JupyterHub Documentation

Release 0.6.0

Project Jupyter team

Jul 23, 2016

User Documentation

JupyterHub is a server that gives multiple users access to Jupyter notebooks, running an independent Jupyter notebook server for each user.

To use JupyterHub, you need a Unix server (typically Linux) running somewhere that is accessible to your team on the network. The JupyterHub server can be on an internal network at your organisation, or it can run on the public internet (in which case, take care with security). Users access JupyterHub in a web browser, by going to the IP address or domain name of the server.

Different *authenticators* control access to JupyterHub. The default one (pam) uses the user accounts on the server where JupyterHub is running. If you use this, you will need to create a user account on the system for each user on your team. Using other authenticators, you can allow users to sign in with e.g. a Github account, or with any single-sign-on system your organisation has.

Next, *spawners* control how JupyterHub starts the individual notebook server for each user. The default spawner will start a notebook server on the same machine running under their system username. The other main option is to start each server in a separate container, often using Docker.

JupyterHub runs as three separate parts:

- The multi-user Hub (Python & Tornado)
- A configurable http proxy (NodeJS)
- Multiple single-user Jupyter notebook servers (Python & Tornado)

Basic principles:

- Hub spawns proxy
- Proxy forwards ~all requests to hub by default
- Hub handles login, and spawns single-user servers on demand
- Hub configures proxy to forward url prefixes to single-user servers

Contents:

CHAPTER 1

Getting started with JupyterHub

This document describes some of the basics of configuring JupyterHub to do what you want. JupyterHub is highly customizable, so there's a lot to cover.

1.1 Installation

See the readme for help installing JupyterHub.

1.2 Overview

JupyterHub is a set of processes that together provide a multiuser Jupyter Notebook server. There are three main categories of processes run by the `jupyterhub` command line program:

- **Single User Server**: a dedicated, single-user, Jupyter Notebook is started for each user on the system when they log in. The object that starts these processes is called a Spawner.
- **Proxy**: the public facing part of the server that uses a dynamic proxy to route HTTP requests to the Hub and Single User Servers.
- **Hub**: manages user accounts and authentication and coordinates Single Users Servers using a Spawner.

1.3 JupyterHub's default behavior

IMPORTANT: In its default configuration, JupyterHub requires SSL encryption (HTTPS) to run. **You should not run JupyterHub without SSL encryption on a public network.** See *Security documentation* for how to configure JupyterHub to use SSL, and in certain cases, e.g. behind SSL termination in nginx, allowing the hub to run with no SSL by requiring `--no-ssl` (as of version 0.5).

To start JupyterHub in its default configuration, type the following at the command line:

```
sudo jupyterhub
```

The default Authenticator that ships with JupyterHub authenticates users with their system name and password (via PAM). Any user on the system with a password will be allowed to start a single-user notebook server.

The default Spawner starts servers locally as each user, one dedicated server per user. These servers listen on localhost, and start in the given user's home directory.

By default, the **Proxy** listens on all public interfaces on port 8000. Thus you can reach JupyterHub through either:

```
http://localhost:8000
```

or any other public IP or domain pointing to your system.

In their default configuration, the other services, the **Hub** and **Single-User Servers**, all communicate with each other on localhost only.

By default, starting JupyterHub will write two files to disk in the current working directory:

- `jupyterhub.sqlite` is the sqlite database containing all of the state of the **Hub**. This file allows the **Hub** to remember what users are running and where, as well as other information enabling you to restart parts of JupyterHub separately.
- `jupyterhub_cookie_secret` is the encryption key used for securing cookies. This file needs to persist in order for restarting the Hub server to avoid invalidating cookies. Conversely, deleting this file and restarting the server effectively invalidates all login cookies. The cookie secret file is discussed in the [Cookie Secret documentation](#Cookie secret).

The location of these files can be specified via configuration, discussed below.

1.4 How to configure JupyterHub

JupyterHub is configured in two ways:

1. Configuration file
2. Command-line arguments

1.4.1 Configuration file

By default, JupyterHub will look for a configuration file (which may not be created yet) named `jupyterhub_config.py` in the current working directory. You can create an empty configuration file with:

```
jupyterhub --generate-config
```

This empty configuration file has descriptions of all configuration variables and their default values. You can load a specific config file with:

```
jupyterhub -f /path/to/jupyterhub_config.py
```

See also: general docs on the config system Jupyter uses.

1.4.2 Command-line arguments

Type the following for brief information about the command-line arguments:

```
jupyterhub -h
```

or:

```
jupyterhub --help-all
```

for the full command line help.

All configurable options are technically configurable on the command-line, even if some are really inconvenient to type. Just replace the desired option, c.Class.trait, with –Class.trait. For example, to configure c.Spawner.notebook_dir = '~/assignments' from the command-line:

```
jupyterhub --Spawner.notebook_dir='~/assignments'
```

1.5 Networking

1.5.1 Configuring the Proxy's IP address and port

The Proxy's main IP address setting determines where JupyterHub is available to users. By default, JupyterHub is configured to be available on all network interfaces (`''`) on port 8000. **Note**: Use of `'*'` is discouraged for IP configuration; instead, use of `'0.0.0.0'` is preferred.

Changing the IP address and port can be done with the following command line arguments:

```
jupyterhub --ip=192.168.1.2 --port=443
```

Or by placing the following lines in a configuration file:

```
c.JupyterHub.ip = '192.168.1.2'
c.JupyterHub.port = 443
```

Port 443 is used as an example since 443 is the default port for SSL/HTTPS.

