Fabric

Nov 30, 2018

Contents

1 Getting started **3**
- 1.1 Getting started . . . 3

2 Upgrading from 1.x **9**

3 Concepts **11**
- 3.1 Authentication . . . 11
- 3.2 Configuration . . . 12
- 3.3 Networking . . . 16

4 The `fab` CLI tool **19**
- 4.1 Command-line interface . . . 19

5 API **23**
- 5.1 `config` . . . 23
- 5.2 `connection` . . . 23
- 5.3 `exceptions` . . . 23
- 5.4 `executor` . . . 23
- 5.5 `group` . . . 23
- 5.6 `runners` . . . 23
- 5.7 `tasks` . . . 23
- 5.8 `testing` . . . 23
- 5.9 `transfer` . . . 24
- 5.10 `tunnels` . . . 24
- 5.11 `util` . . . 24

This site covers Fabric's usage & API documentation. For basic info on what Fabric is, including its public changelog & how the project is maintained, please see the main project website.

CHAPTER 1

Getting started

Many core ideas & API calls are explained in the tutorial/getting-started document:

1.1 Getting started

Welcome! This tutorial highlights Fabric's core features; for further details, see the links within, or the documentation index which has links to conceptual and API doc sections.

1.1.1 A note about imports

Fabric composes a couple of other libraries as well as providing its own layer on top; user code will most often import from the `fabric` package, but you'll sometimes import directly from `invoke` or `paramiko` too:

- Invoke implements CLI parsing, task organization, and shell command execution (a generic framework plus specific implementation for local commands.)
  - Anything that isn't specific to remote systems tends to live in Invoke, and it is often used standalone by programmers who don't need any remote functionality.
  - Fabric users will frequently import Invoke objects, in cases where Fabric itself has no need to subclass or otherwise modify what Invoke provides.
- Paramiko implements low/mid level SSH functionality - SSH and SFTP sessions, key management, etc.
  - Fabric mostly uses this under the hood; users will only rarely import from Paramiko directly.
- Fabric glues the other libraries together and provides its own high level objects too, e.g.:
  - Subclassing Invoke's context and command-runner classes, wrapping them around Paramiko-level primitives;
  - Extending Invoke's configuration system by using Paramiko's `ssh_config` parsing machinery;
  - Implementing new high-level primitives of its own, such as port-forwarding context managers. (These may, in time, migrate downwards into Paramiko.)

1.1.2 Run commands via Connections and `run`

The most basic use of Fabric is to execute a shell command on a remote system via SSH, then (optionally) interrogate the result. By default, the remote program's output is printed directly to your terminal, *and* captured. A basic example:

```
>>> from fabric import Connection
>>> c = Connection('web1')
>>> result = c.run('uname -s')
Linux
>>> result.stdout.strip() == 'Linux'
True
>>> result.exited
0
>>> result.ok
True
>>> result.command
'uname -s'
>>> result.connection
<Connection host=web1>
>>> result.connection.host
'web1'
```

Meet `Connection`, which represents an SSH connection and provides the core of Fabric's API, such as `run`. `Connection` objects need at least a hostname to be created successfully, and may be further parameterized by username and/or port number. You can give these explicitly via args/kwargs:

```
Connection(host='web1', user='deploy', port=2202)
```

Or by stuffing a `[user@]host[:port]` string into the `host` argument (though this is purely convenience; always use kwargs whenever ambiguity appears!):

```
Connection('deploy@web1:2202')
```

`Connection` objects' methods (like `run`) usually return instances of `invoke.runners.Result` (or subclasses thereof) exposing the sorts of details seen above: what was requested, what happened while the remote action occurred, and what the final result was.

Note: Many lower-level SSH connection arguments (such as private keys and timeouts) can be given directly to the SSH backend by using the connect_kwargs argument.

