Fabric Documentation

Release 0.9

Jeff Forcier

April 08, 2014

CHAPTER 1

About

PLEASE NOTE: This is a release-candidate release of Fabric and is not intended for use in production. However, please do test it out on non-critical systems and let us know of any issues you encounter. **END NOTE**

Fabric is a Python library and command-line tool for streamlining the use of SSH for application deployment or systems administration tasks.

It provides a basic suite of operations for executing local or remote shell commands (normally or via `sudo`) and uploading/downloading files, as well as auxiliary functionality such as prompting the running user for input, or aborting execution.

Typical use involves creating a Python module containing one or more functions, then executing them via the `fab` command-line tool. Below is a small but complete "fabfile" containing a single task:

```
from fabric.api import run

def host_type():
    run('uname -s')
```

Once a task is defined, it may be run on one or more servers, like so:

```
$ fab -H localhost,linuxbox host_type
[localhost] run: uname -s
[localhost] out: Darwin
[linuxbox] run: uname -s
[linuxbox] out: Linux

Done.
Disconnecting from localhost... done.
Disconnecting from linuxbox... done.
```

In addition to use via the `fab` fool, Fabric's components may be imported into other Python code, providing a Pythonic interface to the SSH protocol suite at a higher level than that provided by e.g. Paramiko (which Fabric itself leverages.)

CHAPTER 2

Installation

Stable releases of Fabric are best installed via `easy_install` or `pip`; or you may download TGZ or ZIP source archives from a couple of official locations. Detailed instructions and links may be found on the *Installation* page.

We recommend using the latest stable version of Fabric; releases are made often to prevent any large gaps in functionality between the latest stable release and the development version.

However, if you want to live on the edge, you can pull down the latest source code from our Git repository, or fork us on Github. The *Installation* page has details for how to access the source code.

CHAPTER 3

Development

Any hackers interested in improving Fabric (or even users interested in how Fabric is put together or released) please see the *Development* page. It contains comprehensive info on contributing, repository layout, our release strategy, and more.

CHAPTER 4

Documentation

Please note that all documentation is currently written with Python 2.5 users in mind, but with an eye for eventual Python 3.x compatibility. This leads to the following patterns that may throw off readers used to Python 2.4 or who have already upgraded to Python 2.6:

- `from __future__ import with_statement`: a "future import" required to use the `with` statement in Python 2.5 – a feature you'll be using frequently. Python 2.6 users don't need to do this.
- `<true_value> if <expression> else <false_value>`: Python's relatively new ternary statement, available in 2.5 and newer. Python 2.4 and older used to fake this with `<expression> and <true_value> or <false_value>` (which isn't quite the same thing and has some logical loopholes.)
- `print(<expression>)` instead of `print <expression>`: We use the `print` statement's optional parentheses where possible, in order to be more compatible with Python 3.x (in which `print` becomes a function.)

4.1 Overview and Tutorial

Welcome to Fabric!

This document is a whirlwind tour of Fabric's features and a quick guide to its use. Additional documentation (which is linked to throughout) can be found in the *usage documentation* – please make sure to check it out.

4.1.1 What is Fabric?

As the `README` says:

> **PLEASE NOTE:** This is a release-candidate release of Fabric and is not intended for use in production. However, please do test it out on non-critical systems and let us know of any issues you encounter. **END NOTE**
>
> Fabric is a Python library and command-line tool for streamlining the use of SSH for application deployment or systems administration tasks.

More specifically, Fabric is:

- A tool that lets you execute **arbitrary Python functions** via the **command line**;
- A library of subroutines (built on top of a lower-level library) to make executing shell commands over SSH **easy** and **Pythonic**.

Naturally, most users combine these two things, using Fabric to write and execute Python functions, or **tasks**, to automate interactions with remote servers. Let's take a look.

4.1.2 Hello, `fab`

This wouldn't be a proper tutorial without "the usual":

```
def hello():
    print("Hello world!")
```

Placed in a file called `fabfile.py`, that function can be executed with the `fab` tool (installed as part of Fabric) and does just what you'd expect:

```
$ fab hello
Hello world!

Done.
```

That's all there is to it. This functionality allows Fabric to be used as a (very) basic build tool even without importing any of its API.

See also:

Execution strategy, *Defining tasks*, *fab options and arguments*

4.1.3 Local commands

As used above, `fab` only really saves a couple lines of `if __name__ == "__main__"` boilerplate. It's mostly designed for use with Fabric's API, which contains functions (or **operations**) for executing shell commands, transferring files, and so forth.

Let's build a hypothetical Web application fabfile. Fabfiles usually work best at the root of a project:

```
.
|-- __init__.py
|-- app.wsgi
|-- fabfile.py <-- our fabfile!
|-- manage.py
`-- my_app
    |-- __init__.py
    |-- models.py
    |-- templates
    |   `-- index.html
    |-- tests.py
    |-- urls.py
    `-- views.py
```

Note: We're using a Django application here, but only as an example – Fabric is not tied to any external codebase, save for its SSH library.

For starters, perhaps we want to run our tests and then pack up a copy of our app so we're ready for a deploy:

```
from fabric.api import local

def prepare_deploy():
    local('./manage.py test my_app', capture=False)
    local('tar czf /tmp/my_project.tgz .', capture=False)
```

The output of which might look a bit like this:

```
$ fab prepare_deploy
[localhost] run: ./manage.py test my_app
Creating test database...
Creating tables
Creating indexes
.....................................
----------------------------------------------------------------------
Ran 42 tests in 9.138s

OK
Destroying test database...

[localhost] run: tar czf /tmp/my_project.tgz .

Done.
```

The code itself is straightforward: import a Fabric API function, `local`, and use it to run local shell commands. The rest of Fabric's API is similar – it's all just Python.

See also:

Operations, *Fabfile discovery*

4.1.4 Organize it your way

Because Fabric is "just Python" you're free to organize your fabfile any way you want. For example, it's often useful to start splitting things up into subtasks:

```
from fabric.api import local

def test():
    local('./manage.py test my_app', capture=False)

def pack():
    local('tar czf /tmp/my_project.tgz .', capture=False)

def prepare_deploy():
    test()
    pack()
```

The `prepare_deploy` task can be called just as before, but now you can make a more granular call to one of the sub-tasks, if desired.

4.1.5 Failure

Our base case works fine now, but what happens if our tests fail? Chances are we want to put on the brakes and fix them before deploying.

Fabric checks the return value of programs called via operations and will abort if they didn't exit cleanly. Let's see what happens if one of our tests encounters an error:

```
$ fab prepare_deploy
[localhost] run: ./manage.py test my_app
Creating test database...
Creating tables
Creating indexes
............E............................
```

```
======================================================================
ERROR: testSomething (my_project.my_app.tests.MainTests)
----------------------------------------------------------------------
Traceback (most recent call last):
[...]

----------------------------------------------------------------------
Ran 42 tests in 9.138s

FAILED (errors=1)
Destroying test database...

Fatal error: local() encountered an error (return code 2) while executing './manage.py test my_app'

Aborting.
```

Great! We didn't have to do anything ourselves: Fabric detected the failure and aborted, never running the `pack` task.

See also:

Failure handling (usage documentation)

Failure handling

But what if we wanted to be flexible and give the user a choice? A setting (or **environment variable**, usually shortened to **env var**) called *warn_only* lets you turn aborts into warnings, allowing flexible error handling to occur.

Let's flip this setting on for our `test` function, and then inspect the result of the `local` call ourselves:

```
from __future__ import with_statement
from fabric.api import local, settings, abort
from fabric.contrib.console import confirm

def test():
    with settings(warn_only=True):
        result = local('./manage.py test my_app', capture=False)
    if result.failed and not confirm("Tests failed. Continue anyway?"):
        abort("Aborting at user request.")

[...]
```

In adding this new feature we've introduced a number of new things:

- The `__future__` import required to use `with:` in Python 2.5;
- Fabric's `contrib.console` submodule, containing the `confirm` function, used for simple yes/no prompts;
- The `settings` context manager, used to apply settings to a specific block of code;
- Command-running operations like `local` return objects containing info about their result (such as `.failed`, or also `.return_code`);
- And the `abort` function, used to manually abort execution.

However, despite the additional complexity, it's still pretty easy to follow, and is now much more flexible.

See also:

Context Managers, *Full list of env vars*

4.1.6 Making connections

Let's start wrapping up our fabfile by putting in the keystone: a `deploy` task:

```
def deploy():
    put('/tmp/my_project.tgz', '/tmp/')
    with cd('/srv/django/my_project/'):
        run('tar xzf /tmp/my_project.tgz')
        run('touch app.wsgi')
```

Here again, we introduce a handful of new functions:

- `put`, which simply uploads a file to a remote server;
- `cd`, an easy way of prefixing commands with a `cd /to/some/directory` call;
- `run`, which is similar to `local` but runs remotely instead of locally.

And because at this point, we're using a nontrivial number of Fabric's API functions, let's switch our API import to use `*` (as mentioned in the *fabfile* documentation):

```
from __future__ import with_statement
from fabric.api import *
from fabric.contrib.console import confirm
```

With these changes in place, let's deploy:

```
$ fab deploy
No hosts found. Please specify (single) host string for connection: my_server
[my_server] put: /tmp/my_project.tgz -> /tmp/my_project.tgz
[my_server] run: touch app.wsgi

Done.
```

We never specified any connection info in our fabfile, so Fabric prompted us at runtime. Connection definitions use SSH-like "host strings" (e.g. `user@host:port`) and will use your local username as a default – so in this example, we just had to specify the hostname, `my_server`.

See also:

Importing Fabric

Defining connections beforehand

Specifying connection info at runtime gets old real fast, so Fabric provides a handful of ways to do it in your fabfile or on the command line. We won't cover all of them here, but we will show you the most common one: setting the global host list, *env.hosts*.

env is a global dictionary-like object driving many of Fabric's settings, and can be written to with attributes as well (in fact, `settings`, seen above, is simply a wrapper for this.) Thus, we can modify it at module level near the top of our fabfile like so:

```
from __future__ import with_statement
from fabric.api import *
from fabric.contrib.console import confirm

env.hosts = ['my_server']

def test():
[...]
```

When `fab` loads up our fabfile, our modification of `env` will execute, storing our settings change. The end result is exactly as above: our `deploy` task will run against the `my_server` server.

This is also how you can tell Fabric to run on multiple remote systems at once: because `env.hosts` is a list, `fab` iterates over it, calling the given task once for each connection.

See also:

The environment dictionary, env, *How host lists are constructed*

4.1.7 Conclusion

Our completed fabfile is still pretty short, as such things go. Here it is in its entirety:

```
from __future__ import with_statement
from fabric.api import *
from fabric.contrib.console import confirm

env.hosts = ['my_server']

def test():
    with settings(warn_only=True):
        result = local('./manage.py test my_app', capture=False)
    if result.failed and not confirm("Tests failed. Continue anyway?"):
        abort("Aborting at user request.")

def pack():
    local('tar czf /tmp/my_project.tgz .', capture=False)

def prepare_deploy():
    test()
    pack()

def deploy():
    put('/tmp/my_project.tgz', '/tmp/')
    with cd('/srv/django/my_project/'):
        run('tar xzf /tmp/my_project.tgz')
        run('touch app.wsgi')
```

This fabfile makes use of a large portion of Fabric's feature set:

- defining fabfile tasks and running them with *fab*;
- calling local shell commands with `local`;
- modifying env vars with `settings`;
- handling command failures, prompting the user, and manually aborting;
- and defining host lists and `run`-ning remote commands.

However, there's still a lot more we haven't covered here! Please make sure you follow the various "see also" links, and check out the documentation table of contents on *the main index page*.

Thanks for reading!

4.2 Installation

The most direct way to install Fabric is to obtain the source code and run `python setup.py install`. This method works for both release and development versions of the code, and requires nothing but a basic Python installation and the setuptools library.

