JupyterHub Documentation

Release 0.7.2

Project Jupyter team

Jul 20, 2017

User Guide

1 Contents **3**

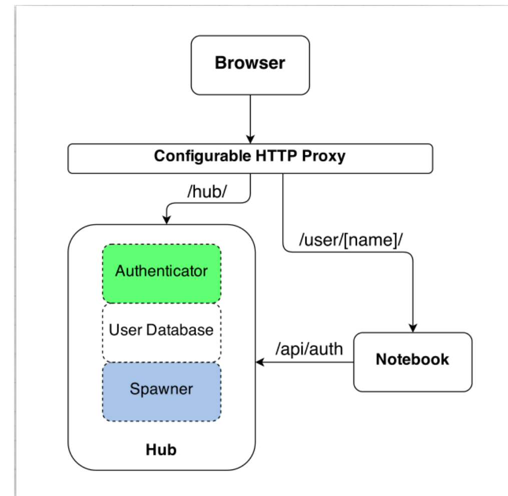
1.1 Quickstart - Installation . . . 3
1.2 Getting started with JupyterHub . . . 6
1.3 How JupyterHub works . . . 14
1.4 Web Security in JupyterHub . . . 16
1.5 Using JupyterHub's REST API . . . 17
1.6 Authenticators . . . 18
1.7 Spawners . . . 20
1.8 Services . . . 24
1.9 Configuration examples . . . 29
1.10 Upgrading JupyterHub and its database . . . 33
1.11 Troubleshooting . . . 35
1.12 The JupyterHub API . . . 39
1.13 Change log summary . . . 47
1.14 Contributors . . . 50

2 Indices and tables **53**

3 Questions? Suggestions? **55**

Python Module Index **57**

With JupyterHub you can create a **multi-user Hub** which spawns, manages, and proxies multiple instances of the single-user Jupyter notebook server. Due to its flexibility and customization options, JupyterHub can be used to serve notebooks to a class of students, a corporate data science group, or a scientific research group.



Three subsystems make up JupyterHub:

- a multi-user **Hub** (tornado process)
- a **configurable http proxy** (node-http-proxy)
- multiple **single-user Jupyter notebook servers** (Python/IPython/tornado)

JupyterHub's basic flow of operations includes:

- The Hub spawns a proxy
- The proxy forwards all requests to the Hub by default
- The Hub handles user login and spawns single-user servers on demand
- The Hub configures the proxy to forward URL prefixes to the single-user notebook servers

For convenient administration of the Hub, its users, and *Services* (added in version 7.0), JupyterHub also provides a REST API.

CHAPTER 1

Contents

User Guide

- *Quickstart - Installation*
- *Getting started with JupyterHub*
- *How JupyterHub works*
- *Web Security in JupyterHub*
- *Using JupyterHub's REST API*

1.1 Quickstart - Installation

1.1.1 Prerequisites

Before installing JupyterHub, you will need:

- Python 3.3 or greater

  An understanding of using `pip` or `conda` for installing Python packages is helpful.

- nodejs/npm

  Install nodejs/npm, using your operating system's package manager. For example, install on Linux (Debian/Ubuntu) using:

  ```
  sudo apt-get install npm nodejs-legacy
  ```

  (The `nodejs-legacy` package installs the `node` executable and is currently required for npm to work on Debian/Ubuntu.)

- TLS certificate and key for HTTPS communication
- Domain name

Before running the single-user notebook servers (which may be on the same system as the Hub or not):

- Jupyter Notebook version 4 or greater

1.1.2 Installation

JupyterHub can be installed with `pip` or `conda` and the proxy with `npm`:

pip, npm:

```
python3 -m pip install jupyterhub
npm install -g configurable-http-proxy
```

conda (one command installs jupyterhub and proxy):

```
conda install -c conda-forge jupyterhub
```

To test your installation:

```
jupyterhub -h
configurable-http-proxy -h
```

If you plan to run notebook servers locally, you will need also to install Jupyter notebook:

pip:

```
python3 -m pip install notebook
```

conda:

```
conda install notebook
```

1.1.3 Start the Hub server

To start the Hub server, run the command:

```
jupyterhub
```

Visit `https://localhost:8000` in your browser, and sign in with your unix credentials.

To allow multiple users to sign into the Hub server, you must start `jupyterhub` as a *privileged user*, such as root:

```
sudo jupyterhub
```

The wiki describes how to run the server as a *less privileged user*, which requires additional configuration of the system.

1.1.4 Basic Configuration

The getting started document contains detailed information abouts configuring a JupyterHub deployment.

The JupyterHub **tutorial** provides a video and documentation that explains and illustrates the fundamental steps for installation and configuration. Repo | Tutorial documentation

Generate a default configuration file

Generate a default config file:

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

Customize the configuration, authentication, and process spawning

Spawn the server on `10.0.1.2:443` with **https**:

```
jupyterhub --ip 10.0.1.2 --port 443 --ssl-key my_ssl.key --ssl-cert my_ssl.cert
```

The authentication and process spawning mechanisms can be replaced, which should allow plugging into a variety of authentication or process control environments. Some examples, meant as illustration and testing of this concept, are:

- Using GitHub OAuth instead of PAM with OAuthenticator
- Spawning single-user servers with Docker, using the DockerSpawner

---

1.1.5 Alternate Installation using Docker

A ready to go docker image for JupyterHub gives a straightforward deployment of JupyterHub.

Note: This `jupyterhub/jupyterhub` docker image is only an image for running the Hub service itself. It does not provide the other Jupyter components, such as Notebook installation, which are needed by the single-user servers. To run the single-user servers, which may be on the same system as the Hub or not, Jupyter Notebook version 4 or greater must be installed.

Starting JupyterHub with docker

The JupyterHub docker image can be started with the following command:

```
docker run -d --name jupyterhub jupyterhub/jupyterhub jupyterhub
```

This command will create a container named `jupyterhub` that you can **stop and resume** with `docker stop/start`.

The Hub service will be listening on all interfaces at port 8000, which makes this a good choice for **testing JupyterHub on your desktop or laptop**.

If you want to run docker on a computer that has a public IP then you should (as in MUST) **secure it with ssl** by adding ssl options to your docker configuration or using a ssl enabled proxy.

Mounting volumes will allow you to **store data outside the docker image (host system) so it will be persistent**, even when you start a new image.

The command `docker exec -it jupyterhub bash` will spawn a root shell in your docker container. You can **use the root shell to create system users in the container**. These accounts will be used for authentication in JupyterHub's default configuration.

