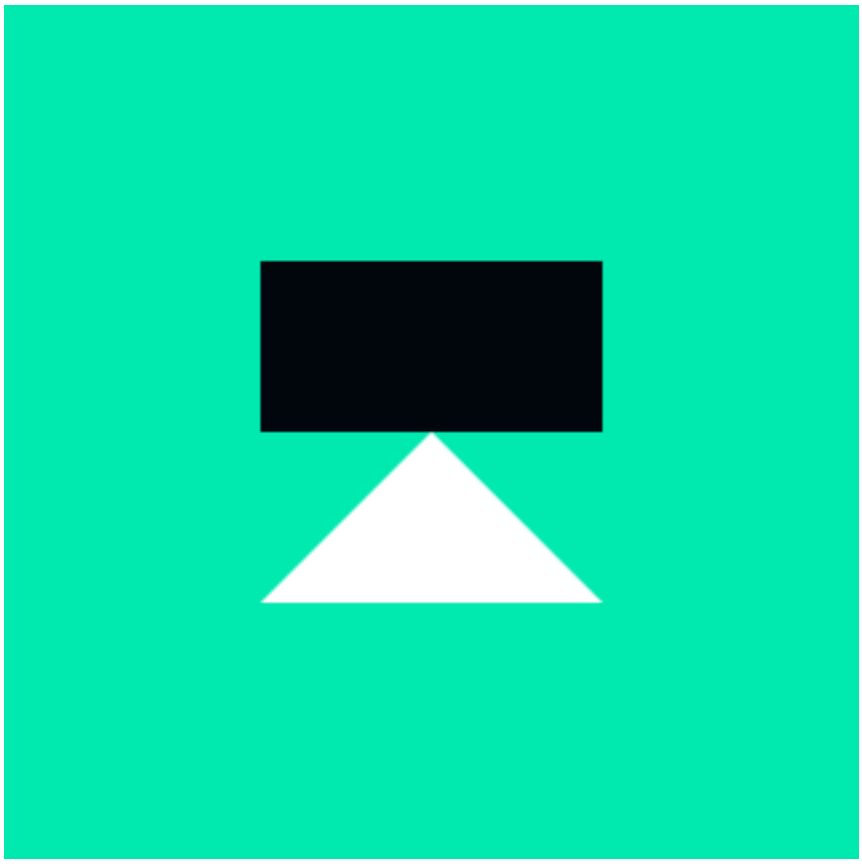

Thorn Documentation

Release 1.5.0

Robinhood Markets

Oct 22, 2016

Contents

Python Webhook and Event Framework

CHAPTER 1

Copyright

Thorn User Manual

by Robinhood Markets, Inc. and individual contributors.

Note: While the *Thorn* documentation is offered under the Creative Commons *Attribution-ShareAlike 4.0 International* license the Thorn *software* is offered under the BSD License (3 Clause)

Python Webhook and Event Framework

CHAPTER 2

Introduction

Version 1.5.0

Web http://thorn.readthedocs.io/

Download http://pypi.python.org/pypi/thorn/

Source http://github.com/robinhood/thorn/

Keywords event driven, webhooks, callback, http, django

Table of Contents:

2.1 About

Thorn is a webhook framework for Python, focusing on flexibility and ease of use, both when getting started and when maintaining a production system.

The goal is for webhooks to thrive on the web, by providing Python projects with an easy solution to implement them and keeping a repository of patterns evolved by the Python community.

- **Simple**

  Add webhook capabilities to your database models using a single decorator, including filtering for specific changes to the model.

- **Flexible**

  All Thorn components are pluggable, reusable and extendable.

- **Scalable**

  Thorn can perform millions of HTTP requests every second by taking advantage of Celery for asynchronous processing.

2.2 What are webhooks?

A webhook is a fancy name for an HTTP callback.

Users and other services can subscribe to events happening in your system by registering a URL to be called whenever the event occurs.

The canonical example would be GitHub where you can register URLs to be called whenever a new change is committed to your repository, a new bugtracker issue is created, someone publishes a comment, and so on.

Another example is communication between internal systems, traditionally dominated by complicated message consumer daemons, using webhooks is an elegant and REST friendly way to implement event driven systems, requiring only a web-server (and optimally a separate service to dispatch the HTTP callback requests).

Webhooks are also composable, so you can combine multiple HTTP callbacks to form complicated workflows, executed as events happen across multiple systems.

2.2.1 In use

Notable examples of webhooks in use are:

Site	Documentation
Github	https://developer.github.com/webhooks/
Stripe	https://stripe.com/docs/webhooks
PayPal	http://bit.ly/1TbDtvj

2.2.2 Example

This example adds four webhook events to the Article model of an imaginary blog engine:

```
from thorn import ModelEvent, webhook_model

@webhook_model   # <--- activate webhooks for this model
class Article(models.Model):
```

```
    uuid = models.UUIDField()
    title = models.CharField(max_length=100)
    body = models.TextField()

    class webhooks:
        on_create = ModelEvent('article.created')
        on_change = ModelEvent('article.changed'),
        on_delete = ModelEvent('article.removed'),
        on_publish = ModelEvent(
            'article.published',
            state__eq='PUBLISHED',
        ).dispatches_on_change(),

    @models.permalink
    def get_absolute_url(self):
        return 'article:detail', None, {'uuid': self.uuid}
```

Users can now subscribe to the four events individually, or all of them by subscribing to `article.*`, and will be notified every time an article is created, changed, removed or published:

```
$ curl -X POST                                                          \
> -H "Authorization: Bearer <secret login token>"                       \
> -H "Content-Type: application/json"                                   \
> -d '{"event": "article.*", "url": "https://e.com/h/article?u=1"}'     \
> http://example.com/hooks/
```

The API is expressive, so may require you to learn more about the arguments to understand it fully. Luckily it's all described in the *Events Guide* for you to consult after reading the quick start tutorial.

2.3 What do I need?

Version Requirements

Thorn version 1.0 runs on

- Python (2.7, 3.4, 3.5)
- PyPy (5.1.1)
- Jython (2.7).
- **Django (1.8, 1.9, 1.10)** Django 1.9 adds the `transaction.on_commit()` feature, and Thorn takes advantage of this to send events only when the transaction is committed.

Thorn currently only supports Django, and an API for subscribing to events is only provided for Django REST Framework.

Extending Thorn is simple so you can also contribute support for your favorite frameworks.

For dispatching web requests we recommend using Celery, but you can get started immediately by dispatching requests locally.

Using Celery for dispatching requests will require a message transport like RabbitMQ or Redis.

You can also write custom dispatchers if you have an idea for efficient payload delivery, or just want to reuse a technology you already deploy in production.

2.4 Quick Start

Go immediately to the *Django Integration* guide to get started using Thorn in your Django projects.

If you are using a different web framework, please consider contributing to the project by implementing a new environment type.

2.5 Alternatives

Thorn was inspired by multiple Python projects:

- dj-webhooks
- django-rest-hooks
- durian

2.6 Installation

2.6.1 Installing the stable version

You can install thorn either via the Python Package Index (PyPI) or from source.

To install using *pip*,:

```
$ pip install -U thorn
```

2.6.2 Downloading and installing from source

Download the latest version of thorn from http://pypi.python.org/pypi/thorn/

You can install it by doing the following,:

```
$ tar xvfz thorn-0.0.0.tar.gz
$ cd thorn-0.0.0
$ python setup.py build
# python setup.py install
```

The last command must be executed as a privileged user if you are not currently using a virtualenv.

2.6.3 Using the development version

With pip

You can install the latest snapshot of thorn using the following pip command:

```
$ pip install https://github.com/robinhood/thorn/zipball/master#egg=thorn
```

2.7 Getting Help

2.7.1 Mailing list

For discussions about the usage, development, and future of Thorn, please join the thorn-users mailing list.

2.7.2 IRC

Come chat with us on IRC. The **#thorn** channel is located at the Freenode network.

2.8 Bug tracker

If you have any suggestions, bug reports or annoyances please report them to our issue tracker at https://github.com/robinhood/thorn/issues/

2.9 Contributing

Development of *Thorn* happens at GitHub: https://github.com/robinhood/thorn

You are highly encouraged to participate in the development of *thorn*. If you don't like GitHub (for some reason) you're welcome to send regular patches.

Be sure to also read the Contributing to Thorn section in the documentation.

2.10 License

This software is licensed under the *New BSD License*. See the `LICENSE` file in the top distribution directory for the full license text.

CHAPTER 3

Getting Started

Release 1.5

Date Oct 22, 2016

3.1 Django Integration

Table of Contents:

3.1.1 Installation

To use Thorn with your Django project you must

1. Install Thorn

```
$ pip install thorn
```

2. Add `thorn.django` to `INSTALLED_APPS`:

```
INSTALLED_APPS = (
    # ...,
    'thorn.django',
)
```

3. Migrate your database to add the subscriber model:

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

4. Webhook-ify your models by adding the `webhook_model` decorator.

   Read all about it in the *Events Guide*.

5. (Optional) Install views for managing subscriptions:

   Only *Django REST Framework* is supported yet, please help us by contributing more view types.

6. Specify the recommended HMAC signing method in your `settings.py`:

```
THORN_HMAC_SIGNER = 'thorn.utils.hmac:sign'
```

3.1.2 Django Rest Framework Views

The library comes with a standard set of views you can add to your Django Rest Framework API, that enables your users to subscribe and unsubscribe from events.

The views all map to the *`Subscriber`* model.

To enable them add the following URL configuration to your `urls.py`:

```
url(r"^hooks/",
    include("thorn.django.rest_framework.urls", namespace="webhook"))
```

Supported operations

Note: All of these curl examples omit the important detail that you need to be logged in as a user of your API.

Subscribing to events

Adding a new subscription is as simple as posting to the `/hooks/` endpoint, including the mandatory event and url arguments:

```
$ curl -X POST                                                    \
> -H "Content-Type: application/json"                             \
> -d '{"event": "article.*", "url": "https://e.com/h/article?u=1"}' \
> http://example.com/hooks/
```

Returns the response:

```
{"id": "c91fe938-55fb-4190-a5ed-bd92f5ea8339",
 "url": "http:\/\/e.com\/h/article?u=1",
 "created_at": "2016-01-13T23:12:52.205785Z",
 "updated_at": "2016-01-13T23:12:52.205813Z",
 "user": 1,
 "hmac_secret": "C=JTX)v3~dQCl];[_h[{q{CScm]oglLoe&>ga:>R~jR$.x?t|kW!FH:s@|4bu:11",
 "hmac_digest": "sha256",
 "content_type": "application\/json",
 "subscription": "http://localhost/hooks/c91fe938-55fb-4190-a5ed-bd92f5ea8339",
 "event": "article.*"}
```

Parameters

- `event` (mandatory)

  The type of event you want to receive notifications about. Events are composed of dot-separated words, so this argument can also be specified as a pattern matching words in the event name (e.g. `article.*`, `*.created`, or `article.created`).

- `url` (mandatory)

  The URL destination where the event will be sent to, using a HTTP POST request.

- `content_type` (optional)

  The content type argument specifies the MIME-type of the format required by your endpoint. The default is `application/json`, but you can also specify `application/x-www-form-urlencoded`..

- `hmac_digest` (optional)

  Specify custom HMAC digest type, which must be one of: sha1, sha256, sha512.

  Default is sha256.

- `hmac_secret` (optional)

  Specify custom HMAC secret key.

  This key can be used to verify the sender of webhook events received.

  A random key of 64 characters in length will be generated by default, and can be found in the response.

The only important part of the response data at this stage is the `id`, which is the unique identifier for this subscription, and the `subscription` url which you can use to unsubscribe later.

Listing subscriptions

Perform a *GET* request on the `/hooks/` endpoint to retrieve a list of all the subscriptions owned by user:

```
$ curl -X GET                                          \
> -H "Content-Type: application/json"                  \
> http://example.com/hooks/
```

Returns the response:

```
[
    {"id": "c91fe938-55fb-4190-a5ed-bd92f5ea8339",
     "url": "http:\/\/e.com\/h/article?u=1",
     "created_at": "2016-01-15T23:12:52.205785Z",
     "updated_at": "2016-01-15T23:12:52.205813Z",
     "user": 1,
     "content_type": "application\/json",
     "event": "article.*"}
]
```

Unsubscribing from events

Perform a *DELETE* request on the `/hooks/<UUID>` endpoint to unsubscribe from a subscription by unique identifier:

```
$ curl -X DELETE                                       \
> -H "Content-Type: application/json"                  \
> http://example.com/hooks/c91fe938-55fb-4190-a5ed-bd92f5ea8339/
```

3.1.3 Example consumer endpoint

This is an example Django webhook receiver view, using the json content type:

```
from __future__ import absolute_import, unicode_literals

import hmac
import base64
import json
import hashlib

from django.http import HttpResponse
from django.views.decorators.http import require_POST
from django.views.decorators.csrf import csrf_exempt

ARTICLE_SECRET = 'C=JTX)v3~dQCl];[_h[{q{CScm]oglLoe&>ga:>R~jR$.x?t|kW!FH:s@|4bu:11'
ARTICLE_DIGEST_TYPE = 'sha256'

# also available at `thorn.utils.hmac.verify`
def verify(hmac_header, digest_method, secret, message):
    digestmod = getattr(hashlib, digest_method)
    signed = base64.b64encode(
        hmac.new(secret, message, digestmod).digest(),
    ).strip()
    return hmac.compare_digest(signed, hmac_header)

@require_POST()
@csrf_exempt()
def handle_article_changed(request):
    digest = request.META.get('HTTP_HOOK_HMAC')
    body = request.body
    if verify(digest, ARTICLE_DIGEST_TYPE, ARTICLE_SECRET, body):
        payload = json.loads(body)
        print('Article changed: {0[ref]}'.format(payload)
        return HttpResponse(status=200)
```

Using the `csrf_exempt()` is important here, as by default Django will refuse POST requests that do not specify the CSRF protection token.

3.1.4 Verify HMAC Ruby

This example is derived from Shopify's great examples found here: https://help.shopify.com/api/tutorials/webhooks#verify-webhook

```
require 'rubygems'
require 'base64'
require 'openssl'
require 'sinatra'

ARTICLE_SECRET = 'C=JTX)v3~dQCl];[_h[{q{CScm]oglLoe&>ga:>R~jR$.x?t|kW!FH:s@|4bu:11'

helpers do

    def verify_webhook(secret, data, hmac_header):
        digest = OpenSSL::Digest::Digest.new('sha256')
        calculated_hmac = Base64.encode64(OpenSSL:HMAC.digest(
            digest, secret, data)).strip
```

```
        return calculated_hmac == hmac_header
    end
end

post '/' do
    request.body.rewind
    data = request.body.read
    if verify_webhook(ARTICLE_SECRET, env["HTTP_HOOK_HMAC"])
        # deserialize data' using json and process webhook
    end
end
```

3.1.5 Verify HMAC PHP

This example is derived from Shopify's great examples found here: https://help.shopify.com/api/tutorials/webhooks#verify-webhook

```
<?php

define('ARTICLE_SECRET',␣
↪'C=JTX)v3~dQCl];[_h[{q{CScm]oglLoe&>ga:>R~jR$.x?t|kW!FH:s@|4bu:11')

function verify_webhook($data, $hmac_header)
{
    $calculated_hmac = base64_encode(hash_hmac('sha256', $data,
        ARTICLE_SECRET, true));
    return ($hmac_header == $calculated_hmac);
}

$hmac_header = $_SERVER['HTTP_HOOK_HMAC'];
$data = file_get_contents('php://input');
$verified = verify_webhook($data, $hmac_header);

?>
```

CHAPTER 4

User Guide

Release 1.5

Date Oct 22, 2016

4.1 Events

Table of Contents:

4.1.1 Introduction

An event can be anything happening in your system that something external wants to be notified about.

The most basic type of event is the *Event* class, from which other more complicated types of events can be built. The basic event does not have any protocol specification, so any payload is accepted.

An *Event* is decoupled from subscribers and dispatchers and simply describes an event that can be subscribed to and dispatched to subscribers.

