pyrowire Documentation

Release 0.1.0

Keith Hamilton

April 06, 2015

Contents

pyrowire is a framework you can use to quickly create Twilio-based SMS/MMS applications, licensed under the BSD 3-Clause license.

CHAPTER 1

Quickstart

For the purposes of this quickstart, it is assumed that you have an account with both Heroku and Twilio, and that you have at minimum the following installed:

- pip
- virtualenv

In your virtual environment's root directory, execute:

```
$ pip install pyrowire && pyrowire --init
```

This will install `pyrowire`, and copy into the root folder the following files:

- app.py (the application file)
- settings.py (the configuration file)
- Procfile (a Heroku Procfile)
- requirements.txt (pip requirements file)

1.1 Usage

```
$ ENV=(DEV|STAGING|PROD) [RUN=(WEB|WORKER)] [TOPIC=] python my_app.py
```

1.2 Sample Application

Here's what the my_app.py file (created by running `pyrowire --init`) looks like:

```
import pyrowire
import my_settings

# configure the pyrowire application
pyrowire.configure(my_settings)

# all app.processor methods need to be annotated with the topic
# for which they process and take one kwarg, 'message_data'
@pyrowire.handler(topic='my_topic')
```

```
def my_processor(message_data):
    if not message_data:
        raise TypeError("message_data must not be None")

    # insert handler logic here
    return message_data

# all pyro.filter methods need to be annotated with the name
# of the filter and take one kwarg, 'message_data'
@pyrowire.validator(name='my_validator')
def my_filter(message_data):
    if not message_data:
        raise TypeError("message_data must not be None")

    # insert validation logic here
    # validators should try to prove that the message is invalid, i.e., return True
    return True

if __name__ == '__main__':
    pyrowire.run()
```

As you can see, it's rather straightforward; to start out you are given placeholders for both a handler and a validator. The handler is where you will write the business logic for your Twilio application, and additional validators can be added if needed, or removed altogether. See their Handlers and Message Validation for more information.

1.2.1 pyrowire Tutorial

If you feel like getting the walkthrough now, head over to the tutorial section.

CHAPTER 2

Slowstart

2.1 Installation

pyrowire relies on a few underlying tools to run. We recommend using *pip/virtualenv*, but there are a few other ways you can do this, depending on your inclinations.

2.1.1 Dependencies

pip

We lean heavily towards *pip* for installing *virtualenv*, since ultimately it is our preferred distribution method for pyrowire. Fortunately, installing *pip* is super easy.

OS X

If you are a Homebrew user, and have installed python with

```
$ brew install python
```

you should already have *pip* on your machine. Otherwise, you can use easy_install:

```
$ sudo easy_install *pip*
```

Linux

To install *pip* on Linux, you can use the default package manager for your Linux flavor:

```
$ sudo (yum|apt-get) install python-*pip*
```

virtualenv

virtualenv is a great tool for development, because it isolates python versions from the system python version. This is great for you because it means you can develop your python applications independent of each other, regardless of whether they require different versions of python and your application's various dependencies.

OS X/Linux

Getting *virtualenv* installed is pretty straightforward, using either easy_install:

```
$ sudo easy_install *virtualenv*
```

or, with *pip* (our fav):

```
$ sudo *pip* install *virtualenv*
```

Redis

---

pyrowire currently has a hard dependency on Redis, so you will need to install that on your dev machine:

OS X

Do it with Homebrew (if you don't use Homebrew, you really should check it out):

```
$ brew install redis
```

Linux

Ubuntu/Debian

```
$ sudo apt-get install redis-server
```

RHEL

```
$ sudo yum install redis
```

2.1.2 Optional dependencies

Whereas you do not require the dependencies in this section, they may come in handy for testing/development.

ngrok

---

ngrok is a great tool for testing that forwards a public-facing URL to your local machine. This is great for testing your pyrowire app, since you can set your ngrok URL as your Twilio number's messaging endpoint and test your app without actually deploying to Heroku or another environment.

OS X

If you are Homebrew user, simply run:

```
$ brew install ngrok
```

If you don't use homebrew, you can download the binary here and run it as an executable.

Linux

Ubuntu/Debian

```
$ sudo apt-get install ngrok-server
```

RHEL

```
$ sudo yum install ngrok-server
```

2.1.3 Installing pyrowire

Via *pip*

Once you have the *pip* and *virtualenv* dependencies met, you are clear to install pyrowire. Our preferred method is via *pip*:

```
$ mkdir my_pyrowire_project
$ *virtualenv* my_pyrowire_project
$ cd my_pyrowire_project && source bin/activate
$ *pip* install pyrowire
```

Installing from Source

If you really want to download and install pyrowire, you are welcome to do that as well. Visit the release page, and grab the latest version, then just run `python setup.py install` to install it locally.

2.2 Application Landscape

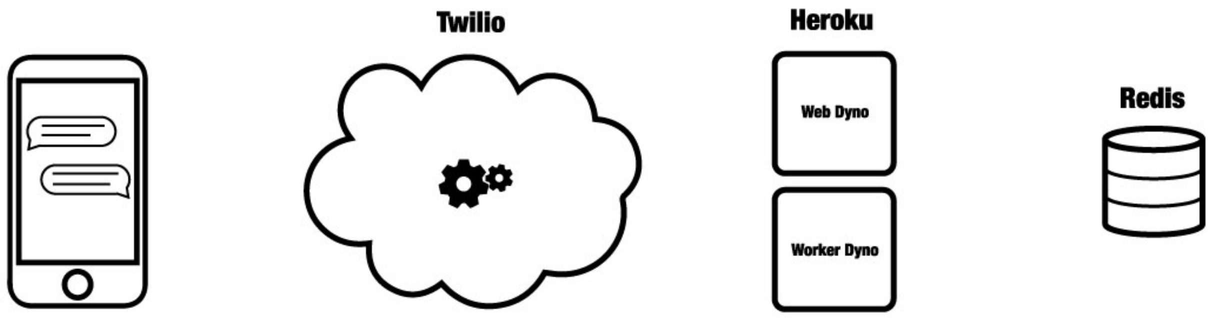
This section describes the general application landscape and event chain for pyrowire. You will note that in the flow presented below, Heroku is used as the deployment platform. This is the platform for which pyrowire was initially designed, but other platforms like AWS and Google Compute Engine could be used in its place.

Where you see *Heroku Web Dyno*, and *Heroku Worker Dyno*, you may mentally substitute them with, for example, *EC2 Web Application Server*, and *EC2 Worker Instance*, respectively.