Note: If you've obtained the Fabric source via source control and plan on updating your checkout in the future, we highly suggest using `python setup.py develop` instead – it will use symbolic links instead of file copies, ensuring that imports of the library or use of the command-line tool will always refer to your checkout.

4.2.1 Dependencies

In order to install Fabric, you will need three primary pieces of software: the Python programming language, the setuptools library, and the PyCrypto cryptography library. Please read on for important details on each dependency – there are a few gotchas.

Python

Fabric requires Python version 2.5 or 2.6. Some caveats and notes about other Python versions:

- We are not planning on supporting **Python 2.4** given its age and the number of useful tools in Python 2.5 such as context managers and new modules. That said, the actual amount of 2.5-specific functionality is not prohibitively large, and we would link to – but not support – a third-party 2.4-compatible fork. (No such fork exists at this time, to our knowledge.)
- Fabric has not yet been tested on **Python 3.x** and is thus likely to be incompatible with that line of development. However, we try to be at least somewhat forward-looking (e.g. using `print()` instead of `print`) and will definitely be porting to 3.x in the future once our dependencies and the rest of the ecosystem does so as well.

setuptools

Setuptools comes with some Python installations by default; if yours doesn't, you'll need to grab it. In such situations it's typically packaged as `python-setuptools`, `py25-setuptools` or similar. Fabric may drop its setuptools dependency in the future, or include alternative support for the Distribute project, but for now setuptools is required for installation.

PyCrypto

PyCrypto is a dependency of Paramiko (which Fabric uses internally for SSH support), providing the low-level (C-based) encryption algorithms used to run SSH. You will need version 1.9 or newer, and may install PyCrypto from `easy_install` or `pip` without worry. However, unless you are installing from a precompiled source such as a Debian apt repository or RedHat RPM, you will need the ability to build Python C-based modules from source – read on.

Users on **Unix-based platforms** such as Ubuntu or Mac OS X will need the traditional C build toolchain installed (e.g. Developer Tools / XCode Tools on the Mac, or the `build-essential` package on Ubuntu or Debian Linux – basically, anything with `gcc`, `make` and so forth) as well as the Python development libraries, often named `python-dev` or similar.

For **Windows** users we recommend either installing a C development environment such as Cygwin or obtaining a precompiled Win32 PyCrypto package from voidspace's Python modules page.

Development dependencies

If you are interested in doing development work on Fabric (or even just running the test suite), you may also need to install some or all of the following packages:

- git and Mercurial, in order to obtain some of the other dependencies below;
- Nose >=0.10
- Coverage >=2.85
- PyLint >=0.18
- Fudge >=0.9.2
- Sphinx >= 0.6.1

4.2.2 Downloads

To obtain a tar.gz or zip archive of the Fabric source code, you may visit either of the following locations:

- The official downloads are available via git.fabfile.org. Our Git repository viewer provides downloads of all tagged releases. See the "Download" column, next to the "Tag" column in the middle of the front page.
- Our GitHub mirror has downloads of all tagged releases as well – just click the 'Download' button near the top of the main page.
- Fabric's PyPI page offers manual downloads as well as being the entry point for *Easy_install and Pip*.

4.2.3 Source code checkouts

The Fabric developers manage the project's source code with the Git DVCS. To follow Fabric's development via Git instead of downloading official releases, you have the following options:

- Clone the canonical Git repository, `git://fabfile.org/fabric.git` (note that a Web view of this repository can be found at git.fabfile.org)
- Clone the official Github mirror/collaboration repository, `git://github.com/bitprophet/fabric.git`
- Make your own fork of the Github repository by making a Github account, visiting GitHub/bitprophet/fabric and clicking the "fork" button.

For information on the hows and whys of Fabric development, including which branches may be of interest and how you can help out, please see the *Development* page.

4.2.4 Easy_install and Pip

Fabric may be installed via either easy_install or pip.

Fabric's source distribution also comes with a `pip` requirements file called `requirements.txt`, containing the various development requirements listed above (note, that's *development* requirements – not necessary for simply using the software.) At time of writing, some of the listed third-party packages don't play well with `pip`, so we aren't officially recommending use of the requirements file just yet.

4.3 Development

The Fabric development team consists of two programmers, Jeff Forcier and Christian Vest Hansen, with Jeff taking the lead role. However, dozens of other developers pitch in by submitting patches and ideas, via individual emails, Redmine, the mailing list and GitHub.

4.3.1 Get the code

Please see the *Source code checkouts* section of the *Installation* page for details on how to obtain Fabric's source code.

4.3.2 Contributing

There are a number of ways to get involved with Fabric:

- **Use Fabric and send us feedback!** This is both the easiest and arguably the most important way to improve the project – let us know how you currently use Fabric and how you want to use it. (Please do try to search the ticket tracker first, though, when submitting feature ideas.)
- **Report bugs.** Pretty much a special case of the previous item: if you think you've found a bug in Fabric, check on the ticket tracker to see if anyone's reported it yet, and if not – file a bug! If possible, try to make sure you can replicate it repeatedly, and let us know the circumstances (what version of Fabric you're using, what platform you're on, and what exactly you were doing when the bug cropped up.)
- **Submit patches or new features.** See the *Source code checkouts* documentation, grab a Git clone of the source, and either email a patch to the mailing list or make your own GitHub fork and send us a pull request. While we may not always reply promptly, we do try to make time eventually to inspect all contributions and either incorporate them or explain why we don't feel the change is a good fit.

Style

Fabric tries hard to honor PEP-8, especially (but not limited to!) the following:

- Keep all lines under 80 characters. This goes for the ReST documentation as well as code itself.
  - Exceptions are made for situations where breaking a long string (such as a string being `print`-ed from source code, or an especially long URL link in documentation) would be kind of a pain.
- Typical Python 4-space (soft-tab) indents. No tabs! No 8 space indents! (No 2- or 3-space indents, for that matter!)
- `CamelCase` class names, but `lowercase_underscore_separated` everything else.

4.3.3 Branching/Repository Layout

While Fabric's development methodology isn't set in stone yet, the following items detail how we currently organize the Git repository and expect to perform merges and so forth. This will be chiefly of interest to those who wish to follow a specific Git branch instead of released versions, or to any contributors.

- We use a combined 'release and feature branches' methodology, where every minor release (e.g. 0.9, 1.0, 1.1, 1.2 etc; see *Releases* below for details on versioning) gets a release branch for bugfixes, and big feature development is performed in a central `master` branch and/or in feature-specific feature branches (e.g. a branch for reworking the internals to be threadsafe, or one for overhauling task dependencies, etc.)
  - At time of writing, this means that Fabric maintains an `0.9` release branch, from which the 0.9 betas are cut, and from which the final release and bugfix releases will continue to be generated from.

  - New features intended for the next major release (Fabric 1.0) will be kept in the `master` branch. Once the 1.0 alpha or beta period begins, this work will be split off into a `1.0` branch and `master` will start forming Fabric 1.1.
  - While we try our best not to commit broken code or change APIs without warning, as with many other open-source projects we can only have a guarantee of stability in the release branches. Only follow `master` if you're willing to deal with a little pain.
  - Conversely, because we try to keep release branches relatively stable, you may find it easier to use Fabric from a source checkout of a release branch instead of upgrading to new released versions. This can provide a decent middle ground between stability and the ability to get bugfixes or backported features easily.
- The core developers will take care of performing merging/branching on the official repositories. Since Git is Git, contributors may of course do whatever they wish in their own clones/forks.
- Bugfixes are to be performed on release branches and then merged into `master` so that `master` is always up-to-date (or nearly so; while it's not mandatory to merge after every bugfix, doing so at least daily is a good idea.)
- Feature branches, if used, should periodically merge in changes from `master` so that when it comes time for them to merge back into `master` things aren't quite as painful.

4.3.4 Releases

Fabric tries to follow open-source standards and conventions in its release tagging, including typical version numbers such as 2.0, 1.2.5, or 1.2-beta1. Each release will be marked as a tag in the Git repositories, and are broken down as follows:

Major

Major releases update the first number, e.g. going from 0.9 to 1.0, and indicate that the software has reached some very large milestone.

For example, the upcoming 1.0 will mean that we feel Fabric has reached its primary design goals of a solid core API and well-defined area for additional functionality to live. Version 2.0 might, for example, indicate a rewrite using a new underlying network technology (though this isn't necessarily planned.)

Major releases will often be backwards-incompatible with the previous line of development, though this is not a requirement, just a usual happenstance. Users should expect to have to make at least some changes to their fabfiles when switching between major versions.

Minor

Minor releases, such as moving from 1.0 to 1.1, typically mean that a new, large feature has been added. They are also sometimes used to mark off the fact that a lot of bug fixes or small feature modifications have occurred since the previous minor release.

These releases are guaranteed to be backwards-compatible with all other releases containing the same major version number, so a fabfile that works with 1.0 should also work fine with 1.1 or even 1.9.

Note: This policy marks a departure from early versions of Fabric, wherein the minor release number was the backwards-compatibility boundary – e.g. Fabric 0.1 was incompatible with Fabric 0.0.x.

Bugfix/tertiary

The third and final part of version numbers, such as the '3' in 1.0.3, generally indicate a release containing one or more bugfixes, although minor feature additions or modifications are also common.

This third number is sometimes omitted for the first major or minor release in a series, e.g. 1.2 or 2.0, and in these cases it can be considered an implicit zero (e.g. 2.0.0). Fabric will likely include the explicit zero in these cases, however – after all, explicit is better than implicit.

4.4 Tutorial

For new users, and/or for an overview of Fabric's basic functionality, please see the *Overview and Tutorial*. The rest of the documentation will assume you're at least passingly familiar with the material contained within.

4.5 Usage documentation

The following list contains all major sections of Fabric's prose (non-API) documentation, which expands upon the concepts outlined in the *Overview and Tutorial* and also covers advanced topics.

4.5.1 The environment dictionary, `env`

A simple but integral aspect of Fabric is what is known as the "environment": a Python dictionary subclass which is used as a combination settings registry and shared inter-task data namespace.

The environment dict is currently implemented as a global singleton, `fabric.state.env`, and is included in `fabric.api` for convenience. Keys in `env` are sometimes referred to as "env variables".

Environment as configuration

Most of Fabric's behavior is controllable by modifying env variables, such as `env.hosts` (as seen in *the tutorial*). Other commonly-modified env vars include:

- `user`: Fabric defaults to your local username when making SSH connections, but you can use `env.user` to override this if necessary. The *Execution model* documentation also has info on how to specify usernames on a per-host basis.
- `password`: Used to explicitly set your connection or sudo password if desired. Fabric will prompt you when necessary if this isn't set or doesn't appear to be valid.
- `warn_only`: a Boolean setting determining whether Fabric exits when detecting errors on the remote end. See *Execution model* for more on this behavior.

There are a number of other env variables; for the full list, see *Full list of env vars* at the bottom of this document.

The `settings` context manager

In many situations, it's useful to only temporarily modify `env` vars so that a given settings change only applies to a block of code. Fabric provides a `settings` context manager, which takes any numbr of key/value pairs and will use them to modify `env` within its wrapped block.

For example, there are many situations where setting `warn_only` (see below) is useful. To apply it to a few lines of code, use `settings(warn_only=True)`, as seen in this simplified version of the `contrib exists` function:

```
from fabric.api import settings, run

def exists(path):
    with settings(warn_only=True):
        return run('test -e %s' % path)
```

See the *Context Managers* API documentation for details on `settings` and other, similar tools.

Environment as shared state

As mentioned, the `env` object is simply a dictionary subclass, so your own fabfile code may store information in it as well. This is sometimes useful for keeping state between multiple tasks within a single execution run.

Note: This aspect of `env` is largely historical: in the past, fabfiles were not pure Python and thus the environment was the only way to communicate between tasks. Nowadays, you may call other tasks or subroutines directly, and even keep module-level shared state if you wish.