In this guide we will first be describing the basic event building blocks, so that you understand how they work, then move on to the API you are most likely to be using: the `ModelEvent` class and `@webhook_model` decorator used to associate events with database models.

Defining events

Say you would like to define an event that dispatches whenever a new user is created, you can do so by creating a new *Event* object, giving it a name and assigning it to a variable:

```
from thorn import Event

on_user_created = Event('user.created')
```

Currently this event is merely defined, and won't be dispatched under any circumstance unless you manually do so by calling `on_user_created.send()`.

Since the event name is `user.created` it's easy to imagine this being sent from something like a web view responsible for creating users, or whenever a user model is changed.

Naming events

Subscribers can filter events by simple pattern matching, so event names should normally be composed out of a category name and an event name, separated by a single dot:

```
"category.name"
```

A subscription to `"user.*"` will match events `"user.created"`, `"user.changed"`, and `"user.removed"`; while a subscription to `"*.created"` will match `"user.created"`, `"article.created"`, and so on.

`ModelEvent` names may include model instance's field values. For example, you could define `"user.{.username}"`, and events will be fired as `user.alice`, `user.bob` and so on.

Sending events

```
from userapp import events
from userapp.models import User


def create_user(request):
    username = request.POST['username']
    password = request.POST['password']
    user = User.objects.create(username=username, password=password)
    events.on_user_created.send({
        'uuid': user.uuid,
```

```
        'username': user.username,
        'url': 'http://mysite/users/{0}/'.format(user.uuid),
    })
```

Timeouts and retries

Dispatching an event will ultimately mean performing one or more HTTP requests if there are subscribers attached to that event.

Many HTTP requests will be quick, but some of them will be problematic, especially if you let arbitrary users register external URL callbacks.

A web server taking too long to respond can be handled by setting a socket timeout such that an error is raised. This timeout error can be combined with retries to retry at a later time when the web server is hopefully under less strain.

Slow HTTP requests is usually fine when using the Celery dispatcher, merely blocking that process/thread from doing other work, but when dispatching directly from a web server process it can be deadly, especially if the timeout settings are not tuned properly.

The default timeout for web requests related to an event is configured by the *THORN_EVENT_TIMEOUT* setting, and is set to 3 seconds by default.

Individual events can override the default timeout by providing either a `timeout` argument when creating the event:

```
>>> on_user_created = Event('user.created', timeout=10.0)
```

or as an argument to the *send()* method:

```
>>> on_user_created.send(timeout=1.5)
```

In addition to the web server being slow to respond, there are other intermittent problems that can occur, such as a 500 (Internal Server Error) response, or even a 404 (Resource Not Found).

The right way to deal with these errors is to retry performing the HTTP request at a later time and this is configured by the event retry policy settings:

```
>>> on_user_created = Event(
...     'user.created',
...     retry=True,
...     retry_max=10,
...     retry_delay=60.0,
... )
```

The values used here also happen to be the default setting, and can be configured for all events using the *THORN_RETRY*, *THORN_RETRY_MAX* and *THORN_RETRY_DELAY* settings.

Serialization

Events are always serialized using the json serialization format *[0], which means the data you provide in the webhook payload must be representable in *json* or an error will be raised.

The built-in data types supported by *json* are:

- `int`

[0] Thorn can easily be extended to support additional serialization formats. If this is something you would like to work on then please create an issue on the Github issue tracker or otherwise get in touch with the project.

- `float`
- `string`
- `dictionary`
- `list`

In addition Thorn adds the capability to serialize the following Python types:

- `datetime.datetime`: converted to ISO-8601 string.
- `datetime.date`: converted to ISO-8601 string.
- `datetime.time`: converted to ISO-8601 string.
- **`decimal.Decimal`:** converted to string as the *json* float type is unreliable.
- `uuid.UUID`: converted to string.
- **`django.utils.functional.Promise`:** if django is installed, converted to string.

4.1.2 Model events

In most cases your events will actually be related to a database model being created, changed, or deleted, which is why Thorn comes with a convenience event type just for this purpose, and even a decorator to easily add webhook-capabilities to your database models.

This is the *`thorn.ModelEvent`* event type, and the *`@webhook_model()`* decorator.

We will be giving an example in a moment, but first we will discuss the message format for model events.

Message format

The model events have a standard message format specification, which is really more of a header with arbitrary data attached.

An example model event message serialized by json would look like this:

```
{"event": "(str)event_name",
 "ref": "(URL)model_location",
 "sender": "(User pk)optional_sender",
 "data": {"event specific data": "value"}}
```

The most important part here is `ref`, which is optional but lets you link back to the resource affected by the event.

We will discuss reversing models to provide the `ref` later in this chapter.

Decorating models

The easiest way to add webhook-capabilities to your models is by using the `@webhook_model()` decorator.

Here's an example decorating a Django ORM model:

```
from django.db import models

from thorn import ModelEvent, webhook_model


@webhook_model
```

```
class Article(models.Model):
    uuid = models.UUIDField()
    title = models.CharField(max_length=128)
    state = models.CharField(max_length=128, default='PENDING')
    body = models.TextField()

    class webhooks:
        on_create = ModelEvent('article.created')
        on_change = ModelEvent('article.changed')
        on_delete = ModelEvent('article.removed')
        on_publish = ModelEvent(
            'article.published', state__now_eq='PUBLISHED',
        ).dispatches_on_change()

        def payload(self, article):
            return {
                'title': article.title,
            }

    @models.permalink
    def get_absolute_url(self):
        return ('blog:article-detail', None, {'uuid': self.uuid})
```

Why is this example using Django?

Rest assured that Thorn is not a Django-specific library and should be flexible enough to integrate with any framework, but we have to use something for these generic examples, and Django is the only framework currently supported.

Please get in touch if you want to add support for additional frameworks, it's not as tricky as it sounds and we can help!

The webhooks we want to define is deferred to a private class inside the model.

The attributes of this class are probably a bit confusing at first, but how expressive this interface is will be apparent once you learn more about them.

So let's discuss the decorator arguments one by one:

1. `on_create = ModelEvent('article.created')`

   Here we specify an event to be sent every time a new object of this model type is created.

   The webhook model decorator can accept an arbitrary number of custom events, but there are three types of events the decorator already knows how to dispatch: `on_create`, `on_change` and `on_delete`. For any additional events you are required to specify the dispatch mechanism (see later explanation of the `on_publish` argument).

   The name `"article.created"` here is the event name that subscribers can use to subscribe to this event.

2. `on_change = ModelEvent('article.changed')`

   Just like `on_create` and `on_delete` the decorator does not need to know when an `on_change` event is to be dispatched: it will be sent whenever an object of this model type is changed.

3. `on_delete = ModelEvent('article.deleted')`

I'm sure you can guess what this one does already! This event will be sent whenever an object of this model type is deleted.

4. `on_publish = ModelEvent('article.published',state__now_eq='PUBLISHED')`

   Here we define a custom event type with an active filter.

   The filter (`state__now_eq='PUBLISHED'`) in combination with the specified dispatch type (`.dispatched_on_change`) means the event will only be sent when 1) an Article is changed and 2) the updated state changed from something else to `"PUBLISHED"`.

   The decorator will happily accept any argument starting with `on_` as an event associated with this model, and any argument to *`ModelEvent`* ending with `__eq`, `__ne`, `__gt`, `__gte`, `__lt`, `__lte`, `__is`, `__is_not`, `__contains`, `__startswith` or `__endswith` will be regarded as a filter argument.

   You can even use `Q` objects to create elaborate boolean structures, which is described in detail in the *Filtering* section.

5. `def webhook_payload`

   This method defines what to include in the `data` section of the webhooks sent for this model.

6. `@models.permalink`

   This tells Thorn how to get the canonical URL of an object of this model type, which is used as the `ref` field in the webhook *message payload*.

   In this case, when using Django, will translate directly into:

```
>>> from django.core.urlresolvers import reverse
>>> reverse('blog:article_detail', kwargs={'uuid': article.uuid})
http://example.com/blog/article/3d90c42c-d61e-4579-ab8f-733d955529ad/
```

4.1.3 `ModelEvent` objects

This section describes the *`ModelEvent`* objects used with the *`@webhook_model()`* decorator in greater detail.

Signal dispatch

Django signals and bulk updates

A limitation with database signals in Django is that signals are not dispatched for bulk operations (`objects.delete()`/ `objects.update()`), so you need to dispatch events manually when you use this functionality.

A model event will usually be dispatched in reaction to a signal [*]_, on Django this means connecting to the `post_save` and `post_delete` signals.

By signals we mean an implementation of the Observer Pattern, such as `django.dispatch.Signal`, `celery.utils.dispatch.Signal`, or blinker (used by Flask).

There are three built-in signal dispatch handlers:

1. Send when a new model object is created:

```
>>> ModelEvent('...').dispatches_on_create()
```

2. Send when an existing model object is changed:

```
>>> ModelEvent('...').dispatches_on_change()
```

3. Send when an existing model object is deleted:

```
>>> ModelEvent('...').dispatches_on_delete()
```

4. Send when a many-to-many relation is added

```
>>> ModelEvent('...').dispatches_on_m2m_add('tags')
```

   Argument is the related field name, and in this example tags is defined on the model as `tags = ManyToManyField(Tag)`. The event will dispatch whenever `Model.tags.add(related_object)` happens.

5. Send when a many-to-many relation is removed

```
>>> ModelEvent('...').dispatches_on_m2m_remove('tags')
```

   Argument is the related field name, and in this example tags is defined on the model as `tags = ManyToManyField(Tag)`. The event will dispatch whenever `Model.tags.remove(related_object)` happens.

6. Send when a many-to-many field is cleared

```
>>> ModelEvent('...').dispatches_on_m2m_clear('tags')
```

   Argument is the related field name, and in this example tags is defined on the model as `tags = ManyToManyField(Tag)`. The event will dispatch whenever `Model.tags.clear()` happens.

The webhook model decorator treats the `on_create`, `on_change`, and `on_delete` arguments specially so that you don't have to specify the dispatch mechanism for these, but that is not true for any custom events you specify by using the `on_` argument prefix to *`webhook_model`*.

Side effects in signals

Performing side-effects such as network operations inside a signal handler can make your code harder to reason about.

You can always send events manually, so you can opt-out of using signal-invalidation, but it's also a very convenient feature and it tends to work well.

Using signal-invalidation means that whenever a model instance is saved (using `model.save()`), or deleted, the signal handler will automatically also invalidate the cache for you by communicating with the cache server.

This has the potential of disrupting your database transaction in several ways, but we do include some options for you to control this:

- **`signal_honors_transaction=True`**

  default `False` (see note below)

  New in version 1.5.

  Example enabling this option:

```
ModelEvent(signal_honors_transaction=True, ...)
```

When this option is enabled, the actual communication with the cache server to invalidate your keys will be moved to a `transaction.on_commit` handler.

This means that if there are multiple webhooks being sent in the same database transaction they will be sent together in one go at the point when the transaction is committed.

It also means that if the database transaction is rolled back, all the webhooks assocatied with that transaction will be discarded.

This option requires Django 1.9+ and is disabled by default. It will be enabled by default in Thorn 2.0.

- **propagate_errors**

  default `True`

  New in version 1.5.

  Example disabling this option:

```
ModelEvent(propagate_errors=False, ...)
```

  By default errors raised while sending a webhook will be logged and ignored (make sure you have Python logging setup in your application).

  You can disable this option to have errors propagate up to the caller, but note that this means a `model.save()` call will roll back the database transaction if there's a problem sending the webhook.

Modifying event payloads

The `data` field part of the resulting *model event message* will be empty by default, but you can define a special method on your model class to populate this with data relevant for the event.

This callback must be named `webhook_payload`, takes no arguments, and can return anything as long as it's json-serializable:

```
class Article(models.Model):
    uuid = models.UUIDField()
    title = models.CharField(max_length=128)
    state = models.CharField(max_length=128, default='PENDING')
    body = models.TextField()

    def webhook_payload(self):
        return {
            'title': self.title,
            'state': self.state,
            'body': self.body[:1024],
        }
```

You should carefully consider what you include in the payload to make sure your messages are as small and lean as possible, so in this case we truncate the body of the article to save space.

Tip: Do we have to include the article body at all?

Remember that the webhook message will include the `ref` field containing a URL pointing back to the affected resource, so the recipient can request the full contents of the article if they want to.

Including the body will be a question of how many of your subscribers will require the full article text. If the majority of them will, including the body will save them from having to perform an extra HTTP request, but if not, you have drastically increased the size of your messages.

Modifying event headers

You can include additional headers for the resulting *model event message* by defining a special method on your model class.

This callback must be named `webhook_headers`, takes no arguments, and must return a dictionary:

```
from django.conf import settings
from django.db import models

class Article(models.Model):
    uuid = models.UUIDField()
    title = models.CharField(max_length=128)
    state = models.CharField(max_length=128, default='PENDING')
    body = models.TextField()
    user = models.ForeignKey(settings.AUTH_USER_MODEL)

    class webhooks:

        def headers(self, article):
            return {
                'Authorization':
                    'Bearer {}'.format(article.user.access_token),
            }
```

Event senders

If your model is associated with a user and you want subscribers to filter based on the owner/author/etc. of the model instance, you can include the `sender_field` argument:

```
from django.contrib.auth import get_user_model
from django.db import models


@webhook_model(
    sender_field='author.account.user',
)
class Article(models.Model):
    author = models.ForeignKey(Author)


class Author(models.Model):
    account = models.ForeignKey(Account)


class Account(models.Model):
    user = models.ForeignKey(get_user_model())
```

URL references

To be able to provide a URL reference back to your model object the event needs to know how to call `django.core.urlresolvers.reverse()` (or equivalent in your web framework) and what arguments to use.

A best practice when writing Django apps is to always add a `get_absolute_url` method to your models:

```
class Article(models.Model):

    @models.permalink
    def get_absolute_url(self):
        return ('article:detail', None, {'uuid': self.uuid})
```

If you define this method, then Thorn will happily use it, but some times you may also want to define alternate reversing strategies for specific events (such as `article.deleted`: when the article is deleted referring to the URL of the article does not make sense, but you could point to the category an article belongs to for example).

This is where the *model_reverser* helper comes in, which simply describes how to turn an instance of your model into the arguments used for *reverse*.

The signature of *model_reverser* is:

```
model_reverser(view_name, *reverse_args, **reverse_kwargs)
```

The positional arguments will be the names of attributes to take from the model instance, and the same for keyword arguments.

So if we imagine that the REST API view of our article app is included like this:

```
url(r'^article/', include(
    'apps.article.urls', namespace='article'))
```

and the URL routing table of the Article app looks like this:

```
urlpatterns = [
    url(r'^$',
        views.ArticleList.as_view(), name='list'),
    url(r'^(?P<uuid>[0-9a-fA-F-]+)/$',
        views.ArticleDetail.as_view(), name='detail'),
]
```

We can see that to get the URL of a specific article we need 1) the name of the view (`article:detail`), and 2) a named *uuid* keyword argument:

```
>>> from django.core.urlresolvers import reverse
>>> article = Article.objects.get(uuid='f3f2b22b-8630-412a-a320-5b2644ed723a')
>>> reverse('article:detail', kwargs={'uuid': article.uuid})
http://example.com/article/f3f2b22b-8630-412a-a320-5b2644ed723a/
```

So to define a reverser for this model we can use:

```
model_reverser('article:detail', uuid='uuid')
```

The `uuid='uuid'` here means take the `uuid` argument from the identically named field on the instance (`article.uuid`).

Any attribute name is accepted as a value, and even nested attributes are supported:

```
model_reverser('broker:position',
               account='user.profile.account')
#               ^^ will be taken from instance.user.profile.account
```

Filtering

Model events can filter models by matching attributes on the model instance.

