Fabric

Release

Jun 06, 2021

Contents

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')
```

```
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')

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 invoke 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`.
- `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

class fabric.config.**Config**(**args*, ***kwargs*)

An invoke.config.Config subclass with extra Fabric-related behavior.

This class behaves like invoke.config.Config in every way, with the following exceptions:

- its *global_defaults* staticmethod has been extended to add/modify some default settings (see its documentation, below, for details);
- it triggers loading of Fabric-specific env vars (e.g. FABRIC_RUN_HIDE=true instead of INVOKE_RUN_HIDE=true) and filenames (e.g. /etc/fabric.yaml instead of /etc/invoke.yaml).
- it extends the API to account for loading ssh_config files (which are stored as additional attributes and have no direct relation to the regular config data/hierarchy.)

Intended for use with *Connection*, as using vanilla invoke.config.Config objects would require users to manually define port, user and so forth.

See also:

Configuration, *Loading and using ssh_config files*

New in version 2.0.

__init__(**args*, ***kwargs*)

Creates a new Fabric-specific config object.

For most API details, see invoke.config.Config.__init__. Parameters new to this subclass are listed below.

Parameters

- **ssh_config** – Custom/explicit `paramiko.config.SSHConfig` object. If given, prevents loading of any SSH config files. Default: `None`.
- **runtime_ssh_path** (*str*) – Runtime SSH config path to load. Prevents loading of system/user files if given. Default: `None`.
- **system_ssh_path** (*str*) – Location of the system-level SSH config file. Default: `/etc/ssh/ssh_config`.
- **user_ssh_path** (*str*) – Location of the user-level SSH config file. Default: `~/.ssh/config`.
- **lazy** (*bool*) – Has the same meaning as the parent class' `lazy`, but additionally controls whether SSH config file loading is deferred (requires manually calling *load_ssh_config* sometime.) For example, one may need to wait for user input before calling *set_runtime_ssh_path*, which will inform exactly what *load_ssh_config* does.

static global_defaults()

Default configuration values and behavior toggles.

Fabric only extends this method in order to make minor adjustments and additions to Invoke's `global_defaults`; see its documentation for the base values, such as the config subtrees controlling behavior of `run` or how `tasks` behave.

For Fabric-specific modifications and additions to the Invoke-level defaults, see our own config docs at *Default configuration values*.

New in version 2.0.

load_ssh_config()

Load SSH config file(s) from disk.

Also (beforehand) ensures that Invoke-level config re: runtime SSH config file paths, is accounted for.

New in version 2.0.

set_runtime_ssh_path(*path*)

Configure a runtime-level SSH config file path.

If set, this will cause *load_ssh_config* to skip system and user files, as OpenSSH does.

New in version 2.0.

5.2 connection

class fabric.connection.**Connection**(*host*, *user=None*, *port=None*, *config=None*, *gateway=None*, *forward_agent=None*, *connect_timeout=None*, *connect_kwargs=None*)

A connection to an SSH daemon, with methods for commands and file transfer.

Basics

This class inherits from Invoke's `Context`, as it is a context within which commands, tasks etc can operate. It also encapsulates a Paramiko `SSHClient` instance, performing useful high level operations with that `SSHClient` and `Channel` instances generated from it.

Note: Many SSH specific options – such as specifying private keys and passphrases, timeouts, disabling SSH agents, etc – are handled directly by Paramiko and should be specified via the *connect_kwargs argument* of the

constructor.

Lifecycle

Connection has a basic "*create*, *connect/open*, *do work*, *disconnect/close*" lifecycle:

- *Instantiation* imprints the object with its connection parameters (but does **not** actually initiate the network connection).
- Methods like *run*, *get* etc automatically trigger a call to *open* if the connection is not active; users may of course call *open* manually if desired.
- Connections do not always need to be explicitly closed; much of the time, Paramiko's garbage collection hooks or Python's own shutdown sequence will take care of things. **However**, should you encounter edge cases (for example, sessions hanging on exit) it's helpful to explicitly close connections when you're done with them.

  This can be accomplished by manually calling *close*, or by using the object as a contextmanager:

```
with Connection('host') as c:
    c.run('command')
    c.put('file')
```

Note: This class rebinds `invoke.context.Context.run` to *local* so both remote and local command execution can coexist.