In future versions, Fabric will become threadsafe, at which point `env` may be the only easy/safe way to keep global state.

Other considerations

While it subclasses `dict`, Fabric's `env` has been modified so that its values may be read/written by way of attribute access, as seen in some of the above material. In other words, `env.host_string` and `env['host_string']` are functionally identical. We feel that attribute access can often save a bit of typing and makes the code more readable, so it's the recommended way to interact with `env`.

The fact that it's a dictionary can be useful in other ways, such as with Python's dict-based string interpolation, which is especially handy if you need to insert multiple env vars into a single string. Using "normal" string interpolation might look like this:

```
print("Executing on %s as %s" % (env.host, env.user))
```

Using dict-style interpolation is more readable and slightly shorter:

```
print("Executing on %(host)s as %(user)s" % env)
```

Full list of env vars

Below is a list of all predefined (or defined by Fabric itself during execution) environment variables. While any of them may be manipulated directly, it's often best to use `context_managers`, either generally via `settings` or via specific context managers such as `cd`.

Note that many of these may be set via `fab`'s command-line switches – see *fab options and arguments* for details. Cross-links will be provided where appropriate.

`all_hosts`

Default: `None`

Set by `fab` to the full host list for the currently executing command. For informational purposes only.

See also:

Execution model

command

Default: `None`

Set by `fab` to the currently executing command name (e.g. when executed as `$ fab task1 task2`, `env.command` will be set to `"task1"` while `task1` is executing, and then to `"task2"`.) For informational purposes only.

See also:

Execution model

cwd

Default: `''`

Current working directory. Used to keep state for the `cd` context manager.

disable_known_hosts

Default: `False`

If `True`, the SSH layer will skip loading the user's known-hosts file. Useful for avoiding exceptions in situations where a "known host" changing its host key is actually valid (e.g. cloud servers such as EC2.)

See also:

SSH behavior

fabfile

Default: `fabfile.py`

Filename which `fab` searches for when loading fabfiles. Obviously, it doesn't make sense to set this in a fabfile, but it may be specified in a `.fabricrc` file or on the command line.

See also:

fab options and arguments

host_string

Default: `None`

Defines the current user/host/port which Fabric will connect to when executing `run`, `put` and so forth. This is set by `fab` when iterating over a previously set host list, and may also be manually set when using Fabric as a library.

See also:

Execution model

host

Default: `None`

Set to the hostname part of `env.host_string` by `fab`. For informational purposes only.

hosts

Default: `[]`

The global host list used when composing per-task host lists.

See also:

Execution model

key_filename

Default: `None`

May be a string or list of strings, referencing file paths to SSH key files to try when connecting. Passed through directly to the SSH layer.

See also:

Paramiko's documentation for SSHClient.connect()

password

Default: `None`

The password used by the SSH layer when connecting to remote hosts, **and/or** when answering `sudo` prompts.

When empty, the user will be prompted, with the result stored in this env variable and used for connecting/sudoing. (In other words, setting this prior to runtime is not required, though it may be convenient in some cases.)

Given a session where multiple different passwords are used, only the first one will be stored into `env.password`. Put another way, the only time `env.password` is written to by Fabric itself is when it is empty. This may change in the future.

See also:

Execution model

port

Default: `None`

Set to the port part of `env.host_string` by `fab` when iterating over a host list. For informational purposes only.

real_fabfile

Default: `None`

Set by `fab` with the path to the fabfile it has loaded up, if it got that far. For informational purposes only.

See also:

fab options and arguments

rcfile

Default: `$HOME/.fabricrc`

Path used when loading Fabric's local settings file.

See also:

fab options and arguments

reject_unknown_hosts

Default: `False`

If `True`, the SSH layer will raise an exception when connecting to hosts not listed in the user's known-hosts file.

See also:

SSH behavior

roledefs

Default: `{}`

Dictionary defining role name to host list mappings.

See also:

Execution model

roles

Default: `[]`

The global role list used when composing per-task host lists.

See also:

Execution model

shell

Default: `/bin/bash -l -c`

Value used as shell wrapper when executing commands with e.g. `run`. Must be able to exist in the form `<env.shell> "<command goes here>"` – e.g. the default uses Bash's `-c` option which takes a command string as its value.

See also:

Execution model

sudo_prompt

Default: `sudo password:`

Passed to the `sudo` program on remote systems so that Fabric may correctly identify its password prompt. This may be modified by fabfiles but there's no real reason to.

See also:

The `sudo` operation

use_shell

Default: `True`

Global setting which acts like the `use_shell` argument to `run`/`sudo`: if it is set to `False`, operations will not wrap executed commands in `env.shell`.

user

Default: User's local username

The username used by the SSH layer when connecting to remote hosts. May be set globally, and will be used when not otherwise explicitly set in host strings. However, when explicitly given in such a manner, this variable will be temporarily overwritten with the current value – i.e. it will always display the user currently being connected as.

To illustrate this, a fabfile:

```
from fabric.api import env, run

env.user = 'implicit_user'
env.hosts = ['host1', 'explicit_user@host2', 'host3']

def print_user():
    with hide('running'):
        run('echo "%(user)s"' % env)
```

and its use:

```
$ fab print_user

[host1] out: implicit_user
[explicit_user@host2] out: explicit_user
[host3] out: implicit_user

Done.
Disconnecting from host1... done.
Disconnecting from host2... done.
Disconnecting from host3... done.
```

As you can see, during execution on `host2`, `env.user` was set to `"explicit_user"`, but was restored to its previous value (`"implicit_user"`) afterwards.

Note: `env.user` is currently somewhat confusing (it's used for configuration **and** informational purposes) so expect this to change in the future – the informational aspect will likely be broken out into a separate env variable.

See also:

Execution model

version

Default: current Fabric version string

Mostly for informational purposes. Modification is not recommended, but probably won't break anything either.

warn_only

Default: `False`

Specifies whether or not to warn, instead of abort, when `run`/`sudo`/`local` encounter error conditions.

See also:

Execution model

4.5.2 Execution model

If you've read the *Overview and Tutorial*, you should already be familiar with how Fabric operates in the base case (a single task on a single host.) However, in many situations you'll find yourself wanting to execute multiple tasks and/or on multiple hosts. Perhaps you want to split a big task into smaller reusable parts, or crawl a collection of servers looking for an old user to remove. Such a scenario requires specific rules for when and how tasks are executed.

This document explores Fabric's execution model, including the main execution loop, how to define host lists, how connections are made, and so forth.

Note: Most of this material applies to the *fab* tool only, as this mode of use has historically been the main focus of Fabric's development. When writing version 0.9 we straightened out Fabric's internals to make it easier to use as a library, but there's still work to be done before this is as flexible and easy as we'd like it to be.

Execution strategy

Fabric currently provides a single, serial execution method, though more options are planned for the future:

- A list of tasks is created. Currently this list is simply the arguments given to *fab*, preserving the order given.
- For each task, a task-specific host list is generated from various sources (see *How host lists are constructed* below for details.)
- The task list is walked through in order, and each task is run once per host in its host list.
- Tasks with no hosts in their host list are considered local-only, and will always run once and only once.

Thus, given the following fabfile:

```
from fabric.api import run, env

env.hosts = ['host1', 'host2']

def taskA():
    run('ls')

def taskB():
    run('whoami')
```

and the following invocation:

```
$ fab taskA taskB
```

you will see that Fabric performs the following:

- `taskA` executed on `host1`
- `taskA` executed on `host2`
- `taskB` executed on `host1`
- `taskB` executed on `host2`

While this approach is simplistic, it allows for a straightforward composition of task functions, and (unlike tools which push the multi-host functionality down to the individual function calls) enables shell script-like logic where you may introspect the output or return code of a given command and decide what to do next.

Defining tasks

When looking for tasks to execute, Fabric imports your fabfile and will consider any callable object, **except** for the following:

- Callables whose name starts with an underscore (_). In other words, Python's usual "private" convention holds true here.
- Callables defined within Fabric itself. Fabric's own functions such as `run` and `sudo` will not show up in your task list.

Note: To see exactly which callables in your fabfile may be executed via `fab`, use *`fab --list`*.

Imports

Python's `import` statement effectively includes the imported objects in your module's namespace. Since Fabric's fabfiles are just Python modules, this means that imports are also considered as possible tasks, alongside anything defined in the fabfile itself.

Because of this, we strongly recommend that you use the `import module` form of importing, followed by `module.callable()`, which will result in a cleaner fabfile API than doing `from module import callable`.

For example, here's a sample fabfile which uses `urllib.urlopen` to get some data out of a webservice:

```
from urllib import urlopen

from fabric.api import run

def webservice_read():
    objects = urlopen('http://my/web/service/?foo=bar').read().split()
    print(objects)
```

This looks simple enough, and will run without error. However, look what happens if we run *`fab --list`* on this fabfile:

```
$ fab --list
Available commands:

  my_task    List some directories.
  urlopen    urlopen(url [, data]) -> open file-like object
```

Our fabfile of only one task is showing two "tasks", which is bad enough, and an unsuspecting user might accidentally try to call `fab urlopen`, which probably won't work very well. Imagine any real-world fabfile, which is likely to be much more complex, and hopefully you can see how this could get messy fast.

For reference, here's the recommended way to do it:

```
import urllib

from fabric.api import run

def webservice_read():
    objects = urllib.urlopen('http://my/web/service/?foo=bar').read().split()
    print(objects)
```

It's a simple change, but it'll make anyone using your fabfile a bit happier.

Defining host lists

Unless you're using Fabric as a simple build system (which is possible, but not the primary use-case) having tasks won't do you any good without the ability to specify remote hosts on which to execute them. There are a number of ways to do so, with scopes varying from global to per-task, and it's possible mix and match as needed.

Hosts

Hosts, in this context, refer to what are also called "host strings": Python strings specifying a username, hostname and port combination, in the form of `username@hostname:port`. User and/or port (and the associated `@` or `:`) may be omitted, and will be filled by the executing user's local username, and/or port 22, respectively. Thus, `admin@foo.com:222`, `deploy@website` and `nameserver1` could all be valid host strings.

In other words, Fabric expects the same format as the command-line `ssh` program.

During execution, Fabric normalizes the host strings given and then stores each part (username/hostname/port) in the environment dictionary, for both its use and for tasks to reference if the need arises. See *The environment dictionary, env* for details.

Roles

Host strings map to single hosts, but sometimes it's useful to arrange hosts in groups. Perhaps you have a number of Web servers behind a load balancer and want to update all of them, or want to run a task on "all client servers". Roles provide a way of defining strings which correspond to lists of host strings, and can then be specified instead of writing out the entire list every time.

This mapping is defined as a dictionary, `env.roledefs`, which must be modified by a fabfile in order to be used. A simple example:

```
from fabric.api import env

env.roledefs['webservers'] = ['www1', 'www2', 'www3']
```

Since `env.roledefs` is naturally empty by default, you may also opt to re-assign to it without fear of losing any information (provided you aren't loading other fabfiles which also modify it, of course):

```
from fabric.api import env

env.roledefs = {
    'web': ['www1', 'www2', 'www3'],
    'dns': ['ns1', 'ns2']
}
```

Use of roles is not required in any way – it's simply a convenience in situations where you have common groupings of servers.

How host lists are constructed

There are a number of ways to specify host lists, either globally or per-task, and generally these methods override one another instead of merging together (though this may change in future releases.) Each such method is typically split into two parts, one for hosts and one for roles.

Globally, via `env` The most common method of setting hosts or roles is by modifying two key-value pairs in the environment dictionary, *env*: `hosts` and `roles`. The value of these variables is checked at runtime, while constructing each tasks's host list.

Thus, they may be set at module level, which will take effect when the fabfile is imported:

```
from fabric.api import env, run

env.hosts = ['host1', 'host2']

def mytask():
    run('ls /var/www')
```

Such a fabfile, run simply as `fab mytask`, will run `mytask` on `host1` followed by `host2`.

