`weblogin`: Automates logging into web UIs to access unofficial APIs

Daniel Bosk

KTH EECS
`dbosk@kth.se`

1st December 2022

Abstract

We provide a Python package that can be used to automate logging in to web UIs to access the APIs that the UIs are using.

Contents

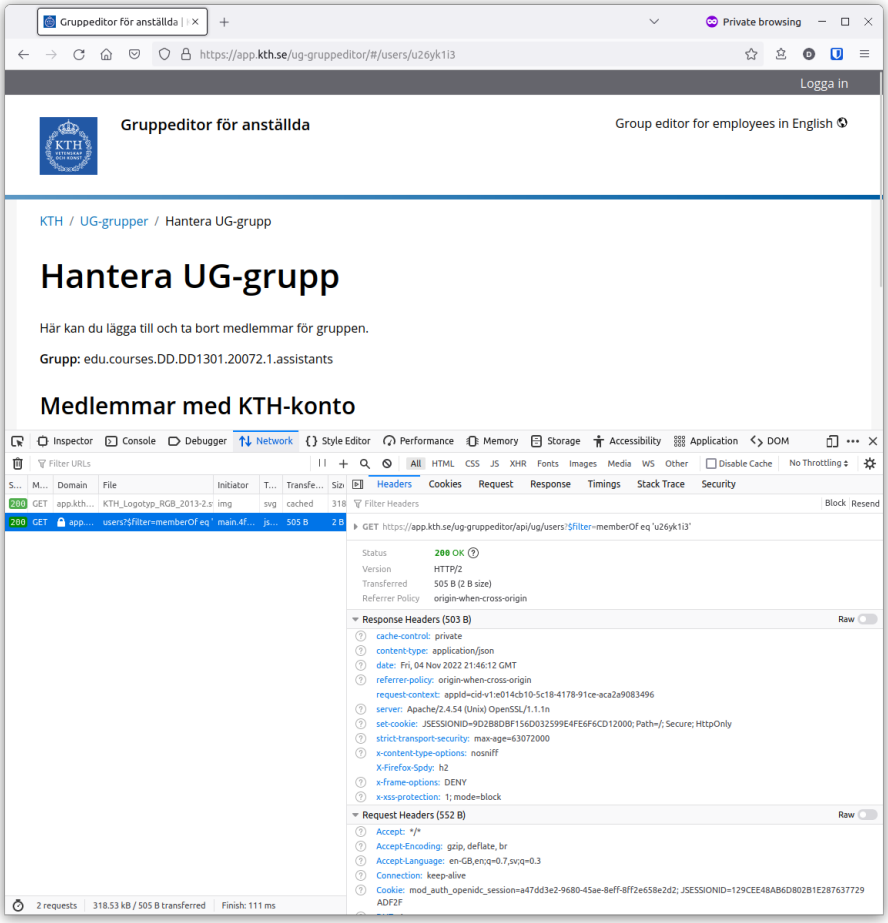

Figure 1: Screenshot of the KTH UG Editor with Firefox's Developer Tools open, showing network requests made.

1 Introduction

We want to use APIs from web UIs that require login. What we want to do is to set up a session, login and then use the API of the page. The reason we want to do this is to use existing APIs. For instance, the user group management service at KTH, see Fig. 1. We can then use the service, track the requests in the browser's developer tools. Then we can simply make the same requests from Python.

For instance, we can redo the request in Fig. 1 like this:

```
1 import weblogin
2 import weblogin.kth
3 import os
4
5 ug = weblogin.AutologinSession([
6                 weblogin.kth.UGlogin(os.environ["KTH_LOGIN"],
```

```
                                        os.environ["KTH_PASSWD"],
                                        "https://app.kth.se/ug-gruppeditor/")
            ])

response = ug.get("https://app.kth.se/ug-gruppeditor/api/ug/users"
                  "?filter=memberOf eq 'u26yk1i3'")
```

The code above will access the API used by the KTH UG group editor service. It will automatically sign in when needed. The API URLs don't trigger a redirect to log in, they just give a 401 unauthorized error. However, we can use the main URL to the UI to trigger such an event, log in and then access the API. All this happens automatically in the background.

