django-twilio Documentation

Release 0.8.0

Randall Degges

Jun 08, 2017

Contents

Integrate Twilio into your Django apps with ease.

CHAPTER 1

Meta

- Author: Randall Degges
- Email: rdegges@gmail.com
- Maintainer: Paul Hallett
- Email: paul@twilio.com
- Status: active development, stable, maintained

Twilio is a powerful HTTP API that allows you to build powerful voice & SMS apps. The goal of this library is to help make building telephony applications as simple as humanly possible with twilio.

CHAPTER 2

User's Guide

This part of the documentation contains an extensive walk through of django-twilio. It begins with some background information about django-twilio, then focuses on step-by-step integration of django-twilio into your existing Django web applications.

Foreword

Read this before you get started with `django-twilio`. This will hopefully answer some questions about the purpose of the project, and why you should (or shouldn't) be using it.

Purpose

Building telephony applications has always been something of a complex and time consuming task for developers. With the advent of Twilio, developers were able to build large scale telephony applications for the first time, without the massive learning curve associated with traditional telephony development.

While Twilio's APIs allow you to build powerful voice and SMS apps in your favorite programming language, it can still be quite difficult and time consuming to roll out your own telephony apps for your Django-powered website.

The core purpose of `django-twilio` is to abstract away as much telephony knowledge as possible, so that you can focus on the functionality and logic of your telephony app, and have it seamlessly integrate into your website without confusion.

Use Case

`django-twilio` is a complete solution for anyone who wants to integrate voice or SMS functionality into their Django website without requiring any additional infrastructure.

Here are some common use cases:

- You want to build one or more telephone conference rooms for talking with co-workers or clients.

- You want be able to accept SMS messages from your clients, and respond to them programmatically.
- You want to record important phone calls (incoming or outgoing) and store the recordings for analysis.
- You want to track telephone marketing campaigns, and review detailed data.
- You want to build rich, interactive telephone systems. (EG: "Press 1 to talk with a sales agent, press 2 to schedule a reservation, press 3 to purchase a hat.")
- And many more.

Prerequisite Knowledge

Before getting started with `django-twilio`, it will serve you best to read the How it Works page on Twilio's website. That describes the architecture and API flow that all of your applications will be using, and that `django-twilio` will help to abstract.

`django-twilio` also depends on the official Twilio python library, so you may want to familiarize yourself with their docs before continuing so you have a good idea of how things work.

Other than that, you're good to go!

Installation

Installing `django-twilio` is simple.

Firstly, download and install the `django-twilio` package.

The easiest way is with pip

```
$ pip install django-twilio
```

Requirements

`django-twilio` will automatically install the official twilio-python library. The `twilio-python` library helps you rapidly build Twilio applications, and it is heavily suggested that you check out that project before using `django-twilio`.

If you are using `django-twilio` with Django version 1.6.* or less, you **will** need to install South 1.0. Older versions of South will not work. This is due to to recent database migration changes that have happened with Django 1.7, you can read more about the changes here. You will also need to install South 1.0 or higher if you are using Python 3.

If you already have South installed, upgrading is easy:

```
$ pip install --upgrade South
```

Django Integration

After `django-twilio` is installed, add it to your `INSTALLED_APPS` tuple in your `settings.py` file:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'django_twilio',
    ...
)
```

Note: Please note the underscore!

Databases for Django 1.6 or lower

To use `django-twilio` with Django 1.6.* or less, you will need to install South 1.0 with the following terminal command:

```
$ pip install South==1.0
```

Or upgrade South to version 1.0:

```
$ pip install --upgrade South
```

Add south to your installed apps:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'django_twilio',
    'South'
    ...
)
```

and run the default `syncdb` command:

```
$ python manage.py syncdb
```

To sync the `django-twilio` models to the database run:

```
$ python manage.py migrate django_twilio
```

Databases for Django 1.7

Django 1.7 has built in migrations, so there is no need to install any third-party schema management tool. To sync the `django-twilio` models with your Django 1.7 project, just run:

```
$ python manage.py migrate
```

Upgrading

Upgrading `django-twilio` gracefully is easy using pip:

```
$ pip install --upgrade django-twilio
```

Then you just need to update the models:

```
$ python manage.py migrate django_twilio
```

Authentication Token Management

`django-twilio` offers multiple ways to add your Twilio credentials to your Django application.

The `TWILIO_ACCOUNT_SID` and `TWILIO_AUTH_TOKEN` variables can be found by logging into your Twilio account dashboard. These tokens are used to communicate with the Twilio API, so be sure to keep these credentials safe!

