django-otp Documentation

Release 0.4.0

Peter Sagerson

Jul 22, 2017

Contents

Package Documentation

This project makes it easy to add support for one-time passwords (OTPs) to Django. It can be integrated at various levels, depending on how much customization is required. It integrates with `django.contrib.auth`, although it is not a Django authentication backend. The primary target is developers wishing to incorporate OTPs into their Django projects as a form of two-factor authentication.

This project includes several simple OTP plugins and more are available separately. This package also includes an implementation of OATH HOTP and TOTP for convenience, as these are standard OTP algorithms used by multiple plugins.

This version is supported on Python 2.7 and 3.4+; and Django 1.8 and 1.10+.

CHAPTER 1

Overview and Key Concepts

The django_otp package contains a framework for processing one-time passwords as well as support for *several types* of OTP devices. Support for additional devices is handled by plugins, *distributed separately*.

Adding two-factor authentication to your Django site involves four main tasks:

1. Installing the django-otp plugins you want to use.
2. Adding one or more OTP-enabled login views.
3. Restricting access to all or portions of your site based on whether users have been verified by a registered OTP device.
4. Providing mechanisms to register OTP devices to user accounts (or relying on the Django admin interface).

Installation

Basic installation has only two steps:

1. Install `django_otp` and any *plugins* that you'd like to use. These are simply Django apps to be installed in the usual way.
2. Add `django_otp.middleware.OTPMiddleware` to `MIDDLEWARE` or `MIDDLEWARE_CLASSES`. It must be installed *after* `AuthenticationMiddleware`.

For example:

```
INSTALLED_APPS = [
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.admin',
    'django.contrib.admindocs',
```

```
    'django_otp',
    'django_otp.plugins.otp_totp',
    'django_otp.plugins.otp_hotp',
    'django_otp.plugins.otp_static',
]

MIDDLEWARE = [
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django_otp.middleware.OTPMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
]
```

The plugins contain models that must be migrated.

Upgrading

Version 0.2.4 of django-otp introduced a South migration to the otp_totp plugin. Version 0.3.0 added Django 1.7 and South migrations to all apps. Care must be taken when upgrading in certain cases.

The recommended procedure is:

1. Upgrade django-otp to 0.2.7, as described below.
2. Upgrade Django to 1.7 or later.
3. Upgrade django-otp to the latest version.

django-otp 0.4 will drop support for Django < 1.7.

Upgrading from 0.2.3 or Earlier

If you're using django-otp <= 0.2.3, you need to convert otp_totp to South before going any further:

```
pip install 'django-otp==0.2.7'
python manage.py migrate otp_totp 0001 --fake
python manage.py migrate otp_totp
```

If you're not using South, you can run `python manage.py sql otp_totp` to see the definition of the new `last_t` field and then construct a suitable `ALTER TABLE` SQL statement for your database.

Upgrading to Django 1.7+

Once you've upgraded django-otp to version 0.2.4 or later (up to 0.2.7), it's safe to switch to Django 1.7 or later. You should not have South installed at this point, so any old migrations will simply be ignored.

Once on Django 1.7+, it's safe to upgrade django-otp to 0.3 or later. All plugins with models have Django migrations, which will be ignored if the tables have already been created.

If you're already on django-otp 0.3 or later when you move to Django 1.7+ (see below), you'll want to make sure Django knows that all migrations have already been run:

```
python manage.py migrate --fake <otp_plugin>
...
```

Upgrading to 0.3.x with South

If you want to upgrade django-otp to 0.3.x under South, you'll need to convert all of the remaining plugins. First make sure you're running South 1.0, as earlier versions will not find the migrations. Then convert any plugin that you have installed:

```
pip install 'django-otp>=0.3'
python manage.py migrate otp_hotp 0001 --fake
python manage.py migrate otp_static 0001 --fake
python manage.py migrate otp_yubikey 0001 --fake
python manage.py migrate otp_twilio 0001 --fake
```

Authentication and Verification

In a normal Django deployment, the user associated with a request is either authenticated or not. With the introduction of two-factor authentication, the situation becomes a little more complicated: while it is certainly possible to design a site such that two factors are required for any authentication, that's only one option. It's entirely reasonable to allow users to log in with either one or two factors and grant then access accordingly.

In this documentation, a user that has passed Django's authentication API is called *authenticated*. A user that has additionally been accepted by a registered OTP device is called *verified*. On an OTP-enabled Django site, there are thus three levels of authentication:

- anonymous
- authenticated
- authenticated + verified

When planning your site, you'll want to consider whether different views will require different levels of authentication. As a convenience, we provide the decorator `django_otp.decorators.otp_required()`, which is analogous to `login_required()`, but requires the user to be both authenticated and verified.