Configuring only the main IP and port of JupyterHub should be sufficient for most deployments of JupyterHub. However, more customized scenarios may need additional networking details to be configured.

1.5.2 Configuring the Proxy's REST API communication IP address and port (optional)

The Hub service talks to the proxy via a REST API on a secondary port, whose network interface and port can be configured separately. By default, this REST API listens on port 8081 of localhost only.

If running the Proxy separate from the Hub, configure the REST API communication IP address and port with:

```
# ideally a private network address
c.JupyterHub.proxy_api_ip = '10.0.1.4'
c.JupyterHub.proxy_api_port = 5432
```

1.5.3 Configuring the Hub if Spawners or Proxy are remote or isolated in containers

The Hub service also listens only on localhost (port 8080) by default. The Hub needs needs to be accessible from both the proxy and all Spawners. When spawning local servers, an IP address setting of localhost is fine. If *either* the Proxy *or* (more likely) the Spawners will be remote or isolated in containers, the Hub must listen on an IP that is accessible.

```
c.JupyterHub.hub_ip = '10.0.1.4'
c.JupyterHub.hub_port = 54321
```

1.6 Security

IMPORTANT: In its default configuration, JupyterHub requires SSL encryption (HTTPS) to run. **You should not run JupyterHub without SSL encryption on a public network.**

Security is the most important aspect of configuring Jupyter. There are three main aspects of the security configuration:

1. SSL encryption (to enable HTTPS)
2. Cookie secret (a key for encrypting browser cookies)
3. Proxy authentication token (used for the Hub and other services to authenticate to the Proxy)

1.7 SSL encryption

Since JupyterHub includes authentication and allows arbitrary code execution, you should not run it without SSL (HTTPS). This will require you to obtain an official, trusted SSL certificate or create a self-signed certificate. Once you have obtained and installed a key and certificate you need to specify their locations in the configuration file as follows:

```
c.JupyterHub.ssl_key = '/path/to/my.key'
c.JupyterHub.ssl_cert = '/path/to/my.cert'
```

It is also possible to use letsencrypt (https://letsencrypt.org/) to obtain a free, trusted SSL certificate. If you run letsencrypt using the default options, the needed configuration is (replace `your.domain.com` by your fully qualified domain name):

```
c.JupyterHub.ssl_key = '/etc/letsencrypt/live/your.domain.com/privkey.pem'
c.JupyterHub.ssl_cert = '/etc/letsencrypt/live/your.domain.com/fullchain.pem'
```

Some cert files also contain the key, in which case only the cert is needed. It is important that these files be put in a secure location on your server, where they are not readable by regular users.

Note: In certain cases, e.g. behind SSL termination in nginx, allowing no SSL running on the hub may be desired. To run the Hub without SSL, you must opt in by configuring and confirming the `--no-ssl` option, added as of version 0.5.

1.8 Cookie secret

The cookie secret is an encryption key, used to encrypt the browser cookies used for authentication. If this value changes for the Hub, all single-user servers must also be restarted. Normally, this value is stored in a file, the location of which can be specified in a config file as follows:

```
c.JupyterHub.cookie_secret_file = '/srv/jupyterhub/cookie_secret'
```

The content of this file should be a long random string encoded in MIME Base64. An example would be to generate this file as:

```
openssl rand -base64 2048 > /srv/jupyterhub/cookie_secret
```

In most deployments of JupyterHub, you should point this to a secure location on the file system, such as `/srv/jupyterhub/cookie_secret`. If the cookie secret file doesn't exist when the Hub starts, a new cookie secret is generated and stored in the file. The file must not be readable by group or other or the server won't start. The recommended permissions for the cookie secret file are 600 (owner-only rw).

If you would like to avoid the need for files, the value can be loaded in the Hub process from the `JPY_COOKIE_SECRET` environment variable, which is a hex-encoded string. You can set it this way:

```
export JPY_COOKIE_SECRET=`openssl rand -hex 1024`
```

For security reasons, this environment variable should only be visible to the Hub. If you set it dynamically as above, all users will be logged out each time the Hub starts.

You can also set the secret in the configuration file itself as a binary string:

```
c.JupyterHub.cookie_secret = bytes.fromhex('VERY LONG SECRET HEX STRING')
```

1.9 Proxy authentication token

The Hub authenticates its requests to the Proxy using a secret token that the Hub and Proxy agree upon. The value of this string should be a random string (for example, generated by `openssl rand -hex 32`). You can pass this value to the Hub and Proxy using either the `CONFIGPROXY_AUTH_TOKEN` environment variable:

```
export CONFIGPROXY_AUTH_TOKEN=`openssl rand -hex 32`
```

This environment variable needs to be visible to the Hub and Proxy.

Or you can set the value in the configuration file:

```
c.JupyterHub.proxy_auth_token =
↪'0bc02bede919e99a26de1e2a7a5aadfaf6228de836ec39a05a6c6942831d8fe5'
```

If you don't set the Proxy authentication token, the Hub will generate a random key itself, which means that any time you restart the Hub you **must also restart the Proxy**. If the proxy is a subprocess of the Hub, this should happen automatically (this is the default configuration).

Another time you must set the Proxy authentication token yourself is if you want other services, such as nbgrader to also be able to connect to the Proxy.