The order in which Django will check for credentials is:

1. Environment variables in your environment
2. Settings variables in the Django settings

We recommend using environment variables so you can keep secret tokens out of your code base. This practice is far more secure.

Using `virtualenv`, open up your `/bin/activate.sh` file and add the following to the end:

```
export TWILIO_ACCOUNT_SID=XXXXXXXXXXXX
export TWILIO_AUTH_TOKEN=YYYYYYYYYYY
```

You'll need to deactivate and restart your virtualenv for it to take effect.

To use settings variables, you'll need to add them to your `settings.py` file:

```
TWILIO_ACCOUNT_SID = 'ACXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
TWILIO_AUTH_TOKEN = 'YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY'
```

Note: Storing tokens in `settings.py` is very bad for security! Only do this if you are certain you will not be sharing the file publicly.

And optionally add the default caller:

```
TWILIO_DEFAULT_CALLERID = 'NNNNNNNNNN'
```

If you specify a value for `TWILIO_DEFAULT_CALLERID`, then all SMS and voice messages sent through `django-twilio` functions will use the default caller id as a convenience.

You can create a Credential object to store your variables if you want to use multiple Twilio accounts or provide your users with Twilio compatibility.

When you want to use the credentials in a Credential object you need to manually build a TwilioRestClient like so:

```
from twilio.rest import TwilioRestClient
from django_twilio.utils import discover_twilio_credentials

from django.contrib.auth.models import User

my_user = User.objects.get(pk=USER_ID)

account_sid, auth_token = discover_twilio_credentials(my_user)

# Here we'll build a new Twilio_client with different credentials
twilio_client = TwilioRestClient(account_sid, auth_token)
```

Settings

Django-twilio has various settings that can be used to turn features on and off.

Here we explain each setting, its requirement, and why you might want to use it.

Each setting should be placed in your `settings.py` file, except the `TWILIO_ACCOUNT_SID` and `TWILIO_AUTH_TOKEN` variables, which should be kept in your environment variables to ensure their security.

TWILIO_ACCOUNT_SID (REQUIRED)

The `TWILIO_ACCOUNT_SID` setting is required, and will throw an exception if it is not configured correctly:

```
TWILIO_ACCOUNT_SID = 'SIDXXXXXXXXXXXXXXX'
```

This setting is used to authenticate your Twilio account.

> **Note:** This setting can be placed in your Python environment instead, if you wish. This is a far more secure option, and is recommended.

To add this setting to your environment, using `virtualenv` open up your `/bin/activate.sh` file and add the following to the end:

```
export TWILIO_ACCOUNT_SID=XXXXXXXXXXXX
```

TWILIO_AUTH_TOKEN (REQUIRED)

The `TWILIO_AUTH_TOKEN` setting is required, and will throw an exception if it is not configured correctly:

```
TWILIO_AUTH_TOKEN = 'ATXXXXXXXXXXXXXXX'
```

This setting is used to authenticate your Twilio account.

> **Note:** This setting can be placed in your Python environment instead, if you wish. This is a far more secure option, and is recommended.

Using `virtualenv` open up your `/bin/activate.sh` file and add the following to the end:

```
export TWILIO_AUTH_TOKEN=XXXXXXXXXXXXX
```

DJANGO_TWILIO_FORGERY_PROTECTION (optional)

The `DJANGO_TWILIO_FORGERY_PROTECTION` setting is optional. This setting is a boolean, and should be placed in the `settings.py` file:

DJANGO_TWILIO_FORGERY_PROTECTION = False

This setting is used to determine the forgery protection used by `django-twilio`. If not set, this will always be the opposite of `settings.DEBUG`, so in production mode the forgery protection will be on, and in debug mode the protection will be turned off.

It is recommended that you leave the protection off in debug mode, but this setting will allow you to test the forgery protection with a tool like ngrok

DJANGO_TWILIO_BLACKLIST_CHECK (optional)

The `DJANGO_TWILIO_BLACKLIST_CHECK` setting is optional. This setting is a boolean, and should be placed in the `settings.py` file:

```
DJANGO_TWILIO_BLACKLIST_CHECK = True
```

This setting will determine if `django-twilio` will run a database query comparing the incoming request `From` attribute with any potential `Caller` objects in your database.

In short: turning this off will remove an unnecessary database query if you are not using any blacklists.

Decorators

One of the key features of `django-twilio` is making it easy to build Django views that return TwiML instructions back to Twilio, without having to deal with all the complex security issues.

