JupyterHub Documentation

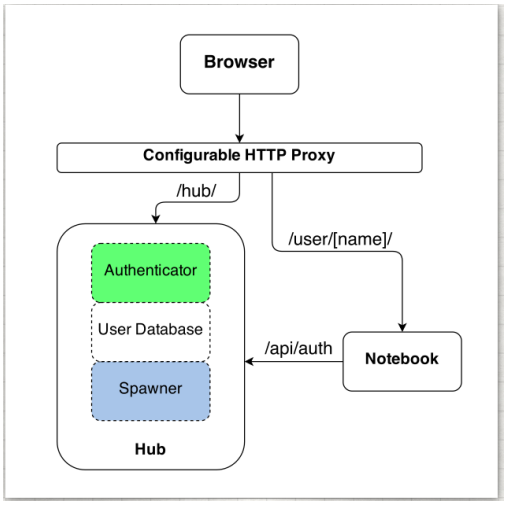
Release 0.9.0.dev

Project Jupyter team

Mar 07, 2018

Contents

JupyterHub, a multi-user **Hub**, spawns, manages, and proxies multiple instances of the single-user Jupyter notebook server. 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 performs the following functions:

- 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, JupyterHub also provides a REST API.

CHAPTER 1

Contents

CHAPTER 2

Indices and tables

- genindex
- modindex

CHAPTER 3

Questions? Suggestions?

- Jupyter mailing list
- Jupyter website

CHAPTER 4

Full Table of Contents

4.1 Installation Guide

4.1.1 Quickstart

Prerequisites

Before installing JupyterHub, you will need:

- a Linux/Unix based system
- Python 3.4 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), you will need:

- Jupyter Notebook version 4 or greater

Installation

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

pip, npm:

```
python3 -m pip install jupyterhub
npm install -g configurable-http-proxy
python3 -m pip install notebook  # needed if running the notebook servers locally
```

conda (one command installs jupyterhub and proxy):

```
conda install -c conda-forge jupyterhub  # installs jupyterhub and proxy
conda install notebook  # needed if running the notebook servers locally
```

Test your installation. If installed, these commands should return the packages' help contents:

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

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 in** to 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*. This requires additional configuration of the system.

4.1.2 Using Docker

Important: We highly recommend following the Zero to JupyterHub tutorial for installing JupyterHub.

Alternate installation using Docker

A ready to go docker image 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.

4.1.3 Installation Basics

Platform support

JupyterHub is supported on Linux/Unix based systems. 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).

JupyterHub officially **does not** support Windows. You may be able to use JupyterHub on Windows if you use a Spawner and Authenticator that work on Windows, but the JupyterHub defaults will not. Bugs reported on Windows will not be accepted, and the test suite will not run on Windows. Small patches that fix minor Windows compatibility issues (such as basic installation) **may** be accepted, however. For Windows-based systems, we would recommend running JupyterHub in a docker container or Linux VM.

Additional Reference: Tornado's documentation on Windows platform support

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

4.2 Getting Started

4.2.1 Configuration Basics

The section contains basic information about configuring settings for a JupyterHub deployment. The Technical Reference documentation provides additional details.

This section will help you learn how to:

- generate a default configuration file, `jupyterhub_config.py`
- start with a specific configuration file
- configure JupyterHub using command line options
- find information and examples for some common deployments

Generate a default config file

On startup, JupyterHub will look by default for a configuration file, `jupyterhub_config.py`, in the current working directory.

To generate a default config file, `jupyterhub_config.py`:

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

This default `jupyterhub_config.py` file contains comments and guidance for all configuration variables and their default values. We recommend storing configuration files in the standard UNIX filesystem location, i.e. `/etc/jupyterhub`.

Start with a specific config file

You can load a specific config file and start JupyterHub using:

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

If you have stored your configuration file in the recommended UNIX filesystem location, `/etc/jupyterhub`, the following command will start JupyterHub using the configuration file:

```
jupyterhub -f /etc/jupyterhub/jupyterhub_config.py
```

The IPython documentation provides additional information on the config system that Jupyter uses.

Configure using command line options

To display all command line options that are available for configuration:

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

Configuration using the command line options is done when launching JupyterHub. For example, to start JupyterHub on `10.0.1.2:443` with https, you would enter:

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

All configurable options may technically be set on the command-line, though some are inconvenient to type. To set a particular configuration parameter, `c.Class.trait`, you would use the command line option, `--Class.trait`, when starting JupyterHub. For example, to configure the `c.Spawner.notebook_dir` trait from the command-line, use the `--Spawner.notebook_dir` option:

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

Configure for various deployment environments

The default authentication and process spawning mechanisms can be replaced, and specific authenticators and spawners can be set in the configuration file. This enables JupyterHub to be used with a variety of authentication methods or process control and deployment environments. Some examples, meant as illustration, are:

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

4.2.2 Networking basics

This section will help you with basic proxy and network configuration to:

- set the proxy's IP address and port
- set the proxy's REST API URL
- configure the Hub if the Proxy or Spawners are remote or isolated
- set the `hub_connect_ip` which services will use to communicate with the hub

Set 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 Proxy's main IP address and port can be done with the following JupyterHub **command line options**:

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

Or by placing the following lines in a **configuration file**, `jupyterhub_config.py`:

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

Port 443 is used in the examples 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.

Set the Proxy's REST API communication URL (optional)

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

Set api_url

The URL to access the API, `c.configurableHTTPProxy.api_url`, is configurable. An example entry to set the proxy's API URL in `jupyterhub_config.py` is:

```
c.ConfigurableHTTPProxy.api_url = 'http://10.0.1.4:5432'
```

proxy_api_ip and proxy_api_port (Deprecated in 0.8)

If running the Proxy separate from the Hub, configure the REST API communication IP address and port by adding this to the `jupyterhub_config.py` file:

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

We recommend using the proxy's `api_url` setting instead of the deprecated settings, `proxy_api_ip` and `proxy_api_port`.

Configure the Hub if the Proxy or Spawners are remote or isolated

The Hub service listens only on `localhost` (port 8081) by default. The Hub 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
```

Added in 0.8: The `c.JupyterHub.hub_connect_ip` setting is the ip address or hostname that other services should use to connect to the Hub. A common configuration for, e.g. docker, is:

```
c.JupyterHub.hub_ip = '0.0.0.0'  # listen on all interfaces
c.JupyterHub.hub_connect_ip = '10.0.1.4'  # ip as seen on the docker network. Can␣
↪also be a hostname.
```

4.2.3 Security settings

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

Security is the most important aspect of configuring Jupyter. Three configuration settings are the main aspects of 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)

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 minimal impact on your deployment; if your database has been compromised, it is still a good idea to revoke existing tokens.

Enabling SSL encryption

Since JupyterHub includes authentication and allows arbitrary code execution, you should not run it without SSL (HTTPS).

Using an SSL certificate

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 `jupyterhub_config.py` configuration file as follows:

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

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.

If you are using a **chain certificate**, see also chained certificate for SSL in the JupyterHub troubleshooting FAQ.

Using letsencrypt

It is also possible to use letsencrypt 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'
```

If SSL termination happens outside of the Hub

In certain cases, e.g. behind SSL termination in NGINX, allowing no SSL running on the hub may be the desired configuration option.

Cookie secret

The cookie secret is an encryption key, used to encrypt the browser cookies which are used for authentication. Three common methods are described for generating and configuring the cookie secret.

Generating and storing as a cookie secret file

The cookie secret should be 32 random bytes, encoded as hex, and is typically stored in a `jupyterhub_cookie_secret` file. An example command to generate the `jupyterhub_cookie_secret` file is:

```
openssl rand -hex 32 > /srv/jupyterhub/jupyterhub_cookie_secret
```

In most deployments of JupyterHub, you should point this to a secure location on the file system, such as `/srv/jupyterhub/jupyterhub_cookie_secret`.

The location of the `jupyterhub_cookie_secret` file can be specified in the `jupyterhub_config.py` file as follows:

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

Generating and storing as an environment variable

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 32`
```

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.

Generating and storing as a binary string

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('64 CHAR HEX STRING')
```

Important: If the cookie secret value changes for the Hub, all single-user notebook servers must also be restarted.

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`).

Generating and storing token in the configuration file

Or you can set the value in the configuration file, `jupyterhub_config.py`:

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

Generating and storing as an environment variable

You can pass this value of the proxy authentication token to the Hub and Proxy using 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.

Default if token is not set

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).

4.2.4 Authentication and User Basics

The default Authenticator uses PAM to authenticate system users with their username and password. With the default Authenticator, any user with an account and password on the system will be allowed to login.

Create a whitelist of users

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

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

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

Configure admins (`admin_users`)

Admin users of JupyterHub, `admin_users`, can add and remove users from the user `whitelist`. `admin_users` can take actions on other users' behalf, such as stopping and restarting their servers.

A set of initial admin users, `admin_users` can configured be as follows:

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

Users in the admin list are automatically added to the user `whitelist`, if they are not already present.

Give admin access to other users' notebook servers (`admin_access`)

Since the default `JupyterHub.admin_access` setting is False, the admins do not have permission to log in to the single user notebook servers owned by *other users*. If `JupyterHub.admin_access` is set to True, then admins have permission to log in *as other users* on their respective machines, for debugging. **As a courtesy, you should make sure your users know if admin_access is enabled.**

Add or remove users from the Hub

Users can be added to and removed from the Hub via either the admin panel or the REST API. When 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 JupyterHub's admin page, or you can clear the `jupyterhub.sqlite` database and start fresh.

Use LocalAuthenticator to create system users

The `LocalAuthenticator` is a special kind of authenticator that has the ability to manage users on the local system. When you try to add a new user to the Hub, a `LocalAuthenticator` will check if the user already exists. If you set the configuration value, `create_system_users`, to `True` in the configuration file, the `LocalAuthenticator` has the privileges to add users to the system. The setting in the config file is:

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

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. This approach is not recommended when running JupyterHub in situations where JupyterHub users map directly onto the system's UNIX users.

Use OAuthenticator to support OAuth with popular service providers

JupyterHub's OAuthenticator currently supports the following popular services:

- Auth0
- Bitbucket
- CILogon
- GitHub
- GitLab
- Globus
- Google
- MediaWiki
- Okpy
- OpenShift

A generic implementation, which you can use for OAuth authentication with any provider, is also available.

4.2.5 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 `jupyter_notebook_config.py` config file. Each user may have one of these files in `$HOME/.jupyter/`. Jupyter also supports loading system-wide config files from `/etc/jupyter/`, which is the place to put configuration that you want to affect all of your users.

4.2.6 External services

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, shutting down individuals' single user notebook servers that have been is a good example of a task that could be automated by a Service. Let's look at how the cull_idle_servers script can be used as a Service.

Real-world example to cull idle servers

JupyterHub has a REST API that can be used by external services. This document will:

- explain some basic information about API tokens
- clarify that API tokens can be used to authenticate to single-user servers as of version 0.8.0
- show how the cull_idle_servers script can be:
  - used in a Hub-managed service
  - run as a standalone script

Both examples for `cull_idle_servers` will communicate tasks to the Hub via the REST API.

