Django uWSGI taskmanager

Release 1.0.2

G. Giaccari, G. Lucci, G. Celata, P. Melchiorre

Jan 15, 2023

CONTENTS

uWSGI taskmanager is a Django application that can be used to launch management tasks *asynchronously*, via the standard Django admin interface, using uWSGI spooler.

The rationale for this app is to let people having access to the django admin interface, launch or schedule management tasks, without having to consult the *developers* or *operations* teams.

The **features** include:

- start and stop tasks via the django admin interface
- schedule tasks for future executions
- program periodic tasks launch
- check, filter and download the generated log messages, watching how created live
- simply write a standard Django *Command* class (your app doesn't need to interact with Django uWSGI Taskmanager)
- get notifications via Slack, email or build a custom notification class

An animated GIF of how it all works. Click to enlarge.

CHAPTER

ONE

GET STARTED

Following the demo tutorial, it will be possible to install, configure and use **django-uwsgi-taskmanager** for a simple demo django project and have an idea of its basic workings.

Further knowledge can be found in the *How-to guides*.

1.1 The demo tutorial

Clone the project from github onto your hard disk:

```
git clone https://github.com/openpolis/django-uwsgi-taskmanager
cd django-uwsgi-taskmanager
```

There is a basic Django project under the `demo` directory, with a `uwsgi.ini` file and four directories (`media`, `spooler`, `static`, `venv`).

```
demo/
├── demo/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── manage.py
├── media/
├── spooler/
├── static/
├── uwsgi.ini
└── venv/
```

Modify the content of `uwsgi.ini`, if needed, for example by changing the port, if already in use, and adding the number of processes.

Following is the content of my file, while writing this tutorial:

```
[uwsgi]
chdir = %d
env = DJANGO_SETTINGS_MODULE=demo.settings
http-socket = :8000
master = true
module = demo.wsgi
plugin = python3
```

(continues on next page)

(continued from previous page)

```
pythonpath = %d
processes = 2
spooler-processes = 1
spooler = %dspooler
static-map = /static/=%dstatic
virtualenv = %dvenv
```

Note: Remember not to use this configuration in production, as it lets uWSGI handle all http connections, even for static content. Usually a frontend server, and/or CDN connections are used along the uWSGI app server.

1.1.1 Installation

Enter the `demo` directory, then create and activate the virtual environments:

```
$ cd demo
$ mkdir -p venv
$ python3 -m venv venv
$ source venv/bin/activate
```

Install Django uWSGI taskmanager:

```
(venv) $ pip install django-uwsgi-taskmanager
```

Install uWSGI (if you use the uWSGI binary from your OS, you can skip this step):

```
(venv) $ pip install uwsgi
```

Collect all static files:

```
(venv) $ python manage.py collectstatic
```

Create all the tables:

```
(venv) $ python manage.py migrate
```

Collect all commands[1]:

```
(venv) $ python manage.py collectcommands --excludecore
```

Create a super user to login to the admin interface:

```
(venv) $ python manage.py createsuperuser
```

Start the project with uWSGI:

```
(venv) $ uwsgi --ini uwsgi.ini
```

[1] *excludecore* ensures that core django tasks are not fetched.

1.1.2 Usage

Visit http://127.0.0.1:8000/admin/[2] and login with the credentials set in the `createsuperuser` task.

Add and launch a task

Proceed as indicated in the video, to create a test task and launch it.

Please note that the video refers to an older release and the UI interface may have changes slightly. The sense of the operations still are perfectly valid.

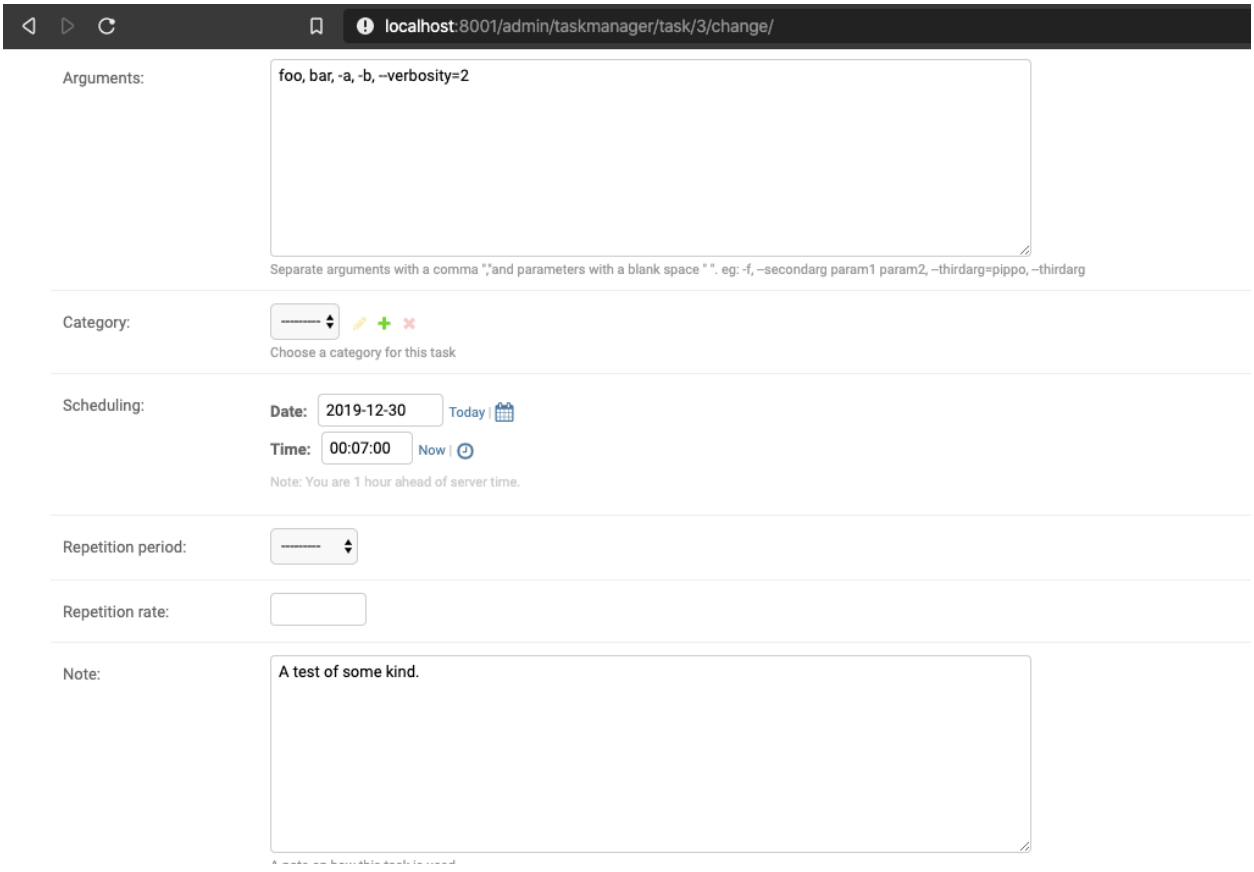
Follow task execution in the lov-viewer window