The most simple filter would be to match a single field only:

```
ModelEvent('article.changed', state__eq='PUBLISHED')
```

and this will basically transform into the predicate:

```
if instance.state == 'PUBLISHED':
    send_event(instance)
```

This may not be what you want since it will always match even if the value was already set to `"PUBLISHED"` before. To only match on the transition from some other value to `"PUBLISHED"` you can use `now_eq` instead:

```
ModelEvent('article.changed', state__now_eq='PUBLISHED')
```

which will transform into the predicate:

```
if (old_value(instance, 'state') != 'PUBLISHED' and
        instance.state == 'PUBLISHED'):
    send_event(instance)
```

Transitions and performance

Using the `now_*` operators means Thorn will have to fetch the old object from the database before the new version is saved, so an extra database hit is required every time you save an instance of that model.

You can combine as many filters as you want:

```
ModelEvent('article.changed',
           state__eq='PUBLISHED',
           title__startswith('The'))
```

In this case the filters form an **AND** relationship and will only continue if all of the filters match:

```
if instance.state == 'PUBLISHED' and instance.title.startswith('The'):
    send_event(instance)
```

If you want an `OR` relationship or to combine boolean gates, you will have to use *Q* objects:

```
from thorn import ModelEvent, Q


ModelEvent(
    'article.changed',
    Q(state__eq='PUBLISHED') | Q(state__eq='PREVIEW'),
)
```

You can also negate filters using the ~ operator:

```
ModelEvent(
    'article.changed',
    (
        Q(state__eq='PUBLISHED') |
        Q(state__eq='PREVIEW') &
        ~Q(title__startswith('The'))
    )
)
```

Which as our final example will translate into the following pseudo-code:

```
if (not instance.title.startswith('The') and
        (instance.state == 'PUBLISHED' or instance.state == 'PREVIEW')):
    send_event(instance)
```

Tip: Thorn will happily accept Django's `Q` objects, so you don't have to import Q from Thorn when you already have one from Django.

Note that you are always required to specify `__eq` when specifying filters:

```
ModelEvent('article.created', state='PUBLISHED')      # <--- DOES NOT WORK

ModelEvent('article.created', state__eq='PUBLISHED')  # <-- OK! :o)
```

Supported operators

Operator	Description
`eq=B`	value equal to B (`__eq=True` tests for truth)
`now_eq=B`	value equal to B and was previously not equal to B
`ne=B`	value not equal to B (`__eq=False` tests for falsiness)
`now_ne=B`	value now not equal to B, but was previously equal to B
`gt=B`	value is greater than B: `A > B`
`now_gt=B`	value is greater than B, but was previously less than B
`gte=B`	value is greater than or equal to B: `A >= B`
`now_gte=B`	value greater or equal to B, previously less or equal
`lt=B`	value is less than B: `A < B`
`now_lt=B`	value is less than B, previously greater than B
`lte=B`	value is less than or equal to B: `A <= B`
`now_lte=B`	value less or equal to B, previously greater or equal.
`is=B`	test for object identity: `A is B`
`now_is=B`	value is now identical, but was not previously
`is_not=B`	negated object identity: `A is not B`
`now_is_not=B`	value is no longer identical, but was previously
`in={B,...}`	value is a member of set: `A in {B,...}`
`now_in={B,...}`	value is now member of set, but was not before
`not_in={B,...}`	value is not a member of set: `A not in {B,...}`
`now_not_in={B,...}`	value is not a member of set, but was before
`contains=B`	value contains element B: `B in A`
`now_contains=B`	value now contains element B, but did not previously
`startswith=B`	string starts with substring B
`now_startswith=B`	string now startswith B, but did not previously
`endswith=B`	string ends with substring B
`now_endswith=B`	string now ends with B, but did not previously

Tips

- Test for truth/falsiness

  There are two special cases for the `eq` operator: `__eq=True` and `_eq=False` is functionally equivalent to `if A` and `if not A` so any true-ish or false-ish value will be a match.

  Similarly with `ne` the cases `__ne=True` and `__ne=False` are special and translates to `if not A` and `if A` respectively.

- Use `A__is=None` for testing that `A is None`

- `contains` is not limited to strings!

  This operator supports any object supporting the `__contains__` protocol so in addition to strings it can also be used for sets, lists, tuples, dictionaries and other containers. E.g.: `B in {1,2,3,4}`.

- The transition operators (`__now_*`) may affect Django database performance.

  Django signals does provide a way to get the previous value of a database row when saving an object, so Thorn is required to manually re-fetch the object from the database shortly before the object is saved.

Sending model events manually

The webhook model decorator will add a new `webhooks` attribute to your model that can be used to access the individual model events:

```
>>> on_create = Article.webhooks.events['on_create']
```

With this you can send the event manually just like any other *Event*:

```
>>> on_create.send(instance=article, data=article.webhook_payload())
```

There's also `.send_from_instance` which just takes a model instance as argument and will send the event as if a signal was triggered:

```
>>> on_create.send_from_instance(instance)
```

The payload will then look like:

```
{
    "event": "article.created",
    "ref": "http://example.com/article/5b841406-60d6-4ca0-b45e-72a9847391fb/",
    "sender": null,
    "data": {"title": "The Mighty Bear"},
}
```

4.1.4 Security

The REST Hooks project has an excellent guide on security and webhooks here: http://resthooks.org/docs/security/

4.2 Subscribers

Table of Contents:

4.2.1 Introduction

Interested parties can subscribe to webhook events by registering a *Subscriber*.

Subscribers are stored in the database.

The subscription can match an event by simple pattern matching, and also filter by events related to a specific user (requires the event to be sent with a `sender` argument‘‘).

A simple subscriber can be created from the repl, like in this example where all `article.` related events will be sent to the URL: http://example.com/receive/article, and the payload is requested to be in *json* format:

```
>>> Subscriber.objects.create(
        event='article.*',
        url='http://example.com/receive/article',
        content_type='application/json',
... )
```

4.3 Dispatch

Table of Contents:

4.3.1 Introduction

As soon as you call `event.send` the webhook will be dispatched by performing HTTP requests to all the subscriber URL's matching the event.

The dispatch mechanism is configurable, and even supports pluggable backends.

There are three built-in dispatcher backends available:

- `"default"`

  Dispatch requests directly in the current process.

  In a web server the HTTP request will not complete until all of the Webhook requests have finished, so this is only suited for use in small installations and in development environments.

- `"disabled"`

  Does not dispatch requests at all, useful for development.

- `"celery"`

  Dispatch requests by sending a single Celery task for every event. The task will then be received by a worker which will start sending requests in batches to subscribers.

  Since performing HTTP requests are entirely I/O bound, routing these tasks to workers running the eventlet or gevent pools are recommended (see *Optimization and Performance*).

  The HTTP requests are also sorted by URL so that requests for the same domain have a high chance of being routed to the same process, to benefit from connection keep-alive settings, etc.

To configure the dispatcher used you need to change the *`THORN_DISPATCHER`* setting.

4.3.2 HTTP Client

Thorn uses the requests library to perform HTTP requests, and will reuse a single `Session` for every thread/process.

4.3.3 HTTP Headers

Thorn will provide the endpoints with standard HTTP header values

Header	Description
`Hook-Event`	Name of the event that triggered this delivery.
`Hook-Delivery`	Unique id for this delivery.
`Hook-HMAC`	HMAC digest that can be used to verify the sender
`Hook-Subscription`	Subscription UUID (can be used to cancel/modify)
`User-Agent`	User agent string, including Thorn and client version.
`Content-Type`	Delivery content type (e.g. application/json).

4.3.4 HTTPS/SSL Requests

Thorn supports using `https://` URLs as callbacks, but for that to work the destination web server must be properly configured for HTTPS and have a valid server certificate.

4.3.5 Buffering

By default Thorn will dispatch events as they happen, but you can also enable event buffering:

```
import thorn

with thorn.buffer_events():
    ...
```

All events sent within this block will be moved to a list, to be dispatched as soon as the block exits, or the buffer is explicitly flushed.

If you want to flush the buffer manually, you may keep a reference to the context:

```
with thorn.buffer_events() as buffer:
    Article.objects.create(...)
    Article.objects.create(...)
    buffer.flush()
    Article.objects.create(...)
    buffer.flush()
```

The dispatching backend decides what happens when you flush the buffer:

- `celery` dispatcher

  Flushing the buffer will chunk buffered requests together in sizes defined by the *THORN_CHUNKSIZE* setting.

  If the chunk size is 10 (default), this means 100 events will be delivered to workers in 10 messages.

- `default` dispatcher

  Flushing the buffer will send each event in turn, blocking the current process until all events have been sent.

Nested contexts

If you have nested `buffer_events` contexts, then only the outermost context will be active:

```
with thorn.buffer_events():
    Article.objects.create(name='A')

    with thorn.buffer_events():
        Article.objects.create(name='B')
```

```
    # << context exit delegates flush to outermost buffering context.

    Article.objects.create(name='C')
# << events for A, B, C dispatched here.
```

Note that this does NOT apply if you call `buffer.flush()` manually: that will flush events from all contexts.

Periodic flush

The context can also be used to flush the buffer periodically, using the `flush_freq` and `flush_timeout` arguments together with the `maybe_flush` method:

```
# Only flush every 100 calls, or if two seconds passed since last flush.
with thorn.buffer_events(flush_freq=100, flush_timeout=2.0) as buffer:
    for thing in things:
        process_thing_leading_to_webhook_being_sent(thing)
        buffer.maybe_flush()
```

4.4 Optimization and Performance

Table of Contents:

4.4.1 Celery

Eventlet/Gevent

By far the best way to deploy Thorn for optimal web request performance is to use the Celery eventlet/gevent pools. Which one you choose does not matter much, but some will prefer one over the other.

To start a Celery worker with the eventlet/gevent pool set the `-P` option:

```
$ celery -A proj worker -l info -P eventlet -c 1000
```

The `-c 1000` tells the worker to use up to one thousand green-threads for task execution.

Note that this will only start one OS process, so to take advantage of multiple CPUs or CPU-cores you need to start multiple processes.

This can be achived by using the `CELERYD_NODES` option to the Celery generic-init.d script, or by manually starting **`celery multi`**, for example if you have four CPU-cores you may want to start four worker instances, with a thousand green-threads each:

```
$ celery multi start 4 -A proj -P eventlet -c 1000
$ celery multi restart 4 -A proj -P eventlet -c 1000
$ celery multi stop 4 -A proj -P eventlet -c 1000
```

Eventlet: Asynchronous DNS lookups

To use the Celery eventlet pool you should make sure to install the dnspython library, to enable asynchronous DNS lookups:

```
$ pip install dnspython
```

Task retry settings

Prefetch multiplier

4.5 Configuration

Table of Contents:

4.5.1 Reference

THORN_CHUNKSIZE

Used by the Celery dispatcher to decide how many HTTP requests each task will perform.

Default is 10.

THORN_CODECS

Can be used to configure new webhook serializers, or modify existing serializers:

```
THORN_CODECS = {'application/json': serialize_json}
```

THORN_SUBSCRIBERS

This setting enables you to add static event subscribers that are not stored in the database.

This is useful for e.g hardcoded webhooks between internal systems.

The value of this setting should be a mapping between event names and subscribers, where subscribers can be:

- a URL or a list of URLs.
- a dict configured subscriber supported by `from_dict()`, or a list of these.

Example:

```
THORN_SUBSCRIBERS = {
    'user.on_create': 'https://example.com/e/on_user_created',
    'address.on_change': {
        'url': 'https://foo.example.com/e/address_change',
        'content_type': 'application/x-www-form-urlencoded',
```

```
    }
    'balance.negative': [
        'http://accounts.example.com/e/on_negative_balance',
        'http://metrics.example.com/e/on_negative_balance',
    ]
}
```

The value here can also be a callback function that returns more subscribers:

```
# can be generator, or just return list
def address_change_subscribers(event, sender=None, **kwargs):
    for url in subscribers_for('address.change'):
        yield url

THORN_SUBSCRIBERS = {
    'address.on_change': [address_change_subscribers],
}
```

THORN_DISPATCHER

The dispatcher backend to use, can be one of the built-in aliases: *"default"*, *"celery"*, or *"disabled"*, or it can be the fully qualified path to a dispatcher backend class, e.g. *"proj.dispatchers:Dispatcher"*.

Default is *"default"*.

THORN_EVENT_CHOICES

Optional configuration option to restrict the event destination choices for the Subscriber model.

THORN_HMAC_SIGNER

Specify the path to a custom HMAC signing function, taking the arguments `(digest_method,secret,message)`.

The recommended value for this setting is:

```
"thorn.utils.hmac:sign"
```

but for compatibility reasons the default is an HMAC signer using the itsdangerous library:

```
"thorn.utils.hmac:compat_sign"
```

The compat version generates a signature that is difficult for users of non-Python languages to verify, so you're highly discouraged form using the default signer.

THORN_DRF_PERMISSION_CLASSES

List of permission classes to add to the Django Rest Framework views.

THORN_EVENT_TIMEOUT

HTTP request timeout used as default when dispatching events, in seconds int/float.

Default is 3.0 seconds.

THORN_RETRY

Enable/disable retry of HTTP requests that times out or returns an error respons.

Enabled by default.

THORN_RETRY_DELAY

Time in seconds (int/float) to wait between retries. Default is one minute.

THORN_RETRY_MAX

Maximum number of retries before giving up. Default is 10.

Note that subscriptions are currently not cancelled if exceeding the maximum retry amount.

THORN_RECIPIENT_VALIDATORS

List of default validator functions to validate recipient URLs.

Individual events can override this using the `recipient_validators` argument.

The default set of validators will validate that:

- That the IP address of the recipient is not on a local network.

  > **Warning:** This only applies to IP addresses reserved for internal use, such as 127.0.0.1, and 192.168.0.0/16.
  >
  > If you have private networks on a public IP address you can block them by using the *block_cidr_network()* validator.

- The scheme of the recipient is either HTTP or HTTPS.
- The port of the recipient is either 80, or 443.

This is expressed in configuration as:

```
THORN_RECIPIENT_VALIDATORS = [
    validators.block_internal_ips(),
    validators.ensure_protocol('http', 'https'),
    validators.ensure_port(80, 443),
]
```

More validators can be found in the API reference for the *thorn.validators* module.

THORN_SIGNAL_HONORS_TRANSACTION

Default `False`

New in version 1.5.

When enabled the webhook dispatch will be tied to any current database transaction: webhook is sent on transaction commit, and ignored if the transaction rolls back.

> **Warning:** When using Django this requires Django versions 1.9 or above.

THORN_SUBSCRIBER_MODEL

Specify a custom subscriber model as a fully qualified path. E.g. for Django the default is `"thorn.django.models:Subscriber"`.

4.6 Extending

Table of Contents:

4.6.1 Environment

The environment holds framework integration specific features, and will point to a suitable implementation of the subscriber model, database signals, and the function used for reverse URL lookups.

Currently only Django is supported using the *`thorn.environment.django.DjangoEnv`* environment.

If you want to contribute an integration for another framework you can use this environment as a template for your implementation.

Autodetection

An environment is selected by calling the `autodetect()` class method on all registered environments.

The first environment to return a true value will be selected.

As an example, the Django-environment is selected only if the `DJANGO_SETTINGS_MODULE` is set.

CHAPTER 5

API Reference

Release 1.5

Date Oct 22, 2016

5.1 thorn

Python Webhook and Event Framework.

class thorn.**Thorn**(*dispatcher=None*, *set_as_current=True*)
Thorn application.

Dispatcher

Event

ModelEvent

Request

Settings

Subscriber

Subscribers

autodetect_env(*apply=<operator.methodcaller object>*)

config

disable_buffer(*owner=None*)
Disable buffering.

Raises

- *BufferNotEmpty* – if there are still items in the
- buffer when disabling.

dispatcher

dispatchers = {u'default': u'thorn.dispatch.base:Dispatcher', u'celery': u'thorn.dispatch.celery:Dispatcher', u'disabl

enable_buffer(*owner=None*)
Start buffering up events instead of dispatching them directly.

Note: User will be responsible for flushing the buffer via *flush_buffer()*, say periodically or at the end of a web request.

env

environments = set([u'thorn.environment.django:DjangoEnv'])

event_cls = u'thorn.events:Event'

flush_buffer(*owner=None*)
Flush events accumulated while buffering active.

Note: This will force send any buffered events, but the mechanics of how this happens is up to the dispatching backend:

- `default`

  Sends buffered events one by one.