All-In-One Decorator

The most useful decorator that ships with `django-twilio` is `twilio_view`, which will make your life much easier.

The `django_twilio.decorators.twilio_view` decorator:

1. Protects your views against forgery, and ensures that the request which hits your view originated from Twilio's servers. This way, you don't have to worry about fake requests hitting your views that may cause you to launch calls, or waste any money on fraudulent activity.
2. Ensures your view is CSRF exempt. Since Twilio will always POST data to your views, you'd normally have to explicitly declare your view CSRF exempt. The decorator does this automatically.
3. Enforces a blacklist. If you've got any `django_twilio.models.Caller` objects who are blacklisted, any service requests from them will be rejected.

> **Note:** You can manage your blacklist via the Django admin panel (if you have it enabled). `django-twilio` provides a `Caller` admin hook that allows you to create new callers, and blacklist them if you wish.

4. Allows you to (optionally) return raw TwiML responses without building an `HttpResponse` object. This can save a lot of redundant typing.

Example usage

Let's take a look at an example:

```
from twilio import twiml
from django_twilio.decorators import twilio_view

@twilio_view
def reply_to_sms_messages(request):
    r = twiml.Response()
    r.message('Thanks for the SMS message!')
    return r
```

In the example above, we built a view that Twilio can POST data to, and that will instruct Twilio to send a SMS message back to the person who messaged us saying, "Thanks for the SMS message!".

Class based view example

Here's the same thing as above, using a class-based view:

```
from twilio import twiml

from django.views.generic import View
from django.utils.decorators import method_decorator

from django_twilio.decorators import twilio_view

class ThanksView(View):

    @method_decorator(twilio_view)
    def dispatch(self, request, *args, **kwargs):
        return super(ResponseView, self).dispatch(request, *args, **kwargs)

    def post(self, request):
        r = twiml.Response()
        r.message('Thanks for the SMS message!')
        return r
```

How Forgery Protection Works

Forgery protection is extremely important when writing Twilio code. Since your code will be doing stuff that costs money (sending calls, SMS messages, etc.), ensuring all incoming HTTP requests actually originate from Twilio is really important.

The way `django-twilio` implements forgery protection is by checking for a specific flag in the django configuration settings:

```
DJANGO_TWILIO_FORGERY_PROTECTION = False
```

If this setting is not present, it will default to the **opposite** of `settings.DEBUG`; in debug mode, forgery protection will be off.

This behavior has been specifically implemented this way so that, while in development mode, you can:

- Unit test your Twilio views without getting permission denied errors.
- Test your views out locally and make sure they return the code you want.

Because of this, it is extremely important that when your site goes live, you ensure that `settings.DEBUG = False` and `DJANGO_TWILIO_FORGERY_PROTECTION = True`.

Requests

Django-twilio ships with some functionality to help handle inbound HTTP requests from Twilio. Sometimes it is confusing to understand what Twilio will send to your view endpoints, so these tools are designed to make that more explicit and easier to understand.

decompose()

The `decompose` function will strip out the Twilio-specific POST parameters from a Django HttpRequest object and present them back as a TwilioRequest object. Each POST parameter will be an attribute on the new TwilioRequest class. This makes it much easier to discover the parameters sent to you from Twilio and access them without having to use the HttpRequest object. The `decompose` function can also discover the type of Twilio request (''Message" or ''Voice'') based on the parameters that are sent to you. This means you could build a single view endpoint and route traffic based on the type of Twilio request you receive!

Example usage

Here is an example:

```
from twilio import twiml
from django_twilio.decorators import twilio_view
# include decompose in your views.py
from django_twilio.request import decompose

@twilio_view
def inbound_view(request):

    response = twiml.Response()

    # Create a new TwilioRequest object
    twilio_request = decompose(request)

    # See the Twilio attributes on the class
    twilio_request.to
    # >>> '+44123456789'

    # Discover the type of request
    if twilio_request.type is 'message':
        response.message('Thanks for the message!')
        return response

    # Handle different types of requests in a single view
    if twilio_request.type is 'voice':
        return voice_view(request)
```

```
    return response
```

Views

In order to help speed up development of your telephony application, `django-twilio` ships with a few useful views that you can plug straight into your project's URL configuration and immediately use!

Saying Stuff

In a majority of telephony apps, you'll want to say something. It can be tedious to record your own voice prompts for every bit of call flow, which is why you'll want to use the `django_twilio.views.say` view.

The `django_twilio.views.say` view allows you to simply "say stuff" in a variety of languages (in either a male or female voice).

