django-background-tasks Documentation

Release latest

Oct 18, 2018

Contents

Django Background Task is a databased-backed work queue for Django, loosely based around Ruby's DelayedJob library. This project was adopted and adapted from this repo.

To avoid conflicts on PyPI we renamed it to django-background-tasks (plural). For an easy upgrade from django-background-task to django-background-tasks, the internal module structure were left untouched.

In Django Background Task, all tasks are implemented as functions (or any other callable).

There are two parts to using background tasks:

- creating the task functions and registering them with the scheduler
- setup a cron task (or long running process) to execute the tasks

CHAPTER 1

Installation

Install from PyPI:

```
pip install django-background-tasks
```

Add to INSTALLED_APPS:

```
INSTALLED_APPS = (
    # ...
    'background_task',
    # ...
)
```

Migrate your database:

```
python manage.py migrate
```

CHAPTER 2

Supported versions and compatibility

- Python: 2.7, 3.4-3.7
- Django: 1.8, 1.11, 2.1, 2.2

Full Django LTS to LTS compatibility through django-compat.

Note: Django 1.8 is an expired LTS release. It's not advisable to use this version of Django anymore.

CHAPTER 3

Creating and registering tasks

To register a task use the background decorator:

```
from background_task import background
from django.contrib.auth.models import User

@background(schedule=60)
def notify_user(user_id):
    # lookup user by id and send them a message
    user = User.objects.get(pk=user_id)
    user.email_user('Here is a notification', 'You have been notified')
```

This will convert the notify_user into a background task function. When you call it from regular code it will actually create a Task object and stores it in the database. The database then contains serialised information about which function actually needs running later on. This does place limits on the parameters that can be passed when calling the function - they must all be serializable as JSON. Hence why in the example above a user_id is passed rather than a User object.

Calling notify_user as normal will schedule the original function to be run 60 seconds from now:

```
notify_user(user.id)
```

This is the default schedule time (as set in the decorator), but it can be overridden:

```
notify_user(user.id, schedule=90) # 90 seconds from now
notify_user(user.id, schedule=timedelta(minutes=20)) # 20 minutes from now
notify_user(user.id, schedule=timezone.now()) # at a specific time
```

Also you can run original function right now in synchronous mode:

```
notify_user.now(user.id)    # launch a notify_user function and wait for it
notify_user = notify_user.now    # revert task function back to normal function.␣
↪Useful for testing.
```

You can specify a verbose name and a creator when scheduling a task:

```
notify_user(user.id, verbose_name="Notify user", creator=user)
```

The creator is stored as a `GenericForeignKey`, so any model may be used.

To get the functions decorated by `background` picked up by the auto discovery mechanism, they must be placed in a file named `tasks.py` in your module, eg. `myapp/tasks.py`.

CHAPTER 4

Repeating Tasks

Repeating tasks can be initialized like this:

```
notify_user(user.id, repeat=<number of seconds>, repeat_until=<datetime or None>)
```

When a repeating task completes successfully, a new Task with an offset of `repeat` is scheduled. On the other hand, if a repeating task fails and is not restarted, the repetition chain is stopped.

`repeat` is given in seconds. The following constants are provided: `Task.NEVER` (default), `Task.HOURLY`, `Task.DAILY`, `Task.WEEKLY`, `Task.EVERY_2_WEEKS`, `Task.EVERY_4_WEEKS`.

The time offset is computed from the initially scheduled time of the original task, not the time the task was actually executed. If the process command is interrupted, the interval between the original task and its repetition may be shorter than `repeat`.

CHAPTER 5

Multiple Queues

You can pass a queue name to the `background` decorator:

```
@background(queue='my-queue')
def notify_user(user_id):
    ...
```

If you run the command `process_tasks` with the option `--queue <queue_name>` you can restrict the tasks processed to the given queue.

CHAPTER 6

Running tasks

There is a management command to run tasks that have been scheduled:

```
python manage.py process_tasks
```

This will simply poll the database queue every few seconds to see if there is a new task to run.

The `process_tasks` management command has the following options:

- `duration` - Run task for this many seconds (0 or less to run forever) - default is 0
- `sleep` - Sleep for this many seconds before checking for new tasks (if none were found) - default is 5
- `log-std` - Redirect stdout and stderr to the logging system

You can use the `duration` option for simple process control, by running the management command via a cron job and setting the duration to the time till cron calls the command again. This way if the command fails it will get restarted by the cron job later anyway. It also avoids having to worry about resource/memory leaks too much. The alternative is to use a grown-up program like supervisord to handle this for you.

CHAPTER 7

Settings

There are a few settings options that can be set in your `settings.py` file.

- `MAX_ATTEMPTS` - controls how many times a task will be attempted (default 25)
- `MAX_RUN_TIME` - maximum possible task run time, after which tasks will be unlocked and tried again (default 3600 seconds)
- `BACKGROUND_TASK_RUN_ASYNC` - If `True`, will run the tasks asynchronous. This means the tasks will be processed in parallel (at the same time) instead of processing one by one (one after the other).
- `BACKGROUND_TASK_ASYNC_THREADS` - Specifies number of concurrent threads. Default is `multiprocessing.cpu_count()`.
- `BACKGROUND_TASK_PRIORITY_ORDERING` - Control the ordering of tasks in the queue. Default is `"DESC"` (tasks with a higher number are processed first). Choose `"ASC"` to switch to the "niceness" ordering. A niceness of 20 is the highest priority and 19 is the lowest priority.

CHAPTER 8

Task errors

Tasks are retried if they fail and the error recorded in last_error (and logged). A task is retried as it may be a temporary issue, such as a transient network problem. However each time a task is retried it is retried later and later, using an exponential back off, based on the number of attempts:

```
(attempts ** 4) + 5
```

This means that initially the task will be tried again a few seconds later. After four attempts the task is tried again 261 seconds later (about four minutes). At twenty five attempts the task will not be tried again for nearly four days! It is not unheard of for a transient error to last a long time and this behavior is intended to stop tasks that are triggering errors constantly (i.e. due to a coding error) form dominating task processing. You should probably monitor the task queue to check for tasks that have errors. After `MAX_ATTEMPTS` the task will be marked as failed and will not be rescheduled again.

CHAPTER 9

Known issues

- `django.db.utils.OperationalError: database is locked` when using SQLite. This is a SQLite specific error, see https://docs.djangoproject.com/en/dev/ref/databases/#database-is-locked-errors for more details.

CHAPTER 10

Example project

Hiroaki Nakamura has written an example project demonstrating how django-background-tasks works. You find it here.

CHAPTER 11

Tests

You can run the test suite on all supported versions of Django and Python:

```
$ tox
```

CHAPTER 12

Contributing

Anyone and everyone is welcome to contribute. Please take a moment to review the guidelines for contributing.