Technical report

JupyterHub and autograding on bare-metal lab servers

Jens Flemming

Zwickau University of Applied Sciences
jens.flemming@fh-zwickau.de
https://www.fh-zwickau.de/∼jef19jdw

DOI: 10.34806/6f2w-v119

February 2022

The Jupyter ecosystem with JupyterHub and JupyterLab as its most prominent members is the de-facto standard for teaching Python programming and also for research in machine learning and data science. Although the Jupyter project is well documented, there are lots of settings and situations requiring deep knowledge of the internal workings of Jupyter, Linux and related software tools. This report describes three problems and possible solutions arising when installing and configuring a Jupyter-based teaching environment. These three problems are the installation and setup of the autograding tool nbgrader, the interplay between JupyterHub and Linux PAM, and providing access to WebDAV resources for users of JupyterHub.

1 JupyterHub and nbgrader in a small multi-class lab environment

Given a bare-metal GPU server in a lab environment installing nbgrader shouldn't be too difficult. Especially, if installing the whole system including JupyterHub and GPU accelerated machine learning stuff was straight-forward. But it turns out that due to the progress in JupyterHub and JupyterLab development, nbgrader is a bit outdated and its installation requires lots of patching and tweaking.

The purpose of this chapter is to collect all the customization and patching steps required to get nbgrader 0.6.2 running in a multi-class and multi-instructor environment, including integration into JupyterHub and JupyterLab's classical Jupyter Notebook interface. Basic installation instructions are taken from the nbgrader documentation.

1.1 Development of nbgrader

In past years nbgrader development slowed down and seems to be stuck to some extent. There is no integration into JupyterLab available, but only in the classical Jupyter Notebook interface (in JupyterLab click "Help" > "Launch Classical Notebook"). Also there are lots of open issues. Some hope for this really useful project comes from creating a new maintainer team.

This lack of progress seems to be the reason for complicated installation. Neighboring projects moved one, while nbgrader did not.

1.2 Where we start

Following software is up and running an the server:

- Ubuntu 20.04
- JupyterHub 2.1.1
- JupyterLab 3.2.8
- Python 3.8

There is an LDAP server in the background which is used for user authentication. User may login via SSH to the server and each user has its own home directory on the server. The nbgrader admin (the one who installs nbgrader and creates new nbgrader courses) has access to everything on the server. In particular, he's able to create new local user accounts and to change file permissions.

JupyterHub installation was done very similar to JupyterHub the hard way, no Kubernetes, no docker. So there is a virtual environment in `/opt/jupyterhub/` containing all hub related installation. Also nbgrader will go there.

1.3 What we want

At least the student facing part of nbgrader should be accessible through the webbrowser. For instructors as much as possible should be manageable via browser. For the nbgrader admin command line is okay, but for everyday work a browser interface is preferred.

We want to use nbgrader for managing assignments for different courses and we want to have different instructors as well as multiple instructors per course.

Autograding should be relatively save, that is, students shouldn't be able to destroy the system or delete an instructor's files. It's possible to run student submissions in a docker container, but this has several drawbacks (complicated configuration, synchronization of Python environments with the outside world,...). So we have to find (and will find) a solution without docker.

1.4 Basic installation

To get nbgrader we install it using `pip` in the virtual environment of JupyterHub:

```
sudo /opt/jupyterhub/bin/python -m pip install nbgrader
```

There are unresolvable dependencies between nbgrader and JupyterHub due to conflicting versions of `traitlets` and `nbconvert`. Installing the most recent versions keeps JupyterHub running, but nbgrader will need some patching (see below):

```
sudo /opt/jupyterhub/bin/python -m pip install --upgrade
     --upgrade-strategy eager traitlets nbconvert
```

1.5 Global nbgrader configuration and logging

Global (that is, relevant for all users) nbgrader configuration is read from `/opt/jupyterhub/etc/jupyter/nbgrader_config.py`. So we create this file with the following content:

```
from nbgrader.auth import JupyterHubAuthPlugin

c = get_config()

c.Exchange.path_includes_course = True
c.Authenticator.plugin_class = JupyterHubAuthPlugin
c.Exchange.root = '/home/nbgrader_exchange'
c.NbGrader.logfile = '/opt/jupyterhub/share/jupyter/nbgrader.log'
```

This prepares multi-course usage (`path_includes_course`), activates user mangement via JupyterHub (`plugin_class`), sets the directory used for file sharing, and activates logging to a file.