Hello, World!

Let's take a look at a *classic* example:

```
# urls.py
urlpatterns = patterns(
    '',
    # ...
    url(r'^hello_world/$', 'django_twilio.views.say', {
        'text': 'Hello, world!'
    }),
    # ...
)
```

Hook a Twilio number up to that URL, and you'll hear a man say "Hello, world!" when called. Nice!

Changing the Voice (and Language)

By default, Twilio reads off all text in the English language in a man's voice. But it's easy to change that. In the example below, we'll say "goodbye" in Spanish, with a female voice:

```
# urls.py
urlpatterns = patterns(
    '',
    # ...
    url(r'^goodbye/$', 'django_twilio.views.say', {
        'text': 'Adios!',
        'voice': 'woman',
        'language': 'es',
    }),
    # ...
)
```

Simple, right?

Repeating Text

On occasion, you'll also want to repeat some text, without copy/paste. In this situation, you can simply specify an optional `loop` parameter:

```
# urls.py
urlpatterns = patterns(
    '',
    # ...
    url(r'^lol/$', 'django_twilio.views.say', {
        'text': 'lol',
        'loop': 0,  # 0 = Repeat forever, until hangup :)
    }),
    # ...
)
```

In this example, we'll just keep repeating "lol" to the caller until they hang up.

For more information, be sure to read the API docs on `django_twilio.views.say`.

Playing Audio

`django-twilio` makes it easy to play audio files to callers. Below, we'll look at two examples which demonstrate how to do so using the excellent `django_twilio.views.play` view.

Playing a WAV File

In this example, we'll play a simple WAV file to a caller. For simplicity's sake, just assume that this WAV file actually exists:

```
# urls.py
urlpatterns = patterns(
    '',
    # ...
    url(r'^play/$', 'django_twilio.views.play', {
        'url': 'http://mysite.com/greeting.wav',
    })
    # ...
)
```

Assuming the url http://mysite.com/greeting.wav exists, and is a legitimate WAV file, when you call your Twilio application, you should hear the audio file play.

Note: You can play lots of different types of audio files. For a full list of the formats Twilio accepts, look at the API reference material for the `django_twilio.views.play` view.

Looping Audio

In this example, we'll play the same greeting audio clip as we did above, but this time we'll loop it 3 times:

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

```
    # ...
    url(r'^play/$', 'django_twilio.views.play', {
        'url': 'http://mysite.com/greeting.wav',
        'loop': 3,
    })
    # ...
)
```

Not too bad (for no code)!

Grabbing Caller Input

As you begin to build more and more complicated telephony applications, you'll need a way to accept callers' input via their telephone touch pad. For this purpose, `django-twilio` ships with the `django_twilio.views.gather` view.

Below we'll look at a few examples displaying proper usage.

Collecting Touchtone Input

The simplest thing we can do using the `django_twilio.views.gather` view is to collect caller touchtone input until the caller stops hitting keys. To do this, we can write our URL configuration as follows:

```
# urls.py
urlpatterns = patterns(
    '',
    # ...
    url(r'^gather/$', 'django_twilio.views.gather'),
    # ...
)
```

By default, once the caller finishes entering their input, Twilio will send an HTTP POST request to the same URL. So in our example above, if a caller enters '666#', then Twilio would send a POST request to our `/gather/` URL with a `Digits` parameter that contains the value `'666#'`.

Redirect After Collecting Input

Let's say that instead of POSTing the caller's input to the same URL, you want to instead POST the data to another URL (or view). No problem! In fact, we'll even tell Twilio to send the data in GET format instead of POST:

```
# urls.py
urlpatterns = patterns(
    '',
    # ...
    url(r'^gather/$', 'django_twilio.views.gather', {
        'action': '/process_input/',
        'method': 'GET',
    }),
    url(r'^process_input/$', 'mysite.myapp.views.process'),
    # ...
)

# mysite.myapp.views.py
from django.http import HttpResponse
```

```
def process(request):
    print(request.GET)  # Output GET data to terminal (for debug).
    return HttpResponse()
```

If you test out this application, you'll see that the caller's input is sent (via HTTP GET) to the `process` view once the input has been collected.

Controlling Input Patterns

Lastly, the `django_twilio.views.gather` view allows you to control various aspects of the input collection process.