From version 2.2.0, after the task has been launched, the link to log messages brings to the `log-viewer` windows, where log messages can be seen, updating in almost-real-time, and filtered or searched.

The following video shows a sample, using the `test_livelogging_command` task that generates info, debug, warnings and errors messages.

Scheduling

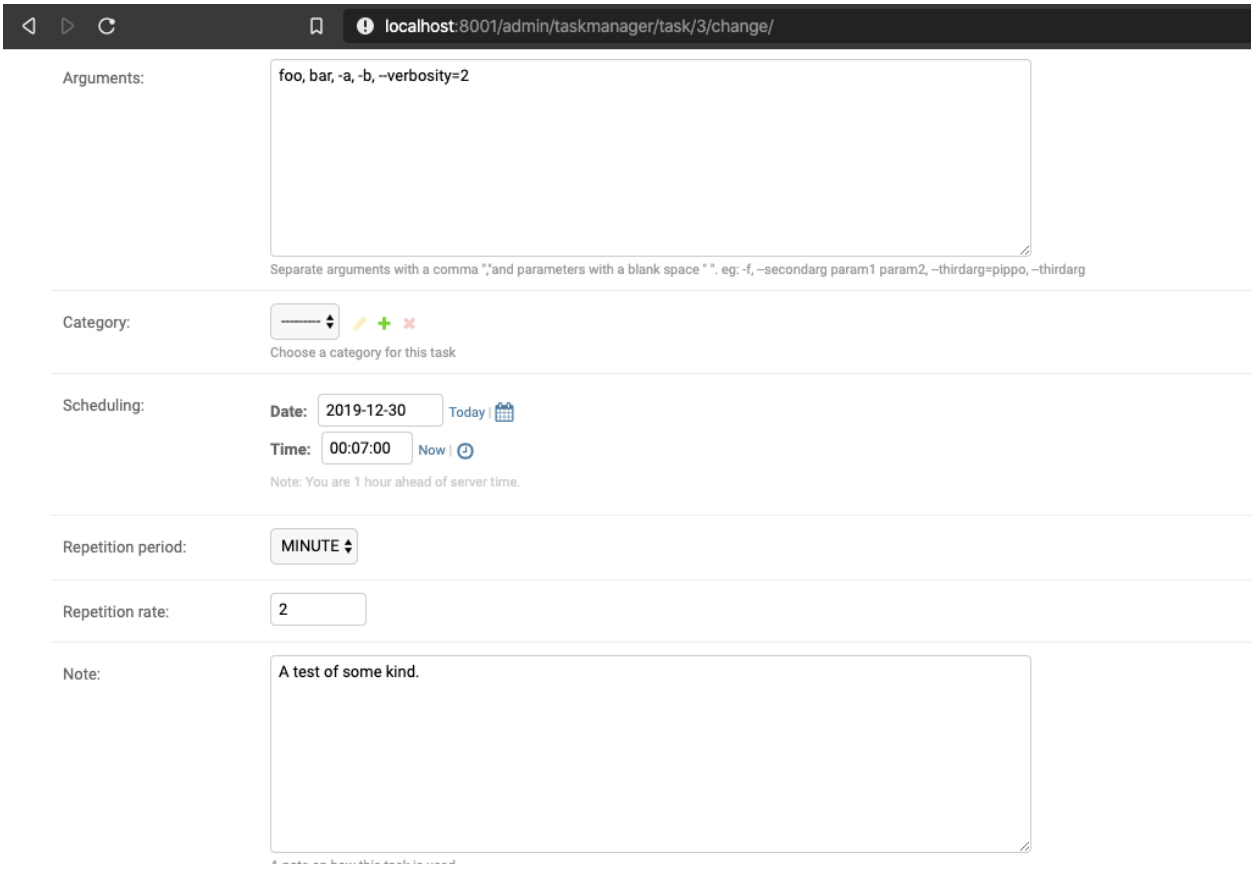
To **schedule** a task and have it starts at a given time, use the **Scheduling** fields:



[2] use the port specified in `uwsgi.ini` (defaults to 8000).