Configuration

Most *Connection* parameters honor *Invoke-style configuration* as well as any applicable *SSH config file directives*. For example, to end up with a connection to `admin@myhost`, one could:

- Use any built-in config mechanism, such as `/etc/fabric.yml`, `~/.fabric.json`, collection-driven configuration, env vars, etc, stating `user: admin` (or `{"user": "admin"}`, depending on config format.) Then `Connection('myhost')` would implicitly have a `user` of `admin`.
- Use an SSH config file containing `User admin` within any applicable `Host` header (`Host myhost`, `Host *`, etc.) Again, `Connection('myhost')` will default to an `admin` user.
- Leverage host-parameter shorthand (described in *Config.__init__*), i.e. `Connection('admin@myhost')`.
- Give the parameter directly: `Connection('myhost', user='admin')`.

The same applies to agent forwarding, gateways, and so forth.

New in version 2.0.

__init__(*host*, *user=None*, *port=None*, *config=None*, *gateway=None*, *forward_agent=None*, *connect_timeout=None*, *connect_kwargs=None*)

Set up a new object representing a server connection.

Parameters

- **host** (*str*) – the hostname (or IP address) of this connection.

  May include shorthand for the `user` and/or `port` parameters, of the form `user@host`, `host:port`, or `user@host:port`.

  Note: Due to ambiguity, IPv6 host addresses are incompatible with the `host:port` shorthand (though `user@host` will still work OK). In other words, the presence of >1

: character will prevent any attempt to derive a shorthand port number; use the explicit `port` parameter instead.

Note: If `host` matches a `Host` clause in loaded SSH config data, and that `Host` clause contains a `Hostname` directive, the resulting *`Connection`* object will behave as if `host` is equal to that `Hostname` value.

In all cases, the original value of `host` is preserved as the `original_host` attribute.

Thus, given SSH config like so:

```
Host myalias
    Hostname realhostname
```

a call like `Connection(host='myalias')` will result in an object whose `host` attribute is `realhostname`, and whose `original_host` attribute is `myalias`.

- **`user`** (*`str`*) – the login user for the remote connection. Defaults to `config.user`.
- **`port`** (*`int`*) – the remote port. Defaults to `config.port`.
- **`config`** – configuration settings to use when executing methods on this *`Connection`* (e.g. default SSH port and so forth).

  Should be a *`Config`* or an `invoke.config.Config` (which will be turned into a *`Config`*).

  Default is an anonymous *`Config`* object.

- **`gateway`** – An object to use as a proxy or gateway for this connection.

  This parameter accepts one of the following:

  – another *`Connection`* (for a `ProxyJump` style gateway);

  – a shell command string (for a `ProxyCommand` style style gateway).

  Default: `None`, meaning no gatewaying will occur (unless otherwise configured; if one wants to override a configured gateway at runtime, specify `gateway=False`.)

  See also:

  SSH connection gateways

- **`forward_agent`** (*`bool`*) – Whether to enable SSH agent forwarding.

  Default: `config.forward_agent`.

- **`connect_timeout`** (*`int`*) – Connection timeout, in seconds.

  Default: `config.timeouts.connect`.

Parameters **`connect_kwargs`** (*`dict`*) – Keyword arguments handed verbatim to `SSHClient.connect` (when *`open`* is called).

`Connection` tries not to grow additional settings/kwargs of its own unless it is adding value of some kind; thus, `connect_kwargs` is currently the right place to hand in paramiko connection parameters such as `pkey` or `key_filename`. For example:

```
c = Connection(
    host="hostname",
    user="admin",
    connect_kwargs={
        "key_filename": "/home/myuser/.ssh/private.key",
    },
)
```

Default: `config.connect_kwargs`.

Raises **ValueError** – if user or port values are given via both `host` shorthand *and* their own arguments. (We refuse the temptation to guess).

close()

Terminate the network connection to the remote end, if open.

If no connection is open, this method does nothing.

New in version 2.0.

forward_local(*local_port*, *remote_port=None*, *remote_host='localhost'*, *local_host='localhost'*)

Open a tunnel connecting `local_port` to the server's environment.

For example, say you want to connect to a remote PostgreSQL database which is locked down and only accessible via the system it's running on. You have SSH access to this server, so you can temporarily make port 5432 on your local system act like port 5432 on the server:

```
import psycopg2
from fabric import Connection

with Connection('my-db-server').forward_local(5432):
    db = psycopg2.connect(
        host='localhost', port=5432, database='mydb'
    )
    # Do things with 'db' here
```

This method is analogous to using the `-L` option of OpenSSH's `ssh` program.