Our example below:

- Limits the number of seconds that Twilio will wait for the caller to press another digit to 5. If no input is entered after 5 seconds, then Twilio will automatically pass the data along to the URL specified in the `action` parameter.
- Automatically end the input collection process if the caller hits the '#' key. This way, if the caller enters '12345#', regardless of what the `timeout` parameter is set to, Twilio will pass the data along to the URL specified in the `action` parameter.
- Limit the total amount of digits collected to 10. Once 10 digits have been collected, Twilio will pass the data along to the URL specified in the `action` parameter.

```
# urls.py
urlpatterns = patterns(
    '',
    # ...
    url(r'^gather/$', 'django_twilio.views.gather', {
        'action': '/process_input/',
        'method': 'GET',
        'timeout': 5,
        'finish_on_key': '#',
        'num_digits': 10,
    }),
    # ...
)
```

Recording Calls

`django-twilio` also comes with a built-in call recording view: `django_twilio.views.record`. In the examples below, we'll walk through plugging the `django_twilio.views.record` view into our fictional Django website in a variety of situations.

Record a Call

Let's start simple. In this example, we'll set up our URL configuration to record our call, then hit another URL in our application to provide TwiML instructions for Twilio:

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

```
    url(r'^record/$', 'django_twilio.views.record', {
        'action': '/call_john/',
        'play_beep': True,
    })
    # ...
)
```

If we call our application, Twilio will start recording our call (after playing a beep), then send a POST request to our `/call_john/` URL and continue executing call logic. This allows us to start recording, then continue on passing instructions to Twilio (maybe we'll call our lawyer :)).

Stop Recording on Silence

In most cases, you'll only want to record calls that actually have talking in them. It's pointless to record silence. That's why Twilio provides a `timeout` parameter that we can use with the `django_twilio.views.record` view:

```
# urls.py
urlpatterns = patterns(
    '',
    # ...
    url(r'^record/$', 'django_twilio.views.record', {
        'action': '/call_john/',
        'play_beep': True,
        'timeout': 5,    # Stop recording after 5 seconds of silence
                         # (default).
    })
    # ...
)
```

By default, Twilio will stop the recording after 5 seconds of silence has been detected, but you can easily adjust this number as you see fit. If you're planning on recording calls that may include hold times or other things, then you should probably bump this number up to avoid ending the recording if you get put on hold.

Transcribe Your Call Recording

On occasion, you may want to transcribe your call recordings. Maybe you're making a call to your secretary to describe your TODO list and want to ensure you get it in text format, or maybe you're just talking with colleagues about how to best take over the world. Whatever the situation may be, Twilio has you covered!

In this example, we'll record our call, and force Twilio to transcribe it after we hang up. We'll also give Twilio a URL to POST to once it's finished transcribing, so that we can do some stuff with our transcribed text (maybe we'll email it to ourselves, or something).

Note: Transcribing is a **paid** feature. See Twilio's pricing page for the current rates. Also, Twilio limits transcription time to 2 minutes or less. If you set the `max_length` attribute to > 120 (seconds), then Twilio will **not** transcribe your call, and will instead write a warning to your debug log (in the Twilio web panel).

```
# urls.py
urlpatterns = patterns(
    '',
    # ...
    url(r'^record/$', 'django_twilio.views.record', {
        'action': '/call_john/',
```

```
        'play_beep': True,
        'transcribe': True,
        'transcribe_callback': '/email_call_transcription/',
    })
    # ...
)
```

Sending SMS Messages

In addition to building plug-n-play voice applications, we can also build plug-n-play SMS applications using the `django_twilio.views.message` view. This view allows us to send off arbitrary SMS messages based on incoming Twilio requests.

Reply With an SMS

This example demonstrates a simple SMS reply. Whenever Twilio sends us an incoming request, we'll simply send back an SMS message to the sender:

```
urlpatterns = patterns(
    '',
    # ...
    url(r'^message/$', 'django_twilio.views.message', {
        'message': 'Thanks for the SMS. Talk to you soon!',
    }),
    # ...
)
```

Sending SMS Messages (with Additional Options)

Like most of our other views, the `django_twilio.views.message` view also allows us to specify some other parameters to change our view's behavior:

```
urlpatterns = patterns('',
    # ...
    url(r'^message/$', 'django_twilio.views.message', {
        'message': 'Yo!',
        'to': '+12223334444',
        'sender': '+18882223333',
        'status_callback': '/message/completed/',
    }),
    # ...
)
```

Here, we instruct `django-twilio` to send an SMS message to the caller '+1-222-333-4444' from the sender '+1-888-222-3333'. As you can see, `django-twilio` allows you to fully customize the SMS sending.

Furthermore, the `status_callback` parameter that we specified will be POSTed to by Twilio once it attempts to send this SMS message. Twilio will send us some metadata about the SMS message that we can use in our application as desired.