We have to create the exchange directory with appropriate permissions:

```
mkdir /home/nbgrader_exchange
sudo chmod ugo+rw /home/nbgrader_exchange
```

Further, the log file has to be created. Permissions of the log file have to be rather weak, else some components of nbgrader run by non-root users will fail.

```
sudo touch /opt/jupyterhub/share/jupyter/nbgrader.log
sudo chmod 666 /opt/jupyterhub/share/jupyter/nbgrader.log
```

1.6 Installing Jupyter extensions

Next we install extensions to Jupyter's web interface to access nbgrader from the web-browser:

```
sudo /opt/jupyterhub/bin/jupyter nbextension install
     --sys-prefix --py nbgrader
sudo /opt/jupyterhub/bin/jupyter nbextension enable
     --sys-prefix --py nbgrader
sudo /opt/jupyterhub/bin/jupyter serverextension enable
     --sys-prefix --py nbgrader
```

We disable all extensions not needed by students. Extensions used by instructors will be enabled later on for each instructor separately.

```
sudo /opt/jupyterhub/bin/jupyter nbextension disable
     --sys-prefix create_assignment/main
sudo /opt/jupyterhub/bin/jupyter nbextension disable
     --sys-prefix formgrader/main --section=tree
sudo /opt/jupyterhub/bin/jupyter serverextension disable
     --sys-prefix nbgrader.server_extensions.formgrader
sudo /opt/jupyterhub/bin/jupyter nbextension disable
     --sys-prefix course_list/main --section=tree
sudo /opt/jupyterhub/bin/jupyter serverextension disable
     --sys-prefix nbgrader.server_extensions.course_list
```

1.7 Some patching

To get nbgrader running in an up-to-date Python environment together with a recent JupyterHub version requires several patches. Some of them already seem to be integrated into nbgrader's GitHub master branch, but not all. Here we show how to patch vanilla nbgrader 0.6.2.

Some, not all, patches listed below are discussed in plenty of GitHub issues on nbgrader. There's also pull request 1405, which gives important hints on how to get nbgrader running.

Rename all files in `/opt/jupyterhub/lib/python3.8/site-packages/nbgrader/-server_extensions/formgrader/templates`. The extension `tpl` has to be replaced by `html.j2`. Else, there will be "template not found" erros. These modifications have to be made, too, wherever the renamed template files are referenced (precise listing of locations follows).

In all files contained in `/opt/jupyterhub/lib/python3.8/site-packages/nbgrader/-server_extensions/formgrader/templates` replace references to `*.tpl` files by corresponding `*.html.j2`.

Replace `.tpl` by `.html.j2` in following files:

- `/opt/jupyterhub/lib/python3.8/site-packages/nbgrader/apps/api.py`
- `/opt/jupyterhub/lib/python3.8/site-packages/nbgrader/converters/-generate_feedback.py`
- `/opt/jupyterhub/lib/python3.8/site-packages/nbgrader/server_-extensions/formgrader/base.py`
- `/opt/jupyterhub/lib/python3.8/site-packages/nbgrader/server_-extensions/formgrader/handlers.py`

Remember that the installed version of nbconvert is newer than allowed by nbgrader. So we have to make some changes to nbgrader's source.