1.1.3 Superuser privileges via auto-response

Need to run things as the remote system's superuser? You could invoke the `sudo` program via `run`, and (if your remote system isn't configured with passwordless sudo) respond to the password prompt by hand, as below. (Note how we need to request a remote pseudo-terminal; most `sudo` implementations get grumpy at password-prompt time otherwise.)

```
>>> from fabric import Connection
>>> c = Connection('db1')
>>> c.run('sudo useradd mydbuser', pty=True)
[sudo] password:
<Result cmd='sudo useradd mydbuser' exited=0>
>>> c.run('id -u mydbuser')
```

(continues on next page)

(continued from previous page)

```
1001
<Result cmd='id -u mydbuser' exited=0>
```

Giving passwords by hand every time can get old; thankfully Invoke's powerful command-execution functionality includes the ability to auto-respond to program output with pre-defined input. We can use this for `sudo`:

```
>>> from invoke import Responder
>>> from fabric import Connection
>>> c = Connection('host')
>>> sudopass = Responder(
...     pattern=r'\[sudo\] password:',
...     response='mypassword\n',
... )
>>> c.run('sudo whoami', pty=True, watchers=[sudopass])
[sudo] password:
root
<Result cmd='sudo whoami' exited=0>
```

It's difficult to show in a snippet, but when the above was executed, the user didn't need to type anything; `mypassword` was sent to the remote program automatically. Much easier!

The `sudo` helper

Using watchers/responders works well here, but it's a lot of boilerplate to set up every time - especially as real-world use cases need more work to detect failed/incorrect passwords.

To help with that, Invoke provides a `Context.sudo` method which handles most of the boilerplate for you (as `Connection` subclasses `Context`, it gets this method for free.) `sudo` doesn't do anything users can't do themselves - but as always, common problems are best solved with commonly shared solutions.

All the user needs to do is ensure the `sudo.password` *configuration value* is filled in (via config file, environment variable, or `--prompt-for-sudo-password`) and `Connection.sudo` handles the rest. For the sake of clarity, here's an example where a library/shell user performs their own `getpass`-based password prompt:

```
>>> import getpass
>>> from fabric import Connection, Config
>>> sudo_pass = getpass.getpass("What's your sudo password?")
What's your sudo password?
>>> config = Config(overrides={'sudo': {'password': sudo_pass}})
>>> c = Connection('db1', config=config)
>>> c.sudo('whoami', hide='stderr')
root
<Result cmd="...whoami" exited=0>
>>> c.sudo('useradd mydbuser')
<Result cmd="...useradd mydbuser" exited=0>
>>> c.run('id -u mydbuser')
1001
<Result cmd='id -u mydbuser' exited=0>
```

We filled in the sudo password up-front at runtime in this example; in real-world situations, you might also supply it via the configuration system (perhaps using environment variables, to avoid polluting config files), or ideally, use a secrets management system.

1.1.4 Transfer files

Besides shell command execution, the other common use of SSH connections is file transfer; `Connection.put` and `Connection.get` exist to fill this need. For example, say you had an archive file you wanted to upload:

```
>>> from fabric import Connection
>>> result = Connection('web1').put('myfiles.tgz', remote='/opt/mydata/')
>>> print("Uploaded {0.local} to {0.remote}".format(result))
Uploaded /local/myfiles.tgz to /opt/mydata/
```

These methods typically follow the behavior of `cp` and `scp`/`sftp` in terms of argument evaluation - for example, in the above snippet, we omitted the filename part of the remote path argument.

1.1.5 Multiple actions

One-liners are good examples but aren't always realistic use cases - one typically needs multiple steps to do anything interesting. At the most basic level, you could do this by calling `Connection` methods multiple times:

```
from fabric import Connection
c = Connection('web1')
c.put('myfiles.tgz', '/opt/mydata')
c.run('tar -C /opt/mydata -xzvf /opt/mydata/myfiles.tgz')
```

You could (but don't have to) turn such blocks of code into functions, parameterized with a `Connection` object from the caller, to encourage reuse:

```
def upload_and_unpack(c):
    c.put('myfiles.tgz', '/opt/mydata')
    c.run('tar -C /opt/mydata -xzvf /opt/mydata/myfiles.tgz')
```

As you'll see below, such functions can be handed to other API methods to enable more complex use cases as well.