Parameters

- **local_port** (*int*) – The local port number on which to listen.
- **remote_port** (*int*) – The remote port number. Defaults to the same value as `local_port`.
- **local_host** (*str*) – The local hostname/interface on which to listen. Default: `localhost`.
- **remote_host** (*str*) – The remote hostname serving the forwarded remote port. Default: `localhost` (i.e., the host this *Connection* is connected to.)

Returns Nothing; this method is only useful as a context manager affecting local operating system state.

New in version 2.0.

forward_remote(*remote_port*, *local_port=None*, *remote_host='127.0.0.1'*, *local_host='localhost'*)

Open a tunnel connecting `remote_port` to the local environment.

For example, say you're running a daemon in development mode on your workstation at port 8080, and want to funnel traffic to it from a production or staging environment.

In most situations this isn't possible as your office/home network probably blocks inbound traffic. But you have SSH access to this server, so you can temporarily make port 8080 on that server act like port 8080 on your workstation:

```
from fabric import Connection

c = Connection('my-remote-server')
with c.forward_remote(8080):
    c.run("remote-data-writer --port 8080")
    # Assuming remote-data-writer runs until interrupted, this will
    # stay open until you Ctrl-C...
```

This method is analogous to using the `-R` option of OpenSSH's `ssh` program.

Parameters

- **`remote_port`** (*int*) – The remote port number on which to listen.
- **`local_port`** (*int*) – The local port number. Defaults to the same value as `remote_port`.
- **`local_host`** (*str*) – The local hostname/interface the forwarded connection talks to. Default: `localhost`.
- **`remote_host`** (*str*) – The remote interface address to listen on when forwarding connections. Default: `127.0.0.1` (i.e. only listen on the remote localhost).

Returns Nothing; this method is only useful as a context manager affecting local operating system state.

New in version 2.0.

`get`(**args*, ***kwargs*)
Get a remote file to the local filesystem or file-like object.

Simply a wrapper for *`Transfer.get`*. Please see its documentation for all details.

New in version 2.0.

`is_connected`
Whether or not this connection is actually open.

New in version 2.0.

`local`(**args*, ***kwargs*)
Execute a shell command on the local system.

This method is effectively a wrapper of `invoke.run`; see its docs for details and call signature.

New in version 2.0.

`open`()
Initiate an SSH connection to the host/port this object is bound to.

This may include activating the configured gateway connection, if one is set.

Also saves a handle to the now-set Transport object for easier access.

Various connect-time settings (and/or their corresponding *SSH config options*) are utilized here in the call to `SSHClient.connect`. (For details, see *the configuration docs*.)

New in version 2.0.

`open_gateway`()
Obtain a socket-like object from `gateway`.

Returns A `direct-tcpip` `paramiko.channel.Channel`, if `gateway` was a *Connection*; or a `ProxyCommand`, if `gateway` was a string.

New in version 2.0.

put(**args*, ***kwargs*)
Put a local file (or file-like object) to the remote filesystem.

Simply a wrapper for *Transfer.put*. Please see its documentation for all details.

New in version 2.0.

run(*command*, ***kwargs*)
Execute a shell command on the remote end of this connection.

This method wraps an SSH-capable implementation of `invoke.runners.Runner.run`; see its documentation for details.

> **Warning:** There are a few spots where Fabric departs from Invoke's default settings/behaviors; they are documented under *Config.global_defaults*.

New in version 2.0.

sftp()
Return a `SFTPClient` object.

If called more than one time, memoizes the first result; thus, any given *Connection* instance will only ever have a single SFTP client, and state (such as that managed by `chdir`) will be preserved.

New in version 2.0.

sudo(*command*, ***kwargs*)
Execute a shell command, via `sudo`, on the remote end.

This method is identical to `invoke.context.Context.sudo` in every way, except in that – like *run* – it honors per-host/per-connection configuration overrides in addition to the generic/global ones. Thus, for example, per-host sudo passwords may be configured.

New in version 2.0.

5.3 exceptions

exception fabric.exceptions.**GroupException**(*result*)
Lightweight exception wrapper for *GroupResult* when one contains errors.