API Token basics

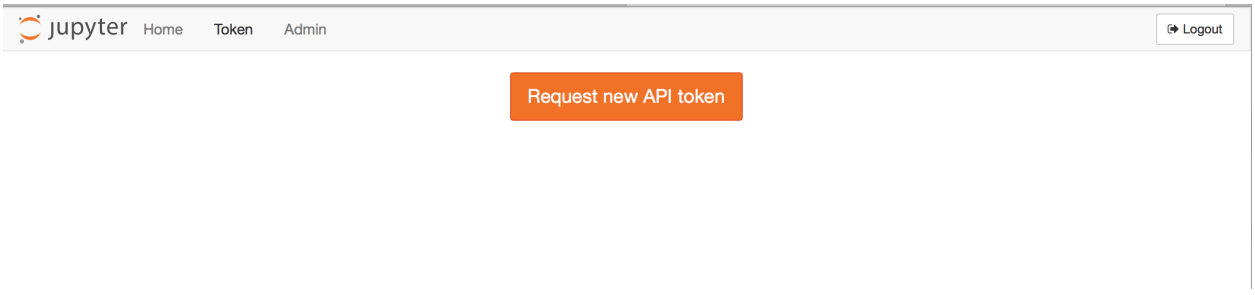
Create an API token

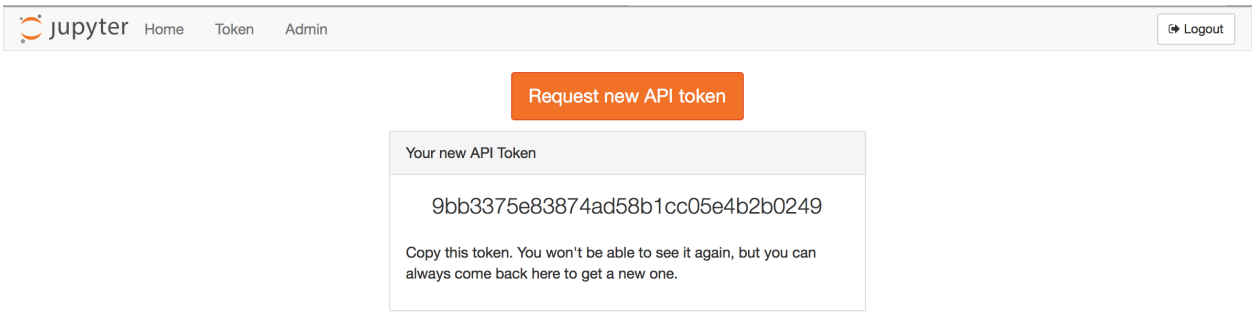
To run such an external service, an API token must be created and provided to the service.

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

```
openssl rand -hex 32
```

In version 0.8.0, a TOKEN request page for generating an API token is available from the JupyterHub user interface:





Pass environment variable with token to the Hub

In the case of `cull_idle_servers`, it is passed as the environment variable called `JUPYTERHUB_API_TOKEN`.

Use API tokens for services and tasks that require external access

While API tokens are often associated with a specific user, API tokens can be used by services that require external access for activities that may not correspond to a specific human, e.g. adding users during setup for a tutorial or workshop. Add a service and its API token to the JupyterHub configuration file, `jupyterhub_config.py`:

```
c.JupyterHub.services = [
    {'name': 'adding-users', 'api_token': 'super-secret-token'},
]
```

Restart JupyterHub

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

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

Authenticating to single-user servers using API token

In JupyterHub 0.7, there is no mechanism for token authentication to single-user servers, and only cookies can be used for authentication. 0.8 supports using JupyterHub API tokens to authenticate to single-user servers.

Configure `cull-idle` to run as a Hub-Managed Service

In `jupyterhub_config.py`, add the following dictionary for the `cull-idle` Service to the `c.JupyterHub.services` list:

```
c.JupyterHub.services = [
    {
        'name': 'cull-idle',
        'admin': True,
        'command': 'python cull_idle_servers.py --timeout=3600'.split(),
    }
]
```

where:

- `'admin': True` indicates that the Service has 'admin' permissions, and
- `'command'` indicates that the Service will be launched as a subprocess, managed by the Hub.

Run `cull-idle` manually as a standalone script

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.

This will run `cull-idle` manually. `cull-idle` can be run as a standalone script anywhere with access to the Hub, and will periodically check for idle servers and shut them down via the Hub's REST API. In order to shutdown the servers, the token given to cull-idle must have admin privileges.

Generate an API token and store it in the `JUPYTERHUB_API_TOKEN` environment variable. Run `cull_idle_servers.py` manually.

```
export JUPYTERHUB_API_TOKEN='token'
python cull_idle_servers.py [--timeout=900] [--url=http://127.0.0.1:8081/hub/api]
```

4.3 Technical Reference

4.3.1 Technical Overview

The **Technical Overview** section gives you a high-level view of:

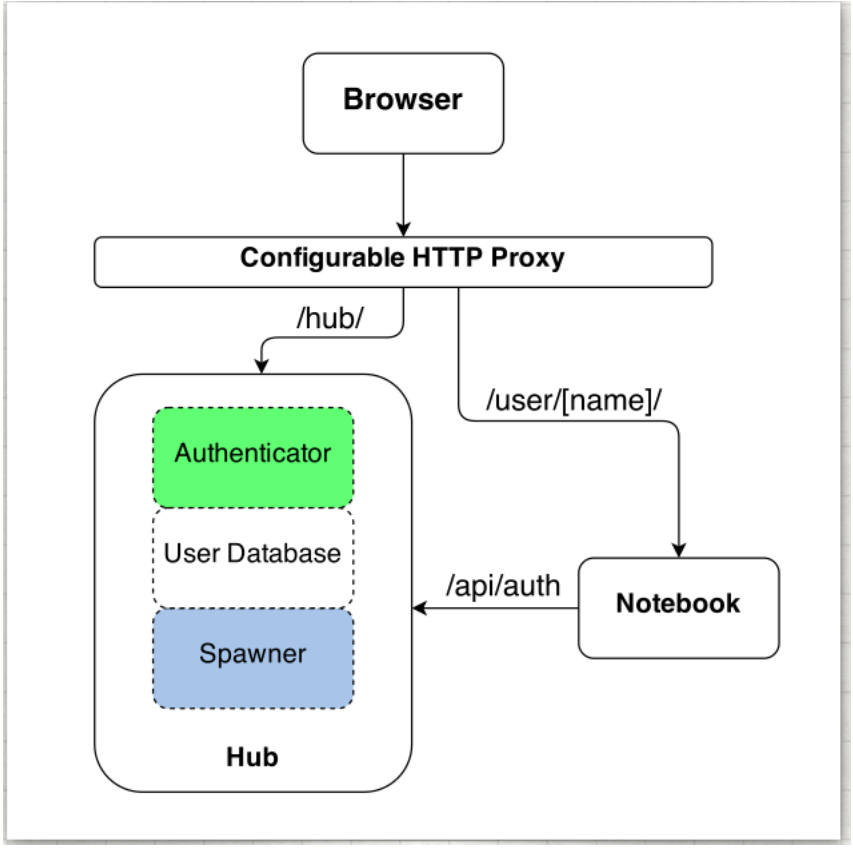
- JupyterHub's Subsystems: Hub, Proxy, Single-User Notebook Server
- how the subsystems interact
- the process from JupyterHub access to user login
- JupyterHub's default behavior
- customizing JupyterHub

The goal of this section is to share a deeper technical understanding of JupyterHub and how it works.

The Subsystems: Hub, Proxy, Single-User Notebook Server

JupyterHub is a set of processes that together provide a single user Jupyter Notebook server for each person in a group. Three major subsystems are started by the `jupyterhub` command line program:

- **Hub** (Python/Tornado): manages user accounts, authentication, and coordinates Single User Notebook Servers using 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. configurable http proxy (node-http-proxy) is the default proxy.
- **Single-User Notebook Server** (Python/Tornado): a dedicated, single-user, Jupyter Notebook server is started for each user on the system when the user logs in. The object that starts the single-user notebook servers is called a **Spawner**.



How the Subsystems Interact

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

The basic principles of operation are:

- The Hub spawns the proxy (in the default JupyterHub configuration)
- The proxy forwards all requests to the Hub by default
- The Hub handles login, and spawns single-user notebook servers on demand
- The Hub configures the proxy to forward url prefixes to single-user notebook servers

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]`.

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.

The Process from JupyterHub Access to User Login

When a user accesses JupyterHub, the following events take place:

- Login data is handed to the Authenticator instance for validation
- The Authenticator returns the username if the login information is valid
- A single-user notebook server instance is spawned for the logged-in user
- When the single-user notebook server starts, the proxy is notified to forward requests to `/user/[username]/*` to the single-user notebook server.
- A cookie is set on `/hub/`, containing an encrypted token. (Prior to version 0.8, a cookie for `/user/[username]` was used too.)
- The browser is redirected to `/user/[username]`, and the request is handled by the single-user notebook server.

The single-user server identifies the user with the Hub via OAuth:

- on request, the single-user server checks a cookie
- if no cookie is set, redirect to the Hub for verification via OAuth
- after verification at the Hub, the browser is redirected back to the single-user server
- the token is verified and stored in a cookie
- if no user is identified, the browser is redirected back to `/hub/login`

Default Behavior

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 Notebook 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 which users are running and where, as well as storing 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 so that a **Hub** server restart will 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 section of the Security Settings document.

The location of these files can be specified via configuration settings. It is recommended that these files be stored in standard UNIX filesystem locations, such as `/etc/jupyterhub` for all configuration files and `/srv/jupyterhub` for all security and runtime files.

Customizing JupyterHub

There are two basic extension points for JupyterHub:

- How users are authenticated by Authenticators
- How user's single-user notebook server processes are started by Spawners

Each is governed by a customizable class, and JupyterHub ships with basic defaults for each.

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

4.3.2 Security Overview

The **Security Overview** section helps you learn about:

- the design of JupyterHub with respect to web security
- the semi-trusted user
- the available mitigations to protect untrusted users from each other
- the value of periodic security audits.

This overview also helps you obtain a deeper understanding of how JupyterHub works.

Semi-trusted and untrusted users

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 by the administrator. Much care is required to secure a Hub, 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*. 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 (i.e. single domain).

Protect users from each other

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 a user writing arbitrary HTML and serving it to another user because only the owner of a given single-user notebook server is allowed to view user-authored pages served by the given single-user notebook server.

To protect all users from each other, JupyterHub administrators must ensure that:

- A user **does not have permission** to modify their single-user notebook server, including:
  - A user **may not** install new packages in the Python environment that runs their single-user server.
  - If the `PATH` is used to resolve the single-user executable (instead of using 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.

Mitigate security issues

Several approaches to mitigating these issues with configuration options provided by JupyterHub include:

Enable subdomains

JupyterHub provides the ability to run single-user servers on their own subdomains. This 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 single-user server will be at `username.jupyter.mydomain.com`. This also 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. **If you do plan to serve untrusted users, enabling subdomains is highly encouraged**, as it resolves the cross-site issues.

Disable user config

If subdomains are not available or not desirable, JupyterHub provides a a configuration option `Spawner.disable_user_config`, which can be set to prevent the user-owned configuration files from being loaded. After implementing this option, PATHs and package installation and PATHs are the other things that the admin must enforce.

Prevent spawners from evaluating shell configuration files

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

Isolate packages using virtualenv

Package isolation is most easily handled by running the single-user server in a virtualenv with disabled system-site-packages. The user should not have permission to install packages into this environment.

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.

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.

4.3.3 Authenticators

The Authenticator is the mechanism for authorizing users to use the Hub and single user notebook servers.

The default PAM Authenticator

JupyterHub ships only with the default PAM-based Authenticator, for logging in with local user accounts via a username and password.

The OAuthenticator

Some login mechanisms, such as OAuth, don't map onto username and password authentication, and instead use tokens. When using these mechanisms, you can override the login handlers.