1.2 Getting started with JupyterHub

This section contains getting started information on the following topics:

- Technical Overview
- Installation
- Configuration
- Networking
- Security
- Authentication and users
- Spawners and single-user notebook servers
- External Services

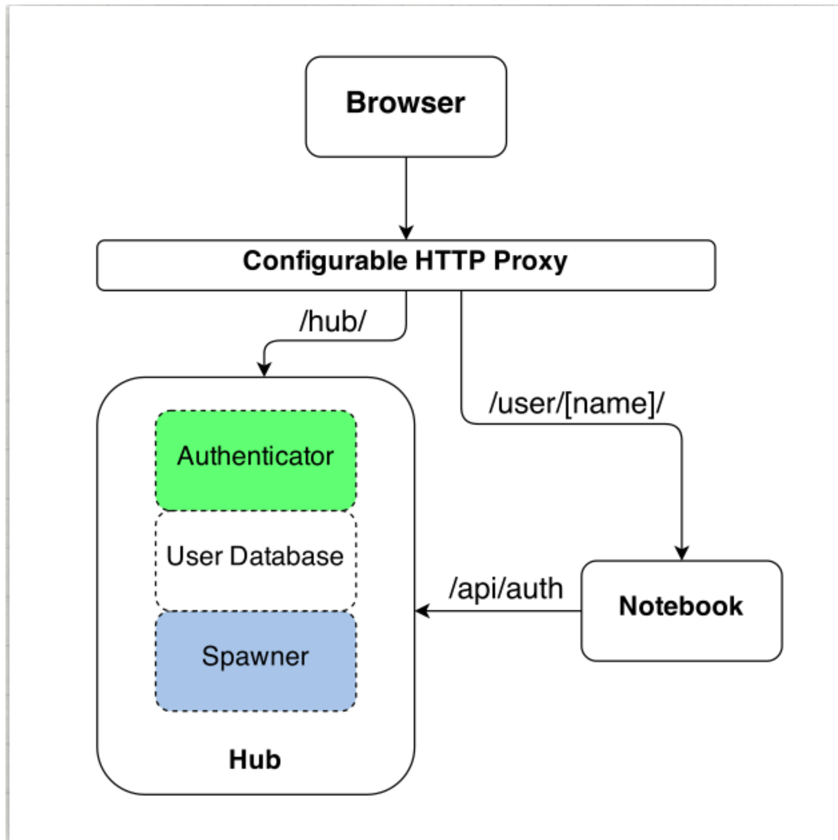
1.2.1 Technical Overview

JupyterHub is a set of processes that together provide a single user Jupyter Notebook server for each person in a group.

Three subsystems

Three major subsystems run by the `jupyterhub` command line program:

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



Deployment server

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 organization, or it can run on the public internet (in which case, take care with the Hub's security).

Basic operation

Users access JupyterHub through a web browser, by going to the IP address or the domain name of the server.

Basic principles of operation:

- 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

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 organization 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.

Default behavior

IMPORTANT: You should not run JupyterHub without SSL encryption on a public network.

See *Security documentation* for how to configure JupyterHub to use SSL, or put it behind SSL termination in another proxy server, such as nginx.

Deprecation note: Removed `--no-ssl` in version 0.7.

JupyterHub versions 0.5 and 0.6 require extra confirmation via `--no-ssl` to allow running without SSL using the command `jupyterhub --no-ssl`. The `--no-ssl` command line option is not needed anymore in version 0.7.

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. It is important to note that this database contains *no* sensitive information other than **Hub** usernames.
- `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*.

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

1.2.2 Installation

See the project's README for help installing JupyterHub.

Planning your installation

Prior to beginning installation, it's helpful to consider some of the following:

- deployment system (bare metal, Docker)
- Authentication (PAM, OAuth, etc.)

- Spawner of singleuser notebook servers (Docker, Batch, etc.)
- Services (nbgrader, etc.)
- JupyterHub database (default SQLite; traditional RDBMS such as PostgreSQL,) MySQL, or other databases supported by SQLAlchemy)

Folders and 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

1.2.3 Configuration

JupyterHub is configured in two ways:

1. Configuration file
2. Command-line arguments

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.

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 the `c.Spawner.notebook_dir` trait from the command-line:

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

1.2.4 Networking

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.

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
```

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
```

1.2.5 Security

IMPORTANT: You should not run JupyterHub without SSL encryption on a public network.

Deprecation note: Removed `--no-ssl` in version 0.7.

JupyterHub versions 0.5 and 0.6 require extra confirmation via `--no-ssl` to allow running without SSL using the command `jupyterhub --no-ssl`. The `--no-ssl` command line option is not needed anymore in version 0.7.

Security is the most important aspect of configuring Jupyter. There are four 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)
4. Periodic security audits

Note that the **Hub** hashes all secrets (e.g., auth tokens) before storing them in its database. A loss of control over read-access to the database should have no security impact on your deployment.

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 `mydomain.tld` by your fully qualified domain name):

```
c.JupyterHub.ssl_key = '/etc/letsencrypt/live/{mydomain.tld}/privkey.pem'
c.JupyterHub.ssl_cert = '/etc/letsencrypt/live/{mydomain.tld}/fullchain.pem'
```

If the fully qualified domain name (FQDN) is `example.com`, the following would be the needed configuration:

```
c.JupyterHub.ssl_key = '/etc/letsencrypt/live/example.com/privkey.pem'
c.JupyterHub.ssl_cert = '/etc/letsencrypt/live/example.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 on **chain certificates**: If you are using a chain certificate, see also chained certificate for SSL in the JupyterHub troubleshooting FAQ).

Note: In certain cases, e.g. **behind SSL termination in nginx**, allowing no SSL running on the hub may be desired.

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 cookie secret in the configuration file itself, `jupyterhub_config.py`, as a binary string:

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

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, `jupyterhub_config.py`:

```
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.

Security audits

We recommend that you do periodic reviews of your deployment's security. It's good practice to keep JupyterHub, configurable-http-proxy, and nodejs versions up to date.

A handy website for testing your deployment is Qualsys' SSL analyzer tool.

1.2.6 Authentication and users

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.

Creating a whitelist of users

You can restrict which users are allowed to login with `Authenticator.whitelist`:

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

Users listed in the whitelist are added to the Hub database when the Hub is started.

Managing Hub administrators

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.**

Note: additional configuration examples are provided in this guide's Configuration Examples section.

Add or remove users from the Hub

Users can be added to and removed from the Hub via either the admin panel or REST API.

If a user is **added**, the user will be automatically 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.

After starting the Hub once, it is not sufficient to **remove** a user from the whitelist in your config file. You must also remove the user from the Hub's database, either by deleting the user from the admin page, or you can clear the `jupyterhub.sqlite` database and start fresh.

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.

1.2.7 Spawners and single-user notebook 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.

1.2.8 External services

JupyterHub has a REST API that can be used by external services like the cull_idle_servers script which monitors and kills idle single-user servers periodically. In order to run such an external service, you need to provide it an API token. In the case of `cull_idle_servers`, it is passed as the environment variable called `JPY_API_TOKEN`.

Currently there are two ways of registering that token with JupyterHub. The first one is to use the `jupyterhub` command to generate a token for a specific hub user:

```
jupyterhub token <username>
```

As of version 0.6.0, the preferred way of doing this is to first generate an API token:

```
openssl rand -hex 32
```

and then write it to your JupyterHub configuration file (note that the **key** is the token while the **value** is the username):

```
c.JupyterHub.api_tokens = {'token' : 'username'}
```

Upon restarting JupyterHub, you should see a message like below in the logs:

```
Adding API token for <username>
```

Now you can run your script, i.e. `cull_idle_servers`, by providing it the API token and it will authenticate through the REST API to interact with it.

1.3 How JupyterHub works

JupyterHub is a multi-user server that manages and proxies multiple instances of the single-user 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]`.

1.3.1 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`

1.3.2 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.

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.

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.

1.4 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.

1.4.1 Mitigations

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

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.

Disabling user config

If subdomains are not available or not desirable, 0.5 also adds an option `Spawner.disable_user_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 can 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.

1.4.2 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.