The way we do this is to subclass the `requests.Session` class to intercept all requests of a session to check for signs indicating that we must log in. When we detect such sign, we log in and resume as if nothing ever happened.

2 The `AutologinSession` class

Here we document the `weblogin` main module. We provide an abstract class, `weblogin.AutologinSession`, through which we interact with the web page's API. This is a subclass to `requests.Session` and intercepts responses to check for redirects to the login service. If we detect a redirect to the login service, we automatically log in, then return the new response. The logging in will become completely invisible, as if it never happened.

5 ⟨*init.py* 5⟩≡

```
import requests

⟨exceptions 6b⟩
⟨classes 6a⟩

class AutologinSession(requests.Session):
  """
  Maintains an authenticated session to a web system. This class intercepts any
  requests made in a requests.Session and ensures that we log in when
  redirected to the login page.
  """

  def __init__(self, handlers):
    """
    Takes a list of handlers. A handler should derive from AutologinHandler.
    """
    super().__init__()
    self.__handlers = handlers

  def request(self, *args, **kwargs):
    """
    Wrapper around requests.Session.request(...) to check we must log in.
    """
    response = super().request(*args, **kwargs)
```

⟨*log in if necessary* 7a⟩

```
return response
```

2.1 The `AutologinHandler` abstract class

A handler should derive from this class:

6a ⟨*classes* 6a⟩≡ (5)

```
class AutologinHandler:
  """
  An abstract class for a handler for the AutologinSession class.
  """

  def login(self, session, response, args=None, kwargs=None):
    """
    Performs a login based on the response from a request.
    - ‘session‘ is an instance of requests.Session, most likely an instance of
      AutologinSession.
    - ‘response‘ is the response of the latest request.
    - ‘args‘ and ‘kwargs‘ are the options from the latest request, this is so
      that we can redo that request after logging in.

    Raises an AuthenticationError exception if authentication fails.
    """
    raise NotImplementedError()

  def need_login(self, response):
    """
    Checks a response to determine if logging in is needed,
    returns True if needed.
    """
    raise NotImplementedError()
```

2.2 Errors

We must report authentication errors. For this we need specialized exceptions.

6b ⟨*exceptions* 6b⟩≡ (5)

```
class AuthenticationError(Exception):
  pass
```

2.3 Log in if necessary

The login procedure is quite straightforward, we simply call each handler. Each handler will itself keep track of state if they require several steps when logging

in. Each handler will possibly update the response for the next handler.

7a ⟨*log in if necessary* 7a⟩≡ (5)

```
for handler in self.__handlers:
  if handler.need_login(response):
    response = handler.login(self, response, args, kwargs)
```

The benefit of this design comes when trying to log in using an SSO: We first have a handler for the main service and another handler for the SSO.

3 Logging in at KTH

Here we provide the module `weblogin.kth`, which serves as an example of how to write a login handler for use with `weblogin.AutologinSession`.

There are two login handlers needed for KTH.

(1) The SAML handler, which handles the SAML protocol at KTH.

(2) The UG handler, which is the actual password-based authentication server. The SAML service forwards to UG for password authentication.

The module looks like this.

7b ⟨*kth.py* 7b⟩≡

```
from lxml import html
from pprint import pprint
import requests
import urllib.parse
import weblogin
import weblogin.seamlessaccess as sa

⟨UG login handler 7c⟩
⟨SAML login handler 11b⟩
```

See the documentation of the SeamlessAccess module `weblogin.seamlessaccess` in Section 6. We need that to make the `weblogin.kth.SAMLlogin` handler (Section 5) more robust.

