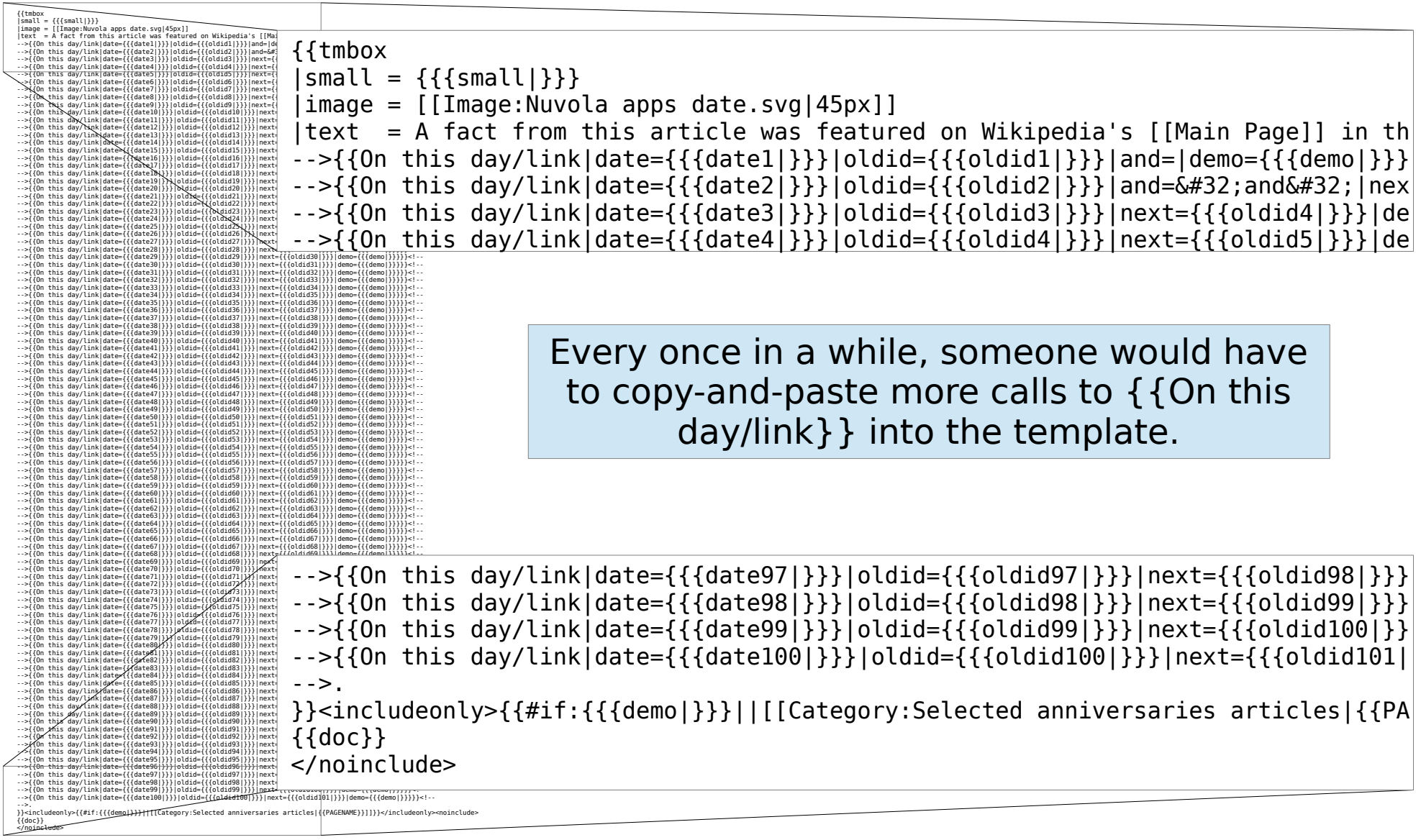
{{On this day/link}}

Note how each invocation needs to decide whether to output “, and date” or just “, date”, with special handling for the first and second dates.

```
{{#if:{{{oldid|}}}|{{#if:{{{next|}}}|,&#32;|{{{and|,
and&#32;}}}}} [{{fullurl:Wikipedia:Selected anniversaries/
{{#time:F j|{{{date}}}}}|oldid={{{oldid}}}}} {{#formatdate:
{{{date}}}|mdy}}]{{#if:{{{demo|}}}||[[Category:Selected
anniversaries ({{#time:F Y|{{{date}}}}})|{{PAGENAME}}]]}}}}
```