`OTPMiddleware` populates `request.user.otp_device` to the OTP device object that verified the current user (if any). As a convenience, it also adds `user.is_verified()` as a counterpart to `user.is_authenticated()`. It is not possible for a user to be verified without also being authenticated.[1]

Plugins and Devices

A django-otp plugin is simply a Django app that contains one or more models that are subclassed from `django_otp.models.Device`. Each model class supports a single type of OTP device. Remember that when we use the term *device* in this context, we're not necessarily referring to a physical device. At the code level, a device is a model object that can verify a particular type of OTP. For example, you might have a YubiKey that supports both the Yubico OTP algorithm and the HOTP standard: these would be represented as different devices and likely served by different plugins. A device that delivered HOTP values to a user by SMS would be a third device defined by another plugin.

[1] If you'd like the second factor to persist across sessions, see django-agent-trust and django-otp-agents. The former deals with assigning trust to user agents (i.e. browsers) across sessions and the latter includes tools to use OTPs to establish that trust.

OTP plugins are distributed as Django apps; to install a plugin, just add it to `INSTALLED_APPS` like any other. The order can be significant: any time we enumerate a user's devices, such as when we ask the user which device they would like to authenticate with, we will present them according to the order in which the apps are installed.

OTP devices come in two general flavors: passive and interactive. A passive device is one that can accept a token from the user and verify it with no preparation. Examples include devices corresponding to dedicated hardware or smartphone apps that generate sequenced or time-based tokens. An interactive device needs to communicate something to the user before it can accept a token. Two common types are devices that use a challenge-response OTP algorithm and devices that deliver a token to the user through an independent channel, such as SMS.

Internally, device instances can be flagged as confirmed or unconfirmed. By default, devices are confirmed as soon as they are created, but a plugin or deployment that wishes to include a confirmation step can mark a device unconfirmed initially. Unconfirmed devices will be ignored by the high-level OTP APIs.

Built-in Plugins

django-otp includes support for several standard device types. *HOTPDevice* and *TOTPDevice* handle standard OTP algorithms, which can be used with a variety of OTP generators. For example, it's easy to pair these devices with Google Authenticator using the otpauth URL scheme. If you have the qrcode package installed, the admin interface will generate QR Codes for you.

HOTP Devices

HOTP is an algorithm that generates a pseudo-random sequence of codes based on an incrementing counter. Every time a prover generates a new code or a verifier verifies one, they increment their respective counters. This algorithm will fail if the prover generates too many codes without a successful verification.

class django_otp.plugins.otp_hotp.models.**HOTPDevice**(**args*, ***kwargs*)

A generic HOTP *Device*. The model fields mostly correspond to the arguments to *django_otp.oath.hotp()*. They all have sensible defaults, including the key, which is randomly generated.

key

CharField: A hex-encoded secret key of up to 40 bytes. (Default: 20 random bytes)

digits

PositiveSmallIntegerField: The number of digits to expect from the token generator (6 or 8). (Default: 6)

tolerance

PositiveSmallIntegerField: The number of missed tokens to tolerate. (Default: 5)

counter

BigIntegerField: The next counter value to expect. (Initial: 0)

bin_key

The secret key as a binary string.

config_url

A URL for configuring Google Authenticator or similar.

See https://github.com/google/google-authenticator/wiki/Key-Uri-Format. The issuer is taken from *OTP_HOTP_ISSUER*, if available.

class django_otp.plugins.otp_hotp.admin.**HOTPDeviceAdmin**(*model*, *admin_site*)

ModelAdmin for *HOTPDevice*.

HOTP Settings

OTP_HOTP_ISSUER Default: `None`

The `issuer` parameter for the otpauth URL generated by *config_url*.

TOTP Devices

TOTP is an algorithm that generates a pseudo-random sequence of codes based on the current time. A typical implementation will change codes every 30 seconds, although this is configurable. This algorithm will fail if the prover and verifier have clocks that drift too far apart.

class `django_otp.plugins.otp_totp.models.`**`TOTPDevice`**(**args*, ***kwargs*)

A generic TOTP *Device*. The model fields mostly correspond to the arguments to *django_otp.oath.totp()*. They all have sensible defaults, including the key, which is randomly generated.

`key`
CharField: A hex-encoded secret key of up to 40 bytes. (Default: 20 random bytes)

`step`
PositiveSmallIntegerField: The time step in seconds. (Default: 30)

`t0`
BigIntegerField: The Unix time at which to begin counting steps. (Default: 0)

`digits`
PositiveSmallIntegerField: The number of digits to expect in a token (6 or 8). (Default: 6)

`tolerance`
PositiveSmallIntegerField: The number of time steps in the past or future to allow. For example, if this is 1, we'll accept any of three tokens: the current one, the previous one, and the next one. (Default: 1)

`drift`
SmallIntegerField: The number of time steps the prover is known to deviate from our clock. If *OTP_TOTP_SYNC* is `True`, we'll update this any time we match a token that is not the current one. (Default: 0)

`last_t`
BigIntegerField: The time step of the last verified token. To avoid verifying the same token twice, this will be updated on each successful verification. Only tokens at a higher time step will be verified subsequently. (Default: -1)

`bin_key`
The secret key as a binary string.

