django-browserid Documentation

Release 0.11

Mozilla Foundation

September 25, 2014

Contents

1 User Guide **3**
- 1.1 Introduction . . . 3
- 1.2 Quickstart . . . 4
- 1.3 Customization . . . 5
- 1.4 Extras . . . 9
- 1.5 Settings . . . 10
- 1.6 Deploying in Production . . . 12
- 1.7 Upgrading . . . 12
- 1.8 Troubleshooting . . . 13

2 API Documentation **17**
- 2.1 Python API . . . 17
- 2.2 JavaScript API . . . 22

3 Contributor Guide **25**
- 3.1 Contributor Setup . . . 25
- 3.2 Contributing Guidelines . . . 26
- 3.3 Changelog . . . 27
- 3.4 Authors . . . 29

Python Module Index **33**

Release v0.11. (*Quickstart*)

django-browserid is a Python library that integrates BrowserID authentication into Django.

BrowserID is an open, decentralized protocol for authenticating users based on email addresses. django-browserid provides the necessary hooks to get Django to authenticate users via BrowserID. By default, django-browserid relies on Persona for the client-side JavaScript shim and for assertion verification.

django-browserid is tested on Python 2.6 to 3.3 and Django 1.4 to 1.6. See tox.ini for more details.

django-browserid depends on:

- Requests >= 1.0.0
- fancy_tag == 0.2.0
- jQuery >= 1.8 (if you are using `api.js` and `browserid.js`).

django-browserid is a work in progress. Contributions are welcome. Feel free to fork and contribute!

CHAPTER 1

User Guide

1.1 Introduction

1.1.1 How does it work?

At a high level, this is what happens when a user wants to log into a site that uses django-browserid:

1. A user clicks a login button on your web page.
2. The JavaScript shim (hosted by Persona) displays a pop-up asking for the email address the user wants to log in with.
3. If necessary, the pop-up prompts the user for additional info to authenticate them. For example, if the user enters an *@mozilla.com* email, the Mozilla LDAP Identity Provider will prompt them for their LDAP password.
4. The JavaScript receives an "assertion" from the Identity Provider and submits it to the site's backend via AJAX.
5. The backend sends the assertion to the Remote verification service, which verifies the assertion and returns the result, including the email address of the user if verification was successful.
6. The backend finds a user account matching that email (creating it if one isn't found) and logs the user in as that account.
7. The backend returns a URL that the JavaScript redirects the user to.

Note that this is just an example flow. Several of these steps can be customized for your site; for example, you may not want user accounts to be created automatically. This behavior can be changed to suit whatever needs you have.

A detailed explanation of the BrowserID protocol is available on MDN.

1.1.2 Persona

By default, django-browserid relies on Persona, which is a set of BrowserID-related services hosted by Mozilla. It's possible, but annoying, to use django-browserid without these dependencies.

Currently, django-browserid relies on Persona for:

- The Cross-browser API Library, which implements the `navigator.id` API for browsers that don't natively support BrowserID.
- The Fallback Identity Provider for emails from servers that don't support BrowserID.
- The Remote verification service, which handles assertion verification for sites that don't want to verify assertions themselves.

In the future, django-browserid will remove the need to depend on these Mozilla-centric services. Local verification and a self-hosted cross-browser API will greatly reduce the reliance on Mozilla's servers for authentication.

1.2 Quickstart

Follow these instructions to get set up with a basic install of django-browserid:

1.2.1 Installation

You can use pip to install django-browserid and requirements:

```
$ pip install django-browserid
```

1.2.2 Configuration

After installation, you'll need to configure your site to use django-browserid. Start by making the following changes to your `settings.py` file:

```
# Add 'django_browserid' to INSTALLED_APPS.
INSTALLED_APPS = (
    # ...
    'django.contrib.auth',
    'django_browserid',  # Load after auth
    # ...
)

# Add the django_browserid authentication backend.
AUTHENTICATION_BACKENDS = (
   # ...
   'django.contrib.auth.backends.ModelBackend',
   'django_browserid.auth.BrowserIDBackend',
   # ...
)
```

Next, edit your `urls.py` file and add the following:

```
urlpatterns = patterns('',
    # ...
    (r'', include('django_browserid.urls')),
    # ...
)
```

Note: The django-browserid urlconf *must not* have a regex with the include. Use a blank string, as shown above.

Finally, you'll need to add the login button and info tag to your Django templates, along with the CSS and JS files necessary to make it work:

```
{% load browserid %}
<html>
  <head>
    <link rel="stylesheet" href="{% static 'browserid/persona-buttons.css' %}">
  </head>
  <body>
```

```
    {% browserid_info %}
    {% if user.is_authenticated %}
      <p>Current user: {{ user.email }}</p>
      {% browserid_logout text='Logout' %}
    {% else %}
      {% browserid_login text='Login' color='dark' %}
    {% endif %}

    <script src="https://code.jquery.com/jquery-1.9.1.min.js"></script>
    <script src="https://login.persona.org/include.js"></script>
    <script src="{% static 'browserid/api.js' %}"></script>
    <script src="{% static 'browserid/browserid.js' %}"></script>
  </body>
</html>
```

Note: `api.js` and `browserid.js` require jQuery 1.8 or higher.

Note: The `browserid_info` tag is required on any page that users can log in from. It's recommended to put it just below the `<body>` tag.

And that's it! You can now log into your site using Persona!

Once you're ready, you should check out *how to customize django-browserid* to your liking.