1.5 Using JupyterHub's REST API

Using the JupyterHub REST API, you can perform actions on the Hub, such as:

- checking which users are active
- adding or removing users
- stopping or starting single user notebook servers
- authenticating services

A REST API provides a standard way for users to get and send information to the Hub.

1.5.1 Creating an API token

To send requests using JupyterHub API, you must pass an API token with the request. You can create a token for an individual user using the following command:

```
jupyterhub token USERNAME
```

1.5.2 Adding tokens to the config file

You may also add a dictionary of API tokens and usernames to the hub's configuration file, `jupyterhub_config.py`:

```
c.JupyterHub.api_tokens = {
    'secret-token': 'username',
}
```

1.5.3 Making an API request

To authenticate your requests, pass the API token in the request's Authorization header.

Example: List the hub's users

Using the popular Python requests library, the following code sends an API request and an API token for authorization:

```
import requests

api_url = 'http://127.0.0.1:8081/hub/api'

r = requests.get(api_url + '/users',
```

```
    headers={
             'Authorization': 'token %s' % token,
            }
    )

r.raise_for_status()
users = r.json()
```

1.5.4 Learning more about the API

You can see the full JupyterHub REST API for details. The same REST API Spec can be viewed in a more interactive style on swagger's petstore. Both resources contain the same information and differ only in its display. Note: The Swagger specification is being renamed the OpenAPI Initiative.

Configuration Guide

- *Authenticators*
- *Spawners*
- *Services*
- *Configuration examples*
- *Upgrading JupyterHub and its database*
- *Troubleshooting*

1.6 Authenticators

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.

1.6.1 Basics of Authenticators

A basic Authenticator has one central method:

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']
```

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.

1.6.2 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'
}
```

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.*'
```

1.6.3 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.

1.6.4 Writing a custom authenticator

If you are interested in writing a custom authenticator, you can read this tutorial.

1.7 Spawners

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

1.7.1 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
  - both `DockerSpawner` and `SystemUserSpawner` also work with Docker Swarm for launching containers on remote machines
- 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

1.7.2 Spawner control methods

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.

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.

Spawner.stop

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

1.7.3 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
```

1.7.4 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.

`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:

```python
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.start` is called, this dictionary is accessible as `self.user_options`.

1.7.5 Writing a custom spawner

If you are interested in building a custom spawner, you can read this tutorial.

1.7.6 Spawners, resource limits, and guarantees (Optional)

Some spawners of the single-user notebook servers allow setting limits or guarantees on resources, such as CPU and memory. To provide a consistent experience for sysadmins and users, we provide a standard way to set and discover these resource limits and guarantees, such as for memory and CPU. For the limits and guarantees to be useful, the spawner must implement support for them.

Memory Limits & Guarantees

`c.Spawner.mem_limit`: A **limit** specifies the *maximum amount of memory* that may be allocated, though there is no promise that the maximum amount will be available. In supported spawners, you can set `c.Spawner.mem_limit` to limit the total amount of memory that a single-user notebook server can allocate. Attempting to use more memory than this limit will cause errors. The single-user notebook server can discover its own memory limit by looking at the environment variable `MEM_LIMIT`, which is specified in absolute bytes.

`c.Spawner.mem_guarantee`: Sometimes, a **guarantee** of a *minumum amount of memory* is desirable. In this case, you can set `c.Spawner.mem_guarantee` to to provide a guarantee that at minimum this much memory will always be available for the single-user notebook server to use. The environment variable `MEM_GUARANTEE` will also be set in the single-user notebook server.

The spawner's underlying system or cluster is responsible for enforcing these limits and providing these guarantees. If these values are set to `None`, no limits or guarantees are provided, and no environment values are set.

CPU Limits & Guarantees

`c.Spawner.cpu_limit`: In supported spawners, you can set `c.Spawner.cpu_limit` to limit the total number of cpu-cores that a single-user notebook server can use. These can be fractional - `0.5` means 50% of one CPU core, `4.0` is 4 cpu-cores, etc. This value is also set in the single-user notebook server's environment variable `CPU_LIMIT`. The limit does not claim that you will be able to use all the CPU up to your limit as other higher priority applications might be taking up CPU.

`c.Spawner.cpu_guarantee`: You can set `c.Spawner.cpu_guarantee` to provide a guarantee for CPU usage. The environment variable `CPU_GUARANTEE` will be set in the single-user notebook server when a guarantee is being provided.

The spawner's underlying system or cluster is responsible for enforcing these limits and providing these guarantees. If these values are set to `None`, no limits or guarantees are provided, and no environment values are set.

1.8 Services

With version 0.7, JupyterHub adds support for **Services**.

This section provides the following information about Services:

- Definition of a Service
- Properties of a Service
- Hub-Managed Services
- Launching a Hub-Managed Service
- Externally-Managed Services
- Writing your own Services
- Hub Authentication and Services

1.8.1 Definition of a Service

When working with JupyterHub, a **Service** is defined as a process that interacts with the Hub's REST API. A Service may perform a specific or action or task. For example, the following tasks can each be a unique Service:

- shutting down individuals' single user notebook servers that have been idle for some time
- registering additional web servers which should use the Hub's authentication and be served behind the Hub's proxy.

Two key features help define a Service:

- Is the Service **managed** by JupyterHub?
- Does the Service have a web server that should be added to the proxy's table?

Currently, these characteristics distinguish two types of Services:

- A **Hub-Managed Service** which is managed by JupyterHub
- An **Externally-Managed Service** which runs its own web server and communicates operation instructions via the Hub's API.

1.8.2 Properties of a Service

A Service may have the following properties:

- `name: str` - the name of the service
- `admin: bool (default - false)` - whether the service should have administrative privileges
- `url: str (default - None)` - The URL where the service is/should be. If a url is specified for where the Service runs its own web server, the service will be added to the proxy at `/services/:name`

If a service is also to be managed by the Hub, it has a few extra options:

- `command: (str/Popen list)` - Command for JupyterHub to spawn the service. - Only use this if the service should be a subprocess. - If command is not specified, the Service is assumed to be managed externally. - If a command is specified for launching the Service, the Service will be started and managed by the Hub.
- `environment: dict` - additional environment variables for the Service.
- `user: str` - the name of a system user to manage the Service. If unspecified, run as the same user as the Hub.

1.8.3 Hub-Managed Services

A **Hub-Managed Service** is started by the Hub, and the Hub is responsible for the Service's actions. A Hub-Managed Service can only be a local subprocess of the Hub. The Hub will take care of starting the process and restarts it if it stops.

While Hub-Managed Services share some similarities with notebook Spawners, there are no plans for Hub-Managed Services to support the same spawning abstractions as a notebook Spawner.

If you wish to run a Service in a Docker container or other deployment environments, the Service can be registered as an **Externally-Managed Service**, as described below.

1.8.4 Launching a Hub-Managed Service

A Hub-Managed Service is characterized by its specified `command` for launching the Service. For example, a 'cull idle' notebook server task configured as a Hub-Managed Service would include:

- the Service name,
- admin permissions, and
- the `command` to launch the Service which will cull idle servers after a timeout interval

This example would be configured as follows in `jupyterhub_config.py`:

```
c.JupyterHub.services = [
    {
        'name': 'cull-idle',
        'admin': True,
        'command': ['python', '/path/to/cull-idle.py', '--timeout']
    }
]
```

A Hub-Managed Service may also be configured with additional optional parameters, which describe the environment needed to start the Service process:

- `environment: dict` - additional environment variables for the Service.
- `user: str` - name of the user to run the server if different from the Hub. Requires Hub to be root.
- `cwd: path` directory in which to run the Service, if different from the Hub directory.

The Hub will pass the following environment variables to launch the Service:

```
JUPYTERHUB_SERVICE_NAME:   The name of the service
JUPYTERHUB_API_TOKEN:      API token assigned to the service
JUPYTERHUB_API_URL:        URL for the JupyterHub API (default, http://127.0.0.1:8080/
↪hub/api)
JUPYTERHUB_BASE_URL:       Base URL of the Hub (https://mydomain[:port]/)
JUPYTERHUB_SERVICE_PREFIX: URL path prefix of this service (/services/:service-name/)
```

```
JUPYTERHUB_SERVICE_URL:    Local URL where the service is expected to be listening.
                           Only for proxied web services.
```

For the previous 'cull idle' Service example, these environment variables would be passed to the Service when the Hub starts the 'cull idle' Service:

```
JUPYTERHUB_SERVICE_NAME: 'cull-idle'
JUPYTERHUB_API_TOKEN: API token assigned to the service
JUPYTERHUB_API_URL: http://127.0.0.1:8080/hub/api
JUPYTERHUB_BASE_URL: https://mydomain[:port]
JUPYTERHUB_SERVICE_PREFIX: /services/cull-idle/
```

See the JupyterHub GitHub repo for additional information about the `cull-idle` example.

1.8.5 Externally-Managed Services

You may prefer to use your own service management tools, such as Docker or systemd, to manage a JupyterHub Service. These **Externally-Managed Services**, unlike Hub-Managed Services, are not subprocesses of the Hub. You must tell JupyterHub which API token the Externally-Managed Service is using to perform its API requests. Each Externally-Managed Service will need a unique API token, because the Hub authenticates each API request and the API token is used to identify the originating Service or user.

A configuration example of an Externally-Managed Service with admin access and running its own web server is:

```
c.JupyterHub.services = [
    {
        'name': 'my-web-service',
        'url': 'https://10.0.1.1:1984',
        'api_token': 'super-secret',
    }
]
```

In this case, the `url` field will be passed along to the Service as `JUPYTERHUB_SERVICE_URL`.

1.8.6 Writing your own Services

When writing your own services, you have a few decisions to make (in addition to what your service does!):

1. Does my service need a public URL?
2. Do I want JupyterHub to start/stop the service?
3. Does my service need to authenticate users?

When a Service is managed by JupyterHub, the Hub will pass the necessary information to the Service via the environment variables described above. A flexible Service, whether managed by the Hub or not, can make use of these same environment variables.

When you run a service that has a url, it will be accessible under a `/services/` prefix, such as `https://myhub.horse/services/my-service/`. For your service to route proxied requests properly, it must take `JUPYTERHUB_SERVICE_PREFIX` into account when routing requests. For example, a web service would normally service its root handler at `'/'`, but the proxied service would need to serve `JUPYTERHUB_SERVICE_PREFIX + '/'`.

1.8.7 Hub Authentication and Services

JupyterHub 0.7 introduces some utilities for using the Hub's authentication mechanism to govern access to your service. When a user logs into JupyterHub, the Hub sets a **cookie (`jupyterhub-services`)**. The service can use this cookie to authenticate requests.

JupyterHub ships with a reference implementation of Hub authentication that can be used by services. You may go beyond this reference implementation and create custom hub-authenticating clients and services. We describe the process below.

The reference, or base, implementation is the `HubAuth` class, which implements the requests to the Hub.

To use HubAuth, you must set the `.api_token`, either programmatically when constructing the class, or via the `JUPYTERHUB_API_TOKEN` environment variable.

Most of the logic for authentication implementation is found in the `HubAuth.user_for_cookie` method, which makes a request of the Hub, and returns:

- None, if no user could be identified, or
- a dict of the following form:

```
{
  "name": "username",
  "groups": ["list", "of", "groups"],
  "admin": False, # or True
}
```

You are then free to use the returned user information to take appropriate action.

HubAuth also caches the Hub's response for a number of seconds, configurable by the `cookie_cache_max_age` setting (default: five minutes).

Flask Example

For example, you have a Flask service that returns information about a user. JupyterHub's HubAuth class can be used to authenticate requests to the Flask service. See the `service-whoami-flask` example in the JupyterHub GitHub repo for more details.

```
from functools import wraps
import json
import os
from urllib.parse import quote

from flask import Flask, redirect, request, Response

from jupyterhub.services.auth import HubAuth

prefix = os.environ.get('JUPYTERHUB_SERVICE_PREFIX', '/')

auth = HubAuth(
    api_token=os.environ['JUPYTERHUB_API_TOKEN'],
    cookie_cache_max_age=60,
)

app = Flask(__name__)


def authenticated(f):
```

```python
    """Decorator for authenticating with the Hub"""
    @wraps(f)
    def decorated(*args, **kwargs):
        cookie = request.cookies.get(auth.cookie_name)
        if cookie:
            user = auth.user_for_cookie(cookie)
        else:
            user = None
        if user:
            return f(user, *args, **kwargs)
        else:
            # redirect to login url on failed auth
            return redirect(auth.login_url + '?next=%s' % quote(request.path))
    return decorated


@app.route(prefix + '/')
@authenticated
def whoami(user):
    return Response(
        json.dumps(user, indent=1, sort_keys=True),
        mimetype='application/json',
        )
```

Authenticating tornado services with JupyterHub

Since most Jupyter services are written with tornado, we include a mixin class, `HubAuthenticated`, for quickly authenticating your own tornado services with JupyterHub.

Tornado's `@web.authenticated` method calls a Handler's `.get_current_user` method to identify the user. Mixing in `HubAuthenticated` defines `get_current_user` to use HubAuth. If you want to configure the HubAuth instance beyond the default, you'll want to define an `initialize` method, such as:

```python
class MyHandler(HubAuthenticated, web.RequestHandler):
    hub_users = {'inara', 'mal'}

    def initialize(self, hub_auth):
        self.hub_auth = hub_auth

    @web.authenticated
    def get(self):
        ...
```

The HubAuth will automatically load the desired configuration from the Service environment variables.

If you want to limit user access, you can whitelist users through either the `.hub_users` attribute or `.hub_groups`. These are sets that check against the username and user group list, respectively. If a user matches neither the user list nor the group list, they will not be allowed access. If both are left undefined, then any user will be allowed.

Implementing your own Authentication with JupyterHub

If you don't want to use the reference implementation (e.g. you find the implementation a poor fit for your Flask app), you can implement authentication via the Hub yourself. We recommend looking at the `HubAuth` class implementation for reference, and taking note of the following process:

1. retrieve the cookie `jupyterhub-services` from the request.

2. Make an API request `GET /hub/api/authorizations/cookie/jupyterhub-services/cookie-value`, where cookie-value is the url-encoded value of the `jupyterhub-services` cookie. This request must be authenticated with a Hub API token in the `Authorization` header. For example, with requests:

```
r = requests.get(
    '/'.join((["http://127.0.0.1:8081/hub/api",
               "authorizations/cookie/jupyterhub-services",
               quote(encrypted_cookie, safe=''),
    ]),
    headers = {
        'Authorization' : 'token %s' % api_token,
    },
)
r.raise_for_status()
user = r.json()
```

3. On success, the reply will be a JSON model describing the user:

```
{
  "name": "inara",
  "groups": ["serenity", "guild"],

}
```

An example of using an Externally-Managed Service and authentication is nbviewer, and an example of its configuration is found here. nbviewer can also be run as a Hub-Managed Service as described here.

1.9 Configuration examples

This section provides configuration files and tips for the following configurations:

- Example with GitHub OAuth
- Example with nginx reverse proxy

1.9.1 Example with GitHub OAuth

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 (using oauthenticator) 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.extra_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][oauth-setup], 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
```

1.9.2 Example with nginx reverse proxy

In the following example, we show configuration files for a JupyterHub server running locally on port `8000` but accessible from the outside on the standard SSL port `443`. This could be useful if the JupyterHub server machine is also hosting other domains or content on `443`. The goal here is to have the following be true:

- JupyterHub is running on a server, accessed *only* via `HUB.DOMAIN.TLD:443`
- On the same machine, `NO_HUB.DOMAIN.TLD` strictly serves different content, also on port `443`
- `nginx` is used to manage the web servers / reverse proxy (which means that only nginx will be able to bind two servers to `443`)
- After testing, the server in question should be able to score an A+ on the Qualys SSL Labs SSL Server Test

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

```
# Force the proxy to only listen to connections to 127.0.0.1
c.JupyterHub.ip = '127.0.0.1'
```

The `nginx` server config files are fairly standard fare except for the two `location` blocks within the `HUB.DOMAIN.TLD` config file:

```
# HTTP server to redirect all 80 traffic to SSL/HTTPS
server {
    listen 80;
    server_name HUB.DOMAIN.TLD;

    # Tell all requests to port 80 to be 302 redirected to HTTPS
    return 302 https://$host$request_uri;
}

# HTTPS server to handle JupyterHub
server {
    listen 443;
    ssl on;

    server_name HUB.DOMAIN.TLD;

    ssl_certificate /etc/letsencrypt/live/HUB.DOMAIN.TLD/fullchain.pem
    ssl_certificate_key /etc/letsencrypt/live/HUB.DOMAIN.TLD/privkey.pem

    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
    ssl_prefer_server_ciphers on;
    ssl_dhparam /etc/ssl/certs/dhparam.pem;
    ssl_ciphers 'ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-
↪AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-DSS-
↪AES128-GCM-SHA256:kEDH+AESGCM:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-
↪SHA256:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES256-SHA384:ECDHE-
↪ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-AES128-
↪SHA256:DHE-RSA-AES128-SHA:DHE-DSS-AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-DSS-
↪AES256-SHA:DHE-RSA-AES256-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-
↪SHA256:AES256-SHA256:AES128-SHA:AES256-SHA:AES:CAMELLIA:DES-CBC3-SHA:!aNULL:!eNULL:!
↪EXPORT:!DES:!RC4:!MD5:!PSK:!aECDH:!EDH-DSS-DES-CBC3-SHA:!EDH-RSA-DES-CBC3-SHA:!KRB5-
↪DES-CBC3-SHA';
    ssl_session_timeout 1d;
    ssl_session_cache shared:SSL:50m;
    ssl_stapling on;
    ssl_stapling_verify on;
    add_header Strict-Transport-Security max-age=15768000;
```

```
    # Managing literal requests to the JupyterHub front end
    location / {
        proxy_pass https://127.0.0.1:8000;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    # Managing WebHook/Socket requests between hub user servers and external proxy
    location ~* /(api/kernels/[^/]+/(channels|iopub|shell|stdin)|terminals/websocket)/
↪? {
        proxy_pass https://127.0.0.1:8000;

        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        # WebSocket support
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $connection_upgrade;

    }

    # Managing requests to verify letsencrypt host
    location ~ /.well-known {
        allow all;
    }


}
```

nginx will now be the front facing element of JupyterHub on 443 which means it is also free to bind other servers, like NO_HUB.DOMAIN.TLD to the same port on the same machine and network interface. In fact, one can simply use the same server blocks as above for NO_HUB and simply add line for the root directory of the site as well as the applicable location call:

```
server {
    listen 80;
    server_name NO_HUB.DOMAIN.TLD;

    # Tell all requests to port 80 to be 302 redirected to HTTPS
    return 302 https://$host$request_uri;
}

server {
    listen 443;
    ssl on;

    # INSERT OTHER SSL PARAMETERS HERE AS ABOVE

    # Set the appropriate root directory
    root /var/www/html

    # Set URI handling
    location / {
        try_files $uri $uri/ =404;
    }
```

```
    # Managing requests to verify letsencrypt host
    location ~ /.well-known {
        allow all;
    }

}
```

Now just restart nginx, restart the JupyterHub, and enjoy accessing https://HUB.DOMAIN.TLD while serving other content securely on https://NO_HUB.DOMAIN.TLD.

1.10 Upgrading JupyterHub and its database

From time to time, you may wish to upgrade JupyterHub to take advantage of new releases. Much of this process is automated using scripts, such as those generated by alembic for database upgrades. Before upgrading a JupyterHub deployment, it's critical to backup your data and configurations before shutting down the JupyterHub process and server.

1.10.1 Databases: SQLite (default) or RDBMS (PostgreSQL, MySQL)

The default database for JupyterHub is a SQLite database. We have chosen SQLite as JupyterHub's default for its lightweight simplicity in certain uses such as testing, small deployments and workshops.

When running a long term deployment or a production system, we recommend using a traditional RDBMS database, such as PostgreSQL or MySQL, that supports the SQL `ALTER TABLE` statement.

For production systems, SQLite has some disadvantages when used with JupyterHub:

- `upgrade-db` may not work, and you may need to start with a fresh database
- `downgrade-db` **will not** work if you want to rollback to an earlier version, so backup the `jupyterhub.sqlite` file before upgrading

The sqlite documentation provides a helpful page about when to use sqlite and where traditional RDBMS may be a better choice.

1.10.2 The upgrade process

Four fundamental process steps are needed when upgrading JupyterHub and its database:

1. Backup JupyterHub database
2. Backup JupyterHub configuration file
3. Shutdown the Hub
4. Upgrade JupyterHub
5. Upgrade the database using run `jupyterhub upgrade-db`

Let's take a closer look at each step in the upgrade process as well as some additional information about JupyterHub databases.

Backup JupyterHub database

To prevent unintended loss of data or configuration information, you should back up the JupyterHub database (the default SQLite database or a RDBMS database using PostgreSQL, MySQL, or others supported by SQLAlchemy):

- If using the default SQLite database, back up the `jupyterhub.sqlite` database.
- If using an RDBMS database such as PostgreSQL, MySQL, or other supported by SQLAlchemy, back up the JupyterHub database.

Losing the Hub database is often not a big deal. Information that resides only in the Hub database includes:

- active login tokens (user cookies, service tokens)
- users added via GitHub UI, instead of config files
- info about running servers

If the following conditions are true, you should be fine clearing the Hub database and starting over:

- users specified in config file
- user servers are stopped during upgrade
- don't mind causing users to login again after upgrade

Backup JupyterHub configuration file

Additionally, backing up your configuration file, `jupyterhub_config.py`, to a secure location.