2.2.1 Major Components

Four major components are part of the pyrowire application landscape:

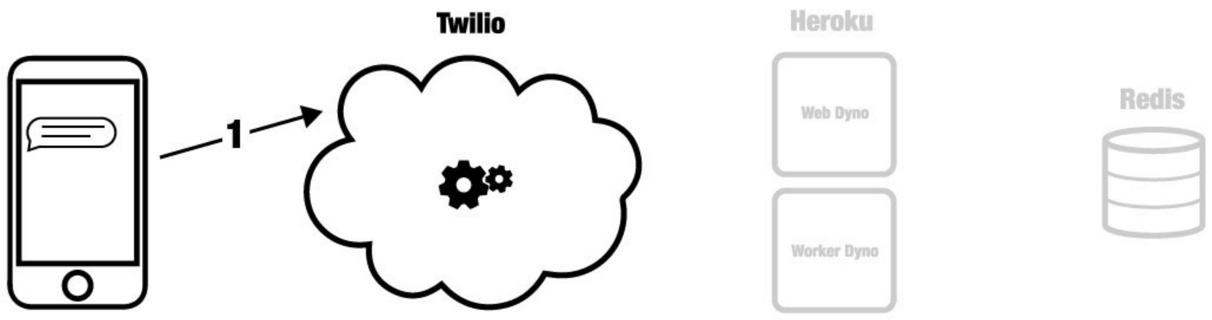
- a mobile phone
- Twilio service
- Heroku web/worker dynos
- Redis instance (local, RedisCloud, RedisToGo)



2.2.2 Distinct Events

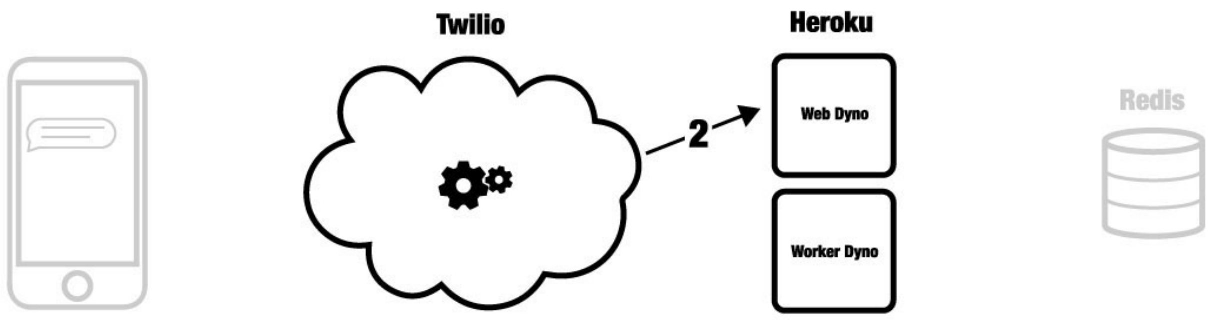
The following seven distinct events describe most of what pyrowire is doing. For further clarification, consult the source code.

1. An SMS is sent to a Twilio number or shortcode



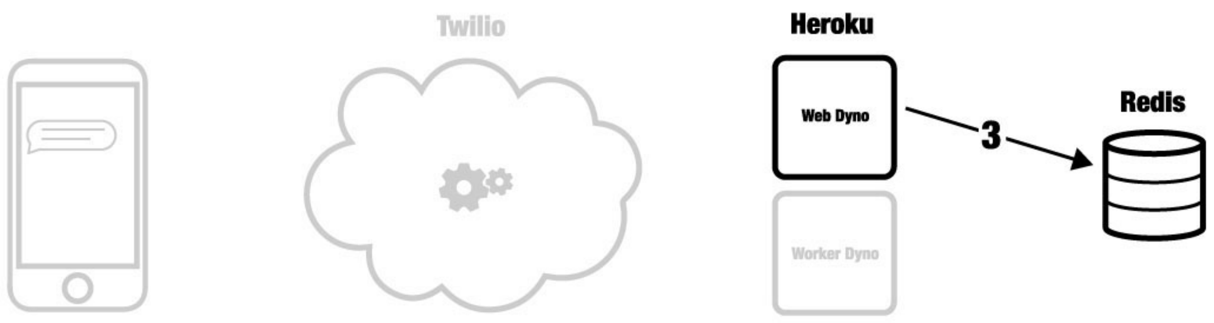
During this phase, Twilio's service receives an SMS or MMS message at some phone number or shortcode that you have defined as pointing at your application endpoint. Twilio constructs a TwiML payload and forwards it to your application in Heroku.

2. Your application's web dyno receives the TwiML message



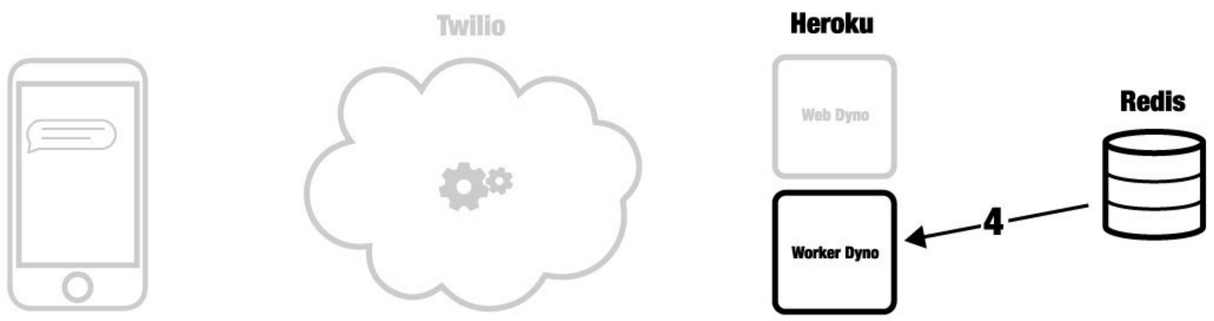
When your application receives a TwiML message, the message is restructured into a python dict containing all of the necessary properties to be handled by pyrowire. The message is then run through the validators defined for your topic in your settings.py file. If the message passes all validators, it is queued in Redis for a worker to pick up.

3. Queueing the message for your worker(s)



Once validated, messages are queued in Redis for retrieval by your workers. Workers will only ever dequeue messages that are assigned to their respective topic.