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

JupyterHub's OAuthenticator currently supports the following popular services:

- Auth0
- Bitbucket
- CILogon
- GitHub
- GitLab
- Globus
- Google
- MediaWiki
- Okpy
- OpenShift

A generic implementation, which you can use for OAuth authentication with any provider, is also available.

Additional Authenticators

- ldapauthenticator for LDAP
- tmpauthenticator for temporary accounts

Technical Overview of Authentication

How the Base Authenticator works

The base authenticator uses simple username and password authentication.

The base Authenticator has one central method:

Authenticator.authenticate method

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

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

- `username`
- `password`

The `authenticate` method's job is simple:

- return the 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']
```

Normalize 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'
}
```

Validate 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.*'
```

How to write a custom authenticator

You can use custom Authenticator subclasses to enable authentication via other mechanisms. 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. For example, the Authenticator methods, pre_spawn_start(user, spawner) and post_spawn_stop(user, spawner), are hooks that can be used to do auth-related startup (e.g. opening PAM sessions) and cleanup (e.g. closing PAM sessions).

See a list of custom Authenticators on the wiki.

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

Authentication state

JupyterHub 0.8 adds the ability to persist state related to authentication, such as auth-related tokens. If such state should be persisted, `.authenticate()` should return a dictionary of the form:

```
{
  'name': username,
  'auth_state': {
    'key': 'value',
  }
}
```

where `username` is the username that has been authenticated, and `auth_state` is any JSON-serializable dictionary.

Because `auth_state` may contain sensitive information, it is encrypted before being stored in the database. To store auth_state, two conditions must be met:

1. persisting auth state must be enabled explicitly via configuration

   ```
   c.Authenticator.enable_auth_state = True
   ```

2. encryption must be enabled by the presence of `JUPYTERHUB_CRYPT_KEY` environment variable, which should be a hex-encoded 32-byte key. For example:

   ```
   export JUPYTERHUB_CRYPT_KEY=$(openssl rand -hex 32)
   ```

JupyterHub uses Fernet to encrypt auth_state. To facilitate key-rotation, `JUPYTERHUB_CRYPT_KEY` may be a semicolon-separated list of encryption keys. If there are multiple keys present, the **first** key is always used to persist any new auth_state.

Using auth_state

Typically, if `auth_state` is persisted it is desirable to affect the Spawner environment in some way. This may mean defining environment variables, placing certificate in the user's home directory, etc. The `Authenticator.pre_spawn_start` method can be used to pass information from authenticator state to Spawner environment:

```
class MyAuthenticator(Authenticator):
    @gen.coroutine
    def authenticate(self, handler, data=None):
        username = yield identify_user(handler, data)
        upstream_token = yield token_for_user(username)
        return {
            'name': username,
            'auth_state': {
                'upstream_token': upstream_token,
            },
        }

    @gen.coroutine
    def pre_spawn_start(self, user, spawner):
        """Pass upstream_token to spawner via environment variable"""
        auth_state = yield user.get_auth_state()
```

(continues on next page)

(continued from previous page)

```
        if not auth_state:
            # auth_state not enabled
            return
        spawner.environment['UPSTREAM_TOKEN'] = auth_state['upstream_token']
```

JupyterHub as an OAuth provider

Beginning with version 0.8, JupyterHub is an OAuth provider.

4.3.4 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

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

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.

The return value of `Spawner.start` should be the (ip, port) of the running 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.ip = '127.0.0.1'
    self.port = random_port()
    yield self._actually_start_server_somehow()
    return (self.ip, self.port)
```

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.

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:

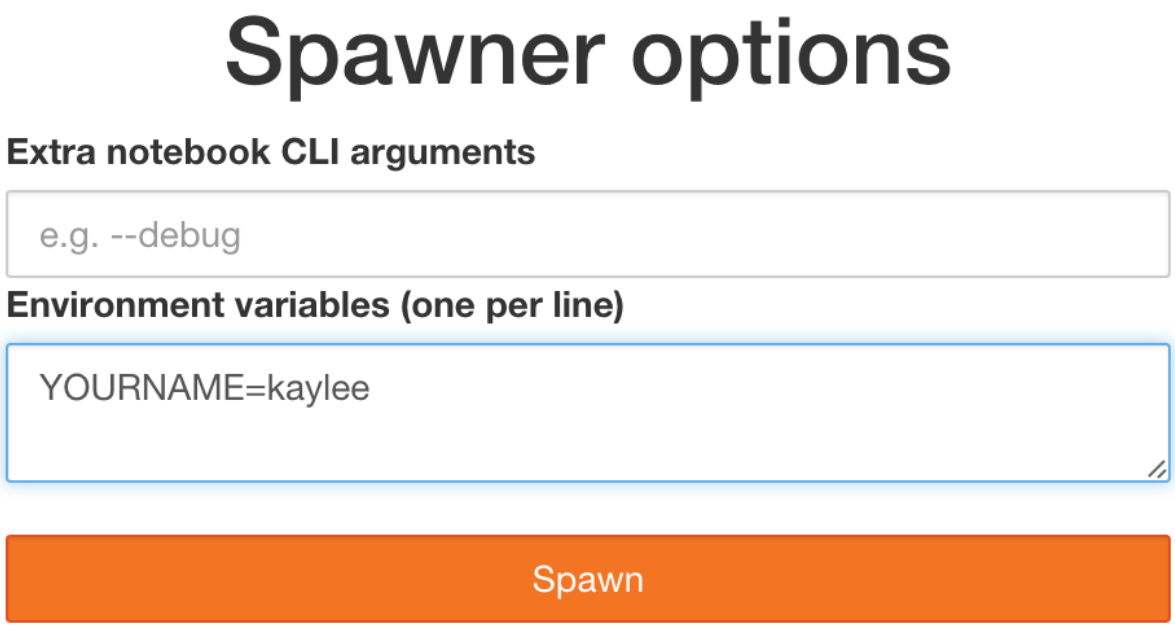
```
def get_state(self):
    """get the current state"""
    state = super().get_state()
    if self.pid:
        state['pid'] = self.pid
    return state

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
```

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:



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:

```
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
```

(continues on next page)

(continued from previous page)

```
    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`.

Writing a custom spawner

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

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**. For example, LocalProcessSpawner, the default spawner, does not support limits and guarantees. One of the spawners that supports limits and guarantees is the `systemdspawner`.

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.

4.3.5 Services

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

This section provides the following information about Services:

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.

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`

- `api_token: str (default - None)` - For Externally-Managed Services you need to specify an API token to perform API requests to the Hub

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.

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.

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.

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`.

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.

Note that JUPYTERHUB_SERVICE_PREFIX will contain a trailing slash. This must be taken into consideration when creating the service routes. If you include an extra slash you might get unexpected behavior. For example if your service has a /foo endpoint, the route would be JUPYTERHUB_SERVICE_PREFIX + foo, and /foo/bar would be JUPYTERHUB_SERVICE_PREFIX + foo/bar.

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 and in the HubAuth.user_for_token methods, 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', '/')
```

(continues on next page)

(continued from previous page)

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

app = Flask(__name__)


def authenticated(f):
    """Decorator for authenticating with the Hub"""
    @wraps(f)
    def decorated(*args, **kwargs):
        cookie = request.cookies.get(auth.cookie_name)
        token = request.headers.get(auth.auth_header_name)
        if cookie:
            user = auth.user_for_cookie(cookie)
        elif token:
            user = auth.user_for_token(token)
        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:

```
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 in [nbviewer README]_ section on securing the notebook viewer, and an example of its configuration is found here. nbviewer can also be run as a Hub-Managed Service as described [nbviewer README]_ section on securing the notebook viewer.

4.3.6 Writing a custom Proxy implementation

JupyterHub 0.8 introduced the ability to write a custom implementation of the proxy. This enables deployments with different needs than the default proxy, configurable-http-proxy (CHP). CHP is a single-process nodejs proxy that they Hub manages by default as a subprocess (it can be run externally, as well, and typically is in production deployments).

The upside to CHP, and why we use it by default, is that it's easy to install and run (if you have nodejs, you are set!). The downsides are that it's a single process and does not support any persistence of the routing table. So if the proxy process dies, your whole JupyterHub instance is inaccessible until the Hub notices, restarts the proxy, and restores the routing table. For deployments that want to avoid such a single point of failure, or leverage existing proxy infrastructure in their chosen deployment (such as Kubernetes ingress objects), the Proxy API provides a way to do that.

In general, for a proxy to be usable by JupyterHub, it must:

1. support websockets without prior knowledge of the URL where websockets may occur
2. support trie-based routing (i.e. allow different routes on `/foo` and `/foo/bar` and route based on specificity)
3. adding or removing a route should not cause existing connections to drop

Optionally, if the JupyterHub deployment is to use host-based routing, the Proxy must additionally support routing based on the Host of the request.

Subclassing Proxy

To start, any Proxy implementation should subclass the base Proxy class, as is done with custom Spawners and Authenticators.

```
from jupyterhub.proxy import Proxy

class MyProxy(Proxy):
    """My Proxy implementation"""
    ...
```

Starting and stopping the proxy

If your proxy should be launched when the Hub starts, you must define how to start and stop your proxy:

```
from tornado import gen
class MyProxy(Proxy):
    ...
    @gen.coroutine
    def start(self):
        """Start the proxy"""

    @gen.coroutine
    def stop(self):
        """Stop the proxy"""
```

These methods **may** be coroutines.

`c.Proxy.should_start` is a configurable flag that determines whether the Hub should call these methods when the Hub itself starts and stops.

Purely external proxies

Probably most custom proxies will be externally managed, such as Kubernetes ingress-based implementations. In this case, you do not need to define `start` and `stop`. To disable the methods, you can define `should_start = False` at the class level:

```
class MyProxy(Proxy):
    should_start = False
```

Adding and removing routes

At its most basic, a Proxy implementation defines a mechanism to add, remove, and retrieve routes. A proxy that implements these three methods is complete. Each of these methods **may** be a coroutine.

Definition: routespec

A routespec, which will appear in these methods, is a string describing a route to be proxied, such as `/user/name/`. A routespec will:

1. always end with `/`
2. always start with `/` if it is a path-based route `/proxy/path/`
3. precede the leading `/` with a host for host-based routing, e.g. `host.tld/proxy/path/`

Adding a route

When adding a route, JupyterHub may pass a JSON-serializable dict as a `data` argument that should be attacked to the proxy route. When that route is retrieved, the `data` argument should be returned as well. If your proxy implementation doesn't support storing data attached to routes, then your Python wrapper may have to handle storing the `data` piece itself, e.g in a simple file or database.

```
@gen.coroutine
def add_route(self, routespec, target, data):
    """Proxy `routespec` to `target`.

    Store `data` associated with the routespec
    for retrieval later.
    """
```

Adding a route for a user looks like this:

```
proxy.add_route('/user/pgeorgiou/', 'http://127.0.0.1:1227',
                {'user': 'pgeorgiou'})
```

Removing routes

`delete_route()` is given a routespec to delete. If there is no such route, `delete_route` should still succeed, but a warning may be issued.