1.1.6 Multiple servers

Most real use cases involve doing things on more than one server. The straightforward approach could be to iterate over a list or tuple of `Connection` arguments (or `Connection` objects themselves, perhaps via `map`):

```
>>> from fabric import Connection
>>> for host in ('web1', 'web2', 'mac1'):
>>>     result = Connection(host).run('uname -s')
...     print("{}: {}".format(host, result.stdout.strip()))
...
...
web1: Linux
web2: Linux
mac1: Darwin
```

This approach works, but as use cases get more complex it can be useful to think of a collection of hosts as a single object. Enter `Group`, a class wrapping one-or-more `Connection` objects and offering a similar API; specifically, you'll want to use one of its concrete subclasses like `SerialGroup` or `ThreadingGroup`.

The previous example, using `Group` (`SerialGroup` specifically), looks like this:

```
>>> from fabric import SerialGroup as Group
>>> results = Group('web1', 'web2', 'mac1').run('uname -s')
>>> print(results)
<GroupResult: {
    <Connection 'web1'>: <CommandResult 'uname -s'>,
    <Connection 'web2'>: <CommandResult 'uname -s'>,
    <Connection 'mac1'>: <CommandResult 'uname -s'>,
}>
>>> for connection, result in results.items():
...     print("{0.host}: {1.stdout}".format(connection, result))
...
...
web1: Linux
web2: Linux
mac1: Darwin
```

Where `Connection` methods return single `Result` objects (e.g. `fabric.runners.Result`), `Group` methods return `GroupResult` - `dict`-like objects offering access to individual per-connection results as well as metadata about the entire run.

When any individual connections within the `Group` encounter errors, the `GroupResult` is lightly wrapped in a `GroupException`, which is raised. Thus the aggregate behavior resembles that of individual `Connection` methods, returning a value on success or raising an exception on failure.

1.1.7 Bringing it all together

Finally, we arrive at the most realistic use case: you've got a bundle of commands and/or file transfers and you want to apply it to multiple servers. You *could* use multiple `Group` method calls to do this:

```
from fabric import SerialGroup as Group
pool = Group('web1', 'web2', 'web3')
pool.put('myfiles.tgz', '/opt/mydata')
pool.run('tar -C /opt/mydata -xzvf /opt/mydata/myfiles.tgz')
```

That approach falls short as soon as logic becomes necessary - for example, if you only wanted to perform the copy-and-untar above when `/opt/mydata` is empty. Performing that sort of check requires execution on a per-server basis.

You could fill that need by using iterables of `Connection` objects (though this foregoes some benefits of using `Groups`):

```
from fabric import Connection
for host in ('web1', 'web2', 'web3'):
    c = Connection(host)
    if c.run('test -f /opt/mydata/myfile', warn=True).failed:
        c.put('myfiles.tgz', '/opt/mydata')
        c.run('tar -C /opt/mydata -xzvf /opt/mydata/myfiles.tgz')
```

Alternatively, remember how we used a function in that earlier example? You can go that route instead:

```
from fabric import SerialGroup as Group

def upload_and_unpack(c):
    if c.run('test -f /opt/mydata/myfile', warn=True).failed:
        c.put('myfiles.tgz', '/opt/mydata')
        c.run('tar -C /opt/mydata -xzvf /opt/mydata/myfiles.tgz')
```

(continues on next page)

(continued from previous page)

```
for connection in Group('web1', 'web2', 'web3'):
    upload_and_unpack(connection)
```

The only convenience this final approach lacks is a useful analogue to `Group.run` - if you want to track the results of all the `upload_and_unpack` call as an aggregate, you have to do that yourself. Look to future feature releases for more in this space!

1.1.8 Addendum: the `fab` command-line tool

It's often useful to run Fabric code from a shell, e.g. deploying applications or running sysadmin jobs on arbitrary servers. You could use regular Invoke tasks with Fabric library code in them, but another option is Fabric's own "network-oriented" tool, `fab`.

`fab` wraps Invoke's CLI mechanics with features like host selection, letting you quickly run tasks on various servers - without having to define `host` kwargs on all your tasks or similar.