4. Handling messages

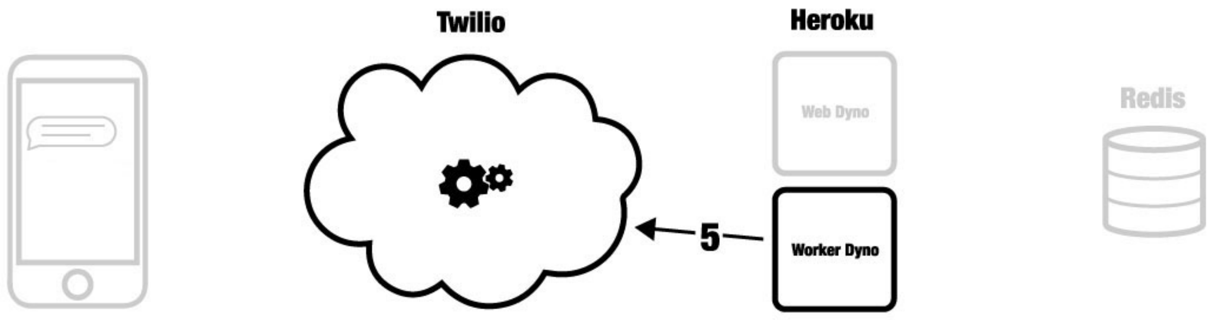


This event is where the meat of the application resides, where your worker dyno(s) will block the thread and pop items off of their topic's queue, and do whatever you have defined should happen for the messages' respective topic.

For instance, you may take the nouns out of the message and return the first image retrieved by a Google image search on those words.

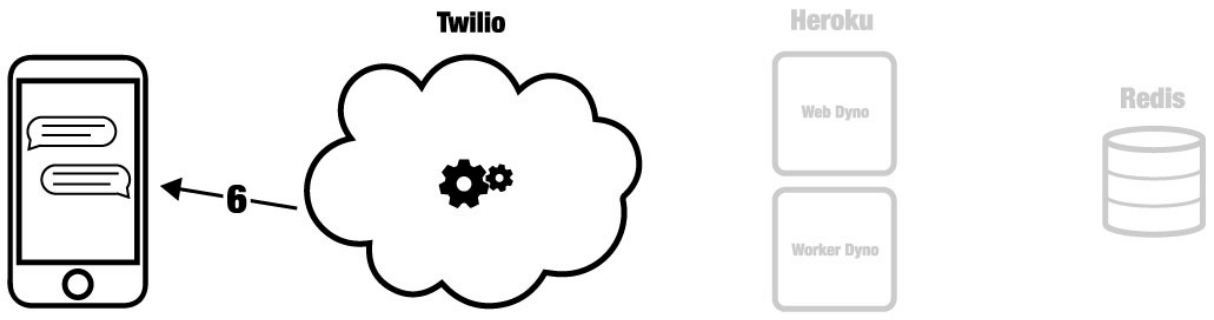
Once your message handler has finished its work, it will typically send the message back.

5. Sending back TwiML



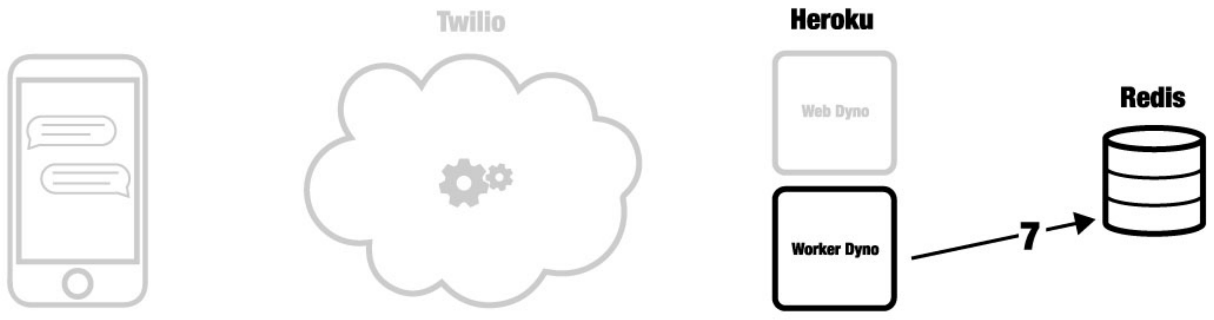
pyrowire has built-in handlers for sending SMS and MMS messages, and it does so by constructing a TwiML message object via the Twilio REST API.

6. Message received



Once Twilio gets your outbound response message, it will forward it back to the original sender, using the mobile number attached to the original message.

7. Recording the message



Once the message has been validated, handled, and sent back, your worker dyno will record the completed event in Redis, as a backup record of what took place.

2.3 Receiving Messages

pyrowire's primary object of note is a message. When an SMS (or MMS) is received by your application, pyrowire will construct a dictionary object from it, using most of the properties provided by the TwiML request body.

See Anatomy of a Message for more information message properties.

2.3.1 Message Endpoints

pyrowire uses a topic-based approach to handling incoming messages. One pyrowire instance can scale to handle many, many different Twilio SMS applications, and separates logic for each by the use of topics. Each topic is considered to be a separate Twilio application, has its own definition in your config file, and has the endpoint:

```
http(s)://my-rad-pyrowire-instance.mydomain.com/queue/<topic>
```

where `<topic>` is a keyword of your choice that identifies messages as being for a specific application.

Because pyrowire handles incoming messages, and can assign workers, on a per-topic basis, you could run as many different applications off of one cluster as you want, provided you scale up for it. Every time a message is received via Twilio's REST interface, it will be forwarded to your pyrowire instance, queued by its topic, then routed to, and processed by, a handler specifically designed for that topic/application. Business logic across applications can vary as much as you need it to, as each topic is handled independently by its defined handler.

2.3.2 Message Validation

pyrowire has three default message validators. By default, all messages received will be passed through the following:

- **profanity**: checks the incoming message against a list of about 1,000 graphically profane terms (trust us).
- **length**: checks that the length of the incoming message does not exceed some threshold; Twilio, by default, uses 160 characters as a limit, so we do too. Also ensures incoming messages have a length > 0.
- **parseable**: Twilio can't parse everything. Take emoji for example. The default parseable validator allows inclusion of all alphanumeric characters and most punctuation characters. For a comprehensive list of valid characters, see Valid Message Characters.

2.3.3 Custom Validators

As described in the previous section, pyrowire uses the concept of topics to distinguish handling for each message, but you can also create custom validators that can be used on messages for one or more topics.