All this to format just one of the day links within the template!

{{On this day}}

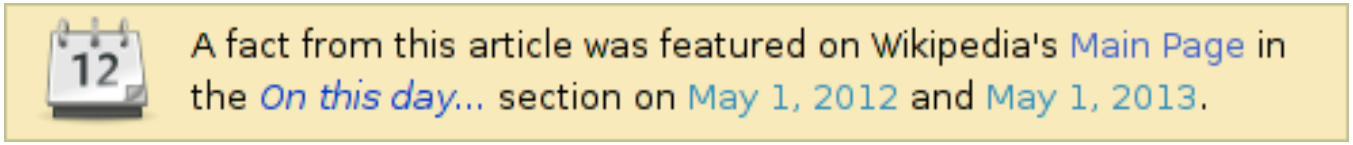


{{On this day}}, with Scribunto

```
{{tmbox
|small = {{{small|}}}
|image = [[Image:Nuvola apps date.svg|45px]]
|text  = A fact from this article was featured on Wikipedia's
[[Main Page]] in the ''[[Wikipedia:Selected anniversaries|On
this day...]]'' section on {{#invoke:On this day|showDates}}.
}}<includeonly>{{#if:{{{demo|}}}|[[Category:Selected
anniversaries
articles|{{PAGENAME}}]]}}</includeonly><noinclude>
{{doc}}
</noinclude>
```

All those calls to {{On this day/link}} get replaced by a single #invoke. And we don't even have to pass any parameters, thanks to frame:getParent().

Hands-on: Template:On this day




```
{{On this day|oldid1=12345|date1=2012-05-01|oldid2=67890|date2=2013-05-01}}
```

So, how would you write the module?

The wikitext from the previous slide may be copied from

http://test.wikipedia.org/wiki/User:BJorsch_(WMF)/Template:On_this_day

Don't forget to adjust the #invoke call to point to your own version of the module!

Module:On this day

```
function p.showDates( frame )
    local args = frame:getParent().args
    local ret = {}
    local page = mw.title.getCurrentTitle().text
    local fmt = '[//en.wikipedia.org/wiki/Wikipedia:Selected_anniversaries/%s?oldid=%s %s]'
    if not args.demo then
        fmt = fmt .. '[[Category:Selected anniversaries (%s)|%s]]'
    end
    local lang = mw.getContentLanguage()

    local i = 1
    while true do
        local date = args['date' .. i] or ''
        local oldid = args['oldid' .. i] or ''
        if oldid == '' then
            break
        end
        ret[i] = string.format( fmt,
            lang:formatDate( 'F_j', date ),
            oldid,
            lang:formatDate( 'F j, Y', date ),
            lang:formatDate( 'F Y', date ),
            page
        )
        i = i + 1
    end

    i = #ret -- i = number of entries in ret
    if i > 1 then
        ret[i] = 'and ' .. ret[i]
    end
    return table.concat( ret, i > 2 and ', ' or ' ' )
end
```

Like {{PAGENAME}}

Formatting string, like for printf

Loop, starting with i=1, and fetch the date*i* and oldid*i* until we find one that isn't specified.

Plug each date (properly formatted) and oldid into the formatting string, and store it into a table.

Then insert the “and” just before the last element,

And join all the strings together to be returned.

