django-issue Documentation

Release 3.1.2

Josh Marlow

Jan 03, 2019

Contents

CHAPTER 1

Overview

Sometimes things go wrong in production; if it is a repeating or ongoing error, it makes sense to represent and track this in some way. This is the purpose of the `Issue` class. When an error is detected, an `Issue` can be created to store details about it.

Possible advantages of this:

- When something goes wrong, corrective actions that are taken often should not be repeated.
- It gives admins an ability to view at a glance a history of actions taken by the system to address the issue.

Once an `Issue` is created, it is often desirable to *act* on it. For this, django-issue provides a `Responder` model. A `Responder` specifies a pattern to match against ``Issue`s; when a pattern matches for an :class:`Issue`` to a 'Responder', the `Responder` executes some configured action.

How are `Issues` created? They can be easily created by any bit of code. Alternatively, you can use the `Assertion`. The goal of an `Assertion` is to provide a means for detecting when certain properties of your system no longer hold true.

Think of it as a cross between the classic `assert` statement available in many programming langauge and traditional software monitoring systems like Nagios.

CHAPTER 2

Philosophy

It's often the case that you know how your system should behave (you built it). The problem is, your system doesn't know how it's supposed to behave. So when it missbehaves (due to bugs, unexpected edge-cases, user error, malware, a cat climbing on a keyboard), it doesn't realize that something is amiss and continues on it's way doing something terribly wrong. If the system had a more explicit notion of it's expected behavior, then it could try to correct deviations, or at the very least, to escalate to a human and ask for help.

django-issue is an initial exploration into these ideas; how can we detect when things go wrong, represent the fact that they have gone wrong, and then respond to them?

CHAPTER 3

Examples

CHAPTER 4

Representing the fact that something is amiss

Suppose an error occurs in the middle of the night that needs to be addressed in the morning (but is not pressing enough to wake someone up). We could do something like this:

```
from isssue.models import Issue

try:
    // a problem occurs
except ValueError as ve:
    Issue.objects.create(name='That *impossible* edge case finally happened...',␣
↪details=str(ve))
```

CHAPTER 5

How do you know if something has gone wrong?

Enter the `Assertion` class. Suppose you have a model that tracks a heartbeat from some external service/software components. If, after some amount of time, the sytem does not receive a heartbeat, a human should be notified. Suppose you have an app called ‘heartbeat’ and your models.py file looks like this:

```
# Models.py
from datetime import datetime, timedelta

from django.db import models


class HeartbeatKeeper(model.Model):
    last_heartbeat = models.DateTimeField(auto_now=True)


def check_for_recent_heartbeat(**kwargs):
    """
    Returns (True, None) when all is well.
    Returns (False, None) otherwise.
    """
    delta = timedelta(minutes=30)
    interval = (datetime.utcnow() - HeartbeatKeeper.objects.get().last_heartbeat)
    return (interval < delta, None)
```

Now you create an `Assertion` to call your `check_for_recent_heartbeat()` function and create an Issue when it returns a tuple beginning with False:

```
Assertion.objects.create(target_function='heartbeat.models.check_for_recent_heartbeat
↪', name='Check for heartbeat')
```

When the check_for_recent_heartbeat function returns a False tuple, then an Issue is created with the name ‘Check for heartbeat’).

There is a special type of a `Assertion` called a `ModelAssertion`. A `ModelAssertion` is designed to ensure that certain properties hold true for the models in your database.

Suppose you have a Profile model for your Users. After 5 days of signing up, you want to be notified if the user hasn't created a profile pic yet. You have an app, 'profile', and your models.py file looks like this:

```
# Models.py
from datetime import datetime, timedelta

from django.contrib.auth.models import Group, User
from django.db import models


class Profile(model.Model):
    user = models.ForeignKey(User)
    # Note: selfies are preferred
    pic_url = models.URLField(null=True, blank=True)


def should_have_pic_by_now(record, **kwargs):
    """
    Check if the specified user has a pic or still has time for one.
    """
    interval = timedelta(days=5)

    has_pic = record.pic_url is not None
    date_joined = record.user.date_joined

    okay = has_pic or (datetime.utcnow() - date_joined) < interval

    return (okay, None)
```

Now you create an `ModelAssertion` to call your `check_for_recent_heartbeat()` function and create an Issue when it returns a tuple beginning with False:

```
from django.contrib.contenttypes.models import ContentType

from profile.models import Profile


ModelAssertion.objects.create(
    target_function='profile.models.should_have_pic_by_now', name='Check for pic',␣
↪model_type=ContentType.get_for_model(Profile))
```

Now whenever a `User` account (and associated `Profile`) is created, an Issue is created if the user does not set a profile pic within 5 days.

CHAPTER 6

Addressing an ongoing problem

Now when this exception occurs, we will have a record in the database along with details about what happened and when. Now suppose we want an email notification when this happens. Well we could add the following:

```
from issue.models import Responder, ResponderAction


r = Responder.objects.create(watch_pattern='That \*impossible\* edge case finally␣
↪happened')

ResponderAction.objects.create(responder=r, delay_sec=30, target_function='issue.
↪actions.email',
    function_kwargs={
        'subject': 'Doh!',
        'recipients': 'john.smith@example.com',
    })
```

There is a helper function, `build_responder()` for constructing a `Responder` and one or more associated `ResponderAction` from json:

```
from issue.builder import build_responder


build_responder({
    'watch_pattern': 'That \'impossible\' edge case finally happened...',
    'actions': [
        {
            'target_function': 'issue.actions.email',
            'function_kwargs': {
                'subject': 'Doh!',
                'recipients': 'john.smith@example.com',
            },
            'delay_sec': 30,
        },
        {
```

(continues on next page)

(continued from previous page)

```
            'target_function': 'issue.actions.email',
            'function_kwargs': {
                'subject': 'Doh-2!',
                'recipients': 'john.smith-boss@example.com',
            },
            'delay_sec': 1800,
        },
    ]})
```

The `delay_sec` may be ommitted; when this happens the ResponderAction will be executed as soon as the Responder matches against an Issue.

CHAPTER 7

When do these checks happen?

Two management commands are provided, **`check_assertions`** and **`respond_to_issues`** which should be ran periodically.