`config_url`
A URL for configuring Google Authenticator or similar.

See https://github.com/google/google-authenticator/wiki/Key-Uri-Format. The issuer is taken from *OTP_TOTP_ISSUER*, if available.

class `django_otp.plugins.otp_totp.admin.`**`TOTPDeviceAdmin`**(*model*, *admin_site*)

`ModelAdmin` for *TOTPDevice*.

TOTP Settings

OTP_TOTP_ISSUER Default: `None`

The `issuer` parameter for the otpauth URL generated by *config_url*.

OTP_TOTP_SYNC Default: `True`

If true, then TOTP devices will keep track of the difference between the prover's clock and our own. Any time a *TOTPDevice* matches a token in the past or future, it will update *drift* to the number of time steps that the two sides are out of sync. For subsequent tokens, we'll slide the window of acceptable tokens by this number.

Static Devices

class django_otp.plugins.otp_static.models.**StaticDevice**(**args*, ***kwargs*)
A static *Device* simply consists of random tokens shared by the database and the user. These are frequently used as emergency tokens in case a user's normal device is lost or unavailable. They can be consumed in any order; each token will be removed from the database as soon as it is used.

This model has no fields of its own, but serves as a container for *StaticToken* objects.

token_set
The RelatedManager for our tokens.

class django_otp.plugins.otp_static.models.**StaticToken**(**args*, ***kwargs*)
A single token belonging to a *StaticDevice*.

device
ForeignKey: A foreign key to *StaticDevice*.

token
CharField: A random string up to 16 characters.

static random_token()
Returns a new random string that can be used as a static token.

Return type str

class django_otp.plugins.otp_static.admin.**StaticDeviceAdmin**(*model*, *admin_site*)
ModelAdmin for *StaticDevice*.

The static plugin also includes a management command called `addstatictoken`, which will add a single static token to any account. This is useful for bootstrapping and emergency access. Run `manage.py addstatictoken -h` for details.

Email Devices

class django_otp.plugins.otp_email.models.**EmailDevice**(**args*, ***kwargs*)
A *Device* that delivers a token to the user's registered email address (`user.email`). This is intended for demonstration purposes; if you allow users to reset their passwords via email, then this provides no security benefits.

key
CharField: A hex-encoded secret key of up to 40 bytes. (Default: 20 random bytes)

class django_otp.plugins.otp_email.admin.**EmailDeviceAdmin**(*model*, *admin_site*)
ModelAdmin for *EmailDevice*.

Other Plugins

The framework author also maintains a couple of other plugins for less common devices. Third-party plugins are not listed here.

- django-otp-yubikey supports YubiKey USB devices.

- django-otp-twilio supports delivering OTPs via Twilio's SMS service.

Settings

OTP_LOGIN_URL

Default: alias for `LOGIN_URL`

The URL where requests are redirected for two-factor authentication, especially when using the *`otp_required()`* decorator.

Glossary

authenticated A user whose credentials have been accepted by Django's authentication API is considered authenticated.

device A mechanism by which a user can acquire an OTP. This might correspond to a physical device dedicated to such a purpose, a virtual device such as a smart phone app, or even a set of stored single-use tokens.

OTP A one-time password. This is a generated value that a user can present as evidence of their identity. OTPs are only valid for a single use or, in some cases, for a strictly limited period of time.

prover An entity that is using an OTP to prove its identity. For example, a user who is providing an OTP token.

token An encoded OTP. Some OTPs consist of structured data, in which case they will be encoded into a printable string for transport.

two-factor authentication An authentication policy that requires a user to present two proofs of identity. The first is typically a password and the second is frequently tied to some physical device in the user's possession.

verified A user whose credentials have been accepted by Django's authentication API and also by a registered OTP device is considered verified.

verifier An entity that verifies tokens generated by a prover. For example, a web service that accepts OTPs as proof of identity.

CHAPTER 2

Authentication and Authorization

This section describes the process for verifying users against their registered OTP devices as well as limiting access based on this verification.

Authenticating Users

Soliciting an OTP token from a user is more complicated than soliciting a password. For one thing, each user may have any number of OTP devices registered to their account and the token itself won't tell us which one is intended. And, of course, we won't even know which devices we should check until after we've identified the user based on their username and password. Complicating this further is the fact some plugins are interactive, in which case verifying the user is at least a two-step process.

Verifying a user can happen in one or two stages. One option is to require an OTP up front along with a password. Alternatively, we can accept single-factor authentication initially, but allow (or require) the user to provide a second factor later on. The following sections begin with the simpler strategies and proceed to the lower-level APIs that will allow you to implement more complex policies.