Lua Reference

- Scribunto Lua reference manual: [[mw:Extension:Scribunto/Lua reference manual]]
- Standard Lua 5.1 reference manual: http://www.lua.org/manual/5.1/manual.html
- *Programming in Lua*: (Covers Lua 5.0, but much is still relevant) http://www.lua.org/pil/
- Lua users wiki: (sadly, not running MediaWiki) http://lua-users.org/wiki/

Scribunto Lua vs standard Lua

- Based on Lua 5.1
- Many standard libraries removed:
  - All local filesystem access.
  - Process state query/manipulation.
  - Dynamic code loading, to allow static analysis and avoid Lua code being embedded in articles/templates.
  - print(), in favor of returning strings to MediaWiki
  - Coroutines, mainly because no pressing use was known. But see bug 47799.

Scribunto Lua vs standard Lua

- Standard library modifications:
  - Package library can load from the Module namespace
  - tostring() omits pointer addresses of functions and tables
  - Function environment and metatable manipulation is limited.
  - __pairs and __ipairs metamethod support is added

Additional Scribunto libraries

- All default-loaded libraries are “namespaced” under the mw global:
  - mw: Parser frames and some utility functions
  - mw.language: Interface to MediaWiki's Language class
  - mw.message: Access to interface messages
  - mw.site: Access to site configuration, e.g. namespaces
  - mw.text: Text processing
  - mw.title: Interface to MediaWiki's Title class
  - mw.uri: URI manipulation
  - mw.ustring: UTF-8-aware string functions
- Other libraries may be loaded with require():
  - bit32: Implementation of Lua 5.2's bit32 library
  - libraryUtil: Type-checking functions

How and Why

Now that we've seen it in action, let's take a look behind the scenes why Lua was chosen and how the extension actually works.

What's wrong with ParserFunctions?

```
{{#if:{{{var}}}|{{template|{{{var}}}}}|<strong
class=”error”>Error</stong>}}
```

- Hard to read syntax, lots of braces.
- Mixes code and presentation.
- No local variables, looping, or recursion.
- Little capability for string manipilation.
- Slow, especially when things get complicated.

Scribunto Design Goals

- Use a real language with clean syntax.
- Separate code from content. No code embedded in wikitext!
- Each invocation in a page is independent, no communication from one to the next.
- Fast, well-sandboxed, and limited in both time and memory usage.
- Usable for non-WMF sites.

Scribunto History

- Embedding a real scripting language has been discussed since 2009 (see bug 6455, comment 81), if not earlier.
- Volunteer Victor Vasiliev developed a prototype in 2011.
- Development began in earnest in January 2012, with Tim Starling working on cleaning up and improving the prototype.

Scribunto History

- In November 2012, Brad Jorsch was assigned to the project and began writing libraries such as mw.ustring.
- On February 18, 2013, Scribunto was deployed to the English Wikipedia and several others.
- On March 13, 2013, Scribunto was enabled on all WMF wikis.

How Scribunto works

(deep MediaWiki magic)

- It adds a new namespace, “Module”, to hold the scripts.
  - The CanonicalNamespaces hook adds the namespace.
  - CodeEditorGetPageLanguage tells CodeEditor the code language, if available.
  - EditPageBeforeEditChecks and EditPageBeforeEditButtons manipulate the edit form, adding a checkbox to skip validation, removing the “Preview” button, adding the script for the debug console.
  - EditFilterMerged validates the script, to avoid saving scripts with syntax errors.
  - ContentHandlerDefaultModelFor specifies a custom ContentHandler model for Scribunto modules.

How Scribunto works

(deep MediaWiki magic)

- It hooks into the parser to add {{#invoke:}}
  - ParserFirstCallInit adds {{#invoke:}}
  - ParserLimitReport adds information to the “NewPP Limit Report” comment.
  - ParserClearState and ParserCloned handle additional parser-related changes.
  - SoftwareInfo adds Lua version info to Special:Version.
  - And a parser output hook is registered for handling the “Script Error” popup.