New in version 2.0.

__weakref__
list of weak references to the object (if defined)

5.4 group

class fabric.group.**Group**(**hosts*)
A collection of *Connection* objects whose API operates on its contents.

> **Warning:** **This is a partially abstract class**; you need to use one of its concrete subclasses (such as *SerialGroup* or *ThreadingGroup*) or you'll get `NotImplementedError` on most of the methods.

Most methods in this class mirror those of *Connection*, taking the same arguments; however their return values and exception-raising behavior differs:

- Return values are dict-like objects (*GroupResult*) mapping *Connection* objects to the return value for the respective connections: *Group.run* returns a map of *Connection* to *runners.Result*, *Group.get* returns a map of *Connection* to *transfer.Result*, etc.
- If any connections encountered exceptions, a *GroupException* is raised, which is a thin wrapper around what would otherwise have been the *GroupResult* returned; within that wrapped *GroupResult*, the excepting connections map to the exception that was raised, in place of a `Result` (as no `Result` was obtained.) Any non-excepting connections will have a `Result` value, as normal.

For example, when no exceptions occur, a session might look like this:

```
>>> group = SerialGroup('host1', 'host2')
>>> group.run("this is fine")
{
    <Connection host='host1'>: <Result cmd='this is fine' exited=0>,
    <Connection host='host2'>: <Result cmd='this is fine' exited=0>,
}
```

With exceptions (anywhere from 1 to "all of them"), it looks like so; note the different exception classes, e.g. `UnexpectedExit` for a completed session whose command exited poorly, versus `socket.gaierror` for a host that had DNS problems:

```
>>> group = SerialGroup('host1', 'host2', 'notahost')
>>> group.run("will it blend?")
{
    <Connection host='host1'>: <Result cmd='will it blend?' exited=0>,
    <Connection host='host2'>: <UnexpectedExit: cmd='...' exited=1>,
    <Connection host='notahost'>: gaierror(...),
}
```

New in version 2.0.

__init__(**hosts*)
Create a group of connections from one or more shorthand strings.

See *Connection* for details on the format of these strings - they will be used as the first positional argument of *Connection* constructors.

__weakref__
list of weak references to the object (if defined)

classmethod from_connections(*connections*)
Alternate constructor accepting *Connection* objects.

New in version 2.0.

get(**args*, ***kwargs*)
Executes *Connection.get* on all member *Connections*.

Returns a *GroupResult*.

New in version 2.0.

run(**args*, ***kwargs*)
Executes *Connection.run* on all member *Connections*.

Returns a *GroupResult*.

New in version 2.0.

class fabric.group.**GroupResult**(**args*, ***kwargs*)
Collection of results and/or exceptions arising from *Group* methods.

Acts like a dict, but adds a couple convenience methods, to wit:

- Keys are the individual *Connection* objects from within the *Group*.
- Values are either return values / results from the called method (e.g. *runners.Result* objects), *or* an exception object, if one prevented the method from returning.
- Subclasses dict, so has all dict methods.
- Has *succeeded* and *failed* attributes containing sub-dicts limited to just those key/value pairs that succeeded or encountered exceptions, respectively.
  - Of note, these attributes allow high level logic, e.g. `if mygroup.run('command').failed` and so forth.

New in version 2.0.

__weakref__
list of weak references to the object (if defined)

failed
A sub-dict containing only failed results.

New in version 2.0.

succeeded
A sub-dict containing only successful results.

New in version 2.0.

class fabric.group.**SerialGroup**(**hosts*)
Subclass of *Group* which executes in simple, serial fashion.

New in version 2.0.

class fabric.group.**ThreadingGroup**(**hosts*)
Subclass of *Group* which uses threading to execute concurrently.

New in version 2.0.

5.5 runners

class fabric.runners.**Remote**(*context*)
Run a shell command over an SSH connection.

This class subclasses invoke.runners.Runner; please see its documentation for most public API details.

Note: *Remote*'s __init__ method expects a *Connection* (or subclass) instance for its context argument.

New in version 2.0.

class fabric.runners.**Result**(***kwargs*)
An invoke.runners.Result exposing which *Connection* was run against.

Exposes all attributes from its superclass, then adds a `.connection`, which is simply a reference to the *Connection* whose method yielded this result.

New in version 2.0.

5.6 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!