Periodicity

To have a task run **repeatedly**, set both the sheduling fields to a date in the future and the **Repetition rate** and **Repetition period** fields to the desired quantities.

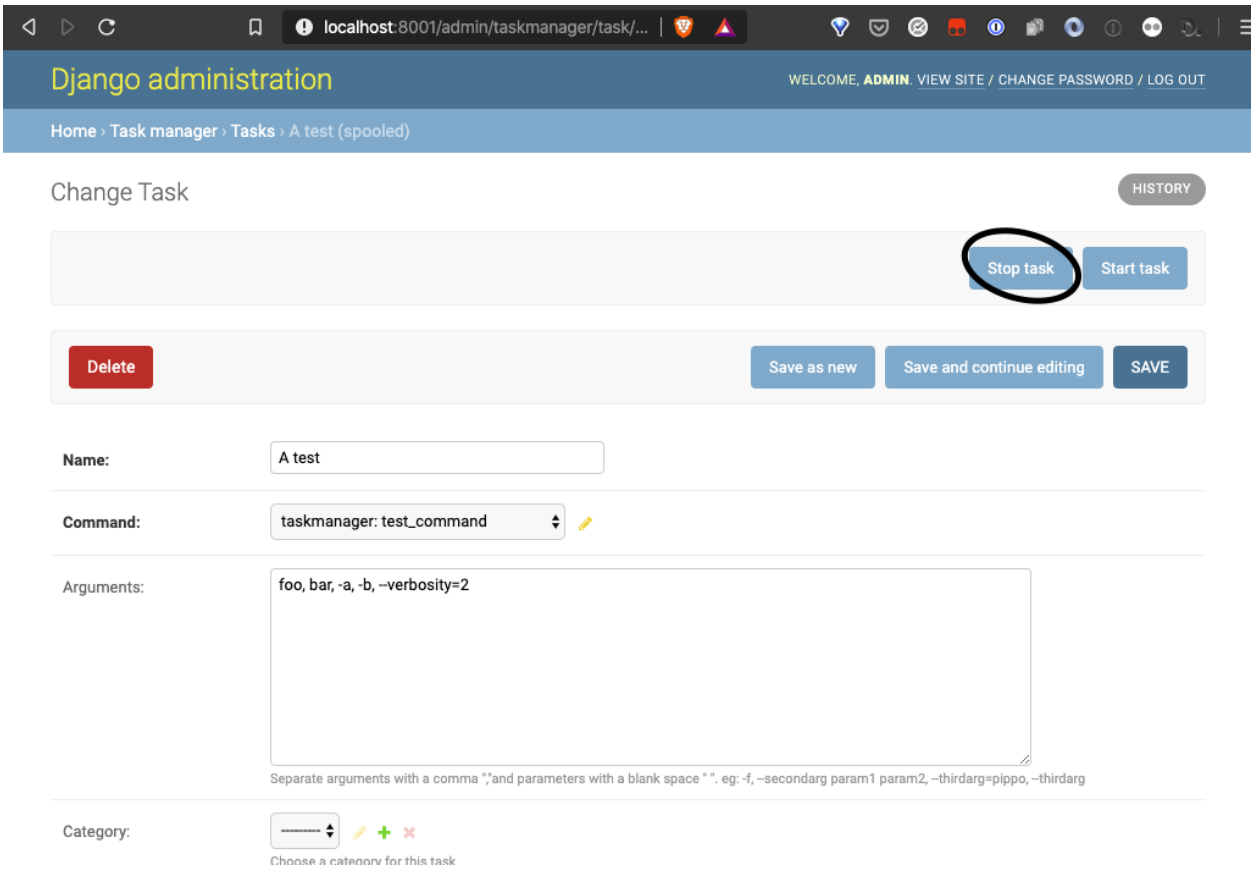


Note: Please observe the following events in order to verify that the tasks are executed (refresh the page):

- the **Last datetime** and **Next** read only fields change in time
- new reports are generated and shown in the Reposts section (only the last five are kept)
- the *uwsgi* task logs in the console show the scheduler executing the process at the right moments

Stop

Finally, to stop a running task, press the **Stop task** button and check that the executions stop.



