Bolt

Contents

Writing tasks

Tasks allow you to perform ad hoc actions on remote nodes. Tasks can be written in any programming language the remote node will run.

Naming tasks

Task names are named based on the filename of the task, the name of the module, and the path to the task within the module.

You can write tasks in any language that will run on the target nodes. Give task files the extension for the language they are written in (such as `.rb` for Ruby), and place them in your module's `./tasks` directory.

Task names are composed of two or more name segments, indicating:

- The name of the module the task is located in.
- The name of the task file, withouth the extension.
- The path within the module, if the task is in a subdirectory of `./tasks`.

For example, the `puppetlabs-mysql` module has the `sql` task in `./mymodule/tasks/sql.rb`, so the task name is `mysql::sql`. A task defined in `./mymodule/tasks/service/start.rb` would be `mymodule::service::start`. This name is how you refer to the task when you run task commands.

The task filename `init` is special: the task it defines is referenced using the module name only. For example, in the `puppetlabs-service` module, the task defined in `init.rb` is the `service` task. Use this kind of naming sparingly, if at all, as top named elements like this can clash with other constructs.

Task names must be unique. If there are two tasks with the same name in a module, the task runner won't load either of them.

Task names must not have the same name as any Puppet data type, because when you run tasks with Puppet, they are loaded as unique data types. For a complete list of Puppet data types, see the *data type documentation*.

Each task or plan name segment must begin with a lowercase letter and:

- May include lowercase letters.
- May include digits.
- May include underscores.
- Must not use *reserved words*.
- Must not use the reserved extensions `.md` or `.json`.
- Must not have the same name as any Puppet data types.
- Namespace segments must match the following regular expression `\A[a-z][a-z0-9_]*\Z`

Defining parameters in tasks

Add parameters to your task as either environment variables or as a JSON hash on standard input.

Tasks can receive input as either environment variables, a JSON hash on standard input, or as PowerShell arguments. By default, the task runner submits parameters as both environment variables and as JSON on `stdin`.

If your task should receive parameters only in a certain way, such as `stdin` only, you can set the input method in your task metadata. For Windows tasks, it's usually better to use tasks written in PowerShell. See the related topic about task metadata for information about setting the input method.

Environment variables are the easiest way to implement parameters, and they work well for simple JSON types such as strings and numbers. For arrays and hashes, use structured input instead. See the related topic about structured input and output for more information.

To add parameters to your task as environment variables, pass the argument prefixed with the Puppet task prefix `PT_` .

For example, to add a `message` parameter to your task, pass it in the task code as `PT_message`. When the user runs the task, they specify the value for the parameter on the command line as `message=hello` , and the task runner submits the value `hello` to the PT_message variable.

```
#!/usr/bin/env bash
echo your message is $PT_message
```

Defining parameters in Windows

For Windows tasks, you can pass parameters as environment variables, but it's usually more useful to write your task in PowerShell and use named arguments. To support PowerShell standard argument handling, set the `input_method` in your metadata to `powershell`.

For example, this PowerShell task takes a process name as an argument and returns information about the process. If no parameter is passed by the user, the task returns all of the processes.

```
[CmdletBinding()]
Param(
  [Parameter(Mandatory = $False)]
 [String]
  $Name
  )

if ($Name -eq $null -or $Name -eq "") {
  Get-Process
} else {
  $processes = Get-Process -Name $Name
  $result = @()
  foreach ($process in $processes) {
    $result += @{"Name" = $process.ProcessName;
                 "CPU" = $process.CPU;
                 "Memory" = $process.WorkingSet;
                 "Path" = $process.Path;
                 "Id" = $process.Id}
  }
  if ($result.Count -eq 1) {
    ConvertTo-Json -InputObject $result[0] -Compress
  } elseif ($result.Count -gt 1) {
    ConvertTo-Json -InputObject @{"_items" = $result} -Compress
  }
}
```

To pass parameters in your task as environment variables (`PT_parameter`), you must set `input_method` in your task metadata to `environment`. To run Ruby tasks on Windows, the Puppet agent must be installed on the target nodes. Tasks written in languages other than Ruby or PowerShell are run with the command shell.

Related topics

Structured input and output on page 8
If you have a task that has many options, returns a lot of information, or is part of a task plan, you might want to use structured input and output with your task.

Task metadata on page 7
Task metadata files describe task parameters, validate input, and control how the task runner executes the task.

Converting scripts to tasks

To convert an existing script to a task, you can either write a task that wraps the script or you can replace the logic in your script to check for environment variables instead of assigning arguments.