Defining a Validator

Defining a custom validator is easy. Just annotate a method that takes a message, checks to make sure it conforms to some rule the method identifies, and returns a boolean, and you're set.

```
@pyrowire.validator(name='my_min_length_validator')
def min_length(message_data):
    if not message_data:
        raise TypeError("message_data must not be None.")
    # return True if message is less than 5 chars long
    return len(message_data['message']) < 5
```

Validator Criteria

All validators must satisfy the following criteria:

1. the `pyrowire.validator` annotation must take one kwarg, *name*, and should be used to identify the validator.
2. the method definition must take one arg, *message_data*
3. validators must be designed to return True if the message is not valid, *i.e., they are trying to prove that the message received is invalid.*

Sample Validator

Let's check it out by creating, say, a validator that requires the word 'yo' be present in all messages:

```
# all app.validator methods need to be annotated with the name of the validator
# and take one kwarg, 'message_data'
@pyrowire.validator(name='must_include_yo')
def must_include_yo(message_data):
    if not message_data:
        raise TypeError("message_data must not be None.")

    import re.search
    # assert that 'yo' is not found in the message
    return not re.search(r'*yo*', message_data['message'].lower())
```

By using the `@pyrowire.validator` annotation, any twilio topics you define in your configuration file that require the validator 'must_include_yo' will have to pass this validator in addition to the three defaults. By convention, the name of the method should match the name passed into the `@pyrowire.validator` decorator, but it doesn't have to.

Overriding Default Validators

If you want to omit a validator from your application, you can just remove it from your configuration file for the topic in question (see this example).

If you want to change the validator's behavior, just define it again in your app file:

```
# profanity validator that considers 'reaver' to be the only bad word in the verse
@pyrowire.validator(name='profanity')
def profanity(message_data):
    if not message_data:
        raise TypeError("message_data must not be None.")

    import re.search
    return re.search(r'\breaver\b', message_data['message'].lower())
```

2.4 Working with Messages

It's all well and good just being able to receive messages, but once a message has been received, how do you work with it?

To help work with messages, pyrowire relies on *handlers*; methods assigned to a specific message topic.

2.4.1 Handlers

With pyrowire, the only logic you need to think about (other than optional message validators), is what happens to the message after it's been successfully received. Enter the handler method, a function that defines the business logic for your application, and is annotated:

`@pyrowire.handler(topic='whatever_topic_it_is_for')`

where 'whatever_topic_its_for' corresponds to a defined topic block in your settings file.

Defining a Handler

Defining a handler is super easy. Just annotate a method that takes a message, performs your topic-specific logic, and does whatever else it needs to do.

```
@pyrowire.handler(topic='my_topic')
def my_handler(message_data):
    if not message_data:
        raise TypeError("message_data must not be None")
    # handler logic
    return message_data
```

Handler Criteria

All handlers must satisfy the following criteria:

1. the pyrowire.handler annotation must take one kwarg, *topic*, and must be set to the topic for which the handler is to work (this topic must be in your settings file).
2. the method definition must take one arg, *message_data*
3. the method should return the message_data object

Sample Handler

Let's take a look at a very simple handler that just receives an incoming message, randomizes the order, then returns it:

```
# all app.handler methods need to be annotated with the topic for which they process
# and take one kwarg, 'message_data'
@pyrowire.handler(topic='sms_randomizer')
def my_handler(message_data):
    if not message_data:
        raise TypeError("message_data must not be None")

    import random
    # randomize the message and save it as 'return_message'
    message = message_data['message'].split()
    random.shuffle(message)
    message_data['reply'] = ' '.join(message)

    # send the message data back along with the key of the message body
    # to send to initiate a Twilio SMS reply
    pyrowire.sms(message_data)

    return message_data
```

As you can see, all we need to do to process and return a message is tell a method annotated with `@pyrowire.handler``(topic='my_topic_name')` what to do with the message data that is received from the pyrowire app worker, then send it using `pyrowire.sms` method. To use this method, we pass the message_data object back to the `sms` method. By default the key used to source the message to send is 'reply', but this can be changed by adding an optional kwarg, 'key'.

See below, Changing the Reply Key for more information.

2.4.2 Sending a Message

pyrowire supports sending both SMS and MMS, both using a very simple syntax. Currently, to send MMS in the US and UK, you need a shortcode, but in Canada you can use either a shortcode or a full phone number.

Replying with SMS

All message objects that you work with will by default have a `reply` property, which you can populate with the reply message you wish to send back to the original sender. You can also use your own key for a message, if you pass it to the `pyrowire.sms` method as a kwarg. Let's take a look:

```
@pyrowire.handler(topic='sample')
def sample_handler(message_data):
    if not message_data:
        raise TypeError("message_data must not be None")

    reply = ''
    for index, item in enumerate([x for x in message_data['message'].split()]):
        if index % 2 == 0:
           reply += ' foo%s' % item
        else:
            reply += ' bar%s' % item
    message_data['reply'] = reply.strip()

    # here's where you send back
    pyrowire.sms(message_data)

    return message_data
```

So that's it. Just add `pyrowire.sms(message_data)` before the return, and an SMS will be returned back to the original sender with the 'reply' key as the message body.

Replying with MMS

pyrowire is rigged up to support MMS as well.

To send an MMS message, you just substitute `pyrowire.sms` method with `pyrowire.mms`.

```
@pyrowire.handler(topic='sample')
def random_cat_image(message_data):
    if not message_data:
        raise TypeError("message_data must not be None")

    import random
    import mycats
    url = random.choice(mycats.images)

    # here's where you send back
    pyrowire.mms(message_data, media_url=url)

    return message_data
```

If you want to include text with the media message, you can do so by setting a reply, and using the `include_text` kwarg:

```
@pyrowire.handler(topic='sample')
def random_cat_image(message_data):
    if not message_data:
        raise TypeError("message_data must not be None")

    import random
    import mycats
    url = random.choice(mycats.images)

    # adding an additional reply message
    message_data['reply'] = "Meeeeeeeeeowww!"

    # here's where you send back
    pyrowire.mms(message_data, include_text=True, media_url=url)

    return message_data
```

Changing the Reply Key

If you would like, you can change the reply key from 'reply' to a key of your choice. All you need to do to use it with either `pyrowire.sms` or `pyrowire.mms` is add a kwarg:

```
# for sms
pyrowire.sms(message_data, key='my_custom_key')

# for mms
pyrowire.mms(message_data,
             key='my_custom_key',
             include_text=True,
             media_url='http://bit.ly/IC394d')
```

2.5 Settings

Once you have your validators and handlers set up, you'll need to dial in your settings file.

pyrowire uses a python file for settings configuration. To check out the sample settings file, look here.

pyrowire's settings files are broken down into two sections:

- **Topics** (Twilio application-specific settings). The Topics block can have as many topic dictionaries as are needed.
- **Profiles** (environment profile-specific settings). There is one block per run environment *(DEV/STAGING/PROD)*