5.6.1 testing.base

This module contains helpers/fixtures to assist in testing Fabric-driven code.

It is not intended for production use, and pulls in some test-oriented dependencies such as mock. You can install an 'extra' variant of Fabric to get these dependencies if you aren't already using them for your own testing purposes: `pip install fabric[testing]`.

Note: If you're using pytest for your test suite, you may be interested in grabbing `fabric[pytest]` instead, which encompasses the dependencies of both this module and the *fabric.testing.fixtures* module, which contains pytest fixtures.

New in version 2.1.

class fabric.testing.base.**Command**(*cmd=None*, *out=b''*, *err=b''*, *in_=None*, *exit=0*, *waits=0*)
Data record specifying params of a command execution to mock/expect.

Parameters

- **cmd** (*str*) – Command string to expect. If not given, no expectations about the command executed will be set up. Default: `None`.
- **out** (*bytes*) – Data yielded as remote stdout. Default: `b""`.
- **err** (*bytes*) – Data yielded as remote stderr. Default: `b""`.
- **exit** (*int*) – Remote exit code. Default: `0`.
- **waits** (*int*) – Number of calls to the channel's `exit_status_ready` that should return `False` before it then returns `True`. Default: `0` (`exit_status_ready` will return `True` immediately).

New in version 2.1.

__weakref__
list of weak references to the object (if defined)

class fabric.testing.base.**MockChannel**(**args*, ***kwargs*)
Mock subclass that tracks state for its `recv(_stderr)?` methods.

Turns out abusing function closures inside MockRemote to track this state only worked for 1 command per session!

New in version 2.1.

class fabric.testing.base.**MockRemote**
Class representing mocked remote state.

By default this class is set up for start/stop style patching as opposed to the more common context-manager or decorator approach; this is so it can be used in situations requiring setup/teardown semantics.

Defaults to setting up a single anonymous *Session*, so it can be used as a "request & forget" pytest fixture. Users requiring detailed remote session expectations can call methods like *expect*, which wipe that anonymous Session & set up a new one instead.

New in version 2.1.

__weakref__
list of weak references to the object (if defined)

expect(**args*, ***kwargs*)
Convenience method for creating & 'expect'ing a single *Session*.

Returns the single *MockChannel* yielded by that Session.

New in version 2.1.

expect_sessions(**sessions*)
Sets the mocked remote environment to expect the given `sessions`.

Returns a list of *MockChannel* objects, one per input *Session*.

New in version 2.1.

sanity()
Run post-execution sanity checks (usually 'was X called' tests.)

New in version 2.1.

start()
Start patching SSHClient with the stored sessions, returning channels.

New in version 2.1.

stop()
Stop patching SSHClient.

New in version 2.1.

class fabric.testing.base.**MockSFTP**(*autostart=True*)
Class managing mocked SFTP remote state.

Used in start/stop fashion in eg doctests; wrapped in the SFTP fixtures in conftest.py for main use.

New in version 2.1.

__weakref__
list of weak references to the object (if defined)

class fabric.testing.base.**Session**(*host=None*, *user=None*, *port=None*, *commands=None*, *cmd=None*, *out=None*, *in_=None*, *err=None*, *exit=None*, *waits=None*)
A mock remote session of a single connection and 1 or more command execs.

Allows quick configuration of expected remote state, and also helps generate the necessary test mocks used by *MockRemote* itself. Only useful when handed into *MockRemote*.

The parameters `cmd`, `out`, `err`, `exit` and `waits` are all shorthand for the same constructor arguments for a single anonymous *Command*; see *Command* for details.

To give fully explicit *Command* objects, use the `commands` parameter.

Parameters

- **user** (*str*) –
- **host** (*str*) –
- **port** (*int*) – Sets up expectations that a connection will be generated to the given user, host and/or port. If `None` (default), no expectations are generated / any value is accepted.
- **commands** – Iterable of *Command* objects, used when mocking nontrivial sessions involving >1 command execution per host. Default: `None`.

Note: Giving `cmd`, `out` etc alongside explicit `commands` is not allowed and will result in an error.

New in version 2.1.

__weakref__
list of weak references to the object (if defined)

generate_mocks()
Mocks `SSHClient` and `Channel`.

Specifically, the client will expect itself to be connected to `self.host` (if given), the channels will be associated with the client's `Transport`, and the channels will expect/provide command-execution behavior as specified on the *Command* objects supplied to this *Session*.

