Integrating JupyterHub with Nextcloud

If you don't know what Jupyter Notebooks are, I encourage you to check them out. They give you a way to write programs, in a variety of programming languages, and share those with your friends and colleagues (even if they themselves are not as tech savvy as you are). JupyterHub in turn gives you a way to easily manage a large number of notebooks presented on the internet.

Now, wouldn't it be cool if you could display Jupyter notebooks inside of Nextcloud, and have access to all your Nextcloud files inside the Jupyter interface?

I think so.

TL;DR;

Our code and k8s manifests are available here, in the `jupyter` directory: https://platform.sunet.se/Drive/k8s-manifests

Prerequisites

For our use case we will be deploying JupyterHub in Kubernetes using the wonderful Zero to JupyterHub with Kubernetes. To make the file syncing work as expected later, we will need to patch the Helm charts unfortunately (I will see if I can upstream these changes, so stay tuned for future updates).

hub.patch

```
diff --git a/jupyterhub/templates/hub/deployment.yaml b/jupyterhub
/templates/hub/deployment.yaml
index d6e1c63..359e981 100644
--- a/jupyterhub/templates/hub/deployment.yaml+++ b/jupyterhub/templates
/hub/deployment.yaml
@@ -33,6 +33,10 @@ spec:
         {{- . | toYaml | nindent 8 }}
         {{- end }}
     spec:
+      hostAliases:
+        - ip: "127.0.0.1"
+          hostnames:
+            - "hub"
       {{- if .Values.scheduling.podPriority.enabled }}
       priorityClassName: {{ include "jupyterhub.priority.fullname" . }}
       {{- end }}
@@ -211,6 +215,8 @@ spec:
           ports:
             - name: http
               containerPort: 8081
+            - name: refresh-token
+              containerPort: 8082
           {{- if .Values.hub.livenessProbe.enabled }}
           {{- /* NOTE:
             We don't know how long hub database upgrades could take so
having a
```

```
helm repo add jupyterhub https://hub.jupyter.org/helm-chart/
helm repo update
helm fetch jupyterhub/jupyterhub --version 3.2.1 --untar --untardir .
patch -p1 < hub.patch
```

Authenticating Jupyter with Nextcloud

The first piece of the puzzle is to use Nextcloud for authentication in JupyterHub. This can be achieved out of the box, with some configuration. In our case, we will implement a sub class of the GenericOAuthenticator class. The code will take care of authentication with oauth2 against Nextcloud and and will also refresh the access_token for the user, when it expires.

You need to create a oauth2 client in Nextcloud and get a client_id and client_secret from Nextcloud: https://docs.nextcloud.com/server/latest/admin_manual/configuration_server/oauth2.html

These needs to be stored as kubernetes secrets, look at the very bottom of this post to find out what they sould be called.