4 The UG login handler for KTH

We need a class for KTH that detects logins at KTH. Then we implement the missing methods in the `weblogin.AutologinHandler` class.

7c ⟨*UG login handler* 7c⟩≡ (7b)

```
class UGlogin(weblogin.AutologinHandler):
  """
  Login handler (weblogin.AutologinHandler) for UG logins, i.e. through
  login.ug.kth.se.
  """
  LOGIN_URL = "https://login.ug.kth.se"

  def __init__(self, username, password, login_trigger_url=None,
               rerun_requests=False):
```

```
    """
    Creates a login handler that automatically logs into KTH.
    - Requires username and password.
    - Optional ‘login_trigger_url‘ is a page that redirects to the login page,
      for instance, the API URLs don’t redirect, but the UI URLs do.
    - Optional ‘rerun_requests‘ specifies whether we want to rerun the original
      request that triggered authentication failure.
    """
    super().__init__()
    self.__username = username
    self.__password = password
    self.__login_trigger_url = login_trigger_url
    self.__rerun_requests = rerun_requests
    self.__logging_in = False

  def need_login(self, response):
    """
    Checks a response to determine if logging in is needed,
    returns True if needed
    """
    if self.__logging_in:
      return False

    ⟨check if we’re redirected to login server 9a⟩

  def login(self, session, response, args=[], kwargs={}):
    """
    Performs a login based on the response ‘response‘ from a request to session
    ‘session‘.
    ‘args‘ and ‘kwargs‘ are the options from the request triggering the login
    procedure, this is so that we can redo that request after logging in.

    Raises an AuthenticationError exception if authentication fails.
    """
    print(f"UGlogin: {response.request.method} {response.url}")
    self.__logging_in = True
    ⟨log in to login server, produce new response 9b⟩
    self.__logging_in = False

    if self.__rerun_requests and args:
      print(f"UGlogin rerun request: {args} {kwargs}")
      return session.request(*args, **kwargs)
    return final_response
```

We note that while we’re logging in, we don’t want those requests interrupted by another login session. Hence, we block any new login procedures from starting by setting `self.__logging_in`.

4.1 Check if we need to log in at KTH

There are two cases:

(1) We get a 401 unauthorized for a `kth.se` URL.

(2) We get redirected to the login server.

Thus, we can detect this with either the return code or if the URL starts with the URL to the login service, then we need to log in. However, as noted above, this only applies if we're not already underway with a login.

9a ⟨*check if we're redirected to login server* 9a⟩≡ (7c)

```
if response.status_code == requests.codes.unauthorized and \
          "kth.se" in response.url:
  self.__rerun_requests = True
  return True
elif response.url.find(self.LOGIN_URL) == 0:
  return True

return False
```

4.2 Log in at KTH

If we need to log in, we have three cases:

(1) Either we got a 401 response, because we called an API URL without being authenticated.

(2) Or, we tried to access a UI URL which redirected to the login page.

(3) Or, we used SAML and were redirected to the login server to handle a SAML request. (This will later result in Item (2).)

If we got the first case (Item (1)), we called an API URL which gave a 401 unauthorized response, we want to use the `login_trigger_url` to get into the second case (Item (2)). In the second case, response will contain the login page. We just need to fill the login form and submit it. In the third case (Item (3)), we just want to continue the redirect.

9b ⟨*log in to login server, produce new response* 9b⟩≡ (7c)

```
if response.status_code == requests.codes.unauthorized:
  ⟨trigger redirect to login page 9c⟩
else:
  ⟨parse login page, post login form, or handle SAML request 10a⟩
```

In the case where we just get the unauthorized, we have to simulate a redirect. We do this using the `login_trigger_url`. We make a request to `login_trigger_url`, this will trigger a redirect and we can log in using the method above. When we've done that, we must re-run the original request (the one that didn't trigger a redirect) and return the reply from the new request.