1.10 Configuring authentication

The default Authenticator uses PAM to authenticate system users with their username and password. The default behavior of this Authenticator is to allow any user with an account and password on the system to login. You can restrict which users are allowed to login with `Authenticator.whitelist`:

```
c.Authenticator.whitelist = {'mal', 'zoe', 'inara', 'kaylee'}
```

Admin users of JupyterHub have the ability to take actions on users' behalf, such as stopping and restarting their servers, and adding and removing new users from the whitelist. Any users in the admin list are automatically added to the whitelist, if they are not already present. The set of initial Admin users can configured as follows:

```
c.Authenticator.admin_users = {'mal', 'zoe'}
```

If `JupyterHub.admin_access` is True (not default), then admin users have permission to log in *as other users* on their respective machines, for debugging. **You should make sure your users know if admin_access is enabled.**

1.10.1 Adding and removing users

Users can be added and removed to the Hub via the admin panel or REST API. These users will be added to the whitelist and database. Restarting the Hub will not require manually updating the whitelist in your config file, as the users will be loaded from the database. This means that after starting the Hub once, it is not sufficient to remove users from the whitelist in your config file. You must also remove them from the database, either by discarding the database file, or via the admin UI.

The default `PAMAuthenticator` is one case of a special kind of authenticator, called a `LocalAuthenticator`, indicating that it manages users on the local system. When you add a user to the Hub, a `LocalAuthenticator` checks if that user already exists. Normally, there will be an error telling you that the user doesn't exist. If you set the configuration value

```
c.LocalAuthenticator.create_system_users = True
```

however, adding a user to the Hub that doesn't already exist on the system will result in the Hub creating that user via the system `adduser` command line tool. This option is typically used on hosted deployments of JupyterHub, to avoid the need to manually create all your users before launching the service. It is not recommended when running JupyterHub in situations where JupyterHub users maps directly onto UNIX users.

1.11 Configuring single-user servers

Since the single-user server is an instance of `jupyter notebook`, an entire separate multi-process application, there are many aspect of that server can configure, and a lot of ways to express that configuration.

At the JupyterHub level, you can set some values on the Spawner. The simplest of these is `Spawner.notebook_dir`, which lets you set the root directory for a user's server. This root notebook directory is the highest level directory users will be able to access in the notebook dashboard. In this example, the root notebook directory is set to `~/notebooks`, where `~` is expanded to the user's home directory.

```
c.Spawner.notebook_dir = '~/notebooks'
```

You can also specify extra command-line arguments to the notebook server with:

```
c.Spawner.args = ['--debug', '--profile=PHYS131']
```

This could be used to set the users default page for the single user server:

```
c.Spawner.args = ['--NotebookApp.default_url=/notebooks/Welcome.ipynb']
```

Since the single-user server extends the notebook server application, it still loads configuration from the `ipython_notebook_config.py` config file. Each user may have one of these files in `$HOME/.ipython/profile_default/`. IPython also supports loading system-wide config files from `/etc/ipython/`, which is the place to put configuration that you want to affect all of your users.

1.12 External services

JupyterHub has a REST API that can be used to run external services. More detail on this API will be added in the future.

1.13 File locations

It is recommended to put all of the files used by JupyterHub into standard UNIX filesystem locations.

- `/srv/jupyterhub` for all security and runtime files
- `/etc/jupyterhub` for all configuration files
- `/var/log` for log files

1.14 Example

In the following example, we show a configuration files for a fairly standard JupyterHub deployment with the following assumptions:

- JupyterHub is running on a single cloud server
- Using SSL on the standard HTTPS port 443
- You want to use GitHub OAuth for login
- You need the users to exist locally on the server
- You want users' notebooks to be served from `~/assignments` to allow users to browse for notebooks within other users home directories
- You want the landing page for each user to be a Welcome.ipynb notebook in their assignments directory.
- All runtime files are put into `/srv/jupyterhub` and log files in `/var/log`.

Let's start out with `jupyterhub_config.py`:

```
# jupyterhub_config.py
c = get_config()

import os
pjoin = os.path.join

runtime_dir = os.path.join('/srv/jupyterhub')
ssl_dir = pjoin(runtime_dir, 'ssl')
if not os.path.exists(ssl_dir):
    os.makedirs(ssl_dir)


# https on :443
c.JupyterHub.port = 443
c.JupyterHub.ssl_key = pjoin(ssl_dir, 'ssl.key')
c.JupyterHub.ssl_cert = pjoin(ssl_dir, 'ssl.cert')

# put the JupyterHub cookie secret and state db
# in /var/run/jupyterhub
c.JupyterHub.cookie_secret_file = pjoin(runtime_dir, 'cookie_secret')
c.JupyterHub.db_url = pjoin(runtime_dir, 'jupyterhub.sqlite')
# or `--db=/path/to/jupyterhub.sqlite` on the command-line

# put the log file in /var/log
c.JupyterHub.log_file = '/var/log/jupyterhub.log'

# use GitHub OAuthenticator for local users
```

```
c.JupyterHub.authenticator_class = 'oauthenticator.LocalGitHubOAuthenticator'
c.GitHubOAuthenticator.oauth_callback_url = os.environ['OAUTH_CALLBACK_URL']
# create system users that don't exist yet
c.LocalAuthenticator.create_system_users = True

# specify users and admin
c.Authenticator.whitelist = {'rgbkrk', 'minrk', 'jhamrick'}
c.Authenticator.admin_users = {'jhamrick', 'rgbkrk'}

# start single-user notebook servers in ~/assignments,
# with ~/assignments/Welcome.ipynb as the default landing page
# this config could also be put in
# /etc/ipython/ipython_notebook_config.py
c.Spawner.notebook_dir = '~/assignments'
c.Spawner.args = ['--NotebookApp.default_url=/notebooks/Welcome.ipynb']
```

Using the GitHub Authenticator requires a few additional env variables, which we will need to set when we launch the server:

```
export GITHUB_CLIENT_ID=github_id
export GITHUB_CLIENT_SECRET=github_secret
export OAUTH_CALLBACK_URL=https://example.com/hub/oauth_callback
export CONFIGPROXY_AUTH_TOKEN=super-secret
jupyterhub -f /path/to/aboveconfig.py
```

CHAPTER 2

Further reading

- Custom Authenticators
- Custom Spawners
- Troubleshooting

CHAPTER 3

How JupyterHub works

JupyterHub is a multi-user server that manages and proxies multiple instances of the single-user IPython Jupyter notebook server.

