Tasks and plans

Automate your workflow with tasks and plans.

Sometimes you need to do work in your infrastructure that isn't about monitoring and enforcing the desired state of machines. You might need to restart a service, run a troubleshooting script, or get a list of the network connections to a given node. You perform actions like these with Puppet tasks and plans.

Tasks

Tasks are single actions that you run on target machines in your infrastructure. You use tasks to make as-needed changes to remote systems.

You can write tasks in any programming language that can run on the target nodes, such as Bash, Python, or Ruby. Tasks are packaged within modules, so you can reuse, download, and share tasks on the Forge. Task metadata describes the task, validates input, and controls how the task runner executes the task.

Plans

Plans are sets of tasks that can be combined with other logic. This allows you to do complex task operations, such as running multiple tasks with one command, computing values for the input for a task, or running certain tasks based on results of another task. You write plans in the Puppetlanguage. And like tasks, plans are packaged in modules and can be shared on the Forge.

- Inspecting tasks and plans
  Before you run tasks or plans in your environment, inspect them to determine what effect they have on your target nodes.
- Running tasks
  Bolt can run Puppet tasks on remote nodes without requiring any Puppet infrastructure.
- Running plans
  Bolt can run plans, allowing multiple tasks to be tied together.
- Installing modules
  Tasks and plans are packaged in Puppet modules, so you can install them as you would any module and manage them with a Puppetfile.
- Directory structures for tasks and plans
  Puppet tasks, plans, functions, classes and types must exist inside a Puppet, module in order forBolt to load them. Bolt loads modules by searching for module directories on the modulepath.
- Writing tasks
  Tasks are similar to scripts, but they are kept in modules and can have metadata. This allows you to reuse and share them.
- Writing plans
  Plans allow you to run more than one task with a single command, compute values for the input to a task, process the results of tasks, or make decisions based on the result of running a task.

Inspecting tasks and plans

Before you run tasks or plans in your environment, inspect them to determine what effect they have on your target nodes.

Run in no operation mode

You can run some tasks in no-operation mode (`noop`) to view changes without taking any action on your target nodes. This way, you ensure the tasks perform as designed. If a task doesn't support no-operation mode, you get an error.

```
bolt task run package name=vim action=install --noop -n example.com
```

Show a task list

View a list of what tasks are installed in the current module path. Note that tasks marked with the `private` metadata key are not shown:

```
bolt task show
```

Show documentation for a task

View parameters and other details for a task, including whether a task supports `--noop`:

```
bolt task show <TASK NAME>
```

Discover plans

View a list of what plans are installed on the current module path:

```
bolt plan show
```

Show documentation for a plan

View parameters and other details for a plan, including whether a plan supports `--noop`:

```
bolt plan show <PLAN NAME>
```

Running tasks

Bolt can run Puppet tasks on remote nodes without requiring any Puppet infrastructure.

To execute a task, run `bolt task run`, specifying:

- The full name of the task, formatted as `<MODULE::TASK>`, or as `<MODULE>` for a module's main task (the `init` task).
- Any task parameters, as `parameter=value`.
- The nodes on which to run the task and the connection protocol, with the `--nodes` flag.
- If credentials are required to connect to the target node, the username and password, with the `--user` and `--password` flags.

For example, to run the `sql` task from the `mysql` module on node named neptune:

```
bolt task run mysql::sql database=mydatabase sql="SHOW TABLES" --nodes neptune --modulepath ~/modules
```

To run the main module task defined in `init`, refer to the task by the module name only. For example, the `puppetlabs-package` module contains only one task, defined as `init`, but this task can execute several actions. To run

the `status` action from this module to check whether the vim package is installed, you run:

```
bolt task run package action=status name=vim --nodes neptune --
modulepath ~/modules
```

Passing structured data

If one of your task or plan parameters accept structured data like an `array` or `hash`, it can be passed as JSON from the command line.

If a single parameter can be parsed as JSON and the parsed value matches the parameter's type specification in the task metadata or plan definition, it can be passed with `<>param=value` syntax. Make sure to wrap the JSON value in single quotes to prevent `"` characters from being swallowed by the shell.

```
bolt task run mymodule::mytask --nodes app1.myorg.com
load_balancers='["lb1.myorg.com", "lb2.myorg.com"]'

bolt plan run mymodule::myplan load_balancers='["lb1.myorg.com",
"lb2.myorg.com"]'
```

If you want to pass multiple structured values or are having trouble with the magic parsing of single parameters, you can pass a single JSON object for all parameters with the `--params` flag.

```
bolt task run mymodule::mytask --nodes app1.myorg.com --params
'{"load_balancers": ["lb1.myorg.com", "lb2.myorg.com"]}'

bolt plan run mymodule::myplan --params '{"load_balancers":
["lb1.myorg.com", "lb2.myorg.com"]}'
```

You can also load parameters from a file by putting `@` before the file name.

```
bolt task run mymodule::mytask --nodes app1.myorg.com --params
@param_file.json

bolt plan run mymodule::myplan --params @param_file.json
```

To pass JSON values in PowerShell without worrying about escaping, use `ConvertTo-Json`

```
bolt task run mymodule::mytask --nodes app1.myorg.com --params
$(@{load_balancers=@("lb1.myorg.com","lb2.myorg.com")} | ConvertTo-
Json)

bolt plan run mymodule::myplan --nodes app1.myorg.com --params
$(@{load_balancers=@("lb1.myorg.com","lb2.myorg.com")} | ConvertTo-
Json)
```

Specifying the module path

In order for Bolt to find a task or plan, the task or plan must be in a module on the `modulepath`. By default, the `modulepath` includes `modules/` and `site-modules/` directories inside the Bolt project directory.