The code looks like this:

```
      import time
      import requests
      from datetime import datetime
      from oauthenticator.generic import GenericOAuthenticator
      token_url = 'https://' + os.environ['NEXTCLOUD_HOST'] + '/index.php
/apps/oauth2/api/v1/token'
      debug = os.environ.get('NEXTCLOUD_DEBUG_OAUTH', 'false').lower() in
['true', '1', 'yes']

      def get_nextcloud_access_token(refresh_token):
        client_id = os.environ['NEXTCLOUD_CLIENT_ID']
        client_secret = os.environ['NEXTCLOUD_CLIENT_SECRET']

        code = refresh_token
        data = {
          'grant_type': 'refresh_token',
          'code': code,
          'refresh_token': refresh_token,
          'client_id': client_id,
          'client_secret': client_secret
        }
        response = requests.post(token_url, data=data)
        if debug:
          print(response.text)
        return response.json()

      def post_auth_hook(authenticator, handler, authentication):
        user = authentication['auth_state']['oauth_user']['ocs']['data']
['id']
        auth_state = authentication['auth_state']
        auth_state['token_expires'] =  time.time() + auth_state
['token_response']['expires_in']
        authentication['auth_state'] = auth_state
        return authentication

      class NextcloudOAuthenticator(GenericOAuthenticator):
        def __init__(self, *args, **kwargs):
          super().__init__(*args, **kwargs)
          self.user_dict = {}

        async def pre_spawn_start(self, user, spawner):
          super().pre_spawn_start(user, spawner)
          auth_state = await user.get_auth_state()
          if not auth_state:
            return
          access_token = auth_state['access_token']
          spawner.environment['NEXTCLOUD_ACCESS_TOKEN'] = access_token

        async def refresh_user(self, user, handler=None):
          auth_state = await user.get_auth_state()
          if not auth_state:
            if debug:
              print(f'auth_state missing for {user}')
            return False
          access_token = auth_state['access_token']
```

```
          refresh_token = auth_state['refresh_token']
          token_response = auth_state['token_response']
          now = time.time()
          now_hr = datetime.fromtimestamp(now)
          expires = auth_state['token_expires']
          expires_hr = datetime.fromtimestamp(expires)
          expires = 0
          if debug:
            print(f'auth_state for {user}: {auth_state}')
          if now >= expires:
            if debug:
              print(f'Time is: {now_hr}, token expired: {expires_hr}')
              print(f'Refreshing token for {user}')
            try:
              token_response = get_nextcloud_access_token(refresh_token)
              auth_state['access_token'] = token_response['access_token']
              auth_state['refresh_token'] = token_response['refresh_token']
              auth_state['token_expires'] = now + token_response
['expires_in']
              auth_state['token_response'] = token_response
              if debug:
                print(f'Successfully refreshed token for {user.name}')
                print(f'auth_state for {user.name}: {auth_state}')
              return {'name': user.name, 'auth_state': auth_state}
            except Exception as e:
              if debug:
                print(f'Failed to refresh token for {user}')
              return False
            return False
          if debug:
            print(f'Time is: {now_hr}, token expires: {expires_hr}')
          return True

      c.JupyterHub.authenticator_class = NextcloudOAuthenticator
      c.NextcloudOAuthenticator.client_id = os.environ
['NEXTCLOUD_CLIENT_ID']
      c.NextcloudOAuthenticator.client_secret = os.environ
['NEXTCLOUD_CLIENT_SECRET']
      c.NextcloudOAuthenticator.login_service = 'Sunet Drive'
      c.NextcloudOAuthenticator.username_claim = lambda r: r.get('ocs',
{}).get('data', {}).get('id')
      c.NextcloudOAuthenticator.userdata_url = 'https://' + os.environ
['NEXTCLOUD_HOST'] + '/ocs/v2.php/cloud/user?format=json'
      c.NextcloudOAuthenticator.authorize_url = 'https://' + os.environ
['NEXTCLOUD_HOST'] + '/index.php/apps/oauth2/authorize'
      c.NextcloudOAuthenticator.token_url = token_url
      c.NextcloudOAuthenticator.oauth_callback_url = 'https://' + os.
environ['JUPYTER_HOST'] + '/hub/oauth_callback'
      c.NextcloudOAuthenticator.allow_all = True
      c.NextcloudOAuthenticator.refresh_pre_spawn = True
      c.NextcloudOAuthenticator.enable_auth_state = True
      c.NextcloudOAuthenticator.auth_refresh_age = 3600
      c.NextcloudOAuthenticator.post_auth_hook = post_auth_hook
```

Syncing files from Nextcloud to JupyterHub

To sync the files from Nextcloud to JupyterHub we will create a custom Jupyter Labs/Notebook image that has a sync client in it.

Dockerfile

```
FROM quay.io/jupyter/scipy-notebook:lab-4.0.10
USER root
RUN apt-get update && apt-get upgrade -y && apt-get install -y \
  curl \
  jq \
  nextcloud-desktop-cmd \
  python3-pip
COPY ./nc-sync /usr/local/bin/
USER jovyan
```