There are three basic processes involved:

- multi-user Hub (Python/Tornado)
- configurable http proxy (node-http-proxy)
- multiple single-user IPython notebook servers (Python/IPython/Tornado)

The proxy is the only process that listens on a public interface. The Hub sits behind the proxy at `/hub`. Single-user servers sit behind the proxy at `/user/[username]`.

3.1 Logging in

When a new browser logs in to JupyterHub, the following events take place:

- Login data is handed to the *Authenticator* instance for validation
- The Authenticator returns the username, if login information is valid
- A single-user server instance is *Spawned* for the logged-in user
- When the server starts, the proxy is notified to forward `/user/[username]/*` to the single-user server
- Two cookies are set, one for `/hub/` and another for `/user/[username]`, containing an encrypted token.
- The browser is redirected to `/user/[username]`, which is handled by the single-user server

Logging into a single-user server is authenticated via the Hub:

- On request, the single-user server forwards the encrypted cookie to the Hub for verification
- The Hub replies with the username if it is a valid cookie
- If the user is the owner of the server, access is allowed
- If it is the wrong user or an invalid cookie, the browser is redirected to `/hub/login`

3.2 Customizing JupyterHub

There are two basic extension points for JupyterHub: How users are authenticated, and how their server processes are started. Each is governed by a customizable class, and JupyterHub ships with just the most basic version of each.

To enable custom authentication and/or spawning, subclass Authenticator or Spawner, and override the relevant methods.

3.2.1 Authentication

Authentication is customizable via the Authenticator class. Authentication can be replaced by any mechanism, such as OAuth, Kerberos, etc.

JupyterHub only ships with PAM authentication, which requires the server to be run as root, or at least with access to the PAM service, which regular users typically do not have (on Ubuntu, this requires being added to the `shadow` group).

More info on custom Authenticators.

See a list of custom Authenticators on the wiki.

3.2.2 Spawning

Each single-user server is started by a Spawner. The Spawner represents an abstract interface to a process, and needs to be able to take three actions:

1. start the process
2. poll whether the process is still running
3. stop the process

More info on custom Spawners.

See a list of custom Spawners on the wiki.

CHAPTER 4

Web Security in JupyterHub

JupyterHub is designed to be a simple multi-user server for modestly sized groups of semi-trusted users. While the design reflects serving semi-trusted users, JupyterHub is not necessarily unsuitable for serving untrusted users. Using JupyterHub with untrusted users does mean more work and much care is required to secure a Hub against untrusted users, with extra caution on protecting users from each other as the Hub is serving untrusted users.

One aspect of JupyterHub's design simplicity for semi-trusted users is that the Hub and single-user servers are placed in a single domain, behind a proxy. As a result, if the Hub is serving untrusted users, many of the web's cross-site protections are not applied between single-user servers and the Hub, or between single-user servers and each other, since browsers see the whole thing (proxy, Hub, and single user servers) as a single website.

To protect users from each other, a user must never be able to write arbitrary HTML and serve it to another user on the Hub's domain. JupyterHub's authentication setup prevents this because only the owner of a given single-user server is allowed to view user-authored pages served by their server. To protect all users from each other, JupyterHub administrators must ensure that:

- A user does not have permission to modify their single-user server:
  - A user may not install new packages in the Python environment that runs their server.
  - If the PATH is used to resolve the single-user executable (instead of an absolute path), a user may not create new files in any PATH directory that precedes the directory containing jupyterhub-singleuser.
  - A user may not modify environment variables (e.g. PATH, PYTHONPATH) for their single-user server.
- A user may not modify the configuration of the notebook server (the ~/.jupyter or JUPYTER_CONFIG_DIR directory).

If any additional services are run on the same domain as the Hub, the services must never display user-authored HTML that is neither sanitized nor sandboxed (e.g. IFramed) to any user that lacks authentication as the author of a file.

4.1 Mitigations

There are two main configuration options provided by JupyterHub to mitigate these issues:

4.1.1 Subdomains

JupyterHub 0.5 adds the ability to run single-user servers on their own subdomains, which means the cross-origin protections between servers has the desired effect, and user servers and the Hub are protected from each other. A user's server will be at `username.jupyter.mydomain.com`, etc. This requires all user subdomains to point to the same address, which is most easily accomplished with wildcard DNS. Since this spreads the service across multiple domains, you will need wildcard SSL, as well. Unfortunately, for many institutional domains, wildcard DNS and SSL are not available, but if you do plan to serve untrusted users, enabling subdomains is highly encouraged, as it resolves all of the cross-site issues.

4.1.2 Disabling user config

If subdomains are not available or not desirable, 0.5 also adds an option `Spawner.disable_use_config`, which you can set to prevent the user-owned configuration files from being loaded. This leaves only package installation and PATHs as things the admin must enforce.

For most Spawners, PATH is not something users an influence, but care should be taken to ensure that the Spawn does *not* evaluate shell configuration files prior to launching the server.