Since the env vars are checked for *each* host, this means that if you have the need, you can actually modify `env` in one task and it will affect all following tasks:

```
from fabric.api import env, run

def set_hosts():
    env.hosts = ['host1', 'host2']

def mytask():
    run('ls /var/www')
```

When run as `fab set_hosts mytask`, `set_hosts` is a "local" task – its own host list is empty – but `mytask` will again run on the two hosts given.

Note: This technique used to be a common way of creating fake "roles", but is less necessary now that roles are fully implemented. It may still be useful in some situations, however.

Alongside `env.hosts` is `env.roles` (not to be confused with `env.roledefs`!) which, if given, will be taken as a list of role names to look up in `env.roledefs`.

Globally, via the command line In addition to modifying `env.hosts` and `env.roles` at the module level, you may define them by passing comma-separated string arguments to the command-line switches *`--hosts/-H`* and *`--roles/-R`*, e.g.:

```
$ fab -H host1,host2 mytask
```

Such an invocation is directly equivalent to `env.hosts = ['host1', 'host2']` – the argument parser knows to look for these arguments and will modify `env` at parse time.

Note: It's possible, and in fact common, to use these switches to set only a single host or role. Fabric simply calls `string.split(',')` on the given string, so a string with no commas turns into a single-item list.

It is important to know that these command-line switches are interpreted **before** your fabfile is loaded: any reassignment to `env.hosts` or `env.roles` in your fabfile will overwrite them.

If you wish to nondestructively merge the command-line hosts with your fabfile-defined ones, make sure your fabfile uses `env.hosts.extend()` instead:

```
from fabric.api import env, run

env.hosts.extend(['host3', 'host4'])

def mytask():
    run('ls /var/www')
```

When this fabfile is run as `fab -H host1,host2 mytask`, `env.hosts` will end contain `['host1', 'host2', 'host3', 'host4']` at the time that `mytask` is executed.

Note: `env.hosts` is simply a Python list object – so you may use `env.hosts.append()` or any other such method you wish.

Per-task, via the command line Globally setting host lists only works if you want all your tasks to run on the same host list all the time. This isn't always true, so Fabric provides a few ways to be more granular and specify host lists which apply to a single task only. The first of these uses task arguments.

As outlined in *fab options and arguments*, it's possible to specify per-task arguments via a special command-line syntax. In addition to naming actual arguments to your task function, this may be used to set the `host`, `hosts`, `role` or `roles` "arguments", which are interpreted by Fabric when building host lists (and removed from the arguments passed to the task itself.)

Note: Since commas are already used to separate task arguments from one another, semicolons must be used in the `hosts` or `roles` arguments to delineate individual host strings or role names. Furthermore, the argument must be quoted to prevent your shell from interpreting the semicolons.

Take the below fabfile, which is the same one we've been using, but which doesn't define any host info at all:

```
from fabric.api import run

def mytask():
    run('ls /var/www')
```

To specify per-task hosts for `mytask`, execute it like so:

```
$ fab mytask:hosts="host1;host2"
```

This will override any other host list and ensure `mytask` always runs on just those two hosts.

Per-task, via decorators If a given task should always run on a predetermined host list, you may wish to specify this in your fabfile itself. This can be done by decorating a task function with the `hosts` or `roles` decorators. These decorators take a variable argument list, like so:

```
from fabric.api import hosts, run

@hosts('host1', 'host2')
def mytask():
    run('ls /var/www')
```

When used, they override any checks of `env` for that particular task's host list (though `env` is not modified in any way – it is simply ignored.) Thus, even if the above fabfile had defined `env.hosts` or the call to *fab* uses *`--hosts/-H`*, `mytask` would still run on a host list of `['host1', 'host2']`.

However, decorator host lists do **not** override per-task command-line arguments, as given in the previous section.

Order of precedence We've been pointing out which methods of setting host lists trump the others, as we've gone along. However, to make things clearer, here's a quick breakdown:

- Per-task, command-line host lists (`fab mytask:host=host1`) override absolutely everything else.
- Per-task, decorator-specified host lists (`@hosts('host1')`) override the `env` variables.
- Globally specified host lists set in the fabfile (`env.hosts = ['host1']`) *can* override such lists set on the command-line, but only if you're not careful (or want them to.)
- Globally specified host lists set on the command-line (`--hosts=host1`) will initialize the `env` variables, but that's it.

This logic may change slightly in the future to be more consistent (e.g. having *`--hosts`* somehow take precedence over `env.hosts` in the same way that command-line per-task lists trump in-code ones) but only in a backwards-incompatible release.

Combining host lists

There is no "unionizing" of hosts between the various sources mentioned in *How host lists are constructed*. If `env.hosts` is set to `['host1', 'host2', 'host3']`, and a per-function (e.g. via `hosts`) host list is set to just `['host2', 'host3']`, that function will **not** execute on `host1`, because the per-task decorator host list takes precedence.

However, for each given source, if both roles **and** hosts are specified, they will be merged together into a single host list. Take, for example, this fabfile where both of the decorators are used:

```
from fabric.api import env, hosts, roles, run

env.roledefs = {'role1': ['b', 'c']}

@hosts('a', 'b')
@roles('role1')
def mytask():
    run('ls /var/www')
```

Assuming no command-line hosts or roles are given when `mytask` is executed, this fabfile will call `mytask` on a host list of `['a', 'b', 'c']` – the union of `role1` and the contents of the `hosts` call.

Failure handling

Once the task list has been constructed, Fabric will start executing them as outlined in *Execution strategy*, until all tasks have been run on the entirety of their host lists. However, Fabric defaults to a "fail-fast" behavior pattern: if anything goes wrong, such as a remote program returning a nonzero return value or your fabfile's Python code encountering an exception, execution will halt immediately.

This is typically the desired behavior, but there are many exceptions to the rule, so Fabric provides `env.warn_only`, a Boolean setting. It defaults to `False`, meaning an error condition will result in the program aborting immediately. However, if `env.warn_only` is set to `True` at the time of failure – with, say, the `settings` context manager – Fabric will emit a warning message but continue executing.

Connections

`fab` itself doesn't actually make any connections to remote hosts. Instead, it simply ensures that for each distinct run of a task on one of its hosts, the env var `env.host_string` is set to the right value. Users wanting to leverage Fabric as a library may do so manually to achieve similar effects.

`env.host_string` is (as the name implies) the "current" host string, and is what Fabric uses to determine what connections to make (or re-use) when network-aware functions are run. Operations like `run` or `put` use `env.host_string` as a lookup key in a shared dictionary which maps host strings to SSH connection objects.

Note: The connections dictionary (currently located at `fabric.state.connections`) acts as a cache, opting to return previously created connections if possible in order to save some overhead, and creating new ones otherwise.

Lazy connections

Because connections are driven by the individual operations, Fabric will not actually make connections until they're necessary. Take for example this task which does some local housekeeping prior to interacting with the remote server:

```
from fabric.api import *

@hosts('host1')
def clean_and_upload():
    local('find assets/ -name "*.DS_Store" -exec rm '{}' \;')
    local('tar czf /tmp/assets.tgz assets/')
    put('/tmp/assets.tgz', '/tmp/assets.tgz')
    with cd('/var/www/myapp/'):
        run('tar xzf /tmp/assets.tgz')
```

What happens, connection-wise, is as follows:

1. The two `local` calls will run without making any network connections whatsoever;
2. `put` asks the connection cache for a connection to `host1`;
3. The connection cache fails to find an existing connection for that host string, and so creates a new SSH connection, returning it to `put`;
4. `put` uploads the file through that connection;
5. Finally, the `run` call asks the cache for a connection to that same host string, and is given the existing, cached connection for its own use.

Extrapolating from this, you can also see that tasks which don't use any network-borne operations will never actually initiate any connections (though they will still be run once for each host in their host list, if any.)

Closing connections

Fabric's connection cache never closes connections itself – it leaves this up to whatever is using it. The *fab* tool does this bookkeeping for you: it iterates over all open connections and closes them just before it exits (regardless of whether the tasks failed or not.)

Library users will need to ensure they explicitly close all open connections before their program exits, though we plan to makes this easier in the future. An example of this can be seen in `the tutorial`.

4.5.3 `fab` options and arguments

The most common method for utilizing Fabric is via its command-line tool, `fab`, which should have been placed on your shell's executable path when Fabric was installed. `fab` tries hard to be a good Unix citizen, using a standard style of command-line switches, help output, and so forth.

Basic use

In its most simple form, `fab` may be called with no options at all, and with one or more arguments, which should be task names, e.g.:

```
$ fab task1 task2
```

As detailed in *Overview and Tutorial* and *Execution model*, this will run `task1` followed by `task2`, assuming that Fabric was able to find a fabfile nearby containing Python functions with those names.

However, it's possible to expand this simple usage into something more flexible, by using the provided options and/or passing arguments to individual tasks.

Command-line options

A quick overview of all possible command line options can be found via `fab --help`. If you're looking for details on a specific option, we go into detail below.

Note: `fab` uses Python's optparse library, meaning that it honors typical Linux or GNU style short and long options, as well as freely mixing options and arguments. E.g. `fab task1 -H hostname task2 -i path/to/keyfile` is just as valid as the more straightforward `fab -H hostname -i path/to/keyfile task1 task2`.

`-h, --help`
Displays a standard help message, with all possible options and a brief overview of what they do, then exits.

`-V, --version`
Displays Fabric's version number, then exits.

`-l, --list`
Imports a fabfile as normal, but then prints a list of all discovered tasks and exits. Will also print the first line of each task's docstring, if it has one, next to it (truncating if necessary.)

`-d` `COMMAND`, `--display`=`COMMAND`
Prints the entire docstring for the given task, if there is one. Does not currently print out the task's function signature, so descriptive docstrings are a good idea. (They're *always* a good idea, of course – just moreso here.)

`-r, --reject-unknown-hosts`
Sets *env.reject_unknown_hosts* to `True`, causing Fabric to abort when connecting to hosts not found in the user's SSH known_hosts file.

-D, --disable-known-hosts
Sets *env.disable_known_hosts* to `True`, preventing Fabric from loading the user's SSH known_hosts file.

-u USER, **--user**=USER
Sets *env.user* to the given string; it will then be used as the default username when making SSH connections.

-p PASSWORD, **--password**=PASSWORD
Sets *env.password* to the given string; it will then be used as the default password when making SSH connections or calling the `sudo` program.

-H HOSTS, **--hosts**=HOSTS
Sets *env.hosts* to the given comma-delimited list of host strings.

-R ROLES, **--roles**=ROLES
Sets *env.roles* to the given comma-separated list of role names.

-i KEY_FILENAME
When set to a file path, will load the given file as an SSH identity file (usually a private key.) This option may be repeated multiple times.

-f FABFILE, **--fabfile**=FABFILE
The fabfile name pattern to search for (defaults to `fabfile.py`), or alternately an explicit file path to load as the fabfile (e.g. `/path/to/my/fabfile.py`.)

See also:

Fabfile construction and use

-w, --warn-only
Sets *env.warn_only* to `True`, causing Fabric to continue execution even when commands encounter error conditions.

-s SHELL, **--shell**=SHELL
Sets *env.shell* to the given string, overriding the default shell wrapper used to execute remote commands.

See also:

`run`, `sudo`

-c RCFILE, **--config**=RCFILE
Sets *env.rcfile* to the given file path, which Fabric will try to load on startup and use to update environment variables.

--hide=LEVELS
A comma-separated list of *output levels* to hide by default.

--show=LEVELS
A comma-separated list of *output levels* to show by default.

Per-task arguments

The options given in *Command-line options* apply to the invocation of `fab` as a whole; even if the order is mixed around, options still apply to all given tasks equally. Additionally, since tasks are just Python functions, it's often desirable to pass in arguments to them at runtime.