In `/opt/jupyterhub/lib/python3.8/site-packages/nbgrader/server_extensions/formgrader/templates/formgrade.html.j2` and `.../feedback.html.j2` replace the line

```
{%- extends 'basic.html.j2' -%}
```

by

```
{%- extends 'classic/index.html.j2' -%}
```

In `/opt/jupyterhub/lib/python3.8/site-packages/nbgrader/converters/generate_feedback.py` replace

```
if 'template_path' not in self.config.HTMLExporter:
```

by

```
if 'template_paths' not in self.config.HTMLExporter:
```

and

```
c.HTMLExporter.template_path = ['.', template_path]
```

by

```
c.HTMLExporter.template_path = ['.', template_path]
```

In the definition of `build_extra_config` in `/opt/jupyterhub/lib/python3.8/site-packages/nbgrader/server_extensions/formgrader/formgrader.py` replace

```
extra_config.HTMLExporter.template_path = [handlers.template_path]
```

by

```
extra_config.HTMLExporter.template_paths.append(handlers.template_path)
```

In `/opt/jupyterhub/lib/python3.8/site-packages/nbgrader/server_extensions/formgrader/templates/feedback.html.j2` add the line

```
{{ resources.include_css("classic/static/style.css") }}
```

below the line

```
{{ mathjax() }}
```

Else, student feedback looks scrambled.

1.8 Adding configuration option for hiding hidden tests in feedbacks

By default, hidden tests for autograding will show up in student feedback. To add an option to remove hidden tests from feedback, we have to apply a further patch.

In `/opt/jupyterhub/lib/python3.8/site-packages/nbgrader/converters/generate_feedback.py` replace the last line of

```
preprocessors = List([
    GetGrades,
    CSSHTMLHeaderPreprocessor
])
```

by

```
]).tag(config=True)
```

1.9 JupyterHub configuration

JupyterHub requires some configuration for multi-class nbgrader usage. We have to create a service running the autograding tool. One reason for this is, that this way autograding is accessible to more than one instructor. Another reason is, that student code shouldn't run in an instructors user account (with access to the instructor's files).

Configuration is described in nbgraders documentation (outdated). Some fixes for getting nbgrader running with recent JupyterHub versions are suggested in issue 1533. Here we adapt the proposed fixes to our setting.

In the definition of `get_jupyterhub_authorization` in `/opt/jupyterhub/lib/-python3.8/site-packages/nbgrader/auth/jupyterhub.py` replace `JUPYTERHUB_API_TOKEN` by `JUPYTERHUB_API_TOKEN_CUSTOM`.

Go to JupyterHub's Hub Control Panel, click the admin button, and create an API token. Then add the following line to your JupyterHub config file:

```
c.Spawner.environment = { 'JUPYTERHUB_API_TOKEN_CUSTOM':
                                '0123456789abcdef0123456789abcdef' }
```

This sends the API token to the spawner environment. So every (!) spawned single-user server may access it, but only servers using nbgrader require it. Note a nice solution, but it works.

In JupyterHub's config file create a token service for this API token:

```
c.JupyterHub.services = [
    {
        'name': 'nbgrader_token_service',
        'api_token': '0123456789abcdef0123456789abcdef'
    }
]

c.JupyterHub.load_roles = [
    {
        'name': 'nbgrader_token_role',
        'scopes': ['read:users:groups', 'list:services',
                   'groups', 'admin:users'],
        'services': ['nbgrader_token_service']
    }
]
```

If there already are services or roles, you have to add the new ones to the existing lists. Lines `c.JupyterHub.services = [` and `c.JupyterHub.load_roles = [` should only appear once in the config file. The purpose of such a token service is to set permissions for API requests using this token (see JupyterHub documentation for details).

1.10 Promote regular hub user to instructor

Intructors are hub users with admin access (maybe not necessary) having some nbgrader Jupyter extensions enabled.

Add the username to `c.Authenticator.admin_users` in JupyterHub's config file. Then login to the instructor user's account via SSH or JupyterLab terminal and run

```
/opt/jupyterhub/bin/jupyter nbextension enable
    --user course_list/main --section=tree
/opt/jupyterhub/bin/jupyter serverextension enable
    --user nbgrader.server_extensions.course_list
```

1.11 Create a new course

To create a new nbgrader course we have to create a new user (the grader) running the autograding tool:

```
sudo adduser grader-test-course
```

In JupyterHub's config file we grant the grader access to the hub by adding its username `grader-test-course` to `c.Authenticator.allowed_users`. Then we create two groups `formgrade-test-course` and `nbgrader-test-course` in JupyterHub's config file:

```
c.JupyterHub.load_groups = {
    'formgrade-test-course': ['instructor', 'grader-test-course'],
    'nbgrader-test-course': []
}
```

Group names are prescribed by nbgrader. And it's really `formgrade...`, not `formgrader...`! The second group remains empty.