- `celery`

  Sends a single message containing all buffered events, a worker will then pick that up and execute the web requests.

hmac_sign

model_event_cls = u'thorn.events:ModelEvent'

model_reverser

on_commit

request_cls = u'thorn.request:Request'

reverse

set_current()

set_default()

settings

settings_cls = u'thorn.conf:Settings'

signals

subclass_with_self(*Class*, *name=None*, *attribute=u'app'*, *reverse=None*, *keep_reduce=False*, ***kw*)
Subclass an app-compatible class.

App-compatible means the class has an 'app' attribute providing the default app, e.g.: `class Foo(object): app = None`.

Parameters **Class** (*Any*) – The class to subclass.

Keyword Arguments

- **name** (*str*) – Custom name for the target subclass.
- **attribute** (*str*) – Name of the attribute holding the app. Default is `"app"`.
- **reverse** (*str*) – Reverse path to this object used for pickling purposes. E.g. for `app.AsyncResult` use `"AsyncResult"`.

- **keep_reduce** (*bool*) – If enabled a custom `__reduce__` implementation will not be provided.

webhook_model

class thorn.**Event**(*name*, *timeout=None*, *dispatcher=None*, *retry=None*, *retry_max=None*, *retry_delay=None*, *app=None*, *recipient_validators=None*, *subscribers=None*, *request_data=None*, *allow_keepalive=None*, ***kwargs*)

Webhook Event.

Parameters **name** (*str*) – Name of this event. Namespaces can be dot-separated, and if so subscribers can glob-match based on the parts in the name, e.g. `"order.created"`.

Keyword Arguments

- **timeout** (*float*) – Default request timeout for this event.
- **retry** (*bool*) – Enable/disable retries when dispatching this event fails Disabled by default.
- **retry_max** (*int*) – Max number of retries (3 by default).
- **retry_delay** (*float*) – Delay between retries (60 seconds by default).
- **recipient_validators** (*Sequence*) – List of functions validating the recipient URL string. Functions must raise an error if the URL is blocked. Default is to only allow HTTP and HTTPS, with respective reserved ports 80 and 443, and to block internal IP networks, and can be changed using the *THORN_RECIPIENT_VALIDATORS* setting:

```
recipient_validators=[
    thorn.validators.block_internal_ips(),
    thorn.validators.ensure_protocol('http', 'https'),
    thorn.validators.ensure_port(80, 443),
]
```

- **subscribers** – Additional subscribers, as a list of URLs, subscriber model objects, or callback functions returning these
- **request_data** – Optional mapping of extra data to inject into event payloads,
- **allow_keepalive** – Flag to disable HTTP connection keepalive for this event only. Keepalive is enabled by default.

Warning: *block_internal_ips()* will only test for reserved internal networks, and not private networks with a public IP address. You can block those using *block_cidr_network*.

allow_keepalive = True

app = None

dispatcher

prepare_payload(*data*)

prepare_recipient_validators(*validators*)

Prepare recipient validator list (instance-wide).

Note: This value will be cached

Return v

prepared_recipient_validators

recipient_validators = None

send(*data*, *sender=None*, *on_success=None*, *on_error=None*, *timeout=None*, *on_timeout=None*)
Send event to all subscribers.

Parameters **data** (*Any*) – Event payload (must be json serializable).

Keyword Arguments

- **sender** (*Any*) – Optional event sender, as a `User` instance.
- **context** (*Dict*) – Extra context to pass to subscriber callbacks.
- **timeout** (*float*) – Specify custom HTTP request timeout overriding the *THORN_EVENT_TIMEOUT* setting.
- **on_success** (*Callable*) – Callback called for each HTTP request if the request succeeds. Must take single *Request* argument.
- **on_timeout** (*Callable*) – Callback called for each HTTP request if the request times out. Takes two arguments: a *Request*, and the time out exception instance.
- **on_error** (*Callable*) – Callback called for each HTTP request if the request fails. Takes two arguments: a *Request* argument, and the error exception instance.

subscribers

class thorn.**ModelEvent**(*name*, **args*, ***kwargs*)
Event related to model changes.

This event type follows a specific payload format:

```
{"event": "(str)event_name",
 "ref": "(URL)model_location",
 "sender": "(User pk)optional_sender",
 "data": {"event specific data": "value"}}
```

Parameters **name** (*str*) – Name of event.

Keyword Arguments

- **reverse** (*Callable*) – A function that takes a model instance and returns the canonical URL for that resource.
- **sender_field** (*str*) – Field used as a sender for events, e.g. `"account.user"`, will use `instance.account.user`.
- **signal_honors_transaction** (*bool*) – If enabled the webhook dispatch will be tied to any current database transaction: webhook is sent on transaction commit, and ignored if the transaction rolls back.

  Default is True (taken from the *THORN_SIGNAL_HONORS_TRANSACTION* setting), but

  requires Django 1.9 or greater. Earlier Django versions will execute the dispatch immediately.

  New in version 1.5.
- **propagate_errors** (*bool*) – If enabled errors will propagate up to the caller (even when called by signal).

  Disabled by default.

New in version 1.5.

- **signal_dispatcher** (*~thorn.django.signals.signal_dispatcher*) – Custom signal_dispatcher used to connect this event to a model signal.
- **$field__$op** (*Any*) – Optional filter arguments to filter the model instances to dispatch for. These keyword arguments can be defined just like the arguments to a Django query set, the only difference being that you have to specify an operator for every field: this means `last_name="jerry"` does not work, and you have to use `last_name__eq="jerry"` instead.

  See *Q* for more information.

See also:

In addition the same arguments as *Event* is supported.

connect_model(*model*)

dispatches_on_change()

dispatches_on_create()

dispatches_on_delete()

dispatches_on_m2m_add(*related_field*)

dispatches_on_m2m_clear(*related_field*)

dispatches_on_m2m_remove(*related_field*)

get_absolute_url(*instance*)

instance_data(*instance*)
Get event data from `instance.webhooks.payload()`.

instance_headers(*instance*)
Get event headers from `instance.webhooks.headers()`.

instance_sender(*instance*)
Get event `sender` from model instance.

on_signal(*instance*, ***kwargs*)

send(*instance*, *data=None*, *sender=None*, ***kwargs*)
Send event for model `instance`.

Keyword Arguments **data** (*Any*) – Event specific data.

See also:

Event.send() for more arguments supported.

send_from_instance(*instance*, *context={}*, ***kwargs*)

should_dispatch(*instance*, ***kwargs*)

signal_dispatcher

signal_honors_transaction

to_message(*data*, *instance=None*, *sender=None*, *ref=None*)

class `thorn.`**Q**(**args*, ***kwargs*)
Object query node.

This class works like `django.db.models.Q`, but is used for filtering regular Python objects instead of database rows.

Examples

```
>>> # Match object with `last_name` attribute set to "Costanza":
>>> Q(last_name__eq="Costanza")
```

```
>>> # Match object with `author.last_name` attribute set to "Benes":
>>> Q(author__last_name__eq="Benes")
```

```
>>> # You are not allowed to specify any key without an operator,
>>> # even when the following would be fine using Django`s Q objects:
>>> Q(author__last_name="Benes")    # <-- ERROR, will raise ValueError
```

```
>>> # Attributes can be nested arbitrarily deep:
>>> Q(a__b__c__d__e__f__g__x__gt=3.03)
```

```
>>> # The special `*__eq=True` means "match any *true-ish* value":
>>> Q(author__account__is_staff__eq=True)
```

```
>>> # Similarly the `*__eq=False` means "match any *false-y*" value":
>>> Q(author__account__is_disabled=False)
```

See also:

Supported operators.

Returns

to match an object with the given predicates, call the return value with the object to match: `Q(x__eq==808)(obj)`.

Return type Callable

apply_op(*getter*, *op*, *rhs*, *obj*, **args*)

apply_trans_op(*getter*, *op*, *rhs*, *obj*)

branches = {False: <built-in function truth>, True: <built-in function not_>}

compile(*fields*)

compile_node(*field*)

Compile node into a cached function that performs the match.

Returns taking the object to match.

Return type Callable

compile_op(*lhs*, *rhs*, *opcode*)

gate

gates = {u'AND': <built-in function all>, u'OR': <built-in function any>}

operators = {u'gt': <built-in function gt>, u'is': <built-in function is_>, u'now_eq': <function compare>, u'now_endsw

prepare_opcode(*O*, *rhs*)

prepare_statement(*lhs*, *rhs*)

stack

class thorn.**model_reverser**(*view_name*, **args*, ***kwargs*)
Describes how to get the canonical URL for a model instance.

Examples

```
>>> # This:
>>> model_reverser('article-detail', uuid='uuid')
>>> # for an article instance will generate the URL by calling:
>>> reverse('article_detail', kwargs={'uuid': instance.uuid})
```

```
>>> # And this:
>>> model_reverser('article-detail', 'category.name', uuid='uuid')
>>> # for an article instance will generate the URL by calling:
>>> reverse('article-detail',
...         args=[instance.category.name],
...         kwargs={'uuid': instance.uuid},
... )
```

thorn.**webhook_model**(**args*, ***kwargs*)
Decorate model to send webhooks based on changes to that model.

Keyword Arguments

- **on_create** (*~thorn.Event*) – Event to dispatch whenever an instance of this model is created (`post_save`).
- **on_change** (*~thorn.Event*) – Event to dispatch whenever an instance of this model is changed (`post_save`).
- **on_delete** (*~thorn.Event*) – Event to dispatch whenever an instance of this model is deleted (`post_delete`).
- **on_$event** (*~thorn.Event*) – Additional user defined events.,
- **sender_field** (*str*) – Default field used as a sender for events, e.g. `"account.user"`, will use `instance.account.user`.

  Individual events can override the sender field user.
- **reverse** (*Callable*) – A *thorn.reverse.model_reverser* instance (or any callable taking an model instance as argument), that describes how to get the URL for an instance of this model.

  Individual events can override the reverser used.

  Note: On Django you can instead define a *get_absolute_url* method on the Model.

Examples

Simple article model, where the URL reference is retrieved by `reverse('article-detail',kwargs={'uuid': article.uuid})`:

```
@webhook_model
class Article(models.Model):
    uuid = models.UUIDField()

    class webhooks:
        on_create = ModelEvent('article.created')
        on_change = ModelEvent('article.changed')
        on_delete = ModelEvent('article.removed')
        on_deactivate = ModelEvent(
            'article.deactivate', deactivated__eq=True,
        )

    @models.permalink
    def get_absolute_url(self):
        return ('blog:article-detail', None, {'uuid': self.uuid})
```

The URL may not actually exist after deletion, so maybe we want to point the reference to something else in that special case, like a category that can be reversed by doing `reverse('category-detail',args=[article.category.name])`.

We can do that by having the `on_delete` event override the method used to get the absolute url (reverser), for that event only:

```
@webhook_model
class Article(model.Model):
    uuid = models.UUIDField()
    category = models.ForeignKey('category')

    class webhooks:
        on_create = ModelEvent('article.created')
        on_change = ModelEvent('article.changed')
        on_delete = ModelEvent(
            'article.removed',
            reverse=model_reverser(
                'category:detail', 'category.name'),
        )
        on_hipri_delete = ModelEvent(
            'article.internal_delete', priority__gte=30.0,
        ).dispatches_on_delete()

    @models.permalink
    def get_absolute_url(self):
        return ('blog:article-detail', None, {'uuid': self.uuid})
```

class `thorn.`**`buffer_events`**(*flush_freq=None*, *flush_timeout=None*, *app=None*)

Context that enables event buffering.

The buffer will be flushed at context exit, or when the buffer is flushed explicitly:

```
with buffer_events() as buffer:
    ...
    buffer.flush()  # <-- flush here.
# <-- # implicit flush here.
```

`flush`()

`maybe_flush`()

`should_flush`()

5.2 thorn.app

Thorn Application.

class thorn.app.**Thorn**(*dispatcher=None*, *set_as_current=True*)

Thorn application.

Dispatcher

Event

ModelEvent

Request

Settings

Subscriber

Subscribers

autodetect_env(*apply=<operator.methodcaller object>*)

config

disable_buffer(*owner=None*)

Disable buffering.

Raises

- *BufferNotEmpty* – if there are still items in the
- buffer when disabling.

dispatcher

dispatchers = {u'default': u'thorn.dispatch.base:Dispatcher', u'celery': u'thorn.dispatch.celery:Dispatcher', u'disabl

enable_buffer(*owner=None*)

Start buffering up events instead of dispatching them directly.

Note: User will be responsible for flushing the buffer via *flush_buffer()*, say periodically or at the end of a web request.

---

env

environments = set([u'thorn.environment.django:DjangoEnv'])

event_cls = u'thorn.events:Event'

flush_buffer(*owner=None*)

Flush events accumulated while buffering active.

Note: This will force send any buffered events, but the mechanics of how this happens is up to the dispatching backend:

•default

Sends buffered events one by one.

•celery

Sends a single message containing all buffered events, a worker will then pick that up and execute the web requests.

hmac_sign

model_event_cls = u'thorn.events:ModelEvent'

model_reverser

on_commit

request_cls = u'thorn.request:Request'

reverse

set_current()

set_default()

settings

settings_cls = u'thorn.conf:Settings'

signals

subclass_with_self(*Class*, *name=None*, *attribute=u'app'*, *reverse=None*, *keep_reduce=False*, ***kw*)

Subclass an app-compatible class.

App-compatible means the class has an 'app' attribute providing the default app, e.g.: `class Foo(object): app = None`.

Parameters **Class** (*Any*) – The class to subclass.

Keyword Arguments

- **name** (*str*) – Custom name for the target subclass.
- **attribute** (*str*) – Name of the attribute holding the app. Default is `"app"`.
- **reverse** (*str*) – Reverse path to this object used for pickling purposes. E.g. for `app.AsyncResult` use `"AsyncResult"`.
- **keep_reduce** (*bool*) – If enabled a custom `__reduce__` implementation will not be provided.

webhook_model

5.3 thorn.decorators

Webhook decorators.

class thorn.decorators.**WebhookCapable**(*on_create=None*, *on_change=None*, *on_delete=None*, *reverse=None*, *sender_field=None*, ***kwargs*)

Implementation of model.webhooks.

The decorator sets model.webhooks to be an instance of this type.

connect_events(*events*, ***kwargs*)

contribute_to_event(*event*)

contribute_to_model(*model*)

delegate_to_model(*instance*, *meth*, **args*, ***kwargs*)

events = None

headers(*instance*)

payload(*instance*)

reverse = None

sender_field = None

update_events(*events*, ***kwargs*)

thorn.decorators.**webhook_model**(**args*, ***kwargs*)

Decorate model to send webhooks based on changes to that model.

Keyword Arguments

- **on_create** (*~thorn.Event*) – Event to dispatch whenever an instance of this model is created (`post_save`).
- **on_change** (*~thorn.Event*) – Event to dispatch whenever an instance of this model is changed (`post_save`).
- **on_delete** (*~thorn.Event*) – Event to dispatch whenever an instance of this model is deleted (`post_delete`).
- **on_$event** (*~thorn.Event*) – Additional user defined events.,
- **sender_field** (*str*) – Default field used as a sender for events, e.g. `"account.user"`, will use `instance.account.user`.

  Individual events can override the sender field user.
- **reverse** (*Callable*) – A *thorn.reverse.model_reverser* instance (or any callable taking an model instance as argument), that describes how to get the URL for an instance of this model.

  Individual events can override the reverser used.

  Note: On Django you can instead define a *get_absolute_url* method on the Model.

Examples

Simple article model, where the URL reference is retrieved by `reverse('article-detail',kwargs={'uuid': article.uuid})`:

```
@webhook_model
class Article(models.Model):
    uuid = models.UUIDField()

    class webhooks:
        on_create = ModelEvent('article.created')
        on_change = ModelEvent('article.changed')
        on_delete = ModelEvent('article.removed')
        on_deactivate = ModelEvent(
            'article.deactivate', deactivated__eq=True,
        )

    @models.permalink
    def get_absolute_url(self):
        return ('blog:article-detail', None, {'uuid': self.uuid})
```

The URL may not actually exist after deletion, so maybe we want to point the reference to something else in that special case, like a category that can be reversed by doing `reverse('category-detail',args=[article.category.name])`.