Shutdown JupyterHub

Prior to shutting down JupyterHub, you should notify the Hub users of the scheduled downtime. This gives users the opportunity to finish any outstanding work in process.

Next, shutdown the JupyterHub service.

Upgrade JupyterHub

Follow directions that correspond to your package manager, `pip` or `conda`, for the new JupyterHub release. These directions will guide you to the specific command. In general, `pip install -U jupyterhub` or `conda upgrade jupyterhub`

Upgrade JupyterHub databases

To run the upgrade process for JupyterHub databases, enter:

```
jupyterhub upgrade-db
```

1.10.3 Upgrade checklist

1. Backup JupyterHub database:
   - `jupyterhub.sqlite` when using the default sqlite database
   - Your JupyterHub database when using an RDBMS

2. Backup JupyterHub configuration file: `jupyterhub_config.py`
3. Shutdown the Hub
4. Upgrade JupyterHub
   - `pip install -U jupyterhub` when using `pip`
   - `conda upgrade jupyterhub` when using `conda`
5. Upgrade the database using run `jupyterhub upgrade-db`

1.11 Troubleshooting

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

Behavior

- JupyterHub proxy fails to start
- sudospawner fails to run

Errors

- 500 error after spawning my single-user server

How do I...?

- Use a chained SSL certificate
- Install JupyterHub without a network connection
- I want access to the whole filesystem, but still default users to their home directory
- How do I increase the number of pySpark executors on YARN?
- How do I use JupyterLab's prerelease version with JupyterHub?
- How do I set up JupyterHub for a workshop (when users are not known ahead of time)?

Troubleshooting commands

1.11.1 Behavior

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`

sudospawner fails to run

If the sudospawner script is not found in the path, sudospawner will not run. To avoid this, specify sudospawner's absolute path. For example, start jupyterhub with:

```
jupyterhub --SudoSpawner.sudospawner_path='/absolute/path/to/sudospawner'
```

or add:

```
c.SudoSpawner.sudospawner_path = '/absolute/path/to/sudospawner'
```

to the config file, `jupyterhub_config.py`.

1.11.2 Errors

500 error after spawning my single-user server

You receive a 500 error when accessing the URL `/user/<your_name>/...`. This is often seen when your single-user server cannot verify your user cookie 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)

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/<your_name>` after logging in or clicking "Start my server". When a single-user notebook server receives a request, the notebook server 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, the response logged will be similar to 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 notebook server. If you don't see this message in the log, it may mean that your single-user notebook 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 notebook server, which may have more detailed information on the cause.

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 (typically 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.

1.11.3 How do I...?

Use a chained SSL certificate

Some certificate providers, i.e. Entrust, may provide you with a chained certificate that contains multiple files. If you are using a chained certificate you will need to concatenate the individual files by appending the chain cert and root cert to your host cert:

```
cat your_host.crt chain.crt root.crt > your_host-chained.crt
```

You would then set in your `jupyterhub_config.py` file the `ssl_key` and `ssl_cert` as follows:

```
c.JupyterHub.ssl_cert = your_host-chained.crt
c.JupyterHub.ssl_key = your_host.key
```

Example

Your certificate provider gives you the following files: `example_host.crt`, `Entrust_L1Kroot.txt` and `Entrust_Root.txt`.

Concatenate the files appending the chain cert and root cert to your host cert:

```
cat example_host.crt Entrust_L1Kroot.txt Entrust_Root.txt > example_host-chained.crt
```

You would then use the `example_host-chained.crt` as the value for JupyterHub's `ssl_cert`. You may pass this value as a command line option when starting JupyterHub or more conveniently set the `ssl_cert` variable in JupyterHub's configuration file, `jupyterhub_config.py`. In `jupyterhub_config.py`, set:

```
c.JupyterHub.ssl_cert = /path/to/example_host-chained.crt
c.JupyterHub.ssl_key = /path/to/example_host.key
```

where `ssl_cert` is example-chained.crt and ssl_key to your private key.

Then restart JupyterHub.

See also JupyterHub SSL encryption.

Install JupyterHub without a network connection

Both conda and pip can be used without a network connection. You can make your own repository (directory) of conda packages and/or wheels, and then install from there instead of the internet.

For instance, you can install JupyterHub with pip and configurable-http-proxy with npmbox:

```
pip wheel jupyterhub
npmbox configurable-http-proxy
```

I want access to the whole filesystem, but still default users to their home directory

Setting the following in `jupyterhub_config.py` will configure access to the entire filesystem and set the default to the user's home directory.

```
c.Spawner.notebook_dir = '/'
c.Spawner.default_url = '/home/%U' # %U will be replaced with the username
```

How do I increase the number of pySpark executors on YARN?

From the command line, pySpark executors can be configured using a command similar to this one:

```
pyspark --total-executor-cores 2 --executor-memory 1G
```

Cloudera documentation for configuring spark on YARN applications provides additional information. The pySpark configuration documentation is also helpful for programmatic configuration examples.

How do I use JupyterLab's prerelease version with JupyterHub?

While JupyterLab is still under active development, we have had users ask about how to try out JupyterLab with JupyterHub.

You need to install and enable the JupyterLab extension system-wide, then you can change the default URL to `/lab`.

For instance:

```
pip install jupyterlab
jupyter serverextension enable --py jupyterlab --sys-prefix
```

The important thing is that jupyterlab is installed and enabled in the single-user notebook server environment. For system users, this means system-wide, as indicated above. For Docker containers, it means inside the single-user docker image, etc.

In `jupyterhub_config.py`, configure the Spawner to tell the single-user notebook servers to default to JupyterLab:

```
c.Spawner.default_url = '/lab'
```

How do I set up JupyterHub for a workshop (when users are not known ahead of time)?

1. Set up JupyterHub using OAuthenticator for GitHub authentication
2. Configure whitelist to be an empty list in`jupyterhub_config.py`

3. Configure admin list to have workshop leaders be listed with administrator privileges.

Users will need a GitHub account to login and be authenticated by the Hub.

1.11.4 Troubleshooting commands

The following commands provide additional detail about installed packages, versions, and system information that may be helpful when troubleshooting a JupyterHub deployment. The commands are:

- System and deployment information

```
jupyter troubleshooting
```

- Kernel information

```
jupyter kernelspec list
```

- Debug logs when running JupyterHub

```
jupyterhub --debug
```

1.11.5 Toree integration with HDFS rack awareness script

The Apache Toree kernel will an issue, when running with JupyterHub, if the standard HDFS rack awareness script is used. This will materialize in the logs as a repeated WARN:

```
16/11/29 16:24:20 WARN ScriptBasedMapping: Exception running /etc/hadoop/conf/
↪topology_script.py some.ip.address
ExitCodeException exitCode=1:   File "/etc/hadoop/conf/topology_script.py", line 63
    print rack
             ^
SyntaxError: Missing parentheses in call to 'print'

    at `org.apache.hadoop.util.Shell.runCommand(Shell.java:576)`
```

In order to resolve this issue, there are two potential options.

1. Update HDFS core-site.xml, so the parameter "net.topology.script.file.name" points to a custom script (e.g. /etc/hadoop/conf/custom_topology_script.py). Copy the original script and change the first line point to a python two installation (e.g. /usr/bin/python).
2. In spark-env.sh add a Python 2 installation to your path (e.g. export PATH=/opt/anaconda2/bin:$PATH).