The Easy Way

`django_otp.views.`**`login`**(*request*, ***kwargs*)

This is a replacement for `django.contrib.auth.views.login()` that requires two-factor authentication. It's slightly clever: if the user is already authenticated but not verified, it will only ask the user for their OTP token. If the user is anonymous or is already verified by an OTP device, it will use the full username/password/token form. In order to use this, you must supply a template that is compatible with both *`OTPAuthenticationForm`* and *`OTPTokenForm`*. This is a good view for *`OTP_LOGIN_URL`*.

Parameters are the same as `login()` except that this view always overrides `authentication_form`.

The Authentication Form

Django provides some high-level APIs to make it easy to authenticate users. If you're accustomed to using Django's built-in login view, this section will show you how to turn it into a two-factor login view.

In Django, user authentication actually takes place not in a view, but in an `AuthenticationForm` or a subclass. If you're using Django's `built-in login view`, you're already using the default AuthenticationForm. This form performs authentication as part of its validation; validation only succeeds if the supplied credentials pass `django.contrib.auth.authenticate()`.

If you want to require two-factor authentication in the default login view, the easiest way is to use *`django_otp.forms.OTPAuthenticationForm`* instead. This form includes additional fields and behavior to solicit an OTP token from the user and verify it against their registered devices. This form's validation only succeeds if it is able to both authenticate the user with the username and password and also verify them with an OTP token. The form can be used with `django.contrib.auth.views.login()` simply by passing it in the `authentication_form` keyword parameter:

```
from django_otp.forms import OTPAuthenticationForm

urlpatterns = patterns('django.contrib.auth.views',
    url(r'^accounts/login/$', 'login', kwargs={'authentication_form':␣
↪OTPAuthenticationForm}),
)
```

class `django_otp.forms.`**`OTPAuthenticationForm`**(*request=None*, **args*, ***kwargs*)

This form provides the one-stop OTP authentication solution. It should only be used when two-factor authentication is required: it does not have an OTP-optional mode. The form has four fields:

1. `username` is inherited from `AuthenticationForm`.
2. `password` is inherited from `AuthenticationForm`.
3. `otp_device` uses a `Select` to allow the user to choose one of their registered devices. It will be empty as long as `form.get_user()` is `None` and should generally be omitted from the template in that case.
4. `otp_token` is the field for entering an OTP token. It should always be included.

In addition, if `form.get_user()` is not `None`, the template should include an additional submit button named `otp_challenge`. Pressing this button when `otp_device` is set to an interactive device will cause us to generate a challenge value for the user. Pressing the challenge button with a non-interactive device selected has no effect.

The intended behavior of the form is as follows:

- Initially the `username`, `password`, and `otp_token` fields should be visible. Validation of `username` and `password` is the same as for `AuthenticationForm`. If we are able to authenticate the user based on username and password, then one of two things happens:
  - If the user submitted an OTP token, we will enumerate all of the user's OTP devices, asking each one to verify it in turn. If one of them succeeds, then authentication is fully successful and the user is logged in.
  - If the user did not submit an OTP token or none of user's devices accepted it, then a `ValidationError` is raised.
- In either case, as long as the user is authenticated by their password, `form.get_user()` will return the authenticated `User` object. From here on, this documentation assumes that username/password authentication succeeds on all subsequent submissions. If validation was not successful, then the form will be displayed again and this time the template should be sure to include the (now populated) `otp_device` field as well as the `otp_challenge` submit button.

- The user will then have to choose a specific device to authenticate against (or accept the default). If they press the `otp_challenge` button, we will ask that device to generate a challenge. The device will return a message for the user, which will be incorporated into the `ValidationError` message.
- If the user presses any other submit button, we will authenticate the username and password as always and then verify the OTP token against the chosen device. When that succeeds, authentication and verification are successful and the user is logged in.

Following is a sample template snippet that's designed for *`OTPAuthenticationForm`*:

```
<form action="." method="POST">
    <div class="form-row"> {{ form.username.errors }}{{ form.username.label_tag }}{{
↪form.username }} </div>
    <div class="form-row"> {{ form.password.errors }}{{ form.password.label_tag }}{{
↪form.password }} </div>
    {% if form.get_user %}
    <div class="form-row"> {{ form.otp_device.errors }}{{ form.otp_device.label_tag }}
↪{{ form.otp_device }} </div>
    {% endif %}
    <div class="form-row"> {{ form.otp_token.errors }}{{ form.otp_token.label_tag }}{{
↪{ form.otp_token }} </div>
    <div class="submit-row">
        <input type="submit" value="Log in"/>
        {% if form.get_user %}<input type="submit" name="otp_challenge" value="Get
↪Challenge" />{% endif %}
    </div>
</form>
```

The Admin Site

In addition to providing *`OTPAuthenticationForm`* for your normal login views, django-otp includes an `AdminSite` subclass for admin integration.

class `django_otp.admin.`**`OTPAdminSite`**(*name='otpadmin'*)

This is an `AdminSite` subclass that requires two-factor authentication. Only users that can be verified by a registered OTP device will be authorized for this admin site. Unverified users will be treated as if `is_staff` is `False`.