```
@gen.coroutine
def delete_route(self, routespec):
    """Delete the route"""
```

Retrieving routes

For retrieval, you only *need* to implement a single method that retrieves all routes. The return value for this function should be a dictionary, keyed by `routespect`, of dicts whose keys are the same three arguments passed to `add_route` (`routespec`, `target`, `data`)

```
@gen.coroutine
def get_all_routes(self):
    """Return all routes, keyed by routespec"""
```

```
{
  '/proxy/path/': {
    'routespec': '/proxy/path/',
```

(continues on next page)

(continued from previous page)

```
    'target': 'http://...',
    'data': {},
  },
}
```

Note on activity tracking

JupyterHub can track activity of users, for use in services such as culling idle servers. As of JupyterHub 0.8, this activity tracking is the responsibility of the proxy. If your proxy implementation can track activity to endpoints, it may add a `last_activity` key to the `data` of routes retrieved in `.get_all_routes()`. If present, the value of `last_activity` should be an ISO8601 UTC date string:

```
{
  '/user/pgeorgiou/': {
    'routespec': '/user/pgeorgiou/',
    'target': 'http://127.0.0.1:1227',
    'data': {
      'user': 'pgeourgiou',
      'last_activity': '2017-10-03T10:33:49.570Z',
    },
  },
}
```

If the proxy does not track activity, then only activity to the Hub itself is tracked, and services such as cull-idle will not work.

Now that `notebook-5.0` tracks activity internally, we can retrieve activity information from the single-user servers instead, removing the need to track activity in the proxy. But this is not yet implemented in JupyterHub 0.8.0.

4.3.7 Using JupyterHub's REST API

This section will give you information on:

- what you can do with the API
- create an API token
- add API tokens to the config files
- make an API request programmatically using the requests library
- learn more about JupyterHub's API

What you can do with the 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.

Create an API token

To send requests using JupyterHub API, you must pass an API token with the request.

As of version 0.6.0, the preferred way of generating an API token is:

```
openssl rand -hex 32
```

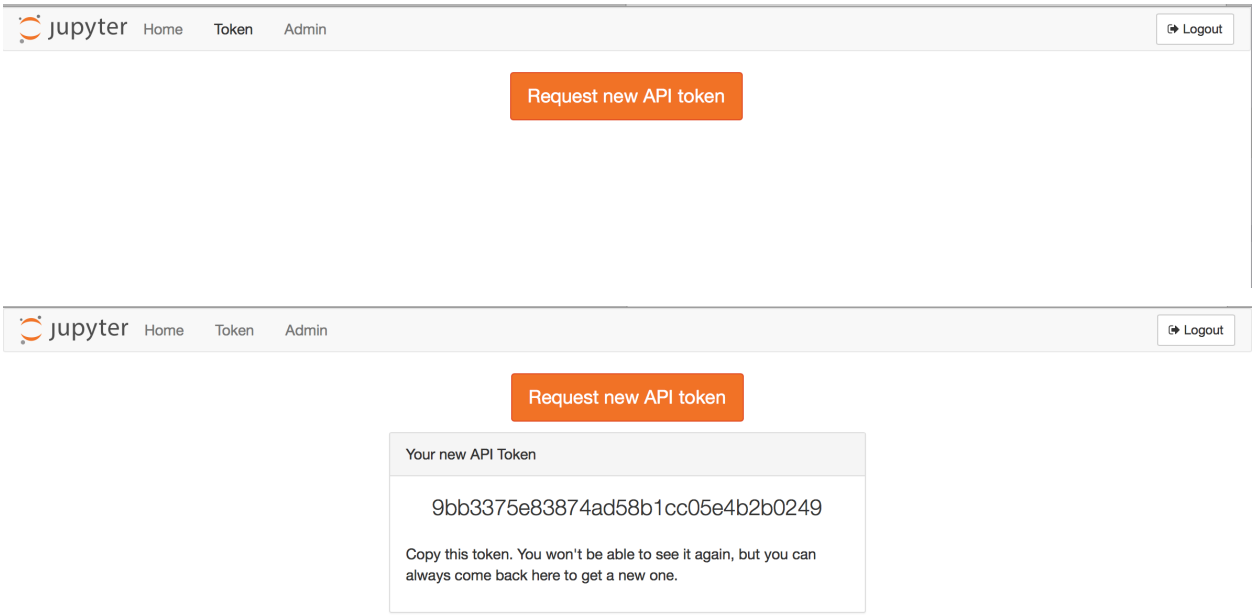
This `openssl` command generates a potential token that can then be added to JupyterHub using `.api_tokens` configuration setting in `jupyterhub_config.py`.

Alternatively, use the `jupyterhub token` command to generate a token for a specific hub user by passing the 'username':

```
jupyterhub token <username>
```

This command generates a random string to use as a token and registers it for the given user with the Hub's database.

In version 0.8.0, a TOKEN request page for generating an API token is available from the JupyterHub user interface:



Add API 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` (note that the **key** is the 'secret-token' while the **value** is the 'username'):

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

Make an API request

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

Use requests

Using the popular Python requests library, here's example code to make an API request for the users of a JupyterHub deployment. An API GET request is made, and the request sends an API token for authorization. The response contains information about the users:

```
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()
```

This example provides a slightly more complicated request, yet the process is very similar:

```
import requests

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

data = {'name': 'mygroup', 'users': ['user1', 'user2']}

r = requests.post(api_url + '/groups/formgrade-data301/users',
    headers={
               'Authorization': 'token %s' % token,
             },
    json=data
)
r.raise_for_status()
r.json()
```

The same API token can also authorize access to the Jupyter Notebook REST API provided by notebook servers managed by JupyterHub if one of the following is true:

1. The token is for the same user as the owner of the notebook
2. The token is tied to an admin user or service **and** `c.JupyterHub.admin_access` is set to `True`

Enabling users to spawn multiple named-servers via the API

With JupyterHub version 0.8, support for multiple servers per user has landed. Prior to that, each user could only launch a single default server via the API like this:

```
curl -X POST -H "Authorization: token <token>" "http://127.0.0.1:8081/hub/api/users/
↪<user>/server"
```

With the named-server functionality, it's now possible to launch more than one specifically named servers against a given user. This could be used, for instance, to launch each server based on a different image.

First you must enable named-servers by including the following setting in the `jupyterhub_config.py` file.

```
c.JupyterHub.allow_named_servers = True
```

If using the zero-to-jupyterhub-k8s set-up to run JupyterHub, then instead of editing the `jupyterhub_config.py` file directly, you could pass the following as part of the `config.yaml` file, as per the tutorial:

```
hub:
  extraConfig: |
    c.JupyterHub.allow_named_servers = True
```

With that setting in place, a new named-server is activated like this:

```
curl -X POST -H "Authorization: token <token>" "http://127.0.0.1:8081/hub/api/users/
↪<user>/servers/<serverA>"
curl -X POST -H "Authorization: token <token>" "http://127.0.0.1:8081/hub/api/users/
↪<user>/servers/<serverB>"
```

The same servers can be stopped by substituting `DELETE` for `POST` above.

Some caveats for using named-servers

The named-server capabilities are not fully implemented for JupyterHub as yet. While it's possible to start/stop a server via the API, the UI on the JupyterHub control-panel has not been implemented, and so it may not be obvious to those viewing the panel that a named-server may be running for a given user.

For named-servers via the API to work, the spawner used to spawn these servers will need to be able to handle the case of multiple servers per user and ensure uniqueness of names, particularly if servers are spawned via docker containers or kubernetes pods.

Learn more about the API

You can see the full JupyterHub REST API for details. This 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.

4.3.8 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.

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.

The upgrade process

Five 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
```

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`

4.3.9 Templates

The pages of the JupyterHub application are generated from Jinja templates. These allow the header, for example, to be defined once and incorporated into all pages. By providing your own templates, you can have complete control over JupyterHub's appearance.

Custom Templates

JupyterHub will look for custom templates in all of the paths in the `JupyterHub.template_paths` configuration option, falling back on the default templates if no custom template with that name is found. (This fallback behavior is new in version 0.9; previous versions searched only those paths explicitly included in `template_paths`.) This means you can override as many or as few templates as you desire.

Extending Templates

Jinja provides a mechanism to extend templates. A base template can define a `block`, and child templates can replace or supplement the material in the block. The JupyterHub templates make extensive use of this feature, which allows you to customize parts of the interface easily.

In general, a child template can extend a base template, `base.html`, by beginning with

```
{% extends "base.html" %}
```

This works, unless you are trying to extend the default template for the same file name. Starting in version 0.9, you may refer to the base file with a `templates/` prefix. Thus, if you are writing a custom `base.html`, start it with

```
{% extends "templates/base.html" %}
```

By defining `blocks` with same name as in the base template, child templates can replace those sections with custom content. The content from the base template can be included with the `{{ super() }}` directive.

Example

To add an additional message to the spawn-pending page, below the existing text about the server starting up, place this content in a file named `spawn_pending.html` in a directory included in the `JupyterHub.template_paths` configuration option.

```
{% extends "templates/spawn_pending.html" %}

{% block message %}
{{ super() }}
<p>Patience is a virtue.</p>
{% endblock %}
```

4.3.10 Configuration examples

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

- Using GitHub OAuth
- Using nginx reverse proxy

Using GitHub OAuth

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

- Running JupyterHub on a single cloud server
- Using SSL on the standard HTTPS port 443
- Using GitHub OAuth (using oauthenticator) for login
- Users exist locally on the server
- 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`.

The `jupyterhub_config.py` file would have these settings:

```
# jupyterhub_config.py file
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)

# Allows multiple single-server per user
c.JupyterHub.allow_named_servers = True

# 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

# 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/jupyter/jupyter_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 environment variable to be set prior to launching JupyterHub:

```
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
# append log output to log file /var/log/jupyterhub.log
jupyterhub -f /etc/jupyterhub/jupyterhub_config.py &>> /var/log/jupyterhub.log
```

Using a 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 in this example is to satisfy the following:

- 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` or `apache` is used as the public access point (which means that only nginx/apache will bind to `443`)
- After testing, the server in question should be able to score at least an A on the Qualys SSL Labs SSL Server Test

Let's start out with needed JupyterHub configuration in `jupyterhub_config.py`:

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

For high-quality SSL configuration, we also generate Diffie-Helman parameters. This can take a few minutes:

```
openssl dhparam -out /etc/ssl/certs/dhparam.pem 4096
```

nginx

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

```
# top-level http config for websocket headers
# If Upgrade is defined, Connection = upgrade
# If Upgrade is empty, Connection = close
map $http_upgrade $connection_upgrade {
    default upgrade;
    ''      close;
}

# 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';
```

(continues on next page)

(continued from previous page)

```
    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 http://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 headers
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $connection_upgrade;
    }

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

If `nginx` is not running on port 443, substitute `$http_host` for `$host` on the lines setting the `Host` header.