9c ⟨*trigger redirect to login page* 9c⟩≡ (9b)

```
trigger_response = session.get(self.__login_trigger_url)
login_response = self.login(session, trigger_response)
```

Now, the remaining case, we must distinguish between two cases. Fortunately, this is quite simple: If the login form is present, we fill it. Otherwise we assume it's the SAML case.

10a ⟨*parse login page, post login form, or handle SAML request* 10a⟩≡ (9b)

```
doc_tree = html.fromstring(response.text)
login_form = doc_tree.xpath("//form[@id='loginForm']")
if len(login_form) < 1:
  ⟨handle the SAML request to UG 10c⟩
else:
  login_form = login_form[0]
  ⟨handle UG login procedure 10b⟩
```

Let's start with the simple case: parsing the login page and posting the login form. This will contain the necessary redirects, so we just have to do it and then the final response will redirect to the original page we were after. This means that we can simply return the

10b ⟨*handle UG login procedure* 10b⟩≡ (10a)

```
data = {}

for variable in login_form.xpath("//input"):
  if variable.value:
    data[variable.name] = variable.value

data["UserName"] = self.__username if "@ug.kth.se" in self.__username \
                                   else self.__username + "@ug.kth.se"
data["Password"] = self.__password
data["Kmsi"] = True

login_response = session.request(
  login_form.method, f"{self.LOGIN_URL}/{login_form.action}",
  data=data)

if login_response.status_code != requests.codes.ok:
  raise weblogin.AuthenticationError(
    f"authentication as {self.__username} to {login_response.url} failed: "
    f"{login_response.text}")

final_response = login_response
```

To handle the SAML request, we simply find the form, read all variables and post the newly formed request.

10c ⟨*handle the SAML request to UG* 10c⟩≡ (10a)

```
form = doc_tree.xpath("//form")[0]

data = {}
for variable in form.xpath("//input"):
  if variable.name:
    data[variable.name] = variable.value or ""

action_url = urllib.parse.urljoin(response.url, form.action)
```

```
  saml_response = session.request(form.method, action_url, data=data)

  if saml_response.status_code != requests.codes.ok:
    raise weblogin.AuthenticationError(
                        f"SAML error: not OK response: {saml_response.text}")

  final_response = saml_response
```

4.3 Tests

It's hard to test the functionality. We basically want to test if the API on the other side still works. We can use the `app.kth.se/ug-gruppeditor` to test the two different behaviours. The first requests the UI, which redirects automatically. The second tests the API, which doesn't redirect the request, but we must do the redirect ourselves using the `login_trigger_url`.

The two tests follow.

11a ⟨*test kth.py* 11a⟩≡

```
from weblogin import AutologinSession
from weblogin.kth import UGlogin
import os
import requests

def test_get_ui():
  ug = AutologinSession([
    UGlogin(os.environ["KTH_LOGIN"], os.environ["KTH_PASSWD"],
            "https://app.kth.se/ug-gruppeditor/")
    ])
  response = ug.get("https://app.kth.se/ug-gruppeditor/")
  assert response.status_code == requests.codes.ok and \
    response.url.find("https://app.kth.se") == 0

def test_get_api():
  ug = AutologinSession([
    UGlogin(os.environ["KTH_LOGIN"], os.environ["KTH_PASSWD"],
            "https://app.kth.se/ug-gruppeditor/")
    ])
  response = ug.get("https://app.kth.se/ug-gruppeditor/api/ug/groups"
                    "?editableBySelf=true")
  assert response.status_code == requests.codes.ok and response.json()
```

5 The SAML login handler for KTH

The SAML handler is quite simple. It requires no login credentials, it just ensures that we forward requests correctly.