The nc-sync script looks like this:

```
#!/bin/bash
rmdir --ignore-fail-on-non-empty /home/jovyan/work
workdir='/home/jovyan'
mkdir -p ${workdir}
server="https://sunet.drive.test.sunet.se"
json_file="${workdir}/.access_token.json"

function refresh_token {
  json="$(curl --header "Authorization: token ${JUPYTERHUB_API_TOKEN}"
https://${JUPYTER_HOST}/services/refresh-token/tokens)"
  # If json is empty here we are early in the process and should have a
fresh token from the environment
  if [[ -z "${json}" ]]; then
    token="${NEXTCLOUD_ACCESS_TOKEN}"
    json="{ \"access_token\": \"${token}\", \"token_expires\": $(date -d
"10 min" +%s).0000000 }"
  fi
  echo "${json}" > "${json_file}"
  token=$(jq -r '.access_token' "${json_file}")
  echo "${token}"
}

function get_token {
  # First we try to use our cache
  if [[ -f "${json_file}" ]]; then
    now=$(date +%s)
    token=$(jq -r '.access_token' "${json_file}")
    expires_at=$(jq -r '.token_expires' "${json_file}"| sed 's/\..*//')
    # If the token is expired, we fetch a new one
    if [[ "${expires_at}" -lt ${now} ]]; then
      token=$(refresh_token)
    fi
  else
    token=$(refresh_token)
  fi
  echo "${token}"
}

function ncsync {
  while true; do
    nextcloudcmd -s --user ${JUPYTERHUB_USER} --password $(get_token) --
path / "${workdir}" "${server}"
    sleep 5s
  done
}
ncsync &
```

For this sync script to work we also need a JupyterHub service:

```
      import sys
      c.JupyterHub.load_roles = [
          {
              "name": "refresh-token",
              "services": [
                "refresh-token"
              ],
              "scopes": [
                "read:users",
                "admin:auth_state"
              ]
          },
          {
              "name": "user",
              "scopes": [
                "access:services!service=refresh-token",
                "read:services!service=refresh-token",
                "self",
              ],
          },
          {
              "name": "server",
              "scopes": [
                "access:services!service=refresh-token",
                "read:services!service=refresh-token",
                "inherit",
              ],
          }
      ]
      c.JupyterHub.services = [
          {
              'name': 'refresh-token',
              'url': 'http://' + os.environ.get('HUB_SERVICE_HOST', 'hub')
+ ':' + os.environ.get('HUB_SERVICE_PORT_REFRESH_TOKEN', '8082'),
              'display': False,
              'oauth_no_confirm': True,
              'api_token': os.environ['JUPYTERHUB_API_KEY'],
              'command': [sys.executable, '/usr/local/etc/jupyterhub
/refresh-token.py']
          }
      ]
      c.JupyterHub.admin_users = {"refresh-token"}
      c.JupyterHub.api_tokens = {
          os.environ['JUPYTERHUB_API_KEY']: "refresh-token",
      }
```

In theory it should be enough to have the load_roles config along with the services config, and the admin_users and api_tokens should not be needed, but for me it does not seem to work without it. YMMV. The executable `refresh-token.py` is a piece of code that looks like this:

refresh-token.py

```
        """A token refresh service authenticating with the Hub.

        This service serves `/services/refresh-token/`,
        authenticated with the Hub,
        showing the user their own info.
        """
        import json
        import os
        import requests
        import socket
        from jupyterhub.services.auth import HubAuthenticated
        from jupyterhub.utils import url_path_join
        from tornado.httpserver import HTTPServer
        from tornado.ioloop import IOLoop
        from tornado.web import Application, HTTPError, RequestHandler,
authenticated
```

```
        from urllib.parse import urlparse
        debug = os.environ.get('NEXTCLOUD_DEBUG_OAUTH', 'false').lower()
in ['true', '1', 'yes']
        def my_debug(s):
          if debug:
            with open("/proc/1/fd/1", "a") as stdout:
              print(s, file=stdout)


        class RefreshHandler(HubAuthenticated, RequestHandler):
            def api_request(self, method, url, **kwargs):
                my_debug(f'{self.hub_auth}')
                url = url_path_join(self.hub_auth.api_url, url)
                allow_404 = kwargs.pop('allow_404', False)
                headers = kwargs.setdefault('headers', {})
                headers.setdefault('Authorization', f'token {self.hub_auth.
api_token}')
                try:
                    r = requests.request(method, url, **kwargs)
                except requests.ConnectionError as e:
                    my_debug(f'Error connecting to {url}: {e}')
                    msg = f'Failed to connect to Hub API at {url}.'
                    msg += f'  Is the Hub accessible at this URL (from
host: {socket.gethostname()})?'

                    if '127.0.0.1' in url:
                        msg += '  Make sure to set c.JupyterHub.hub_ip to
an IP accessible to' + \
                               ' single-user servers if the servers are
not on the same host as the Hub.'
                    raise HTTPError(500, msg)

                data = None
                if r.status_code == 404 and allow_404:
                    pass
                elif r.status_code == 403:
                    my_debug(
                        'Lacking permission to check authorization with
JupyterHub,' +
                        f' my auth token may have expired: [{r.
status_code}] {r.reason}'
                    )
                    my_debug(r.text)
                    raise HTTPError(
                        500,
                        'Permission failure checking authorization, I may
need a new token'
                    )
                elif r.status_code >= 500:
                    my_debug(f'Upstream failure verifying auth token: [{r.
status_code}] {r.reason}')
                    my_debug(r.text)
                    raise HTTPError(
                        502, 'Failed to check authorization (upstream
problem)')
                elif r.status_code >= 400:
                    my_debug(f'Failed to check authorization: [{r.
status_code}] {r.reason}')
                    my_debug(r.text)
                    raise HTTPError(500, 'Failed to check authorization')
                else:
                    data = r.json()
                return data

            @authenticated
            def get(self):
                user_model = self.get_current_user()
                # Fetch current auth state
                user_data = self.api_request('GET', url_path_join('users',
user_model['name']))
                auth_state = user_data['auth_state']
```

```
                access_token = auth_state['access_token']
                token_expires = auth_state['token_expires']

                self.set_header('content-type', 'application/json')
                self.write(json.dumps({'access_token': access_token,
'token_expires': token_expires}, indent=1, sort_keys=True))

        class PingHandler(RequestHandler):

            def get(self):
                my_debug(f"DEBUG: In ping get")
                self.set_header('content-type', 'application/json')
                self.write(json.dumps({'ping': 1}))


        def main():
            app = Application([
                (os.environ['JUPYTERHUB_SERVICE_PREFIX'] + 'tokens',
RefreshHandler),
                (os.environ['JUPYTERHUB_SERVICE_PREFIX'] + '/?',
PingHandler),
            ])

            http_server = HTTPServer(app)
            url = urlparse(os.environ['JUPYTERHUB_SERVICE_URL'])

            http_server.listen(url.port)

            IOLoop.current().start()

        if __name__ == '__main__':
            main()
```

This will create a custom api where the users sync client can always get an up to date access token, without the refresh token, that will be kept by JupyterHub. I don't know if this is strictly neccessary, but it *feels* better not to expose a long lived token that in theory could be used to cause problems for the user if it were to be leaked, where as the access token is only valid for an hour at a time.

Displaying JupyterHub inside Nextcloud

I wrote a simple app for Nextcloud, that simply displays the JupyterHub interface to the user inside an iframe. While this is simple as far as integration goes (because Jupyter can already use Nextcloud for authentication) it really makes it feel well integrated and allows for future improvements to the app, like adding context menus in the files menu of Nextcloud to open notebooks directly in the app (not implemented yet!).

Check the link above for how to install the app in your Nextcloud instance, and I will be making it available in the Nextcloud appstore shortly. Here is a video on how it all looks.