Now the autograding service has to be configured in the hub config file:

```
c.JupyterHub.services = [
    {
        'name': 'test-course',
        'url': 'http://127.0.0.1:8101',
        'command': ['jupyterhub-singleuser',
                    '--group=formgrade-test-course',
                    '--debug'],
        'user': 'grader-test-course',
        'cwd': '/home/grader-test-course',
        'api_token': '0123456789abcdef0123456789abcdef',
            # api_token can be removed (not tested)
        'environment': { 'JUPYTERHUB_API_TOKEN_CUSTOM':
                         '0123456789abcdef0123456789abcdef' }
    }
]
```

```
c.JupyterHub.load_roles = [
    {
        'name': 'formgrader-test-course-role',
        'groups': ['formgrade-test-course'],
        'scopes': ['access:services!service=test-course']
    }
]
```

Remember that only one `c.JupyterHub.services = [` line is allowed (same for `load_roles`).

Log in to the grader account activate relevant nbgrader Jupyter extensions:

```
/opt/jupyterhub/bin/jupyter nbextension enable
    --user create_assignment/main
/opt/jupyterhub/bin/jupyter nbextension enable
    --user formgrader/main --section=tree
/opt/jupyterhub/bin/jupyter serverextension enable
    --user nbgrader.server_extensions.formgrader
/opt/jupyterhub/bin/jupyter nbextension disable
    --user assignment_list/main --section=tree
/opt/jupyterhub/bin/jupyter serverextension disable
    --user nbgrader.server_extensions.assignment_list
```

Then create the grader's `~/.jupyter/nbgrader_config.py`:

```
c = get_config()

c.CourseDirectory.root = '/home/grader-test-course/test-course'
c.CourseDirectory.course_id = 'test-course'

c.GenerateFeedback.preprocessors = [
    'nbgrader.preprocessors.GetGrades',
    'nbconvert.preprocessors.CSSHTMLHeaderPreprocessor',
    # remove: hidden tests from feedback:
    #'nbgrader.preprocessors.ClearHiddenTests',
    # remove tracebacks of hidden tests from feedback
    #'nbgrader.preprocessors.Execute',
]
```

Handling of hidden tests in student feedback is discussed in issue 917.

Finally, create the directory holding all the course assignments:

```
mkdir ~/test-course
```

Students can be added to the course via nbgrader's web interface. But **in addition** the following command has to be run from the grader user's command line:

```
nbgrader db student add STUDENT_NAME
```

This can be done from JupyterLab. Simply go to the URL `https://location/of/hub/-services/test-course` and start a Terminal. That this additional step is necessary is described in issue 1396

2 JupyterHub and Linux PAM

JupyterHub is a tool for starting and managing JupyterLab (and Jupyter Notebook) sessions in a multi-user environment via webbrowser. As such it provides a login facility to a (Linux) server, allowing to work on the server without logging in via SSH. From the admin point of view it is desirable to configure both login processes (via JupyterHub and via SSH, and maybe also local logins) in a unified manner.

Linux PAM (Pluggable Authentication Modules) is the tool of choice for managing user authentication and doing stuff at login and logout (mounting network shares, for instance). It's the standard tool in almost all Linux distributions.

Although JupyterHub ships with native PAM support, the web is full of discussions on how to get JupyterHub/PAM working in the intended way. Taking the time and digging into this reveales that PAM support in JupyterHub (version <= 2.1.1) is essentially broken (by design) and hard to fix. Some GitHub issues:

- pamela issue 22 (pamela is a Python module used by JupyterHub for doing PAM stuff)
- JupyterHub issue 2120, see also pull request 2321
- JupyterHub issue 2973
- JupyterHub issue 810
- JupyterHub issue 809
- JupyterHub issue 3106
- JupyterHub issue 2406
- JupyterHub issue 1658

In this chapter I describe the details of the bug and show how to cope with the situation until JupyterHub's PAM support gets fixed some day. JupyterHub developers are aware of the bug and discussing a path forward. I also tried to fix it myself, but ran into another bug (in `pam_mount`), which presumably won't get fixed. Situation is highly nontrivial and demands for explanation and documentaion...