Package isolation is most easily handled by running the single-user server in a virtualenv with disabled system-site-packages.

4.2 Extra notes

It is important to note that the control over the environment only affects the single-user server, and not the environment(s) in which the user's kernel(s) may run. Installing additional packages in the kernel environment does not pose additional risk to the web application's security.

CHAPTER 5

Writing a custom Authenticator

The Authenticator is the mechanism for authorizing users. Basic authenticators use simple username and password authentication. JupyterHub ships only with a PAM-based Authenticator, for logging in with local user accounts.

You can use custom Authenticator subclasses to enable authentication via other systems. One such example is using GitHub OAuth.

Because the username is passed from the Authenticator to the Spawner, a custom Authenticator and Spawner are often used together.

See a list of custom Authenticators on the wiki.

5.1 Basics of Authenticators

A basic Authenticator has one central method:

5.1.1 Authenticator.authenticate

```
Authenticator.authenticate(handler, data)
```

This method is passed the tornado RequestHandler and the POST data from the login form. Unless the login form has been customized, `data` will have two keys:

- `username` (self-explanatory)
- `password` (also self-explanatory)

`authenticate`'s job is simple:

- return a username (non-empty str) of the authenticated user if authentication is successful
- return `None` otherwise

Writing an Authenticator that looks up passwords in a dictionary requires only overriding this one method:

```
from tornado import gen
from IPython.utils.traitlets import Dict
from jupyterhub.auth import Authenticator

class DictionaryAuthenticator(Authenticator):

    passwords = Dict(config=True,
        help="""dict of username:password for authentication"""
    )

    @gen.coroutine
    def authenticate(self, handler, data):
        if self.passwords.get(data['username']) == data['password']:
            return data['username']
```

5.1.2 Authenticator.whitelist

Authenticators can specify a whitelist of usernames to allow authentication. For local user authentication (e.g. PAM), this lets you limit which users can login.

5.2 Normalizing and validating usernames

Since the Authenticator and Spawner both use the same username, sometimes you want to transform the name coming from the authentication service (e.g. turning email addresses into local system usernames) before adding them to the Hub service. Authenticators can define `normalize_username`, which takes a username. The default normalization is to cast names to lowercase

For simple mappings, a configurable dict `Authenticator.username_map` is used to turn one name into another:

```
c.Authenticator.username_map  = {
  'service-name': 'localname'
}
```

5.2.1 Validating usernames

In most cases, there is a very limited set of acceptable usernames. Authenticators can define `validate_username(username)`, which should return True for a valid username and False for an invalid one. The primary effect this has is improving error messages during user creation.

The default behavior is to use configurable `Authenticator.username_pattern`, which is a regular expression string for validation.

To only allow usernames that start with 'w':

```
c.Authenticator.username_pattern = r'w.*'
```

5.3 OAuth and other non-password logins

Some login mechanisms, such as OAuth, don't map onto username+password. For these, you can override the login handlers.

You can see an example implementation of an Authenticator that uses GitHub OAuth at OAuthenticator.

CHAPTER 6

Writing a custom Spawner

A Spawner starts each single-user notebook server. The Spawner represents an abstract interface to a process, and a custom Spawner needs to be able to take three actions:

- start the process
- poll whether the process is still running
- stop the process

6.1 Examples

Custom Spawners for JupyterHub can be found on the JupyterHub wiki. Some examples include:

- DockerSpawner for spawning user servers in Docker containers
  - dockerspawner.DockerSpawner for spawning identical Docker containers for each users
  - dockerspawner.SystemUserSpawner for spawning Docker containers with an environment and home directory for each users
- SudoSpawner enables JupyterHub to run without being root, by spawning an intermediate process via `sudo`
- BatchSpawner for spawning remote servers using batch systems
- RemoteSpawner to spawn notebooks and a remote server and tunnel the port via SSH
- SwarmSpawner for spawning containers using Docker Swarm

6.2 Spawner control methods

6.2.1 Spawner.start

`Spawner.start` should start the single-user server for a single user. Information about the user can be retrieved from `self.user`, an object encapsulating the user's name, authentication, and server info.

When `Spawner.start` returns, it should have stored the IP and port of the single-user server in `self.user.server`.

NOTE: When writing coroutines, *never* `yield` in between a database change and a commit.

Most `Spawner.start` functions will look similar to this example:

```
def start(self):
    self.user.server.ip = 'localhost' # or other host or IP address, as seen by the␣
↪Hub
    self.user.server.port = 1234 # port selected somehow
    self.db.commit() # always commit before yield, if modifying db values
    yield self._actually_start_server_somehow()
```

When `Spawner.start` returns, the single-user server process should actually be running, not just requested. JupyterHub can handle `Spawner.start` being very slow (such as PBS-style batch queues, or instantiating whole AWS instances) via relaxing the `Spawner.start_timeout` config value.

6.2.2 Spawner.poll

`Spawner.poll` should check if the spawner is still running. It should return `None` if it is still running, and an integer exit status, otherwise.

For the local process case, `Spawner.poll` uses `os.kill(PID,0)` to check if the local process is still running.

6.2.3 Spawner.stop

`Spawner.stop` should stop the process. It must be a tornado coroutine, which should return when the process has finished exiting.

6.3 Spawner state

JupyterHub should be able to stop and restart without tearing down single-user notebook servers. To do this task, a Spawner may need to persist some information that can be restored later. A JSON-able dictionary of state can be used to store persisted information.

Unlike start, stop, and poll methods, the state methods must not be coroutines.