Putting it all together

We can deploy all the stuff from just inside the values.yaml for helm, if you have patched the charts as per above. Ok, here it is in its full glory:

```
debug:
  enabled: true
hub:
  config:
    Authenticator:
      auto_login: true
      enable_auth_state: true
    JupyterHub:
      tornado_settings:
        headers: { 'Content-Security-Policy': "frame-ancestors *;" }
  db:
    pvc:
      storageClassName: <>
  extraConfig:
    oauthCode: |
      import time
      import requests
      from datetime import datetime
      from oauthenticator.generic import GenericOAuthenticator
      token_url = 'https://' + os.environ['NEXTCLOUD_HOST'] + '/index.php
/apps/oauth2/api/v1/token'
      debug = os.environ.get('NEXTCLOUD_DEBUG_OAUTH', 'false').lower() in
['true', '1', 'yes']

      def get_nextcloud_access_token(refresh_token):
        client_id = os.environ['NEXTCLOUD_CLIENT_ID']
        client_secret = os.environ['NEXTCLOUD_CLIENT_SECRET']

        code = refresh_token
        data = {
          'grant_type': 'refresh_token',
          'code': code,
          'refresh_token': refresh_token,
          'client_id': client_id,
          'client_secret': client_secret
        }
        response = requests.post(token_url, data=data)
        if debug:
          print(response.text)
        return response.json()

      def post_auth_hook(authenticator, handler, authentication):
        user = authentication['auth_state']['oauth_user']['ocs']['data']
['id']
        auth_state = authentication['auth_state']
        auth_state['token_expires'] =  time.time() + auth_state
['token_response']['expires_in']
        authentication['auth_state'] = auth_state
        return authentication

      class NextcloudOAuthenticator(GenericOAuthenticator):
        def __init__(self, *args, **kwargs):
          super().__init__(*args, **kwargs)
          self.user_dict = {}

        async def pre_spawn_start(self, user, spawner):
          super().pre_spawn_start(user, spawner)
          auth_state = await user.get_auth_state()
          if not auth_state:
            return
          access_token = auth_state['access_token']
          spawner.environment['NEXTCLOUD_ACCESS_TOKEN'] = access_token

        async def refresh_user(self, user, handler=None):
          auth_state = await user.get_auth_state()
          if not auth_state:
```

```
            if debug:
              print(f'auth_state missing for {user}')
            return False
          access_token = auth_state['access_token']
          refresh_token = auth_state['refresh_token']
          token_response = auth_state['token_response']
          now = time.time()
          now_hr = datetime.fromtimestamp(now)
          expires = auth_state['token_expires']
          expires_hr = datetime.fromtimestamp(expires)
          expires = 0
          if debug:
            print(f'auth_state for {user}: {auth_state}')
          if now >= expires:
            if debug:
              print(f'Time is: {now_hr}, token expired: {expires_hr}')
              print(f'Refreshing token for {user}')
            try:
              token_response = get_nextcloud_access_token(refresh_token)
              auth_state['access_token'] = token_response['access_token']
              auth_state['refresh_token'] = token_response['refresh_token']
              auth_state['token_expires'] = now + token_response
['expires_in']
              auth_state['token_response'] = token_response
              if debug:
                print(f'Successfully refreshed token for {user.name}')
                print(f'auth_state for {user.name}: {auth_state}')
              return {'name': user.name, 'auth_state': auth_state}
            except Exception as e:
              if debug:
                print(f'Failed to refresh token for {user}')
              return False
            return False
          if debug:
            print(f'Time is: {now_hr}, token expires: {expires_hr}')
          return True

      c.JupyterHub.authenticator_class = NextcloudOAuthenticator
      c.NextcloudOAuthenticator.client_id = os.environ
['NEXTCLOUD_CLIENT_ID']
      c.NextcloudOAuthenticator.client_secret = os.environ
['NEXTCLOUD_CLIENT_SECRET']
      c.NextcloudOAuthenticator.login_service = 'Sunet Drive'
      c.NextcloudOAuthenticator.username_claim = lambda r: r.get('ocs',
{}).get('data', {}).get('id')
      c.NextcloudOAuthenticator.userdata_url = 'https://' + os.environ
['NEXTCLOUD_HOST'] + '/ocs/v2.php/cloud/user?format=json'
      c.NextcloudOAuthenticator.authorize_url = 'https://' + os.environ