2.5.1 Defining a Topic

To start out, here's what the topic section of a pyrowire settings file looks like:

```
TOPICS = {
    'my_topic': {
        # send_on_accept determines whether to send an additional accept/success
        # message upon successfully receiving an SMS.
        # NOTE: this will result in two return messages per inbound message
        'send_on_accept': False,
        # global accept (success) and error messages for your app
        'accept_response': 'Great, we\'ll get right back to you.',
        'error_response': 'It seems like an error has occurred...please try again.',
        # key/value pairs for application-specific validators and their responses
        # if a message fails to pass validation.
        # Define your custom validators here, or change the message
        # for an existing validator.
        'validators': {
            'profanity': 'You kiss your mother with that mouth? No profanity, please.',
            'length': 'Your message exceeded the maximum allowable character limit' + \
                        '(or was empty). Please try again .',
            'parseable': 'Please only use alphanumeric and punctuation characters.'
        },
        # properties are any non-pyrowire-specific properties that you will need to
        # run your handler, such as an API key to some external service.
        'properties': {},
        # Twilio account credentials section, where the account credentials for your
        # application-specific account are stored
        'twilio': {
            'account_sid': '',
            'auth_token': '',
            'from_number': '+1234567890'
        },
        # the default max length for a single message segment, per twilio, is 160 chars
        # but you can set this anything under 1600.
        'max_message_length': 160
    }
}
```

Let's break that down a bit.

```
TOPICS = {
    'my_topic': {
```

This is the beginning of the applications dictionary, and, we have defined one topic, `my_topic`.

Response Settings

```
# send_on_accept determines whether to send an additional accept/success message upon
# successfully receiving an SMS.
# NOTE: this will result in two return messages per inbound message
'send_on_accept': False,
# global accept (success) and error messages for your app
'accept_response': 'Great, we\'ll get right back to you.',
'error_response': 'It seems like an error has occurred...please try again later.',
```

- **send_on_accept** enables or disables your app from actually sending a reply message immediately after the incoming SMS was successfully accepted. Setting this to `False` will prevent your app from sending two return messages for every one it receives.
- **accept_response** and **error_response** are respectively the messages that will be returned in the event of a success or error. *Note:* error_response will always send if an error occurs.

Validator Settings

profanity, **length**, and **parseable** are the default validators for your app. To omit any one of these, comment out or remove the item from the application's validators definition. Changing the message will change the return message sent to the user if his/her message fails to pass the validator.

```
# key/value pairs for application-specific validators and their responses if a
# message fails to pass validation.
# Define your custom validators here. If you wish to change the response message
# of a default validator, you can do that here.
'validators': {
    'profanity': 'You kiss your mother with that mouth? No profanity, please.',
    'length': 'Your message exceeded the maximum allowable character limit' + \
                        '(or was empty). Please try again .',
    'parseable': 'Please only use alphanumeric and punctuation characters.'
},
```

It is in the `validators` block that you would add any custom validators and their respective fail messages if you add validators to your application. Remember, excluding a validator from an app config will cause it to not be used on any incoming messages for that application; this means you can selectively apply different validators to different applications.

Properties Settings

Properties are used for very specific application purposes. Say you want to translate all incoming messages into Yoda-speak, and you need to hit an API for that...this is where you can add in your API key. The properties property in the app config is just a catch-all spot for your application-specific custom properties.

```
# properties are any non-pyrowire-specific properties that you will need to
# run your handler, such as an API key to some external service.
'properties': {},
```

In your handler method, then, you could access this as follows:

Twilio Settings

This is where you enter your Twilio account information: SID, auth token, and from number. You can get these from your Twilio account, at Twilio's website. If you don't have an account, setting it up is easy, and you can even use it in a free trial mode to get started.

```
'twilio': {
    # enter your twilio account SID, auth token, and from number here
    'account_sid': ""
    'auth_token': ""
    'from_number': "+1234567890"
}
```

Maximum Message Length Setting

Technically, you can receive messages as long as 1600 characters, but Twilio will break up any message longer than 160 characters to segments of 160. Since 160 characters is the default max for one message segment, it is the default setting for pyrowire apps.

```
# the default max length for a single message segment, per twilio, is 160 chars
# but you can set this anything under 1600.
'max_message_length': 160
```

2.5.2 Environment Settings

pyrowire uses profiles to determine environment-specific details such as debug, Redis host, and web host. The default settings.py file includes profiles for three standard environments: `dev`, `staging`, and `prod`. Let's take a look at one of those, `dev`:

```
PROFILES = {
    'dev': {
        'debug': True,
        'log_level': logging.DEBUG,
        'redis': {
            'host': 'localhost',
            'port': 6379,
            'db': 0,
            'password': ''
        },
        'host': 'localhost',
        'port': 62023
    }
```

The profiles block is defined by the key `PROFILES`. So original. One level down is the keyword `dev` indicating the beginning of the dev profile settings.

Debug and Logging Settings

The first setting in the block is `debug`, which is stored as a boolean. Python's `logging` module is used to indicate the logging level for each profile.

```
PROFILES = {
    'dev': {
      'debug': True,
      'log_level': logging.DEBUG,
```

Redis Settings

```
'redis': {
    'host': 'localhost',
    'port': 6379,
    'database': 0,
    'password': ''
}
```

First, you have the standard Redis connection properties, `host`, `port`, `database`, and `password`. This should be pretty straightforward...just add your connection details in this section.

By default, all profiles connect to localhost over the standard Redis port using the default database with no password. If a password is provided, it will be used, but ignored otherwise.

Host and Port Settings

```
# set to '0.0.0.0' for hosted deployment so pyrowire listens on all interfaces
'host': 'localhost',
# set to 0 for hosted deployment so pyrowire can pick up the environment var $PORT
'port': 62023
```

2.5.3 Hosted Deployment Settings

Of note is that for a hosted deployment, you will want to set the port to `0`, which tells pyrowire to set the port to the value of the web container's $PORT env var. Additionally, it is a good idea to set the host for any hosted deployments to `0.0.0.0` so that pyrowire will listen on all bindings to that web container.

2.6 Running

Once you have all your handlers, validators, and configuration vars in place, it's time to get busy.

2.6.1 Environment Variables

pyrowire requires one environment var to be present when running locally:

- **ENV**: the run profile (DEV|STAGING|PROD) under which you want to run pyrowire

For running on Heroku, there are two additional environment vars required:

- **RUN**: (WEB|WORKER), the type of Heroku dyno you are running.
- **TOPIC**: only required for workers, this is the topic the specific worker should be working for.

See below for more details.