11b ⟨*SAML login handler* 11b⟩≡ (7b)

```
class SAMLlogin(weblogin.AutologinHandler):
  """
```

```
    Login handler (weblogin.AutologinHandler) for SAML at KTH. This will relay to
    UG (login.ug.kth.se) which handles the password-based authentication.
    """

    def __init__(self):
      """
      Creates a login handler that automatically handles the SAML requests used
      at KTH.
      """
      super().__init__()
      self.__logging_in = False

    def need_login(self, response, ignore_logging_in=False):
      """
      Checks a response to determine if we should handle a request.
      Returns True if needed.
      """
      if self.__logging_in:
        return False

      ⟨check if we're redirected to SAML server 12⟩

    def login(self, session, response, args=[], kwargs={}):
      """
      - Performs an action based on the response ‘response‘ from a request to
        session ‘session‘.
      - ‘args‘ and ‘kwargs‘ are the options from the request triggering the login
        procedure, this is so that we can redo that request after logging in.
      - Raises an AuthenticationError exception on fails.
      """
      print(f"SAMLlogin: {response.request.method} {response.url}")
      self.__logging_in = True
      ⟨run SAML procedure, produce new response 13a⟩
      self.__logging_in = False

      return saml_response
```

5.1 Check if we're redirected to SAML server

The SAML URL gotten from SeamlessAccess.org is[1] `saml.sys.kth.se`, but the reverse lookup resolves to `saml-5.sys.kth.se`. So we will simply check if the domain contains both `saml` and `sys.kth.se`.

12 ⟨*check if we're redirected to SAML server* 12⟩≡ (11b)

```
return "saml" in response.url and "sys.kth.se" in response.url
```

[1] As of 2022-11-30.

5.2 Run KTH's SAML procedure

On the first stop in the sequence (contained in `response`), there is a form that must be posted. A browser uses JavaScript to do this, we must do it manually: Find the form, fetch all values from the input tags, then post the form. All in all, we want to produce a new response from the server, `saml_response` from above.

We start by extracting any form. If there is no form, we simply proceed without doing anything.

13a ⟨*run SAML procedure, produce new response* 13a⟩≡ (11b)

```
doc_tree = html.fromstring(response.text)
try:
  form = doc_tree.xpath("//form")[0]
except IndexError:
  ⟨don't handle this request 13b⟩
else:
  ⟨extract form data and action 13c⟩
  ⟨post form action 13d⟩
```

Now, it's easy to ignore this request.

13b ⟨*don't handle this request* 13b⟩≡ (13a)

```
pass
```

To extract the data, we simply traverse the inputs available in the form.

13c ⟨*extract form data and action* 13c⟩≡ (13a)

```
data = {}
for variable in form.xpath("//input"):
  if variable.name:
    data[variable.name] = variable.value or ""

action_url = urllib.parse.urljoin(response.url, form.action)
print(f"action_url = {action_url}")
```

Now here is the trick: The form data extraction and submission above (⟨*extract the form data and action* (never defined)⟩) is so general, that it can handle each of the several steps. However, we can't pretend that we are done and set `self.__logging_in = False`, this will result in infinite recursion. (The requests will be intercepted before they happen, so we will never proceed.) But we can recursively call this handler again and again until done.

13d ⟨*post form action* 13d⟩≡ (13a)

```
saml_response = session.request(form.method, action_url, data=data)

if saml_response.status_code != requests.codes.ok:
  raise weblogin.AuthenticationError(
                      f"SAML error: not OK response: {saml_response.text}")
```

6 Looking up IDPs through SeamlessAccess.org

Here we provide the module `weblogin.seamlessaccess`.

There is a global service, SeamlessAccess.org, that provides SSO login services for academic institutions. Essentially, what it does is to provide a look-up service for academic identity providers. We will provide two functions using SeamlessAccess.org:

(1) Search for an institution by name.

(2) Look up an institution by unique ID.

The module is structured as follows.

14a ⟨*seamlessaccess.py* 14a⟩≡

```
import requests

⟨functions 14c⟩
```

We will test these functions using the following test script.