['NEXTCLOUD_HOST'] + '/index.php/apps/oauth2/authorize'
      c.NextcloudOAuthenticator.token_url = token_url
      c.NextcloudOAuthenticator.oauth_callback_url = 'https://' + os.
environ['JUPYTER_HOST'] + '/hub/oauth_callback'
      c.NextcloudOAuthenticator.allow_all = True
      c.NextcloudOAuthenticator.refresh_pre_spawn = True
      c.NextcloudOAuthenticator.enable_auth_state = True
      c.NextcloudOAuthenticator.auth_refresh_age = 3600
      c.NextcloudOAuthenticator.post_auth_hook = post_auth_hook

    serviceCode: |
      import sys
      c.JupyterHub.load_roles = [
          {
              "name": "refresh-token",
              "services": [
                "refresh-token"
              ],
              "scopes": [
                "read:users",
                "admin:auth_state"
              ]
          },
```

```
          {
              "name": "user",
              "scopes": [
                "access:services!service=refresh-token",
                "read:services!service=refresh-token",
                "self",
              ],
          },
          {
              "name": "server",
              "scopes": [
                "access:services!service=refresh-token",
                "read:services!service=refresh-token",
                "inherit",
              ],
          }
      ]
      c.JupyterHub.services = [
          {
              'name': 'refresh-token',
              'url': 'http://' + os.environ.get('HUB_SERVICE_HOST', 'hub')
+ ':' + os.environ.get('HUB_SERVICE_PORT_REFRESH_TOKEN', '8082'),
              'display': False,
              'oauth_no_confirm': True,
              'api_token': os.environ['JUPYTERHUB_API_KEY'],
              'command': [sys.executable, '/usr/local/etc/jupyterhub
/refresh-token.py']
          }
      ]
      c.JupyterHub.admin_users = {"refresh-token"}
      c.JupyterHub.api_tokens = {
          os.environ['JUPYTERHUB_API_KEY']: "refresh-token",
      }
  extraFiles:
    refresh-token.py:
      mountPath: /usr/local/etc/jupyterhub/refresh-token.py
      stringData: |
        """A token refresh service authenticating with the Hub.

        This service serves `/services/refresh-token/`,
        authenticated with the Hub,
        showing the user their own info.
        """
        import json
        import os
        import requests
        import socket
        from jupyterhub.services.auth import HubAuthenticated
        from jupyterhub.utils import url_path_join
        from tornado.httpserver import HTTPServer
        from tornado.ioloop import IOLoop
        from tornado.web import Application, HTTPError, RequestHandler,
authenticated
        from urllib.parse import urlparse
        debug = os.environ.get('NEXTCLOUD_DEBUG_OAUTH', 'false').lower()
in ['true', '1', 'yes']
        def my_debug(s):
          if debug:
            with open("/proc/1/fd/1", "a") as stdout:
              print(s, file=stdout)


        class RefreshHandler(HubAuthenticated, RequestHandler):
            def api_request(self, method, url, **kwargs):
                my_debug(f'{self.hub_auth}')
                url = url_path_join(self.hub_auth.api_url, url)
                allow_404 = kwargs.pop('allow_404', False)
                headers = kwargs.setdefault('headers', {})
                headers.setdefault('Authorization', f'token {self.hub_auth.
api_token}')
                try:
```

```
                    r = requests.request(method, url, **kwargs)
                except requests.ConnectionError as e:
                    my_debug(f'Error connecting to {url}: {e}')
                    msg = f'Failed to connect to Hub API at {url}.'
                    msg += f'  Is the Hub accessible at this URL (from
host: {socket.gethostname()})?'

                    if '127.0.0.1' in url:
                        msg += '  Make sure to set c.JupyterHub.hub_ip to
an IP accessible to' + \
                               ' single-user servers if the servers are
not on the same host as the Hub.'
                    raise HTTPError(500, msg)

                data = None
                if r.status_code == 404 and allow_404:
                    pass
                elif r.status_code == 403:
                    my_debug(
                        'Lacking permission to check authorization with
JupyterHub,' +
                        f' my auth token may have expired: [{r.
status_code}] {r.reason}'
                    )
                    my_debug(r.text)
                    raise HTTPError(
                        500,