If you are developing a new plan, you can specify `--modulepath <PARENT_DIR_OF/MODULE>` to tell Bolt where to load the module. For example, if your module is in `~/src/modules/my_module/`, run Bolt with `--modulepath ~/src/module`. If you often use the same `modulepath`, you can set `modulepath` in `bolt.yaml`.

Running plans

Bolt can run plans, allowing multiple tasks to be tied together.

To execute a task plan, run `bolt plan run`, specifying:

- The full name of the plan, formatted as `<MODULE>::<PLAN>`.
- Any plan parameters, as `parameter=value`.
- If credentials are required to connect to the target node, pass the username and password with the `--user` and `--password` flags.

For example, if a plan defined in `mymodule/plans/myplan.pp` accepts a `load_balancer` parameter to specify a load balancer node on which to run the tasks or functions in the plan, run:

```
bolt plan run mymodule::myplan load_balancer=lb.myorg.com
```

Note that, like `--nodes`, you can pass a comma-separated list of node names, wildcard patterns, or group names to a plan parameter that is passed to a run function or that the plan resolves using `get_targets`.

Passing structured data

If one of your task or plan parameters accept structured data like an `array` or `hash`, it can be passed as JSON from the command line.

If a single parameter can be parsed as JSON and the parsed value matches the parameter's type specification in the task metadata or plan definition, it can be passed with `<>param=value` syntax. Make sure to wrap the JSON value in single quotes to prevent `"` characters from being swallowed by the shell.

```
bolt task run mymodule::mytask --nodes app1.myorg.com
load_balancers='["lb1.myorg.com", "lb2.myorg.com"]'

bolt plan run mymodule::myplan load_balancers='["lb1.myorg.com",
"lb2.myorg.com"]'
```

If you want to pass multiple structured values or are having trouble with the magic parsing of single parameters, you can pass a single JSON object for all parameters with the `--params` flag.

```
bolt task run mymodule::mytask --nodes app1.myorg.com --params
'{"load_balancers": ["lb1.myorg.com", "lb2.myorg.com"]}'

bolt plan run mymodule::myplan --params '{"load_balancers":
["lb1.myorg.com", "lb2.myorg.com"]}'
```

You can also load parameters from a file by putting `@` before the file name.

```
bolt task run mymodule::mytask --nodes app1.myorg.com --params
@param_file.json

bolt plan run mymodule::myplan --params @param_file.json
```

To pass JSON values in PowerShell without worrying about escaping, use `ConvertTo-Json`

```
bolt task run mymodule::mytask --nodes app1.myorg.com --params
$(@{load_balancers=@("lb1.myorg.com","lb2.myorg.com")} | ConvertTo-
Json)

bolt plan run mymodule::myplan --nodes app1.myorg.com --params
$(@{load_balancers=@("lb1.myorg.com","lb2.myorg.com")} | ConvertTo-
Json)
```

Specifying the module path

In order for Bolt to find a task or plan, the task or plan must be in a module on the `modulepath`. By default, the `modulepath` includes `modules/` and `site-modules/` directories inside the Bolt project directory.
If you are developing a new plan, you can specify `--modulepath <PARENT_DIR_OF/MODULE>` to tell Bolt where to load the module. For example, if your module is in `~/src/modules/my_module/`, run Bolt with `--modulepath ~/src/module`. If you often use the same `modulepath`, you can set `modulepath` in `bolt.yaml`.

Writing tasks

Tasks are similar to scripts, but they are kept in modules and can have metadata. This allows you to reuse and share them.

You can write tasks in any programming language the target nodes run, such as Bash, PowerShell, or Python. A task can even be a compiled binary that runs on the target. Place your task in the `./tasks`directory of a module and add a metadata file to describe parameters and configure task behavior.
For a task to run on remote *nix systems, it must include a shebang (`#!`) line at the top of the file to specify the interpreter.
For example, the Puppet `mysql::sql` task is written in Ruby and provides the path to the Rubyinterpreter. This example also accepts several parameters as JSON on `stdin` and returns an error.

```
#!/opt/puppetlabs/puppet/bin/ruby
require 'json'
require 'open3'
require 'puppet'

def get(sql, database, user, password)
  cmd = ['mysql', '-e', "#{sql} "]
  cmd << "--database=#{database}" unless database.nil?
  cmd << "--user=#{user}" unless user.nil?
  cmd << "--password=#{password}" unless password.nil?
  stdout, stderr, status = Open3.capture3(*cmd) # rubocop:disable
Lint/UselessAssignment
  raise Puppet::Error, _("stderr: ' %{stderr}') % { stderr: stderr }")
if status != 0
  { status: stdout.strip }
end

params = JSON.parse(STDIN.read)
database = params['database']
user = params['user']
password = params['password']
sql = params['sql']

begin
  result = get(sql, database, user, password)
  puts result.to_json
  exit 0
rescue Puppet::Error => e
  puts({ status: 'failure', error: e.message }.to_json)
  exit 1
end
```

Secure coding practices for tasks

Use secure coding practices when you write tasks and help protect your system.
Note: The information in this topic covers basic coding practices for writing secure tasks. It is not an exhaustive list.
One of the methods attackers use to gain access to your systems is remote code execution, where by running an allowed script they gain access to other parts of the system and can make arbitrary changes. Because Bolt executes scripts

across your infrastructure, it is important to be aware of certain vulnerabilities, and to code tasks in a way that guards against remote code execution.
Adding task metadata that validates input is one way to reduce vulnerability. When you require an enumerated (`enum`) or other non-string types, you prevent improper data from being entered. An arbitrary string parameter does not have this assurance.
For example, if your task has a parameter that selects from several operational modes that are passed to a shell command, instead of

```
String $mode = 'file'
```

Use

```
Enum[file,directory,link,socket] $mode = file
```

If your task has a parameter that identifies a file on disk, ensure that a user can't specify a relative path that takes them into areas where they shouldn't be. Reject file names that have slashes.
Instead of

```
String $path
```

Use

```
Pattern[/\A[^\/\\]*\z/] $path
```

In addition to these task restrictions, different scripting languages each have their own ways to validate user input.