`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
    # SSL cert may differ

    # 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;
```

(continues on next page)

(continued from previous page)

```
    }

}
```

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

Apache

As with nginx above, you can use Apache as the reverse proxy. First, we will need to enable the apache modules that we are going to need:

```
a2enmod ssl rewrite proxy proxy_http proxy_wstunnel
```

Our Apache configuration is equivalent to the nginx configuration above:

- Redirect HTTP to HTTPS
- Good SSL Configuration
- Support for websockets on any proxied URL
- JupyterHub is running locally at http://127.0.0.1:8000

```
# redirect HTTP to HTTPS
Listen 80
<VirtualHost HUB.DOMAIN.TLD:80>
  ServerName HUB.DOMAIN.TLD
  Redirect / https://HUB.DOMAIN.TLD/
</VirtualHost>

Listen 443
<VirtualHost HUB.DOMAIN.TLD:443>

  ServerName HUB.DOMAIN.TLD

  # configure SSL
  SSLEngine on
  SSLCertificateFile /etc/letsencrypt/live/HUB.DOMAIN.TLD/fullchain.pem
  SSLCertificateKeyFile /etc/letsencrypt/live/HUB.DOMAIN.TLD/privkey.pem
  SSLProtocol All -SSLv2 -SSLv3
  SSLOpenSSLConfCmd DHParameters /etc/ssl/certs/dhparam.pem
  SSLCipherSuite EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH

  # Use RewriteEngine to handle websocket connection upgrades
  RewriteEngine On
  RewriteCond %{HTTP:Connection} Upgrade [NC]
  RewriteCond %{HTTP:Upgrade} websocket [NC]
  RewriteRule /(.*) ws://127.0.0.1:8000/$1 [P,L]

  <Location "/">
    # preserve Host header to avoid cross-origin problems
    ProxyPreserveHost on
    # proxy to JupyterHub
    ProxyPass         http://127.0.0.1:8000/
    ProxyPassReverse  http://127.0.0.1:8000/
```

(continues on next page)

(continued from previous page)

```
  </Location>
</VirtualHost>
```

4.4 The JupyterHub API

Release 0.9.0.dev

Date Mar 07, 2018

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

- what you can do with the API
- creating an API token
- adding API tokens to the config files
- making an API request programmatically using the requests library
- learning more about JupyterHub's API

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:

4.4.1 Application configuration

Module: `jupyterhub.app`

`JupyterHub`

4.4.2 Authenticators

Module: `jupyterhub.auth`

`Authenticator`

`LocalAuthenticator`

`PAMAuthenticator`

4.4.3 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.

config c.Spawner.args = List()

Extra arguments to be passed to the single-user server.

Some spawners allow shell-style expansion here, allowing you to use environment variables here. Most, including the default, do not. Consult the documentation for your spawner to verify!

config c.Spawner.cmd = Command()

The command used for starting the single-user server.

Provide either a string or a list containing the path to the startup script command. Extra arguments, other than this path, should be provided via `args`.

This is usually set if you want to start the single-user server in a different python environment (with virtualenv/conda) than JupyterHub itself.

Some spawners allow shell-style expansion here, allowing you to use environment variables. Most, including the default, do not. Consult the documentation for your spawner to verify!

config c.Spawner.cpu_guarantee = Float(None)

Minimum number of cpu-cores a single-user notebook server is guaranteed to have available.

If this value is set to 0.5, allows use of 50% of one CPU. If this value is set to 2, allows use of up to 2 CPUs.

This is a configuration setting. Your spawner must implement support for the limit to work. The default spawner, `LocalProcessSpawner`, does **not** implement this support. A custom spawner **must** add support for this setting for it to be enforced.

config c.Spawner.cpu_limit = Float(None)

Maximum number of cpu-cores a single-user notebook server is allowed to use.

If this value is set to 0.5, allows use of 50% of one CPU. If this value is set to 2, allows use of up to 2 CPUs.

The single-user notebook server will never be scheduled by the kernel to use more cpu-cores than this. There is no guarantee that it can access this many cpu-cores.

This is a configuration setting. Your spawner must implement support for the limit to work. The default spawner, `LocalProcessSpawner`, does **not** implement this support. A custom spawner **must** add support for this setting for it to be enforced.

config c.Spawner.debug = Bool(False)

Enable debug-logging of the single-user server

config c.Spawner.default_url = Unicode('')
The URL the single-user server should start in.

`{username}` will be expanded to the user's username

Example uses:

- You can set `notebook_dir` to `/` and `default_url` to `/tree/home/{username}` to allow people to navigate the whole filesystem from their notebook server, but still start in their home directory.
- Start with `/notebooks` instead of `/tree` if `default_url` points to a notebook instead of a directory.
- You can set this to `/lab` to have JupyterLab start by default, rather than Jupyter Notebook.

config c.Spawner.disable_user_config = Bool(False)
Disable per-user configuration of single-user servers.

When starting the user's single-user server, any config file found in the user's $HOME directory will be ignored.

Note: a user could circumvent this if the user modifies their Python environment, such as when they have their own conda environments / virtualenvs / containers.

config c.Spawner.env_keep = List()
Whitelist of environment variables for the single-user server to inherit from the JupyterHub process.

This whitelist is used to ensure that sensitive information in the JupyterHub process's environment (such as `CONFIGPROXY_AUTH_TOKEN`) is not passed to the single-user server's process.

config c.Spawner.environment = Dict()
Extra environment variables to set for the single-user server's process.

Environment variables that end up in the single-user server's process come from 3 sources:

- This `environment` configurable
- The JupyterHub process' environment variables that are whitelisted in `env_keep`
- Variables to establish contact between the single-user notebook and the hub (such as JUPYTERHUB_API_TOKEN)

The `environment` configurable should be set by JupyterHub administrators to add installation specific environment variables. It is a dict where the key is the name of the environment variable, and the value can be a string or a callable. If it is a callable, it will be called with one parameter (the spawner instance), and should return a string fairly quickly (no blocking operations please!).

Note that the spawner class' interface is not guaranteed to be exactly same across upgrades, so if you are using the callable take care to verify it continues to work after upgrades!

config c.Spawner.http_timeout = Int(30)
Timeout (in seconds) before giving up on a spawned HTTP server

Once a server has successfully been spawned, this is the amount of time we wait before assuming that the server is unable to accept connections.

config c.Spawner.ip = Unicode('')
The IP address (or hostname) the single-user server should listen on.

The JupyterHub proxy implementation should be able to send packets to this interface.

config c.Spawner.mem_guarantee = ByteSpecification(None)
Minimum number of bytes a single-user notebook server is guaranteed to have available.

Allows the following suffixes:

- K -> Kilobytes
- M -> Megabytes
- G -> Gigabytes
- T -> Terabytes

This is a configuration setting. Your spawner must implement support for the limit to work. The default spawner, `LocalProcessSpawner`, does **not** implement this support. A custom spawner **must** add support for this setting for it to be enforced.

config c.Spawner.mem_limit = ByteSpecification(None)

Maximum number of bytes a single-user notebook server is allowed to use.

Allows the following suffixes:

- K -> Kilobytes
- M -> Megabytes
- G -> Gigabytes
- T -> Terabytes

If the single user server tries to allocate more memory than this, it will fail. There is no guarantee that the single-user notebook server will be able to allocate this much memory - only that it can not allocate more than this.

This is a configuration setting. Your spawner must implement support for the limit to work. The default spawner, `LocalProcessSpawner`, does **not** implement this support. A custom spawner **must** add support for this setting for it to be enforced.

config c.Spawner.notebook_dir = Unicode('')

Path to the notebook directory for the single-user server.

The user sees a file listing of this directory when the notebook interface is started. The current interface does not easily allow browsing beyond the subdirectories in this directory's tree.

~ will be expanded to the home directory of the user, and {username} will be replaced with the name of the user.

Note that this does *not* prevent users from accessing files outside of this path! They can do so with many other means.

config c.Spawner.options_form = Union()

An HTML form for options a user can specify on launching their server.

The surrounding `<form>` element and the submit button are already provided.

For example:

```
Set your key:
<input name="key" val="default_key"></input>
<br>
Choose a letter:
<select name="letter" multiple="true">
  <option value="A">The letter A</option>
  <option value="B">The letter B</option>
</select>
```

The data from this form submission will be passed on to your spawner in `self.user_options`

Instead of a form snippet string, this could also be a callable that takes as one parameter the current spawner instance and returns a string. The callable will be called asynchronously if it returns a future, rather than a str. Note that the interface of the spawner class is not deemed stable across versions, so using this functionality might cause your JupyterHub upgrades to break.

config c.Spawner.poll_interval = Int(30)
Interval (in seconds) on which to poll the spawner for single-user server's status.

At every poll interval, each spawner's `.poll` method is called, which checks if the single-user server is still running. If it isn't running, then JupyterHub modifies its own state accordingly and removes appropriate routes from the configurable proxy.

config c.Spawner.port = Int(0)
The port for single-user servers to listen on.

Defaults to `0`, which uses a randomly allocated port number each time.

If set to a non-zero value, all Spawners will use the same port, which only makes sense if each server is on a different address, e.g. in containers.

New in version 0.7.

config c.Spawner.pre_spawn_hook = Any(None)
An optional hook function that you can implement to do some bootstrapping work before the spawner starts. For example, create a directory for your user or load initial content.

This can be set independent of any concrete spawner implementation.

Example:

```
from subprocess import check_call
def my_hook(spawner):
    username = spawner.user.name
    check_call(['./examples/bootstrap-script/bootstrap.sh', username])