For the single-process case, the Spawner state is only the process ID of the server:

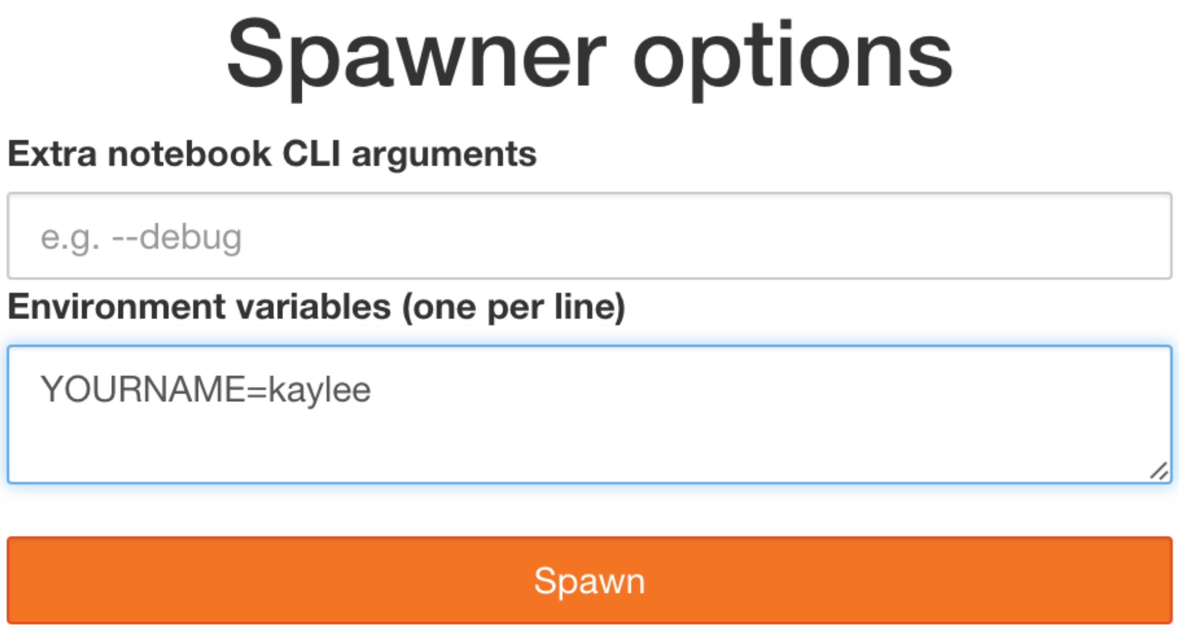
```
def get_state(self):
    """get the current state"""
    state = super().get_state()
    if self.pid:
        state['pid'] = self.pid
    return state

def load_state(self, state):
    """load state from the database"""
    super().load_state(state)
    if 'pid' in state:
        self.pid = state['pid']

def clear_state(self):
    """clear any state (called after shutdown)"""
    super().clear_state()
    self.pid = 0
```

6.4 Spawner options form

(new in 0.4)

Some deployments may want to offer options to users to influence how their servers are started. This may include cluster-based deployments, where users specify what resources should be available, or docker-based deployments where users can select from a list of base images.

This feature is enabled by setting `Spawner.options_form`, which is an HTML form snippet inserted unmodified into the spawn form. If the `Spawner.options_form` is defined, when a user tries to start their server, they will be directed to a form page, like this:


Spawner options
Extra notebook CLI arguments
e.g. --debug
Environment variables (one per line)
YOURNAME=kaylee
Spawn

If `Spawner.options_form` is undefined, the user's server is spawned directly, and no spawn page is rendered.

See this example for a form that allows custom CLI args for the local spawner.

6.4.1 `Spawner.options_from_form`

Options from this form will always be a dictionary of lists of strings, e.g.:

```
{
  'integer': ['5'],
  'text': ['some text'],
  'select': ['a', 'b'],
}
```

When `formdata` arrives, it is passed through `Spawner.options_from_form(formdata)`, which is a method to turn the form data into the correct structure. This method must return a dictionary, and is meant to interpret the lists-of-strings into the correct types. For example, the `options_from_form` for the above form would look like:

```
def options_from_form(self, formdata):
    options = {}
    options['integer'] = int(formdata['integer'][0]) # single integer value
    options['text'] = formdata['text'][0] # single string value
    options['select'] = formdata['select'] # list already correct
    options['notinform'] = 'extra info' # not in the form at all
    return options
```

which would return:

```
{
  'integer': 5,
  'text': 'some text',
  'select': ['a', 'b'],
  'notinform': 'extra info',
}
```

When `Spawner.spawn` is called, this dictionary is accessible as `self.user_options`.

CHAPTER 7

Troubleshooting

This document is under active development.

When troubleshooting, you may see unexpected behaviors or receive an error message. These two lists provide links to identifying the cause of the problem and how to resolve it.

7.1 Behavior problems

- *JupyterHub proxy fails to start*

7.2 Errors

- *500 error after spawning a single-user server*

7.3 JupyterHub proxy fails to start

If you have tried to start the JupyterHub proxy and it fails to start:

- check if the JupyterHub IP configuration setting is `c.JupyterHub.ip = '*'`; if it is, try `c.JupyterHub.ip = ''`
- Try starting with `jupyterhub --ip=0.0.0.0`

7.4 500 error after spawning my single-user server

You receive a 500 error when accessing the URL `/user/you/....` This is often seen when your single-user server cannot check your cookies with the Hub.