2.1 Problem description

In small lab environments using PAM authentication for JupyterHub is a straightforward solution to many authentication and resource allocation problems. Corresponding authenticator class is called `PAMAuthenticator`. It's JupyterHub's default authenticator.

By default next to PAM authentication also PAM session handling is enabled. It can be disabled in JupyterHub's config file via

```
c.PAMAuthenticator.open_sessions = False
```

Note that the default will change to `False` soon as a result of discussing the PAM bug details in the past weeks, see pull request 3787.

With PAM session handling enabled, logging in to JupyterHub should initiate typical login actions like mounting the user's network shares. But this does not happen due to permission issues. Typical errors from the logs are:

- `pam_loginuid(login:session): Cannot open /proc/self/loginuid: Permission denied`
- `pam_systemd(login:session): Failed to create session: Access denied`
- `(pam_mount.c:477): warning: could not obtain password interactively either`
- `(mount.c:76): mount error(13): Permission denied`

Nonetheless, JupyterHub will start JupyterLab and the user is able to work in JupyterLab. But network shares won't be shown and resource management (limits on memory or CPU usage, for instance) won't work as intended. In addition, user sessions won't be managed by `systemd`, leaving the system's process management in an undesired state.

2.2 How PAM works

Understand JupyterHub's PAM session handling problem requires advanced PAM knowledge.

PAM transactions

An application (here JupyterHub or JupyterLab) which wants to use PAM functionality has to start a PAM transaction by calling `libpam`'s `pam_start` function. When all work is done, a call to `pam_end` ends the PAM transaction.

A PAM transaction consists of several optional stages:

- account stage (is the user allowed to access the system?)
- authentication stage (is the user who he claims to be?)
- password stage (do we have to update authentication information (password, for instance)?
- session (what to do at login and logout?)

The most important feature of PAM is that within a PAM transaction the user has to provide it's password only once (authentication stage). All further password requests, for instance if network shares have to be mounted at login, are automatically answered by PAM. What to do during each stage is highly configurable and can be configured on a per-app basis. Configuration files usually reside in `/etc/pam.d/`. See `pam.d` manual page for details.

Authentication is done by calling `libpam`'s pam_authenticate. A call to `pam_open_session` starts a PAM session, that is, performs the configured login tasks. A call to `pam_close_session` ends the session stage, that is, performs the logout tasks.

PAM modules

All PAM functionality is devided into PAM modules. Important PAM modules for the session stage are `pam_mount` (for mounting and unmounting file systems) and `pam_systemd` (for getting some `systemd` stuff running, resource limitations, for instance). Each PAM module is a shared object (shared library).

PAM configuration files determine which modules to use in which stage. Each module provides a function for each supported stage. This function is called by `libpam` during the corresponding stage. For examples an app's call to `pam_authenticate` leads to calling each module's `pam_sm_authenticate` if the module is configured to be part of the app's authentication stage.

Example: The `pam_mount` module supports authentication and session stages. In the authentication stage user provided credentials are passed on to `pam_mount`. In the session stage `pam_mount` uses these credentials to mount network shares or encrypted local drives.

PAM handles

Starting a PAM transaction yields a PAM handle identifying the transaction. Thus, there may be multiple parallel PAM transactions, even per app. The handle is provided to each PAM module at each call to the module. PAM modules have to take care of separating data and states from different transactions.

In `pam_mount` this separation mechanism is buggy. Separation between different apps' PAM transactions works, but multiple parallel PAM transactions per app result in a segmentation fault. See bug report for details on what exactly causes the segfault. Relation to the PAM session handling issue in JupyterHub will become clear below. Fixing this `pam_mount` bug woudl require major modifications to `pam_mount`'s C source code, because a global data structure has to be made a per-PAM-handle data structure.