PowerShell

In PowerShell, code injection exploits calls that specifically evaluate code. Do not call `Invoke-Expression` or `Add-Type` with user input. These commands evaluate strings as C# code.
Reading sensitive files or overwriting critical files can be less obvious. If you plan to allow users to specify a file name or path, use `Resolve-Path` to verify that the path doesn't go outside the locations you expect the task to access. Use `Split-Path -Parent $path` to check that the resolved path has the desired path as a parent.
For more information, see PowerShell Scripting and Powershell's Security Guiding Principles.

Bash

In Bash and other command shells, shell command injection takes advantage of poor shell implementations. Put quotations marks around arguments to prevent the vulnerable shells from evaluating them.

Because the `eval` command evaluates all arguments with string substitution, avoid using it with user input; however you can use `eval` with sufficient quoting to prevent substituted variables from being executed.
Instead of

```
eval "echo $input"
```

use

```
eval "echo '$input'"
```

These are operating system-specific tools to validate file paths: `realpath` or `readlink -f`.

Python

In Python malicious code can be introduced through commands like `eval`, `exec`, `os.system`, `os.popen`, and `subprocess.call` with `shell=True`. Use `subprocess.call` with `shell=False`when you include user input in a command or escape variables.
Instead of

```
os.system('echo '+input)
```

use

```
subprocess.check_output(['echo', input])
```

Resolve file paths with `os.realpath` and confirm them to be within another path by looping over `os.path.dirname` and comparing to the desired path.
For more information on the vulnerabilities of Python or how to escape variables, see Kevin London's blog post on Dangerous Python Functions.

Ruby

In Ruby, command injection is introduced through commands like `eval`, `exec`, `system`, backtick (``) or `%x()` execution, or the Open3 module. You can safely call these functions with user input by passing the input as additional arguments instead of a single string.
Instead of

```
system("echo #{flag1} #{flag2}")
```

use

```
system('echo', flag1, flag2)
```

Resolve file paths with `Pathname#realpath`, and confirm them to be within another path by looping over `Pathname#parent` and comparing to the desired path.
For more information on securely passing user input, see the blog post Stop using backtick to run shell command in Ruby.

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 runs 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 the top level of your module's `./tasks` directory.

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

- The name of the module where the task is located.
- The name of the task file, without the extension.

For example, the `puppetlabs-mysql` module has the `sql` task in `./mysql/tasks/sql.rb`, so the task name is `mysql::sql`. This name is how you refer to the task when you run tasks.

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.

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

- Must start with a lowercase letter.
- May include digits.
- May include underscores.
- Namespace segments must match the following regular expression `\A[a-z][a-z0-9_]*\Z`
- The file extension must not use the reserved extensions .md or .json.

Single-platform tasks

A task can consist of a single executable with or without a corresponding metadata file. For instance, `./mysql/tasks/sql.rb` and `./mysql/tasks/sql.json`. In this case, no other `./mysql/tasks/sql.*` files can exist.

Cross-platform tasks

A task can have multiple implementations, with metadata that explains when to use each one. A primary use case for this is to support different implementations for different target platforms, referred to as *cross-platform tasks*.

A task can also have multiple implementations, with metadata that explains when to use each one. A primary use case for this is to support different implementations for different target platforms, referred to as `cross-platform tasks`. For instance, consider a module with the following files:

```
- tasks
  - sql_linux.sh
  - sql_linux.json
  - sql_windows.ps1
  - sql_windows.json
  - sql.json
```

This task has two executables (`sql_linux.sh` and `sql_windows.ps1`), each with an implementation metadata file and a task metadata file. The executables have distinct names and are compatible with older task runners such as Puppet

Enterprise 2018.1 and earlier. Each implementation has it's own metadata which documents how to use the implementation directly or marks it as private to hide it from UI lists.
An implementation metadata example:

```
{
  "name": "SQL Linux",
  "description": "A task to perform sql operations on linux targets",
  "private": true
}
```

The task metadata file contains an implementations section:

```
{
  "implementations": [
    {"name": "sql_linux.sh", "requirements": ["shell"]},
    {"name": "sql_windows.ps1", "requirements": ["powershell"]}
  ]
}
```

Each implementations has a `name` and a list of `requirements`. The requirements are the set of features which must be available on the target in order for that implementation to be used. In this case, the `sql_linux.sh` implementation requires the `shell` feature, and the `sql_windows.ps1`implementations requires the PowerShell feature.
The set of features available on the target is determined by the task runner. You can specify additional features for a target via `set_feature` or by adding `features` in the inventory. The task runner chooses the first implementation whose requirements are satisfied.
The following features are defined by default:

- `puppet-agent`: present if the target has the Puppet agent package installed
- `shell`: present if the target has a posix shell
- `powershell`: present if the target has PowerShell

Sharing executables

Multiple task implementations can refer to the same executable file.
Executables can access the `_task` metaparameter, which contains the task name. For example, the following creates the tasks `service::stop` and `service::start`, which live in the executable but appear as two separate tasks.

`myservice/tasks/init.rb`

```
#!/usr/bin/env ruby
require 'json'

params = JSON.parse(STDIN.read)
action = params['action'] || params['_task']
if ['start',  'stop'].include?(action)
  `systemctl #{params['_task']} #{params['service']}`
End
```

myservice/tasks/start.json

```
{
  "description": "Start a service",
  "parameters": {
    "service": {
      "type": "String",
      "description": "The service to start"
    }
  },
  "implementations": [
    {"name": "init.rb"}
  ]
}
```

myservice/tasks/stop.json

```
{
  "description": "Stop a service",
  "parameters": {
    "service": {
      "type": "String",
      "description": "The service to stop"
    }
  },
  "implementations": [
    {"name": "init.rb"}
  ]
}
```

Sharing task code

Multiple tasks can share common files between them. Tasks can additionally pull library code from other modules.
To create a task that includes additional files pulled from modules, include the files property in your metadata as an array of paths. A path consists of:

- the module name
- one of the following directories within the module:
  - `files` — Most helper files. This prevents the file from being treated as a task or added to thePuppet Ruby loadpath.
  - `tasks` — Helper files that can be called as tasks on their own.
  - `lib` — Ruby code that might be reused by types, providers, or Puppet functions.