Answering both these needs is the concept of "per-task arguments", which is a special syntax you can tack onto the end of any task name:

- Use a colon (`:`) to separate the task name from its arguments;
- Use commas (`,`) to separate arguments from one another;
- Use equals signs (=) for keyword arguments, or omit them for positional arguments;

Additionally, since this process involves string parsing, all values will end up as Python strings, so plan accordingly. (We hope to improve upon this in future versions of Fabric, provided an intuitive syntax can be found.)

For example, a "create a new user" task might be defined like so (omitting the actual logic for brevity):

```
def new_user(username, admin='no'):
    pass
```

You can specify just the username:

```
$ fab new_user:myusername
```

Or treat it as an explicit keyword argument:

```
$ fab new_user:username=myusername
```

If both args are given, you can again give them as positional args:

```
$ fab new_user:myusername,yes
```

Or mix and match, just like in Python:

```
$ fab new_user:myusername,admin=yes
```

All of the above are translated into the expected Python function calls. For example, the last call above would become:

```
>>> new_user('myusername', admin='yes')
```

Roles and hosts

As mentioned in *the section on task execution*, there are a handful of per-task keyword arguments (`host`, `hosts`, `role` and `roles`) which do not actually map to the task functions themselves, but are used for setting per-task host and/or role lists.

These special kwargs are **removed** from the args/kwargs sent to the task function itself; this is so that you don't run into TypeErrors if your task doesn't define the kwargs in question. (It also means that if you **do** define arguments with these names, you won't be able to specify them in this manner – a regrettable but necessary sacrifice.)

Note: If both the plural and singular forms of these kwargs are given, the value of the plural will win out and the singular will be discarded.

When using the plural form of these arguments, one must use semicolons (`;`) since commas are already being used to separate arguments from one another. Furthermore, since your shell is likely to consider semicolons a special character, you'll want to quote the host list string to prevent shell interpretation, e.g.:

```
$ fab new_user:myusername,hosts="host1;host2"
```

Again, since the `hosts` kwarg is removed from the argument list sent to the `new_user` task function, the actual Python invocation would be `new_user('myusername')`, and the function would be executed on a host list of `['host1', 'host2']`.

Settings files

Fabric currently honors a simple user settings file, or `fabricrc` (think `bashrc` but for `fab`) which should contain one or more key-value pairs, one per line. These lines will be subject to `string.split('=')`, and thus can currently only be used to specify string settings. Any such key-value pairs will be used to update *env* when `fab` runs, and is loaded prior to the loading of any fabfile.

By default, Fabric looks for `~/.fabricrc`, and this may be overridden by specifying the `-c` flag to `fab`.

For example, if your typical SSH login username differs from your workstation username, and you don't want to modify `env.user` in a project's fabfile (possibly because you expect others to use it as well) you could write a `fabricrc` file like so:

```
user = ssh_user_name
```

Then, when running `fab`, your fabfile would load up with `env.user` set to `'ssh_user_name'`. Other users of that fabfile could do the same, allowing the fabfile itself to be cleanly agnostic regarding the default username.

4.5.4 Fabfile construction and use

This document contains miscellaneous sections about fabfiles, both how to best write them, and how to use them once written.

Fabfile discovery

By default, `fab` attempts to find a file named `fabfile.py`, in the invoking user's current working directory or any parent directories. Thus, it is oriented around "project" use, where one keeps a `fabfile.py` at the root of a source code tree. Such a fabfile will then be discovered no matter where in the source code the user invokes `fab`.

The specific name of the fabfile may be overridden on the command-line with the `-f` option, or by adding a line setting the value of `fabfile` to one's *fabricrc*. For example, if you wanted to name your fabfile `fab_tasks.py`, you could create such a file and then call `fab -f fab_tasks.py <task name>`, or add `fabfile = fab_tasks.py` to `~/.fabricrc`.

If the given fabfile name contains path elements other than a filename (e.g. `../fabfile.py` or `/dir1/dir2/other.py`) it will be treated as a file path and directly checked for existence without any sort of searching. When in this mode, tile-expansion will be applied, so one may refer to e.g. `~/personal_fabfile.py`.

Note: Fabric does a normal `import` (actually an `__import__`) of your fabfile in order to access its contents – it does not do any `eval`-ing or similar. In order for this to work, Fabric temporarily adds the found fabfile's containing folder to the Python load path (and removes it immediately afterwards.)

Importing Fabric

Because Fabric is just Python, you *can* import its components any way you want. However, for the purposes of encapsulation and convenience (and to make life easier for Fabric's packaging script) Fabric's public API is maintained in the `fabric.api` module.

All of Fabric's *Operations*, *Context Managers*, *Decorators* and *Utils* are included in this module as a single, flat namespace. This enables a very simple and consistent interface to Fabric within your fabfiles:

```
from fabric.api import *

# call run(), sudo(), etc etc
```

This is not technically best practices (for a number of reasons) and if you're only using a couple of Fab API calls, it *is* probably a good idea to explicitly `from fabric.api import env, run` or similar. However, in most nontrivial fabfiles, you'll be using all or most of the API, and the star import:

```
from fabric.api import *
```

will be a lot easier to write and read than:

```
from fabric.api import abort, cd, env, get, hide, hosts, local, prompt, \
    put, require, roles, run, runs_once, settings, show, sudo, warn
```

so in this case we feel pragmatism overrides best practices.

Defining tasks and importing callables

For important information on what exactly Fabric will consider as a task when it loads your fabfile, as well as notes on how best to import other code, please see *Defining tasks* in the *Execution model* documentation.

4.5.5 Managing output

The `fab` tool is very verbose by default and prints out almost everything it can, including the remote end's stderr and stdout streams, the command strings being executed, and so forth. While this is necessary in many cases in order to know just what's going on, any nontrivial Fabric task will quickly become difficult to follow as it runs.

Output levels

To aid in organizing task output, Fabric output is grouped into a number of non-overlapping levels or groups, each of which may be turned on or off independently. This provides flexible control over what is displayed to the user.

Note: All levels, save for `debug`, are on by default.

The standard, atomic output levels/groups are as follows:

- **status**: Status messages, i.e. noting when Fabric is done running, if the user used a keyboard interrupt, or when servers are disconnected from. These messages are almost always relevant and rarely verbose.
- **aborts**: Abort messages. Like status messages, these should really only be turned off when using Fabric as a library, and possibly not even then. Note that even if this output group is turned off, aborts will still occur – there just won't be any output about why Fabric aborted!
- **warnings**: Warning messages. These are often turned off when one expects a given operation to fail, such as when using `grep` to test existence of text in a file. If paired with setting `env.warn_only` to True, this can result in fully silent warnings when remote programs fail. As with `aborts`, this setting does not control actual warning behavior, only whether warning messages are printed or hidden.
- **running**: Printouts of commands being executed or files transferred, e.g. `[myserver] run: ls /var/www`.
- **stdout**: Local, or remote, stdout, i.e. non-error output from commands.
- **stderr**: Local, or remote, stderr, i.e. error-related output from commands.

Debug output

There is a final atomic output level, `debug`, which behaves slightly differently from the rest:

- **debug**: Turn on debugging (which is off by default.) Currently, this is largely used to view the "full" commands being run; take for example this `run` call:

  ```
  run('ls "/home/username/Folder Name With Spaces/"')
  ```

  Normally, the `running` line will show exactly what is passed into `run`, like so:

```
[hostname] run: ls "/home/username/Folder Name With Spaces/"
```

With `debug` on, and assuming you've left *shell* set to `True`, you will see the literal, full string as passed to the remote server:

```
[hostname] run: /bin/bash -l -c "ls \"/home/username/Folder Name With Spaces\""
```

Note: Where modifying other pieces of output (such as in the above example where it modifies the 'running' line to show the shell and any escape characters), this setting takes precedence over the others; so if `running` is False but `debug` is True, you will still be shown the 'running' line in its debugging form.

Output level aliases

In addition to the atomic/standalone levels above, Fabric also provides a couple of convenience aliases which map to multiple other levels. These may be referenced anywhere the other levels are referenced, and will effectively toggle all of the levels they are mapped to.

- **output**: Maps to both `stdout` and `stderr`. Useful for when you only care to see the 'running' lines and your own print statements (and warnings).
- **everything**: Includes `warnings`, `running` and `output` (see above.) Thus, when turning off `everything`, you will only see a bare minimum of output (just `status` and `debug` if it's on), along with your own print statements.

Hiding and/or showing output levels

You may toggle any of Fabric's output levels in a number of ways; for examples, please see the API docs linked in each bullet point:

- **Direct modification of fabric.state.output**: `fabric.state.output` is a dictionary subclass (similar to *env*) whose keys are the output level names, and whose values are either True (show that particular type of output) or False (hide it.)

  `fabric.state.output` is the lowest-level implementation of output levels and is what Fabric's internals reference when deciding whether or not to print their output.

- **Context managers**: `hide` and `show` are twin context managers that take one or more output level names as strings, and either hide or show them within the wrapped block. As with Fabric's other context managers, the prior values are restored when the block exits.

  See also:

  `settings`, which can nest calls to `hide` and/or `show` inside itself.

- **Command-line arguments**: You may use the `--hide` and/or `--show` arguments to *fab options and arguments*, which behave exactly like the context managers of the same names (but are, naturally, globally applied) and take comma-separated strings as input.

4.5.6 SSH behavior

Fabric currently makes use of the Paramiko SSH library for managing all connections, meaning that there are occasionally spots where it is limited by Paramiko's capabilities. Below are areas of note where Fabric will exhibit behavior that isn't consistent with, or as flexible as, the behavior of the `ssh` command-line program.

Unknown hosts

SSH's host key tracking mechanism keeps tabs on all the hosts you attempt to connect to, and maintains a `~/.ssh/known_hosts` file with mappings between identifiers (IP address, sometimes with a hostname as well) and SSH keys. (For details on how this works, please see the OpenSSH documentation.)

Paramiko is capable of loading up your `known_hosts` file, and will then compare any host it connects to, with that mapping. Settings are available to determine what happens when an unknown host (a host whose username or IP is not found in `known_hosts`) is seen:

- **Reject**: the host key is rejected and the connection is not made. This results in a Python exception, which will terminate your Fabric session with a message that the host is unknown.
- **Add**: the new host key is added to the in-memory list of known hosts, the connection is made, and things continue normally. Note that this does **not** modify your on-disk `known_hosts` file!
- **Ask**: not yet implemented at the Fabric level, this is a Paramiko option which would result in the user being prompted about the unknown key and whether to accept it.

Whether to reject or add hosts, as above, is controlled in Fabric via the *env.reject_unknown_hosts* option, which is False by default for convenience's sake. We feel this is a valid tradeoff between convenience and security; anyone who feels otherwise can easily modify their fabfiles at module level to set `env.reject_unknown_hosts = True`.

Known hosts with changed keys

The point of SSH's key/fingerprint tracking is so that man-in-the-middle attacks can be detected: if an attacker redirects your SSH traffic to a computer under his control, and pretends to be your original destination server, the host keys will not match. Thus, the default behavior of SSH – and Paramiko – is to immediately abort the connection when a host previously recorded in `known_hosts` suddenly starts sending us a different host key.

In some edge cases such as some EC2 deployments, you may want to ignore this potential problem. Paramiko, at the time of writing, doesn't give us control over this exact behavior, but we can sidestep it by simply skipping the loading of `known_hosts` – if the host list being compared to is empty, then there's no problem. Set *env.disable_known_hosts* to True when you want this behavior; it is False by default, in order to preserve default SSH behavior.

> **Warning:** Enabling *env.disable_known_hosts* will leave you wide open to man-in-the-middle attacks! Please use with caution.

4.6 API documentation

Fabric maintains two sets of API documentation, autogenerated from the source code's docstrings (which are typically very thorough.)