We can do that by having the `on_delete` event override the method used to get the absolute url (reverser), for that event only:

```
@webhook_model
class Article(model.Model):
    uuid = models.UUIDField()
    category = models.ForeignKey('category')

    class webhooks:
        on_create = ModelEvent('article.created')
        on_change = ModelEvent('article.changed')
        on_delete = ModelEvent(
            'article.removed',
            reverse=model_reverser(
                'category:detail', 'category.name'),
        )
        on_hipri_delete = ModelEvent(
            'article.internal_delete', priority__gte=30.0,
        ).dispatches_on_delete()

    @models.permalink
    def get_absolute_url(self):
        return ('blog:article-detail', None, {'uuid': self.uuid})
```

5.4 thorn.events

User-defined webhook events.

class thorn.events.**Event**(*name*, *timeout=None*, *dispatcher=None*, *retry=None*, *retry_max=None*, *retry_delay=None*, *app=None*, *recipient_validators=None*, *subscribers=None*, *request_data=None*, *allow_keepalive=None*, ***kwargs*)

Webhook Event.

Parameters **name** (*str*) – Name of this event. Namespaces can be dot-separated, and if so subscribers can glob-match based on the parts in the name, e.g. `"order.created"`.

Keyword Arguments

- **timeout** (*float*) – Default request timeout for this event.
- **retry** (*bool*) – Enable/disable retries when dispatching this event fails Disabled by default.
- **retry_max** (*int*) – Max number of retries (3 by default).
- **retry_delay** (*float*) – Delay between retries (60 seconds by default).
- **recipient_validators** (*Sequence*) – List of functions validating the recipient URL string. Functions must raise an error if the URL is blocked. Default is to only allow HTTP and HTTPS, with respective reserved ports 80 and 443, and to block internal IP networks, and can be changed using the *THORN_RECIPIENT_VALIDATORS* setting:

```
recipient_validators=[
    thorn.validators.block_internal_ips(),
    thorn.validators.ensure_protocol('http', 'https'),
```

```
    thorn.validators.ensure_port(80, 443),
]
```

- **subscribers** – Additional subscribers, as a list of URLs, subscriber model objects, or callback functions returning these
- **request_data** – Optional mapping of extra data to inject into event payloads,
- **allow_keepalive** – Flag to disable HTTP connection keepalive for this event only. Keepalive is enabled by default.

Warning: *block_internal_ips()* will only test for reserved internal networks, and not private networks with a public IP address. You can block those using *block_cidr_network*.

allow_keepalive = True

app = None

dispatcher

prepare_payload(*data*)

prepare_recipient_validators(*validators*)
Prepare recipient validator list (instance-wide).

Note: This value will be cached

Return v

prepared_recipient_validators

recipient_validators = None

send(*data*, *sender=None*, *on_success=None*, *on_error=None*, *timeout=None*, *on_timeout=None*)
Send event to all subscribers.

Parameters **data** (*Any*) – Event payload (must be json serializable).

Keyword Arguments

- **sender** (*Any*) – Optional event sender, as a `User` instance.
- **context** (*Dict*) – Extra context to pass to subscriber callbacks.
- **timeout** (*float*) – Specify custom HTTP request timeout overriding the *THORN_EVENT_TIMEOUT* setting.
- **on_success** (*Callable*) – Callback called for each HTTP request if the request succeeds. Must take single *Request* argument.
- **on_timeout** (*Callable*) – Callback called for each HTTP request if the request times out. Takes two arguments: a *Request*, and the time out exception instance.
- **on_error** (*Callable*) – Callback called for each HTTP request if the request fails. Takes two arguments: a *Request* argument, and the error exception instance.

subscribers

class thorn.events.**ModelEvent**(*name*, **args*, ***kwargs*)
Event related to model changes.

This event type follows a specific payload format:

```
{"event": "(str)event_name",
 "ref": "(URL)model_location",
 "sender": "(User pk)optional_sender",
 "data": {"event specific data": "value"}}
```

Parameters **name** (*str*) – Name of event.

Keyword Arguments

- **reverse** (*Callable*) – A function that takes a model instance and returns the canonical URL for that resource.
- **sender_field** (*str*) – Field used as a sender for events, e.g. `"account.user"`, will use `instance.account.user`.
- **signal_honors_transaction** (*bool*) – If enabled the webhook dispatch will be tied to any current database transaction: webhook is sent on transaction commit, and ignored if the transaction rolls back.

  Default is True (taken from the *THORN_SIGNAL_HONORS_TRANSACTION* setting), but

  requires Django 1.9 or greater. Earlier Django versions will execute the dispatch immediately.

  New in version 1.5.
- **propagate_errors** (*bool*) – If enabled errors will propagate up to the caller (even when called by signal).

  Disabled by default.

  New in version 1.5.
- **signal_dispatcher** (*~thorn.django.signals.signal_dispatcher*) – Custom signal_dispatcher used to connect this event to a model signal.
- **$field__$op** (*Any*) – Optional filter arguments to filter the model instances to dispatch for. These keyword arguments can be defined just like the arguments to a Django query set, the only difference being that you have to specify an operator for every field: this means `last_name="jerry"` does not work, and you have to use `last_name__eq="jerry"` instead.

  See *Q* for more information.

See also:

In addition the same arguments as *Event* is supported.

connect_model(*model*)

dispatches_on_change()

dispatches_on_create()

dispatches_on_delete()

dispatches_on_m2m_add(*related_field*)

dispatches_on_m2m_clear(*related_field*)

dispatches_on_m2m_remove(*related_field*)

get_absolute_url(*instance*)

instance_data(*instance*)
Get event data from `instance.webhooks.payload()`.

instance_headers(*instance*)
Get event headers from `instance.webhooks.headers()`.

instance_sender(*instance*)
Get event `sender` from model instance.

on_signal(*instance*, ***kwargs*)

send(*instance*, *data=None*, *sender=None*, ***kwargs*)
Send event for model `instance`.

Keyword Arguments **data** (*Any*) – Event specific data.

See also:

`Event.send()` for more arguments supported.

send_from_instance(*instance*, *context={}*, ***kwargs*)

should_dispatch(*instance*, ***kwargs*)

signal_dispatcher

signal_honors_transaction

to_message(*data*, *instance=None*, *sender=None*, *ref=None*)

5.5 thorn.reverse

Tools for URL references.

class `thorn.reverse.`**model_reverser**(*view_name*, **args*, ***kwargs*)
Describes how to get the canonical URL for a model instance.

Examples

```
>>> # This:
>>> model_reverser('article-detail', uuid='uuid')
>>> # for an article instance will generate the URL by calling:
>>> reverse('article_detail', kwargs={'uuid': instance.uuid})
```

```
>>> # And this:
>>> model_reverser('article-detail', 'category.name', uuid='uuid')
>>> # for an article instance will generate the URL by calling:
>>> reverse('article-detail',
...         args=[instance.category.name],
...         kwargs={'uuid': instance.uuid},
... )
```

5.6 thorn.request

Webhook HTTP requests.

class thorn.request.**Request**(*event*, *data*, *sender*, *subscriber*, *id=None*, *on_success=None*, *on_error=None*, *timeout=None*, *on_timeout=None*, *retry=None*, *retry_max=None*, *retry_delay=None*, *headers=None*, *user_agent=None*, *app=None*, *recipient_validators=None*, *allow_keepalive=True*)

Webhook HTTP request.

Parameters

- **event** (*str*) – Name of event.
- **data** (*Any*) – Event payload.
- **sender** (*Any*) – Sender of event (or None).
- **subscriber** (*Subscriber*) – Subscriber to dispatch the request for.

Keyword Arguments

- **on_success** (*Callable*) – Optional callback called if the HTTP request succeeds. Must take single argument: request.
- **on_timeout** (*Callable*) – Optional callback called if the HTTP request times out. Must have signature: (request,exc).
- **on_error** (*Callable*) – Optional callback called if the HTTP request fails. Must have signature: (request,exc).
- **headers** (*Mapping*) – Additional HTTP headers to send with the request.
- **user_agent** (*str*) – Set custom HTTP user agent.
- **recipient_validators** (*Sequence*) – List of serialized recipient validators.
- **allow_keepalive** (*bool*) – Allow reusing session for this HTTP request. Enabled by default.
- **retry** (*bool*) – Retry in the event of timeout/failure? Disabled by default.
- **retry_max** (*int*) – Maximum number of times to retry before giving up. Default is 3.
- **retry_delay** (*float*) – Delay between retries in seconds int/float. Default is 60 seconds.

class Session

A Requests session.

Provides cookie persistence, connection-pooling, and configuration.

Basic Usage:

```
>>> import requests
>>> s = requests.Session()
>>> s.get('http://httpbin.org/get')
<Response [200]>
```

Or as a context manager:

```
>>> with requests.Session() as s:
>>>     s.get('http://httpbin.org/get')
<Response [200]>
```

close()

Closes all adapters and as such the session

delete(*url*, ***kwargs*)

Sends a DELETE request. Returns Response object.

Parameters

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

get(*url*, ***kwargs*)

Sends a GET request. Returns `Response` object.

Parameters

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

get_adapter(*url*)

Returns the appropriate connection adapter for the given URL.

Return type requests.adapters.BaseAdapter

head(*url*, ***kwargs*)

Sends a HEAD request. Returns `Response` object.

Parameters

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

merge_environment_settings(*url*, *proxies*, *stream*, *verify*, *cert*)

Check the environment and merge it with some settings.

Return type dict

mount(*prefix*, *adapter*)

Registers a connection adapter to a prefix.

Adapters are sorted in descending order by key length.

options(*url*, ***kwargs*)

Sends a OPTIONS request. Returns `Response` object.

Parameters

- **url** – URL for the new *Request* object.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

patch(*url*, *data=None*, ***kwargs*)

Sends a PATCH request. Returns `Response` object.

Parameters

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

post(*url*, *data=None*, *json=None*, ***kwargs*)

Sends a POST request. Returns `Response` object.

Parameters

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- **json** – (optional) json to send in the body of the *Request*.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

prepare_request(*request*)

Constructs a `PreparedRequest` for transmission and returns it. The `PreparedRequest` has

settings merged from the *Request* instance and those of the Session.

Parameters **request** – *Request* instance to prepare with this session's settings.

Return type requests.PreparedRequest

put(*url*, *data=None*, ***kwargs*)

Sends a PUT request. Returns Response object.

Parameters

- **url** – URL for the new *Request* object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- ****kwargs** – Optional arguments that request takes.

Return type requests.Response

request(*method*, *url*, *params=None*, *data=None*, *headers=None*, *cookies=None*, *files=None*, *auth=None*, *timeout=None*, *allow_redirects=True*, *proxies=None*, *hooks=None*, *stream=None*, *verify=None*, *cert=None*, *json=None*)

Constructs a *Request*, prepares it and sends it. Returns Response object.

Parameters

- **method** – method for the new *Request* object.
- **url** – URL for the new *Request* object.
- **params** – (optional) Dictionary or bytes to be sent in the query string for the *Request*.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the *Request*.
- **json** – (optional) json to send in the body of the *Request*.
- **headers** – (optional) Dictionary of HTTP Headers to send with the *Request*.
- **cookies** – (optional) Dict or CookieJar object to send with the *Request*.
- **files** – (optional) Dictionary of `'filename': file-like-objects` for multipart encoding upload.
- **auth** – (optional) Auth tuple or callable to enable Basic/Digest/Custom HTTP Auth.
- **timeout** (*float or tuple*) – (optional) How long to wait for the server to send data before giving up, as a float, or a (connect timeout, read timeout) tuple.
- **allow_redirects** (*bool*) – (optional) Set to True by default.
- **proxies** – (optional) Dictionary mapping protocol or protocol and hostname to the URL of the proxy.
- **stream** – (optional) whether to immediately download the response content. Defaults to False.
- **verify** – (optional) whether the SSL cert will be verified. A CA_BUNDLE path can also be provided. Defaults to True.
- **cert** – (optional) if String, path to ssl client cert file (.pem). If Tuple, ('cert', 'key') pair.

Return type requests.Response

send(*request*, ***kwargs*)

Send a given PreparedRequest.

Return type requests.Response

Request.**annotate_headers**(*extra_headers*)

Request.**app** = **None**

Request.**as_dict**()

Return dictionary representation of this request.

Note: All values must be json serializable.

Request.**connection_errors** = **(<class 'requests.exceptions.ConnectionError'>,)**

Tuple of exceptions considered a connection error.

Request.**default_headers**

Request.**dispatch**(*session=None*, *propagate=False*)

Request.**handle_connection_error**(*exc*, *propagate=False*)

Request.**handle_timeout_error**(*exc*, *propagate=False*)

Request.**headers**

Request.**post**(*session=None*)

Request.**recipient_validators**

Request.**response** = None
Holds the response after the HTTP request is performed.

Request.**session_or_acquire**(**args*, ***kwds*)

Request.**sign_request**(*subscriber*, *data*)

Request.**timeout_errors** = (<class 'requests.exceptions.Timeout'>,)
Tuple of exceptions considered a timeout error.

Request.**urlident**
Used to order HTTP requests by URL.

Request.**user_agent** = u'Mozilla/5.0 (compatible; thorn/1.5.0; python-requests/2.11.1)'
HTTP User-Agent header.

Request.**validate_recipient**(*url*)

Request.**value**

5.7 thorn.validators

Recipient Validators.

thorn.validators.**ensure_protocol**(**allowed*)
Only allow recipient URLs using specific protocols.

Example

```
>>> ensure_protocol('https', 'http://')
```

thorn.validators.**ensure_port**(**allowed*)
Validator that ensures port is member of set allowed.

thorn.validators.**block_internal_ips**()
Block recipient URLs that have an internal IP address.

> **Warning:** This does not check for *private* networks, it will only make sure the IP address is not in a reserved private block (e.g. 192.168.0.1/24).

thorn.validators.**block_cidr_network**(**blocked_networks*)
Block recipient URLs from a list of CIDR networks.

Example

```
>>> block_cidr_network('192.168.0.0/24', '132.34.23.0/24')
```

5.8 thorn.exceptions

Thorn-related exceptions.

exception thorn.exceptions.**BufferNotEmpty**
Trying to close buffer that is not empty.

exception thorn.exceptions.**ImproperlyConfigured**
Configuration invalid/missing.

exception thorn.exceptions.**SecurityError**
Security related error.

exception thorn.exceptions.**ThornError**
Base class for Thorn exceptions.

5.9 thorn.conf

Webhooks-related configuration settings.

thorn.conf.**event_choices**(*app=None*)
Return a list of valid event choices.

5.10 thorn.tasks

Tasks used by the Celery dispatcher.

(task)thorn.tasks.**send_event**(*event*, *payload*, *sender*, *timeout*, *context={}*)
Task called by process dispatching the event.

Note: This will use the WorkerDispatcher to dispatch the individual HTTP requests in batches (`dispatch_requests -> dispatch_request`).

(task)thorn.tasks.**dispatch_requests**(*reqs*, *app=None*)
Process a batch of HTTP requests.

(task)thorn.tasks.**dispatch_request**(*self*, *event*, *data*, *sender*, *subscriber*, *session=None*, *app=None*, ***kwargs*)
Process a single HTTP request.

5.11 thorn.environment

Framework integration.

5.12 thorn.environment.django

Django web framework environment.

class thorn.environment.django.**DjangoEnv**
Thorn Django environment.