2.3 JupyterHub vs. JupyterLab

JupyterHub is a multi-user system spawning one or multiple JupyterLab instances for each user. Users may login and logout at will while their JupyterLabs keep running. Understanding the interaction between JupyterHub, JupyterLab and PAM requires at least some basic knowledge of JupyterHub's design.

Keeping authentication aside for a moment users can tell JupyterHub to start a so called single-user server, usually a JupyterLab instance. Starting multiple single-user servers per user is possible, too. The user also can tell the hub to shutdown one or serveral of the users's servers. Single-user servers run as separate Linux processes in user space. The hub itself ususally is run by the `root` user.

If the a user logs in to the hub, a single-user server is started if there is no already running one. Logging out from the hub does not stop the user's server. The user may come back later, log in to the hub, and continue working in the already running JupyterLab. So authentication to the hub is more or less unrelated to starting and stopping single-user servers. This has to be taken into account when dealing with PAM and it makes PAM session handling rather difficult.

2.4 JupyterHub's PAM session handling

We first discuss why JupyterHub's PAM code is incorrect. Then we'll have a look at possible solutions.

Broken PAM implementation

Looking at JupyterHub's PAM related source code everything looks fine. In `PAMAuthenticator.authenticate` there is a call to some PAM authentication function and also to some PAM account checking. In `PAMAuthenticator.pre_spawn_start` and `PAMAuthenticator.post_spawn_stop` there are calls for opening and closing a PAM session, respectively.

Calls don't go directly to `libpam`, but to a wrapper Python module called `pamela`. Inspecting `pamela`'s functions JupyterHub's calls to `libpam` are as follows:

1. `pam_start`
2. `pam_authenticate`
3. `pam_end`
4. `pam_start`
5. `pam_pam_acct_mgmt`
6. `pam_end`
7. `pam_start`

8. `pam_open_session`
9. `pam_end`
10. `pam_start`
11. `pam_close_session`
12. `pam_end`

For each PAM stage there is a separate PAM transaction! This explains several seemingly different JupyterHub issues (see issue list and error messages above). Although the user is authenticated successfully, this information gets lost. All the other stages and their PAM modules are run as unauthenticated user resulting in all kinds of permission errors.

Note, that the `pamela` module also provides functions for doing everything in one PAM transaction, but JupyterHub does not use those functions. So it's not a `pamela` issue although recent discussion of the JupyterHub/PAM issue on GitHub took place in `pamela` issue 22.

Attempt 1: fix `pamela` usage (fails due to `pam_mount` bug)

A relatively simple and straight forward attempt to fix JupyterHub's PAM session handling would be to open a PAM session as soon as the user spawns a single-user server and to close the PAM session if the server has terminated. This is what's currently implemented in JuypterHub. But one has to do all the PAM stuff related to a single-user server in one PAM transaction.

I implemented this approach. It works as long as there is only one user with one single-user server on the hub. Starting a second server results in a segmentation fault. Tracking it down, the segfault is caused by `pam_mount` during closing a PAM session. As described above `pam_mount` does not support parallel PAM sessions within one application (JupyterHub). This has to be considered a bug, because PAM documentation explicitly states that "The transaction state is contained entirely within the structure identified by this handle, so it is possible to have multiple transactions in parallel." PAM development seems to by rather inactive, so presumably the bug won't get fixed (and fixing it is not trivial).

Of course, one could abstain from `pam_mount`. Then PAM session handling in JupyterHub would work with the described fix. But for most admins the reason to get JupyterHub with PAM working is to mount network shares at login to JupyterHub. So having `pam_mount` in the PAM stack is essential.

Attempt 2: open PAM session in single-user server (fails due to `libpam` implementation details)

In principle, the single-user server (JupyterLab) could do PAM session handling. But opening a PAM session has to be done within the same PAM transaction as used for

authentication. Authentication is done by JupyterHub, so JupyterHub has to pass on the PAM transaction handle (or PAM handle for short) to the single-user server.

The problem is that PAM handles are not serializable. PAM handles are pointers to somewhere in memory and passing such pointers to other processes directly should segfault. Copying the information pointed to by the PAM handle would require to rely on implementation details of `libpam`. Layout of the data structure is not an API feature of `libpam` and may change from version to version.