`has_permission`(*request*)

In addition to the default requirements, this only allows access to users who have been verified by a registered OTP device.

`login_form`

alias of `OTPAdminAuthenticationForm`

`login_template` = **'otp/admin19/login.html'**

We automatically select a modified login template based on your Django version. If it doesn't look right, your version may not be supported, in which case feel free to replace it.

`name` = **'otpadmin'**

The default instance name of this admin site. You should instantiate this class as `OTPAdminSite(OTPAdminSite.name)` to make sure the admin templates render the correct URLs.

class `django_otp.admin.`**`OTPAdminAuthenticationForm`**(**args*, ***kwargs*)

An `AdminAuthenticationForm` subclass that solicits an OTP token. This has the same behavior as *`OTPAuthenticationForm`*.

See the Django AdminSite documentation for more on installing custom admin sites. If you want to copy the default admin site into an *OTPAdminSite*, we find that the following works well. Note that it relies on a private property, so use this at your own risk:

```
otp_admin_site = OTPAdminSite(OTPAdminSite.name)
for model_cls, model_admin in admin.site._registry.iteritems():
    otp_admin_site.register(model_cls, model_admin.__class__)
```

The Token Form

If you already have an authenticated user and you just want to ask for an OTP token to verify, you can use *django_otp.forms.OTPTokenForm*.

class django_otp.forms.**OTPTokenForm**(*user*, *request=None*, **args*, ***kwargs*)

A form that verifies an authenticated user. It looks very much like *OTPAuthenticationForm*, but without the username and password. The first argument must be an authenticated user; you can use this in place of AuthenticationForm by currying it:

```
from functools import partial

from django.contrib.auth.decoratorrs import login_required
from django.contrib.auth.views import login


@login_required
def verify(request):
    form_cls = partial(OTPTokenForm, request.user)

    return login(request, template_name='my_verify_template.html', authentication_
↪form=form_cls)
```

This form will ask the user to choose one of their registered devices and enter an OTP token. Validation will succeed if the token is verified. See *OTPAuthenticationForm* for details on writing a compatible template (leaving out the username and password, of course).

Parameters

- **user** (User) – An authenticated user.
- **request** (HttpRequest) – The current request.

The Low-Level API

django_otp.**devices_for_user**(*user*, *confirmed=True*)

Return an iterable of all devices registered to the given user.

Returns an empty iterable for anonymous users.

Parameters

- **user** (User) – standard or custom user object.
- **confirmed** – If None, all matching devices are returned. Otherwise, this can be any true or false value to limit the query to confirmed or unconfirmed devices, respectively.

Return type iterable

django_otp.**user_has_device**(*user*, *confirmed=True*)
Return `True` if the user has at least one device.

Returns `False` for anonymous users.

Parameters

- **user** (`User`) – standard or custom user object.
- **confirmed** – If `None`, all matching devices are considered. Otherwise, this can be any true or false value to limit the query to confirmed or unconfirmed devices, respectively.

django_otp.**match_token**(*user*, *token*)
Attempts to verify a *token* on every device attached to the given user until one of them succeeds. When possible, you should prefer to verify tokens against specific devices.

Parameters

- **user** (`User`) – The user supplying the token.
- **token** (*string*) – An OTP token to verify.

Returns The device that accepted `token`, if any.

Return type *Device* or `None`

django_otp.**login**(*request*, *device*)
Persist the given OTP device in the current session. The device will be rejected if it does not belong to `request.user`.

This is called automatically any time `django.contrib.auth.login()` is called with a user having an `otp_device` atribute. If you use Django's `login()` view with the django-otp authentication forms, then you won't need to call this.

Parameters

- **request** (`HttpRequest`) – The HTTP request
- **device** (*Device*) – The OTP device used to verify the user.

class django_otp.models.**Device**(**args*, ***kwargs*)
Abstract base model for a *device* attached to a user. Plugins must subclass this to define their OTP models.

> **Warning:** OTP devices are inherently stateful. For example, verifying a token is logically a mutating operation on the device, which may involve incrementing a counter or otherwise consuming a token. A device must be committed to the database before it can be used in any way.

user
ForeignKey: Foreign key to your user model, as configured by `AUTH_USER_MODEL` (`User` by default).

name
CharField: A human-readable name to help the user identify their devices.

confirmed
BooleanField: A boolean value that tells us whether this device has been confirmed as valid. It defaults to `True`, but subclasses or individual deployments can force it to `False` if they wish to create a device and then ask the user for confirmation. As a rule, built-in APIs that enumerate devices will only include those that are confirmed.

objects
A *DeviceManager*.

generate_challenge()
Generates a challenge value that the user will need to produce a token. This method is permitted to have side effects, such as transmitting information to the user through some other channel (email or SMS, perhaps). And, of course, some devices may need to commit the challenge to the databse.

