django-lazysignup Documentation

Release 1.1.2

Dan Fairs

Sep 09, 2017

Contents

Contents:

CHAPTER 1

Requirements

Tested on Django 1.10.0 and above. It requires `django.contrib.auth` to be in the `INSTALLED_APPS` list.

CHAPTER 2

Installation

django-lazysignup can be installed with your favourite package management tool from PyPI:

```
pip install django-lazysignup
```

Once that's done, you need to add `lazysignup` to your `INSTALLED_APPS`. You will also need to add `lazysignup`'s authentication backend to your site's `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = (
  'django.contrib.auth.backends.ModelBackend',
  'lazysignup.backends.LazySignupBackend',
)
```

Finally, you need to add lazysignup to your URLConf, using something like this:

```
urlpatterns += (
    url(r'^convert/', include('lazysignup.urls')),
)
```

CHAPTER 3

Usage

The package works by creating temporary user accounts based on a user's session key whenever a flagged view is requested. You can specify which views trigger this behaviour using the `lazysignup.decorators.allow_lazy_user` decorator.

When an anonymous user requests such a view, a temporary user account will be created for them, and they will be logged in. The user account will have an unusable password set, so that it can't be used to log in as a regular user. The way to tell a regular use from a temporary user is to call the `is_lazy_user()` function from `lazysignup.templatetags.lazysignup_tags`. If this returns `True`, then the user is temporary. Note that `user.is_anonymous()` will return `False` and `user.is_authenticated()` will return `True`. See below for more information on `is_lazy_user`.

A view is provided to allow such users to convert their temporary account into a real user account by providing a username and a password.

A Django management command is provided to clear out stale, unconverted user accounts - although this depends on your use of database-backed sessions, and assumes that all user accounts with an expired session are safe to delete. This may not be the case for all apps, so you may wish to provide your own cleaning script.

The `allow_lazy_user` decorator

Use this decorator to indicate that accessing the view should cause anonymous users to have temporary accounts created for them.

For example:

```
from django.http import HttpResponse
from lazysignup.decorators import allow_lazy_user

@allow_lazy_user
def my_view(request):
  return HttpResponse(request.user.username)
```

When accessing the above view, a very simple response containing the generated username will be displayed.

The `require_lazy_user` and `require_nonlazy_user` decorators

It is also possible to mark views as requiring only a lazily-created user, or requiring only a non-lazy user, with the `require_lazy_user` and `require_nonlazy_user` decorators respectively. These decorators take arguments and keyword arguments which are passed verbatim to Django's own `redirect` view.

The `is_lazy_user` template filter

This template filter (which can also be imported from `lazysignup.utils` and used in your own code) will return True if the user is a generated user. You need to pass it the user to test. For example, a site navigation template might look like this:

```
{% load i18n lazysignup_tags %}

<nav id="account-bar">
  <ul>
    <li><a href="{% url home %}">{% trans "Home" %}</a></li>
    {% if not user|is_lazy_user %}
      <li><a href="#">{% trans "Account" %}</a></li>
      <li><a href="{% url auth_logout %}">{% trans "Log out" %}</a></li>
    {% else %}
      <li><a href="{% url lazysignup_convert %}">{% trans "Save your data" %}</a> {%␣
↪trans "by setting a username and password" %}</li>
    {% endif %}
  </ul>
</nav>
```

This filter is very simple, and can be used directly in view code, or tests. For example:

```
from lazysignup.utils import is_lazy_user

def testIsLazyUserAnonymous(self):
    user = AnonymousUser()
    self.assertEqual(False, is_lazy_user(user))
```

Note that as of version 0.6.0, the user tested no longer needs to have been authenticated by the `LazySignupBackend` for lazy user detection to work.

User agent blacklisting

The middleware will not create users for certain requests from blacklisted user agents. This is simply a fairly crude method for preventing many spurious users being created by passing search engines.

The blacklist is specified with the `USER_AGENT_BLACKLIST` setting. This should be an iterable of regular expression strings. If the user agent string of a request matches a regex (`search()` is used, so the match can be anywhere in the string) then a user will not be created.

If the list is not specified, then the default is as follows

- slurp
- googlebot
- yandex

- msnbot
- baiduspider

Specifying your own `USER_AGENT_BLACKLIST` will replace this list.

Using the convert view

Users will be able to visit the `/convert/` view. This provides a form with a username, password and password confirmation. As long as they fill in valid details, their temporary user account will be converted into a real user account that they can log in with as usual.

You may specify your own form class into the *convert* view in order to customise user creation. The code requires expects the following:

- It expects to be able to create the form passing in the generated `User` object with an `instance` kwarg (in general, this is fine when using a ModelForm based on the User model)
- It expects to be able to call `save()` on the form to convert the user to a real user
- It expects to be able to call a `get_credentials()` method on the form to obtain a set of credentials to authenticate the new user with. The result of this call should be a dictionary suitable for passing to `django.contrib.auth.authenticate()`. Typically, this would be a dict with `username` and `password` keys - but this may vary if you're using a different authentication backend.

The default configuration, using the provided `UserCreationForm`, should be enough for most users, but the customisation point is there if you need it.

To specify your own form, set the `LAZYSIGNUP_CUSTOM_USER_CREATION_FORM` setting to your settings file like so:

```
LAZYSIGNUP_CUSTOM_USER_CREATION_FORM = 'myproject.apps.myapp.forms.MyForm'
```

The view also supports `template_name` and `ajax_template_name` arguments, to specify templates to render in web and ajax contexts respectively.

The `converted` signal

Whenever a temporary user account is converted into a real user account, the `lazysignup.signals.converted` signal will be sent. If you need to do any processing when an account is converted, you should listen for the signal, eg:

```
from lazysignup.signals import converted
from django.dispatch import receiver

@receiver(converted)
def my_callback(sender, **kwargs):
    print "New user account: %s!" % kwargs['user'].username
```

The signal provides a single argument, `user`, which contains the newly-converted User object.

CHAPTER 4

Custom User classes

Many projects use a custom `User` class, augmenting that from `django.contrib.auth`. If you want to use such a custom class with `lazysignup`, then you can set the `LAZYSIGNUP_USER_MODEL` setting. This should be a standard dotted Django name for a model, eg:

```
LAZYSIGNUP_USER_MODEL = 'myapp.CustomUser'
```

The setting defaults to `settings.AUTH_USER_MODEL`, so if you've set `AUTH_USER_MODEL` to your custom model, there is no need to alter `LAZYSIGNUP_USER_MODEL`.

If you do use a custom user class, note that `lazysignup` expects that class' default manager to have a `create_user` method, with the same signature and semantics as `django.contrib.auth.models.UserManager`. If your model actually subclasses Django's own user model, you may well be able to use this manager directly. For example:

```
from django.contrib.auth.models import AbstractUser, UserManager

class MyCustomUser(AbstractUser):
    objects = UserManager()

    notes = models.TextField(blank=True, null=True)
```

`lazysignup` also expects that it can fetch instances of your custom user class using a `get()` method on the object's manager, and that looking them up by primary key and by `username` will work. See `lazysignup.backends` for more detail.

CHAPTER 5

Maintenance

Over time, a number of user accounts that haven't been converted will build up. To avoid performance problems from an excessive number of user accounts, it's recommended that the `remove_expired_users` management command is run on a regular basis. It runs from the command line:

```
python manage.py remove_expired_users
```

In a production environment, this should be run from cron or similar.

There is also an action in the Django Admin for removing expired users. To use, select all LazyUser instances, select the action "Delete selected lazy users and unconverted users older than settings.SESSION_COOKIE_AGE", and click "Go".

This works by removing user accounts from the system whose associated sessions have expired. `user.delete()` is called for each user, so related data will be removed as well.

Note of course that these deletes will cascade, so if you need to keep data associated with such users, you'll need to write your own cleanup job.

CHAPTER 6

Helping Out

If you want to add a feature or fix a bug, please go ahead! Fork the project on GitHub and when you're done with your changes, let me know. Fixes and features with tests have a greater chance of being merged. To run the tests, do:

```
git clone https://github.com/danfairs/django-lazysignup
cd django-lazysignup

# Install dependencies and requirements
pip install -e .[all]

# To test against a PostgreSQL Database locally
psql -c "CREATE USER lazysignup with login createdb password 'lazysignup';"
psql -c "CREATE DATABASE lazysignup with OWNER lazysignup;"
export DB="local-postgres"

# To test against a MySQL Database locally
mysql -e "CREATE DATABASE lazysignup CHARACTER SET utf8;"
mysql -e "CREATE USER 'lazysignup'@'localhost' IDENTIFIED BY 'lazysignup';"
mysql -e "GRANT ALL PRIVILEGES ON lazysignup.* to 'lazysignup'@'localhost';"
mysql -e "FLUSH PRIVILEGES;"
export DB="local-mysql"

# To test against a SQLite Database locally
export DB="sqlite"

# Run the tests and report coverage
coverage run manage.py test
coverage report --fail-under=98

coverage run manage.py test --settings=custom_user_tests.settings
coverage report --fail-under=98
```

Build the docs

To build and view the documentation, run

```
pip install -e .[all]
python setup.py build_sphinx
open docs/_build/html/index.html
```

Releasing

Releasing to pypi is as simple as:

```
pip install -e .[all]
python setup.py sdist bdist_wheel upload
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search