API Reference

- *The JupyterHub API*

1.12 The JupyterHub API

Release 0.7.2

Date Jul 20, 2017

JupyterHub also provides a REST API for administration of the Hub and users. The documentation on Using JupyterHub's REST API provides information on:

- Creating an API token
- Adding tokens to the configuration file (optional)
- Making an API request

The same JupyterHub API spec, as found here, is available in an interactive form here (on swagger's petstore). The OpenAPI Initiative (fka Swagger™) is a project used to describe and document RESTful APIs.

JupyterHub API Reference:

1.12.1 Authenticators

Module: `jupyterhub.auth`

Base Authenticator class and the default PAM Authenticator

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

Base class for implementing an authentication provider for JupyterHub

`add_user`(*user*)

Hook called when a user is added to JupyterHub

This is called:

- When a user first authenticates
- When the hub restarts, for all users.

This method may be a coroutine.

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.

Note that this should be idempotent, since it is called whenever the hub restarts for all users.

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, or None if Authentication failed

Return type username (str or None)

`check_whitelist`(*username*)

Check if a username is allowed to authenticate based on whitelist configuration

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

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

delete_user(*user*)

Hook called 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*)

Authenticate the user who is attempting to log in

Returns normalized username if successful, None otherwise.

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.

This is the outer API for authenticating a user. 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

Used in conjugation with *login_url* and *logout_url*.

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 handlers (list)

login_url(*base_url*)

Override this when registering a custom login handler

Generally used by authenticators that do not use simple form based authentication.

The subclass overriding this is responsible for making sure there is a handler available to handle the URL returned from this method, using the *get_handlers* method.

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 when registering a custom logout handler

The subclass overriding this is responsible for making sure there is a handler available to handle the URL returned from this method, using the *get_handlers* method.

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 the given username and return it

Override in subclasses if usernames need different normalization rules.

The default attempts to lowercase the username and apply *username_map* if it is set.

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 local UNIX user on the system.

Tested to work on FreeBSD and Linux, at least.

add_user (*user*)
Hook called whenever a new user is added

If self.create_system_users, the user will attempt to be created if it doesn't exist.

check_group_whitelist (*username*)
If group_whitelist is configured, check if authenticating user is part of group.

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

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

1.12.2 Spawners

Module: jupyterhub.spawner

Contains base Spawner class & default implementation

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

As JupyterHub supports multiple users, an instance of the Spawner subclass is created for each user. If there are 20 JupyterHub users, there will be 20 instances of the subclass.

format_string(*s*)
Render a Python format string

Uses *Spawner.template_namespace()* to populate format namespace.

Parameters **s** (*str*) – Python format-string to be formatted.

Returns Formatted string, rendered

Return type str

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

Doesn't expect shell expansion to happen.

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.

When overriding in subclasses, subclasses must call *super().get_env()*, extend the returned dict and return it.

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

get_state()
Save state of spawner into database.

A black box of extra state for custom spawners. The returned value of this is passed to *load_state*.

Subclasses should call *super().get_state()*, augment the state returned from there, and return that state.

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

Returns None if single-user process is running. Integer exit status (0 if unknown), if it is not running.

State transitions, behavior, and return response:

- If the Spawner has not been initialized (neither loaded state, nor called start), it should behave as if it is not running (status=0).
- If the Spawner has not finished starting, it should behave as if it is running (status=None).

Design assumptions about when *poll* may be called:

- On Hub launch: *poll* may be called before *start* when state is loaded on Hub launch. *poll* should return exit status 0 (unknown) if the Spawner has not been initialized via *load_state* or *start*.
- If *.start()* is async: *poll* may be called during any yielded portions of the *start* process. *poll* should return None when *start* is yielded, indicating that the *start* process has not yet completed.

start()
Start the single-user server

Returns the (ip, port) where the Hub can connect to the server.

Return type (str, int)

Changed in version 0.7: Return ip, port instead of setting on self.user.server directly.

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

If *now* is set to *False*, do not wait for the server to stop. Otherwise, wait for the server to stop before returning.

Must be a Tornado coroutine.

template_namespace()
Return the template namespace for format-string formatting.

Currently used on default_url and notebook_dir.

Subclasses may add items to the available namespace.

The default implementation includes:

```
{
  'username': user.name,
  'base_url': users_base_url,
}
```

Returns namespace for string formatting.

Return type ns (dict)

class jupyterhub.spawner.**LocalProcessSpawner**(***kwargs*)
A Spawner that uses *subprocess.Popen* to start single-user servers as local processes.

Requires local UNIX users matching the authenticated users to exist. Does not work on Windows.

This is the default spawner for JupyterHub.

1.12.3 Users

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.

1.12.4 Authenticating Services

Module: `jupyterhub.services.auth`

Authenticating services with JupyterHub

Cookies are sent to the Hub for verification, replying with a JSON model describing the authenticated user.

HubAuth can be used in any application, even outside tornado.

HubAuthenticated is a mixin class for tornado handlers that should authenticate with the Hub.

class `jupyterhub.services.auth.`**HubAuth**(***kwargs*)
A class for authenticating with JupyterHub

This can be used by any application.

If using tornado, use via *HubAuthenticated* mixin. If using manually, use the `.user_for_cookie(cookie_value)` method to identify the user corresponding to a given cookie value.

The following config must be set:

- api_token (token for authenticating with JupyterHub API), fetched from the JUPYTERHUB_API_TOKEN env by default.

The following config MAY be set:

- api_url: the base URL of the Hub's internal API, fetched from JUPYTERHUB_API_URL by default.
- cookie_cache_max_age: the number of seconds responses from the Hub should be cached.
- login_url (the *public* `/hub/login` URL of the Hub).
- cookie_name: the name of the cookie I should be using, if different from the default (unlikely).

get_user(*handler*)
Get the Hub user for a given tornado handler.

Checks cookie with the Hub to identify the current user.

Parameters **handler** (*tornado.web.RequestHandler*) – the current request handler

Returns

The user model, if a user is identified, None if authentication fails.

The 'name' field contains the user's name.

Return type user_model (dict)

user_for_cookie(*encrypted_cookie*, *use_cache=True*)
Ask the Hub to identify the user for a given cookie.

Parameters

- **encrypted_cookie** (*str*) – the cookie value (not decrypted, the Hub will do that)
- **use_cache** (*bool*) – Specify use_cache=False to skip cached cookie values (default: True)

Returns

The user model, if a user is identified, None if authentication fails.

The 'name' field contains the user's name.

Return type user_model (dict)

class jupyterhub.services.auth.**HubAuthenticated**

Mixin for tornado handlers that are authenticated with JupyterHub

A handler that mixes this in must have the following attributes/properties:

- .hub_auth: A HubAuth instance
- .hub_users: A set of usernames to allow. If left unspecified or None, username will not be checked.
- .hub_groups: A set of group names to allow. If left unspecified or None, groups will not be checked.

Examples:

```
class MyHandler(HubAuthenticated, web.RequestHandler):
    hub_users = {'inara', 'mal'}

    def initialize(self, hub_auth):
        self.hub_auth = hub_auth

    @web.authenticated
    def get(self):
        ...
```

check_hub_user(*user_model*)

Check whether Hub-authenticated user should be allowed.

Returns the input if the user should be allowed, None otherwise.

Override if you want to check anything other than the username's presence in hub_users list.