Note: This mode was the primary API of Fabric 1.x; as of 2.0 it's just a convenience. Whenever your use case falls outside these shortcuts, it should be easy to revert to the library API directly (with or without Invoke's less opinionated CLI tasks wrapped around it).

For a final code example, let's adapt the previous example into a `fab` task module called `fabfile.py`:

```
from fabric import task

@task
def upload_and_unpack(c):
    if c.run('test -f /opt/mydata/myfile', warn=True).failed:
        c.put('myfiles.tgz', '/opt/mydata')
        c.run('tar -C /opt/mydata -xzvf /opt/mydata/myfiles.tgz')
```

Not hard - all we did was copy our temporary task function into a file and slap a decorator on it. `task` tells the CLI machinery to expose the task on the command line:

```
$ fab --list
Available tasks:

  upload_and_unpack
```

Then, when `fab` actually invokes a task, it knows how to stitch together arguments controlling target servers, and run the task once per server. To run the task once on a single server:

```
$ fab -H web1 upload_and_unpack
```

When this occurs, `c` inside the task is set, effectively, to `Connection("web1")` - as in earlier examples. Similarly, you can give more than one host, which runs the task multiple times, each time with a different `Connection` instance handed in:

```
$ fab -H web1,web2,web3 upload_and_unpack
```

CHAPTER 2

Upgrading from 1.x

Looking to upgrade from Fabric 1.x? See our detailed upgrade guide on the nonversioned main project site.

CHAPTER 3

Concepts

Dig deeper into specific topics:

3.1 Authentication

Even in the 'vanilla' OpenSSH client, authenticating to remote servers involves multiple potential sources for secrets and configuration; Fabric not only supports most of those, but has more of its own. This document outlines the available methods for setting authentication secrets.

Note: Since Fabric itself tries not to reinvent too much Paramiko functionality, most of the time configuring authentication values boils down to "how to set keyword argument values for `SSHClient.connect`", which in turn means to set values inside either the `connect_kwargs` *config* subtree, or the `connect_kwargs` keyword argument of `Connection`.

3.1.1 Private key files

Private keys stored on-disk are probably the most common auth mechanism for SSH. Fabric offers multiple methods of configuring which paths to use, most of which end up merged into one list of paths handed to `SSHClient.connect(key_filename=[...])`, in the following order:

- If a `key_filename` key exists in the `connect_kwargs` argument to `Connection`, they come first in the list. (This is basically the "runtime" option for non-CLI users.)
- The config setting `connect_kwargs.key_filename` can be set in a number of ways (as per the *config docs*) including via the `--identity` CLI flag (which sets the `overrides` level of the config; so when this flag is used, key filename values from other config sources will be overridden.) This value comes next in the overall list.
- Using an *ssh_config* file with `IdentityFile` directives lets you share configuration with other SSH clients; such values come last.

Encryption passphrases

If your private key file is protected via a passphrase, it can be supplied in a handful of ways:

- The `connect_kwargs.passphrase` config option is the most direct way to supply a passphrase to be used automatically.

  Note: Using actual on-disk config files for this type of material isn't always wise, but recall that the *configuration system* is capable of loading data from other sources, such as your shell environment or even arbitrary remote databases.

- If you prefer to enter the passphrase manually at runtime, you may use the command-line option `--prompt-for-passphrase`, which will cause Fabric to interactively prompt the user at the start of the process, and store the entered value in `connect_kwargs.passphrase` (at the 'overrides' level.)

3.1.2 Private key objects

Instantiate your own `PKey` object (see its subclasses' API docs for details) and place it into `connect_kwargs.pkey`. That's it! You'll be responsible for any handling of passphrases, if the key material you're loading (these classes can load from file paths or strings) is encrypted.

3.1.3 SSH agents

By default (similar to how OpenSSH behaves) Paramiko will attempt to connect to a running SSH agent (Unix style, e.g. a live `SSH_AUTH_SOCK`, or Pageant if one is on Windows). This can be disabled by setting `connect_kwargs.allow_agent` to `False`.