- the remaining path to a file or directory; directories must include a trailing slash `/`
  All path separators must be forward slashes. An example would be `stdlib/lib/puppet/`.
  The `files` property can be included both as a top-level metadata property, and as a property of an implementation, for example:

```
{
  "implementations": [
    {"name": "sql_linux.sh", "requirements": ["shell"], "files":
["mymodule/files/lib.sh"]},
    {"name": "sql_windows.ps1", "requirements": ["powershell"],
"files": ["mymodule/files/lib.ps1"]}
  ],
  "files": ["emoji/files/emojis/"]
}
```

  When a task includes the `files` property, all files listed in the top-level property and in the specific implementation chosen for a target are copied to a temporary directory on that target. The directory structure of the specified files is preserved such that paths specified with the `files` metadata option are available to tasks prefixed with `_installdir`. The task executable itself is located in its module location under the `_installdir` as well, so other files can be found at `../../mymodule/files/`relative to the task executable's location.
  For example, you can create a task and metadata in a module at `~/.puppetlabs/bolt/site-modules/mymodule/tasks/task.{json,rb}`.
  Metadata

```
{
  "files": ["multi_task/files/rb_helper.rb"]
}
```

  File resource

```
multi_task/files/rb_helper.rb

def useful_ruby
  { helper: "ruby" }
End
```

  Task

```
#!/usr/bin/env ruby
require 'json'

params = JSON.parse(STDIN.read)
require_relative File.join(params['_installdir'], 'multi_task',
'files', 'rb_helper.rb')
# Alternatively use relative path
```

```
# require_relative File.join(__dir__, '..', '..', 'multi_task',
'files', 'rb_helper.rb')
 puts useful_ruby.to_json
```

Output

```
Started on localhost...
Finished on localhost:
  {
    "helper": "ruby"
  }
Successful on 1 node: localhost
Ran on 1 node in 0.12 seconds
```

Task helpers

To help with writing tasks, Bolt includes python_task_helper and ruby_task_helper. It also makes a useful demonstration of including code from another module.

Python example

Create task and metadata in a module at `~/.puppetlabs/bolt/site-modules/mymodule/tasks/task.{json,py}`.

Metadata

```
{
  "files": ["python_task_helper/files/task_helper.py"],
  "input_method": "stdin"
}
```

Task

```
#!/usr/bin/env python
import os, sys
sys.path.append(os.path.join(os.path.dirname(__file__), '..', '..',
'python_task_helper', 'files'))
from task_helper import TaskHelper

class MyTask(TaskHelper):
  def task(self, args):
    return {'greeting': 'Hi, my name is '+args['name']}

if __name__ == '__main__':
    MyTask().run()
```

Output

```
$ bolt task run mymodule::task -n localhost name='Julia'
Started on localhost...
Finished on localhost:
  {
    "greeting": "Hi, my name is Julia"
  }
Successful on 1 node: localhost
```

```
Ran on 1 node in 0.12 seconds
```

Ruby example

Create task and metadata in a new module at `~/.puppetlabs/bolt/site-modules/mymodule/tasks/mytask.{json,rb}`.

```
{
  "files": ["ruby_task_helper/files/task_helper.rb"],
  "input_method": "stdin"
}
```

Task

```
#!/usr/bin/env ruby
require_relative '../../ruby_task_helper/files/task_helper.rb'

class MyTask < TaskHelper
  def task(name: nil, **kwargs)
    { greeting: "Hi, my name is #{name}" }
  end
end


MyTask.run if __FILE__ == $0
```

Output

```
$ bolt task run mymodule::mytask -n localhost name="Robert'); DROP TABLE Students;--"
Started on localhost...
Finished on localhost:
  {
    "greeting": "Hi, my name is Robert'); DROP TABLE Students;--"
  }
Successful on 1 node: localhost
Ran on 1 node in 0.12 seconds
```

Writing remote tasks

Some targets are hard or impossible to execute tasks on directly. In these cases, you can write a task that runs on a proxy target and remotely interacts with the real target.

For example, a network device might have a limited shell environment or a cloud service might be driven only by HTTP APIs. By writing a remote task, Bolt allows you to specify connection information for remote targets in their inventory file and injects them into the `_target` metaparam.

This example shows how to write a task that posts messages to Slack and reads connection information from `inventory.yaml`:

```
#!/usr/bin/env ruby
# modules/slack/tasks/message.rb

require 'json'
require 'net/http'

params = JSON.parse(STDIN.read)
# the slack API token is passed in from inventory
token = params['_target']['token']

uri = URI('https://slack.com/api/chat.postMessage')
http = Net::HTTP.new(uri.host, uri.port)
http.use_ssl = true

req = Net::HTTP::Post.new(uri, 'Content-type' => 'application/json')
req['Authorization'] = "Bearer #{params['_target']['token']}"
req.body = { channel: params['channel'], text: params['message']
}.to_json

resp = http.request(req)

puts resp.body
```

To prevent accidentally running a normal task on a remote target and breaking its configuration, Boltwon't run a task on a remote target unless its metadata defines it as remote:

```
{
  "remote": true
}
```

Add Slack as a remote target in your inventory file:

```
---
nodes:
  - name: my_slack
    config:
      transport: remote
      remote:
     token: <SLACK_API_TOKEN>
```

Finally, make `my_slack` a target that can run the `slack::message`:

```
bolt task run slack::message --nodes my_slack message="hello"
channel=<slack channel id>
```

Defining parameters in tasks

Allow your task to accept parameters 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 asPowerShell 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 inPowerShell. 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 because parameters with undefined values (`nil`, `undef`) passed as environment variables have the `String` value `null`. For more information, see Structured input and output.
To add parameters to your task as environment variables, pass the argument prefixed with the Puppettask prefix `PT_` .
For example, to add a `message` parameter to your task, read it from the environment in task code as `PT_message`. When the user runs the task, they can 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 easier to write your task in PowerShell and use named arguments. By default tasks with a `.ps1` extension use PowerShell standard argument handling.
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 Puppetagent must be installed on the target nodes.