Sending MMS Messages

MMS enables you to add images, gif and video to standard SMS messages. This allows you to create rich and engaging experiences without the need of a smart phone.

MMS uses the same view as SMS, but we must include a `media` parameter:

```
urlpatterns = patterns(
    '',
    # ...
    url(r'^message/$', 'django_twilio.views.message', {
        'message': 'Oh my glob, amazing!',
        'media': 'http://i.imgur.com/UMlp0iK.jpg',
    }),
    # ...
)
```

Teleconferencing

A common development problem for telephony developers has traditionally been conference rooms – until now. `django-twilio` provides the simplest possible teleconferencing solution, and it only requires a single line of code to implement!

Let's take a look at a few conference patterns, and see how we can easily implement them in our webapp.

Simple Conference Room

Let's say you want to build the world's simplest conference room. It would consist of nothing more than a phone number that, when called, dumps the callers into a conference room and lets them chat with each other. Assuming you've already installed `django-twilio`, here's how you can build this simple conference room:

1. Edit your project's `urls.py` and add the following:

   ```
   urlpatterns = patterns(
       '',
       # ...
       url(r'^conference/(?P<name>\w+)/$', 'django_twilio.views.conference'),
       # ...
   )
   ```

2. Now, log into your Twilio dashboard and create a new app. Point the voice URL of your app at http://yourserver.com/conference/business/.
3. Call your new application's phone number. Twilio will send an HTTP POST request to your web server at `/conference/business/`, and you should be dumped into your new *business* conference room!

Pretty easy eh? No coding even required!

Simple Conference Room with Rock Music

Let's face it, the simple conference you just built was pretty cool, but the music that twilio plays by default is pretty boring. While you're waiting for other participants to join the conference, you probably want to listen to some rock music, right?

Luckily, that's a quick fix!

Open up your `urls.py` once more, and add the following:

```
urlpatterns = patterns(
    '',
    # ...
    url(r'^conference/(?P<name>\w+)/$', 'django_twilio.views.conference', {
        'wait_url': 'http://twimlets.com/holdmusic?Bucket=com.twilio.music.rock',
        'wait_method': 'GET',
    })
    # ...
)
```

`django_twilio.views.conference` allows you to specify optional parameters easily in your URL configuration. Here, we're using the `wait_url` parameter to instruct Twilio to play the rock music while the participant is waiting for other callers to enter the conference. The `wait_method` parameter is simply for efficiency's sake – telling Twilio to use the HTTP GET method (instead of POST, which is the default) allows Twilio to properly cache the sound files.

Conference Room with Custom Greeting

Messing around with hold music is fine and dandy, but it's highly likely that you'll need to do more than that! In the example below, we'll outline how to build a conference room that greets each user before putting them into the conference.

This example shows off how flexible our views can be, and how much we can do with just the built-in `django_twilio.views.conference` view:

```
# urls.py
urlpatterns = patterns(
    '',
    # ...
    url(r'^say_hi/$', 'mysite.views.say_hi'),
    url(r'^conference/(?P<name>\w+)/$', 'django_twilio.views.conference', {
        'wait_url': 'http://yoursite.com/say_hi/',
    })
    # ...
)

# views.py
from twilio import twiml
from django_twilio.decorators import twilio_view

@twilio_view
def say_hi(request):
    r = twiml.Response()
    r.say('Thanks for joining the conference! Django and Twilio rock!')
    return r
```

If you run this example code, you'll notice that when you call your application, Twilio first says "Thanks for joining the conference..." before joining you – pretty neat, eh?

As you can see, this is a great way to build custom logic into your conference room call flow. One pattern that is commonly requested is to play an estimated wait time – a simple project using `django_twilio.views.conference`.

Other Conferencing Goodies

Now may be a good time to check out the API docs for `django_twilio.views.conference` to see all the other goodies available.

Accessing Twilio Resources

Let's say you're building a Twilio application that needs access to all of your account data – stuff like call logs, recordings, SMS messages, etc. `django-twilio` makes accessing this information extremely easy.

The Twilio REST Client

The official Twilio python library provides a really awesome wrapper around Twilio's REST API. Before continuing on, you may want to read the official documentation for this feature, as understanding this will make the documentation below significantly easier to follow.