c.Spawner.pre_spawn_hook = my_hook
```

config c.Spawner.start_timeout = Int(60)
Timeout (in seconds) before giving up on starting of single-user server.

This is the timeout for start to return, not the timeout for the server to respond. Callers of spawner.start will assume that startup has failed if it takes longer than this. start should return when the server process is started and its location is known.

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 False (default), shutdown the server as gracefully as possible, e.g. starting with SIGINT, then SIGTERM, then SIGKILL. If `now` is True, terminate the server immediately.

The coroutine should return when the single-user server process is no longer running.

Must be a 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)

LocalProcessSpawner

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.

Note: This spawner does not implement CPU / memory guarantees and limits.

config c.LocalProcessSpawner.args = List()
Extra arguments to be passed to the single-user server.

Some spawners allow shell-style expansion here, allowing you to use environment variables here. Most, including the default, do not. Consult the documentation for your spawner to verify!

config c.LocalProcessSpawner.cmd = Command()
The command used for starting the single-user server.

Provide either a string or a list containing the path to the startup script command. Extra arguments, other than this path, should be provided via `args`.

This is usually set if you want to start the single-user server in a different python environment (with virtualenv/conda) than JupyterHub itself.

Some spawners allow shell-style expansion here, allowing you to use environment variables. Most, including the default, do not. Consult the documentation for your spawner to verify!

config c.LocalProcessSpawner.cpu_guarantee = Float(None)
Minimum number of cpu-cores a single-user notebook server is guaranteed to have available.

If this value is set to 0.5, allows use of 50% of one CPU. If this value is set to 2, allows use of up to 2 CPUs.

This is a configuration setting. Your spawner must implement support for the limit to work. The default spawner, `LocalProcessSpawner`, does **not** implement this support. A custom spawner **must** add support for this setting for it to be enforced.

config c.LocalProcessSpawner.cpu_limit = Float(None)
Maximum number of cpu-cores a single-user notebook server is allowed to use.

If this value is set to 0.5, allows use of 50% of one CPU. If this value is set to 2, allows use of up to 2 CPUs.

The single-user notebook server will never be scheduled by the kernel to use more cpu-cores than this. There is no guarantee that it can access this many cpu-cores.

This is a configuration setting. Your spawner must implement support for the limit to work. The default spawner, `LocalProcessSpawner`, does **not** implement this support. A custom spawner **must** add support for this setting for it to be enforced.

config c.LocalProcessSpawner.debug = Bool(False)
Enable debug-logging of the single-user server

config c.LocalProcessSpawner.default_url = Unicode('')
The URL the single-user server should start in.

`{username}` will be expanded to the user's username

Example uses:

- You can set `notebook_dir` to `/` and `default_url` to `/tree/home/{username}` to allow people to navigate the whole filesystem from their notebook server, but still start in their home directory.
- Start with `/notebooks` instead of `/tree` if `default_url` points to a notebook instead of a directory.
- You can set this to `/lab` to have JupyterLab start by default, rather than Jupyter Notebook.

config c.LocalProcessSpawner.disable_user_config = Bool(False)
Disable per-user configuration of single-user servers.

When starting the user's single-user server, any config file found in the user's $HOME directory will be ignored.

Note: a user could circumvent this if the user modifies their Python environment, such as when they have their own conda environments / virtualenvs / containers.

config c.LocalProcessSpawner.env_keep = List()
Whitelist of environment variables for the single-user server to inherit from the JupyterHub process.

This whitelist is used to ensure that sensitive information in the JupyterHub process's environment (such as `CONFIGPROXY_AUTH_TOKEN`) is not passed to the single-user server's process.

config c.LocalProcessSpawner.environment = Dict()
Extra environment variables to set for the single-user server's process.

Environment variables that end up in the single-user server's process come from 3 sources:

- This `environment` configurable
- The JupyterHub process' environment variables that are whitelisted in `env_keep`
- Variables to establish contact between the single-user notebook and the hub (such as JUPYTERHUB_API_TOKEN)

The `environment` configurable should be set by JupyterHub administrators to add installation specific environment variables. It is a dict where the key is the name of the environment variable, and the value can be a string or a callable. If it is a callable, it will be called with one parameter (the spawner instance), and should return a string fairly quickly (no blocking operations please!).

Note that the spawner class' interface is not guaranteed to be exactly same across upgrades, so if you are using the callable take care to verify it continues to work after upgrades!

config c.LocalProcessSpawner.http_timeout = Int(30)
Timeout (in seconds) before giving up on a spawned HTTP server

Once a server has successfully been spawned, this is the amount of time we wait before assuming that the server is unable to accept connections.

config c.LocalProcessSpawner.interrupt_timeout = Int(10)
Seconds to wait for single-user server process to halt after SIGINT.

If the process has not exited cleanly after this many seconds, a SIGTERM is sent.

config c.LocalProcessSpawner.ip = Unicode('')
The IP address (or hostname) the single-user server should listen on.

The JupyterHub proxy implementation should be able to send packets to this interface.

config c.LocalProcessSpawner.kill_timeout = Int(5)
Seconds to wait for process to halt after SIGKILL before giving up.

If the process does not exit cleanly after this many seconds of SIGKILL, it becomes a zombie process. The hub process will log a warning and then give up.

config c.LocalProcessSpawner.mem_guarantee = ByteSpecification(None)
Minimum number of bytes a single-user notebook server is guaranteed to have available.

Allows the following suffixes:

- K -> Kilobytes
- M -> Megabytes
- G -> Gigabytes
- T -> Terabytes

This is a configuration setting. Your spawner must implement support for the limit to work. The default spawner, `LocalProcessSpawner`, does **not** implement this support. A custom spawner **must** add support for this setting for it to be enforced.

config c.LocalProcessSpawner.mem_limit = ByteSpecification(None)
Maximum number of bytes a single-user notebook server is allowed to use.

Allows the following suffixes:

- K -> Kilobytes
- M -> Megabytes
- G -> Gigabytes
- T -> Terabytes

If the single user server tries to allocate more memory than this, it will fail. There is no guarantee that the single-user notebook server will be able to allocate this much memory - only that it can not allocate more than this.

This is a configuration setting. Your spawner must implement support for the limit to work. The default spawner, `LocalProcessSpawner`, does **not** implement this support. A custom spawner **must** add support for this setting for it to be enforced.

config c.LocalProcessSpawner.notebook_dir = Unicode('')
Path to the notebook directory for the single-user server.

The user sees a file listing of this directory when the notebook interface is started. The current interface does not easily allow browsing beyond the subdirectories in this directory's tree.

~ will be expanded to the home directory of the user, and {username} will be replaced with the name of the user.

Note that this does *not* prevent users from accessing files outside of this path! They can do so with many other means.

config c.LocalProcessSpawner.options_form = Union()

An HTML form for options a user can specify on launching their server.

The surrounding `<form>` element and the submit button are already provided.

For example:

```
Set your key:
<input name="key" val="default_key"></input>
<br>
Choose a letter:
<select name="letter" multiple="true">
  <option value="A">The letter A</option>
  <option value="B">The letter B</option>
</select>
```

The data from this form submission will be passed on to your spawner in `self.user_options`

Instead of a form snippet string, this could also be a callable that takes as one parameter the current spawner instance and returns a string. The callable will be called asynchronously if it returns a future, rather than a str. Note that the interface of the spawner class is not deemed stable across versions, so using this functionality might cause your JupyterHub upgrades to break.

config c.LocalProcessSpawner.poll_interval = Int(30)

Interval (in seconds) on which to poll the spawner for single-user server's status.

At every poll interval, each spawner's `.poll` method is called, which checks if the single-user server is still running. If it isn't running, then JupyterHub modifies its own state accordingly and removes appropriate routes from the configurable proxy.

config c.LocalProcessSpawner.popen_kwargs = Dict()

Extra keyword arguments to pass to Popen

when spawning single-user servers.

For example:

```
popen_kwargs = dict(shell=True)
```

config c.LocalProcessSpawner.port = Int(0)

The port for single-user servers to listen on.

Defaults to `0`, which uses a randomly allocated port number each time.

If set to a non-zero value, all Spawners will use the same port, which only makes sense if each server is on a different address, e.g. in containers.

New in version 0.7.

config c.LocalProcessSpawner.pre_spawn_hook = Any(None)

An optional hook function that you can implement to do some bootstrapping work before the spawner starts. For example, create a directory for your user or load initial content.

This can be set independent of any concrete spawner implementation.

Example:

```
from subprocess import check_call
def my_hook(spawner):
    username = spawner.user.name
    check_call(['./examples/bootstrap-script/bootstrap.sh', username])

c.Spawner.pre_spawn_hook = my_hook
```

config c.LocalProcessSpawner.start_timeout = Int(60)
Timeout (in seconds) before giving up on starting of single-user server.

This is the timeout for start to return, not the timeout for the server to respond. Callers of spawner.start will assume that startup has failed if it takes longer than this. start should return when the server process is started and its location is known.

config c.LocalProcessSpawner.term_timeout = Int(5)
Seconds to wait for single-user server process to halt after SIGTERM.

If the process does not exit cleanly after this many seconds of SIGTERM, a SIGKILL is sent.

4.4.4 Proxies

Module: jupyterhub.proxy

API for JupyterHub's proxy.

Custom proxy implementations can subclass *Proxy* and register in JupyterHub config:

```
from mymodule import MyProxy
c.JupyterHub.proxy_class = MyProxy
```

Route Specification:

- A routespec is a URL prefix ([host]/path/), e.g. 'host.tld/path/' for host-based routing or '/path/' for default routing.
- Route paths should be normalized to always start and end with '/'

Proxy

class jupyterhub.proxy.**Proxy**(***kwargs*)
Base class for configurable proxies that JupyterHub can use.

A proxy implementation should subclass this and must define the following methods:

- *get_all_routes()* return a dictionary of all JupyterHub-related routes
- *add_route()* adds a route
- *delete_route()* deletes a route

In addition to these, the following method(s) may need to be implemented:

- *start()* start the proxy, if it should be launched by the Hub instead of externally managed. If the proxy is externally managed, it should set `should_start` to False.
- *stop()* stop the proxy. Only used if *start()* is also used.

And the following method(s) are optional, but can be provided:

- *get_route()* gets a single route. There is a default implementation that extracts data from *get_all_routes()*, but implementations may choose to provide a more efficient implementation of fetching a single route.

config c.Proxy.should_start = Bool(True)
Should the Hub start the proxy

If True, the Hub will start the proxy and stop it. Set to False if the proxy is managed externally, such as by systemd, docker, or another service manager.

add_all_services(*service_dict*)
Update the proxy table from the database.

Used when loading up a new proxy.

add_all_users(*user_dict*)
Update the proxy table from the database.

Used when loading up a new proxy.

add_hub_route(*hub*)
Add the default route for the Hub

add_route(*routespec*, *target*, *data*)
Add a route to the proxy.

Subclasses must define this method

Parameters

- **routespec** (*str*) – A URL prefix ([host]/path/) for which this route will be matched, e.g. host.name/path/
- **target** (*str*) – A full URL that will be the target of this route.
- **data** (*dict*) – A JSONable dict that will be associated with this route, and will be returned when retrieving information about this route.

Will raise an appropriate Exception (FIXME: find what?) if the route could not be added.

The proxy implementation should also have a way to associate the fact that a route came from JupyterHub.

add_service(*service*, *client=None*)
Add a service's server to the proxy table.

add_user(*user*, *server_name=''*, *client=None*)
Add a user's server to the proxy table.

check_routes(*user_dict*, *service_dict*, *routes=None*)
Check that all users are properly routed on the proxy.

delete_route(*routespec*)
Delete a route with a given routespec if it exists.

Subclasses must define this method

delete_service(*service*, *client=None*)
Remove a service's server from the proxy table.

delete_user(*user*, *server_name=''*)
Remove a user's server from the proxy table.

get_all_routes()
Fetch and return all the routes associated by JupyterHub from the proxy.

Subclasses must define this method

Should return a dictionary of routes, where the keys are routespecs and each value is a dict of the form:

```
{
  'routespec': the route specification ([host]/path/)
  'target': the target host URL (proto://host) for this route
  'data': the attached data dict for this route (as specified in add_route)
}
```

get_route(*routespec*)

Return the route info for a given routespec.

Parameters **routespec** (*str*) – A URI that was used to add this route, e.g. `host.tld/path/`

Returns

dict with the following keys:

```
'routespec': The normalized route specification passed in to add_
↪route
    ([host]/path/)
'target': The target host for this route (proto://host)
'data': The arbitrary data dict that was passed in by JupyterHub␣
↪when adding this
        route.
```

None: if there are no routes matching the given routespec

Return type result (dict)

config c.Proxy.should_start = Bool(True)

Should the Hub start the proxy

If True, the Hub will start the proxy and stop it. Set to False if the proxy is managed externally, such as by systemd, docker, or another service manager.

start()

Start the proxy.

Will be called during startup if should_start is True.

Subclasses must define this method if the proxy is to be started by the Hub

stop()

Stop the proxy.

Will be called during teardown if should_start is True.

Subclasses must define this method if the proxy is to be started by the Hub

validate_routespec(*routespec*)

Validate a routespec

- Checks host value vs host-based routing.
- Ensures trailing slash on path.

ConfigurableHTTPProxy

class jupyterhub.proxy.**ConfigurableHTTPProxy**(***kwargs*)

Proxy implementation for the default configurable-http-proxy.

This is the default proxy implementation for running the nodejs proxy `configurable-http-proxy`.

If the proxy should not be run as a subprocess of the Hub, (e.g. in a separate container), set:

```
c.ConfigurableHTTPProxy.should_start = False
```

config c.ConfigurableHTTPProxy.api_url = Unicode('http://127.0.0.1:8001')
The ip (or hostname) of the proxy's API endpoint

config c.ConfigurableHTTPProxy.auth_token = Unicode('')
The Proxy auth token

Loaded from the CONFIGPROXY_AUTH_TOKEN env variable by default.

config c.ConfigurableHTTPProxy.command = Command()
The command to start the proxy

config c.ConfigurableHTTPProxy.debug = Bool(False)
Add debug-level logging to the Proxy.

config c.ConfigurableHTTPProxy.should_start = Bool(True)
Should the Hub start the proxy

If True, the Hub will start the proxy and stop it. Set to False if the proxy is managed externally, such as by systemd, docker, or another service manager.

config c.ConfigurableHTTPProxy.api_url = Unicode('http://127.0.0.1:8001')
The ip (or hostname) of the proxy's API endpoint

config c.ConfigurableHTTPProxy.auth_token = Unicode('')
The Proxy auth token

Loaded from the CONFIGPROXY_AUTH_TOKEN env variable by default.

config c.ConfigurableHTTPProxy.command = Command()
The command to start the proxy

config c.ConfigurableHTTPProxy.debug = Bool(False)
Add debug-level logging to the Proxy.

4.4.5 Users

Module: jupyterhub.user

UserDict

class jupyterhub.user.**UserDict**(*db_factory*, *settings*)
Like defaultdict, but for users

Getting by a user id OR an orm.User instance returns a User wrapper around the orm user.

add(*orm_user*)
Add a user to the UserDict

count_active_users()
Count the number of user servers that are active/pending/ready

Returns dict with counts of active/pending/ready servers

delete(*key*)
Delete a user from the cache and the database

User

class jupyterhub.user.**User**(*orm_user*, *settings=None*, *db=None*)

High-level wrapper around an orm.User object

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.

4.4.6 Services

Module: jupyterhub.services.service

A service is a process that talks to JupyterHub.

Types of services:

Managed:

- managed by JupyterHub (always subprocess, no custom Spawners)
- always a long-running process
- managed services are restarted automatically if they exit unexpectedly

Unmanaged:

- managed by external service (docker, systemd, etc.)
- do not need to be long-running processes, or processes at all

URL: needs a route added to the proxy.

- Public route will always be /services/service-name
- url specified in config
- if port is 0, Hub will select a port

API access:

- admin: tokens will have admin-access to the API
- not admin: tokens will only have non-admin access (not much they can do other than defer to Hub for auth)

An externally managed service running on a URL:

```
{
    'name': 'my-service',
    'url': 'https://host:8888',
    'admin': True,
    'api_token': 'super-secret',
}
```

A hub-managed service with no URL:

```
{
    'name': 'cull-idle',
    'command': ['python', '/path/to/cull-idle']
    'admin': True,
}
```

Service

class jupyterhub.services.service.**Service**(***kwargs*)
An object wrapping a service specification for Hub API consumers.

A service has inputs:

- **name: str** the name of the service
- **admin: bool(false)** whether the service should have administrative privileges
- **url: str (None)** The URL where the service is/should be. If specified, the service will be added to the proxy at /services/:name

If a service is 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, it is assumed to be managed by a
- **environment: dict** Additional environment variables for the service.
- **user: str** The name of a system user to become. If unspecified, run as the same user as the Hub.

kind
The name of the kind of service as a string

- 'managed' for managed services
- 'external' for external services

managed
Am I managed by the Hub?

4.4.7 Services Authentication

Module: jupyterhub.services.auth

Authenticating services with JupyterHub.

Cookies are sent to the Hub for verification. The Hub replies 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.

HubAuth

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).

config c.HubAuth.api_token = Unicode('')
API key for accessing Hub API.

Generate with jupyterhub token [username] or add to JupyterHub.services config.

config c.HubAuth.api_url = Unicode('http://127.0.0.1:8081/hub/api')
The base API URL of the Hub.

Typically http://hub-ip:hub-port/hub/api

config c.HubAuth.base_url = Unicode('/')
The base URL prefix of this application

e.g. /services/service-name/ or /user/name/

Default: get from JUPYTERHUB_SERVICE_PREFIX

config c.HubAuth.cache_max_age = Int(300)
The maximum time (in seconds) to cache the Hub's responses for authentication.

A larger value reduces load on the Hub and occasional response lag. A smaller value reduces propagation time of changes on the Hub (rare).

Default: 300 (five minutes)

config c.HubAuth.cookie_name = Unicode('jupyterhub-services')
The name of the cookie I should be looking for

config c.HubAuth.hub_host = Unicode('')
The public host of JupyterHub

Only used if JupyterHub is spreading servers across subdomains.

config c.HubAuth.hub_prefix = Unicode('/hub/')
The URL prefix for the Hub itself.

Typically /hub/

config c.HubAuth.login_url = Unicode('/hub/login')
The login URL to use

Typically /hub/login

config c.HubAuth.api_token = Unicode('')
API key for accessing Hub API.

Generate with jupyterhub token [username] or add to JupyterHub.services config.

config c.HubAuth.api_url = Unicode('http://127.0.0.1:8081/hub/api')
The base API URL of the Hub.

Typically `http://hub-ip:hub-port/hub/api`

config c.HubAuth.base_url = Unicode('/')
The base URL prefix of this application

e.g. /services/service-name/ or /user/name/

Default: get from JUPYTERHUB_SERVICE_PREFIX

config c.HubAuth.cache_max_age = Int(300)
The maximum time (in seconds) to cache the Hub's responses for authentication.

A larger value reduces load on the Hub and occasional response lag. A smaller value reduces propagation time of changes on the Hub (rare).

Default: 300 (five minutes)

config c.HubAuth.cookie_name = Unicode('jupyterhub-services')
The name of the cookie I should be looking for

get_session_id(*handler*)
Get the jupyterhub session id

from the jupyterhub-session-id cookie.

get_token(*handler*)
Get the user token from a request

- in URL parameters: ?token=<token>
- in header: Authorization: token <token>

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)

config c.HubAuth.hub_host = Unicode('')
The public host of JupyterHub

Only used if JupyterHub is spreading servers across subdomains.

config c.HubAuth.hub_prefix = Unicode('/hub/')
The URL prefix for the Hub itself.

Typically /hub/

config c.HubAuth.login_url = Unicode('/hub/login')
The login URL to use

Typically /hub/login

user_for_cookie(*encrypted_cookie*, *use_cache=True*, *session_id=''*)
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)

user_for_token(*token*, *use_cache=True*, *session_id=''*)

Ask the Hub to identify the user for a given token.

Parameters

- **token** (*str*) – the token
- **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)

HubOAuth

class jupyterhub.services.auth.**HubOAuth**(***kwargs*)

HubAuth using OAuth for login instead of cookies set by the Hub.

config c.HubOAuth.api_token = Unicode('')

API key for accessing Hub API.

Generate with `jupyterhub token [username]` or add to JupyterHub.services config.

config c.HubOAuth.api_url = Unicode('http://127.0.0.1:8081/hub/api')

The base API URL of the Hub.

Typically `http://hub-ip:hub-port/hub/api`

config c.HubOAuth.base_url = Unicode('/')

The base URL prefix of this application

e.g. /services/service-name/ or /user/name/

Default: get from JUPYTERHUB_SERVICE_PREFIX

config c.HubOAuth.cache_max_age = Int(300)

The maximum time (in seconds) to cache the Hub's responses for authentication.

A larger value reduces load on the Hub and occasional response lag. A smaller value reduces propagation time of changes on the Hub (rare).

Default: 300 (five minutes)

config c.HubOAuth.hub_host = Unicode('')
The public host of JupyterHub

Only used if JupyterHub is spreading servers across subdomains.

config c.HubOAuth.hub_prefix = Unicode('/hub/')
The URL prefix for the Hub itself.

Typically /hub/

config c.HubOAuth.login_url = Unicode('/hub/login')
The login URL to use

Typically /hub/login

config c.HubOAuth.oauth_authorization_url = Unicode('/hub/api/oauth2/authorize')
The URL to redirect to when starting the OAuth process

config c.HubOAuth.oauth_client_id = Unicode('')
The OAuth client ID for this application.

Use JUPYTERHUB_CLIENT_ID by default.

config c.HubOAuth.oauth_redirect_uri = Unicode('')
OAuth redirect URI

Should generally be /base_url/oauth_callback

config c.HubOAuth.oauth_token_url = Unicode('')
The URL for requesting an OAuth token from JupyterHub

clear_cookie(*handler*)
Clear the OAuth cookie

cookie_name
Use OAuth client_id for cookie name

because we don't want to use the same cookie name across OAuth clients.

generate_state(*next_url=None*, ***extra_state*)
Generate a state string, given a next_url redirect target

Parameters (str) (*next_url*) –

Returns state (str)

Return type The base64-encoded state string.

get_next_url(*b64_state=''*)
Get the next_url for redirection, given an encoded OAuth state

get_state_cookie_name(*b64_state=''*)
Get the cookie name for oauth state, given an encoded OAuth state

Cookie name is stored in the state itself because the cookie name is randomized to deal with races between concurrent oauth sequences.

config c.HubOAuth.oauth_authorization_url = Unicode('/hub/api/oauth2/authorize')
The URL to redirect to when starting the OAuth process

config c.HubOAuth.oauth_client_id = Unicode('')
The OAuth client ID for this application.

Use JUPYTERHUB_CLIENT_ID by default.

config c.HubOAuth.oauth_redirect_uri = Unicode('')
OAuth redirect URI

Should generally be /base_url/oauth_callback

config c.HubOAuth.oauth_token_url = Unicode('')
The URL for requesting an OAuth token from JupyterHub

set_cookie(*handler*, *access_token*)
Set a cookie recording OAuth result

set_state_cookie(*handler*, *next_url=None*)
Generate an OAuth state and store it in a cookie

Parameters

- **(RequestHandler)** (*handler*) –
- **(str)** (*next_url*) –

Returns state (str)

Return type The OAuth state that has been stored in the cookie (url safe, base64-encoded)

state_cookie_name
The cookie name for storing OAuth state

This cookie is only live for the duration of the OAuth handshake.

token_for_code(*code*)
Get token for OAuth temporary code

This is the last step of OAuth login. Should be called in OAuth Callback handler.

Parameters **code** (*str*) – oauth code for finishing OAuth login

Returns JupyterHub API Token

Return type token (str)

HubAuthenticated

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):
        ...
```

allow_all
Property indicating that all successfully identified user or service should be allowed.

check_hub_user(*model*)
Check whether Hub-authenticated user or service 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 **model** (*dict*) – the user or service 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)

get_login_url()
Return the Hub's login URL

hub_auth_class
alias of *HubAuth*

HubOAuthenticated

class jupyterhub.services.auth.**HubOAuthenticated**
Simple subclass of HubAuthenticated using OAuth instead of old shared cookies

HubOAuthCallbackHandler

class jupyterhub.services.auth.**HubOAuthCallbackHandler**(*application*, *request*, ***kwargs*)
OAuth Callback handler

Finishes the OAuth flow, setting a cookie to record the user's info.

Should be registered at SERVICE_PREFIX/oauth_callback

4.5 Tutorials

This section provides links to documentation that helps a user do a specific task.

- *Upgrading to JupyterHub version 0.8*
- Zero to JupyterHub with Kubernetes

4.5.1 Upgrading to JupyterHub version 0.8

This document will assist you in upgrading an existing JupyterHub deployment from version 0.7 to version 0.8.

Upgrade checklist

0. Review the release notes. Review any deprecated features and pay attention to any backwards incompatible changes
1. **Backup JupyterHub database:**
    - `jupyterhub.sqlite` when using the default sqlite database
    - Your JupyterHub database when using an RDBMS
2. Backup the existing 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 ``
6. Update the JupyterHub configuration file `jupyterhub_config.py`

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.

Note: 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

Backup 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.
- Shutdown the JupyterHub service.

Upgrade JupyterHub

Follow directions that correspond to your package manager, `pip` or `conda`, for the new JupyterHub release:

- `pip install -U jupyterhub` for `pip`
- `conda upgrade jupyterhub` for `conda`

Upgrade the proxy, authenticator, or spawner if needed.

Upgrade JupyterHub database

To run the upgrade process for JupyterHub databases, enter:

```
jupyterhub upgrade-db
```

Update the JupyterHub configuration file

Create a new JupyterHub configuration file or edit a copy of the existing file `jupyterhub_config.py`.

Start JupyterHub

Start JupyterHub with the same command that you used before the upgrade.

4.6 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
- What is the default behavior when none of the lists (admin, whitelist, group whitelist) are set?