How Scribunto works

(extension internals)

- Scribunto is language-agnostic, able to handle “engines” for any scripting language someone cares to write.
  - But currently only Lua support is written, so that's what we'll talk about.
  - Currently, only one engine can be enabled for a wiki. But that could be changed.
- Two equivalent engines are available: LuaSandbox and LuaStandalone.

How Scribunto works

(Lua engines)

- Both Lua engines share a great deal of common code.
  - Built-in libraries of Lua code
  - Loading libraries and modules
  - Creating an inner sandbox for each #invoke
  - Interface with Scribuno and MediaWiki
  - Unit tests, etc.
- Hooks are available for other extensions (e.g. Wikidata) to easily provide additional libraries, Lua and otherwise.

How Scribunto works

(LuaSandbox engine)

- LuaSandbox uses a PHP extension, luasandbox, to embed Lua directly in PHP.
- The PHP extension handles providing the outer sandbox, the CPU and memory limits, and marshaling data between PHP and Lua.
- This is *FAST*: Lua↔PHP calls take only a few microseconds, comparable to PHP↔PHP calls.

How Scribunto works

(LuaStandalone engine)

- LuaStandalone is slower, but portable: only proc_open() and a standalone Lua interpreter are needed.
- Uses IPC to communicate with the stock Lua standalone interpreter running a simple “server” script.
- This server script handles the outer sandbox and data serialization. CPU and memory limits are handled by the shell's ulimit (where available).

Why Lua?

- Pros:
  - Small (170K standalone) and fast. And a mature JIT is available, if more speed is needed.
    - So small, we distribute Linux (32- and 64-bit), Windows (32- and 64-bit), and OS X binaries with Scribunto!
  - Designed for embedding, including easy hooks for CPU and memory limiting.
  - Easy sandboxing, no internal globals.
  - Detailed reference manual, including instructions on embedding.
- Cons:
  - Less well-known than some other options.

Why not JavaScript?

- Pros:
  - Well-known language, likely familiar to our users.
- Cons (V8):
  - Minimal documentation on embedding.
  - Continued support for embedding unclear.
  - No allocation hook.
  - Huge standalone binary.
- Cons (Rhino):
  - Requires Java on the server.
  - Can't be embedded in PHP, would require slow IPC.
  - Very slow startup.

Why not PHP?

- Pros:
  - Well-known language, likely familiar to our users.
  - Anyone running MediaWiki has PHP available.
- Cons:
  - PHP is riddled with global state, almost impossible to sandbox.
  - Trying to pre-parse and then eval would be slow and error-prone.

Why not “WikiScript”?

- Pros:
  - We could make it work however we want.
  - Interpreter could be written in PHP.
- Cons:
  - Not known by anyone, because it doesn't exist yet.
  - An interpreter written in PHP would be slow.
  - We would have to design and implement the whole thing from scratch. Maybe twice, if we wanted a PHP version for portability and a C version for speed.

What we've found

- Impressive performance gains
  - {{rnd}} is 3.25 times faster
  - {{precision}} is 3.3 times faster
  - {{max}} is 2.375 times faster
  - {{str len}} is almost 8 times faster
  - {{str find}} is 54 times faster
  - {{convert}} will be 13 times faster
  - {{weather box}} is 18.7 times faster
  - {{citation}} is 7.6 times faster

Figures courtesy of Robert Rohde, [[en:User:Dragons Flight]]
https://en.wikipedia.org/wiki/User:Dragons_flight/Lua_performance

What we've found

- Interesting new capabilities:
  - Error checking for citation templates
  - Automatic calculation of the dates of Easter and related holidays
  - HTML-formatted bar charts
  - Layout a chess board given Forsyth–Edwards Notation
  - String functions not limited by length
  - And probably a lot I don't know about!

Future plans

- Show Scribunto log output on page preview
- Some sort of “central wiki” for modules (and gadgets and templates, too)
- Performance improvements
- New features?

Questions?