How it Works

If you are using the Twilio python library by itself (without `django-twilio`), you could see a list of all the phone numbers you have provisioned to your Twilio account by running the following code:

```
from twilio.rest import TwilioRestClient


# Your private Twilio API credentials.
ACCOUNT_SID = 'xxx'
AUTH_TOKEN = 'xxx'

client = TwilioRestClient(ACCOUNT_SID, AUTH_TOKEN)
for number in client.phone_numbers.iter():
    print(number.friendly_name)
```

While this is really convenient, it breaks the Don't Repeat Yourself rule of software engineering by making you manually specify your account credentials.

Since `django-twilio` already requires you to enter your Twilio credentials in your `settings.py` file, `django-twilio` provides a simple wrapper around `TwilioRestClient`: `django_twilio.client.twilio_client`.

The `twilio_client` Wrapper

As mentioned in the previous section, `django-twilio` ships with an instantiated `TwilioRestClient`, so that you can use the Twilio REST API with as little effort as possible. :)

Using `django_twilio.client.twilio_client`, you can print a list of all the phone numbers you have provisioned to your Twilio account by running the following code:

```
from django_twilio.client import twilio_client


for number in twilio_client.phone_numbers.iter():
    print(number.friendly_name)
```

See how you didn't have to worry about credentials or anything? Niiiiice.

Further Reading

Twilio's REST API lets you do a lot of awesome stuff. Among other things, you can:

- View and manage your Twilio account and sub-accounts.
- Manage your Twilio applications.
- Authorize Twilio apps.
- View your call logs.
- View all of your authorized caller IDs.
- Check out your connected apps.
- View all Twilio notifications.
- Get a list of all call recordings.
- View all call transcriptions.
- View all SMS messages.
- View and manage all phone numbers.
- Manage your conference rooms.
- Manage your API sandboxes.
- etc...

To learn more about what you can do, I suggest reading the Twilio REST documentation and the twilio-python REST documentation.

Gotchas

Below is a list (which is being continuously expanded) on things which may "get you" at one point or another. We've done our best to try and make `django-twilio` as easy to use as possible, but sometimes problems are unavoidable!

Help! I Get HTTP 403 Forbidden

There are two common problems that cause `django-twilio` to return HTTP 403 errors in your views:

Forgery Protection

`django-twilio` has built in forgery protection in some decorators to help verify that requests made to any of your Twilio views actually originate from Twilio.

We do this by analyzing HTTP requests sent to your views and comparing a special cryptographic hash. This way, attackers are not able to simply POST data to your views and waste your Twilio resources. Attacks of this nature can be expensive and troublesome.

In the event that HTTP requests to your views are determined to be forged, `django-twilio` will return an HTTP 403 (forbidden) response.

Because of the way this forgery protection works, you'll get HTTP 403 errors when hitting `django-twilio` views if you test them yourself and you have `settings.DEBUG = False`. If you'd like to test your views, be sure to do so with Django's DEBUG setting ON.

Missing Settings

`django-twilio` *requires* that you specify the variables `TWILIO_ACCOUNT_SID` and `TWILIO_AUTH_TOKEN`, either as environment variables, or in your site's `settings` module. These are used to verify the legitimacy of HTTP requests to your Twilio views, and to instantiate the TwilioRestClient.

If these variables are missing, `django-twilio` will raise HTTP 403 (forbidden) errors, because it is unable to determine whether or not the HTTP request originated from Twilio.

To fix this, simply set these environment variables or add them to your settings variables.

How To...

This section documents lots of common "How do I..." questions. Since `django-twilio` has a lot of native functionality, some features that don't necessarily fit into other parts of the documentation are only documented here.

Blacklist Callers

One common problem is dealing with users who abuse services. Regardless of what your telephony app does, it can be dangerous and expensive to run your application without the ability to blacklist users.

Luckily, `django-twilio` provides built-in blacklist functionality, and will fit your needs (whatever they may be).

Hypothetical Abuse Scenario

Let's say you run a Twilio app that forwards calls from your Twilio toll-free number to your private business number. Since you have to pay for Twilio calls, it could potentially become very expensive for you if a caller repeatedly calls your toll-free number, causing you to quickly drain your account balance.

Blacklisting Callers via Django Admin

The simplest way to blacklist callers is via the Django admin panel. If you're using the Django admin panel, you'll see a `django-twilio` Caller section that allows you to manage callers.

To blacklist a caller, do the following:

1. Click the `Add` button next to the `Caller` object in the admin panel. If you're running the server locally, the URL would be: http://localhost:8000/admin/django_twilio/caller/add/.
2. Enter in the caller's details that you wish to block. The phone number should be of the form: `+1NXXNXXXXXX` (E.164 format).
3. Check the `blacklisted` box.
4. Click the `Save` button.