1.2.3 Note for Jinja2 / Jingo Users

If you're using Jinja2 via jingo, here's a version of the example above written in Jinja2:

```
<html>
  <head>
    {{ browserid_css() }}
  </head>
  <body>
    {{ browserid_info() }}
    {% if user.is_authenticated() %}
      <p>Current user: {{ user.email }}</p>
      {{ browserid_logout(text='Logout') }}
    {% else %}
      {{ browserid_login(text='Login', color='dark') }}
    {% endif %}

    <script src="https://code.jquery.com/jquery-1.9.1.min.js"></script>
    {{ browserid_js() }}
  </body>
</html>
```

1.3 Customization

Now that you've got django-browserid installed and configured, it's time to see how to customize it to your needs.

1.3.1 Local Assertion Verification

When a user authenticates via django-browserid, they do so by sending your site an assertion, which, when verified, gives you an email address for the user. Normally, this verification is handled by sending the assertion to a verification service hosted by Mozilla.

However, you can also verify assertions locally and avoid relying on the verification service. To do so, you must install PyBrowserID. django-browserid checks for PyBrowserID, and if it is found, it enables the use of the `LocalVerifier` class.

Once you've installed PyBrowserID, add the `LocalBrowserIDBackend` class to your `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = (
    'django_browserid.auth.LocalBrowserIDBackend',
)
```

Note: Because the BrowserID certificate format has not been finalized, PyBrowserID may fail to verify a valid assertion if the format changes. Be aware of the risks before enabling local verification.

1.3.2 Customizing the Verify View

Many common customizations involve overriding methods on the `Verify` class. But how do you use a custom `Verify` subclass?

You can substitute a custom verification view by setting `BROWSERID_VERIFY_CLASS` to the import path for your view:

```
BROWSERID_VERIFY_CLASS = 'project.application.views.MyCustomVerifyClass'
```

1.3.3 Customizing the Authentication Backend

Another common way to customize django-browserid is to subclass `BrowserIDBackend`. To use a custom `BrowserIDBackend` class, simply use the python path to your custom class in the `AUTHENTICATION_BACKENDS` setting instead of the path to `BrowserIDBackend`.

1.3.4 Post-login Response

After logging the user in, the default view redirects the user to `LOGIN_REDIRECT_URL` or `LOGIN_REDIRECT_URL_FAILURE`, depending on if login succeeded or failed. You can modify those settings to change where they are redirected to.

Note: You can use `django.core.urlresolvers.reverse_lazy` to generate a URL for these settings from a URL pattern name or function name.

You can also override the `success_url <django_browserid.views.Verify.success_url` and `failure_url <django_browserid.views.Verify.failure_url` properties on the `Verify` view if you need more control over how the redirect URLs are retrieved.

If you need to control the entire response to the `Verify` view, such as when you're *using custom JavaScript*, you'll want to override `login_success <django_browserid.views.Verify.login_success` and `login_failure <django_browserid.views.Verify.login_failure`.

1.3.5 Automatic User Creation

If a user signs in with an email that doesn't match an existing user, django-browserid automatically creates a new User object for them that is tied to their email address. You can disable this behavior by setting BROWSERID_CREATE_USER to False, which will cause authentication to fail if a user signs in with an unrecognized email address.

If you want to customize how new users are created (perhaps you want to generate a display name for them), you can override the create_user method on BrowserIDBackend:

```
from django_browserid.auth import BrowserIDBackend

class CustomBackend(BrowserIDBackend):
    def create_user(self, email):
        username = my_custom_username_algo()
        return self.User.objects.create_user(username, email)
```

Note: self.User points to the User model defined in AUTH_USER_MODEL for custom User model support. See Custom User Models for more details.

1.3.6 Limiting Authentication

There are two ways to limit who can authenticate with your site: prohibiting certain email addresses, or filtering the queryset that emails are compared to.

filter_users_by_email

filter_users_by_email <django_browserid.auth.BrowserIDBackend.filter_users_by_email returns the queryset that is searched when looking for a user account that matches a user's email. Overriding this allows you to limit the set of users that are searched:

```
from django_browserid.auth import BrowserIDBackend

class CustomBackend(BrowserIDBackend):
    def filter_users_by_email(self, email):
        # Only allow staff users to login.
        return self.User.objects.filter(email=email, is_staff=True)
```

Note: If you customize filter_users_by_email, you should probably make sure that Automatic User Creation is either disabled or customized to only create users that match your limited set.

is_valid_email

is_valid_email <django_browserid.auth.BrowserIDBackend.is_valid_email determines if the email a user attempts to log in with is considered valid. Override this to exclude users with certain emails:

from django_browserid.auth import BrowserIDBackend

class CustomBackend(BrowserIDBackend):

def is_valid_email(self, email): # Ignore users from fakeemails.com return not email.endswith('@fakeemails.com')

1.3.7 Custom User Models

Django allows you to *use a custom User model for authentication <custom_user_model>*. If you are using a custom User model, and the model has an `email` attribute that can store email addresses, django-browserid should work out-of-the-box for you.

If this isn't the case, then you will probably have to override the `is_valid_email <django_browserid.auth.BrowserIDBackend.is_valid_email>`, `filter_users_by_email <django_browserid.auth.BrowserIDBackend.filter_users_by_email>`, and `create_user` methods to work with your custom User class.

1.3.8 Using the JavaScript API

django-browserid comes with two JavaScript files to include in your webpage:

1. `api.js`: An API for triggering logins via BrowserID and verifying assertions via the server.
2. `browserid.js`: A basic example of hooking up links with the JavaScript API.

`browserid.js` only covers basic use cases. If your site has more complex behavior behind trigger login, you should replace `browserid.js` in your templates with your own JavaScript file that uses the django-browserid JavaScript API.

See also:

`JavaScript API` API Documentation for `api.js`.

1.3.9 Django Admin Support

If you want to use BrowserID for login on the built-in Django admin interface, you must use the `django-browserid admin site` instead of the default Django admin site:

```
from django.contrib import admin

from django_browserid.admin import site as browserid_admin

from myapp.foo.models import Bar


class BarAdmin(admin.ModelAdmin):
    pass
browserid_admin.register(Bar, BarAdmin)
```

You must also use the django-browserid admin site in your `urls.py` file:

```
from django.conf.urls import patterns, include, url

# Autodiscover admin.py files in your project.
from django.contrib import admin
admin.autodiscover()

# copy_registry copies ModelAdmins registered with the default site, like
# the built-in Django User model.
from django_browserid.admin import site as browserid_admin
browserid_admin.copy_registry(admin.site)

urlpatterns = patterns('',
    # ...
```

```
    url(r'^admin/', include(browserid_admin.urls)),
)
```

See also:

`django_browserid.admin.BrowserIDAdminSite` API documentation for BrowserIDAdminSite, including how to customize the login page (such as including a normal login alongside BrowserID login).

1.3.10 Alternative Template Languages

By default, django-browserid supports use in Django templates as well as use in Jinja2 templates via the jingo library. Template helpers are registered as helper functions with jingo, so you can use them directly in Jinja2 templates:

```
<div class="authentication">
  {% if user.is_authenticated() %}
    {{ browserid_logout(text='Logout') }}
  {% else %}
    {{ browserid_login(text='Login', color='dark') }}
  {% endif %}
</div>
{{ browserid_js() }}
```

For other libraries or template languages, you will have to register the django-browserid helpers manually. The relevant helper functions can be found in the `django_browserid.helpers` module.

1.4 Extras

django-browserid comes with a few extra pieces to make development easier. They're documented below.

1.4.1 Offline Development

Because django-browsered *relies on the Persona service*, offline development is not supported by default. To work around this, django-browserid includes an auto-login system that lets you specify an email to log the user in with when they click a login button.

> **Warning:** Auto-login is a huge security hole as it bypasses authentication. Only use it for local development on your own computer; **never** use it on a publicly-visible machine or your live, production website.

Enable auto-login

To enable auto-login:

1. Add the `AutoLoginBackend` class to the `AUTHENTICATION_BACKENDS` setting.
2. Set `BROWSERID_AUTOLOGIN_EMAIL` to the email you want to be logged in as.
3. Set `BROWSERID_AUTOLOGIN_ENABLED` to `True`.
4. If you are not using `browserid_js template helper`, you have to manually add `browserid/autologin.js` to your site.

For example:

```
AUTHENTICATION_BACKENDS = (
   'django_browserid.auth.AutoLoginBackend',
   'django_browserid.auth.BrowserIDBackend',  # After auto-login.
)

BROWSERID_AUTOLOGIN_EMAIL = 'bob@example.com'
BROWSERID_AUTOLOGIN_ENABLED = True
```

Once these are set, any login button that uses the *JavaScript API* will not attempt to show the Persona popup, and will immediately log you in with the email you set above.

Disable auto-login

To disable auto-login:

1. Set `BROWSERID_AUTOLOGIN_ENABLED` to `False`.
2. If you added `browserid/autologin.js` to your site, you must remove it.

1.5 Settings

This document describes the Django settings that can be used to customize the behavior of django-browserid.

1.5.1 Core Settings

`django.conf.settings.`**`BROWSERID_AUDIENCES`**

Default No default

List of audiences that your site accepts. An audience is the protocol, domain name, and (optionally) port that users access your site from. This list is used to determine the audience a user is part of (how they are accessing your site), which is used during verification to ensure that the assertion given to you by the user was intended for your site.

Without this, other sites that the user has authenticated with via Persona could use their assertions to impersonate the user on your site.

Note that this does not have to be a publicly accessible URL, so local URLs like `http://localhost:8000` or `http://127.0.0.1` are acceptable as long as they match what you are using to access your site.

1.5.2 Redirect URLs

Note: If you want to use named URLs instead of directly including URLs into your settings file, you can use reverse_lazy to do so.

`django.conf.settings.`**`LOGIN_REDIRECT_URL`**

Default `'/accounts/profile'`

Path to redirect to on successful login. If you don't specify this, the default Django value will be used.

`django.conf.settings.`**`LOGIN_REDIRECT_URL_FAILURE`**

Default `'/'`

Path to redirect to on an unsuccessful login attempt.

django.conf.settings.**LOGOUT_REDIRECT_URL**

Default `'/'`

Path to redirect to on logout.

1.5.3 Customizing the Login Popup

django.conf.settings.**BROWSERID_REQUEST_ARGS**

Default `{}`

Controls the arguments passed to `navigator.id.request`, which are used to customize the login popup box. To see a list of valid keys and what they do, check out the navigator.id.request documentation.

1.5.4 Customizing the Verify View

django.conf.settings.**BROWSERID_VERIFY_CLASS**

Default `django_browserid.views.Verify`

Allows you to substitute a custom class-based view for verifying assertions. For example, the string 'myapp.users.views.Verify' would import *Verify* from *myapp.users.views* and use it in place of the default view.

When using a custom view, it is generally a good idea to subclass the default Verify and override the methods you want to change.

django.conf.settings.**BROWSERID_CREATE_USER**

Default `True`

If `True` or `False`, enables or disables automatic user creation during authentication. If set to a string, it is treated as an import path pointing to a custom user creation function.

django.conf.settings.**BROWSERID_DISABLE_SANITY_CHECKS**

Default False

Controls whether the `Verify` view performs some helpful checks for common mistakes. Useful if you're getting warnings for things you know aren't errors.

1.5.5 Using a Different Identity Provider

django.conf.settings.**BROWSERID_SHIM**

Default 'https://login.persona.org/include.js'

The URL to use for the BrowserID JavaScript shim.

1.5.6 Extras

django.conf.settings.**BROWSERID_AUTOLOGIN_ENABLED**

Default `False`

If `True`, enables auto-login. You must also set the auto-login email and authentication backend for auto-login to function. See the documentation on *offline development* for more info.

django.conf.settings.**BROWSERID_AUTOLOGIN_EMAIL**

Default Not set

The email to log users in as when auto-login is enabled. See the documentation on *offline development* for more info.

1.6 Deploying in Production

Deploying django-browserid in a production environment requires a few extra changes from the setup described in the *Quickstart*:

- The BROWSERID_AUDIENCES setting is required when DEBUG is set to False. Ensure that all the domains that users will access your site from are listed in this setting.
- *Optional*: It is a good idea to minify the static JS and CSS files you're using. django-compressor and jingo-minify are examples of libraries you can use for minification.

1.7 Upgrading

If you're looking to upgrade from an older version of django-browserid, you're in the right place. This document describes the major changes required to get your site up to the latest and greatest!

1.7.1 0.10.1 to 0.11

No changes are necessary to switch from 0.10.1 to 0.11.

1.7.2 0.9 to 0.10.1

- The minimum supported version of requests is now 1.0.0, and six has been removed from the requirements.
- Replace the SITE_URL setting with BROWSERID_AUDIENCES, which is essentially the same setting, but must be a list of strings (wrapping your old SITE_URL value with square brackets to make it a list is fine):