Returns A message to the user. This should be a string that fits comfortably in the template `'OTP Challenge: {0}'`. This may return `None` if this device is not interactive.

Return type string or `None`

Raises Any `Exception` is permitted. Callers should trap `Exception` and report it to the user.

is_interactive()
Returns `True` if this is an interactive device. The default implementation returns `True` if *generate_challenge()* has been overridden, but subclasses are welcome to provide smarter implementations.

Return type bool

verify_token(*token*)
Verifies a token. As a rule, the token should no longer be valid if this returns `True`.

Parameters **token** (*string*) – The OTP token provided by the user.

Return type bool

class django_otp.models.**DeviceManager**
The `Manager` object installed as `Device.objects`.

devices_for_user(*user*, *confirmed=None*)
Returns a queryset for all devices of this class that belong to the given user.

Parameters

- **user** (`User`) – The user.
- **confirmed** – If `None`, all matching devices are returned. Otherwise, this can be any true or false value to limit the query to confirmed or unconfirmed devices, respectively.

Authorizing Users

If you design your site to always require OTP verification in order to log in, then your authorization policies don't need to change. `request.user.is_authenticated()` will be effectively synonymous with `request.user.is_verified()`. If, on the other hand, you anticipate having both verified and unverified users on your site, you're probably intending to limit access to some resources to verified users only. The primary tool for this is otp_required:

@django_otp.decorators.**otp_required**([*redirect_field_name='next'*, *login_url=None*, *if_configured=False*])
Similar to `login_required()`, but requires the user to be *verified*. By default, this redirects users to *OTP_LOGIN_URL*.

Parameters **if_configured** (*bool*) – If `True`, an authenticated user with no confirmed OTP devices will be allowed. Default is `False`.

If you need more fine-grained control over authorization decisions, you can use `request.user.is_verified()` to determine whether the user has been verified by an OTP device. if `is_verified()` is true, then `request.user.otp_device` will be set to the *Device* object that verified the user. This can be useful if you want to include the name of the verifying device in the UI.

If you want to use OTPs to establish trusted user agents (e.g. a browser that the user claims is on a private and secure computer), look at django-agent-trust and django-otp-agents.

Managing Devices

django-otp does not include any standard mechanism for managing a user's devices outside of the admin interface. All plugins are expected to include admin integration, which should be sufficient for many sites. Some sites may want to provide users a self-service API to manage devices, but this will be very site-specific. Fortunately, managing a user's devices is just a matter of managing `Device`-derived model objects, so it will be easy to implement. Be sure to note the *warning* about unsaved `Device` objects.

CHAPTER 3

Extending Django-OTP

A django-otp plugin is defined as a Django app that includes at least one model derived from *django_otp.models.Device*. All Device-derived model objects will be detected by the framework and included in the standard forms and APIs.

Writing a Device

A *Device* subclass is only required to implement one method:

`Device.`**`verify_token`**(*token*)

Verifies a token. As a rule, the token should no longer be valid if this returns `True`.

Parameters **`token`** (*string*) – The OTP token provided by the user.

Return type bool

Most devices will also need to define one or more model fields to do anything interesting. Here's a simple implementation of a generic TOTP device:

```
from binascii import unhexlify

from django.db import models

from django_otp.models import Device
from django_otp.oath import totp
from django_otp.util import random_hex, hex_validator


class TOTPDevice(Device):
    key = models.CharField(max_length=80,
                           validators=[hex_validator()],
                           default=lambda: random_hex(20),
                           help_text=u'A hex-encoded secret key of up to 40 bytes.')

    @property
```

```
    def bin_key(self):
        return unhexlify(self.key)

    def verify_token(self, token):
        """
        Try to verify ``token`` against the current and previous TOTP value.
        """
        try:
            token = int(token)
        except ValueError:
            verified = False
        else:
            verified = any(totp(self.bin_key, drift=drift) == token for drift in [0, -
↪1])

        return verified
```

This example also shows some of the *low-level utilities* django_otp provides for OATH and hex-encoded values.

If a device uses a challenge-response algorithm or requires some other kind of user interaction, it should implement an additional method:

`Device.`**`generate_challenge`**`()`

Generates a challenge value that the user will need to produce a token. This method is permitted to have side effects, such as transmitting information to the user through some other channel (email or SMS, perhaps). And, of course, some devices may need to commit the challenge to the databse.

Returns A message to the user. This should be a string that fits comfortably in the template `'OTP Challenge: {0}'`. This may return `None` if this device is not interactive.

Return type string or `None`

Raises Any `Exception` is permitted. Callers should trap `Exception` and report it to the user.