There are two likely reasons for this:

1. The single-user server cannot connect to the Hub's API (networking configuration problems)
2. The single-user server cannot *authenticate* its requests (invalid token)

7.4.1 Symptoms:

The main symptom is a failure to load *any* page served by the single-user server, met with a 500 error. This is typically the first page at `/user/you` after logging in or clicking "Start my server". When a single-user server receives a request, it makes an API request to the Hub to check if the cookie corresponds to the right user. This request is logged.

If everything is working, it will look like this:

```
200 GET /hub/api/authorizations/cookie/jupyter-hub-token-name/[secret] (@10.0.1.4) 6.
↪10ms
```

You should see a similar 200 message, as above, in the Hub log when you first visit your single-user server. If you don't see this message in the log, it may mean that your single-user server isn't connecting to your Hub.

If you see 403 (forbidden) like this, it's a token problem:

```
403 GET /hub/api/authorizations/cookie/jupyter-hub-token-name/[secret] (@10.0.1.4) 4.
↪14ms
```

Check the logs of the single-user server, which may have more detailed information on the cause.

7.4.2 Causes and resolutions:

No authorization request

If you make an API request and it is not received by the server, you likely have a network configuration issue. Often, this happens when the Hub is only listening on 127.0.0.1 (default) and the single-user servers are not on the same 'machine' (can be physically remote, or in a docker container or VM). The fix for this case is to make sure that `c.JupyterHub.hub_ip` is an address that all single-user servers can connect to, e.g.:

```
c.JupyterHub.hub_ip = '10.0.0.1'
```

403 GET /hub/api/authorizations/cookie

If you receive a 403 error, the API token for the single-user server is likely invalid. Commonly, the 403 error is caused by resetting the JupyterHub database (either removing jupyterhub.sqlite or some other action) while leaving single-user servers running. This happens most frequently when using DockerSpawner, because Docker's default behavior is to stop/start containers which resets the JupyterHub database, rather than destroying and recreating the container every time. This means that the same API token is used by the server for its whole life, until the container is rebuilt.

The fix for this Docker case is to remove any Docker containers seeing this issue (typicaly all containers created before a certain point in time):

```
docker rm -f jupyter-name
```

After this, when you start your server via JupyterHub, it will build a new container. If this was the underlying cause of the issue, you should see your server again.

CHAPTER 8

The JupyterHub API

Release 0.6.0

Date Jul 23, 2016

8.1 Authenticators

8.1.1 Module: `jupyterhub.auth`

Base Authenticator class and the default PAM Authenticator

class `jupyterhub.auth.`**`Authenticator`**(***kwargs*)

A class for authentication.

The primary API is one method, *authenticate*, a tornado coroutine for authenticating users.

`add_user`(*user*)

Add a new user

By default, this just adds the user to the whitelist.

Subclasses may do more extensive things, such as adding actual unix users, but they should call super to ensure the whitelist is updated.

Parameters **`user`** (`User`) – The User wrapper object

`authenticate`(*handler*, *data*)

Authenticate a user with login form data.

This must be a tornado gen.coroutine. It must return the username on successful authentication, and return None on failed authentication.

Checking the whitelist is handled separately by the caller.

Parameters

- **`handler`** (*`tornado.web.RequestHandler`*) – the current request handler
- **`data`** (*`dict`*) – The formdata of the login form. The default form has 'username' and 'password' fields.

Returns the username of the authenticated user None: Authentication failed

Return type str

check_whitelist(*username*)

Check a username against our whitelist.

Return True if username is allowed, False otherwise. No whitelist means any username should be allowed.

Names are normalized *before* being checked against the whitelist.

delete_user(*user*)

Triggered when a user is deleted.

Removes the user from the whitelist. Subclasses should call super to ensure the whitelist is updated.

Parameters **user** (`User`) – The User wrapper object

get_authenticated_user(*handler*, *data*)

This is the outer API for authenticating a user.

This calls *authenticate*, which should be overridden in subclasses, normalizes the username if any normalization should be done, and then validates the name in the whitelist.

Subclasses should not need to override this method. The various stages can be overridden separately:

- authenticate turns formdata into a username
- normalize_username normalizes the username
- check_whitelist checks against the user whitelist

get_handlers(*app*)

Return any custom handlers the authenticator needs to register

(e.g. for OAuth).

Parameters **app** (*JupyterHub Application*) – the application object, in case it needs to be accessed for info.

Returns

list of `('/url',Handler)` tuples passed to tornado. The Hub prefix is added to any URLs.

Return type list

login_url(*base_url*)

Override to register a custom login handler

Generally used in combination with get_handlers.

Parameters **base_url** (*str*) – the base URL of the Hub (e.g. /hub/)

Returns The login URL, e.g. '/hub/login'

Return type str

logout_url(*base_url*)

Override to register a custom logout handler.

Generally used in combination with get_handlers.

Parameters **base_url** (*str*) – the base URL of the Hub (e.g. /hub/)

Returns The logout URL, e.g. '/hub/logout'

Return type str

normalize_username(*username*)

Normalize a username.

Override in subclasses if usernames should have some normalization. Default: cast to lowercase, lookup in username_map.

`post_spawn_stop`(*user*, *spawner*)
Hook called after stopping a user container.

Can be used to do auth-related cleanup, e.g. closing PAM sessions.

`pre_spawn_start`(*user*, *spawner*)
Hook called before spawning a user's server.

Can be used to do auth-related startup, e.g. opening PAM sessions.

`validate_username`(*username*)
Validate a (normalized) username.