4.6.1 Core API

The **core** API is loosely defined as those functions, classes and methods which form the basic building blocks of Fabric (such as `run` and `sudo`) upon which everything else (the below "contrib" section, and user fabfiles) builds.

Context Managers

Context managers for use with the `with` statement.

Note: When using Python 2.5, you will need to start your fabfile with `from __future__ import with_statement` in order to make use of the `with` statement (which is a regular, non `__future__` feature of Python 2.6+.)

`fabric.context_managers.`**`cd`**(*path*)

Context manager that keeps directory state when calling `run`/`sudo`.

Any calls to `run` or `sudo` within the wrapped block will implicitly have a string similar to `"cd <path> && "` prefixed in order to give the sense that there is actually statefulness involved.

Because use of `cd` affects all `run` and `sudo` invocations, any code making use of `run` and/or `sudo`, such as much of the `contrib` section, will also be affected by use of `cd`. However, at this time, `get` and `put` do not honor `cd`; we expect this to be fixed in future releases.

Like the actual 'cd' shell builtin, `cd` may be called with relative paths (keep in mind that your default starting directory is your remote user's `$HOME`) and may be nested as well.

Below is a "normal" attempt at using the shell 'cd', which doesn't work due to how shell-less SSH connections are implemented – state is **not** kept between invocations of `run` or `sudo`:

```
run('cd /var/www')
run('ls')
```

The above snippet will list the contents of the remote user's `$HOME` instead of `/var/www`. With `cd`, however, it will work as expected:

```
with cd('/var/www'):
    run('ls') # Turns into "cd /var/www && ls"
```

Finally, a demonstration (see inline comments) of nesting:

```
with cd('/var/www'):
    run('ls') # cd /var/www && ls
    with cd('website1'):
        run('ls') # cd /var/www/website1 && ls
```

Note: This context manager is currently implemented by appending to (and, as always, restoring afterwards) the current value of an environment variable, `env.cwd`. However, this implementation may change in the future, so we do not recommend manually altering `env.cwd` – only the *behavior* of `cd` will have any guarantee of backwards compatibility.

`fabric.context_managers.`**`hide`**(**groups*)

Context manager for setting the given output `groups` to False.

`groups` must be one or more strings naming the output groups defined in `output`. The given groups will be set to False for the duration of the enclosed block, and restored to their previous value afterwards.

For example, to hide the "[hostname] run:" status lines, as well as preventing printout of stdout and stderr, one might use `hide` as follows:

```
def my_task():
    with hide('running', 'stdout', 'stderr'):
        run('ls /var/www')
```

`fabric.context_managers.`**`settings`**(**args*, ***kwargs*)

Nest context managers and/or override `env` variables.

`settings` serves two purposes:

- Most usefully, it allows temporary overriding/updating of `env` with any provided keyword arguments, e.g. `with settings(user='foo'):`. Original values, if any, will be restored once the `with` block closes.
- In addition, it will use contextlib.nested to nest any given non-keyword arguments, which should be other context managers, e.g. `with settings(hide('stderr'), show('stdout')):`.

These behaviors may be specified at the same time if desired. An example will hopefully illustrate why this is considered useful:

```
def my_task():
    with settings(
        hide('warnings', 'running', 'stdout', 'stderr'),
        warn_only=True
    ):
        if run('ls /etc/lsb-release'):
            return 'Ubuntu'
        elif run('ls /etc/redhat-release'):
            return 'RedHat'
```

The above task executes a `run` statement, but will warn instead of aborting if the `ls` fails, and all output – including the warning itself – is prevented from printing to the user. The end result, in this scenario, is a completely silent task that allows the caller to figure out what type of system the remote host is, without incurring the handful of output that would normally occur.

Thus, `settings` may be used to set any combination of environment variables in tandem with hiding (or showing) specific levels of output, or in tandem with any other piece of Fabric functionality implemented as a context manager.

`fabric.context_managers.`**`show`**(**groups*)

Context manager for setting the given output `groups` to True.

`groups` must be one or more strings naming the output groups defined in `output`. The given groups will be set to True for the duration of the enclosed block, and restored to their previous value afterwards.

For example, to turn on debug output (which is typically off by default):

```
def my_task():
    with show('debug'):
        run('ls /var/www')
```

As almost all output groups are displayed by default, `show` is most useful for turning on the normally-hidden `debug` group, or when you know or suspect that code calling your own code is trying to hide output with `hide`.

Decorators

Convenience decorators for use in fabfiles.

`fabric.decorators.`**`hosts`**(**host_list*)

Decorator defining which host or hosts to execute the wrapped function on.

For example, the following will ensure that, barring an override on the command line, `my_func` will be run on `host1`, `host2` and `host3`, and with specific users on `host1` and `host3`:

```
@hosts('user1@host1', 'host2', 'user2@host3')
def my_func():
    pass
```

Note that this decorator actually just sets the function's `.hosts` attribute, which is then read prior to executing the function.

fabric.decorators.**roles**(**role_list*)
Decorator defining a list of role names, used to look up host lists.

A role is simply defined as a key in `env` whose value is a list of one or more host connection strings. For example, the following will ensure that, barring an override on the command line, `my_func` will be executed against the hosts listed in the `webserver` and `dbserver` roles:

```
env.roledefs.update({
    'webserver': ['www1', 'www2'],
    'dbserver': ['db1']
})

@roles('webserver', 'dbserver')
def my_func():
    pass
```

Note that this decorator actually just sets the function's `.roles` attribute, which is then read prior to executing the function.

fabric.decorators.**runs_once**(*func*)
Decorator preventing wrapped function from running more than once.

By keeping internal state, this decorator allows you to mark a function such that it will only run once per Python interpreter session, which in typical use means "once per invocation of the `fab` program".

Any function wrapped with this decorator will silently fail to execute the 2nd, 3rd, ..., Nth time it is called, and will return None in that instance.

Operations

Functions to be used in fabfiles and other non-core code, such as run()/sudo().

fabric.operations.**sudo**(*command*, *shell=True*, *user=None*, *pty=False*)
Run a shell command on a remote host, with superuser privileges.

As with `run()`, `sudo()` executes within a shell command defaulting to the value of `env.shell`, although it goes one step further and wraps the command with `sudo` as well. Like `run`, this behavior may be disabled by specifying `shell=False`.

You may specify a `user` keyword argument, which is passed to `sudo` and allows you to run as some user other than root (which is the default). On most systems, the `sudo` program can take a string username or an integer userid (uid); `user` may likewise be a string or an int.

Some remote systems may be configured to disallow sudo access unless a terminal or pseudoterminal is being used (e.g. when `Defaults requiretty` exists in `/etc/sudoers`.) If updating the remote system's `sudoers` configuration is not possible or desired, you may pass `pty=True` to `sudo` to force allocation of a pseudo tty on the remote end.

`sudo` will return the result of the remote program's stdout as a single (likely multiline) string. This string will exhibit a `failed` boolean attribute specifying whether the command failed or succeeded, and will also include the return code as the `return_code` attribute.

Examples:

```
sudo("~/install_script.py")
sudo("mkdir /var/www/new_docroot", user="www-data")
sudo("ls /home/jdoe", user=1001)
result = sudo("ls /tmp/")
```

fabric.operations.**put**(*local_path*, *remote_path*, *mode=None*)
Upload one or more files to a remote host.

`local_path` may be a relative or absolute local file path, and may contain shell-style wildcards, as understood by the Python `glob` module. Tilde expansion (as implemented by `os.path.expanduser`) is also performed.

`remote_path` may also be a relative or absolute location, but applied to the remote host. Relative paths are relative to the remote user's home directory, but tilde expansion (e.g. `~/.ssh/`) will also be performed if necessary.

By default, `put` preserves file modes when uploading. However, you can also set the mode explicitly by specifying the `mode` keyword argument, which sets the numeric mode of the remote file. See the `os.chmod` documentation or `man chmod` for the format of this argument.

Examples:

```
put('bin/project.zip', '/tmp/project.zip')
put('*.py', 'cgi-bin/')
put('index.html', 'index.html', mode=0755)
```

fabric.operations.**run**(*command*, *shell=True*, *pty=False*)
Run a shell command on a remote host.

If `shell` is True (the default), `run()` will execute the given command string via a shell interpreter, the value of which may be controlled by setting `env.shell` (defaulting to something similar to `/bin/bash -l -c "<command>"`.) Any double-quote (`"`) characters in `command` will be automatically escaped when `shell` is True.

`run` will return the result of the remote program's stdout as a single (likely multiline) string. This string will exhibit a `failed` boolean attribute specifying whether the command failed or succeeded, and will also include the return code as the `return_code` attribute.

You may pass `pty=True` to force allocation of a pseudo tty on the remote end. This is not normally required, but some programs may complain (or, even more rarely, refuse to run) if a tty is not present.

Examples:

```
run("ls /var/www/")
run("ls /home/myuser", shell=False)
output = run('ls /var/www/site1')
```

fabric.operations.**get**(*remote_path*, *local_path*)
Download a file from a remote host.

`remote_path` should point to a specific file, while `local_path` may be a directory (in which case the remote filename is preserved) or something else (in which case the downloaded file is renamed). Tilde expansion is performed on both ends.

For example, `get('~/info.txt', '/tmp/')` will create a new file, `/tmp/info.txt`, because `/tmp` is a directory. However, a call such as `get('~/info.txt', '/tmp/my_info.txt')` would result in a new file named `/tmp/my_info.txt`, as that path didn't exist (and thus wasn't a directory.)

If `local_path` names a file that already exists locally, that file will be overwritten without complaint.

Finally, if `get` detects that it will be run on more than one host, it will suffix the current host string to the local filename, to avoid clobbering when it is run multiple times.

For example, the following snippet will produce two files on your local system, called `server.log.host1` and `server.log.host2` respectively:

```
@hosts('host1', 'host2')
def my_download_task():
    get('/var/log/server.log', 'server.log')
```

However, with a single host (e.g. `@hosts('host1')`), no suffixing is performed, leaving you with a single, pristine `server.log`.

`fabric.operations.`**`local`**(*command*, *capture=True*)

Run a command on the local system.

`local` is simply a convenience wrapper around the use of the builtin Python `subprocess` module with `shell=True` activated. If you need to do anything special, consider using the `subprocess` module directly.

`local` will, by default, capture and return the contents of the command's stdout as a string, and will not print anything to the user (the command's stderr is captured but discarded.)

Note: This differs from the default behavior of `run` and `sudo` due to the different mechanisms involved: it is difficult to simultaneously capture and print local commands, so we have to choose one or the other. We hope to address this in later releases.

If you need full interactivity with the command being run (and are willing to accept the loss of captured stdout) you may specify `capture=False` so that the subprocess' stdout and stderr pipes are connected to your terminal instead of captured by Fabric.

When `capture` is False, global output controls (`output.stdout` and `output.stderr` will be used to determine what is printed and what is discarded.

`fabric.operations.`**`prompt`**(*text*, *key=None*, *default=''*, *validate=None*)

Prompt user with `text` and return the input (like `raw_input`).

A single space character will be appended for convenience, but nothing else. Thus, you may want to end your prompt text with a question mark or a colon, e.g. `prompt("What hostname?")`.

If `key` is given, the user's input will be stored as `env.<key>` in addition to being returned by `prompt`. If the key already existed in `env`, its value will be overwritten and a warning printed to the user.

If `default` is given, it is displayed in square brackets and used if the user enters nothing (i.e. presses Enter without entering any text). `default` defaults to the empty string. If non-empty, a space will be appended, so that a call such as `prompt("What hostname?", default="foo")` would result in a prompt of `What hostname? [foo]` (with a trailing space after the `[foo]`.)

The optional keyword argument `validate` may be a callable or a string:

- If a callable, it is called with the user's input, and should return the value to be stored on success. On failure, it should raise an exception with an exception message, which will be printed to the user.
- If a string, the value passed to `validate` is used as a regular expression. It is thus recommended to use raw strings in this case. Note that the regular expression, if it is not fully matching (bounded by `^` and `$`) it will be made so. In other words, the input must fully match the regex.