14b ⟨*test seamlessaccess.py* 14b⟩≡

```
from weblogin.seamlessaccess import *

⟨test functions 14d⟩
```

6.1 Look up an institution by unique ID

We can look up an institution by unique ID (what appears to be a SHA1 hash). This is simply a GET request resulting in a JSON-formatted response.

14c ⟨*functions* 14c⟩≡ (14a) 15a▷

```
def get_entity_data_by_id(id):
  """
  Requests entity data from SeamlessAccess.org for entity with unique ID ‘id‘.
  Returns JSON (dictionary) containing data.
  """
  response = requests.get(f"https://md.seamlessaccess.org/entities/{id}.json")
  try:
    return response.json()
  except:
    raise Exception(f"invalid response from SeamlessAccess.org for ID {id}")
```

We can test this functions as follows:

14d ⟨*test functions* 14d⟩≡ (14b) 15b▷

```
def test_get_entity_data_by_id():
  data = get_entity_data_by_id("{sha1}e26e5d098f073536d0351577c98c83825f0f922c")
  assert data["id"] == "{sha1}e26e5d098f073536d0351577c98c83825f0f922c"
```

The resulting JSON data for this particular ID is:

```
{
  "title": "KTH Royal Institute of Technology",
  "descr": "Identity Provider for KTH",
  "title_langs": {
    "en": "KTH Royal Institute of Technology",
    "sv": "Kungliga Tekniska h\u00f6gskolan (KTH)"
  },
```

```
  "descr_langs": {
    "en": "Identity Provider for KTH",
    "sv": "Identity Provider f\u00f6r KTH"
  },
  "auth": "saml",
  "entity_id": "https://saml.sys.kth.se/idp/shibboleth",
  "entityID": "https://saml.sys.kth.se/idp/shibboleth",
  "type": "idp",
  "hidden": "false",
  "scope": "kth.se",
  "domain": "kth.se",
  "name_tag": "KTH",
  "entity_icon_url": {
    "url": "https://saml-5.sys.kth.se/idp/images/logo.png",
    "width": "225",
    "height": "225"
  },
  "keywords": "stockholm",
  "privacy_statement_url": "https://intra.kth.se/en/it/natverk/regler-policys/policy-for-h
  "geo": {
    "lat": "9.34698",
    "long": "18.07213"
  },
  "id": "{sha1}e26e5d098f073536d0351577c98c83825f0f922c"
}
```

6.2 Searching for an institution by name

We can also search for an institution by name. This yields a list of JSON objects as in Section 6.1.

15a ⟨*functions* 14c⟩+≡ (14a) ◁14c

```
def find_entity_data_by_name(name):
  """
  Searches SeamlessAccess.org for an institution by name ‘name‘.
  Returns a list of institutions’ data.
  """
  response = requests.get(f"https://md.seamlessaccess.org/entities/?q={name}")
  try:
    return response.json()
  except:
    raise Exception(f"invalid response from SeamlessAccess.org for name {name}")
```

We can test this functions as follows:

15b ⟨*test functions* 14d⟩+≡ (14b) ◁14d

```
def test_find_entity_data_by_name():
  data = find_entity_data_by_name("KTH")
  assert data[0]["id"] == "{sha1}e26e5d098f073536d0351577c98c83825f0f922c"
```