CHAPTER

TWO

HOW-TO GUIDES

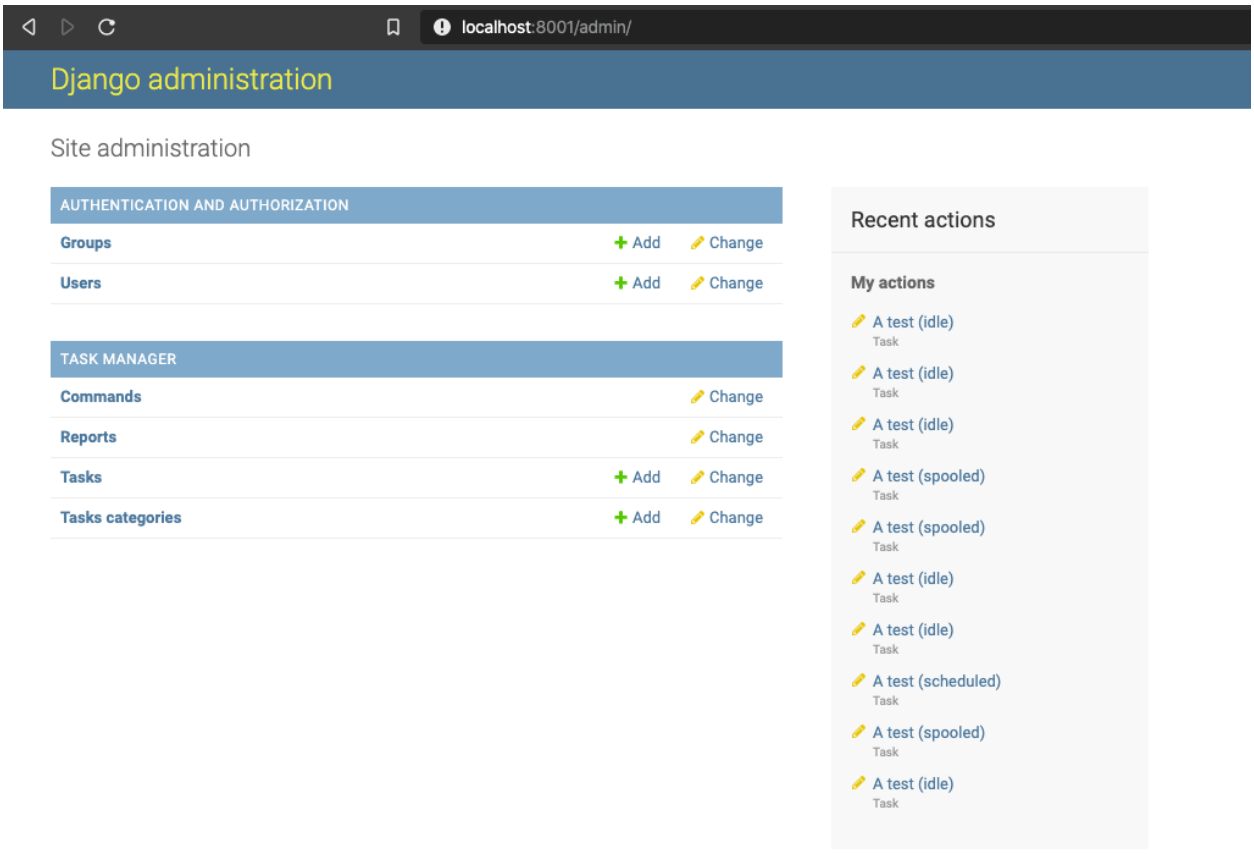
2.1 How to manage tasks in the django admin site

This documentation is for **users** that want to manage tasks within the django admin site.

It is supposed that the users know the basic usage of a django admin interface, so CRUD operations will not be descibed here.

Once you log into the admin site of your app, you'll find a **Task manager** section, where you can manage the tasks.

In the django admin site, a **Task manager** section will appear, containing the app's views.

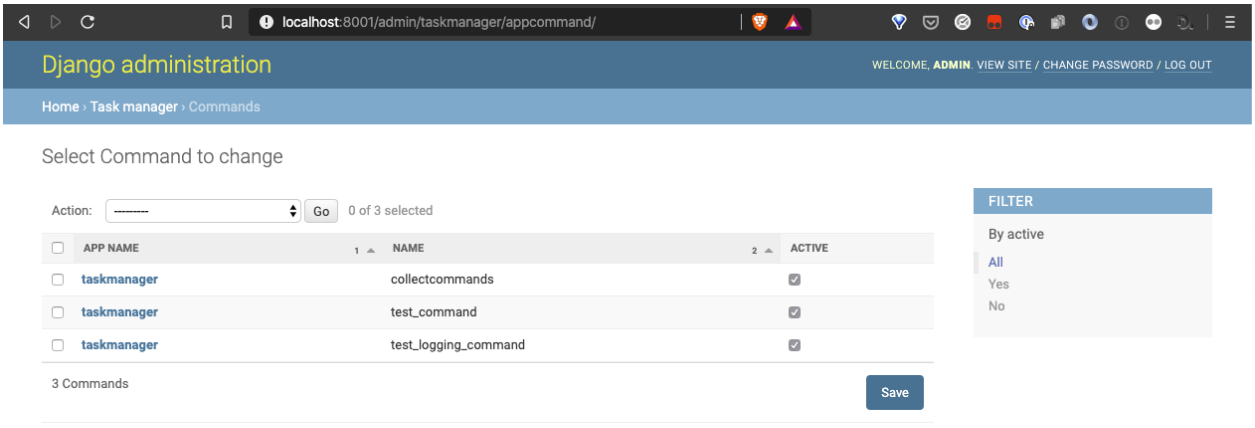


2.1.1 Commands

The commands to use in tasks must be collected from the hosting project's apps, among the defined management tasks, in order to be available as *launchable* commands.