Returning errors in tasks

To return a detailed error message if your task fails, include an `Error` object in the task's result.

When a task exits non-zero, the task runner checks for an error key (`_error`). If one is not present, the task runner generates a generic error and adds it to the result. If there is no text on `stdout` but text is present on `stderr`, the `stderr` text is included in the message.

```
{ "_error": {
    "msg": "Task exited 1:\nSomething on stderr",
    "kind": "puppetlabs.tasks/task-error",
    "details": { "exitcode": 1 }
}
```

An error object includes the following keys:

msg
: A human readable string that appears in the UI.

kind
: A standard string for machines to handle. You may share kinds between your modules or namespace kinds per module.

details
: An object of structured data about the tasks.

Tasks can provide more details about the failure by including their own error object in the result at `_error`.

```
#!/opt/puppetlabs/puppet/bin/ruby

require 'json'

begin
  params = JSON.parse(STDIN.read)
```

```
  result = {}
  result['result'] = params['dividend'] / params['divisor']

rescue ZeroDivisionError
  result[:_error] = { msg: "Cannot divide by zero",
                      # namespace the error to this module
                      kind: "puppetlabs-example_modules/dividebyzero",
                      details: { divisor: divisor },
                    }
rescue Exception => e
  result[:_error] = { msg: e.message,
                      kind: "puppetlabs-example_modules/unknown",
                      details: { class: e.class.to_s },
                    }
end

puts result.to_json
```

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, consider using 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 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 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
```

If your task accepts input on `stdin` it should specify `"input_method": "stdin"` in its `metadata.json` file, or it may not work with sudo for some users.

Returning structured output

To return structured data from your task, print only a single JSON object to `stdout` in your task.
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)
```

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 add logic in your script to check for parameters in environment variables.
If the script is already installed on the target nodes, you can write a task that wraps the script. In the task, read 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 so 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 read them instead of arguments.
Warning: For any tasks that you intend to use with PE and assign RBAC permissions, make sure the script safely handles parameters or validate them to prevent shell injection vulnerabilities.
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
```

Wrapping an existing script

If a script is not already installed on targets and you don't want to edit it, for example if it's a script someone else maintains, you can wrap the script in a small task without modifying it.

Warning: For any tasks that you intend to use with PE and assign RBAC permissions, make sure the script safely handles parameters or validate them to prevent shell injection vulnerabilities.

Given a script, `myscript.sh`, that accepts 2 positional args, `filename` and `version`:

1. Copy the script to the module's `files/` directory.
2. Create a metadata file for the task that includes the parameters and file dependency.

```
{
     "input_method": "environment",
     "parameters": {
       "filename": { "type": "String[1]" },
       "version": { "type": "String[1]" }
     },
     "files": [ "script_example/files/myscript.sh" ]
   }
```

3 Create a small wrapper task that reads environment variables and calls the task.

```
#!/usr/bin/env bash
set -e

script_file="$PT__installdir/script_example/files/myscript.sh"
# If this task is going to be run from windows nodes the wrapper must
make sure it's exectutable
chmod +x $script_file
commandline=("$script_file" "$PT_filename" "$PT_version")
# If the stderr output of the script is important redirect it to
stdout.
"${commandline[@]}" 2>&1
```

Supporting no-op in tasks

Tasks 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 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. No-op is supported only in Puppet Enterprise.

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 or the task runner will refuse to run the task in noop mode.

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):
  error = {
      "_error": {
          "kind": "file_error",
          "msg": msg,
          "details": {},
      }
  }
  return error

try:
  if noop:
    path = os.path.abspath(os.path.join(filename, os.pardir))
    file_exists = os.access(filename, os.F_OK)
    file_writable = os.access(filename, os.W_OK)
    path_writable = os.access(path, os.W_OK)
```

```
    if path_writable == False:
      exitcode = 1
      result = make_error("Path %s is not writable" % path)
    elif file_exists == True and file_writable == False:
      exitcode = 1
      result = make_error("File %s is not writable" % filename)
    else:
      result = { "success": True , '_noop': True }
  else:
    with open(filename, 'w') as fh:
      fh.write(content)
      result = { "success": True }
except Exception as e:
  exitcode = 1
  result = make_error("Could not open file %s: %s" % (filename,
str(e)))
print(json.dumps(result))
exit(exitcode)
```

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. Specify task metadata in a JSON file with the naming convention `<TASKNAME>.json` . Place this file in the module's `./tasks`folder along with your task file.

For example, the module `puppetlabs-mysql` includes the `mysql::sql` task with the metadata file, `sql.json`.

```
{
  "description": "Allows you to execute arbitrary SQL",
  "input_method": "stdin",
  "parameters": {
    "database": {
      "description": "Database to connect to",
      "type": "Optional[String[1]]"
    },
    "user": {
      "description": "The user",
      "type": "Optional[String[1]]"
    },
    "password": {
      "description": "The password",
      "type": "Optional[String[1]]",
      "sensitive": true
    },
     "sql": {
      "description": "The SQL you want to execute",
      "type": "String[1]"
    }
  }
}
```

Adding parameters to metadata

To document and validate task parameters, add the parameters to the task metadata as JSON object, `parameters`.