2.6.2 Standalone vs Web vs Worker

pyrowire is designed to be able to run in one of three modes:

- **standalone**: In standalone mode, at least two threaded processes are started, one for the web application, and one worker process for each topic included in your settings file.
- **web**: In web mode, only the web application is started. This is most commonly used in Heroku deployment, and can be achieved by including `RUN=WEB` in your environment variables.
- **worker**: In worker mode, only a worker process is started. This also is most commonly used in Heroku deployment, and can be achieved by including both `RUN=WORKER` and `TOPIC=[some-topic]` in your environment variables.

2.6.3 Running Locally

Typically, when running locally, pyrowire will run in standalone mode. Once you have your handler, optional additional validator(s), and configuration all set up, running pyrowire is easy:

```
ENV=DEV python app.py
```

This will spin up one worker for your topic (or one per topic if you have multiple topics configured), and a web server running on localhost:62023 to handle incoming messages. After that, you can start sending it GET/POST requests using your tool of choice. You won't be able to use Twilio for inbound messages yet, (unless your local DNS name is published to the world) but you should receive them back from requests made locally.

2.6.4 A Note on Deployment

When we built pyrowire, we designed it to be deployed to Heroku; however, pyrowire could certainly be deployed to any hosted service, such as AWS or Google Compute Engine.

For example, on AWS, pyrowire could easily be run as a set of three EC2 instances (minimally):

- EC2 instance for web application
- EC2 instance for a topic worker
- EC2 instance for Redis

Again, we built it for Heroku, but with a little thought, it can be run anywhere.

2.6.5 Heroku Deployment

When you are ready to move to staging or production, it's time to get the app installed on Heroku. Remember, the host setting should be `0.0.0.0` and the port setting for your profile should be `0` when deploying to Heroku. You can get through 90% of the work by running:

```
pyrowire --deploy-heroku
```

from the root of your project directory.

This will walk you through logging into your Heroku account, if you haven't already, setting up an app, if you haven't already, and adding Redis as an add-on, if you haven't already. It will take you all the way to the point where you will just need to add any changes to git, commit, and push to Heroku.

2.6.6 Deploying to Heroku Manually

If you would like to set your Heroku stuff up manually, that's totally up to you. We won't get deep into how to manually deploy to Heroku here, since it isn't really in the scope of this document, but the basics are:

1. Set up a Heroku application with at least one web dyno and at least one worker.
2. Set up a Redis database as a Heroku add-on, such as RedisToGo or RedisCloud, through a service, such as RedisLabs, or on an external server.
3. Add the Redis host, port, database, and password information to your config file for Staging and/or Production profiles.
4. Add the heroku remote git endpoint to your project (`git remote add heroku.com:my-heroku-app.git`).
5. Push the project up to heroku and let it spin up.
6. Add the remote endpoint to your Twilio account number (e.g., for SMS: `http://my-heroku-app.herokuapp.com/queue/my_topic`).
7. Profit.

2.6.7 Heroku Procfile

When you ran `pyrowire --init` a sample Procfile was placed in the root of your application folder. Taking a look at it, you can see:

```
web: ENV=STAGING RUN=web python ./app.py --server run_gunicorn 0.0.0.0:$PORT --workers=1
worker: ENV=STAGING RUN=worker TOPIC=my_topic python ./app.py
```

You will need to include a `RUN` environment var set to either `web` or `worker` with respect to the purpose of the command item.

For workers, a `TOPIC` environment var is required to indicate which topic the worker(s) should work for. You can see in the `web` line, the default setting in the Procfile is one worker. Scale as needed.

If you would like to run pyrowire for more than one topic, you will need to add additional worker definitions accordingly. For example:

```
worker: ENV=STAGING RUN=worker TOPIC=my_topic python ./app.py
worker: ENV=STAGING RUN=worker TOPIC=my_other_topic python ./app.py
```

2.7 Tutorial

2.7.1 Yo Momma

For our sample application, we are going to make an SMS application that sends a random 'yo momma' joke to the person who sends the initial text, because there is no excuse to not have a 'yo momma' joke handy.

The application parameters:

- Text 'yo momma' to some number
- Receive a 'yo momma' joke back
- No profanity validation required
- Inbound message must say 'yo momma' at some point

2.7.2 Installation

Firstly, let's install pyrowire in a *virtualenv*.

```
$ cd ~/src
$ mkdir yo_mamma
$ *virtualenv* yo_mamma
$ cd yo_mamma && source bin/activate
$ *pip* install pyrowire
```

2.7.3 Generating Stub Files

Now that we have pyrowire installed, let's create the stub files.

```
$ pyrowire --init
```

We now have our stub files all set in the root of our project folder. Let's take a look at the raw sample app file:

```python
import pyrowire
import settings

# configure the pyrowire application
pyrowire.configure(settings)

# all app.processor methods need to be annotated with the topic for which they process
# and take one kwarg, 'message_data'
@pyrowire.handler(topic='my_topic')
def my_processor(message_data):
    if not message_data:
        raise TypeError("message_data must not be None")
    # insert handler logic here
    return message_data

# all pyro.filter methods need to be annotated with the name of the filter
# and take one kwarg, 'message_data'
@pyrowire.validator(name='my_validator')
def my_filter(message_data):
    if not message_data:
        raise TypeError("message_data must not be None")
    # insert validation logic here
    # validators should try to prove that the message is invalid, i.e., return True
    return True

if __name__ == '__main__':
    pyrowire.run()
```

2.7.4 Defining the Handler

We want to zero in on the handler, since that is where we are going to get our joke to send back to the texter. Let's look at how we could implement this:

```python
@pyrowire.handler(topic='yo_momma')
def yo_momma_handler(message_data):
    if not message_data:
        raise TypeError("message_data must not be None")

    import urllib2
    import json

    resp = urllib2.urlopen('http://api.yomomma.info').read()
    content = resp.split('<body>')[1].split('</body>')[0].strip()

    message_data['reply'] = json.loads(content)['joke']
    pyrowire.sms(message_data=message_data)

    return message_data
```

And that's it. The handler does a very simple thing; it fetches a response from api.yomomma.info, parses out the returned joke, attaches it to the original message_data object, then returns it as an SMS to the sender.

At this point, if you wanted to run the application without any additional validation, you would be good to go.

2.7.5 Adding a Validator

Since we initially said the sender should text ‘yo momma’ to our application number, we should make sure that the message received says ‘yo momma’ and nothing else. Let’s add a custom validator.

```
@pyrowire.validator(name='yo_momma')
def yo_momma_validator(message_data):
    if not message_data:
        raise TypeError("message_data must not be None")

    return not 'yo momma' == message_data['message'].lower().strip()
```

Hmm...this will work, but, maybe it’s too harsh. Let’s back it up so that our custom validator just checks to ensure that the phrase ‘yo momma’ is in the text body.

```
@pyrowire.validator(name='yo_momma')
def yo_momma_validator(message_data):
    if not message_data:
        raise TypeError("message_data must not be None")

    import re
    return not re.search(r'\byo momma\b', message_data['message'].lower().strip())
```

Yeah, that’s nice. Let’s go with that.