The client is then attached as `self.client` and the channels as `self.channels`.

Returns `None` - this is mostly a "deferred setup" method and callers will just reference the above attributes (and call more methods) as needed.

New in version 2.1.

5.6.2 testing.fixtures

pytest fixtures for easy use of Fabric test helpers.

To get Fabric plus this module's dependencies (as well as those of the main *fabric.testing.base* module which these fixtures wrap), `pip install fabric[pytest]`.

The simplest way to get these fixtures loaded into your test suite so Pytest notices them is to import them into a `conftest.py` (docs). For example, if you intend to use the *remote* and *client* fixtures:

```
from fabric.testing.fixtures import client, remote
```

New in version 2.1.

fabric.testing.fixtures.**client**()
Mocks `SSHClient` for testing calls to `connect()`.

Yields a mocked `SSHClient` instance.

This fixture updates `get_transport` to return a mock that appears active on first check, then inactive after, matching most tests' needs by default:

- *Connection* instantiates, with a None `.transport`.
- Calls to `.open()` test `.is_connected`, which returns `False` when `.transport` is falsey, and so the first open will call `SSHClient.connect` regardless.
- `.open()` then sets `.transport` to `SSHClient.get_transport()`, so `Connection.transport` is effectively `client.get_transport.return_value`.
- Subsequent activity will want to think the mocked SSHClient is "connected", meaning we want the mocked transport's `.active` to be `True`.
- This includes *Connection.close*, which short-circuits if `.is_connected`; having a statically `True` active flag means a full open -> close cycle will run without error. (Only tests that double-close or double-open should have issues here.)

End result is that:

- `.is_connected` behaves False after instantiation and before `.open`, then True after `.open`
- `.close` will work normally on 1st call
- `.close` will behave "incorrectly" on subsequent calls (since it'll think connection is still live.) Tests that check the idempotency of `.close` will need to tweak their mock mid-test.

For 'full' fake remote session interaction (i.e. stdout/err reading/writing, channel opens, etc) see *remote*.

New in version 2.1.

`fabric.testing.fixtures.`**`connection`**`()`
Yields a *Connection* object with mocked methods.

Specifically:

- the hostname is set to `"host"` and the username to `"user"`;
- the primary API members (*Connection.run*, *Connection.local*, etc) are replaced with `mock.Mock` instances;
- the `run.in_stream` config option is set to `False` to avoid attempts to read from stdin (which typically plays poorly with pytest and other capturing test runners);

New in version 2.1.

`fabric.testing.fixtures.`**`cxn`**`()`
Yields a *Connection* object with mocked methods.

Specifically:

- the hostname is set to `"host"` and the username to `"user"`;
- the primary API members (*Connection.run*, *Connection.local*, etc) are replaced with `mock.Mock` instances;
- the `run.in_stream` config option is set to `False` to avoid attempts to read from stdin (which typically plays poorly with pytest and other capturing test runners);

New in version 2.1.

`fabric.testing.fixtures.`**`remote`**`()`
Fixture allowing setup of a mocked remote session & access to sub-mocks.

Yields a *MockRemote* object (which may need to be updated via *MockRemote.expect*, *MockRemote.expect_sessions*, etc; otherwise a default session will be used) & calls *MockRemote.sanity* and *MockRemote.stop* on teardown.

New in version 2.1.

fabric.testing.fixtures.**sftp**()
Fixture allowing setup of a mocked remote SFTP session.

Yields a 3-tuple of: Transfer() object, SFTPClient object, and mocked OS module.

For many/most tests which only want the Transfer and/or SFTPClient objects, see *sftp_objs* and *transfer* which wrap this fixture.

New in version 2.1.

fabric.testing.fixtures.**sftp_objs**(*sftp*)
Wrapper for *sftp* which only yields the Transfer and SFTPClient.

New in version 2.1.

fabric.testing.fixtures.**transfer**(*sftp*)
Wrapper for *sftp* which only yields the Transfer object.

New in version 2.1.

5.7 transfer

File transfer via SFTP and/or SCP.

class fabric.transfer.**Result**(*local*, *orig_local*, *remote*, *orig_remote*, *connection*)
A container for information about the result of a file transfer.

See individual attribute/method documentation below for details.