3.1.4 Passwords

Password authentication is relatively straightforward:

- You can configure it via `connect_kwargs.password` directly.
- If you want to be prompted for it at the start of a session, specify `--prompt-for-login-password`.

3.1.5 GSSAPI

Fabric doesn't provide any extra GSSAPI support on top of Paramiko's existing connect-time parameters (see e.g. `gss_kex`/`gss_auth`/`gss_host`/etc in `SSHClient.connect`) and the modules implementing the functionality itself (such as `paramiko.ssh_gss`.) Thus, as usual, you should be looking to modify the `connect_kwargs` configuration tree.

3.2 Configuration

3.2.1 Basics

The heart of Fabric's configuration system (as with much of the rest of Fabric) relies on Invoke functionality, namely `invoke.config.Config` (technically, a lightweight subclass, `fabric.config.Config`). For practical details on what this means re: configuring Fabric's behavior, please see Invoke's configuration documentation.

The primary differences from that document are as follows:

- The configuration file paths sought are all named `fabric.*` instead of `invoke.*` - e.g. `/etc/fabric.yml` instead of `/etc/invoke.yml`, `~/.fabric.py` instead of `~/.invoke.py`, etc.
- In addition to Invoke's own default configuration values, Fabric merges in some of its own, such as the fact that SSH's default port number is 22. See *Default configuration values* for details.
- Fabric has facilities for loading SSH config files, and will automatically create (or update) a configuration subtree on a per `Connection` basis, loaded with the interpreted SSH configuration for that specific host (since an SSH config file is only ever useful via such a lens). See *Loading and using ssh_config files*.
- Fabric plans to offer a framework for managing per-host and per-host-collection configuration details and overrides, though this is not yet implemented (it will be analogous to, but improved upon, the `env.hosts` and `env.roles` structures from Fabric 1.x).
  - This functionality will supplement that of the SSH config loading described earlier; we expect most users will prefer to configure as much as possible via an SSH config file, but not all Fabric settings have `ssh_config` analogues, nor do all use cases fit neatly into such files.

3.2.2 Default configuration values

Overrides of Invoke-level defaults

- `run.replace_env`: `True`, instead of `False`, so that remote commands run with a 'clean', empty environment instead of inheriting a copy of the current process' environment.

  This is for security purposes: leaking local environment data remotely by default would be unsanitary. It's also compatible with the behavior of OpenSSH.

  See also:

  The warning under `paramiko.channel.Channel.set_environment_variable`.

Extensions to Invoke-level defaults

- `runners.remote`: In Invoke, the `runners` tree has a single subkey, `local` (mapping to `Local`). Fabric adds this new subkey, `remote`, which is mapped to `Remote`.

New default values defined by Fabric

Note: Most of these settings are also available in the constructor of `Connection`, if they only need modification on a per-connection basis.

Warning: Many of these are also configurable via *ssh_config files*. **Such values take precedence over those defined via the core configuration**, so make sure you're aware of whether you're loading such files (or *disable them to be sure*).

- `connect_kwargs`: Keyword arguments (`dict`) given to `SSHClient.connect` when `Connection` performs that method call. This is the primary configuration vector for many SSH-related options, such as selecting private keys, toggling forwarding of SSH agents, etc. Default: `{}`.

- `forward_agent`: Whether to attempt forwarding of your local SSH authentication agent to the remote end. Default: `False` (same as in OpenSSH.)
- `gateway`: Used as the default value of the `gateway` kwarg for `Connection`. May be any value accepted by that argument. Default: `None`.
- `load_ssh_configs`: Whether to automatically seek out *SSH config files*. When `False`, no automatic loading occurs. Default: `True`.
- `port`: TCP port number used by `Connection` objects when not otherwise specified. Default: `22`.
- `inline_ssh_env`: Boolean serving as global default for the value of `Connection`'s `inline_ssh_env` parameter; see its docs for details. Default: `False`.
- `ssh_config_path`: Runtime SSH config path; see *Loading and using ssh_config files*. Default: `None`.
- `timeouts`: Various timeouts, specifically:
  - `connect`: Connection timeout, in seconds; defaults to `None`, meaning no timeout / block forever.
- `user`: Username given to the remote `sshd` when connecting. Default: your local system username.