If the script is already installed on the target nodes, you can write a task that wraps the script. In the task, specify the script arguments as task parameters and call the script, passing the parameters as the arguments.

If the script isn't installed or you want to make it into a cohesive task that you can manage its version with code management tools, add code to your script to check for the environment variables, prefixed with `PT_`, and assign them instead of arguments.

Given a script that accepts positional arguments on the command line:

```
version=$1
[ -z "$version" ] && echo "Must specify a version to deploy &&
 exit 1

if [ -z "$2" ]; then
  filename=$2
else
  filename=~/myfile
fi
```

To convert the script into a task, replace this logic with task variables:

```
version=$PT_version #no need to validate if we use metadata
if [ -z "$PT_filename" ]; then
  filename=$PT_filename
else
  filename=~/myfile
fi
```

Supporting no-op in tasks

Tasks can support no-operation functionality, also known as no-op mode. This function shows what changes the task would make, without actually making those changes.

No-op support allows a user to pass the `--noop` flag with a command to quickly test whether the task will succeed on all targets before making changes.

To support no-op, your task must include code that looks for the `_noop` metaparameter.

If the user passes the `--noop` flag with their command, this parameter is set to `true`, and your task must not make changes. You must also set `supports_noop` to `true` in your task metadata.

No-op metadata example

```
{
  "description": "Write content to a file.",
  "supports_noop": true,
  "parameters": {
    "filename": {
      "description": "the file to write to",
      "type": "String[1]"
    },
    "content": {
      "description": "The content to write",
```

```
      "type": "String"
    }
  }
}
```

No-op task example

```
#!/usr/bin/env python
import json
import os
import sys

params = json.load(sys.stdin)
filename = params['filename']
content = params['content']
noop = params.get('_noop', False)

exitcode = 0

def make_error(msg):
  return {
      "_error": {
        "kind": "file_error",
        "msg": "Could not open file %s: %s" % (filename, str(e)),
        "details": {},
        }
      }
try:
  if noop:
    if not os.access(filename, os.W_OK):
      result = make_error("File %s is not writable" % filename)
    else:
      result = { "success": True }
  else:
    with open(filename, 'w') as fh:
      if not noop:
        fh.write(content)
      result = { "success": True }
except Exception as e:
  exitcode = 1
  result = make_error("Could not open file %s: %s" % (filename, str(e)))
json.dump(result, sys.stdout)
exit(exitcode)
```

Related topics

Task metadata on page 7
Task metadata files describe task parameters, validate input, and control how the task runner executes the task.

Handling errors

If your task errors, it should exit with a non-zero exit code to inform the task runner that it has failed.

To return a structured error, include an `_error` object with `msg`, `kind`, and `details` keys in your task result. You can include an `_error` key alongside other keys in your task.

```
_error: {
  msg: e.message,
  kind: "puppet_error",
  details: {}
```

```
}
```

Task metadata

Task metadata files describe task parameters, validate input, and control how the task runner executes the task.

Your task must have metadata to be published and shared on the Forge. Task metadata is specified in a JSON file with the naming convention `<TASKNAME>.json` .

Adding parameters to metadata

To document and validate task parameters, add them to your metadata.

To document and validate task parameters, add the parameters to the task's metadata as JSON object, `parameters`. If a task includes `parameters` in its metadata, it rejects any parameters input to the task which aren't defined in the metadata.

In the `parameter` object, give each parameter a description and specify its Puppet type. For a complete list of types, see the *types documentation*.

For example, to describe a `provider` parameter in the metadata file:

```
"provider": {
  "description": "The provider to use to manage or inspect the
 service, defaults to the system service manager",
  "type": "Optional[String[1]]"
  }
}
```

Task metadata reference

Task metadata keys, values, and defaults.

Task metadata

Metadata key	Description	Value	Default
"description"	A description of what the task does.	A string.	None.
"puppet_task_version"	The version of the spec used.		None.
"supports_noop"	Whether the task supports no-op mode. Required for the task to accept the `--noop` option on the command line.	Boolean.	False.
"input_method"	What input method the task runner should use to pass parameters to the task.	• `environment` • `stdin` • `powershell`	• both `environment` and `stdin` • for `.ps1` tasks, `powershell`
"parameters"	The parameters or input the task accepts listed with a puppet type string	• String specifying the Puppet data type	None.