Now any `django-twilio` built-in views or decorators will automatically reject calls from the specified caller!

Note: This does NOT effect code that does NOT use `django-twilio`. For example, if you write code that places outbound calls or SMS messages, since your code won't be interacting with `django-twilio`, the blacklist will NOT be honored.

Contributing

`django-twilio` is always under development, and welcomes any contributions! If you'd like to get your hands dirty with the source code, please fork the project on our GitHub page.

The guidelines below should help you get started.

Setup

1. Fork the project on Github
2. Create a separate, **well named** branch to work on, off of the **develop** branch.
3. Use the Makefile to set up your development environment:

```
$ make install-test
```

You should now have the `django-twilio` source code and development environment ready to go. Run the tests to ensure everything is okay:

```
$ make test
```

The tests should return with no failures.

Style

When contributing code, please try to keep the style matching that of the codebase. Right now, that means:

- 100% PEP-8 compliance.
- Proper spelling / punctuation in the source code.

After setting up your development environment, you can run:

```
$ make lint
```

This will lint the entire project and ensure PEP8 standards are being stuck to.

Please note: We're pretty relaxed on line length, but make sure you keep it below 90 characters where possible.

Docs

If you'd like to contribute any documentation, just dig right in! There are tons of things that can be improved, so don't feel shy! We use Sphinx to build our documentation, and we host our documentation online at ReadTheDocs.

Tests

In order to ensure high-quality releases, `django-twilio` aims to have an extensive test suite. All test suite patches and additions are welcome, and encouraged for new developers! The tests are well documented, and can be a great way to introduce yourself to the codebase!

To run the tests, you can either use:

```
$ make test
```

You'll see output that looks something like:

```
nosetests --with-coverage --cover-package=django_twilio --verbosity=1
Creating test database for alias 'default'...
....................................
Name                                              Stmts   Miss  Cover   Missing
-------------------------------------------------------------------------------
django_twilio                                         2      0   100%
django_twilio.client                                  4      0   100%
django_twilio.decorators                             50      4    92%   74-75, 103-104
django_twilio.migrations                              0      0   100%
django_twilio.models                                 20      0   100%
django_twilio.settings                                3      0   100%
django_twilio.south_migrations                        0      0   100%
django_twilio.south_migrations.0001_initial          14      2    86%   30-33
django_twilio.utils                                  30      2    93%   44, 49
django_twilio.views                                  38      0   100%
-------------------------------------------------------------------------------
TOTAL                                               161      8    95%
----------------------------------------------------------------------
Ran 38 tests in 0.184s

OK
Destroying test database for alias 'default'...
```

That's it! As you can see, when you run the test suite, `django-twilio` should output not only failing test results, but also the coverage reports.

When you submit patches or add functionality to `django-twilio`, be sure to run the test suite to ensure that no functionality is broken!

Workflow

When contributing to `django-twilio`, here's a typical developer workflow:

```
# Preparing the environment:

$ git clone https://github.com/<your_username>/django-twilio.git
$ cd django_twilio/
$ git remote add upstream https://github.com/rdegges/django-twilio.git
$ git checkout develop
$ git pull upstream develop
$ make install-test

# Hacking:

$ git checkout develop
```

```
$ vim ...
<<< hack time >>>

# Writing tests:

$ cd test_project/test_app/
$ vim ...
<<< hack time >>>

# Running tests:

$ cd django_twilio/
$ make test
<<< check test output >>>
```

Note: Please be sure that if you fork the project, you work on the `develop` branch. When submitting pull requests, please do so only if they're for the `develop` branch.

Bugs / Feature Requests / Comments

If you've got any concerns about `django-twilio`, make your voice heard by posting an issue on our GitHub issue tracker. All bugs / feature requests / comments are welcome.

Contributors

The `django-twilio` library is the result of effort given by the following awesome Twilio / Django / Python community members:

- Randall Degges - original author
- Paul Hallett - current maintainer
- Lee Trout
- Roger Boardman
- Kyle Conroy
- Justin Van Koten
- Issac Kelly
- Zopsi
- Suloev Dmitry
- Matt Carpenter
- Scott Clark
- Felix Schwarz
- Florian Le Goff
- Richard Kolkovich
- Sunah Suh

- Daniel Hawkins
- wbt
- shinriyo

Without their support, we wouldn't have such a well-written library. Thanks folks!

CHAPTER 3

Indices and Tables

The following is auto-generated documentation that can be used to quickly find relevant parts of this documentation.

- genindex
- modindex
- search