Either way, `prompt` will re-prompt until validation passes (or the user hits `Ctrl-C`).

Examples:

```
# Simplest form:
environment = prompt('Please specify target environment: ')

# With default, and storing as env.dish:
prompt('Specify favorite dish: ', 'dish', default='spam & eggs')

# With validation, i.e. requiring integer input:
```

```
prompt('Please specify process nice level: ', key='nice', validate=int)

# With validation against a regular expression:
release = prompt('Please supply a release name',
        validate=r'^\w+-\d+(\.\d+)?$')
```

fabric.operations.**require**(**keys*, ***kwargs*)

Check for given keys in the shared environment dict and abort if not found.

Positional arguments should be strings signifying what env vars should be checked for. If any of the given arguments do not exist, Fabric will abort execution and print the names of the missing keys.

The optional keyword argument `used_for` may be a string, which will be printed in the error output to inform users why this requirement is in place. `used_for` is printed as part of a string similar to:

```
"Th(is|ese) variable(s) (are|is) used for %s"
```

so format it appropriately.

The optional keyword argument `provided_by` may be a list of functions or function names which the user should be able to execute in order to set the key or keys; it will be included in the error output if requirements are not met.

Note: it is assumed that the keyword arguments apply to all given keys as a group. If you feel the need to specify more than one `used_for`, for example, you should break your logic into multiple calls to `require()`.

Utils

Internal subroutines for e.g. aborting execution with an error message, or performing indenting on multiline output.

fabric.utils.**abort**(*msg*)

Abort execution, printing given message and exiting with error status. When not invoked as the `fab` command line tool, raise an exception instead.

fabric.utils.**indent**(*text*, *spaces=4*, *strip=False*)

Returns text indented by the given number of spaces.

If text is not a string, it is assumed to be a list of lines and will be joined by `\n` prior to indenting.

When `strip` is `True`, a minimum amount of whitespace is removed from the left-hand side of the given string (so that relative indents are preserved, but otherwise things are left-stripped). This allows you to effectively "normalize" any previous indentation for some inputs.

fabric.utils.**warn**(*msg*)

Print warning message, but do not abort execution.

4.6.2 Contrib API

Fabric's **contrib** package contains commonly useful tools (often merged in from user fabfiles) for tasks such as user I/O, modifying remote files, and so forth. While the core API is likely to remain small and relatively unchanged over time, this contrib section will grow and evolve (while trying to remain backwards-compatible) as more use-cases are solved and added.

Console Output Utilities

Console/terminal user interface functionality.

fabric.contrib.console.**confirm**(*question*, *default=True*)
Ask user a yes/no question and return their response as True or False.

`question` should be a simple, grammatically complete question such as "Do you wish to continue?", and will have a string similar to " [Y/n] " appended automatically. This function will *not* append a question mark for you.

By default, when the user presses Enter without typing anything, "yes" is assumed. This can be changed by specifying `default=False`.

File and Directory Management

Module providing easy API for working with remote files and folders.

fabric.contrib.files.**append**(*text*, *filename*, *use_sudo=False*)
Append string (or list of strings) `text` to `filename`.

When a list is given, each string inside is handled independently (but in the order given.)

If `text` is already found as a discrete line in `filename`, the append is not run, and None is returned immediately. Otherwise, the given text is appended to the end of the given `filename` via e.g. `echo '$text' >> $filename`.

Because `text` is single-quoted, single quotes will be transparently backslash-escaped.

If `use_sudo` is True, will use `sudo` instead of `run`.

fabric.contrib.files.**comment**(*filename*, *regex*, *use_sudo=False*, *char='#'*, *backup='.bak'*)
Attempt to comment out all lines in `filename` matching `regex`.

The default commenting character is `#` and may be overridden by the `char` argument.

This function uses the `sed` function, and will accept the same `use_sudo` and `backup` keyword arguments that `sed` does.

`comment` will prepend the comment character to the beginning of the line, so that lines end up looking like so:

```
this line is uncommented
#this line is commented
#   this line is indented and commented
```

In other words, comment characters will not "follow" indentation as they sometimes do when inserted by hand. Neither will they have a trailing space unless you specify e.g. `char='# '`.

Note: In order to preserve the line being commented out, this function will wrap your `regex` argument in parentheses, so you don't need to. It will ensure that any preceding/trailing `^` or `$` characters are correctly moved outside the parentheses. For example, calling `comment(filename, r'^foo$')` will result in a `sed` call with the "before" regex of `r'^(foo)$'` (and the "after" regex, naturally, of `r'#\1'`.)

---

fabric.contrib.files.**contains**(*text*, *filename*, *exact=False*, *use_sudo=False*)
Return True if `filename` contains `text`.

By default, this function will consider a partial line match (i.e. where the given text only makes up part of the line it's on). Specify `exact=True` to change this behavior so that only a line containing exactly `text` results in a True return value.

Double-quotes in either `text` or `filename` will be automatically backslash-escaped in order to behave correctly during the remote shell invocation.

If `use_sudo` is True, will use `sudo` instead of `run`.

fabric.contrib.files.**exists**(*path*, *use_sudo=False*, *verbose=False*)
Return True if given path exists on the current remote host.

If use_sudo is True, will use sudo instead of run.

exists will, by default, hide all output (including the run line, stdout, stderr and any warning resulting from the file not existing) in order to avoid cluttering output. You may specify verbose=True to change this behavior.

fabric.contrib.files.**first**(**args*, ***kwargs*)
Given one or more file paths, returns first one found, or None if none exist. May specify use_sudo which is passed to exists.

fabric.contrib.files.**sed**(*filename*, *before*, *after*, *limit=''*, *use_sudo=False*, *backup='.bak'*)
Run a search-and-replace on filename with given regex patterns.

Equivalent to sed -i<backup> -r -e "/<limit>/ s/<before>/<after>/g <filename>".

For convenience, before and after will automatically escape forward slashes (and **only** forward slashes) for you, so you don't need to specify e.g. http:\/\/foo\.com, instead just using http://foo\.com is fine.

If use_sudo is True, will use sudo instead of run.

sed will pass shell=False to run/sudo, in order to avoid problems with many nested levels of quotes and backslashes.

fabric.contrib.files.**uncomment**(*filename*, *regex*, *use_sudo=False*, *char='#'*, *backup='.bak'*)
Attempt to uncomment all lines in filename matching regex.

The default comment delimiter is # and may be overridden by the char argument.

This function uses the sed function, and will accept the same use_sudo and backup keyword arguments that sed does.

uncomment will remove a single whitespace character following the comment character, if it exists, but will preserve all preceding whitespace. For example, # foo would become foo (the single space is stripped) but '' # foo'' would become '' foo'' (the single space is still stripped, but the preceding 4 spaces are not.)

fabric.contrib.files.**upload_template**(*filename*, *destination*, *context=None*, *use_jinja=False*, *template_dir=None*, *use_sudo=False*)
Render and upload a template text file to a remote host.

filename should be the path to a text file, which may contain Python string interpolation formatting and will be rendered with the given context dictionary context (if given.)

Alternately, if use_jinja is set to True and you have the Jinja2 templating library available, Jinja will be used to render the template instead. Templates will be loaded from the invoking user's current working directory by default, or from template_dir if given.

The resulting rendered file will be uploaded to the remote file path destination (which should include the desired remote filename.) If the destination file already exists, it will be renamed with a .bak extension.

By default, the file will be copied to destination as the logged-in user; specify use_sudo=True to use sudo instead.

Project Tools

Useful non-core functionality, e.g. functions composing multiple operations.

fabric.contrib.project.**rsync_project**(**args*, ***kwargs*)
Synchronize a remote directory with the current project directory via rsync.

Where `upload_project()` makes use of `scp` to copy one's entire project every time it is invoked, `rsync_project()` uses the `rsync` command-line utility, which only transfers files newer than those on the remote end.

`rsync_project()` is thus a simple wrapper around `rsync`; for details on how `rsync` works, please see its manpage. `rsync` must be installed on both your local and remote systems in order for this operation to work correctly.

This function makes use of Fabric's `local()` operation, and returns the output of that function call; thus it will return the stdout, if any, of the resultant `rsync` call.

`rsync_project()` takes the following parameters:

- `remote_dir`: the only required parameter, this is the path to the **parent** directory on the remote server; the project directory will be created inside this directory. For example, if one's project directory is named `myproject` and one invokes `rsync_project('/home/username/')`, the resulting project directory will be `/home/username/myproject/`.
- `local_dir`: by default, `rsync_project` uses your current working directory as the source directory; you may override this with `local_dir`, which should be a directory path.
- `exclude`: optional, may be a single string, or an iterable of strings, and is used to pass one or more `--exclude` options to `rsync`.
- `delete`: a boolean controlling whether `rsync`'s `--delete` option is used. If True, instructs `rsync` to remove remote files that no longer exist locally. Defaults to False.
- `extra_opts`: an optional, arbitrary string which you may use to pass custom arguments or options to `rsync`.

For reference, the approximate `rsync` command-line call that is constructed by this function is the following:

> **rsync [–delete] [–exclude exclude[0][, –exclude[1][, ...]]]** -pthrvz [extra_opts] <local_dir> <host_string>:<remote_dir>

`fabric.contrib.project.`**`upload_project`**`()`

Upload the current project to a remote system, tar/gzipping during the move.

This function makes use of the `/tmp/` directory and the `tar` and `gzip` programs/libraries; thus it will not work too well on Win32 systems unless one is using Cygwin or something similar.

`upload_project` will attempt to clean up the tarfiles when it finishes executing.

4.7 Changes from previous versions

4.7.1 Changes in version 0.9

This document details the various backwards-incompatible changes made during Fabric's rewrite between versions 0.1 and 0.9. The codebase has been almost completely rewritten and reorganized and an attempt has been made to remove "magical" behavior and make things more simple and Pythonic; the `fab` command-line component has also been redone to behave more like a typical Unix program.

Major changes

You'll want to at least skim the entire document, but the primary changes that will need to be made to one's fabfiles are as follows:

Imports

You will need to **explicitly import any and all methods or decorators used**, at the top of your fabfile; they are no longer magically available. Here's a sample fabfile that worked with 0.1 and earlier:

```
@hosts('a', 'b')
def my_task():
    run('ls /var/www')
    sudo('mkdir /var/www/newsite')
```

The above fabfile uses `hosts`, `run` and `sudo`, and so in Fabric 0.9 one simply needs to import those objects from the new API module `fabric.api`:

```
from fabric.api import hosts, run, sudo

@hosts('a', 'b')
def my_task():
    run('ls /var/www')
    sudo('mkdir /var/www/newsite')
```

You may, if you wish, use `from fabric.api import *`, though this is technically not Python best practices; or you may import directly from the Fabric submodules (e.g. `from fabric.decorators import hosts`.) See *Fabfile construction and use* for more information.

Python version

Fabric started out Python 2.5-only, but became largely 2.4 compatible at one point during its lifetime. Fabric is once again **only compatible with Python 2.5 or newer**, in order to take advantage of the various new features and functions available in that version.

With this change we're setting an official policy to support the two most recent stable releases of the Python 2.x line, which at time of writing is 2.5 and 2.6. We feel this is a decent compromise between new features and the reality of operating system packaging concerns. Given that most users use Fabric from their workstations, which are typically more up-to-date than servers, we're hoping this doesn't cut out too many folks.

Finally, note that while we will not officially support a 2.4-compatible version or fork, we may provide a link to such a project if one arises.