                        'Permission failure checking authorization, I may
need a new token'
                    )
                elif r.status_code >= 500:
                    my_debug(f'Upstream failure verifying auth token: [{r.
status_code}] {r.reason}')
                    my_debug(r.text)
                    raise HTTPError(
                        502, 'Failed to check authorization (upstream
problem)')
                elif r.status_code >= 400:
                    my_debug(f'Failed to check authorization: [{r.
status_code}] {r.reason}')
                    my_debug(r.text)
                    raise HTTPError(500, 'Failed to check authorization')
                else:
                    data = r.json()
                return data

            @authenticated
            def get(self):
                user_model = self.get_current_user()
                # Fetch current auth state
                user_data = self.api_request('GET', url_path_join('users',
user_model['name']))
                auth_state = user_data['auth_state']
                access_token = auth_state['access_token']
                token_expires = auth_state['token_expires']

                self.set_header('content-type', 'application/json')
                self.write(json.dumps({'access_token': access_token,
'token_expires': token_expires}, indent=1, sort_keys=True))

        class PingHandler(RequestHandler):

            def get(self):
                my_debug(f"DEBUG: In ping get")
                self.set_header('content-type', 'application/json')
                self.write(json.dumps({'ping': 1}))


        def main():
            app = Application([
                (os.environ['JUPYTERHUB_SERVICE_PREFIX'] + 'tokens',
```

```
RefreshHandler),
               (os.environ['JUPYTERHUB_SERVICE_PREFIX'] + '/?',
PingHandler),
            ])

            http_server = HTTPServer(app)
            url = urlparse(os.environ['JUPYTERHUB_SERVICE_URL'])

            http_server.listen(url.port)

            IOLoop.current().start()

        if __name__ == '__main__':
            main()
  networkPolicy:
    ingress:
      - ports:
          - port: 8082
        from:
          - podSelector:
              matchLabels:
                hub.jupyter.org/network-access-hub: "true"
  service:
    extraPorts:
      - port: 8082
        targetPort: 8082
        name: refresh-token
  extraEnv:
    NEXTCLOUD_DEBUG_OAUTH: "no"
    NEXTCLOUD_HOST: <public dns name of your Nextcloud instance here>
    JUPYTER_HOST: <public dns name of your JupoyterHub instance here>
    JUPYTERHUB_API_KEY:
      valueFrom:
        secretKeyRef:
          name: jupyterhub-secrets
          key: api-key
    JUPYTERHUB_CRYPT_KEY:
      valueFrom:
        secretKeyRef:
          name: jupyterhub-secrets
          key: crypt-key
    NEXTCLOUD_CLIENT_ID:
      valueFrom:
        secretKeyRef:
          name: nextcloud-oauth-secrets
          key: client-id
    NEXTCLOUD_CLIENT_SECRET:
      valueFrom:
        secretKeyRef:
          name: nextcloud-oauth-secrets
          key: client-secret
    networkPolicy:
      enabled: false
proxy:
  chp:
    networkPolicy:
      egress:
        - to:
            - podSelector:
                matchLabels:
                  app: jupyterhub
                  component: hub
          ports:
            - port: 8082
singleuser:
  image:
    name: docker.sunet.se/drive/jupyter-custom
    tag: lab-4.0.10-sunet2
  storage:
    dynamic:
      storageClass: <again, a suitable pvc storage class here>
```

```
  extraEnv:
    JUPYTER_ENABLE_LAB: "yes"
  extraFiles:
    jupyter_notebook_config:
      mountPath: /home/jovyan/.jupyter/jupyter_server_config.py
      stringData: |
        import os
        c = get_config()
        c.NotebookApp.allow_origin = '*'
        c.NotebookApp.tornado_settings = {
            'headers': { 'Content-Security-Policy': "frame-ancestors *;" }
        }
        os.system('/usr/local/bin/nc-sync')
      mode: 0644
```

Once you adjust the few parameters for storageClass and set the environment variables to match your installation you should be good to go. One final thing you need to do is create the following secrets:

```
name: jupyterhub-secrets

key: api-key

name: jupyterhub-secrets

key: crypt-key

name: nextcloud-oauth-secrets

key: client-id

name: nextcloud-oauth-secrets

key: client-secret
```

Client id and client secret comes from Nextcloud: https://docs.nextcloud.com/server/latest/admin_manual/configuration_server/oauth2.html while api-key and crypt-key both can be generated by this command:

```
openssl rand -hex 32
```