Metadata key	Description	Value	Default
	and optional description. See *adding parameters to metadata* for usage information.	• String describing the parameter	

Task metadata types

this is a shortdesc

Task metadata types

Task metadata can accept most Puppet data types, but these are the most commonly used. For a complete list of available types, see the *types* documentation.

Restriction:

Some types supported by Puppet can not be represented as JSON, such as `Hash[Integer, String],Object,` or `Resource`. These should not be used in tasks, because they can never be matched.

Type	Description
`String`	Accepts any string.
`String[1]`	Accepts any non-empty string (a String of at least length 1).
`Enum[choice1, choice2]`	Accepts one of the listed choices.
`Pattern[/\A\w+\Z/]`	Accepts Strings matching the regex `` `/\w+/` `` or non-empty strings of word characters.
`Integer`	Accepts integer values. JSON has no Integer type so this can vary depending on input.
`Optional[String[1]]`	Optional makes the parameter optional and permits null values. Tasks have no required nullable values.
`Array[String]`	Matches an array of strings.
`Hash`	Matches a JSON object.
`Variant[Integer, Pattern[/\A\d+\Z/]]`	Matches an integer or a String of an integer

Structured input and output

If you have a task that has many options, returns a lot of information, or is part of a task plan, you might want to use structured input and output with your task.

The task API is based on JSON. Task parameters are encoded in JSON, and the task runner attempts to parse the output of the tasks as a JSON object.

The task runner can inject keys into that object, prefixed with _. If the task does not return a JSON object, the task runner creates one and places the output in an `_output` key.

Structured input

For more complex input, such as hashes and arrays, you can accept structured JSON in your task.

By default, the task runner passes task parameters as both environment variables and as a single JSON object on stdin. The JSON input allows the task to accept more complex data structures.

To accept parameters as JSON on stdin, set the `params` key to accept JSON on `stdin`.

```
#!/opt/puppetlabs/puppet/bin/ruby
require 'json'

params = JSON.parse(STDIN.read)

exitcode = 0
params['files'].each do |filename|
  begin
    FileUtils.touch(filename)
    puts "updated file #{filename}"
  rescue
    exitcode = 1
    puts "couldn't update file #{filename}"
  end
end
exit exitcode
```

Returning structured output

To return structured data from your task, include code in your task that prints a JSON object to `stdout`.

Structured output is useful if you want to use the output in another program, or if you want to use the result of the task in a Puppet task plan.

```
#!/usr/bin/env python
import json
import sys
minor = sys.version_info
result = { "major": sys.version_info.major, "minor":
 sys.version_info.minor }
json.dump(result, sys.stdout)
```

Writing plans

Plans allow you to run more than one task with a single command, or compute values for the input to a task, or make decisions based on the result of running a task. Plans are written in the Puppet Language.

Naming plans

Plan names are named based on the filename of the plan, the name of the module containing the plan, and the path to the plan within the module.

Write plan files in Puppet, give them the extension `.pp` , and place them in your module's `./plans` directory.

Plan names are composed of two or more name segments, indicating:

- The name of the module the plan is located in.
- The name of the plan file, without the extension.
- The path within the module, if the plan is in a subdirectory of `./plans`.

For example, given a module called `mymodule` with a plan defined in `./mymodule/plans/myplan.pp`, the plan name is `mymodule::myplan`. A plan defined in `./mymodule/plans/service/myplan.pp` would be `mymodule::service::myplan`. This name is how you refer to the plan when you run commands.

Plan names must be unique. If there are two plans with the same name in a module, the task runner won't load either of them.

The plan filename `init` is special: the plan it defines is referenced using the module name only. For example, in a module called `mymodule`, the plan defined in `init.pp` is the `mymodule` plan. Use this kind of naming sparingly, if at all, as top named elements like this can clash with other constructs.

Do not give plans the same names as constructs in the Puppet language. Although plans do not share their namespace with other language constructs, giving plans these names makes your code difficult to read.

Each plan name segment must begin with a lowercase letter and:

- May include lowercase letters.
- May include digits.
- May include underscores.
- Must not use *reserved words*.
- Must not use the reserved extensions `.md` or `.json`.
- Must not have the same name as any Puppet data types.
- Namespace segments must match the following regular expression `\A[a-z][a-z0-9_]*\Z`

Defining plan parameters

You can specify parameters in your plan.

Specify the parameter in your plan with its data type. For example, you might want parameters to specify which nodes to run different parts of your plan on.