Environment/config variables

The `config` object previously used to access and set internal state (including Fabric config options) **has been renamed** to `env`, but otherwise remains mostly the same (it allows both dictionary and object-attribute style access to its data.) `env` resides in the `state` submodule and is importable via `fabric.api`, so where before one might have seen fabfiles like this:

```
def my_task():
    config.foo = 'bar'
```

one will now be explicitly importing the object like so:

```
from fabric.api import env

def my_task():
    env.foo = 'bar'
```

Execution mode

Fabric's default mode of use, in prior versions, was what we called "broad mode": your tasks, as Python code, ran only once, and any calls to functions that made connections (such as `run` or `sudo`) would run once per host in the current host list. We also offered "deep mode", in which your entire task function would run once per host.

In Fabric 0.9, this dichotomy has been removed, and **"deep mode" is the method Fabric uses to perform all operations**. This allows you to treat your Fabfiles much more like regular Python code, including the use of `if` statements and so forth, and allows operations like `run` to unambiguously return the output from the server.

Other modes of execution such as the old "broad mode" may return as Fabric's internals are refactored and expanded, but for now we've simplified things, and broad mode made the most sense as the primary mode of use.

"Lazy" string interpolation

Because of how Fabric used to run in "broad mode" (see previous section) a special string formatting technique – the use of a bash-like dollar sign notation, e.g. `"hostname: $(fab_host)"` – had to be used to allow the current state of execution to be represented in one's operations. **This is no longer necessary and has been removed**. Because your tasks are executed once per host, you may build strings normally (e.g. with the `%` operator) and refer to `env.host_string`, `env.user` and so forth.

For example, Fabric 0.1 had to insert the current username like so:

```
print("Your current username is $(fab_user)")
```

Fabric 0.9 and up simply reference `env` variables as normal:

```
print("Your current username is %s" % env.user)
```

As with the execution modes, a special string interpolation function or method that automatically makes use of `env` values may find its way back into Fabric at some point if a need becomes apparent.

Other backwards-incompatible changes

In no particular order:

- The Fabric config file location used to be `~/.fabric`; in the interests of honoring Unix filename conventions, it's now `~/.fabricrc`.
- The old `config` object (now `env`) had a `getAny` method which took one or more key strings as arguments, and returned the value attached to the first valid key. This method still exists but has been renamed to `first`.
- Environment variables such as `fab_host` have been renamed to simply e.g. `host`. This looks cleaner and feels more natural, and requires less typing. Users will naturally need to be careful not to override these variables, but the same holds true for e.g. Python's builtin methods and types already, so we felt it was worth the tradeoff.
- Fabric's version header is no longer printed every time the program runs; you should now use the standard `--version`/`-V` command-line options to print version and exit.
- The old `about` command has been removed; other Unix programs don't typically offer this. Users can always view the license and warranty info in their respective text files distributed with the software.
- The old `help` command is now the typical Unix options `-h`/`--help`.
  - Furthermore, there is no longer a listing of Fabric's programming API available through the command line – those topics impact fabfile authors, not fab users (even though the former is a subset of the latter) and should stay in the documentation only.

- `prompt`'s primary function is now to return a value to the caller, although it may still optionally store the entered value in `env` as well.
- `prompt` now considers the empty string to be valid input; this allows other functions to wrap `prompt` and handle "empty" input on their own terms.
- In addition to the above changes, `prompt` has been updated to behave more obviously, as its previous behavior was confusing in a few ways:
  - It will now overwrite pre-existing values in the environment dict, but will print a warning to the user if it does so.
  - Additionally, (and this appeared to be undocumented) the `default` argument could take a callable as well as a string, and would simply set the default message to the return value if a callable was given. This seemed to add unnecessary complexity (given that users may call e.g. `prompt(blah, msg, default=my_callable())` so it has been removed.
- When connecting, Fabric used to use the undocumented `fab_pkey` env variable as a method of passing in a Paramiko `PKey` object to the SSH client's `connect` method. This has been removed in favor of an `ssh`-like `-i` option, which allows one to specify a private key file to use; that should generally be enough for most users.
- `download` is now `get` in order to match up with `put` (the name mismatch was due to `get` being the old method of getting env vars.)
- The `noshell` argument to `sudo` (added late in its life to previous Fabric versions) has been renamed to `shell` (defaults to True, so the effective behavior remains the same) and has also been extended to the `run` operation.
  - Additionally, the global `sudo_noshell` option has been renamed to `use_shell` and also applies to both `run` and `sudo`.
- `local_per_host` has been removed, as it only applied to the now-removed "broad mode".
- `load` has been removed; Fabric is now "just Python", so use Python's import mechanisms in order to stitch multiple fabfiles together.
- `abort` is no longer an "operation" *per se* and has been moved to `fabric.utils`. It is otherwise the same as before, taking a single string message, printing it to the user and then calling `sys.exit(1)`.
- `rsyncproject` and `upload_project` have been moved into `fabric.contrib` (specifically, `fabric.contrib.project`), which is intended to be a new tree of submodules for housing "extra" code which may build on top of the core Fabric operations.
- `invoke` has been turned on its head, and is now the `runs_once` decorator (living in `fabric.decorators`). When used to decorate a function, that function will only execute one time during the lifetime of a `fab` run. Thus, where you might have used `invoke` multiple times to ensure a given command only runs once, you may now use `runs_once` to decorate the function and then call it multiple times in a normal fashion.
- It looks like the regex behavior of the `validate` argument to `prompt` was never actually implemented. It now works as advertised.
- Couldn't think of a good reason for `require` to be a decorator *and* a function, and the function is more versatile in terms of where it may be used, so the decorator has been removed.
- As things currently stand with the execution model, the `depends` decorator doesn't make a lot of sense: instead, it's safest/best to simply make "meta" commands that just call whatever chain of "real" commands you need performed for a given overarching task.

  For example, instead of having command A say that it "depends on" command B, create a command C which calls A and B in the right order, e.g.:

```
def build():
    local('make clean all')

def upload():
    put('app.tgz', '/tmp/app.tgz')
    run('tar xzf /tmp/app.tgz')

def symlink():
    run('ln -s /srv/media/photos /var/www/app/photos')

def deploy():
    build()
    upload()
    symlink()
```

Note: The execution model is still subject to change as Fabric evolves. Please don't hesitate to email the list or the developers if you have a use case that needs something Fabric doesn't provide right now!

- Removed the old `fab shell` functionality, since the move to "just Python" should make vanilla `python`/`ipython` usage of Fabric much easier.
  - We may add it back in later as a convenient shortcut to what basically amounts to running `ipython` and performing a handful of `from fabric.foo import bar` calls.
- The undocumented `fab_quiet` option has been replaced by a much more granular set of output controls. For more info, see *Managing output*.

Changes from alpha 1 to alpha 2

The below list was generated by running `git shortlog 0.9a1..0.9a2` and then manually sifting through and editing the resulting commit messages. This will probably occur for the rest of the alphas and betas; we hope to use Sphinx-specific methods of documenting changes once the final release is out the door.

- Various minor tweaks to the (still in-progress) documentation, including one thanks to Curt Micol.
- Added a number of TODO items based on user feedback (thanks!)
- Host information now available in granular form (user, host, port) in the env dict, alongside the full `user@host:port` host string.
- Parsing of host strings is now more lenient when examining the username (e.g. hyphens.)
- User/host info no longer cleared out between commands.
- Tweaked `setup.py` to use `find_packages`. Thanks to Pat McNerthney.
- Added 'capture' argument to `local` to allow local interactive tasks.
- Reversed default value of `local`'s `show_stderr` kwarg; local stderr now prints by default instead of being hidden by default.
- Various internal fabfile tweaks.

Changes from alpha 2 to alpha 3

- Lots of updates to the documentation and TODO
- Added contrib.files with a handful of file-centric subroutines

- Added contrib.console for console UI stuff (so far, just `confirm`)
- Reworked config file mechanisms a bit, added CLI flag for setting it.
- Output controls (including CLI args, documentation) have been added
- Test coverage tweaked and grown a small amount (thanks in part to Peter Ellis)
- Roles overhauled/fixed (more like hosts now)
- Changed `--list` linewrap behavior to truncate instead.
- Make private key passphrase prompting more obvious to users.
- Add `pty` option to `sudo`. Thanks to José Muanis for the tip-off re: get_pty()
- Add CLI argument for setting the shell used in commands (thanks to Steve Steiner)
- Only load host keys when `env.reject_unknown_keys` is True. Thanks to Pat McNerthney.
- And many, many additional bugfixes and behavioral tweaks too small to merit cluttering up this list! Thanks as always to everyone who contributed bugfixes, feedback and/or patches.

Changes from alpha 3 to beta 1

This is closer to being a straight dump of the Git changelog than the previous sections; apologies for the overall change in tense.

- Add autodocs for fabric.contrib.console.
- Minor cleanup to package init and setup.py.
- Handle exceptions with strerror attributes that are None instead of strings.
- contrib.files.append may now take a list of strings if desired.
- Straighten out how prompt() deals with trailing whitespace
- Add 'cd' context manager.
- Update upload_template to correctly handle backing up target directories.
- upload_template() can now use Jinja2 if it's installed and user asks for it.
- Handle case where remote host SSH key doesn't match known_hosts.
- Fix race condition in run/sudo.
- Start fledgling FAQ; extended pty option to run(); related doc tweaks.
- Bring local() in line with run()/sudo() in terms of .failed attribute.
- Add dollar-sign backslash escaping to run/sudo.
- Add FAQ question re: backgrounding processes.
- Extend some of put()'s niceties to get(), plus docstring/comment updates
- Add debug output of chosen fabfile for troubleshooting fabfile discovery.
- Fix Python path bug which sometimes caused Fabric's internal fabfile to pre-empt user's fabfile during load phase.
- Gracefully handle "display" for tasks with no docstring.
- Fix edge case that comes up during some auth/prompt situations.
- Handle carriage returns in output_thread correctly. Thanks to Brian Rosner.

Changes from beta 1 to release candidate 1

As with the previous changelog, this is also mostly a dump of the Git log. We promise that future changelogs will be more verbose :)

- Near-total overhaul and expansion of documentation (this is the big one!) Other mentions of documentation in this list are items deserving their own mention, e.g. FAQ updates.
- Add FAQ question re: passphrase/password prompt
- Vendorized Paramiko: it is now included in our distribution and is no longer an external dependency, at least until upstream fixes a nasty 1.7.5 bug.
- Fix #34: switch upload_template to use mkstemp (also removes Python 2.5.2+ dependency – now works on 2.5.0 and up)
- Fix #62 by escaping backticks.
- Replace "ls" with "test" in exists()
- Fixes #50. Thanks to Alex Koshelev for the patch.
- `local`'s return value now exhibits `.return_code`.
- Abort on bad role names instead of blowing up.
- Turn off DeprecationWarning when importing paramiko.
- Attempted fix re #32 (dropped output)
- Update role/host initialization logic (was missing some edge cases)
- Add note to install docs re: PyCrypto on win32.
- Add FAQ item re: changing env.shell.
- Rest of TODO migrated to tickets.
- `fab test` (when in source tree) now uses doctests.
- Add note to compatibility page re: fab_quiet.
- Update local() to honor context_managers.cd()

CHAPTER 5

Getting help

If you've scoured the *prose* and *API* documentation and still can't find an answer to your question, below are various support resources that should help. We do request that you do at least skim the documentation before posting tickets or mailing list questions, however!

5.1 Mailing list

The best way to get help with using Fabric is via the fab-user mailing list (currently hosted at `nongnu.org`.) The Fabric developers do their best to reply promptly, and the list contains an active community of other Fabric users and contributors as well.

5.2 Bugs/ticket tracker

To file new bugs or search existing ones, you may visit Fabric's Redmine instance, located at code.fabfile.org. Due to issues with spam, you'll need to (quickly and painlessly) register an account in order to post new tickets.

5.3 Wiki

There is an official Fabric MoinMoin wiki reachable at wiki.fabfile.org, although as of this writing its usage patterns are still being worked out. Like the ticket tracker, spam has forced us to put anti-spam measures up: the wiki has a simple, easy captcha in place on the edit form.

f