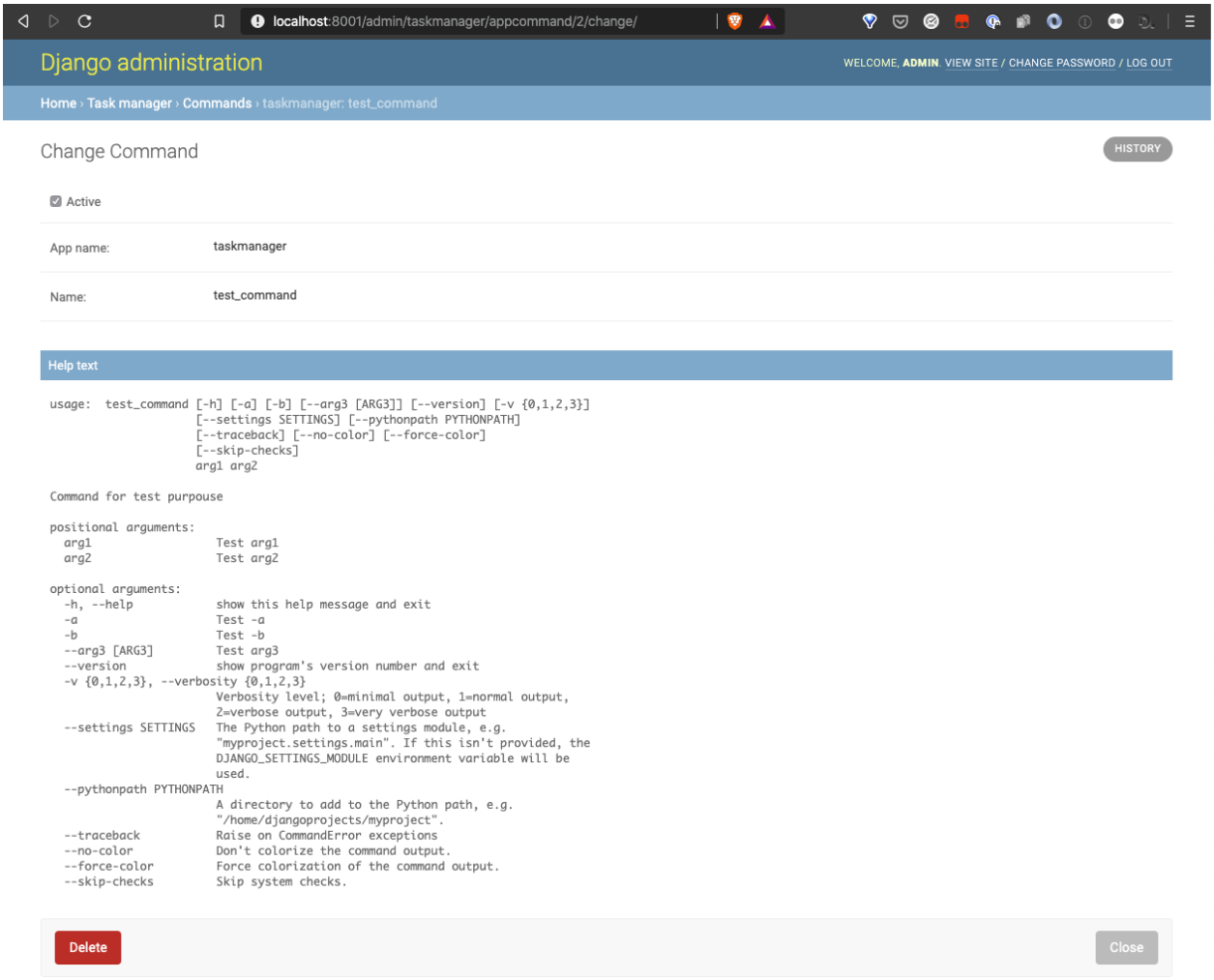
This can be done through the `collectcommands` management task[1]:

```
python manage.py collect_commands --excludecore -v2
```



The complete command's syntax is visible in the command details page (click on the app name in the row of the command).

[1] *excludecore* ensures that core django tasks are not fetched.



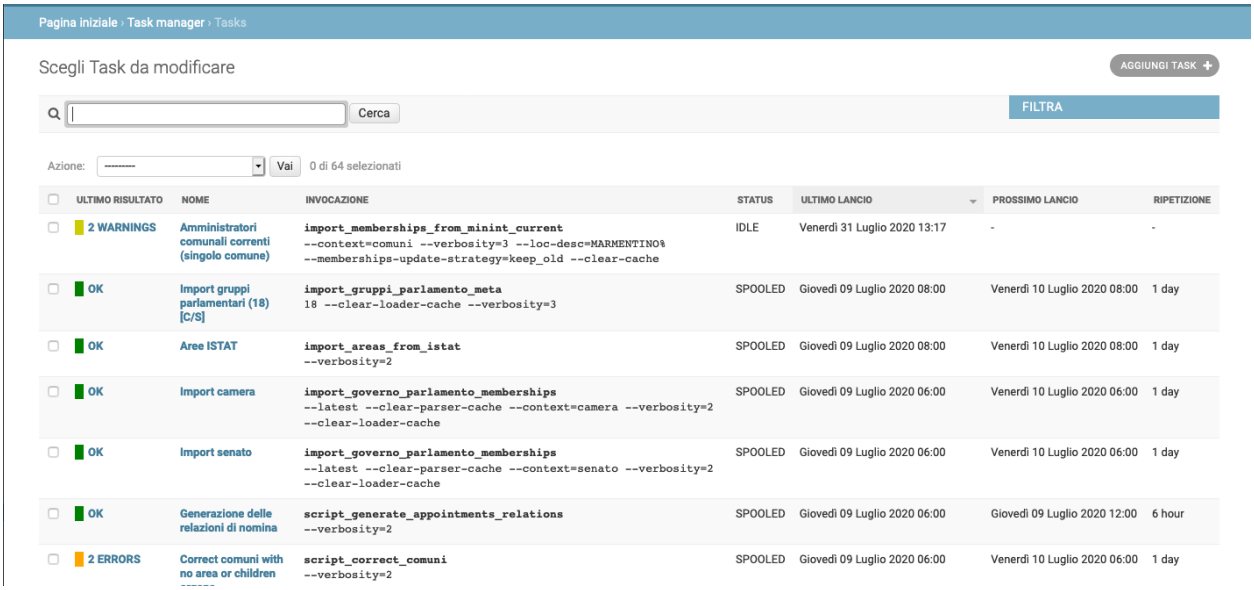
Commands can be deleted. This means that in order to create tasks out of them you will need to use the `collectcommands` task again.

