mold Documentation

Release 0.1

Matt Haggard

November 01, 2012

CONTENTS

The idea is to use existing standards and to stay out of the way. To that end, the master and minion consist of normal scripts that read stdin and write to stdout and stderr.

CHAPTER

ONE

SCRIPT INTERFACE

All scripts are expected to use the standard input/output file descriptors plus an optional logging control file descriptor (fd 3):

- `stdin` (0): input comes from stdin. Usually this will be a JSON document.
- `stdout` (1): output is written to stdout. Usually this will be a JSON document.
- `stderr` (2): errors and debugging are written to stderr. The script may write to stderr and still be considered successful. Success is determined solely by the exit code. Things written to stderr do NOT need to be JSON documents.
- `channel3` (3): Things written to this channel are passed through to the historian. It is expected that this channel will be used to upload log files, indicate steps in a process, label stdin/out/err for each spawned process, etc...

  A script should not depend on this file descriptor being available. So these two calls should have the same stdout, stderr and exit code given the same stdin:

  ```
  /bin/bash some_script
  /bin/bash some_script 3>/dev/null
  ```

Scripts must return 0 to indicate success and any other exit code to indicate failure.

CHAPTER

TWO

MASTER

On a master machine, there will be a master directory with a layout similar to this:

```
master/
    certs/
    actors/
        prescribe
        choreograph
```

2.1 `prescribe` script

This script accepts a fact document on stdin and produces a list of desired resources states for the minion identified by the given fact document.

2.2 `choreograph` script

Given a set of facts, a prescription and the current state of each resource in the prescription, this script produces a list of steps including:

1. desired resource states
2. one-time resource actions

CHAPTER

THREE

MINION

On a minion machine, there will be a minion directory with a layout similar to this:

```
minion/
    certs/
    facts/
        os
    resources/
        file
        cron
        user
        service
```

3.1 Fact scripts

The executable scripts in `minion/facts/` accept no arguments or stdin. They return facts about the system on stdout.

When a minion is asked for the facts of the system, all the scripts in the `minion/facts/` directory are run and combined into a single fact document. For instance, the output of `minion/facts/foo` might be:

```
{
    "cats": 10,
    "dogs": 20,
    "gorillas": "no gorillas"
}
```

This would be combined into the single fact document by using the filename `foo` as the key:

```
{
    "foo": {
        "cats": 10,
        "dogs": 20,
        "gorillas": "no gorillas"
    }
}
```

Adding custom facts is as simple as putting an executable file in `minion/facts/` that writes a fact document to stdout.

3.2 Resource scripts

The executable scripts in `minion/resources/` each define the way a resource is handled. They must accept as a first command line argument the action to be performed for that resource. For instance, to inspect the state of the file `/tmp/foo` you would do something like:

```
$ echo '{"path":"/tmp/foo"}' | minion/resources/file inspect
{
    "kind": "file",
    "path": "/tmp/foo",
    "exists": false
}
```

And to make `/tmp/foo` conform to an expected state, you could do:

```
$ cat | minion/resources/file conform
{
    "path": "/tmp/foo",
    "user": "joe",
    "src": "http://www.example.com/foo.png"
}
^D
```

Some resources support one-time actions (such as restarting a service). These are supported by using a custom command-line argument (in place of `inspect` or `conform`). To restart a service you might do:

```
$ cat | minion/resources/service restart
{
    "name": "sshd"
}
^D
```

To add a custom resource, put an executable file in `minion/resources/` that behaves as indicated above.

CHAPTER

FOUR

CHANNEL3 PROTOCOL

Channel3 is meant for getting all stdin, stdout, stderr and other logging/debugging information back to the historian.

Things written to the channel are encoded in JSON tuples wrapped in netstrings. Each tuple has 3 items:

1. Child process name or `null` if the current process
2. Key
3. Data

For instance, if I were indicating to my parent process that I received stdout from my child process (named `jim`), I would write this to the `log` fd:

```
57:["jim", "stdout", {"line": "This is a line of stdout\n"}],
```

4.1 Data format for various keys

4.1.1 stdout, stdin, stderr

```
{
    "type": "object",
    "properties": {
        "line": {
            "type": "string",
            "required": true,
            "description": "Line of data",
        },
        "encoding": {
            "type": "string",
            "required": false,
            "description": "Encoding of `line`; no encoding if not provided; options include `b64`"
        }
    }
}
```

For example:

```
('jim', 'stdout', {'line': 'this is a line\n'})
```

Or for binary data:

```
('joe', 'stderr', {'line': 'AAH/\n', 'encoding': 'base64'})
```

4.1.2 spawn

```
{
    "type": "object",
    "properties": {
        "path": {
            "type": "string"
        },
        "env": {
            "type": "object"
        },
        "args": {
            "type": "array"
        },
        "user": {
            "type": "string"
        },
        "group": {
            "type": "string"
        }
    }
}
```

For example:

```
('newchild', 'spawn', {
    'path': '/tmp/foo',
    'env': {
        'FOO': 'something',
        'USER': 'joe',
    },
    'args': ['cat', 'afile'],
    'user': 'joe',
    'group': 'joe',
})
```

4.1.3 exitcode

```
{
    "type": "integer",
}
```

For example:

```
('newchild', 'exitcode', 3)
```

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*