Utilities

django_otp provides several low-level utilities as a convenience to plugin implementors.

django_otp.oath

`django_otp.oath.`**`hotp`**`(`*key*, *counter*, *digits=6*`)`

Implementation of the HOTP algorithm from RFC 4226.

Parameters

- **key** (*bytes*) – The shared secret. A 20-byte string is recommended.
- **counter** (*int*) – The password counter.
- **digits** (*int*) – The number of decimal digits to generate.

Returns The HOTP token.

Return type int

```
>>> key = b'12345678901234567890'
>>> for c in range(10):
...     hotp(key, c)
```

```
755224
287082
359152
969429
338314
254676
287922
162583
399871
520489
```

django_otp.oath.**totp**(*key*, *step=30*, *t0=0*, *digits=6*, *drift=0*)

Implementation of the TOTP algorithm from RFC 6238.

Parameters

- **key** (*bytes*) – The shared secret. A 20-byte string is recommended.
- **step** (*int*) – The time step in seconds. The time-based code changes every `step` seconds.
- **t0** (*int*) – The Unix time at which to start counting time steps.
- **digits** (*int*) – The number of decimal digits to generate.
- **drift** (*int*) – The number of time steps to add or remove. Delays and clock differences might mean that you have to look back or forward a step or two in order to match a token.

Returns The TOTP token.

Return type int

```
>>> key = b'12345678901234567890'
>>> now = int(time())
>>> for delta in range(0, 200, 20):
...     totp(key, t0=(now-delta))
755224
755224
287082
359152
359152
969429
338314
338314
254676
287922
```

class django_otp.oath.**TOTP**(*key*, *step=30*, *t0=0*, *digits=6*, *drift=0*)

An alternate TOTP interface.

This provides access to intermediate steps of the computation. This is a living object: the return values of `t` and `token` will change along with other properties and with the passage of time.

Parameters

- **key** (*bytes*) – The shared secret. A 20-byte string is recommended.
- **step** (*int*) – The time step in seconds. The time-based code changes every `step` seconds.
- **t0** (*int*) – The Unix time at which to start counting time steps.
- **digits** (*int*) – The number of decimal digits to generate.

- **drift** (*int*) – The number of time steps to add or remove. Delays and clock differences might mean that you have to look back or forward a step or two in order to match a token.

```
>>> key = b'12345678901234567890'
>>> totp = TOTP(key)
>>> totp.time = 0
>>> totp.t()
0
>>> totp.token()
755224
>>> totp.time = 30
>>> totp.t()
1
>>> totp.token()
287082
>>> totp.verify(287082)
True
>>> totp.verify(359152)
False
>>> totp.verify(359152, tolerance=1)
True
>>> totp.drift
1
>>> totp.drift = 0
>>> totp.verify(359152, tolerance=1, min_t=3)
False
>>> totp.drift
0
>>> del totp.time
>>> totp.t0 = int(time()) - 60
>>> totp.t()
2
>>> totp.token()
359152
```

t()
The computed time step.

time
The current time.

By default, this returns time.time() each time it is accessed. If you want to generate a token at a specific time, you can set this property to a fixed value instead. Deleting the value returns it to its 'live' state.

token()
The computed TOTP token.

verify(*token*, *tolerance=0*, *min_t=None*)
A high-level verification helper.

Parameters

- **token** (*int*) – The provided token.
- **tolerance** (*int*) – The amount of clock drift you're willing to accommodate, in steps. We'll look for the token at t values in [t - tolerance, t + tolerance].
- **min_t** (*int*) – The minimum t value we'll accept. As a rule, this should be one larger than the largest t value of any previously accepted token.

Return type bool

Iff this returns True, *self.drift* will be updated to reflect the drift value that was necessary to match the token.

django_otp.util

django_otp.util.**hex_validator**(*length=0*)

Returns a function to be used as a model validator for a hex-encoded CharField. This is useful for secret keys of all kinds:

```
def key_validator(value):
    return hex_validator(20)(value)

key = models.CharField(max_length=40, validators=[key_validator], help_text=u'A␣
↪hex-encoded 20-byte secret key')
```

Parameters **length** (*int*) – If greater than 0, validation will fail unless the decoded value is exactly this number of bytes.

Return type function

```
>>> hex_validator()('0123456789abcdef')
>>> hex_validator(8)(b'0123456789abcdef')
>>> hex_validator()('phlebotinum')
Traceback (most recent call last):
    ...
ValidationError: ['phlebotinum is not valid hex-encoded data.']
>>> hex_validator(9)('0123456789abcdef')
Traceback (most recent call last):
    ...
ValidationError: ['0123456789abcdef does not represent exactly 9 bytes.']
```

django_otp.util.**random_hex**(*length=20*)

Returns a string of random bytes encoded as hex. This uses `os.urandom()`, so it should be suitable for generating cryptographic keys.

Parameters **length** (*int*) – The number of (decoded) bytes to return.

Returns A string of hex digits.