  ```
  BROWSERID_AUDIENCES = ['https://www.example.com']
  ```

  - On local development installs, you can remove SITE_URL entirely, as BROWSERID_AUDIENCES isn't required when DEBUG is True.
- In your root urlconf, remove any regex in front of the include for django-browserid urls. Because the new JavaScript relies on views being available at certain URLs, you must not change the path that the django-browserid views are served:

  ```
  urlpatterns = patterns('',
      # ...
      (r'', include('django_browserid.urls')),
      # ...
  )
  ```

- Remove django_browserid.context_processors.browserid from your TEMPLATE_CONTEXT_PROCESSORS setting, as the context processor no longer exists.

- `browserid.js` has been split into `api.js`, which contains just the JavaScript API, and `browserid.js`, which contains the sample code for hooking up login buttons. If you aren't using the `browserid_js` helper to include the JavaScript on the page, you probably need to update your project to either include both or just `api.js`.
- The included JavaScript requires jQuery 1.8 or higher instead of jQuery 1.7.

1.7.3 0.8 to 0.9

- Six v1.3 or higher is now required.

1.7.4 0.7.1 to 0.8

- fancy_tag 0.2.0 has been added to the required libraries.
- Rename the `browserid_form` context processor to `browserid` in the `TEMPLATE_CONTEXT_PROCESSORS` setting:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    # ...
    'django_browserid.context_processors.browserid',
    # ...
)
```

- Replace custom login button code with the new template helpers, `browserid_info`, `browserid_login`, and `browserid_logout`.
  - `browserid_info` should be added just below `<body>` on any page that includes a login button.
  - `browserid_login` and `browserid_logout` output login and logout links respectively.
- It's now recommended to include the JavaScript for the login buttons using the `browserid_js` helper, which outputs the appropriate `<script>` tags.
- The included JavaScript requires jQuery 1.7 or higher instead of jQuery 1.6.

1.8 Troubleshooting

If you are having trouble getting django-browserid to work properly, try reading through the sections below for help on dealing with common issues.

1.8.1 Logging Errors

Before you do anything else, check to see if django-browserid is logging issues by setting up a logger for `django_browserid` in your logging config. Here's a sample config that will log messages from django-browserid to the console:

```
LOGGING = {
   'version': 1,
   'handlers': {
       'console':{
           'level': 'DEBUG',
           'class': 'logging.StreamHandler'
       },
```

```
    },
    'loggers': {
        'django_browserid': {
            'handlers': ['console'],
            'level': 'DEBUG',
        }
    },
}
```

1.8.2 If you recently updated...

If you are hitting problems after updating django-browserid, check to make sure your installed copy matches the tagged version on Github. In particular, leftover `*.pyc` files may cause unintended side effects. This is common when installing without using a package manager like `pip`.

1.8.3 Nothing happens when clicking the login button

If nothing happens when you click the login button on your website, check that you've included `api.js` and `browserid.js` on your webpage:

```
<script src="{% static 'browserid/api.js' %}"></script>
<script src="{% static 'browserid/browserid.js' %}"></script>
```

CSP WARN: Directive "..." violated by https://browserid.org/include.js

You may see this warning in your browser's error console when your site uses Content Security Policy without making an exception for the persona.org external JavaScript include.

To fix this, include https://login.persona.org in your script-src and frame-src directive. If you're using the django-csp library, the following settings will work:

```
CSP_SCRIPT_SRC = ("'self'", 'https://login.persona.org')
CSP_FRAME_SRC = ("'self'", 'https://login.persona.org')
```

1.8.4 Login fails silently after the Persona popup closes

There are a few reasons why login may fail without an error message after the Persona popup closes:

SESSION_COOKIE_SECURE is False

SESSION_COOKIE_SECURE controls if the *secure* flag is set on the session cookie. If set to True for site running in an environment that doesn't use HTTPS, the session cookie won't be sent by your browser because you're using an HTTP connection.

The solution is to set *SESSION_COOKIE_SECURE* to False on your local instance in your settings file:

```
SESSION_COOKIE_SECURE = False
```

No cache configured

Several projects (especially projects based on playdoh, which uses django-session-csrf) store session info in the cache rather than the database, and if your local instance has no cache configured, the session information will not be stored and login will fail silently.