Only commands checked with the `active` flag will be available to generate tasks. So the best option to remove a command and not allow users to geneate tasks out of it is to set its `active` status to false.

Note: It is possible to generate a task starting from the `collectcommands` command, so that the collection of available commands can be launched through the django-uwsgi-taskmanager, too.

2.1.2 Tasks

`Tasks` is the main admin view, where all the action happens. Tasks can be listed, filtered, searched, created, modified and removed using the standard CRUD processes available in django-admin.



Actions are available to have a task start or stop, both in the *list view* and in the *detail view*.

Tasks are sorted, by default, by the latest launch time. This way the most used tasks are shown first, avoiding to clutter the list with unused tasks. Other sort criterion may be chosen by clicking on the column headers, as usual.

Tasks last results are shown both with a color code and with a verbose indication of the number of errors/warnings, if any are there. A task with warnings and errors (yellow and orange color codes), may be perfectly ok, as many times the errors may indicate some problems in the data source. A failed task (red code) requires immediate intervention, as it indicates some missing code or logic in the task itself.

Clicking on the last result status opens a new tab with the log messages for that particular execution.

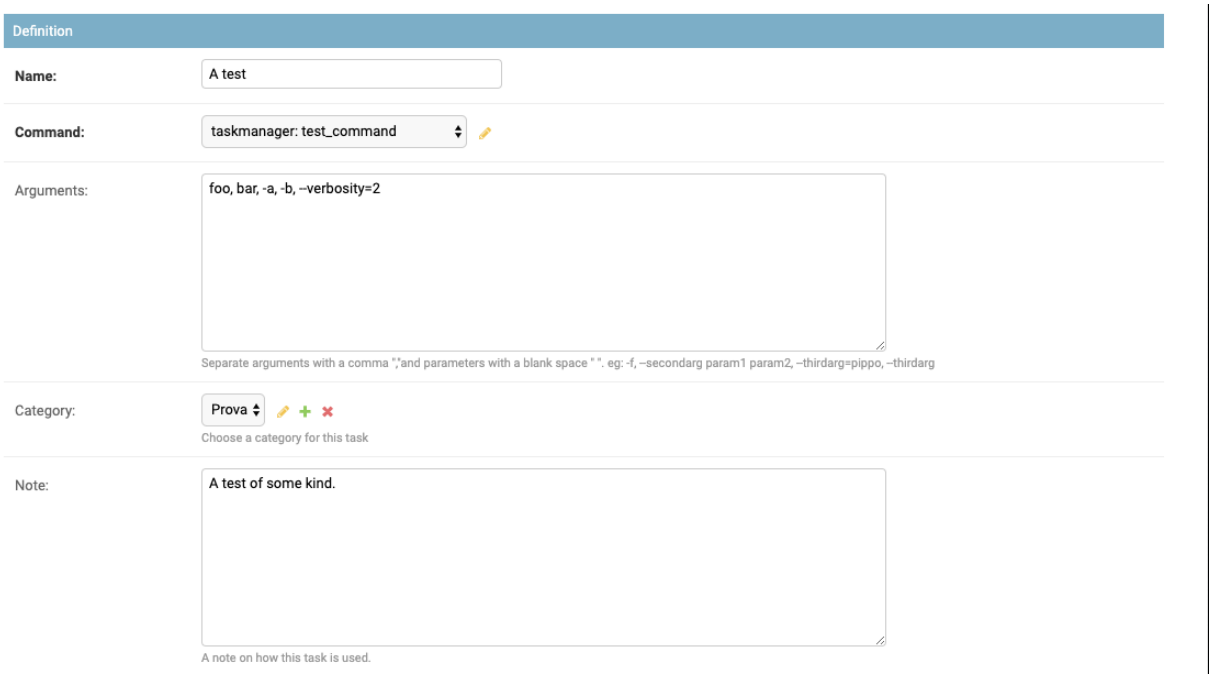
Hovering over the name of the task shows the descriptive note, if inserted by the task authors. This may describe aspects of that task instance and peculiarities of the arguments to pass.

2.1.3 Task structure

A task has four main sections:

- **Definition**: name, command, arguments, category and note;
- **Scheduling**: time of start and repetition period and rate;
- **Last execution**: spooler id, status, last execution datetime, last result, next execution, n. of errors and warnings;
- **Reports**: Each task's execution generates a **Report**. Only the last 5 reports are kept and shown in the Task's detail view.

2.1.4 Defining a task



Fields in the **definition** section:

- **name**: name a task, use unique names with prefixes, to identify tasks visually

---

Note: It is important to understand that a command can be used multiple times in various tasks, with different arguments. Use different **names** and specify differences verbosely in the **note** field to let other users make the right choices on which task to use.