Note: Unlike similar classes such as invoke.runners.Result or *fabric.runners.Result* (which have a concept of "warn and return anyways on failure") this class has no useful truthiness behavior. If a file transfer fails, some exception will be raised, either an OSError or an error from within Paramiko.

New in version 2.0.

__weakref__
list of weak references to the object (if defined)

class fabric.transfer.**Transfer**(*connection*)
Connection-wrapping class responsible for managing file upload/download.

New in version 2.0.

__weakref__
list of weak references to the object (if defined)

get(*remote*, *local=None*, *preserve_mode=True*)
Download a file from the current connection to the local filesystem.

Parameters

- **remote** (*str*) – Remote file to download.

  May be absolute, or relative to the remote working directory.

Note: Most SFTP servers set the remote working directory to the connecting user's home directory, and (unlike most shells) do *not* expand tildes (~).

For example, instead of saying `get("~/tmp/archive.tgz")`, say `get("tmp/archive.tgz")`.

- **`local`** – Local path to store downloaded file in, or a file-like object.

  If None or another 'falsey'/empty value is given (the default), the remote file is downloaded to the current working directory (as seen by `os.getcwd`) using its remote filename.

  If a string is given, it should be a path to a local directory or file and is subject to similar behavior as that seen by common Unix utilities or OpenSSH's `sftp` or `scp` tools.

  For example, if the local path is a directory, the remote path's base filename will be added onto it (so `get('foo/bar/file.txt', '/tmp/')` would result in creation or overwriting of `/tmp/file.txt`).

  Note: When dealing with nonexistent file paths, normal Python file handling concerns come into play - for example, a `local` path containing non-leaf directories which do not exist, will typically result in an `OSError`.

  If a file-like object is given, the contents of the remote file are simply written into it.

- **`preserve_mode`** (*bool*) – Whether to `os.chmod` the local file so it matches the remote file's mode (default: `True`).

Returns A *Result* object.

New in version 2.0.

`put`(*local*, *remote=None*, *preserve_mode=True*)

Upload a file from the local filesystem to the current connection.

Parameters

- **`local`** – Local path of file to upload, or a file-like object.

  If a string is given, it should be a path to a local (regular) file (not a directory).

  Note: When dealing with nonexistent file paths, normal Python file handling concerns come into play - for example, trying to upload a nonexistent `local` path will typically result in an `OSError`.

  If a file-like object is given, its contents are written to the remote file path.

- **`remote`** (*str*) – Remote path to which the local file will be written.

  Note: Most SFTP servers set the remote working directory to the connecting user's home directory, and (unlike most shells) do *not* expand tildes (~).

  For example, instead of saying `put("archive.tgz", "~/tmp/")`, say `put("archive.tgz", "tmp/")`.

In addition, this means that 'falsey'/empty values (such as the default value, `None`) are allowed and result in uploading to the remote home directory.

Note: When `local` is a file-like object, `remote` is required and must refer to a valid file path (not a directory).

- **preserve_mode** (*bool*) – Whether to `chmod` the remote file so it matches the local file's mode (default: `True`).

Returns A *Result* object.

New in version 2.0.

5.8 tunnels

Tunnel and connection forwarding internals.

If you're looking for simple, end-user-focused connection forwarding, please see *Connection*, e.g. *Connection.forward_local*.

class fabric.tunnels.**Tunnel**(*channel*, *sock*, *finished*)

Bidirectionally forward data between an SSH channel and local socket.

New in version 2.0.

read_and_write(*reader*, *writer*, *chunk_size*)

Read `chunk_size` from `reader`, writing result to `writer`.

Returns `None` if successful, or `True` if the read was empty.

New in version 2.0.

class fabric.tunnels.**TunnelManager**(*local_host*, *local_port*, *remote_host*, *remote_port*, *transport*, *finished*)

Thread subclass for tunnelling connections over SSH between two endpoints.

Specifically, one instance of this class is sufficient to sit around forwarding any number of individual connections made to one end of the tunnel or the other. If you need to forward connections between more than one set of ports, you'll end up instantiating multiple TunnelManagers.

Wraps a `Transport`, which should already be connected to the remote server.

New in version 2.0.

5.9 util

fabric.util.**get_local_user**()

Return the local executing username, or `None` if one can't be found.

New in version 2.0.

Python Module Index

f

Index

Symbols

C

E

F

G

I

L

M

O

P

R

S

T