3.2.3 Loading and using `ssh_config` files

How files are loaded

Fabric uses Paramiko's SSH config file machinery to load and parse `ssh_config`-format files (following OpenSSH's behavior re: which files to load, when possible):

- An already-parsed `SSHConfig` object may be given to `Config.__init__` via its `ssh_config` keyword argument; if this value is given, no files are loaded, even if they exist.
- A runtime file path may be specified via configuration itself, as the `ssh_config_path` key; such a path will be loaded into a new `SSHConfig` object at the end of `Config.__init__` and no other files will be sought out.
  - It will be filled in by the `fab` CLI tool if the `--ssh-config` flag is given.
- If no runtime config (object or path) was given to `Config.__init__`, it will automatically seek out and load `~/.ssh/config` and/or `/etc/ssh/ssh_config`, if they exist (and in that order.)

Note: Rules present in both files will result in the user-level file 'winning', as the first rule found during lookup is always used.

- If none of the above vectors yielded SSH config data, a blank/empty `SSHConfig` is the final result.
- Regardless of how the object was generated, it is exposed as `Config.base_ssh_config`.

`Connection`'s use of `ssh_config` values

`Connection` objects expose a per-host 'view' of their config's SSH data (obtained via `lookup`) as `Connection.ssh_config`. `Connection` itself references these values as described in the following subsections, usually as simple defaults for the appropriate config key or parameter (`port`, `forward_agent`, etc.)

Unless otherwise specified, these values override regular configuration values for the same keys, but may themselves be overridden by `Connection.__init__` parameters.

Take for example a `~/.fabric.yaml`:

```
user: foo
```

Absent any other configuration, `Connection('myhost')` connects as the `foo` user.

If we also have an `~/.ssh/config`:

```
Host *
    User bar
```

then `Connection('myhost')` connects as `bar` (the SSH config wins over the Fabric config.)

However, in both cases, `Connection('myhost', user='biz')` will connect as `biz`.

Note: The below sections use capitalized versions of `ssh_config` keys for easier correlation with `man ssh_config`, **but** the actual `SSHConfig` data structure is normalized to lowercase keys, since SSH config files are technically case-insensitive.

Connection parameters

- `Hostname`: replaces the original value of `host` (which is preserved as `.original_host`.)
- `Port`: supplies the default value for the `port` config option / parameter.
- `User`: supplies the default value for the `user` config option / parameter.
- `ConnectTimeout`: sets the default value for the `timeouts.connect` config option / `timeout` parameter.

Proxying

- `ProxyCommand`: supplies default (string) value for `gateway`.
- `ProxyJump`: supplies default (`Connection`) value for `gateway`.
  - Nested-style `ProxyJump`, i.e. `user1@hop1.host,user2@hop2.host,...`, will result in an appropriate series of nested `gateway` values under the hood - as if the user had manually specified `Connecton(..., gateway=Connection('user1@hop1.host', gateway=Connection('user2@hop2.host', gateway=...)))`.

Note: If both are specified for a given host, `ProxyJump` will override `ProxyCommand`. This is slightly different from OpenSSH, where the order the directives are loaded determines which one wins. Doing so on our end (where we view the config as a dictionary structure) requires additional work.

Authentication

- `ForwardAgent`: controls default behavior of `forward_agent`.
- `IdentityFile`: appends to the `key_filename` key within `connect_kwargs` (similar to *`--identity`*.)

Disabling (most) `ssh_config` loading

Users who need tighter control over how their environment gets configured may want to disable the automatic loading of system/user level SSH config files; this can prevent hard-to-expect errors such as a new user's `~/.ssh/config` overriding values that are being set in the regular config hierarchy.

To do so, simply set the top level config option `load_ssh_configs` to `False`.

Note: Changing this setting does *not* disable loading of runtime-level config files (e.g. via `-F`). If a user is explicitly telling us to load such a file, we assume they know what they're doing.