Return type str

CHAPTER 4

Change Log

v0.4.0 - July 19, 2017 - Update support matrix

- Fix addstatictoken on Django 1.10+.
- Drop support for versions of Django that are past EOL.

v0.3.14 - May 30, 2017 - addstatictoken fix

- Update addstatictoken command for current Django versions.

v0.3.13 - April 11, 2017 - Pickle compatibility

- Allow verified users to be pickled.

v0.3.12 - April 2, 2017 - Forward compatibility

- Minor fixes for Django 1.11 and 2.0.

v0.3.11 - March 8, 2017 - Built-in QR Code support

- #20: Generate HOTP and TOTP otpauth URLs and corresponding QR Codes. To enable this feature, install `django-otp[qrcode]` or just install the qrcode package.
- Support for Python 2.6 and Django 1.4 were dropped in this version (long overdue).

v0.3.8 - November 27, 2016 - Forward compatbility for Django 2.0

- Treat `is_authenticated` and `is_anonymous` as properties in Django 1.10 and later.
- Add explict on_delete behavior for all foreign keys.

v0.3.7 - September 24, 2016 - Convenience API

- Added a convenience API for verifying TOTP tokens: *`django_otp.oath.TOTP.verify()`*.

v0.3.6 - September 4, 2016 - Django 1.10

- #11: Don't break the laziness of `request.user`.
- #16: Improved error message for invalid tokens.
- #17: Support the new middleware API in Django 1.10.

v0.3.5 - April 13, 2016 - Fix default TOTP key

- The default (random) key for a new TOTP device is now forced to a unicode string.

v0.3.4 - January 10, 2016 - Python 3 cleanup

- All modules include all four Python 3 __future__ imports for consistency.
- Migrations no longer have byte strings in them.

v0.3.3 - October 15, 2015 - Django 1.9

- Fix the addstatictoken management command under Django 1.9.

v0.3.2 - October 11, 2015 - Django 1.8

- Stop importing models into the root of the package.
- Use ModelAdmin.raw_id_fields for foreign keys to users.
- General cleanup and compatibility with Django 1.9a1.

v0.3.1 - April 3, 2015 - Django 1.8

- Add support for the new app registry, when available.
- Add Django 1.8 to the test matrix and fix a few test bugs.

v0.3.0 - February 7, 2015 - Support Django migrations

- All plugins now have both Django and South migrations. Please see the upgrade notes for details on upgrading from previous versions.

v0.2.7 - April 26, 2014 - Fix for Custom user models with South

- Updated the otp_totp South migrations to support custom user models. Thanks to https://bitbucket.org/robirichter.

v0.2.6 - April 18, 2014 - Fix for Python 3.2 with South

- Removed South-generated unicode string literals.

v0.2.4 - April 15, 2014 - TOTP plugin fix (migration warning)

- Per the RFC, *TOTPDevice* will no longer verify the same token twice.
- Cosmetic fixes to the admin login form on Django 1.6.

Warning: This includes a model change in TOTPDevice. If you are upgrading and your project uses South, you should first convert it to South with `manage migrate otp_totp 0001 --fake`. If you're not using South, you will need to generate and run the appropriate SQL manually.

v0.2.3 - March 3, 2014 - Fix pickling

- OTPMiddleware no longer interferes with pickling request.user.

v0.2.2 - December 31, 2013 - Require Django 1.4.2

- Update Django requirement to 1.4.2, the first version with django.utils.six.

v0.2.1 - November 19, 2013 - Bug fix

- Fix unicode representation of devices in some exotic scenarios.

v0.2.0 - November 10, 2013 - Django 1.6

- Now supports Django 1.4 to 1.6 on Python 2.6, 2.7, 3.2, and 3.3. This is the first release for Python 3.

v0.1.8 - August 20, 2013 - user_has_device API

- Add *django_otp.user_has_device()* to detect whether a user has any devices configured. This change supports a fix in django-otp-agents 0.1.4.

v0.1.7 - July 3, 2013 - Decorator improvement

- Add if_configured argument to *otp_required()*.

v0.1.6 - May 9, 2013 - Unit test improvements

- Major unit test cleanup. Tests should pass or be skipped under all supported versions of Django, with or without custom users and timzeone support.

v0.1.5 - May 8, 2013 - OTPAdminSite improvement

- OTPAdminSite now selects an apporpriate login template automatically, based on the current Django version. Django versions 1.3 to 1.5 are currently supported.
- Unit test cleanup.

v0.1.3 - March 10, 2013 - Django 1.5 compatibility

- Add support for custom user models in Django 1.5.
- Stop using `Device.objects`: Django doesn't allow access to an abstract model's manager any more.

v0.1.2 - October 8, 2012 - Bug fix

- Fix an exception when an empty login form is submitted.

v0.1.0 - August 20, 2012 - Initial Release

Initial release.

CHAPTER 5

License

Python Module Index

d

Index