The resulting JSON data for this particular ID is the following. Note that the difference compared to the data in Section 6.1 is that this is one element in a list.

```
[
  {
    "title": "KTH Royal Institute of Technology",
    "descr": "Identity Provider for KTH",
    "title_langs": {
      "en": "KTH Royal Institute of Technology",
      "sv": "Kungliga Tekniska h\u00f6gskolan (KTH)"
    },
    "descr_langs": {
      "en": "Identity Provider for KTH",
      "sv": "Identity Provider f\u00f6r KTH"
    },
    "auth": "saml",
    "entity_id": "https://saml.sys.kth.se/idp/shibboleth",
    "entityID": "https://saml.sys.kth.se/idp/shibboleth",
    "type": "idp",
    "hidden": "false",
    "scope": "kth.se",
    "domain": "kth.se",
    "name_tag": "KTH",
    "entity_icon_url": {
      "url": "https://saml-5.sys.kth.se/idp/images/logo.png",
      "width": "225",
      "height": "225"
    },
    "keywords": "stockholm",
    "privacy_statement_url": "https://intra.kth.se/en/it/natverk/regler-policys/policy-for
    "geo": {
      "lat": "9.34698",
      "long": "18.07213"
    },
    "id": "{sha1}e26e5d098f073536d0351577c98c83825f0f922c"
  }
]
```

7 Logging in to LADOK

Here we provide the module `weblogin.ladok`, which serves as a more complex example of how to write a login handler for use with `weblogin.AutologinSession`.

We will create a login handler for LADOK. As LADOK supports logging in using SSO from all Swedish universities, we will use this handler in conjunction with a login handler for a university (for instance the ones for KTH in Sections 4 and 5).

We need a class for LADOK that detects logins to LADOK. Then we implement the missing methods in the `weblogin.AutologinHandler` class.

16 ⟨*ladok.py* 16⟩≡

```
from lxml import html
import requests
import sys
```

```
import weblogin
import weblogin.seamlessaccess as sa
import urllib.parse

sys.setrecursionlimit(100)

class SSOlogin(weblogin.AutologinHandler):
  """
  Login handler (weblogin.AutologinHandler) for LADOK logins.
  """
  LOGIN_URL = "https://www.start.ladok.se/gui/loggain"

  def __init__(self,
      institution,
      login_trigger_url="https://www.start.ladok.se/gui/"):
    """
    Creates a login handler that automates the LADOK part of authentication.
    - Requires ‘institution‘. A string identifying the instututionat
      SeamlessAccess.org.
    - Optional ‘login_trigger_url‘ is a page that redirects to the login page,
      for instance, the API URLs don’t redirect, but the UI URLs do.

    This login handler must be used in conjunction with a university login
    handler.
    """
    super().__init__()
    self.__institution = institution
    self.__login_trigger_url = login_trigger_url
    self.__logging_in = False

  def need_login(self, response):
    """
    Checks a response to determine if logging in is needed,
    returns True if needed
    """
    ⟨check if we’re redirected to login server 19a⟩

  def login(self, session, response, args=None, kwargs=None):
    """
    Performs a login based on the response ‘response‘ from a request to session
    ‘session‘.
    ‘args‘ and ‘kwargs‘ are the options from the request triggering the login
    procedure, this is so that we can redo that request after logging in.

    Raises an AuthenticationError exception if authentication fails.
    """
    print(f"SSOlogin: {response.request.method} {response.url}")
    self.__logging_in = True
    ⟨log in to login server 19b⟩
    self.__logging_in = False
```

```
    if args:
      return session.request(*args, **kwargs)
    return ladok_response
```

We note that while we're logging in, we don't want those requests interrupted by another login session. Hence, we block any new login procedures from starting by setting `self.__logging_in`.

7.1 Tests and intended usage

We will test this by a simple API call to LADOK. We will get the record of the currently logged in user from LADOK. We will use author's institution for this test, which means only the author can run this test.

Now this illustrates that this login handler only fulfills a small part of the login procedure. We need two more handlers in this case, one for the SAML implementation (`weblogin.kth.SAMLlogin` below) and one for the actual logging in at the institution (`weblogin.kth.UGlogin` below).