To solve this issue, you should configure your local instance to use an in-memory cache with the following in your local settings file:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'unique-snowflake'
    }
}
```

1.8.5 Login fails with an error message on a valid account

If you see a login error page after attempting to login, but you know that your Persona account is valid and should be able to login, check for these issues:

Your website uses HTTPS but django-browserid thinks it's using HTTP

If you are using django-browserid behind a load balancer that uses HTTP internally for your SSL connections, you may experience failed logins. The `request.is_secure()` method determines if a request is using HTTPS by checking for the header specified by the SECURE_PROXY_SSL_HEADER setting. If this is unset or the header is missing, Django assumes the request uses HTTP.

Because the audiences stored in `BROWSERID_AUDIENCES` include the protocol used to access the site, you may get an error when django-browserid checks the audiences against the URL from the request due to the request thinking it's not using SSL when it is.

Make sure that `SECURE_PROXY_SSL_HEADER` is set to an appropriate value for your load balancer. An example configuration using nginx might look like this:

```
# settings.py
SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTOCOL', 'https')

# nginx config
location / {
    proxy_pass http://127.0.0.1:8000;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Protocol https; # Tell django we're using https
}
```

1.8.6 Still having issues? Ask for help!

If your issue isn't listed above and you're having trouble tracking it down, you can try asking for help from:

- The #webdev channel on irc.mozilla.org,
- The dev-webdev@lists.mozilla.org mailing list,
- or by emailing *the maintainers* directly.

CHAPTER 2

API Documentation

2.1 Python API

This part of the documentation describes the interfaces for using django-browserid.

2.1.1 Template Helpers

Template helpers are the functions used in your templates that output HTML for login and logout buttons, as well as the CSS and JS tags for making the buttons function and display correctly.

`django_browserid.helpers.`**`browserid_info`**`()`

Output the HTML for the info tag, which contains the arguments for navigator.id.request from the BROWSERID_REQUEST_ARGS setting. Should be called once at the top of the page just below the <body> tag.

`django_browserid.helpers.`**`browserid_login`**(*text='Sign in'*, *color=None*, *next=None*, *link_class='browserid-login persona-button'*, *attrs=None*, *fallback_href='#'*)

Output the HTML for a BrowserID login link.

Parameters

- **text** – Text to use inside the link. Defaults to 'Sign in', which is not localized.
- **color** – Color to use for the login button; this will only work if you have included the default CSS provided by `django_browserid.helpers.browserid_css()`.

  Supported colors are: *'dark'*, *'blue'*, and *'orange'*.
- **next** – URL to redirect users to after they login from this link. If omitted, the LOGIN_REDIRECT_URL setting will be used.
- **link_class** – CSS class for the link. Defaults to *browserid-login persona-button*.
- **attrs** – Dictionary of attributes to add to the link. Values here override those set by other arguments.

  If given a string, it is parsed as JSON and is expected to be an object.
- **fallback_href** – Value to use for the href of the link. If the user has disabled JavaScript, the login link will bring them to this page, which can be used as a non-JavaScript login fallback.

`django_browserid.helpers.`**`browserid_logout`**(*text='Sign out'*, *next=None*, *link_class='browserid-logout'*, *attrs=None*)

Output the HTML for a BrowserID logout link.

Parameters

- **text** – Text to use inside the link. Defaults to 'Sign out', which is not localized.
- **link_class** – CSS classes for the link. The classes will be appended to the default class *browserid-logout*.
- **attrs** – Dictionary of attributes to add to the link. Values here override those set by other arguments.

  If given a string, it is parsed as JSON and is expected to be an object.

`django_browserid.helpers.`**`browserid_js`**(*include_shim=True*)

Return <script> tags for the JavaScript required by the BrowserID login button. Requires use of the staticfiles app.

If the BROWSERID_AUTOLOGIN_ENABLED setting is True, an extra JavaScript file for mocking out Persona will be included, and the shim won't be included regardless of the value of the `include_shim` setting.

Parameters **include_shim** – A boolean that determines if the persona.org JavaScript shim is included in the output. Useful if you want to minify the button JavaScript using a library like django-compressor that can't handle external JavaScript.

`django_browserid.helpers.`**`browserid_css`**()

Return <link> tag for the optional CSS included with django-browserid. Requires use of the staticfiles app.

2.1.2 Admin Site

Admin site integration allows you to support login via django-browserid on the Django built-in admin interface.

class `django_browserid.admin.`**`BrowserIDAdminSite`**(*name='admin'*, *app_name='admin'*)

Support logging in to the admin interface via BrowserID.

`include_password_form` = **False**

If True, include the normal username and password form as well as the BrowserID button.

`copy_registry`(*site*)

Copy the ModelAdmins that have been registered on another site so that they are available on this site as well.

Useful when used with `django.contrib.admin.autocomplete()`, allowing you to copy the ModelAdmin entries registered with the default site, such as the User ModelAdmin. For example, in `urls.py`:

```
from django.contrib import admin
admin.autodiscover()

from django_browserid.admin import site as browserid_admin
browserid_admin.copy_registry(admin.site)