3.3 Networking

3.3.1 SSH connection gateways

Background

When connecting to well-secured networks whose internal hosts are not directly reachable from the Internet, a common pattern is "bouncing", "gatewaying" or "proxying" SSH connections via an intermediate host (often called a "bastion", "gateway" or "jump box").

Gatewaying requires making an initial/outer SSH connection to the gateway system, then using that connection as a transport for the "real" connection to the final/internal host.

At a basic level, one could `ssh gatewayhost`, then `ssh internalhost` from the resulting shell. This works for individual long-running sessions, but becomes a burden when it must be done frequently.

There are two gateway solutions available in Fabric, mirroring the functionality of OpenSSH's client: `ProxyJump` style (easier, less overhead, can be nested) or `ProxyCommand` style (more overhead, can't be nested, sometimes more flexible). Both support the usual range of configuration sources: Fabric's own config framework, SSH config files, or runtime parameters.

`ProxyJump`

This style of gateway uses the SSH protocol's `direct-tcpip` channel type - a lightweight method of requesting that the gateway's `sshd` open a connection on our behalf to another system. (This has been possible in OpenSSH server for a long time; support in OpenSSH's client is new as of 7.3.)

Channel objects (instances of `paramiko.channel.Channel`) implement Python's socket API and are thus usable in place of real operating system sockets for nearly any Python code.

`ProxyJump` style gatewaying is simple to use: create a new `Connection` object parameterized for the gateway, and supply it as the `gateway` parameter when creating your inner/real `Connection`:

```
from fabric import Connection

c = Connection('internalhost', gateway=Connection('gatewayhost'))
```

As with any other `Connection`, the gateway connection may be configured with its own username, port number, and so forth. (This includes `gateway` itself - they can be chained indefinitely!)

ProxyCommand

The traditional OpenSSH command-line client has long offered a `ProxyCommand` directive (see man ssh_config), which pipes the inner connection's input and output through an arbitrary local subprocess.

Compared to `ProxyJump` style gateways, this adds overhead (the extra subprocess) and can't easily be nested. In trade, it allows for advanced tricks like use of SOCKS proxies, or custom filtering/gatekeeping applications.

`ProxyCommand` subprocesses are typically another `ssh` command, such as `ssh -W %h:%p gatewayhost`; or (on SSH versions lacking `-W`) the widely available `netcat`, via `ssh gatewayhost nc %h %p`.

Fabric supports `ProxyCommand` by accepting command string objects in the `gateway` kwarg of `Connection`; this is used to populate a `paramiko.proxy.ProxyCommand` object at connection time.

Additional concerns

If you're unsure which of the two approaches to use: use `ProxyJump` style. It performs better, uses fewer resources on your local system, and has an easier-to-use API.

> **Warning:** Requesting both types of gateways simultaneously to the same host (i.e. supplying a `Connection` as the `gateway` via kwarg or config, *and* loading a config file containing `ProxyCommand`) is considered an error and will result in an exception.

CHAPTER 4

The `fab` CLI tool

Details on the CLI interface to Fabric, how it extends Invoke's CLI machinery, and examples of shortcuts for executing tasks across hosts or groups.

4.1 Command-line interface

This page documents the details of Fabric's command-line interface, `fab`.

4.1.1 Options & arguments

Note: By default, `fab` honors all of the same CLI options as Invoke's 'inv' program; only additions and overrides are listed here!

For example, Fabric implements `--prompt-for-passphrase` and `--prompt-for-login-password` because they are SSH specific, but it inherits a related option – –prompt-for-sudo-password – from Invoke, which handles sudo autoresponse concerns.

`-S, --ssh-config`
Takes a path to load as a runtime SSH config file. See *Loading and using ssh_config files*.

`-H, --hosts`
Takes a comma-separated string listing hostnames against which tasks should be executed, in serial. See *Runtime specification of host lists*.

`-i, --identity`
Overrides the `key_filename` value in the `connect_kwargs` config setting (which is read by `Connection`, and eventually makes its way into Paramiko; see the docstring for `Connection` for details.)