18 ⟨*test ladok.py* 18⟩≡

```
from weblogin import AutologinSession
from weblogin import kth, ladok
import os
import requests

def test_get_user_info():
  ls = AutologinSession([
      ladok.SSOlogin("KTH Royal Institute of Technology"),
      kth.SAMLlogin(),
      kth.UGlogin(os.environ["KTH_LOGIN"], os.environ["KTH_PASSWD"])
    ])

  headers = {}
  headers["Accept"] =       "application/vnd.ladok-resultat+json"
  headers["Accept"] +=    ", application/vnd.ladok-kataloginformation+json"
  headers["Accept"] +=    ", application/vnd.ladok-extra+json"
  headers["Accept"] +=    ", application/json, text/plain"
  headers["Content-Type"] = "application/vnd.ladok-kataloginformation+json"

  response = ls.get(
    "https://www.start.ladok.se/gui/proxy"
      "/kataloginformation/anvandare/anvandarinformation",
    headers=headers
  )

  assert response.status_code == requests.codes.ok \
     and response.json()["Epost"] == "dbosk@kth.se"
```

7.2 Check if we need to log in to LADOK

There are two cases:

(1) We get a 401 unauthorized for a `ladok.se` URL.

(2) We get redirected to the login server.

Thus, we can detect this with either the return code or if the URL starts with the URL to the login service, then we need to log in. However, as noted above, this only applies if we're not already underway with a login.

19a ⟨*check if we're redirected to login server* 19a⟩≡ (16)

```
if self.__logging_in:
  return False
elif response.status_code == requests.codes.unauthorized \
     and "ladok.se" in response.url:
  return True
elif response.url.find(self.LOGIN_URL) == 0:
  return True

return False
```

7.3 Log in to LADOK

If we need to log in we have a few steps to do:

19b ⟨*log in to login server* 19b⟩≡ (16)

⟨*start ladok login* 19c⟩
⟨*fetch return URL from LADOK's SeamlessAccess request* 19d⟩
⟨*fetch institution SAML URL from SeamlessAccess* 20a⟩
⟨*run return request with added institution SAML URL as entityID* 20b⟩

Once these steps are completed, the institution SAML and login handler should take over.

This takes place in the body of

```
login(self, session, response, args=None, kwargs=None).
```

So those arguments are what we have to work with.

We don't really need the `/gui/loggain` page, we simply start the login with the following URL and it will trigger the steps needed.

19c ⟨*start ladok login* 19c⟩≡ (19b)

```
response = session.get("https://www.start.ladok.se/Shibboleth.sso/Login"
                       "?target=https://www.start.ladok.se/gui/shiblogin")
```

That URL will redirect to SeamlessAccess.org. But instead of using that page, we use the `weblogin.seamlessaccess` module. The aim is to find the URL to the institution's SAML server, which we do later. Since we don't use it, we want to extract the return URL, because that contains some session information for the SAML request later.

19d ⟨*fetch return URL from LADOK's SeamlessAccess request* 19d⟩≡ (19b)

```
parsed_url = urllib.parse.urlparse(response.url, allow_fragments=False)
if "seamlessaccess.org" not in parsed_url.netloc:
  raise weblogin.AuthenticationError(
                  f"seamlessaccess.org not in {parsed_url.netloc}")

return_url = urllib.parse.unquote(
                        urllib.parse.parse_qs(parsed_url.query)["return"][0])
```

Now we actually fetch the SAML server URL, which is called by `entityID`.

20a ⟨*fetch institution SAML URL from SeamlessAccess* 20a⟩≡ (19b)

```
if "{sha1}" in self.__institution:
  entityID = sa.get_entity_data_by_id(self.__institution)["entityID"]
else:
  entityID = sa.find_entity_data_by_name(self.__institution)[0]["entityID"]
```

Finally, we append that to the return URL and make the request.

20b ⟨*run return request with added institution SAML URL as entityID* 20b⟩≡ (19b)

```
if "?" in return_url:
  return_url += f"&entityID={entityID}"
else:
  return_url += f"?entityID={entityID}"

ladok_response = session.get(return_url)
```

We note that we can return the response to this last request. This response will be intercepted by the other login handlers and eventually the real response will be returned to the original request.