Note, that this approach also would contradict JupyterHub's design. JupyterHub should do all the authentication and user management stuff, while the single-user server does not have to care such things. In particular, JupyterHub's single-user server should be as similar as possible to usual Jupyter servers running without the hub.

Attempt 3: start PAM transaction in single-user server (fails due to security concerns and complexity)

Starting PAM transaction in the single-user server would be a clean solution. But then JupyterHub would have to pass user credentials to the single-user server. Without authentication no `pam_mount` for network shares in a PAM transaction. Without obtaining user credentials from the hub the single-user server has to ask the user for the password. So the user has to type its password several times (for the hub and for each spawned server).

Passing credentials to the single-user server in clear text is not a good idea considering security implications. So communication has to be encrypted, but there is no standard encrypted communication channel between hub and servers. Usually, information is passed via environment variables. This path is very complex to implement and will remain dubious regarding security.

Again, this approach contradicts JupyterHub's design (cf. above).

Attempt 4: use intermediate processes (maybe successful, but not yet implemented)

The idea described in a comment to `pamela` issue 22 and previously also outlined in a comment to JupyterHub pull request could be successful approach to getting PAM sessions work in JupyterHub. But implementation is complex, too complex for me as an average JupyterHub admin. There are chances that things get fixed this year by the JupyterHub team, now that there is much more light on the problem's details.

JupyterHub without PAM sessions

Until PAM session handling in JupyerHub gets fixed, one has to live without PAM sessions. The first thing to do is to put

```
c.PAMAuthenticator.open_sessions = False
```

into JupyterHub's configuration file.

Mounting file system at login is hard to do without PAM. Seems that `systemd` also has mounting capabilities. But this would require major changes to the system configuration. A simple solution, which requires no configuration, is to tell hub users to start a terminal in JupyterLab and initiate a SSH connection via `ssh localhost`, if they want to see their network shares.

For getting resource limiting without PAM one can use JupyterHub's `SystemdSpawner` class instead of the default `LocalProcessSpawner`. This requires installation of `systemdspawner`, the line

```
c.JupyterHub.spawner_class = 'systemdspawner.SystemdSpawner'
```

in JupyterHub's config file, and maybe some additional spawner configuration lines.

3 Mounting WebDAV resources with `pam_mount`

Trying to mount a WebDAV resource at SSH login via `pam_mount` on a Ubuntu 20.04 server I had to realize that this is not as straight forward as it should be. Although there is `mount.davfs`, a buggy security feature in `pam_mount` and permission problems with storing secrets in a user-owned file prevent using the obvious combination of `pam_mount` and `mount.davfs`. In the end I had to use the rather old and no longer maintained `wdfs` tool and FUSE to get things working.

Fiddling with this problem took several hours. Mounting WebDAV resources via `pam_mount` seems to be a rarely used feature, at least in my desired user configurable setting. Searching the web yields only very few results, most of them outdated. So here comes an up-to-date account of mounting WebDAV resources at login on a Linux machine.

3.1 What we want to achieve

On a multi-user Linux server we want to mount WebDAV resources (cloud storage) specified and owned by the user. The user shall be able to control which resources to mount (personal cloud storage) and the admin shouldn't be involved in managing secrets required for accessing the user's WebDAV resources. Of course, if the user stores its secrets to a file in his home directory the admin may see them. But writing secrets to a file shouldn't be done by the admin, so a well-behaved admin won't ever see them.

3.2 Attempt 1: per-user `pam_mount` configuration (failed)

Having a line

```
<luserconf name=".pam_mount.conf.xml" />
```

in `pam_mount`'s global configuration file `/etc/security/pam_mount.conf.xml` allows for additional user-owned configuration files `/home/username/.pam_mount.conf.xml`. Such user-owned configuration files may contain additional `<volume>...</volume>` sections,

which are only evaluated if the corresponding user logs in to the server. If you use this feature, don't forget to set

```
<mntoptions allow="..." />
<mntoptions deny="..." />
<mntoptions require="..." />
```

in the global configuration file to values appropriate for your setting and users.