This example shows node parameters specified as `Variant[String[1], Array[String[1]]`. `String[1]` means the String can not be empty, and `Array[String[1]]` is an array of non-empty strings. For `Variant[String[1]`, the value can be either `String[1]` or `Array[String[1]]`.

This allows the user to pass, for each parameter, either a simple node name or a URI that describes the protocol to use, the hostname, username and password.

The plan then calls the `run_task` function, specifying which nodes the tasks should be run on.

```
plan mymodule::my_plan(
  String[1]                              $load_balancer,
  Variant[String[1], Array[String[1]]]   $frontends,
  Variant[String[1], Array[String[1]]]   $backends,
) {

  # process frontends
  run_task('mymodule::lb_remove', nodes => $frontends, $load_balancer)
  run_task('mymodule::update_frontend_app', version => '1.2.3', $frontends)
  run_task('mymodule::lb_add', nodes => $frontends, $load_balancer)
}
```

To execute this plan from the command line, pass the parameters as `parameter=value`:

```
bolt plan run mymodule::myplan --modules ./PATH/TO/MODULES
 load_balancer=lb.myorg.com frontends=kermit.myorg.com,gonzo.myorg.com
 backends=waldorf.myorg.com,statler.myorg.com
```

Plan execution functions

Your plan can execute multiple functions on remote systems.

Your plan can include functions to run commands, scripts, tasks, and other plans on remote nodes. These execution functions correspond to task runner commands.

- `run_command`: Runs a command on one or more nodes.
- `run_script`: Runs a script (non task executable) on one or more nodes.
- `run_task`: Runs a task on one or more nodes.
- `run_plan`: Runs a plan on one or more nodes.
- `upload_file`: Uploads a file to one or more nodes.

Calling basic plan functions

Add basic functions to your plan by calling the function with its data type and arguments.

Basic functions in plans share a similar structure. Call these functions with their data type and parameters.

Function	Description	Data type	Parameters
`run_script`	Runs a script on one or more nodes.	`String[1]`	`$fileref`, `$nodes`
`file_upload`	Uploads a file to one or more nodes.	`String [1]`	`$source`, `$nodes`
`run_command`	Runs a command on one or more nodes.	`String[1]`	`$cmd`, `$nodes`

For the functions `run_script` and `file_upload`, the `$fileref` and `$source` parameters accept either an absolute path or a module relative path, `<MODULE NAME>/<FILE>` reference, which will search for `<FILE>` relative to a module's `files` directory. For example, the reference `mysql/mysqltuner.pl` searches for the file `<MODULES DIRECTORY>/mysql/files/mysqltuner.pl`.

For example, to have your plan run the script located in `./mymodule/files/my_script.sh` on a set of nodes:

```
run_script("mymodule/my_script.sh", $nodes)
```

Running tasks from plans

When you need to run multiple tasks, or you need some tasks to depend on others, you can call the tasks from a task plan.

To run a task from your plan, call the `run_task` function, specifying the task to run, any task parameters, and the nodes on which to run the task. Specify the full task name, as `<MODULE>::<TASK>`.

For example, this plan runs several tasks, each on a different set of nodes:

```
run_task('mymodule::lb_remove', nodes => $frontends, $load_balancer)
run_task('mymodule::update_frontend_app', version => '1.2.3', $frontends)
run_task('mymodule::lb_add', nodes => $frontends, $load_balancer)
```

Running plans in a plan

Your plan can run another plan.

A plan can be run from within another plan by using the function `run_plan`. This function accepts the name of the plan to run and a hash of arguments to the plan.

This example plan, `mymodule::update_everything`, runs the plan `mymodule::myplan`, passing the necessary parameter values to `mymodule::myplan`.

```
plan mymodule::update_everything {
  run_plan('mymodule::myplan',
    load_balancer => 'lb.myorg.com',
    frontends => ['kermit.myorg.com', 'gonzo.myorg.com'],
    backends => ['waldorf.myorg.com', 'statler.myorg.com' ])
```

```
}
```

Responding to errors in plans

In your plan, you can use the `Error` data type in a case expression to match against different kind of errors.

This allows you to include conditionals based on errors that occur while your plan runs, so your plan can try to recover from certain errors, while failing on or ignoring others. For example, you might want the plan to retry a task if there is a timeout error, but to fail if there is an authentication error.

The `Error` type includes information about:

- `message` : The error message `String`.
- `kind`: A `String` that defines the kind of error.
- `issue_code` : A `String` that is a code for the error message.
- `details` : A `Hash` with details about the error such as `exit_code` or `stack_trace`, depending on the task and the language it is written in.