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)?
- How do I set up rotating daily logs?
- Toree integration with HDFS rack awareness script

- Where do I find Docker images and Dockerfiles related to JupyterHub?

Troubleshooting commands

4.6.1 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`

Note: If this occurs on Ubuntu/Debian, check that the you are using a recent version of node. Some versions of Ubuntu/Debian come with a version of node that is very old, and it is necessary to update node.

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`.

What is the default behavior when none of the lists (admin, whitelist, group whitelist) are set?

When nothing is given for these lists, there will be no admins, and all users who can authenticate on the system (i.e. all the unix users on the server with a password) will be allowed to start a server. The whitelist lets you limit this to a particular set of users, and the admin_users lets you specify who among them may use the admin interface (not necessary, unless you need to do things like inspect other users' servers, or modify the userlist at runtime).

4.6.2 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/jupyterhub-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/jupyterhub-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.

4.6.3 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.

How do I set up rotating daily logs?

You can do this with logrotate, or pipe to `logger` to use syslog instead of directly to a file.

For example, with this logrotate config file:

```
/var/log/jupyterhub.log {
  copytruncate
  daily
}
```

and run this daily by putting a script in `/etc/cron.daily/`:

```
logrotate /path/to/above-config
```

Or use syslog:

```
jupyterhub | logger -t jupyterhub
```

4.6.4 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
```

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).

Where do I find Docker images and Dockerfiles related to JupyterHub?

Docker images can be found at the JupyterHub organization on DockerHub. The Docker image jupyterhub/singleuser provides an example single user notebook server for use with DockerSpawner.

Additional single user notebook server images can be found at the Jupyter organization on DockerHub and information about each image at the jupyter/docker-stacks repo.

4.7 Contributors

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

- Analect
- anderbubble
- apetresc
- barrachri
- betatim
- Carreau
- charnpreetsingh
- ckald
- CRegenschein
- cwaldbieser
- danielballen
- danoventa
- daradib
- datapolitan
- dblockow-d2dcrc
- DeepHorizons
- dhirschfeld
- dietmarw
- dmartzol
- DominicFollettSmith
- dsblank
- ellisonbg
- evanlinde
- Fokko
- fperez
- iamed18
- JamiesHQ
- jbweston
- jdavidheiser
- jencabral
- jhamrick
- josephtate
- kinuax
- KrishnaPG
- kroq-gar78
- ksolan

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

- zonca

4.8 A Gallery of JupyterHub Deployments

A JupyterHub Community Resource

We've compiled this list of JupyterHub deployments to help the community see the breadth and growth of JupyterHub's use in education, research, and high performance computing.

Please submit pull requests to update information or to add new institutions or uses.

4.8.1 Academic Institutions, Research Labs, and Supercomputer Centers

University of California Berkeley

- BIDS - Berkeley Institute for Data Science
  - Teaching with Jupyter notebooks and JupyterHub
- Data 8
  - GitHub organization
- NERSC
  - Press release on Jupyter and Cori
  - Moving and sharing data
- Research IT
  - JupyterHub server supports campus research computation

University of California Davis

- Spinning up multiple Jupyter Notebooks on AWS for a tutorial

Although not technically a JupyterHub deployment, this tutorial setup may be helpful to others in the Jupyter community.

Thank you C. Titus Brown for sharing this with the Software Carpentry mailing list.

```
* I started a big Amazon machine;
* I installed Docker and built a custom image containing my software of
  interest;
* I ran multiple containers, one connected to port 8000, one on 8001,
  etc. and gave each student a different port;
* students could connect in and use the Terminal program in Jupyter to
  execute commands, and could upload/download files via the Jupyter
  console interface;
* in theory I could have used notebooks too, but for this I didn't have
  need.

I am aware that JupyterHub can probably do all of this including manage
the containers, but I'm still a bit shy of diving into that; this was
fairly straightforward, gave me disposable containers that were isolated
for each individual student, and worked almost flawlessly.  Should be
easy to do with RStudio too.
```

Cal Poly San Luis Obispo

- jupyterhub-deploy-teaching based on work by Brian Granger for Cal Poly's Data Science 301 Course

Clemson University

- Advanced Computing
  - Palmetto cluster and JupyterHub

University of Colorado Boulder

- (CU Research Computing) CURC
  - JupyterHub User Guide
    - Slurm job dispatched on Crestone compute cluster
    - log troubleshooting
    - Profiles in IPython Clusters tab
  - Parallel Processing with JupyterHub tutorial
  - Parallel Programming with JupyterHub document
- Earth Lab at CU
  - Tutorial on Parallel R on JupyterHub

HTCondor

- HTCondor Python Bindings Tutorial from HTCondor Week 2017 includes information on their JupyterHub tutorials

University of Illinois

- https://datascience.business.illinois.edu

MIT and Lincoln Labs

Michigan State University

- Setting up JupyterHub

University of Minnesota

- JupyterHub Inside HPC

University of Missouri

- https://dsa.missouri.edu/faq/

University of Rochester CIRC

- JupyterHub Userguide - Slurm, beehive

University of California San Diego

- San Diego Supercomputer Center - Andrea Zonca
  - Deploy JupyterHub on a Supercomputer with SSH
  - Run Jupyterhub on a Supercomputer
  - Deploy JupyterHub on a VM for a Workshop
  - Customize your Python environment in Jupyterhub
  - Jupyterhub deployment on multiple nodes with Docker Swarm
  - Sample deployment of Jupyterhub in HPC on SDSC Comet
- Educational Technology Services - Paul Jamason
  - jupyterhub.ucsd.edu

TACC University of Texas

Texas A&M

- Kristen Thyng - Oceanography
  - Teaching with JupyterHub and nbgrader

4.8.2 Service Providers

AWS

- running-jupyter-notebook-and-jupyterhub-on-amazon-emr

Google Cloud Platform

- Using Tensorflow and JupyterHub in Classrooms
- using-tensorflow-and-jupyterhub blog post

Everware

Everware Reproducible and reusable science powered by jupyterhub and docker. Like nbviewer, but executable. CERN, Geneva website

Microsoft Azure

- https://docs.microsoft.com/en-us/azure/machine-learning/machine-learning-data-science-linux-dsvm-intro

Rackspace Carina

- https://getcarina.com/blog/learning-how-to-whale/
- http://carolynvanslyck.com/talk/carina/jupyterhub/#/

jcloud.io

- Open to public JupyterHub server
  - https://jcloud.io

4.8.3 Miscellaneous

- https://medium.com/@ybarraud/setting-up-jupyterhub-with-sudospawner-and-anaconda-844628c0dbee#.rm3yt87e1
- https://groups.google.com/forum/#!topic/jupyter/nkPSEeMr8c0 Mailing list UT deployment
- JupyterHub setup on Centos https://gist.github.com/johnrc/604971f7d41ebf12370bf5729bf3e0a4
- Deploy JupyterHub to Docker Swarm https://jupyterhub.surge.sh/#/welcome
- http://www.laketide.com/building-your-lab-part-3/
- http://estrellita.hatenablog.com/entry/2015/07/31/083202
- http://www.walkingrandomly.com/?p=5734
- https://wrdrd.com/docs/consulting/education-technology
- https://bitbucket.org/jackhale/fenics-jupyter
- LinuxCluster blog
- Network Technology Spark Cluster on OpenStack with Multi-User Jupyter Notebook

4.9 Changelog

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.

4.9.1 Unreleased

4.9.2 0.8

0.8.1 2017-11-07

JupyterHub 0.8.1 is a collection of bugfixes and small improvements on 0.8.

Added

- Run tornado with AsyncIO by default
- Add `jupyterhub --upgrade-db` flag for automatically upgrading the database as part of startup. This is useful for cases where manually running `jupyterhub upgrade-db` as a separate step is unwieldy.
- Avoid creating backups of the database when no changes are to be made by `jupyterhub upgrade-db`.

Fixed

- Add some further validation to usernames - `/` is not allowed in usernames.
- Fix empty logout page when using auto_login
- Fix autofill of username field in default login form.
- Fix listing of users on the admin page who have not yet started their server.
- Fix ever-growing traceback when re-raising Exceptions from spawn failures.
- Remove use of deprecated `bower` for javascript client dependencies.

0.8.0 2017-10-03

JupyterHub 0.8 is a big release!

Perhaps the biggest change is the use of OAuth to negotiate authentication between the Hub and single-user services. Due to this change, it is important that the single-user server and Hub are both running the same version of JupyterHub. If you are using containers (e.g. via DockerSpawner or KubeSpawner), this means upgrading jupyterhub in your user images at the same time as the Hub. In most cases, a

```
pip install jupyterhub==version
```

in your Dockerfile is sufficient.

Added

- JupyterHub now defined a `Proxy` API for custom proxy implementations other than the default. The defaults are unchanged, but configuration of the proxy is now done on the `ConfigurableHTTPProxy` class instead of the top-level JupyterHub. TODO: docs for writing a custom proxy.
- Single-user servers and services (anything that uses HubAuth) can now accept token-authenticated requests via the Authentication header.
- Authenticators can now store state in the Hub's database. To do so, the `authenticate` method should return a dict of the form

  ```
  {
      'username': 'name'
      'state': {}
  }
  ```

  This data will be encrypted and requires `JUPYTERHUB_CRYPT_KEY` environment variable to be set and the `Authenticator.enable_auth_state` flag to be True. If these are not set, auth_state returned by the Authenticator will not be stored.

- There is preliminary support for multiple (named) servers per user in the REST API. Named servers can be created via API requests, but there is currently no UI for managing them.
- Add `LocalProcessSpawner.popen_kwargs` and `LocalProcessSpawner.shell_cmd` for customizing how user server processes are launched.
- Add `Authenticator.auto_login` flag for skipping the "Login with..." page explicitly.
- Add `JupyterHub.hub_connect_ip` configuration for the ip that should be used when connecting to the Hub. This is promoting (and deprecating) `DockerSpawner.hub_ip_connect` for use by all Spawners.
- Add `Spawner.pre_spawn_hook(spawner)` hook for customizing pre-spawn events.
- Add `JupyterHub.active_server_limit` and `JupyterHub.concurrent_spawn_limit` for limiting the total number of running user servers and the number of pending spawns, respectively.

Changed

- more arguments to spawners are now passed via environment variables (`.get_env()`) rather than CLI arguments (`.get_args()`)
- internally generated tokens no longer get extra hash rounds, significantly speeding up authentication. The hash rounds were deemed unnecessary because the tokens were already generated with high entropy.
- `JUPYTERHUB_API_TOKEN` env is available at all times, rather than being removed during single-user start. The token is now accessible to kernel processes, enabling user kernels to make authenticated API requests to Hub-authenticated services.
- Cookie secrets should be 32B hex instead of large base64 secrets.
- pycurl is used by default, if available.

Fixed

So many things fixed!

- Collisions are checked when users are renamed
- Fix bug where OAuth authenticators could not logout users due to being redirected right back through the login process.
- If there are errors loading your config files, JupyterHub will refuse to start with an informative error. Previously, the bad config would be ignored and JupyterHub would launch with default configuration.
- Raise 403 error on unauthorized user rather than redirect to login, which could cause redirect loop.
- Set `httponly` on cookies because it's prudent.
- Improve support for MySQL as the database backend
- Many race conditions and performance problems under heavy load have been fixed.
- Fix alembic tagging of database schema versions.

Removed

- End support for Python 3.3

4.9.3 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

4.9.4 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

4.9.5 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.

4.9.6 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.

4.9.7 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

4.9.8 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

4.9.9 0.1 - 2015-03-07

First preview release

Python Module Index

j

Index

K

L

M

N

O

P

S

T

U

V