If a task includes `parameters` in its metadata, the task runner rejects any parameters input to the task that 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, the following code in a metadata file describes a `provider` parameter:

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

Define sensitive parameters

You can define task parameters as sensitive, for example, passwords and API keys. These values are masked when they appear in logs and API responses. When you want to view these values, set the log file to `level: debug`.

To define a parameter as sensitive within the JSON metadata, add the `"sensitive": true` property.

```
{
  "description": "This task has a sensitive property denoted by its metadata",
  "input_method": "stdin",
  "parameters": {
    "user": {
      "description": "The user",
      "type": "String[1]"
    },
    "password": {
      "description": "The password",
      "type": "String[1]",
      "sensitive": true
    }
  }
}
```

Task metadata reference

The following table shows task metadata keys, values, and default values.

Task metadata

Metadata key	Description	Value	Default
"description"	A description of what the task does.	String	None

Metadata key	Description	Value	Default
"input_method"	What input method the task runner should use to pass parameters to the task.	`environment` `stdin` `powershell`	Both `environment`and `stdin`unless `.ps1`task in which case `powershell`
"parameters"	The parameters or input the task accepts listed with a puppet type string and optional description. See adding parameters to metadata for usage information.	Array of objects describing each parameter	None
"puppet_task_version"	The version of the spec used.	Integer	1 (This is the only valid value.)
"supports_noop"	Whether the task supports no-op mode. Required for the task to accept the `--noop` option on the command line.	Boolean	False
"implementations"	A list of task implementations and the requirements used to select one to run. See Cross-platform tasks for usage information.	Array of Objects describing each implementation	None
"files"	A list of files to be provided when running the task, addressed by module. See Sharing task code for usage information.	Array of Strings	None
"private"	Do not display task by default when listing for UI.	Boolean	False

Task metadata types

Task metadata can accept most Puppet data types.

Common task data types

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).

Type	Description
`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 dependin on input.
`Optional[String[1]]`	Optional makes the parameter optional and permits null values. Tasks have 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
`Boolean`	Accepts Boolean values.

Related information

- Data type syntax

Specifying parameters

Parameters for tasks can be passed to the `bolt` command as CLI arguments or as a JSON hash.

To pass parameters individually to your task or plan, specify the parameter value on the command line in the format `parameter=value`. Pass multiple parameters as a space-separated list. Bolt attempts to parse each parameter value as JSON and compares that to the parameter type specified by the task or plan. If the parsed value matches the type, it is used; otherwise, the original string is used.

For example, to run the `mysql::sql` task to show tables from a database called `mydatabase`:

```
bolt task run mysql::sql database=mydatabase sql="SHOW TABLES" --nodes
neptune --modules ~/modules
```

To pass a string value that is valid JSON to a parameter that would accept both quote the string. For example to pass the string `true` to a parameter of type `Variant[String, Boolean]` use `'foo="true"'`. To pass a String value wrapped in `"` quote and escape it `'string="\"val\""'`. Alternatively, you can specify parameters as a single JSON object with the `--params` flag, passing either a JSON object or a path to a parameter file.

To specify parameters as JSON, use the parameters flag followed by the JSON: `--params '{"name": "openssl"}'`
To set parameters in a file, specify parameters in JSON format in a file, such as `params.json`. For example, create a `params.json` file that contains the following JSON:

```
{
  "name":"openssl"
}
```

Then specify the path to that file (starting with an at symbol, `@`) on the command line with the parameters flag: `--params @params.json`

Writing plans

Plans allow you to run more than one task with a single command, compute values for the input to a task, process the results of tasks, or make decisions based on the result of running a task.

Write plans in the Puppet language, giving them a `.pp` extension, and place them in the module's `/plans` directory.

Plans can use any combination of Bolt functions or built-in Puppet functions.

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.

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.

Avoid giving 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 be a reserved word.
- 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 each 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.

The following example shows node parameters specified as data type `TargetSpec`. This allows this parameter to be passed as a single URL, comma-separated URL list, Target data type, or Array of either. For more information about these data types, see the common data types table in the related metadata type topic.

This allows the user to pass, for each parameter, either a 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,
  TargetSpec  $frontends,
  TargetSpec  $backends,
) {

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

To execute this plan from the command line, pass the parameters as `parameter=value`. The `Targetspec` accepts either an array as json or a comma separated string of target names.

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

Parameters that are passed to the `run_*` plan functions are serialized to JSON.

To illustrate this concept, consider this plan:

```
plan test::parameter_passing (
  TargetSpec $nodes,
  Optional[String[1]] $example_nul = undef,
) {
  return run_task('test::demo_undef_bash', $nodes, example_nul =>
$example_nul)
     }
```

The default value of `$example_nul` is `undef`. The plan calls the `test::demo_undef_bash` with the `example_nul` parameter. The implementation of the `demo_undef_bash.sh` task is:

```
#!/bin/bash
example_env=$PT_example_nul
echo "Environment: $PT_example_nul"
echo "Stdin:"
     cat -
```

By default, the task expects parameters passed as a JSON string on stdin to be accessible in prefixed environment variables.

Consider the output of running the plan against localhost:

```
bolt@bolt: bolt plan run test::parameter_passing -n localhost
Starting: plan test::parameter_passing
Starting: task test::demo_undef_bash on localhost
Finished: task test::demo_undef_bash with 0 failures in 0.0 sec
Finished: plan test::parameter_passing in 0.01 sec
Finished on localhost:
  Environment: null
  Stdin:
  {"example_nul":null,"_task":"test::demo_undef_bash"}
  {
  }
Successful on 1 node: localhost
     Ran on 1 node
```

The parameters `example_nul` and `_task` metadata are passed to the task as a JSON string over stdin.