Typically this can be thought of as identical to `ssh -i <path>`, i.e. supplying a specific, runtime private key file. Like `ssh -i`, it builds an iterable of strings and may be given multiple times.

Default: `[]`.

--prompt-for-passphrase
: Causes Fabric to prompt 'up front' for a value to store as the `connect_kwargs.passphrase` config setting (used by Paramiko to decrypt private key files.) Useful if you do not want to configure such values in on-disk conf files or via shell environment variables.

--prompt-for-login-password
: Causes Fabric to prompt 'up front' for a value to store as the `connect_kwargs.password` config setting (used by Paramiko when authenticating via passwords and, in some versions, also used for key passphrases.) Useful if you do not want to configure such values in on-disk conf files or via shell environment variables.

4.1.2 Seeking & loading tasks

`fab` follows all the same rules as Invoke's collection loading, with the sole exception that the default collection name sought is `fabfile` instead of `tasks`. Thus, whenever Invoke's documentation mentions `tasks` or `tasks.py`, Fabric substitutes `fabfile` / `fabfile.py`.

For example, if your current working directory is `/home/myuser/projects/mywebapp`, running `fab --list` will cause Fabric to look for `/home/myuser/projects/mywebapp/fabfile.py` (or `/home/myuser/projects/mywebapp/fabfile/__init__.py` - Python's import system treats both the same). If it's not found there, `/home/myuser/projects/fabfile.py` is sought next; and so forth.

4.1.3 Runtime specification of host lists

While advanced use cases may need to take matters into their own hands, you can go reasonably far with the core *`--hosts`* flag, which specifies one or more hosts the given task(s) should execute against.

By default, execution is a serial process: for each task on the command line, run it once for each host given to *`--hosts`*. Imagine tasks that simply print `Running <task name> on <host>!`:

```
$ fab --hosts host1,host2,host3 taskA taskB
Running taskA on host1!
Running taskA on host2!
Running taskA on host3!
Running taskB on host1!
Running taskB on host2!
Running taskB on host3!
```

Note: When *`--hosts`* is not given, `fab` behaves similarly to Invoke's command-line interface, generating regular instances of `Context` instead of `Connections`.

4.1.4 Executing arbitrary/ad-hoc commands

`fab` leverages a lesser-known command line convention and may be called in the following manner:

```
$ fab [options] -- [shell command]
```

where everything after the `--` is turned into a temporary `Connection.run` call, and is not parsed for `fab` options. If you've specified a host list via an earlier task or the core CLI flags, this usage will act like a one-line anonymous task.

For example, let's say you wanted kernel info for a bunch of systems:

```
$ fab -H host1,host2,host3 -- uname -a
```

Such a command is equivalent to the following Fabric library code:

```
from fabric import Group

Group('host1', 'host2', 'host3').run("uname -a")
```

Most of the time you will want to just write out the task in your fabfile (anything you use once, you're likely to use again) but this feature provides a handy, fast way to dash off an SSH-borne command while leveraging predefined connection settings.

CHAPTER 5

API

Know what you're looking for & just need API details? View our auto-generated API documentation:

5.1 config

5.2 connection

5.3 exceptions

5.4 executor

5.5 group

5.6 runners

5.7 tasks

5.8 testing

The `fabric.testing` subpackage contains a handful of test helper modules:

- `fabric.testing.base` which only depends on things like `mock` and is appropriate in just about any test paradigm;
- `fabric.testing.fixtures`, containing `pytest` fixtures and thus only of interest for users of `pytest`.

All are documented below. Please note the module-level documentation which contains install instructions!

5.8.1 testing.base

5.8.2 testing.fixtures

5.9 transfer

5.10 tunnels

5.11 util

Index

Symbols

–prompt-for-login-password
    command line option, 20
–prompt-for-passphrase
    command line option, 19
-H, –hosts
    command line option, 19
-S, –ssh-config
    command line option, 19
-i, –identity
    command line option, 19

C

command line option
    –prompt-for-login-password, 20
    –prompt-for-passphrase, 19
    -H, –hosts, 19
    -S, –ssh-config, 19
    -i, –identity, 19