An Introduction to the Puppet 4 language
The puppet 4.x language has a lot of cool features, and why should you care
Puppetlabs has a bad rep for language design

“puppet's language design is like what you'd expect if someone spent years poring over the designs of all the greatest languages in the history of computing, and brilliantly deduced the best possible combination of attributes for the One True Perfect Language. and then someone hit that guy over the head with a hammer, put his brains in a blender, fed the mush to a flock of seagulls, had the seagulls eaten by a killer whale, and then interpreted the killer whale's shit as a design document.” --bblack
Everyone dislikes the puppet DSL.

It seems like Puppetlabs finally listened.
New features

- Data Types
- Loops and data manipulation
- Puppet native functions
- New ruby function API
- Resource defaults and ordering
- Miscellanea
### Data Types

- String, Integer, Float, Numeric, Boolean, Undef
- Array<Integer>, Hash[String][Integer]
- Regexp[/\s{3}123/]  
- Struct
- Variant, Optional
How to use data types

# Puppet 3.x
class foo($bar) { validate_string($bar) ... }

# Puppet 4.x
class foo(String $bar) { ... }
Define your own types

# modules/wmflib/types/useripport.pp

type Wmflib::UserIpPort = Integer[1024, 49151]
Define your own types

type Mcrouter::Ssl = Variant[Undefined, Struct[{ 
    'port' => Wmflib::IpPort, 
    'ca_cert' => Stdlib::Unixpath, 
    'cert' => Stdlib::Unixpath, 
    'key' => Stdlib::Unixpath ]}]

How to use data types

- All parameters of all classes/defines in modules should use types as a validation system. Less important for profiles.
- Puppet didn’t become strongly typed overnight, but will fail with obvious mishandling of parameters.
- Custom data types should be in wmflib if general enough, in the module otherwise.
Loops

- each
- map
- reduce
- slice
- filter
Iteration

# Looping over an array
$elements = ['He', 'C', 'N']
$elements.each { |$el| {
    file { "/tmp/$el":
        content => "I found $el!\n"
    }
}

# Loop over a hash

$elements = {'He' => 2, 'C' => 6, 'N' => 7}

$elements.each |$el, $at| {
    file { "/tmp/$el":
        content => "The AN for $el is $at!\n"
    }
}

Iteration/2
Native data merges

$even = [1, 3, 5, 7]
$odd = [2, 4, 6, 8]
$even + $odd  # numbers 1..8
$a = { 'foo' => 0, 'bar' => 5 }
$b = { 'bar' => 3, 'baz' => 2 }
$a + b  # merge($a, $b)
Things get nastier with map/reduce

```php
$pool_configs = $servers_by_datacenter.map | {$region, $servers} |
    # We need to get the servers from the current datacenter, and the proxies from the others
    if $region == $::site {
        profile::mcrouter_pools($region, $servers)
    } else {
        profile::mcrouter_pools($region, $proxies_by_datacenter[$region])
    }

$pools = $pool_configs.reduce | {$memo, $value} |
    $memo + $value
```
How to use loops

Use loops for:

- Modify, compose data structures, in the way we did with templates or parser functions before
- Declare sets of resources with variable arguments
- Wherever you would’ve used create_resources
How NOT to use loops

It’s very tempting to use loops everywhere. They make less obvious what is declared and how, so they should NOT be used when:

- A simple declaration of a static resource with an array of titles could work
- You could easily use a series of resources with many defaults
New functions interface

- Puppet native functions are easy to write but not as powerful as Ruby functions
- New function Ruby API, the only one that will be supported in the future
Native functions

# <modulepath>/foo/functions/bar.pp

function foo::bar(String $baz) >> Integer {
    if $baz == "foobar" {
        1
    } else {
        0
    }
}


New function API

- Are located under `<module_name>/lib/functions`
- Function signature (multiple of them!) need to be defined in dispatch stanzas
- Can be namespaced (so stdlib::merge could coexist with wmflib::merge)
- Can read, not write to the puppet scope
- Can accept puppet code blocks!
Puppet::Functions.create_function(:'mediawiki::dsn') do
  dispatch :standard_port do
    param 'String', :host_name
  end
  dispatch :all_params do
    param 'String', :host_name
    param 'Integer', :port
  end
  def standard_port(host_name)
    "mysql://#{host_name}:3306"
  end
  def all_params(host_name, port)
    "mysql://#{host_name}:#{port}"
  end
end

New function API
Resource defaults

```yaml
file {
  default:
    mode   => 0444,
    owner  => 'root',
    group  => 'root'
  ;
  '/etc/default/foobar':
    content => 'baz',
    mode    => 0400,
  ;
  ...
}
```
Resource ordering

- Resources in a manifest will be ordered according to the position in the file.
- That's unless you declare “require/before/after/notify”, that takes precedence.
- Containment of resources is still not possible (do NOT use “include” if not in roles).
Miscellanea

- $\text{facts}['\text{processorcount}']$ vs $\text{::processorcount}$
- $y = [\text{'hello'}, \text{'world'}].\text{join}(',, ')$
- $\text{redis\_settings} = \text{Redis::Instance}[\text{'6379']}][\text{'settings'}]$ 
- lookup() vs hiera()
That’s all for today!

Take the time to learn the new constructs in puppet 4.x; it is definitely worth it.

For examples of good use of puppet 4 constructs, you can use the “httpd” and the “php” modules as examples.

Also take your time to go back to your old code and see wherever you did one of those crazy workarounds for puppet’s lack of data structure iteration, and maybe fix it. Code will be clearer and more maintainable.

As an example, check [this patchset](#).
THANK YOU