Similarly, parameters are made available to the task as environment variables where the name of the parameter is converted to an environment variable prefixed with `PT_`. The prefixed environment variable points to the `String` representation in `JSON` format of the parameter value. So, the `PT_example_nul` environment variable has the value of `null` of type `String`.

Related information

- Task metadata types

Returning results from plans

Use plans to return results that you can use in other plans or save for use outside of Bolt.

Plans, unlike functions, are primarily run for side effects but they can optionally return a result. To return a result from a plan use the `return` function. Any plan that does not call the `return` function returns `undef`.

```
plan return_result(
  $nodes
) {
  return run_task('mytask', $nodes)
}
```

The result of a plan must match the `PlanResult` type alias. This roughly includes JSON types as well as the Plan language types which have well defined JSON representations in Bolt.

- `Undef`
- `String`
- `Numeric`
- `Boolean`
- `Target`
- `Result`
- `ResultSet`
- `Error`
- `Array` with only `PlanResult`
- Hash with `String` keys and `PlanResult` values

or

```
Variant[Data, String, Numeric, Boolean, Error, Result, ResultSet,
Target, Array[Boltlib::PlanResult], Hash[String, Boltlib::PlanResult]]
```

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'
    )
  }
```

Success and failure in plans

Indicators that a plan has run successfully or failed.

Any plan that completes execution without an error is considered successful. The `bolt` command exits 0 and any calling plans continue execution. If any calls to `run_` functions fail without `_catch_errors` then the plan halts execution and is considered a failure. Any calling plans also halt until a `run_plan` call with `_catch_errors` is reached. If one isn't reached, the `bolt` command performs an exit 2. When writing a plan if you have reason to believe it has failed, you can fail the plan with the `fail_plan` function. This causes the bolt command to exit 2 and prevents calling plans executing any further, unless `run_plan` was called with `_catch_errors`.

Failing plans

If `upload_file`, `run_command`, `run_script`, or `run_task` are called without the `_catch_errors` option and they fail on any nodes, the plan itself fails. To fail a plan directly call the `fail_plan` function. Create a new error with a message and include the kind, details, or issue code, or pass an existing error to it.

```
fail_plan('The plan is failing', 'mymodules/pear-shaped',
{'failednodes' => $result.error_set.names})
# or
fail_plan($errorobject)
```

Responding to errors in plans

When you call `run_plan` with `_catch_errors` or call the `error` method on a result, you may get an error.

The `Error` data type includes:

- `msg`: The error message string.
- `kind`: A string that defines the kind of error similar to an error class.
- `details`: A hash with details about the error from a task or from information about the state of a plan when it fails, for example, `exit_code` or `stack_trace`.
- `issue_code`: A unique code for the message that can be used for translation.

Use the `Error` data type in a case expression to match against different kind of errors. To recover from certain errors, while failing on or ignoring others, set up your plan to include conditionals based on errors that occur while your plan runs. For example, you can set up a plan to retry a task when a timeout error occurs, but to fail when there is an authentication error.

Below, the first plan continues whether it succeeds or fails with a `mymodule/not-serious` error. Other errors cause the plan to fail.

```
plan mymodule::handle_errors {
  $result = run_plan('mymodule::myplan', '_catch_errors' => true)
  case $result {
    Error['mymodule/not-serious'] : {
      notice("${result.message}")
    }
    Error : { fail_plan($result) } }
  run_plan('mymodule::plan2')
}
```

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, such as `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 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.
- Most Puppet settings are empty and not-configurable when using Bolt.
- 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 `ResultSet`. For each node that the execution takes place on, this object contains a `Result` object. The apply action returns a `ResultSet` containing `ApplyResult` objects.

  A `ResultSet` 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.
- `first()`: The first `Result` object, useful to unwrap single results.
- `find(String $target_name)`: Look up the `Result` for a specific target.
- `error_set()`: A `ResultSet` containing only the results of failed nodes.
- `ok_set()`: A `ResultSet` containing only the successful results.
- `targets()`: An array of all the `Target` objects from every `Result` in the set.
- `ok()`: `Boolean` that is the same as `error_nodes.empty`.

  A `Result` has the following methods:

- `value()`: The hash containing the value of the `Result`.
- `target()`: The `Target` object that the `Result` is from.
- `error()`: An `Error` object constructed from the `_error` in the value.
- `message()`: The `_output` key from the value.
- `ok()`: Returns `true` if the `Result` was successful.
- `[]`: Accesses the value hash directly.

  An `ApplyResult` has the following methods:

- `report()`: The hash containing the Puppet report from the application.
- `target()`: The `Target` object that the `Result` is from.
- `error()`: An `Error` object constructed from the `_error` in the value.
- `ok()`: Returns `true` if the `Result` was successful.

  An instance of `ResultSet` is `Iterable` as if it were an `Array[Variant[Result, ApplyResult]]`so that iterative functions such as `each`, `map`, `reduce`, or `filter` work directly on the ResultSet returning each result.

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

```
$r = run_task('sometask', ..., '_catch_errors' => true)
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 feedback about the result of a task:

```
$r = run_task('sometask', ..., '_catch_errors' => true)
$r.each |$result| {
  $node = $result.target.name
  if $result.ok {
    notice("${node} returned a value: ${result.value}")
  } else {
    notice("${node} errored with a message: ${result.error.message}")
  }
}
```

Passing sensitive data to tasks

Task parameters defined as sensitive are masked when they appear in plans.

You define a task parameter as sensitive with the metadata property `"sensitive": true`. When a task runs, the values for these sensitive parameters are masked.

```
run_task('task_with_secrets', ..., password => '$ecret!')
```

Working with the sensitive function

In Puppet you use the `Sensitive` function to mask data in output logs. Since plans are written in Puppet DSL you can use this type freely.
The `run_task()` function does not allow parameters of `Sensitive` function to be passed. When you need to pass a sensitive value to a task, you must unwrap it prior to calling `run_task()`.

```
$pass = Sensitive('$ecret!')
run_task('task_with_secrets', ..., password => $pass.unwrap)
```

Related information

- Adding parameters to metadata

Target objects

The `Target` object represents a node and its specific connection options.