- **command**: select the command from the collected ones, in the command popup list;
- **arguments**: the command's arguments in a special syntax:

Note: Single arguments should be separated by a *comma* (","), while multiple values in a single argument should be separated by a blank space,

eg: `-f, --secondarg param1 param2, --thirdarg=pippo, --thirdarg`

- **category**: select from an existing one, or add a new one

- **note**: a descriptive note on how the command or its arguments are used

2.1.5 Task categories

In order to ease the search of tasks when they start to grow in numbers, a category can be assigned to each one. The tasks list can then be filtered by category.

Note: Use simple, short words as categories and try to have less than 10 categories in all, in order not to confuse other users.

2.1.6 Scheduling a task

Scheduling is performed through the following fields:

- **scheduling**: date and time, sets the moment in time when the task is going to be launched for the first time.
- **repetition period**: select one among *minute*, *hour*, *day*, *month*
- **repetition rate**: set an integer

To **schedule a task to start in the future only once**: set the scheduling field to a point in time in the future and press the start button.

To **schedule a task to start in the future and run periodically**: set **both** the scheduling field and the repetition fields, then press the start button.

To **stop a scheduled start**: press the stop button.

2.1.7 Reading the task's last execution status

The fields in this section are *read-only* and are meant to show information on the task's lat execution.

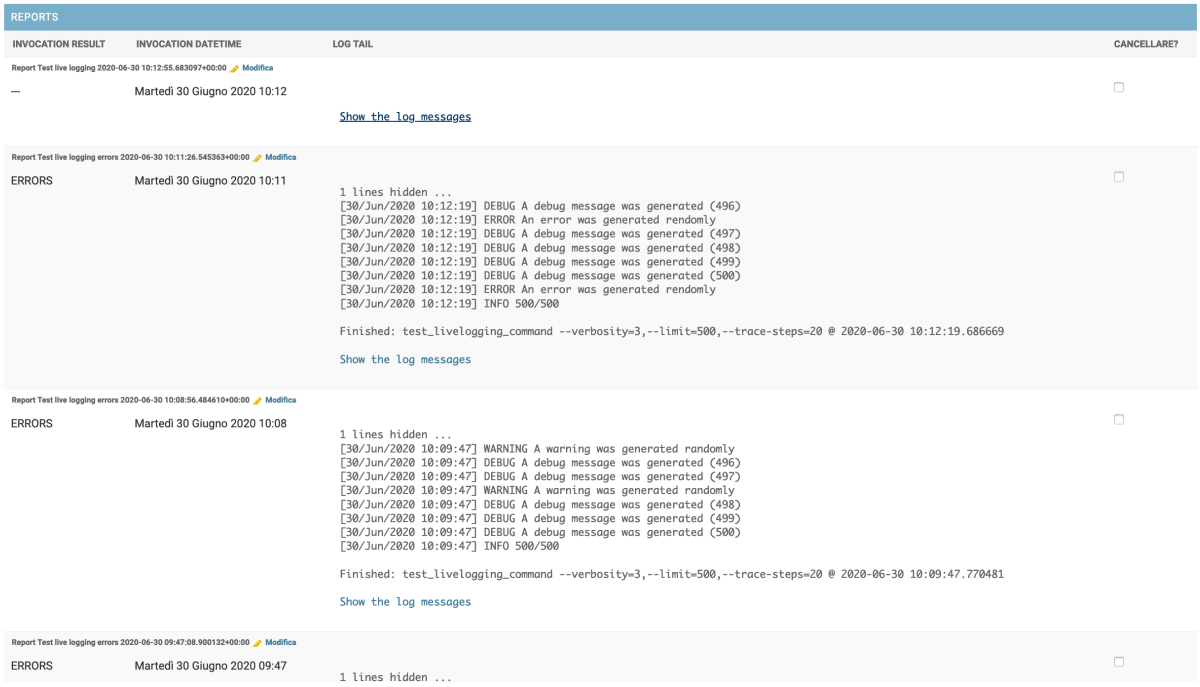
- **spooled at**: the complete path to the file in the spooler, can be useful when debugging errors, but it's an *internal* information and should not be needed by standard users
- **status**: can be one of:
  - `IDLE`: the task never started or was stopped,
  - `STARTED`: the task is currently running,
  - `SCHEDULED`: the task is going to start for the first time in the future,
  - `SPOOLED`: the task has been put in the spooler and is going to start again in the future
- **last datetime**: the last execution date and time
- **last result**: last execution result
  - `OK`: correctly executed, with no warnings, nor errors
  - `WARNINGS`: correctly executed, but contains warnings, see the report
  - `ERRORS`: correctly executed, but contains errors, see the report
  - `FAILED`: there was an error while execution, see the report
- **errors**: the number of errors detected in the last execution
- **warnings**: the number of warnings detected in the last execution

---

Note: Consider that before starting for the first time, the task is being put in the spooler, so whenever checking the status of a task, it can happen that its status shows `SPOOLED`, and after a few moments, refreshing the page, it will show `STARTED`.

This is perfectly normal.