Parameters **user_model** (*dict*) – the user model returned from *HubAuth*

Returns The user model if the user should be allowed, None otherwise.

Return type user_model (dict)

get_current_user()

Tornado's authentication method

Returns The user model, if a user is identified, None if authentication fails.

Return type user_model (dict)

About JupyterHub

- *Change log summary*
- *Contributors*

1.13 Change log summary

For detailed changes from the prior release, click on the version number, and its link will bring up a GitHub listing of changes. Use `git log` on the command line for details.

1.13.1 Unreleased 0.8

1.13.2 0.7

[0.7.2] - 2017-01-09

Added

- Support service environment variables and defaults in `jupyterhub-singleuser` for easier deployment of notebook servers as a Service.
- Add `--group` parameter for deploying `jupyterhub-singleuser` as a Service with group authentication.
- Include URL parameters when redirecting through `/user-redirect/`

Fixed

- Fix group authentication for HubAuthenticated services

0.7.1 - 2017-01-02

Added

- `Spawner.will_resume` for signaling that a single-user server is paused instead of stopped. This is needed for cases like `DockerSpawner.remove_containers = False`, where the first API token is re-used for subsequent spawns.
- Warning on startup about single-character usernames, caused by common `set('string')` typo in config.

Fixed

- Removed spurious warning about empty `next_url`, which is AOK.

0.7.0 - 2016-12-2

Added

- Implement Services API #705
- Add `/api/` and `/api/info` endpoints #675
- Add documentation for JupyterLab, pySpark configuration, troubleshooting, and more.
- Add logging of error if adding users already in database. #689
- Add HubAuth class for authenticating with JupyterHub. This class can be used by any application, even outside tornado.

- Add user groups.
- Add `/hub/user-redirect/...` URL for redirecting users to a file on their own server.

Changed

- Always install with setuptools but not eggs (effectively require `pip install .`) #722
- Updated formatting of changelog. #711
- Single-user server is provided by JupyterHub package, so single-user servers depend on JupyterHub now.

Fixed

- Fix docker repository location #719
- Fix swagger spec conformance and timestamp type in API spec
- Various redirect-loop-causing bugs have been fixed.

Removed

- Deprecate `--no-ssl` command line option. It has no meaning and warns if used. #789
- Deprecate `%U` username substitution in favor of `{username}`. #748
- Removed deprecated SwarmSpawner link. #699

1.13.3 0.6

0.6.1 - 2016-05-04

Bugfixes on 0.6:

- statsd is an optional dependency, only needed if in use
- Notice more quickly when servers have crashed
- Better error pages for proxy errors
- Add Stop All button to admin panel for stopping all servers at once

0.6.0 - 2016-04-25

- 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

1.13.4 0.5 - 2016-03-07

- 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.

1.13.5 0.4

0.4.1 - 2016-02-03

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

0.4.0 - 2016-02-01

- 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.

1.13.6 0.3 - 2015-11-04

- 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

1.13.7 0.2 - 2015-07-12

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

1.13.8 0.1 - 2015-03-07

First preview release

1.14 Contributors

Project Jupyter thanks the following people for their help and contribution on JupyterHub:

- anderbubble
- betatim
- Carreau
- ckald
- cwaldbieser
- danielballen
- daradib
- datapolitan
- dblockow-d2dcrc
- dietmarw
- DominicFollettSmith
- dsblank
- ellisonbg
- evanlinde
- Fokko
- iamed18
- JamiesHQ
- jdavidheiser
- jhamrick
- josephtate
- kinuax

- KrishnaPG
- ksolan
- mbmilligan
- minrk
- mistercrunch
- Mistobaan
- mwmarkland
- nthiery
- ObiWahn
- ozancaglayan
- parente
- PeterDaveHello
- peterruppel
- rafael-ladislau
- rgbkrk
- robnagler
- ryanlovett
- Scrypy
- shreddd
- spoorthyv
- ssanderson
- takluyver
- temogen
- TimShawver
- Todd-Z-Li
- toobaz
- tsaeger
- vilhelmen
- willingc
- YannBrrd
- yuvipanda
- zoltan-fedor

CHAPTER 2

Indices and tables

- genindex
- modindex

CHAPTER 3

Questions? Suggestions?

- Jupyter mailing list
- Jupyter website

Python Module Index

j

jupyterhub.auth, 40
jupyterhub.services.auth, 45
jupyterhub.spawner, 42
jupyterhub.user, 44

Index

A

add_system_user() (jupyterhub.auth.LocalAuthenticator method), 42
add_user() (jupyterhub.auth.Authenticator method), 40
add_user() (jupyterhub.auth.LocalAuthenticator method), 42
authenticate() (jupyterhub.auth.Authenticator method), 40
Authenticator (class in jupyterhub.auth), 40

C

check_group_whitelist() (jupyterhub.auth.LocalAuthenticator method), 42
check_hub_user() (jupyterhub.services.auth.HubAuthenticated method), 46
check_whitelist() (jupyterhub.auth.Authenticator method), 40

D

delete_user() (jupyterhub.auth.Authenticator method), 40

E

escaped_name (jupyterhub.user.User attribute), 45

F

format_string() (jupyterhub.spawner.Spawner method), 43

G

get_args() (jupyterhub.spawner.Spawner method), 43
get_authenticated_user() (jupyterhub.auth.Authenticator method), 41
get_current_user() (jupyterhub.services.auth.HubAuthenticated method), 46
get_env() (jupyterhub.spawner.Spawner method), 43
get_handlers() (jupyterhub.auth.Authenticator method), 41
get_state() (jupyterhub.spawner.Spawner method), 43
get_user() (jupyterhub.services.auth.HubAuth method), 45

H

HubAuth (class in jupyterhub.services.auth), 45
HubAuthenticated (class in jupyterhub.services.auth), 46

J

jupyterhub.auth (module), 40
jupyterhub.services.auth (module), 45
jupyterhub.spawner (module), 42
jupyterhub.user (module), 44

L

LocalAuthenticator (class in jupyterhub.auth), 42
LocalProcessSpawner (class in jupyterhub.spawner), 44
login_url() (jupyterhub.auth.Authenticator method), 41
logout_url() (jupyterhub.auth.Authenticator method), 41

N

name (jupyterhub.user.User attribute), 44
normalize_username() (jupyterhub.auth.Authenticator method), 41

O

options_from_form() (jupyterhub.spawner.Spawner method), 43

P

PAMAuthenticator (class in jupyterhub.auth), 42
poll() (jupyterhub.spawner.Spawner method), 43
post_spawn_stop() (jupyterhub.auth.Authenticator method), 42
pre_spawn_start() (jupyterhub.auth.Authenticator method), 42

S

Server (class in jupyterhub.user), 44

server (jupyterhub.user.User attribute), 45
Spawner (class in jupyterhub.spawner), 42
spawner (jupyterhub.user.User attribute), 45
start() (jupyterhub.spawner.Spawner method), 44
stop() (jupyterhub.spawner.Spawner method), 44
system_user_exists() (jupyterhub.auth.LocalAuthenticator static method), 42

T

template_namespace() (jupyterhub.spawner.Spawner method), 44

U

User (class in jupyterhub.user), 44
user_for_cookie() (jupyterhub.services.auth.HubAuth method), 45

V

validate_username() (jupyterhub.auth.Authenticator method), 42