Subscriber

Subscribers

static autodetect(*env=u'DJANGO_SETTINGS_MODULE'*)

config

on_commit(*fun*, **args*, ***kwargs*)

reverse

reverse_cls = u'django.core.urlresolvers:reverse'

settings_cls = u'django.conf:settings'

signals

signals_cls = u'thorn.django.signals'

subscriber_cls = u'thorn.django.models:Subscriber'

5.13 thorn.django.models

Django models required to dispatch webhook events.

class thorn.django.models.**Subscriber**(*id*, *uuid*, *event*, *url*, *user*, *hmac_secret*, *hmac_digest*, *content_type*, *created_at*, *updated_at*)

exception DoesNotExist

exception Subscriber.**MultipleObjectsReturned**

Subscriber.**content_type**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

Subscriber.**created_at**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

Subscriber.**event**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

Subscriber.**get_content_type_display**(**moreargs*, ***morekwargs*)

Subscriber.**get_hmac_digest_display**(**moreargs*, ***morekwargs*)

Subscriber.**get_next_by_created_at**(**moreargs*, ***morekwargs*)

Subscriber.**get_next_by_updated_at**(**moreargs*, ***morekwargs*)

Subscriber.**get_previous_by_created_at**(**moreargs*, ***morekwargs*)

Subscriber.**get_previous_by_updated_at**(**moreargs*, ***morekwargs*)

Subscriber.**hmac_digest**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

Subscriber.**hmac_secret**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

Subscriber.**id**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

Subscriber.**objects = <thorn.django.managers.SubscriberManager object>**

Subscriber.**updated_at**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

Subscriber.**url**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

Subscriber.**user**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

Subscriber.**user_id**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

Subscriber.**user_ident**()

Subscriber.**uuid**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

5.14 thorn.django.managers

Django Managers and query sets.

class thorn.django.managers.**SubscriberQuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)

matching(*event*, *user=None*)

matching_event(*event*)

matching_user_or_all(*user*)

class thorn.django.managers.**SubscriberManager**

5.15 thorn.django.signals

Django signal dispatchers.

class thorn.django.signals.**dispatch_on_create**(*fun*, *use_transitions=False*, ***kwargs*)

setup_signals()

should_dispatch(*instance*, *raw=False*, *created=False*, ***kwargs*)

class thorn.django.signals.**dispatch_on_change**(*fun*, *use_transitions=False*, ***kwargs*)

on_pre_save(*instance*, *sender*, *raw=False*, ***kwargs*)

setup_signals()

should_dispatch(*instance*, *created=False*, *raw=False*, ***kwargs*)

class thorn.django.signals.**dispatch_on_delete**(*fun*, *use_transitions=False*, ***kwargs*)

setup_signals()

class thorn.django.signals.**dispatch_on_m2m_add**(*fun*, *related_field*, ***kwargs*)

setup_actions()

class thorn.django.signals.**dispatch_on_m2m_remove**(*fun*, *related_field*, ***kwargs*)

setup_actions()

class thorn.django.signals.**dispatch_on_m2m_clear**(*fun*, *related_field*, ***kwargs*)

setup_actions()

5.16 thorn.django.rest_framework.urls

DRF URL dispatch table.

5.16.1 Usage

To include the rest-framework API views in your project use the `django.conf.urls.include()` function, with a proper namespace argument:

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

urlpatterns = [
    url(r'^hooks/',
```

```
        include('thorn.django.rest_framework.urls', namespace='webhook')),
]
```

5.16.2 Endpoints

Two new API endpoints will now be available in your application:

- `GET /hooks/`

  List of webhook subscriptions related to the currently logged in user.

- `POST /hooks/`

  Create new subscription owned by the currently logged in user.

- `GET /hooks/<uuid>/`

  Get detail about specific subscription by unique identifier (uuid).

- `POST|PATCH /hook/<uuid>/`

  Update subscription given its unique identifier (uuid).

- `DELETE /hook/<uuid>/`

  Delete subscription given its unique identifier (uuid).

5.17 thorn.django.rest_framework.views

DRF Views.

API endpoints for users to create and manage their webhook subscriptions.

class thorn.django.rest_framework.views.**SubscriberList**(***kwargs*)
List and create new subscriptions for the currently logged in user.

get_queryset()

model
alias of Subscriber

perform_create(*serializer*)

permission_classes

serializer_class
alias of SubscriberSerializer

class thorn.django.rest_framework.views.**SubscriberDetail**(***kwargs*)
Update, delete or get details for specific subscription.

Note: User must be logged in, and user can only see subscriptions owned by them.

---

get_object()

lookup_field = u'uuid'

model
alias of Subscriber

permission_classes

serializer_class
alias of SubscriberSerializer

5.18 thorn.django.rest_framework.serializers

DRF serializers.

class thorn.django.rest_framework.serializers.**SubscriberSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Serializer for *Subscriber*.

class Meta
Serializer configuration.

fields = (u'event', u'url', u'content_type', u'user', u'id', u'created_at', u'updated_at', u'subscription', u'hmac_se

model
alias of Subscriber

read_only_fields = (u'id', u'created_at', u'updated_at', u'subscription')

5.19 thorn.django.utils

Django-related utilities.

thorn.django.utils.**serialize_model**(*m*)

thorn.django.utils.**deserialize_model**(*m*)

5.20 thorn.dispatch

5.21 thorn.dispatch.base

Default webhook dispatcher.

class thorn.dispatch.base.**Dispatcher**(*timeout=None*, *app=None*, *buffer=False*)

app = None

disable_buffer(*owner=None*)

dispatch_request(*request*)

enable_buffer(*owner=None*)

encode_cached(*payload*, *cache*, *ctype*)

encode_payload(*data*, *content_type*)

flush_buffer(*owner=None*)

prepare_requests(*event*, *payload*, *sender*, *timeout=None*, *context=None*, *extra_subscribers=None*, ***kwargs*)

send(*event*, *payload*, *sender*, *context=None*, *extra_subscribers=None*, *allow_keepalive=True*, ***kwargs*)

subscribers_for_event(*name*, *sender=None*, *context={}*, *extra_subscribers=None*)
Return a list of *Subscriber* subscribing to an event by name (optionally filtered by sender).

5.22 thorn.dispatch.disabled

Dispatcher doing nothing.

class thorn.dispatch.disabled.**Dispatcher**(*timeout=None*, *app=None*, *buffer=False*)

send(**args*, ***kwargs*)

5.23 thorn.dispatch.celery

Celery-based webhook dispatcher.

class thorn.dispatch.celery.**Dispatcher**(*timeout=None*, *app=None*, *buffer=False*)
Dispatcher using Celery tasks to dispatch events.

Note: Overrides what happens when `thorn.webhook.Event.send()` is called so that dispatching the HTTP request tasks is performed by a worker, instead of in the current process.

---

flush_buffer()

send(*event*, *payload*, *sender*, *timeout=None*, *context=None*, ***kwargs*)

class thorn.dispatch.celery.**WorkerDispatcher**(*timeout=None*, *app=None*, *buffer=False*)
Dispatcher used by the *thorn.tasks.send_event()* task.

send(*event*, *payload*, *sender*, *timeout=None*, *context=None*, ***kwargs*)

5.24 thorn.generic.models

Generic base model mixins.

class thorn.generic.models.**AbstractSubscriber**
Abstract class for Subscriber identity.

as_dict()
Dictionary representation of Subscriber.

content_type
MIME-type to use for web requests made to the subscriber *url*.

event
Event pattern this subscriber is subscribed to (e.g. `article.*`).

from_dict(**args*, ***kwargs*)
Create subscriber from dictionary representation.

> **Note:** Accepts the same arguments as `dict`.

hmac_digest
HMAC digest type (e.g. `"sha512"`).

The value used must be a member of `hashlib.algorithms_available`.

hmac_secret
HMAC secret key, of arbitrary length.

classmethod register(*other*)

sign(*message*)
Sign message using HMAC.

> **Note:** *hmac_secret* and the current *hmac_digest* type must be set.

url
Destination URL to dispatch this event.

user
User filter – when set only dispatch if the event sender matches.

user_ident()
Return *user* identity.

> **Note:** Value must be json serializable like a database primary key.

uuid
Unique identifier.

class thorn.generic.models.**SubscriberModelMixin**
Mixin for subscriber models.

as_dict()

classmethod from_dict(**args*, ***kwargs*)

sign(*message*)

5.25 thorn.generic.signals

Dispatching by signal.

class thorn.generic.signals.**signal_dispatcher**(*fun*, *use_transitions=False*, ***kwargs*)
Signal dispatcher abstraction.

connect(*sender=None*, *weak=False*, ***kwargs*)

context(*instance*, ***kwargs*)

load_signals(*signals*)

prepare_sender(*sender*)

setup_signals()

should_dispatch(*instance*, ***kwargs*)

signals = **None**

5.26 thorn.utils.compat

Python version compatibility utilities.

thorn.utils.compat.**bytes_if_py2**(*s*)
Convert str to bytes.

5.27 thorn.utils.functional

Functional-style utilities.

thorn.utils.functional.**chunks**(*it*, *n*)
Split an iterator into chunks with *n* elements each.

Example

n == 2 >>> x = chunks(iter([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]), 2) >>> list(x) [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9], [10]]

n == 3 >>> x = chunks(iter([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]), 3) >>> list(x) [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10]]

class thorn.utils.functional.**Q**(**args*, ***kwargs*)
Object query node.

This class works like django.db.models.Q, but is used for filtering regular Python objects instead of database rows.

Examples

```
>>> # Match object with `last_name` attribute set to "Costanza":
>>> Q(last_name__eq="Costanza")
```

```
>>> # Match object with `author.last_name` attribute set to "Benes":
>>> Q(author__last_name__eq="Benes")
```

```
>>> # You are not allowed to specify any key without an operator,
>>> # even when the following would be fine using Django`s Q objects:
>>> Q(author__last_name="Benes")    # <-- ERROR, will raise ValueError
```

```
>>> # Attributes can be nested arbitrarily deep:
>>> Q(a__b__c__d__e__f__g__x__gt=3.03)
```

```
>>> # The special `*__eq=True` means "match any *true-ish* value":
>>> Q(author__account__is_staff__eq=True)
```

```
>>> # Similarly the `*__eq=False` means "match any *false-y*" value":
>>> Q(author__account__is_disabled=False)
```

See also:

Supported operators.

Returns

to match an object with the given predicates, call the return value with the object to match: `Q(x__eq==808)(obj)`.

Return type Callable

apply_op(*getter*, *op*, *rhs*, *obj*, **args*)

apply_trans_op(*getter*, *op*, *rhs*, *obj*)

branches = {False: <built-in function truth>, True: <built-in function not_>}
If the node is negated (`~a` / `a.negate()`), branch will be True, and we reverse the query into a `not a` one.

compile(*fields*)

compile_node(*field*)
Compile node into a cached function that performs the match.

Returns taking the object to match.

Return type Callable

compile_op(*lhs*, *rhs*, *opcode*)

gate

gates = {u'AND': <built-in function all>, u'OR': <built-in function any>}
The gate decides the boolean operator of this tree node. A node can either be *OR* (`a | b`), or an *AND* note (`a & b`). - Default is *AND*.

operators = {u'gt': <built-in function gt>, u'is': <built-in function is_>, u'now_eq': <function compare>, u'now_endsw
Mapping of opcode to binary operator function – `f(a,b)`. Operators may return any true-ish or false-y value.

prepare_opcode(*O*, *rhs*)

prepare_statement(*lhs*, *rhs*)

stack

5.28 thorn.utils.hmac

HMAC Message signing utilities.

thorn.utils.hmac.**compat_sign**(*digest_method*, *key*, *message*)
Sign message using old itsdangerous signer.

thorn.utils.hmac.**get_digest**(*d*)
Get digest type by name (e.g. `"sha512"`).

thorn.utils.hmac.**random_secret**(*length*, *chars=u'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01 -./:;<=>?@[\\]^_`{|}~'*)
Generate random secret (letters, digits, punctuation).

thorn.utils.hmac.**sign**(*digest_method*, *key*, *message*)
Sign HMAC digest.

thorn.utils.hmac.**verify**(*digest*, *digest_method*, *key*, *message*)
Verify HMAC digest.

5.29 thorn.utils.json

Json serialization utilities.

class thorn.utils.json.**JsonEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)
Thorn custom Json encoder.

Notes

Same as django.core.serializers.json.JSONEncoder but preserves datetime microsecond information.

default(*o*, *dates=(<type 'datetime.datetime'>, <type 'datetime.date'>)*, *times=(<type 'datetime.time'>,)*, *textual=(<class 'decimal.Decimal'>, <class 'uuid.UUID'>, <class 'django.utils.functional.Promise'>)*, *isinstance=<built-in function isinstance>*, *datetime=<type 'datetime.datetime'>*, *text_type=<type 'unicode'>*)

thorn.utils.json.**dumps**(*obj*, *encode=<function dumps>*, *cls=<class 'thorn.utils.json.JsonEncoder'>*)
Serialize object as json string.

5.30 thorn.utils.log

Logging utilities.

thorn.utils.log.**get_logger**(*name*, *parent=<celery.utils.log.ProcessAwareLogger object>*)
Get logger by name.

5.31 thorn.funtests.base

Base-class for functional test suites.

Extends Cyanide stress test suite with utilities used to test Thorn.

class thorn.funtests.base.**WebhookSuite**(*app*, *no_color=False*, ***kwargs*)
Thorn integration test suite.

assert_article_event_received(*article*, *event*, *sub=None*, *reverse=None*, *ref=None*, *n=1*)

assert_log(*ref=None*, *maxlen=1*)

assert_log_matches(*log*, ***expected*)

assert_ok_pidbox_response(*replies*)

assert_webhook_not_received(*ref=None*)

create_article(*title*, *state=u'PENDING'*, *author=None*)

delete(**path*, ***data*)

get(**path*, ***data*)

headers()

hook_clear(*event*)

hook_subscribe(*event*, *url*, *callback=None*)

hook_unsubscribe(*event*, *url*)

list_subscriptions()

override_worker_setting(**args*, ***kwds*)

post(**path*, ***data*)

reverse_article(*article*)

setenv(*setting_name*, *new_value*)

setup()

subscribe(*event*, *ref=None*, *rest=None*)

token = None

token_type = u'Token'

unsubscribe(*url*)

user = None

user2 = None

wait_for_webhook_received(*ref=None*, *maxlen=1*)

worker_subscribe_to(**args*, ***kwds*)

thorn.funtests.base.**new_ref**()
Create new reference ID.

thorn.funtests.base.**url**(**s*)
Create URL by components in *s.

thorn.funtests.base.**event_url**(*event*, *ref=None*, *rest=None*)
Return url for event.

thorn.funtests.base.**testcase**(**groups*, ***kwargs*)

5.32 thorn.funtests.suite

Functional test suite.

5.32.1 Instructions

1. Start the celery worker:

```
$ celery -A thorn.funtests worker -l info -P eventlet -c 1000
```

2. Start the development web server:

```
$ python manage.py runserver
```

3. Then execute the functional test suite:

```
$ celery -A thorn.funtests cyanide
```

5.32.2 Tips

For a list of tests that you can select see:

```
$ celery -A thorn.funtests cyanide -l
```

class thorn.funtests.suite.**Default**(*app*, *no_color=False*, ***kwargs*)
Thorn integration test suite.

endpoints()

hundred_subscribers(*event=u'article.created'*)

sender_mismatch_does_not_dispatch(*event=u'article.changed'*)

subscribe_to_article_changed(*event=u'article.changed'*)

subscribe_to_article_created(*event=u'article.created'*)

subscribe_to_article_published(*event=u'article.published'*)

subscribe_to_article_removed(*event=u'article.removed'*)

subscribe_to_tag_added(*event=u'article.tag_added'*)

subscribe_to_tag_all_cleared(*event=u'article.tag_all_cleared'*)

subscribe_to_tag_removed(*event=u'article.tag_removed'*)

subscriber_callback_setting(*event=u'article.changed'*)

subscriber_setting(*event=u'article.changed'*)

unsubscribe_does_not_dispatch(*event=u'article.created'*)

thorn.funtests.suite.**callback_subscribers**(*event*, *sender=None*, ***kwargs*)
Return a dummy set of callback subscribers.

5.33 thorn.funtests.tasks

Tasks used for functional testing.

Custom Celery worker remote control commands used in the Thorn functional test suite.

thorn.funtests.tasks.**find_subscriber**(*self*, *subs*, *url*)
Find specific subscriber by URL.

thorn.funtests.tasks.**hook_clear**(*state*, *event*)
Clear recorded hooks.

thorn.funtests.tasks.**hook_subscribe**(*state*, *event*, *url=None*, *callback=None*)
Subscribe to webhook.

thorn.funtests.tasks.**hook_unsubscribe**(*state*, *event*, *url*)
Unsubscribe from webhook.

thorn.funtests.tasks.**setenv**(*state*, *setting_name*, *new_value*)
Remote control command to set Thorn application setting.

thorn.funtests.tasks.**subscribers_for_event**(*event*)
Get a list of subscribers for an even by name.

CHAPTER 6

Changelog

6.1 1.5.0

release-date 2016-10-20 11:08 A.M PDT

release-by Ask Solem

- New API for *ModelEvent*.

  After having used Thorn for a while, there was a realization that passing lots of arguments to a decorator looks very messy when there are many events for a model.

  We have come up with a new way to add webhooks to models, that we believe is more tidy.

  The new API moves declaration of webhooks related things into a nested class:

```
@webhook_model
class Article(models.Model):
    uuid = models.UUIDField()
    title = models.CharField(max_length=128)
    state = models.CharField(max_length=128, default='PENDING')
    body = models.TextField()
    user = models.ForeignKey(settings.AUTH_USER_MODEL)

    class webhooks:
        on_create = ModelEvent('article.created')
        on_change = ModelEvent('article.changed')
        on_delete = ModelEvent('article.removed')
        on_publish = ModelEvent(
            'article.published', state__now_eq='PUBLISHED',
        ).dispatches_on_change()

        def payload(self, article):
            return {
                'title': article.title,
            }

        def headers(self, article):
            return {
                'Authorization':
                    'Bearer {}'.format(article.user.access_token),
            }
```

> **Note:** The old API is still supported, and there's no current plans of deprecating the arguments to the decorator itself.

- Adds support for buffering events - moving dispatch out of signal handlers.

  See *Buffering* for more information.

- Models can now define additional headers to be passed on to webhooks.

  See *Modifying event headers*.

  Contributed by **Flavio Curella**.

- Django: *ModelEvent* now takes advantage of `Model.get_absolute_url()`.

  Instead of defining a reverser you can now simply have your model define a `get_absolute_url` method (which is a convention already):

```
class Article(models.Model):

    @models.permalink
    def get_absolute_url(self):
        return 'article:detail', (), {'slug': self.slug}
```

  For more information on defining this method, refer to the Django documentation: https://docs.djangoproject.com/en/stable/ref/models/instances/#get-absolute-url

- New *THORN_SIGNAL_HONORS_TRANSACTION* setting.

  If enabled the Django model events will dispatch only after the transaction is committed, and should the transaction be rolled back the events are discarded.

  You can also enable this setting on individual events:

```
ModelEvent(..., signal_honors_transaction=True)
```

  Disabled by default.

  To be enabled by default in Thorn 2.0.

- *ModelEvent* now logs errors instead of propagating them.

  `ModelEvent(propagate_errors=True)` may be set to revert to the old behavior.

- URLs now ordered by scheme,host,port (was host,port,scheme).

- Documentation improvements by:

  - Flavio Curella

6.2 1.4.2

release-date 2016-07-28 11:45 A.M PDT

- Serialize uuid as string when calling Celery tasks.

6.3 1.4.1

release-date 2016-07-26 01:06 P.M PDT

release-by Ask Solem

- Fixed webhook dispatch crash where validator was deserialized twice.
- Celery dispatcher did not properly forward the *Hook-Subscription* HTTP header.
- Source code now using Google-style docstrings.

6.4 1.4.0

release-date 2016-07-11 04:27 P.M PDT

release-by Ask Solem

- New HTTP header now sent with events: `Hook-Subscription`

  This contains the UUID of the subscription, which means a webhook consumer may now react by cancelling or modifying the subscription.

- Fixed missing default value for the *THORN_SUBSCRIBER_MODEL* setting.
- Fixed HMAC signing wrong message value.

6.5 1.3.0

release-date 2016-07-07 07:40 P.M PDT

release-by Ask Solem

- New and improved method for HMAC signing.

  The new method must be enabled manually by setting:

  ```
  THORN_HMAC_SIGNER = 'thorn.utils.hmac:sign'
  ```

  It turns out itsdangerous did not do what we expected it to, instead it does this:

  - The secret key is transformed into:

    ```
     key = hashlib.sha256(salt + 'signer' + HMAC_SECRET)
    # strip = from beginning and end of the base64 string
    key = key.strip('=')
    ```

  - If you don't specify a salt, which we don't, there is a default salt(!) which is:

    ```
    "itsdangerous.Signer"
    ```

  - The extra "signer" in the key transformation is there as the default key_derivation method is called "django-concat".
  - The final signature is encoded using "urlsafe_b64encode"

    So in Python to recreate the signature using the built-in hmac library you would have to do:

```
import hashlib
import hmac
from base64 import urlsafe_b64encode

# everything hardcoded to SHA256 here

def create_signature(secret_key, message):
    key = hashlib.sha256(
        'itsdangerous.Signer' + 'signer' + secret_key).digest()
    digest = hmac.new(key, message, digestmod=hashlib.sha256).
↪digest()
    return urlsafe_b64encode(digest).replace('=')
```

which is much more complicated than what we can expect of users.

You're highly encouraged to enable the new HMAC method, but sadly it's not backwards compatible.

We have also included new examples for verifying HMAC signatures in Django, Ruby, and PHP in the documentation.

- New *THORN_SUBSCRIBER_MODEL* setting.
- New *THORN_HMAC_SIGNER* setting.
- Requirements: Tests now depends on case 1.2.2
- JSON: Make sure simplejson does not convert `Decimal` to `float`.
- class:*~thorn.events.ModelEvent*: name can now be a string format.

  Contributed by Flavio Curella.

  The format expands using the model instance affected, e.g:

  ```
  on_created=ModelEvent('created.{.occasion}')
  ```

  means the format will expand into `instance.occasion`.

  Subclasses of *ModelEvent* can override how the name is expanded by defining the `_get_name` method.

6.6 1.2.1

release-date 2016-06-06 06:30 P.M PDT

release-by Ask Solem

- Celery: Forward event signal context to the tasks.

6.7 1.2.0

release-date 2016-06-02 01:00 P.M PDT

release-by Ask Solem

- Event: Adds `request_data` option.

  This enables you to inject additional data into the webhook payload, used for integration with quirky HTTP endpoints.

- Event: Adds `allow_keepalive` option.

  HTTP connections will not be reused for an event if this flag is set to False. Keepalive is enabled by default.

- Event: Adds `subscribers` argument that can be used to add default subscribers for the event.

  This argument can hold the same values as the *`THORN_SUBSCRIBERS`* setting.

- **Decorator: `model.webhook_events` is now a UserDict proxy** to `model.webhook_events.events`.

- Subscriber: `thorn.generic.models.AbstractSubscribers` is a new abstract interface for subscriber models.

  This should be used if you want to check if an object is a subscriber in `isinstance()` checks.

- **Q: `now__*` operators now properly handles the case when there's** no previous version of the object.

- Django: `django.db.models.signals.pre_save` signal handler now ignores `ObjectDoesNotExist` errors.

- Events: Adds new `prepare_recipient_validators` method, enabling subclasses to e.g. set default validators.

- Windows: Unit test suite now passing on win32/win64.

- Module `thorn.models` renamed to *`thorn.generic.models`*.

6.8 1.1.0

release-date 2016-05-23 12:00 P.M PDT

release-by Ask Solem

- Fixed installation on Python 3

  Fix contributed by Josh Drake.

- Now depends on
  - itsdangerous
  - ipaddress (Python 2.7)

- Security: Now provides HMAC signing by default.

  The Subscriber model has a new `hmac_secret` field which subscribers can provide to set the secret key for communication. A default secret will be created if none is provided, and can be found in the response of the subscribe endpoint.

  The signed HMAC message found in the `Hook-HMAC` HTTP header can then be used to verify the sender of the webhook.

  An example Django webhook consumer verifying the signature can be found in the *Django guide*.

  Thanks to Timothy Fitz for suggestions.

- Security: No longer dispatches webhooks to internal networks.

  This means Thorn will refuse to deliver webhooks to networks considered internal, like `fd00::/8`, `10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16` and `127.0.0.1`

  This behavior can be changed globally using the *`THORN_RECIPIENT_VALIDATORS`* setting, or on an per-event basis using the `recipient_validators` argument to `event`.

- Security: Now only dispatches to HTTP and HTTPS URLs by default.

  This behavior can be changed globally using the *THORN_RECIPIENT_VALIDATORS* setting, or on an per-event basis using the `recipient_validators` argument to `event`.

- Security: Now only dispatches to ports 80 and 443 by default.

  This behavior can be changed globally using the *THORN_RECIPIENT_VALIDATORS* setting, or on an per-event basis using the `recipient_validators` argument to `event`.

- Security: Adds recipient validators

  You can now validate the recipient URL by providing a list of validators in the `recipient_validators` argument to *Event*.

  The default list of validators is provided by the new *THORN_RECIPIENT_VALIDATORS* setting.

  Thanks to Edmond Wong for reviewing, and Timothy Fitz for suggestions.

- Django: Now properly supports custom user models by using `UserModel.get_username()`.

  Fix contributed by Josh Drake.

- ModelEvent: Adds new many-to-many signal dispatcher types

  - `dispatches_on_m2m_add(related_field)`

    Sent when a new object is added to a many-to-many relation.

  - `dispatches_on_m2m_remove(related_field)`

    Sent when an object is removed from a many-to-many relation.

  - `dispatches_on_m2m_clear(related_field)`

    Sent when a many-to-many relation is cleared.

  Example

  In this blog article model, events are sent whenever a new tag is added or removed:

```
@webhook_model(
    on_add_tag=ModelEvent(
        'article.tagged').dispatches_on_m2m_add('tags'),
    on_remove_tag=ModelEvent(
        'article.untagged').dispatches_on_m2m_remove('tags'),
    on_clear_tags=ModelEvent(
        'article.tags_cleared').dispatches_on_m2m_clear('tags'),
)
class Article(models.Model):
    title = models.CharField(max_length=128)
    tags = models.ManyToManyField(Tag)


class Tag(models.Model):
    name = models.CharField(max_length=64, unique=True)
```

  The `article.tagged` webhook is sent when:

```
>>> python_tag, _ = Tag.objects.get_or_create(name='python')
>>> article.tags.add(python_tag)  # <-- dispatches with this line
```

  and the `article.untagged` webhook is sent when:

```
>>> article.tags.remove(python_tag)
```

finally, the `article.tags_cleared` event is sent when:

```
>>> article.tags.clear()
```

- Documentation fixes contributed by:
  - Matthew Brener

6.9 1.0.0

release-date 2016-05-13 10:10 A.M PDT

release-by Ask Solem

- Initial release :o)

CHAPTER 7

Contributing

Welcome!

This document is fairly extensive and you are not really expected to study this in detail for small contributions;

> The most important rule is that contributing must be easy and that the community is friendly and not nitpicking on details such as coding style.

If you're reporting a bug you should read the Reporting bugs section below to ensure that your bug report contains enough information to successfully diagnose the issue, and if you're contributing code you should try to mimic the conventions you see surrounding the code you are working on, but in the end all patches will be cleaned up by the person merging the changes so don't worry too much.

7.1 Community Code of Conduct

The goal is to maintain a diverse community that is pleasant for everyone. That is why we would greatly appreciate it if everyone contributing to and interacting with the community also followed this Code of Conduct.

The Code of Conduct covers our behavior as members of the community, in any forum, mailing list, wiki, website, Internet relay chat (IRC), public meeting or private correspondence.

The Code of Conduct is heavily based on the Ubuntu Code of Conduct, and the Pylons Code of Conduct.

7.1.1 Be considerate.

Your work will be used by other people, and you in turn will depend on the work of others. Any decision you take will affect users and colleagues, and we expect you to take those consequences into account when making decisions. Even if it's not obvious at the time, our contributions to Thorn will impact the work of others. For example, changes to code, infrastructure, policy, documentation and translations during a release may negatively impact others work.

7.1.2 Be respectful.

The Thorn community and its members treat one another with respect. Everyone can make a valuable contribution to Thorn. We may not always agree, but disagreement is no excuse for poor behavior and poor manners. We might all experience some frustration now and then, but we cannot allow that frustration to turn into a personal attack. It's important to remember that a community where people feel uncomfortable or threatened is not a productive one. We expect members of the Thorn community to be respectful when dealing with other contributors as well as with people outside the Thorn project and with users of Thorn.

7.1.3 Be collaborative.

Collaboration is central to Thorn and to the larger free software community. We should always be open to collaboration. Your work should be done transparently and patches from Thorn should be given back to the community when they are made, not just when the distribution releases. If you wish to work on new code for existing upstream projects, at least keep those projects informed of your ideas and progress. It many not be possible to get consensus from upstream, or even from your colleagues about the correct implementation for an idea, so don't feel obliged to have that agreement before you begin, but at least keep the outside world informed of your work, and publish your work in a way that allows outsiders to test, discuss and contribute to your efforts.

7.1.4 When you disagree, consult others.

Disagreements, both political and technical, happen all the time and the Thorn community is no exception. It is important that we resolve disagreements and differing views constructively and with the help of the community and community process. If you really want to go a different way, then we encourage you to make a derivative distribution or alternate set of packages that still build on the work we've done to utilize as common of a core as possible.

7.1.5 When you are unsure, ask for help.

Nobody knows everything, and nobody is expected to be perfect. Asking questions avoids many problems down the road, and so questions are encouraged. Those who are asked questions should be responsive and helpful. However, when asking a question, care must be taken to do so in an appropriate forum.

7.1.6 Step down considerately.

Developers on every project come and go and Thorn is no different. When you leave or disengage from the project, in whole or in part, we ask that you do so in a way that minimizes disruption to the project. This means you should tell people you are leaving and take the proper steps to ensure that others can pick up where you leave off.

7.2 Reporting Bugs

7.2.1 Security

You must never report security related issues, vulnerabilities or bugs including sensitive information to the bug tracker, or elsewhere in public. Instead sensitive bugs must be sent by email to `security@robinhood.com`.

If you'd like to submit the information encrypted our PGP key is:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.15 (Darwin)

mQENBFJpWDkBCADFIc9/Fpgse4owLNvsTC7GYfnJL19XO0hnL99sPx+DPbfr+cSE
9wiU+Wp2TfUX7pCLEGrODiEP6ZCZbgtiPgId+JYvMxpP6GXbjiIlHRw1EQNH8RlX
cVxy3rQfVv8PGGiJuyBBjxzvETHW25htVAZ5TI1+CkxmuyyEYqgZN2fNd0wEU19D
+c10G1gSECbCQTCbacLSzdpngAt1Gkrc96r7wGHBBSvDaGDD2pFSkVuTLMbIRrVp
lnKOPMsUijiip2EMr2DvfuXiUIUvaqInTPNWkDynLoh69ib5xC19CSVLONjkKBsr
Pe+qAY29liBatatpXsydY7GIUzyBT3MzgMJlABEBAAG0MUNlbGVyeSBTZWN1cml0
eSBUZWFtIDxzZWN1cml0eUBjZWxlcnlwcm9qZWN0Lm9yZz6JATgEEwECACIFAlJp
WDkCGwMGCwkIBwMCBhUIAgkKCwQWAgMBAh4BAheAAAoJEOArFOUDCicIw1IH/26f
CViDC7/P13jr+srRdjAsWvQztia9HmTlY8cUnbmkR9w6b6j3F2ayw8VhkyFWgYEJ
wtPBv8mHKADiVSFARS+0yGsfCkia5wDSQuIv6XqRlIrXUyqJbmF4NUFTyCZYoh+C
ZiQpN9xGhFPr5QDlMx2izWg1rvWlG1jY2Es1v/xED3AeCOB1eUGvRe/uJHKjGv7J
rj0pFcptZX+WDF22AN235WYwgJM6TrNfSu8sv8vNAQOVnsKcgsqhuwomSGsOfMQj
LFzIn95MKBBU1G5wOs7JtwiV9jefGqJGBO2FAvOVbvPdK/saSnB+7K36dQcIHqms
5hU4Xj0RIJiod5idlRC5AQ0EUmlYOQEIAJs8OwHMkrdcvy9kk2HBVbdqhgAREMKy
gmphDp7prRL9FqSY/dKpCbG0u82zyJypdb7QiaQ5pfPzPpQcd2dIcohkkh7G3E+e
hS2L9AXHpwR26/PzMBXyr2iNnNc4vTksHvGVDxzFnRpka6vbI/hrrZmYNYh9EAiv
uhE54b3/XhXwFgHjZXb9i8hgJ3nsO0pRwvUAM1bRGMbvf8e9F+kqgV0yWYNnh6QL
4Vpl1+epqp2RKPHyNQftbQyrAHXT9kQF9pPlx013MKYaFTADscuAp4T3dy7xmiwS
crqMbZLzfrxfFOsNxTUGE5vmJCcm+mybAtRo4aV6ACohAO9NevMx8pUAEQEAAYkB
HwQYAQIACQUCUmlYOQIbDAAKCRDgKxTlAwonCNFbB/9esir/f7TufE+isNqErzR/
aZKZo2WzZR9c75kbqo6J6DYuUHe6xI0OZ2qZ60iABDEZAiNXGulysFLCiPdatQ8x
8zt3DF9BMkEck54ZvAjpNSern6zfZb1jPYWZq3TKxlTs/GuCgBAuV4i5vDTZ7xK/
aF+OFY5zN7ciZHkqLgMiTZ+RhqRcK6FhVBP/Y7d9NlBOcDBTxxE1ZO1ute6n7guJ
ciw4hfoRk8qNN19szZuq3UU64zpkM2sBsIFM9tGF2FADRxiOaOWZHmIyVZriPFqW
RUwjSjs7jBVNq0Vy4fCu/5+e+XLOUBOoqtM5W7ELt0t1w9tXebtPEetV86in8fU2
=0chn
-----END PGP PUBLIC KEY BLOCK-----
```

7.2.2 Other bugs

Bugs can always be described to the *Mailing list*, but the best way to report an issue and to ensure a timely response is to use the issue tracker.