# To include: url(r'^admin/', include(browserid_admin.urls))
```

Parameters **site** – Site to copy registry entries from.

`django_browserid.admin.`**`site`**

Global object for the common case. You can import this in `admin.py` and `urls.py` instead of `django.contrib.admin.site`.

2.1.3 Authentication Backends

There are a few different authentication backends to choose from depending on how you want to authenticate users.

class `django_browserid.auth.`**`BrowserIDBackend`**

`get_verifier``()`
Create a verifier for verifying assertions. Uses a `django_browserid.base.RemoteVerifier` by default.

`filter_users_by_email`(*email*)
Return all users matching the specified email.

`create_user`(*email*)
Return object for a newly created user account.

`is_valid_email`(*email*)
Return True if the email address is ok to log in.

`verify`(*assertion=None*, *audience=None*, *request=None*, ***kwargs*)
Verify the given assertion and audience. See `authenticate` for accepted arguments.

`authenticate`(*assertion=None*, *audience=None*, *request=None*, ***kwargs*)
Authenticate a user by verifying a BrowserID assertion. Defers to the verifier returned by `BrowserIDBackend.get_verifier()` for verification.

You may either pass the `request` parameter to determine the audience from the request, or pass the `audience` parameter explicitly.

Parameters

- **assertion** – Assertion submitted by the user. This asserts that the user controls a specific email address.
- **audience** – The audience to use when verifying the assertion; this prevents another site using an assertion for their site to login to yours. This value takes precedence over the audience pulled from the request parameter, if given.
- **request** – The request that generated this authentication attempt. This is used to determine the audience to use during verification, using the `django_browserid.base.get_audience()` function. If the audience parameter is also passed, it will be used instead of the audience from the request.
- **kwargs** – All remaining keyword arguments are passed to the `verify` function on the verifier.

class `django_browserid.auth.`**`LocalBrowserIDBackend`**
Bases: `django_browserid.auth.BrowserIDBackend`

BrowserID authentication backend that uses local verification instead of remote verification.

2.1.4 Views

django-browserid works primarily through AJAX requests to the views below in order to log users in and out and to send information required for the login process, such as a CSRF token.

class `django_browserid.views.`**`Verify`**(***kwargs*)
Bases: `django_browserid.views.JSONView`

Send an assertion to the remote verification service, and log the user in upon success.

failure_url
URL to redirect users to when login fails. This uses the value of `settings.LOGIN_REDIRECT_URL_FAILURE`, and defaults to `'/'` if the setting doesn't exist.

success_url
URL to redirect users to when login succeeds. This uses the value of `settings.LOGIN_REDIRECT_URL`, and defaults to `'/'` if the setting doesn't exist.

login_success()
Log the user into the site.

login_failure()
Redirect the user to a login-failed page. By default a 403 is returned.

post(**args*, ***kwargs*)
Send the given assertion to the remote verification service and, depending on the result, trigger login success or failure.

dispatch(*request*, **args*, ***kwargs*)
Run some sanity checks on the request prior to dispatching it.

class django_browserid.views.**Logout**(***kwargs*)
Bases: django_browserid.views.JSONView

redirect_url
URL to redirect users to post-login. Uses `settings.LOGOUT_REDIRECT_URL` and defaults to `/` if the setting isn't found.

post(*request*)
Log the user out.

class django_browserid.views.**CsrfToken**(***kwargs*)
Bases: django_browserid.views.JSONView

Fetch a CSRF token for the frontend JavaScript.

2.1.5 Signals

django_browserid.signals.**user_created**
Signal triggered when a user is automatically created during authentication.

Parameters

- **sender** – The function that created the user instance.
- **user** – The user instance that was created.

2.1.6 Exceptions

exception django_browserid.base.**BrowserIDException**(*exc*)
Raised when there is an issue verifying an assertion.

exc = None
Original exception that caused this to be raised.

2.1.7 Verification

The verification classes allow you to verify if a user-provided assertion is valid according to the Identity Provider specified by the user's email address. Generally you don't have to use these directly, but they are available for sites with complex authentication needs.

class `django_browserid.`**`RemoteVerifier`**
Verifies BrowserID assertions using a remote verification service.

By default, this uses the Mozilla Persona service for remote verification.

`verify`(*assertion*, *audience*, ***kwargs*)
Verify an assertion using a remote verification service.

Parameters

- **assertion** – BrowserID assertion to verify.
- **audience** – The protocol, hostname and port of your website. Used to confirm that the assertion was meant for your site and not for another site.
- **kwargs** – Extra keyword arguments are passed on to requests.post to allow customization.

Returns `VerificationResult`

Raises `BrowserIDException`: Error connecting to the remote verification service, or error parsing the response received from the service.

class `django_browserid.`**`LocalVerifier`**(**args*, ***kwargs*)
Verifies BrowserID assertions locally instead of using the remote verification service.

`verify`(*assertion*, *audience*, ***kwargs*)
Verify an assertion locally.

Parameters

- **assertion** – BrowserID assertion to verify.
- **audience** – The protocol, hostname and port of your website. Used to confirm that the assertion was meant for your site and not for another site.

Returns `VerificationResult`

class `django_browserid.`**`MockVerifier`**(*email*, ***kwargs*)
Mock-verifies BrowserID assertions.

`__init__`(*email*, ***kwargs*)

Parameters

- **email** – Email address to include in successful verification result. If None, verify will return a failure result.
- **kwargs** – Extra keyword arguments are used to update successful verification results. This allows for mocking attributes on the result, such as the issuer.

`verify`(*assertion*, *audience*, ***kwargs*)
Mock-verify an assertion. The return value is determined by the parameters given to the constructor.

class `django_browserid.`**`VerificationResult`**(*response*)
Result of an attempt to verify an assertion.

VerificationResult objects can be treated as booleans to test if the verification succeeded or not.

The fields returned by the remote verification service, such as `email` or `issuer`, are available as attributes if they were included in the response. For example, a failure result will raise an AttributeError if you try to access the `email` attribute.

expires
The expiration date of the assertion as a naive `datetime.datetime` in UTC.

`django_browserid.`**get_audience**(*request*)
Determine the audience to use for verification from the given request.

Relies on the BROWSERID_AUDIENCES setting, which is an explicit list of acceptable audiences for your site.

Returns The first audience in BROWSERID_AUDIENCES that has the same origin as the request's URL.

Raises `django.core.exceptions.ImproperlyConfigured`: If BROWSERID_AUDIENCES isn't defined, or if no matching audience could be found.

2.2 JavaScript API

Normally, you simply include `browserid/api.js` and `browserid/browserid.js` on a page, and buttons generated by the *template helpers* will just work. If, however, you want more control, you can use the JavaScript API, defined in `api.js` directly.

For example, if you wanted to trigger login and show a message when there is an error:

```
$('.loginButton').click(function() {
    django_browserid.login().then(function(verifyResult) {
        window.location = verifyResult.redirect;
    }, function(jqXHR) {
        window.alert('There was an error logging in, please try again.');
    });
});
```

Note: See also `browserid/browserid.js` for an example of using the API.

This part of the documentation describes the JavaScript API defined in `api.js`.

django_browserid
Global object containing the JavaScript API for interacting with django-browserid.

Most functions return jQuery Deferreds for registering asynchronous callbacks.

login([*requestArgs*])
Retrieve an assertion and use it to log the user into your site.

Arguments

- **requestArgs** (*object*) – Options to pass to navigator.id.request.

Returns Deferred that resolves once the user has been logged in.

logout()
Log the user out of your site.

Returns Deferred that resolves once the user has been logged out.

getAssertion([*requestArgs*])
Retrieve an assertion via BrowserID.

Returns Deferred that resolves with the assertion once it is retrieved.

verifyAssertion(*assertion*)
Verify that the given assertion is valid, and log the user in.

Arguments

- **assertion** (*string*) – Assertion to verify.

Returns Deferred that resolves with the login view response once login is complete.

getInfo()
Fetch information from the `browserid_info` tag, such as the parameters for the Persona popup.

Returns Object containing the data from the info tag.

getCsrfToken()
Fetch a CSRF token from the `CsrfToken view` via an AJAX request.

Returns Deferred that resolves with the CSRF token.

registerWatchHandlers([*onReady*])
Register callbacks with navigator.id.watch that make the API work. This must be called before calling any other API methods.

Arguments

- **onReady** (*function*) – Callback that will be executed after the user agent is ready to process login requests. This is passed as the `onready` argument to navigator.id.watch

CHAPTER 3

Contributor Guide

3.1 Contributor Setup

So you want to contribute to django-browserid? Great! We really appreciate any help you can give!

The documentation below should help you set up a development environment and run the tests to ensure that your changes work properly.

3.1.1 Get the code

You can check out the code from the github repository:

```
git clone git://github.com/mozilla/django-browserid.git
cd django-browserid
```

It is a good idea to create a virtualenv (the example here uses virtualenvwrapper) for isolating your development environment. To create a virtualenv and install all development packages:

```
mkvirtualenv django-browserid
pip install -r requirements.txt
```

3.1.2 Running tests

To check if your changes break any existing functionality, you can run the test suite:

```
./setup.py test
```

Before submitting a pull request, you should run the test suite in all the Django/Python combinations that we support. We support running the tests in all these combinations via tox:

```
pip install tox
tox
```

3.1.3 Documenation

If you make changes to the documentation, you can build it locally with this command:

```
make -C docs/ html
```

The generated files can be found in `docs/_build/html`.

3.1.4 JavaScript Tests

To run the JavaScript tests, you must have node.js installed. Then, use the npm command to install the test dependencies:

```
npm install
```

After that, you can run the JavaScript tests with the following command from the repo root:

```
npm test
```

3.2 Contributing Guidelines

In order to make our review/development process easier, we have some guidelines to help you figure out how to contribute to django-browserid.

3.2.1 Reporting Issues

We use Github Issues to track issues and bugs for django-browserid.

3.2.2 Development Guidelines

- Python code should be covered by unit tests. JavaScript code for the JavaScript API should be covered by unit tests. We don't yet have tests for non-API JavaScript code, so manual testing is recommended currently.
- Python code should follow Mozilla's general Webdev guidelines. The same goes for our JavaScript guidelines and CSS guidelines.
  - As allowed by PEP8, we use 99-characters-per-line for Python code and 72-characters-per-line for documentation/comments. Feel free to break these guidelines for readability if necessary.

3.2.3 Submitting a Pull Request

When submitting a pull request, make sure to do the following:

- Check that the Python and JavaScript tests pass in all environments. Running the Python tests in all environments is easy using tox:

  ```
  $ pip install tox
  $ tox
  ```

  Running the JavaScript tests requires node.js. To install the test dependencies and run the test suite:

  ```
  $ npm install
  $ npm test
  ```

- Make sure to include new tests or update existing tests to cover your changes.
- If you haven't, add your name, username, or alias to the `AUTHORS.rst` file as a contributor.

3.2.4 Additional Resources

- IRC: #webdev on irc.mozilla.org.
- Mailing list: dev-webdev@lists.mozilla.org.

3.3 Changelog

3.3.1 History

0.11 (2014-09-25)

- Add support for local assertion verification instead of relying on the remote verification service if PyBrowserID is installed.
- Add an auto-login backend to support offline local development when the Persona service isn't available.
- Run automated tests for Django 1.7.
- Use the stateless Persona API, removing the need to work around issues involving Persona attempting to auto-login or auto-logout users.
- Add support for setting an *on_ready* handler to be executed when the Persona API is ready to fetch assertions.
- Fix broken Django admin integration.
- Fix some issues around CSRF tokens used during the login process.
- Improve logging when using the default verify view so that it doesn't look like an error.
- Various documentation updates.

0.10.1 (2014-05-02)

- Add `browserid_info` helper back in. The previous method of fetching the Persona popup customization via AJAX caused browsers to trigger popup warnings when users attempted to log in, so we switched back to the old method of adding the info tag to pages.

0.10 (2014-04-15)

- Massive documentation update, including upgrade instructions for older versions.
- Support and test on Python 3.2 and 3.3, and Django 1.6!
- Disable automatic login and logout coming from Persona. This also fixes logins being triggered in all open tabs on your site.
- Replace in-page form for trigger logins with AJAX calls. Removes need for {% browserid_info %} template tag.
- Drop `six` from requirements.
- Replace `SITE_URL` setting with `BROWSERID_AUDIENCES` and make it optional when `DEBUG` is True.
- Add support for logging-in to the admin interface with Persona.
- Remove need to set custom context processor.
- Replace `verify` function with the Verifier classes like `RemoteVerifier`.

- And more!

0.9 (2013-08-25)

- Add `BROWSERID_VERIFY_CLASS` to make it easier to customize the verification view.
- Add hook to authentication backend for validating the user's email.
- Ensure backend attribute exists on user objects authenticated by django-browserid.
- Prevent installation of the library as an unpackaged egg.
- Add incomplete Python 3 support.
- Fix an issue where users who logged in without Persona were being submitted to `navigator.id.watch` anyway.
- Add CSS to make the login/logout buttons prettier.
- Support for `SITE_URL` being an iterable.
- Add support for lazily-evaluated `BROWSERID_REQUEST_ARGS`.
- Add a small JavaScript API available on pages that include `browserid.js`.
- Support running tests via *python setup.py test*.
- Fix an infinite loop where logging in with a valid Persona account while `BROWSERID_CREATE_USER` is true would cause an infinite redirection.

0.8 (2013-03-05)

- #97: Add BrowserIDException that is raised by verify when there are issues connecting to the remote verification service. Update the Verify view to handle these errors.
- #125: Prevent the Verify view from running reverse on user input and add check to not redirect to URLs with a different host.
- Remove ability to set a custom name for the Verify redirect parameter: it's just `next`.
- Replace `browserid_button` with `browserid_login` and `browserid_logout`, and make `browserid_info` a function.
- #109: Fix issue with unicode strings in the `extra_params` kwarg for `verify`.
- #110: Fix bug where kwargs to `authenticate` get passed as `extra_params` to verify. Instead, you can pass any extra parameters in `browserid_extra`. But please don't, it's undocumented for a reason. <3
- #105: General documentation fixes, add more debug logging for common issues. Add `BROWSERID_DISABLE_SANITY_CHECKS` setting and remove the need to set `SITE_URL` in development.
- Add `form_extras` parameter to `browserid_button`.
- #101, #102: Update the default JavaScript to pass the current user's email address into `navigator.id.watch` to avoid unnecessary auto-login attempts.
- Add template functions/tags to use for embedding login/logout buttons instead of using your own custom HTML.
- Add a `url` kwarg to `verify` that lets you specify a custom verification service to use.
- Add documentation for setting up the library for development.

- #103: `BrowserIDForm` now fails validation if the assertion given is non-ASCII.
- Fix an error in the sample urlconf in the documentation.
- #98: Fix a bug where login or logout buttons might not be detected by the default JavaScript correctly if `<a>` element contained extra HTML.
- Add `pass_mock` kwarg to `mock_browserid`, which adds a new argument to the front of the decorated method that is filled with the Mock object used in place of `_verify_http_request`.
- Any extra kwargs to `BrowserIDBackend.authenticate` are passed in the verify request as POST arguments (this will soon be removed, don't rely on it).

0.7.1 (2012-11-08)

- Add support for a working logout button. Switching to the Observer API in 0.7 made the issue that we weren't calling `navigator.id.logout` more pronounced, so it makes sense to make a small new release to make it easier to add a logout button.

0.7 (2012-11-07)

- Actually start updating the Changelog again.
- Remove deprecated functions `django_browserid.auth.get_audience` and `django_browserid.auth.BrowserIDBackend.verify`, as well as support for `DOMAIN` and `PROTOCOL` settings.
- Add small fix for infinite login loops.
- Add automated testing for Django 1.3.4, 1.4.2, and 1.5a1.
- Switch to using `format` for all string formatting (**breaks Python 2.5 compatibility**).
- Add support for Django 1.5 Custom User Models.
- Fix request timeouts so that they work properly.
- Add ability to customize BrowserID login popup via arguments to `navigator.id.request`.
- Update JavaScript to use the new Observer API.
- Change `browserid.org` urls to `login.persona.org`.

3.4 Authors

`django-browserid` is written and maintained by various contributors:

3.4.1 Current Maintainers

- Michael Kelly <mkelly@mozilla.com>
- Will Kahn-Greene <willkg@mozilla.com>
- Peter Bengtsson <peterbe@mozilla.com>

3.4.2 Previous Maintainers

- Paul Osman
- Austin King
- Ben Adida

3.4.3 Patches and Suggestions

- Thomas Grainger
- Owen Coutts
- Francois Marier
- Andy McKay
- Giorgos Logiotatidis
- Alexis Metaireau
- Rob Hudson
- Ross Bruniges
- Les Orchard
- Charlie DeTar
- Luke Crouch
- shaib
- Kumar McMillan
- Carl Meyer
- ptgolden
- Will Kahn-Greene
- Allen Short
- meehow
- Greg Koberger
- Niran Babalola
- callmekatootie
- Paul Mclanahan
- JR Conlin
- Prasoon Shukla
- Peter Bengtsson
- Javed Khan
- Kalail (Kashif Malik)
- Richard Mansfield
- Francesco Pischedda

- Edward Abraham
- Eric Holscher
- mvasilkov
- Mounir Messelmeni
- Tomo Krajina
- Sergio Oliveira

d

django_browserid, 17
django_browserid.admin, 18
django_browserid.auth, 19
django_browserid.helpers, 17
django_browserid.signals, 20
django_browserid.views, 19