pyopenid Documentation

Release 0.1

MetaMetrics

Nov 15, 2017

Contents

Contents:

CHAPTER 1

Designing an OpenID Library for Humans

1.1 Requesting authentication via OpenID

This is an openid request is made using python-openid, the current openid frontrunner in the python community:

```python
from openid.consumer import consumer
from openid.extensions import pape, sreg

from my_web_framework import HttpResponse

def openid_login(request):
    openid_consumer = consumer.get_consumer()
    openid_request = openid_consumer.begin(request.GET['openid_identifier'])
    openid_request.addExtension(sreg.SRegRequest(
        required=['nickname'],
        optional=['fullname', 'email'],
    ))
    openid_request.addExtension(pape.Request([pape.AUTH_PHISHING_RESISTANT]))
    if openid_request.shouldSendRedirect():
        redirect_url = openid_request.redirectURL(
            trust_root='http://example.com/requesting_site',
            return_to='http://example.com/requesting_site/welcome',
        )
        return HttpResponse(status=302, headers={'Location': redirect_url})
    else:
        response = openid_request.htmlMarkup(
            trust_root='http://example.com/requesting_site',
            return_to='http://example.com/requesting_site/welcome',
            form_tag_attrs={'id': 'openid_message'},
        )
        return HttpResponse(status=200, body=response)
```

Problem #1: It's hard. There's no way you can keep this API in your head.

Problem #2: PAPE and SREG may be extensions to the OpenID spec, but as far as the library is concerned, they should be baked in. Nobody cares how the spec is developed. Further OpenID development will be slow enough that new extensions can also be baked in.

Problem #3: (A generalization of #2), the developer is required to know the spec too well. An OpenID provider might take a redirect with the data as GET parameters. Alternatively, it might need form values, submitted in a POST request from an HTML form. As a developer, I don't care which one it needs. I want to pass in my values, and get back an appropriate response that I can send to my user's browser. To make this as framework agnostic as possible, the response is a 3-tuple comprising an HTTP status code, a dictionary of headers, and a string representing the body of the request. The developer can alter the headers or the body, or just pass them along unharmed.

Problem #4: Too many pieces to do basic things. Importing from multiple modules calling many pieces, and dealing with heterogenous, complex return values all add cognitive load to the user. If the user can be An OpenID consumer should only need one import and two method calls (e.g., generate_request, validate).

Problem #5: PEP-8. If you write a python library that ignores PEP-8 for no reason, you are a liability. You are dragging down the language. Stop it.

A cleaner, human-oriented version might look like this:

```python
import pyopenid
from my_web_framework import HttpResponse

def openid_login(request):
    status, headers, body = pyopenid.generate_request(request.GET['openid_identifier
↪'],
        required_fields=['username'],
        optional_fields=['email', 'dob'],
        privacy_policy='http://example.com/requesting_site/privacy/',
        extras={'openid.key': 'value'})
    return HttpResponse(status, headers=headers, body=body)
```

Still need to figure out a human-oriented way to implement PAPE. Basic policies could have simple string names. Probably something resembling:

```python
pyopenid.generate_request(identifier,
    auth_policies=['phishing-resistant', 'multi-factor', 'physical-multi-factor'])
```

All short names would expand to a canonical PAPE URL. All URLs could be specified in full.

The following are identical:

```python
pyopenid.generate_request(identifier,
    auth_policies=['phishing-resistant'])

pyopenid.generate_request(identifier,
    auth_policies=['http://schemas.openid.net/pape/policies/2007/06/phishing-resistant
↪'])
```

1.2 Processing a response from an OpenID provider

What happens when a response comes back from the server?

```python
from openid.consumer import consumer
from openid.extensions import pape, sreg
from my_web_framework import HttpResponse

def process(request):
    """Handle the response from the OpenID server."""

    sreg_resp = None
    pape_resp = None
    oidconsumer = consumer.get_consumer()

    # Ask the library to check the response that the server sent
    # us.  Status is a code indicating the response type. info is
    # either None or a string containing more information about
    # the return type.

    #REVIEW: What URL is this?  What are we using it for?
    url = 'http://'+ request.headers.get('Host') + '/process' #*
    info = oidconsumer.complete(request.query, url)

    identifier = info.getDisplayIdentifier()

    if info.status == consumer.FAILURE and identifier:
        fmt = "Verification of %s failed: %s"
        message = fmt % (identifier, info.message)
    elif info.status == consumer.SUCCESS:
        sreg_resp = sreg.SRegResponse.fromSuccessResponse(info)
        nickname = sreg_resp.get('nickname', None)
        fmt = 'You have successfully verified %s as your identity.'
        message = fmt % identifier
        if nickname:
            message += ' Your nickname is %s' % nickname
        if info.endpoint.canonicalID:
            message += (' This is an i-name, you should use its'
                        ' canonical ID: %s' % info.endpoint.canonicalID)
    elif info.status == consumer.CANCEL:
        message = 'Verification cancelled'
    elif info.status == consumer.SETUP_NEEDED:
        if info.setup_url:
            message = ('Setup needed. Try <a href="%s">non-immediate'
                       ' mode</a> instead</a>' % info.setup_url)
        else:
            message = 'Setup needed'
    else:
        message = 'Verification failed.'

    return HttpResponse(status=200,
        body='''
            <p>Identifier: %(identifier)s</p>
            <p>%(message)s</p><p>''' % {
            'message': message,
            'identifier': identifier,
        }
    )
```

Now the user-friendly version:

```
import pyopenid
from my_web_framework import HttpResponse

def process(request):
    """Handle the response from the OpenID server."""
    openid = pyopenid.validate(request)

    if openid.success:
        # no need to make a separate request for the canonical identifier
        # just use it
        canonical_identifier = openid.data['identifier']

        # if you want the original identifier, it's in there.
        requested_identifier = openid.data['requested_identifier']
        nickname = openid.data.get('nickname', None)
        fmt = 'You have successfully verified %s as your identity.'
        message = fmt % canonical_identifier
        if nickname:
            message += ' Your nickname is %s.' % nickname
    else:
        message = openid.error.message

    return HttpResponse(status=200,
        body='''
            <p>Identifier: %(identifier)s</p>
            <p>%(message)s</p><p>''' % {
            'message': message,
            'identifier': identifier,
        }
    )
```

Again, there is only one import, one method call, and some attributes describing the response. SREG data is not isolated. It is seamlessly contained with the openid identifier in one data dictionary.

PAPE responses could be implemented as an attribute on the openid response:

```
>>> print openid.auth_policies
['multi-factor', 'http://example.com/custom-auth-policy']
```

But we'd want the library to understand both short names and full PAPE URLs.

```
>>> 'multi-factor' in openid.auth_policies
True
>>> 'http://schemas.openid.net/pape/policies/2007/06/multi-factor' in openid.auth_
↪policies
True
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search