Return True if username is valid, False otherwise.

class `jupyterhub.auth.`**`LocalAuthenticator`**(***kwargs*)
Base class for Authenticators that work with local Linux/UNIX users

Checks for local users, and can attempt to create them if they exist.

`add_system_user`(*user*)
Create a new Linux/UNIX user on the system. Works on FreeBSD and Linux, at least.

`add_user`(*user*)
Add a new user

If self.create_system_users, the user will attempt to be created.

static **`system_user_exists`**(*user*)
Check if the user exists on the system

class `jupyterhub.auth.`**`PAMAuthenticator`**(***kwargs*)
Authenticate local Linux/UNIX users with PAM

8.2 Spawners

8.2.1 Module: `jupyterhub.spawner`

Class for spawning single-user notebook servers.

`Spawner`

class `jupyterhub.spawner.`**`Spawner`**(***kwargs*)
Base class for spawning single-user notebook servers.

Subclass this, and override the following methods:

- load_state
- get_state
- start
- stop
- poll

get_args()
Return the arguments to be passed after self.cmd

get_env()
Return the environment dict to use for the Spawner.

This applies things like *env_keep*, anything defined in *Spawner.environment*, and adds the API token to the env.

Use this to access the env in Spawner.start to allow extension in subclasses.

get_state()
store the state necessary for load_state

A black box of extra state for custom spawners. Subclasses should call *super*.

Returns **state** – a JSONable dict of state

Return type dict

options_from_form(*form_data*)
Interpret HTTP form data

Form data will always arrive as a dict of lists of strings. Override this function to understand single-values, numbers, etc.

This should coerce form data into the structure expected by self.user_options, which must be a dict.

Instances will receive this data on self.user_options, after passing through this function, prior to *Spawner.start*.

poll()
Check if the single-user process is running

return None if it is, an exit status (0 if unknown) if it is not.

start()
Start the single-user process

stop(*now=False*)
Stop the single-user process

class jupyterhub.spawner.**LocalProcessSpawner**(***kwargs*)
A Spawner that just uses Popen to start local processes as users.

Requires users to exist on the local system.

This is the default spawner for JupyterHub.

8.3 Users

8.3.1 Module: `jupyterhub.user`

`User`

class jupyterhub.user.**Server**

class jupyterhub.user.**User**(*orm_user*, *settings*, ***kwargs*)

name
The user's name

`server`
The user's Server data object if running, None otherwise. Has `ip`, `port` attributes.

`spawner`
The user's *`Spawner`* instance.

`escaped_name`
My name, escaped for use in URLs, cookies, etc.

CHAPTER 9

Summary of changes in JupyterHub

See `git log` for a more detailed summary.

9.1 0.6

- JupyterHub has moved to a new `jupyterhub` namespace on GitHub and Docker. What was `juptyer/jupyterhub` is now `jupyterhub/jupyterhub`, etc.
- `jupyterhub/jupyterhub` image on DockerHub no longer loads the jupyterhub_config.py in an ONBUILD step. A new `jupyterhub/jupyterhub-onbuild` image does this
- Add statsd support, via `c.JupyterHub.statsd_{host,port,prefix}`
- Update to traitlets 4.1 `@default`, `@observe` APIs for traits
- Allow disabling PAM sessions via `c.PAMAuthenticator.open_sessions = False`. This may be needed on SELinux-enabled systems, where our PAM session logic often does not work properly
- Add `Spawner.environment` configurable, for defining extra environment variables to load for single-user servers
- JupyterHub API tokens can be pregenerated and loaded via `JupyterHub.api_tokens`, a dict of `token: username`.
- JupyterHub API tokens can be requested via the REST API, with a POST request to `/api/authorizations/token`. This can only be used if the Authenticator has a username and password.
- Various fixes for user URLs and redirects

9.2 0.5

- Single-user server must be run with Jupyter Notebook 4.0
- Require `--no-ssl` confirmation to allow the Hub to be run without SSL (e.g. behind SSL termination in nginx)
- Add lengths to text fields for MySQL support
- Add `Spawner.disable_user_config` for preventing user-owned configuration from modifying single-user servers.
- Fixes for MySQL support.

- Add ability to run each user's server on its own subdomain. Requires wildcard DNS and wildcard SSL to be feasible. Enable subdomains by setting `JupyterHub.subdomain_host = 'https://jupyterhub.domain.tld[:port]'`.
- Use `127.0.0.1` for local communication instead of `localhost`, avoiding issues with DNS on some systems.
- Fix race that could add users to proxy prematurely if spawning is slow.

9.3 0.4

9.3.1 0.4.1

Fix removal of `/login` page in 0.4.0, breaking some OAuth providers.

9.3.2 0.4.0

- Add `Spawner.user_options_form` for specifying an HTML form to present to users, allowing users to influence the spawning of their own servers.
- Add `Authenticator.pre_spawn_start` and `Authenticator.post_spawn_stop` hooks, so that Authenticators can do setup or teardown (e.g. passing credentials to Spawner, mounting data sources, etc.). These methods are typically used with custom Authenticator+Spawner pairs.
- 0.4 will be the last JupyterHub release where single-user servers running IPython 3 is supported instead of Notebook 4.0.

9.4 0.3

- No longer make the user starting the Hub an admin
- start PAM sessions on login
- hooks for Authenticators to fire before spawners start and after they stop, allowing deeper interaction between Spawner/Authenticator pairs.
- login redirect fixes

9.5 0.2

- Based on standalone traitlets instead of IPython.utils.traitlets
- multiple users in admin panel
- Fixes for usernames that require escaping

9.6 0.1

First preview release

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index