The state of a target is stored in the inventory for the duration of a plan allowing you to collect facts or set vars for a target and retrieve them later. You can get a printable representation via the `name`function, as well as access components of the target: `protocol, host, port, user, password`.

TargetSpec
The execution function takes a parameter with the type alias TargetSpec. This alias accepts the pattern strings allowed by `--nodes`, a single Target object, or an Array of Targets and node patterns. Plans that accept a set of targets as a parameter should generally use this type to interact cleanly with the CLI and other plans. To operate on individual nodes, resolve it to a list via `get_targets`. For example to loop over each node in a plan accept a `TargetSpec` argument but call `get_targets` on it before looping.

```
plan loop(TargetSpec $nodes) {
  get_targets($nodes).each |$target| {
    run_task('my_task', $target)
  }
}
```

If your plan accepts a single `TargetSpec` parameter you can call that parameter `nodes` so that it can be specified with the `--nodes` flag from the command line.

Variables and facts on targets
When Bolt runs, it loads transport config values, variables, and facts from the inventory. These can be accessed with the `$target.facts()` and `$target.vars()` functions. During the course of a plan, you can update the facts or variables for any target. Facts usually come from running `facter` or another fact collection application on the target or from a fact store like PuppetDB. Variables are computed externally or assigned directly.

Set variables in a plan using `$target.set_var`:

```
plan vars(String $host) {
     $target = get_targets($host)[0]
     $target.set_var('newly_provisioned', true)
     $targetvars = $target.vars
     run_command("echo 'vars for ${host}: ${$targetvars}'", $host)
}
```

Or set variables in the inventory file using the `vars` key at the group level.

```
groups:
  - name: my_nodes
    nodes:
      - localhost
    vars:
      operatingsystem: windows
    config:
      transport: ssh
```

Collect facts from the targets

The facts plan connects to the target and discovers facts. It then stores these facts on the targets in the inventory for later use.

The methods used to collect facts:

- On `ssh` targets, it runs a Bash script.
- On `winrm` targets, it runs a PowerShell script.
- On `pcp` or targets where the Puppet agent is present, it runs Facter.

This example collects facts with the facts plan and then uses those facts to decide which task to run on the targets.

```
plan run_with_facts(TargetSpec $nodes) {
  # This collects facts on nodes and update the inventory
  run_plan(facts, nodes => $nodes)

  $centos_nodes = get_targets($nodes).filter |$n| {
$n.facts['os']['name'] == 'CentOS' }
  $ubuntu_nodes = get_targets($nodes).filter |$n| {
$n.facts['os']['name'] == 'Ubuntu' }
  run_task(centos_task, $centos_nodes)
  run_task(ubuntu_task, $ubuntu_nodes)
}
```

Collect facts from PuppetDB

When targets are running a Puppet agent and sending facts to PuppetDB, you can use the `puppetdb_fact` plan to collect facts for them. This example collects facts with the `puppetdb_fact` plan, and then uses those facts to decide which task to run on the targets. You must configure the PuppetDB client before you run it.

```
plan run_with_facts(TargetSpec $nodes) {
  # This collects facts on nodes and update the inventory
  run_plan(puppetdb_fact, nodes => $nodes)

  $centos_nodes = get_targets($nodes).filter |$n| {
$n.facts['os']['name'] == 'CentOS' }
  $ubuntu_nodes = get_targets($nodes).filter |$n| {
$n.facts['os']['name'] == 'Ubuntu' }
  run_task(centos_task, $centos_nodes)
  run_task(ubuntu_task, $ubuntu_nodes)
}
```

Related information

- Connecting Bolt to PuppetDB

Plan logging

Set up log files to record certain events that occur when you run plans.

Puppet log functions

To generate log messages from a plan, use the Puppet log function that corresponds to the level you want to track: `error`, `warn`, `notice`, `info`, or `debug`. The default log level for Bolt is `notice` but you can set it to `info` with the `--verbose` flag or `debug` with the `--debug` flag.

Default action logging

Bolt logs actions that a plan takes on targets through the `upload_file`, `run_command`, `run_script`, or `run_task` functions. By default it logs a notice level message when an action starts and another when it completes. If you pass a description to the function, that is used in place of the generic log message.

```
run_task(my_task, $targets, "Better description", param1 => "val")
```

If your plan contains many small actions you may want to suppress these messages and use explicit calls to the Puppet log functions instead. This can be accomplished by wrapping actions in a `without_default_logging` block which causes the action messages to be logged at info level instead of notice. For example to loop over a series of nodes without logging each action.

```
plan deploy( TargetSpec $nodes) {
  without_default_logging() || {
    get_targets($nodes).each |$node| {
      run_task(deploy, $node)
    }
  }
}
```

To avoid complications with parser ambiguity, always call `without_default_logging` with `()` and empty block args `||`.

```
without_default_logging() || { run_command('echo hi', $nodes) }
```

not

```
without_default_logging { run_command('echo hi', $nodes) }
```

puppetdb_query

You can use the `puppetdb_query` function in plans to make direct queries to PuppetDB. For example you can discover nodes from PuppetDB and then run tasks on them. You'll have to configure the puppetdb client before running it.

```
plan pdb_discover {
  $result = puppetdb_query("inventory[certname] { app_role ==
'web_server' }")
  # extract the certnames into an array
  $names = $result.map |$r| { $r["certname"] }
  # wrap in url. You can skip this if the default transport is pcp
  $nodes = $names.map |$n| { "pcp://${n}" }
  run_task('my_task', $nodes)
}
```