To mount a WebDAV resource, the user has to put

```
<volume fstype="davfs" path="WEBDAV_URL" options="SOME_OPTIONS" />
```

into its `.pam_mount.conf.xml`. Secrets will be read from the file `~/.davfs2/secrets` or may be specified as `username=WEBDAV_USERNAME,password=WEBDAV_PASSWORD` in `options` attribute.

This obvious setup fails due to a bug in `pam_mount`. To prevent users doing bad things in their `.pam_mount.conf.xml` `pam_mount` checks whether the resource to mount is a file owned by the user. Of course, there are some exceptions for remote resources, but file systems of type `davfs` are not considered to be remote by `pam_mount`. Logs show

```
(rdconf2.c:132):
user-defined volume (https://location/of/webdav), volume not owned by user
```

Thinking that the specified path goes to a file in the server's file system, `pam_mount` tries to get its ownership, which fails. So using `davfs` in user-specified `volume` descriptions for `pam_mount` doesn't work.

I did not write a bug report on this issue for two reasons:

- Development of `pam_mount` isn't really active, an increasing problem with several widely used but not so prominent open source tools.
- Having the bug fixed, we would run into permission problems caused by `davfs`'s security measures. During login `pam_mount` is run as root and `davfs` expects the secrets file to be owned by the mounting user (that is, by root) and to have permissions `600`. So the user won't be able to access its secrets file. See below for details.

3.3 Attempt 2: `davfs` in the global `pam_mount` config file (failed)

In my setup there is a cloud service potentially used by all users. So at least this WebDAV resource could be configured in the global `pam_mount` config file. Secrets should be kept in user-owned `secrets` files. Using `pam_mount`'s variables like `%(USER)` the `volume` specification looks the same for all users (luckily, user names the resource's URL are identical to the server's user names). The `secrets` file's permission has to be `600` and the file has to be owned by the user and the user's group. That's an explicit requirement by `mount.davfs`.

First log in with this new configuration shows

```
(mount.c:76): /sbin/mount.davfs:
file /home/username/.davfs2/secrets has wrong owner
```

in the logs. The reason for this failure is that `pam_mount` is run as root and root doesn't own the user's `secrets` file.

A way out would be to provide user secretes as options, requiring one `volume` block per user in the config file. But then the admin would have to explicitly ask all users for their cloud passwords. Not a acceptable solution in my setting. So `mount.davfs` is dead for my purposes.

3.4 Attempt 3: using FUSE with something similar to `sshfs` (successful)

Trying to get attempt 1 fixed (configuration by user) we have to find a file system type considered "remote" by `pam_mount` and being flexible enough to cope with WebDAV. A good candidate is `type="fuse"`, which I already use to mount via SSH using `sshfs`. The problem is to find something similar to `sshfs` which mounts WebDAV resources. With `sshfs` we may run `sshfs SERVER MOUNT_POINT` from the shell to mount via SSH. So we need a program doing the same thing with WebDAV.

Searching the web yields two candidates:

- `fusedav`, last updated in 2006,
- `wdfs`, last updated 2007, moved to a GitHub repository by someone in 2010 with minor updates in 2015.

So let's try `wdfs`. It's not packaged with Ubuntu, we have to compile it on our own. First clone the git repository:

```
cd ~
git clone https://github.com/jmesmon/wdfs
cd wdfs
```

Then install dependencies (running `configure` tells us what's missing):

```
sudo apt install libglib2.0-dev libfuse-dev libneon27-dev
```

Compile and install (`autoreconf` seems to be needed due to some time stamp issues caused by using git):

```
autoreconf -f -i
./configure
make
sudo make install
```

Successful?

```
wdfs -h
```

It remains to put

```
<volume
    fstype="fuse"
    path="wdfs#https://location/of/webdav"
    mountpoint="/home/username/mount_point"
    options="password=WEBDAV_PASSWORD,username=WEBDAV_USERNAME,
             nodev,nosuid,uid=%(USERUID),gid=%(USERGID)"
/>
```

into the user-owned `.pam_mount.conf.xml` and mounting at login works like a charme.