2.7.6 Overriding a Default Validator

If you want, you can override a default validator simply by redefining it in your application file. For example, pyrowire comes with a profanity validator to ensure that incoming text messages aren’t profane. This runs by default, but for the yo momma app, we are going to disable it.

```
# custom handler and validator here

@pyrowire.validator(name='profanity')
def override_profanity(message_data=None):
    pass
```

This is somewhat of a trivial example, since if we want to disable a validator, we only need to remove it from the topic validator dictionary in our settings file. Since we haven’t covered settings yet, however, this example stands to illustrate that you can override default validators.

2.7.7 Settings

Sweet, we are almost ready to run this sucker and start dropping momma jokes on people. We will need to dial in our settings file though, so it knows what to do when a message with the topic ‘yo_momma’ is received. To do that, we open the `settings.py` file that we stubbed out earlier.

```
import logging

TOPICS = {
    'my_topic': {
        'send_on_accept': False,
        'accept_response': 'Great, we\'ll get right back to you.',
        'error_response': 'It seems like an error has occurred...please try again later.',
        'validators': {
            'profanity': 'You kiss your mother with that mouth? No profanity, please.',
            'length': 'Your message exceeded the maximum allowable character limit
                       (or was empty). Please try again .',
            'parseable': 'Please only use alphanumeric and punctuation characters.'
        },
        'properties': {},
        'twilio': {
            'account_sid': '',
            'auth_token': '',
            'from_number': '+1234567890'
        },
        'max_message_length': 160
    }
}

PROFILES = {
    'dev': {
        'debug': True,
        'log_level': logging.DEBUG,
        'redis': {
            'host': 'localhost',
            'port': 6379,
            'db': 0,
            'password': ''
        },
        'host': 'localhost',
        'port': 62023
    },
    'staging': {
        'debug': True,
        'log_level': logging.WARN,
        'redis': {
            'host': 'localhost',
            'port': 6379,
            'db': 0,
            'password': ''
        },
        'host': '0.0.0.0',
        'port': 0
    },
    'prod': {
        'debug': False,
        'log_level': logging.ERROR,
        'redis': {
            'host': 'localhost',
            'port': 6379,
            'db': 0,
            'password': ''
        },
        'host': '0.0.0.0',
        'port': 0
    }
}
```

Ooh, looks like it still has all the default settings, which is good. Let's update the `TOPICS` section so it works for our application:

```
import logging
```

```
TOPICS = {
    'yo_momma': {
        'send_on_accept': False,
        'accept_response': 'Yo momma is so fat...',
        'error_response': 'It seems like an error has occurred...please try again later.',
        'validators': {
            # removed the profanity validator, since we don't want to use it.
            'length': 'Your message exceeded the maximum allowable character limit
                      (or was empty). Please try again .',
            'parseable': 'Please only use alphanumeric and punctuation characters.'
        },
        'properties': {},
        'twilio': {
            'account_sid': '<MY_TWILIO_ACCOUNT_SID>',
            'auth_token': '<MY_TWILIO_AUTH_TOKEN>',
            # update with your real number
            'from_number': '+1234567890'
        },
        'max_message_length': 160
    }
}
```

What changed?

- the name of the topic dict object
- the `accept_response` definition
- we removed `profanity` from the list of validators for 'yo_momma'
- we added Twilio information (this step is rather crucial)

Cool, now that we have our topic defined, we can move on to getting our profile/host settings dialed in, which incidentally, should be already done for our dev environment.

At this point you can also go back to your app file and remove the override on the profanity validator. Because we just removed it from the 'yo_momma' topic dictionary's 'validators' sub-dictionary, it won't apply to your incoming messages.

2.7.8 Host Settings

The other part of our settings file are the Profile/Host settings. Since we are still working locally, let's just worry about the 'dev' settings for right now (we'll get to the staging/production settings in a bit):

```
# TOPICS defs up here

PROFILES = {
    'dev': {
        'debug': True,
        'log_level': logging.DEBUG,
        'redis': {
            'host': 'localhost',
            'port': 6379,
            'db': 0,
            'password': ''
        },
        'host': 'localhost',
        'port': 62023
    },
    # staging, prod settings below
}
```

This should all look pretty straightforward. We are developing locally using the port 62023, with a local, password-less, Redis instance, and have debugging flags set to log at a debug level.

2.7.9 Running Locally

Checklist:

- handler set up. Check.
- validator set up. Check.
- profanity filter disabled. Check.
- topic and profile settings in. Check.

Awesome, it's time to run this sucker. Yo momma is about to get rained on.

To run this app, navigate to the root of your project directory, and make sure your virtual environment is active. Next, run:

```
$ ENV=DEV python app.py
```

Note that you need to include the ENV environment var so pyrowire knows which profile to choose. Running the above command will spin up a web application on port 62023, and will spin up one worker per topic defined in your settings file (in the case of this tutorial, it should spin up one worker).

In this example, we've omitted the `RUN=(web|worker)` environment variable, which causes both the web and worker processes to run at the same time. When we move to Heroku, or some other platform like AWS, we will add the RUN variable so we can separate the work across nodes. We will cover running as web or worker in the Heroku section below.

2.7.10 Sending a Test Message

For our test, we are going to do the following:

- run ngrok to get a public-facing URL for our local environment
- add the *ngrok* URL to Twilio
- send a test message