This example matches two different kinds of errors with the `message` and `details` parameters:

```
case $result {
  Error['puppet/authentication'] : {
    notice("Authentication error on: ${node} - '${result.message}'")
  }
  Error['puppet/task-error'] : {
    notice("Task error on: ${node}")
  }
  Error : {
    notice("General error on ${node} - here are all the details
 ${result.details}")
  }
```

Puppet and Ruby functions in plans

You can define and call Puppet language and Ruby functions in plans.

This is useful for packaging common general logic in your plan. You can also call the plan functions, such as `run_task` or `run_plan`, from within a function.

Not all Puppet language constructs are allowed in plans. The following constructs are not allowed:

- Defined types.
- Classes.
- Resource expressions (such as `file { title: mode => '0777' }`).
- Resource default expressions (such as `File { mode => '0666' }`).
- Resource overrides (such as `File['/tmp/foo'] { mode => '0444' }`).
- Relationship operators (`->` `<-` `~>` `<~`).
- Functions that operate on a catalog: `include`, `require`, `contain`, `create_resources`.
- Collector expressions (for example `SomeType <| |>`, `SomeType <<| |>>`).
- ERB templates are not supported; use EPP instead.

You should be aware of some other Puppet behaviors in plans:

- The `--strict_variables` option is on, so if you reference a variable that is not set, you will get an error.
- `--strict=error` is always on, so minor language issues generate errors. For example `{ a => 10, a => 20 }` is an error because there is a duplicate key in the hash.
- Facts are for the machine where the script or plan is running, not for the node(s) on which the tasks are executed.

- Facts are available only as the hash `$facts`.
- Settings are for the machine where the script/plan is running.
- Logs include "source location" (file, line) instead of resource type or name.

Handling plan function results

Plan execution functions each return a result object that returns details about the execution.

Each execution function returns an object type `ExecutionResult`. For each node that the execution took place on, this object returns one execution result. This result is a `Variant[Data, Error]`, where an `Error` value indicates that something went wrong.

An `ExecutionResult` has the following methods:

- `names()`: The `String` names (node URIs) of all nodes in the set as an `Array`.
- `empty()`: Returns `Boolean` if the execution result set is empty.
- `count()`: Returns an `Integer` count of nodes.
- `value(node-uri)`: The `Variant[Error, Any]` for the given node URI.
- `values()` : An array of `Variant[Error, Any]` for the nodes in the result set.
- `ok_nodes()`: Returns an `ExecutionResult` containing all nodes that have a non `Error` result.
- `error_nodes()`: Returns an `ExecutionResult` containing all nodes that have an `Error` result.
- `ok():Boolean` that is the same as `error_nodes.empty`.

An instance of `ExecutionResult` is `Iterable` as if it were a hash of `String` node-uri, to `Variant[Error, Any]`, so that iterative functions such as `each`, `map`, `reduce`, or `filter` work directly on the execution result. You can also get the value for a single node using the access operator `[node-uri]` (the same as the `value(node-uri)` method).

This example checks if a task ran correctly on all nodes; if it did not, the check fails:

```
$r = run_task('sometask', ...)
unless $r.ok {
  fail("Running sometask failed on the nodes ${r.error_nodes.names}")
}
```

You can do iteration and checking if the result is an Error. This example outputs some simple feedback about the result of a task:

```
$r = run_task('sometask', ...)
$r.each |$node, $result | {
  case $result {
    Error : {
      notice("${node} errored with message ${result.message}")
    }
    default: {
      notice("${node} returned a value: ${result}")
    }
  }
}
```

Partial results

If the task produces a partial result and then errors, the value for the node is an `Error`, and you can get the partial output from the error object. This partial result is a special case when an executed task produces both a value (a string or JSON object) on `stdout` and an `_error` structure in the output. For example, this can happen when a task starts to write JSON, encounters an error and includes it, and then ends the JSON output

```
$result = run_task('sometask', ...)
```

```
if $result =~ Error {
  $partial = $result.partial_result
  if $partial {
    notice "partial result was produced: ${partial}"
  }
}
```

Returning errors in plans

To return an error if your plan fails, include an `Error` object in your plan.

Specify `Error` parameters to provide details about the failure.

For example, if called with `run_plan('mymodule::myplan')`, this would return an error to the caller.

```
plan mymodule::myplan {
  Error(
    message    => "sorry, this plan does not work yet",
    kind       => 'mymodule/error',
    issue_code => 'NOT_IMPLEMENTED'
    )
  }
```