1. **Create a GitHub account.**

You need to create a GitHub account to be able to create new issues and participate in the discussion.

2. **Determine if your bug is really a bug.**

You should not file a bug if you are requesting support. For that you can use the *Mailing list*, or *IRC*.

3. **Make sure your bug hasn't already been reported.**

Search through the appropriate Issue tracker. If a bug like yours was found, check if you have new information that could be reported to help the developers fix the bug.

4. **Check if you're using the latest version.**

A bug could be fixed by some other improvements and fixes - it might not have an existing report in the bug tracker. Make sure you're using the latest release of Thorn, and try the development version to see if the issue is already fixed and pending release.

5. **Collect information about the bug.**

To have the best chance of having a bug fixed, we need to be able to easily reproduce the conditions that caused it. Most of the time this information will be from a Python traceback message, though some bugs might be in design, spelling or other errors on the website/docs/code.

1. If the error is from a Python traceback, include it in the bug report.
2. We also need to know what platform you're running (Windows, macOS, Linux, etc.), the version of your Python interpreter, and the version of Thorn, and related packages that you were running when the bug occurred.
6. **Submit the bug.**

By default GitHub will email you to let you know when new comments have been made on your bug. In the event you've turned this feature off, you should check back on occasion to ensure you don't miss any questions a developer trying to fix the bug might ask.

7.2.3 Issue Tracker

The Thorn issue tracker can be found at GitHub: https://github.com/robinhood/thorn

7.3 Versions

Version numbers consists of a major version, minor version and a release number, and conforms to the SemVer versioning spec: http://semver.org.

Stable releases are published at PyPI while development releases are only available in the GitHub git repository as tags. All version tags starts with "v", so version 0.8.0 is the tag v0.8.0.

7.4 Branches

Current active version branches:

- master (https://github.com/robinhood/thorn/tree/master)

You can see the state of any branch by looking at the Changelog:

https://github.com/robinhood/thorn/blob/master/Changelog

If the branch is in active development the topmost version info should contain meta-data like:

```
2.4.0
======
:release-date: TBA
:status: DEVELOPMENT
:branch: master
```

The `status` field can be one of:

- `PLANNING`

  The branch is currently experimental and in the planning stage.

- `DEVELOPMENT`

  The branch is in active development, but the test suite should be passing and the product should be working and possible for users to test.

- `FROZEN`

  The branch is frozen, and no more features will be accepted. When a branch is frozen the focus is on testing the version as much as possible before it is released.

7.4.1 `master` branch

The master branch is where development of the next version happens.

7.4.2 Maintenance branches

Maintenance branches are named after the version, e.g. the maintenance branch for the 2.2.x series is named `2.2`. Previously these were named `releaseXX-maint`.

The versions we currently maintain is:

- 1.0

  This is the current series.

7.4.3 Archived branches

Archived branches are kept for preserving history only, and theoretically someone could provide patches for these if they depend on a series that is no longer officially supported.

An archived version is named `X.Y-archived`.

Thorn does not currently have any archived branches.

7.4.4 Feature branches

Major new features are worked on in dedicated branches. There is no strict naming requirement for these branches.

Feature branches are removed once they have been merged into a release branch.

7.5 Tags

Tags are used exclusively for tagging releases. A release tag is named with the format `vX.Y.Z`, e.g. `v2.3.1`. Experimental releases contain an additional identifier `vX.Y.Z-id`, e.g. `v3.0.0-rc1`. Experimental tags may be removed after the official release.

7.6 Working on Features & Patches

Note: Contributing to Thorn should be as simple as possible, so none of these steps should be considered mandatory.

You can even send in patches by email if that is your preferred work method. We won't like you any less, any contribution you make is always appreciated!

However following these steps may make maintainers life easier, and may mean that your changes will be accepted sooner.

7.6.1 Forking and setting up the repository

First you need to fork the Thorn repository, a good introduction to this is in the GitHub Guide: Fork a Repo.

After you have cloned the repository you should checkout your copy to a directory on your machine:

```
$ git clone git@github.com:username/thorn.git
```

When the repository is cloned enter the directory to set up easy access to upstream changes:

```
$ cd thorn
$ git remote add upstream git://github.com/robinhood/thorn.git
$ git fetch upstream
```

If you need to pull in new changes from upstream you should always use the `--rebase` option to `git pull`:

```
git pull --rebase upstream master
```

With this option you don't clutter the history with merging commit notes. See Rebasing merge commits in git. If you want to learn more about rebasing see the Rebase section in the GitHub guides.

If you need to work on a different branch than `master` you can fetch and checkout a remote branch like this:

```
git checkout --track -b 3.0-devel origin/3.0-devel
```

7.6.2 Running the unit test suite

To run the Thorn test suite you need to install a few dependencies. A complete list of the dependencies needed are located in `requirements/test.txt`.

If you're working on the development version, then you need to install the development requirements first:

```
$ pip install -U -r requirements/dev.txt
```

Both the stable and the development version have testing related dependencies, so install these next:

```
$ pip install -U -r requirements/test.txt
$ pip install -U -r requirements/default.txt
```

After installing the dependencies required, you can now execute the test suite by calling:

```
$ python setup.py test
```

This will run all of the test, to run individual tests you can call **`py.test`** directly:

```
$ py.test
```

Some useful options to **`py.test`** are:

- `-x`

  Stop running the tests at the first test that fails.

- `-s`

  Don't capture output

If you want to run the tests for a single test file only you can do so like this:

```
$ py.test t/unit/test_request.py
```

7.6.3 Running the functional test suite

Thorn uses cyanide for functional/integration tests, but note that this requires a working Celery installation.

1. Start the celery worker:

   ```
   $ celery -A thorn.funtests worker -l info -P eventlet -c 1000
   ```

2. Start the development web server:

   ```
   $ python manage.py runserver)
   ```

3. Then execute the functional test suite:

   ```
   $ celery -A thorn.funtests cyanide
   ```

For a list of tests that you can select see:

```
$ celery -A thorn.funtests cyanide -l
```

7.6.4 Creating pull requests

When your feature/bugfix is complete you may want to submit a pull requests so that it can be reviewed by the maintainers.

Creating pull requests is easy, and also let you track the progress of your contribution. Read the Pull Requests section in the GitHub Guide to learn how this is done.

You can also attach pull requests to existing issues by following the steps outlined here: http://bit.ly/koJoso

Calculating test coverage

To calculate test coverage you must first install the coverage module.

Installing the coverage module:

```
$ pip install -U coverage
```

Code coverage in HTML:

```
$ make cov
```

The coverage output will then be located at `cover/index.html`.

Running the tests on all supported Python versions

There is a tox configuration file in the top directory of the distribution.

To run the tests for all supported Python versions simply execute:

```
$ tox
```

Use the `tox -e` option if you only want to test specific Python versions:

```
$ tox -e 2.7
```

7.6.5 Building the documentation

To build the documentation you need to install the dependencies listed in `requirements/docs.txt`:

```
$ pip install -U -r requirements/docs.txt
```

After these dependencies are installed you should be able to build the docs by running:

```
$ cd docs
$ rm -rf _build
$ make html
```

Make sure there are no errors or warnings in the build output. After building succeeds the documentation is available at `_build/html`.

7.6.6 Verifying your contribution

To use these tools you need to install a few dependencies. These dependencies can be found in `requirements/pkgutils.txt`.

Installing the dependencies:

```
$ pip install -U -r requirements/pkgutils.txt
```

pyflakes & PEP8

To ensure that your changes conform to PEP8 and to run pyflakes execute:

```
$ make flakecheck
```

To not return a negative exit code when this command fails use the `flakes` target instead:

```
$ make flakes
```

API reference

To make sure that all modules have a corresponding section in the API reference please execute:

```
$ make apicheck
$ make configcheck
```

If files are missing you can add them by copying an existing reference file.

If the module is internal it should be part of the internal reference located in `docs/internals/reference/`. If the module is public it should be located in `docs/reference/`.

For example if reference is missing for the module `thorn.awesome` and this module is considered part of the public API, use the following steps:

Use an existing file as a template:

```
$ cd docs/reference/
$ cp thorn.request.rst thorn.awesome.rst
```

Edit the file using your favorite editor:

```
$ vim thorn.awesome.rst

    # change every occurrence of ``thorn.request`` to
    # ``thorn.awesome``
```

Edit the index using your favorite editor:

```
$ vim index.rst

    # Add ``thorn.awesome`` to the index.
```

Commit your changes:

```
# Add the file to git
$ git add thorn.awesome.rst
$ git add index.rst
$ git commit thorn.awesome.rst index.rst \
    -m "Adds reference for thorn.awesome"
```

7.7 Coding Style

You should probably be able to pick up the coding style from surrounding code, but it is a good idea to be aware of the following conventions.

- All Python code must follow the PEP-8 guidelines.

pep8.py is an utility you can use to verify that your code is following the conventions.

- Docstrings must follow the PEP-257 conventions, and use the following style.

  Do this:

  ```
  def method(self, arg):
      """Short description.

      More details.

      """
  ```

  or:

  ```
  def method(self, arg):
      """Short description."""
  ```

  but not this:

  ```
  def method(self, arg):
      """
      Short description.
      """
  ```

- Lines should not exceed 78 columns.

  You can enforce this in **vim** by setting the `textwidth` option:

```
set textwidth=78
```

  If adhering to this limit makes the code less readable, you have one more character to go on, which means 78 is a soft limit, and 79 is the hard limit :)

- Import order
  - Python standard library (*import xxx*)
  - Python standard library ('from xxx import')
  - Third-party packages.
  - Other modules from the current package.

  or in case of code using Django:
  - Python standard library (*import xxx*)
  - Python standard library ('from xxx import')
  - Third-party packages.
  - Django packages.
  - Other modules from the current package.

  Within these sections the imports should be sorted by module name.

  Example:

```
import threading
import time

from collections import deque
from Queue import Queue, Empty

from .datastructures import TokenBucket
from .five import zip_longest, items, range
from .utils import timeutils
```

- Wild-card imports must not be used (*from xxx import **).
- For distributions where Python 2.5 is the oldest support version additional rules apply:
  - Absolute imports must be enabled at the top of every module:

```
from __future__ import absolute_import
```

  - If the module uses the `with` statement and must be compatible with Python 2.5 (thorn is not) then it must also enable that:

```
from __future__ import with_statement
```

  - Every future import must be on its own line, as older Python 2.5 releases did not support importing multiple features on the same future import line:

```
# Good
from __future__ import absolute_import
from __future__ import with_statement
```

```
# Bad
from __future__ import absolute_import, with_statement
```

(Note that this rule does not apply if the package does not include support for Python 2.5)

- Note that we use "new-style' relative imports when the distribution does not support Python versions below 2.5

  This requires Python 2.5 or later:

```
from . import submodule
```

7.8 Contributing features requiring additional libraries

Some features like a new result backend may require additional libraries that the user must install.

We use setuptools *extra_requires* for this, and all new optional features that require third-party libraries must be added.

1. Add a new requirements file in *requirements/extras*

   E.g. for a Cassandra backend this would be `requirements/extras/cassandra.txt`, and the file looks like this:

```
pycassa
```

   These are pip requirement files so you can have version specifiers and multiple packages are separated by newline. A more complex example could be:

```
# pycassa 2.0 breaks Foo
pycassa>=1.0,<2.0
thrift
```

2. Modify `setup.py`

   After the requirements file is added you need to add it as an option to `setup.py` in the `extras_require` section:

```
extra['extras_require'] = {
    # ...
    'cassandra': extras('cassandra.txt'),
}
```

3. Document the new feature in `docs/includes/installation.txt`

   You must add your feature to the list in the Bundles section of `docs/includes/installation.txt`.

   After you've made changes to this file you need to render the distro `README` file:

```
$ pip install -U requirements/pkgutils.txt
$ make readme
```

That's all that needs to be done, but remember that if your feature adds additional configuration options then these needs to be documented in `docs/configuration.rst`. Also all settings need to be added to the `thorn/conf.py` module.

7.9 Contacts

This is a list of people that can be contacted for questions regarding the official git repositories, PyPI packages Read the Docs pages.

If the issue is not an emergency then it is better to *report an issue*.

7.9.1 Committers

Ask Solem

github https://github.com/ask

twitter http://twitter.com/#!/asksol

7.10 Packages

7.10.1 `thorn`

git https://github.com/robinhood/thorn

CI http://travis-ci.org/#!/robinhood/thorn

Windows-CI https://ci.appveyor.com/project/robinhood/thorn

PyPI http://pypi.python.org/pypi/thorn

docs http://thorn.readthedocs.io

7.11 Release Procedure

7.11.1 Updating the version number

The version number must be updated two places:

- `thorn/__init__.py`
- `docs/include/introduction.txt`

After you have changed these files you must render the `README` files. There is a script to convert sphinx syntax to generic reStructured Text syntax, and the make target *readme* does this for you:

```
$ make readme
```

Now commit the changes:

```
$ git commit -a -m "Bumps version to X.Y.Z"
```

and make a new version tag:

```
$ git tag vX.Y.Z
$ git push --tags
```

7.11.2 Releasing

Commands to make a new public stable release:

```
$ make distcheck  # checks pep8, autodoc index, runs tests and more
$ make dist  # NOTE: Runs git clean -xdf and removes files not in the repo.
$ python setup.py sdist upload --sign --identity='Ask Solem'
$ python setup.py bdist_wheel upload --sign --identity='Ask Solem'
```

If this is a new release series then you also need to do the following:

- **Go to the Read The Docs management interface at:** http://readthedocs.org/projects/thorn?fromdocs=thorn
- Enter "Edit project"

  Change default branch to the branch of this series, e.g. `2.4` for series 2.4.
- Also add the previous version under the "versions" tab.

CHAPTER 8

Glossary

celery A distributed task queue library (http://celeryproject.org).

dispatch The act of notifying all subscriptions subscribed to a webhook, by performing one or more HTTP requests.

subscriber An URL subscribing to a webhook.

subscription The actual subscription that can be cancelled. Identified by a universally unique identifier (UUID4).

webhook An HTTP callback.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

t

A

B

C

D

E

F

N

O

P

Q

R

S

T

U

V

W