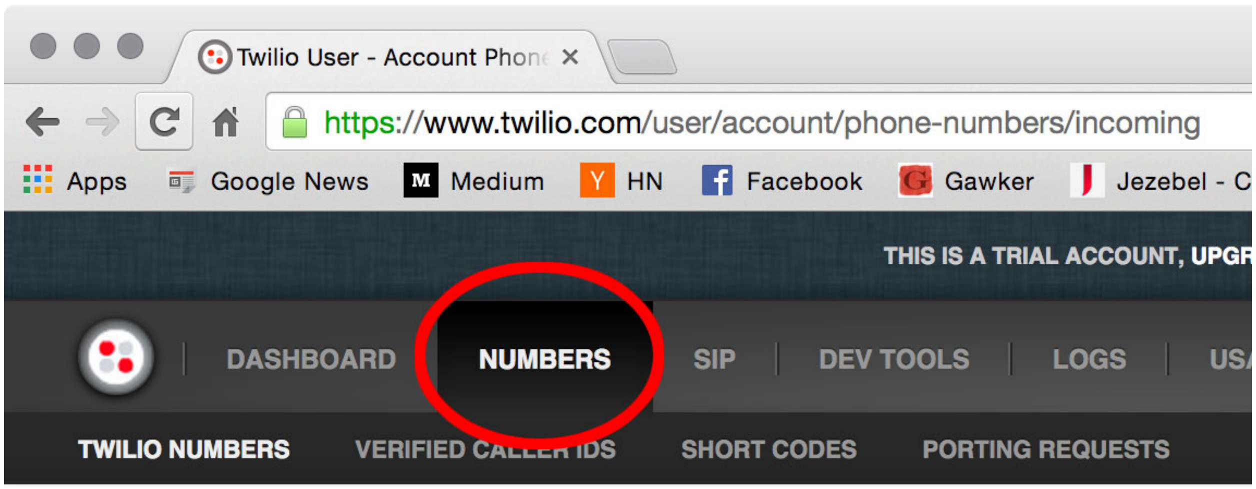
First we need to run *ngrok* to get our public-facing URL. The default port that pyrowire is set up to run on is `62023`. Open up a terminal prompt and run:

```
$ ngrok 62023
```

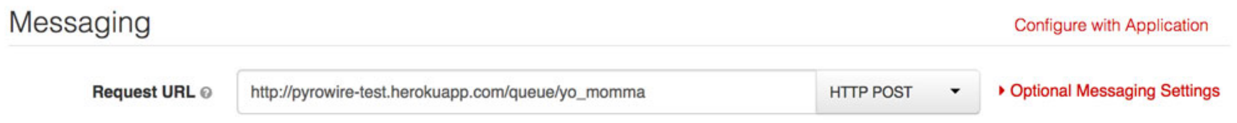
Grab the forwarding http URL (the part before the ->), and copy it. Next, open up your Twilio account page. If you haven't set up a Twilio account yet, there's no time like the present. Head on over to the Twilio website to get started with that.

Next we need to make sure your Twilio number is pointing at our ngrok endpoint:

1. Navigate to your Twilio account page then click the 'Numbers' link in the nav bar.



2. Click on the phone number you want to set up.
3. Add the *ngrok* endpoint, with the topic queue in the URL (e.g., *http://25ab3b9b.ngrok.com/queue/yo_momma*


Messaging

Configure with Application

Request URL: http://pyrowire-test.herokuapp.com/queue/yo_momma — HTTP POST

Optional Messaging Settings

Lastly, it's time to send a test message. Grab your phone, and shoot a message to the number you used for your app endpoint in Twilio, and watch the magic happen.

2.7.11 Deploying to Heroku

pyrowire makes deploying to heroku super easy with a fabric walkthrough. To get started deploying this application, just run:

```
$ pyrowire --deploy-heroku
```

This will walk you through logging into your Heroku account, if you haven't already, setting up an app, if you haven't already, and adding Redis as an addon, if you haven't already. It will take you all the way to the point where you will just need to add any changes to git, commit, and push to Heroku.

You will need to go update your Twilio phone number's endpoint with the Heroku endpoint once your deployment is running.

2.7.12 Configuring Redis

If you didn't initially set up a Redis addon in the above `--deploy-heroku` step, you can always go back later and do that by running:

```
$ pyrowire --add-herkou-redis
```

2.7.13 Bombs Away

So now you have your Twilio endpoint set up, your application is running in Heroku, and you are ready to drop some bombs on people's moms.

2.8 APPENDICES

2.8.1 Appendix A: Definition of Terms

Handler

A handler is one of the fundamental building blocks of pyrowire. It is responsible for the business logic performed for an application, and determines how pyrowire will respond to an inbound message via Twilio's REST API. Applications and handlers have a unique one-to-one relationship.

Handlers can be added by annotating a method with `@pyrowire.handler(topic='some_topic_name')`, where 'some_topic_name' corresponds to an application to be handled by pyrowire.

Validator

A validator is another fundamental building block of pyrowire. Validators are responsible for validating incoming messages, and unlike handlers, are optional. Validators have a many-to-one relationship with applications.

Validators can be added to any application by creating a method annotated with `@pyrowire.validator(name='some_validator_name')` and adding that validator as a key/value member of the application's `validators` set in your settings file.

Each validator added to an application should have a corresponding message, e.g,:

```
'must_say_yo':  'You got to say "yo", yo!'
```

2.8.2 Appendix B: The Anatomy of a pyrowire Message

Messages in pyrowire that are available to you in handlers have the following format (sample data presented):

```
# properties marked with an asterisk are those that Twilio will try to collect,
# and will be included in the message_data dictionary if available.
message_data = {
    'message': 'Some message',
    'number': '+1234567890',
    'sid': 'ugJCgMZwjxzqGjmrmWhXlyAPbnoTECjEHA',
    'topic': 'some_topic',
    'from_country': 'USA',        *
    'from_state': 'OR',           *
    'from_city': 'Portland',      *
    'from_zip': '97209',          *
    'media': {
        'count': 1,
        'media': {
            'http://bit.ly/Icd34Ox': 'image/jpeg'
        }
    }
}
```

Of note here is the media sub-dictionary. If an MMS with attached media was sent, this will be populated with key/value pairs of the media URL as well as the media content type. If no media was attached (SMS) this key will be an empty dict.

2.8.3 Appendix C: Valid Message Characters

By default, pyrowire only permits the following characters in any incoming message:

- alphanumeric: a-z, A-Z, 0-9
- punctuation: ! ” # $ % & ‘ ( ) * , - . / : ? @ ^

2.8.4 Appendix D: Under the Hood

pyrowire is built on top of the following:

- Flask - handles web server process and request routing
- Twilio TwiML & REST APIs - handles communication to and from Twilio
- Redis - used for queuing, and storing received, pending, and completed message transactions

2.8.5 Appendix E: Pull Requests

We love the open source community, and we embrace it. If you have a pull request to submit to pyrowire, do it! Just please make sure to observe the following guidelines in any additions/updates you wish to merge into the master branch:

- use idiomatic python - we may ask you to resubmit if code does not follow PEP or is “un-pythonic” in nature.
- docstrings required in all methods (*except stuff like getters/setters, stuff that is built-in, or has tests already*)
- unittests required for any added/modified code

Other than that, we welcome your input on this project!

2.8.6 Appendix F: Road Map

pyrowire is certainly in its infancy. Thus far, we have a fairly rigid architecture design, and support only for SMS/MMS. Future endeavors include:

- providing voice call/queue support
- building connectors for different message queues (currently only Redis supported)
- add support for creating additional routes, for web views
- history management (visibility into Redis database from web view)

If you are into this, and want to help, fork it!