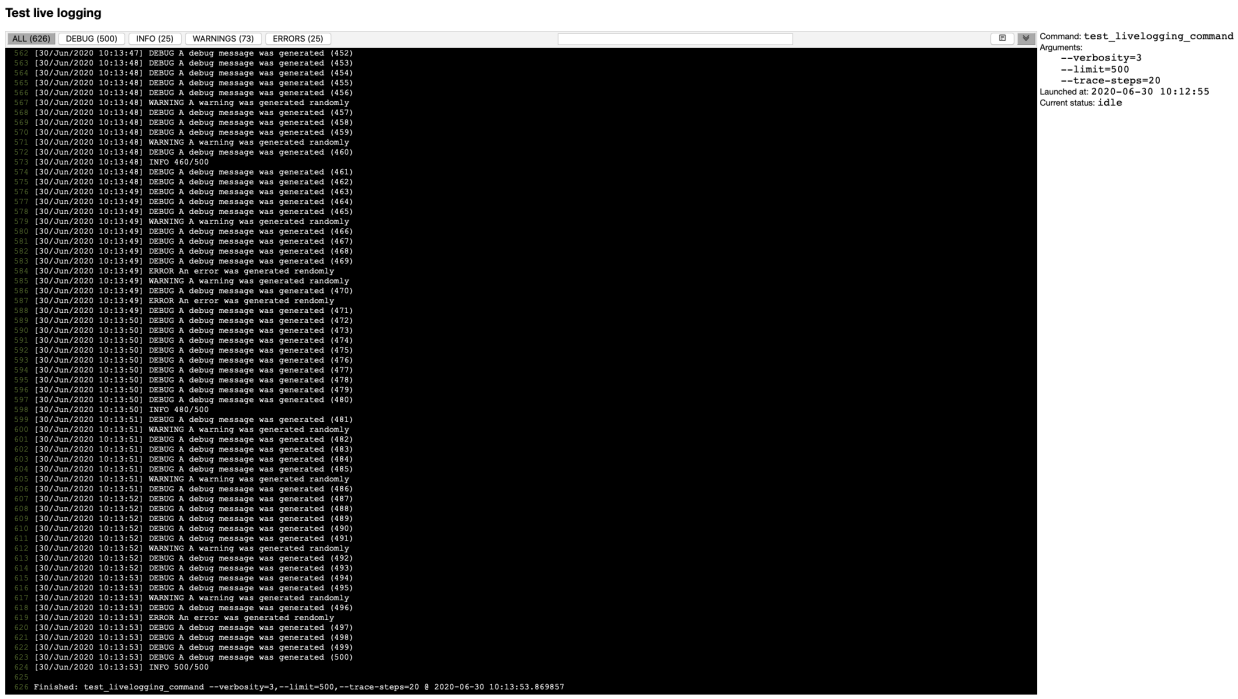
2.1.8 Reading the task's reports



Once a task is finished, a report is generated and added to the **reports** section. Only the last 5 reports are left available to the users, in order to save space.

Each report contains the **result** and **invocation datetime** fields, along with the *tail* of the last 10 lines logged during execution.

Clicking on the *show the log messages* link, a new page cotaining the log messages is opened.



If the task is still executing, the page will be refreshed, in order for the new messages to be added to the page.

On top of the page there is a **toolbar**, divided into three sections:

- the **levels buttons** (`ALL`, `DEBUG`, `INFO`, `WARNING`, `ERROR`) act as filters and clicking on one of them only the messages of that type will be listed; the numbers appearing by each button indicate how many messages of that type have been produced; buttons only appear when some message of that type is added to the log file;
- the **search field** allows to filter messages by a string: only messages containing the string are listed; clicking on the 'x' button by the search field will reset all filters and is equivalent to pressing the `ALL` button;
- as for the commands on the right side of the toolbar:
  - the **raw logs** button allows to open up a new page with the log files in raw text format
  - the **sticky mode** button disable or enable the scrolling of the messages display to the bottom; this can be used in order to disable following the logging messages and concentrating on some research;

Note: The complete list of log messages is rendered on a single page. This can be problematic whenever the list is really long, as rendering times may be long too. The only solution that comes to mind is to implement tasks that doesn't log too many rows.

2.2 How to install django-uwsgi-taskmanager in an existing project

This documentation is for **developers**, that want to add this application to their django project.

Note: As a pre-requisite, the project should already be served through **uWSGI**.

---

0. Install the app with *pip*:

   via **PyPI**:

   ```
   pip install django-uwsgi-taskmanager
   ```

   or via **GitHub**:

   ```
   pip install git+https://github.com/openpolis/django-uwsgi-taskmanager.git
   ```

1. Add "taskmanager" to your *INSTALLED_APPS* setting like this:

   ```
   INSTALLED_APPS = [
       "django.contrib.admin",
       # ...,
       "taskmanager",
   ]
   ```

2. Run `python manage.py migrate` to create the taskmanager tables.

3. Run `collectcommands` management task to create taskmanager commands[1]:

   ```
   python manage.py collectcommands --excludecore
   ```

4. Include the taskmanager URLConf in your project `urls.py` *(optional)*[2]:

[1] *excludecore* ensures that core django tasks are not fetched.
[2] the `/taskmanager/logviewer` view is added to show the complete logs message.

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path("admin/", admin.site.urls),
    path("taskmanager/", include("taskmanager.urls")),
]
```

5. Set parameters in your settings file as below *(optional)*:

```
UWSGI_TASKMANAGER_N_LINES_IN_REPORT_INLINE = 10
UWSGI_TASKMANAGER_N_REPORTS_INLINE = 3
UWSGI_TASKMANAGER_SHOW_LOGVIEWER_LINK = True
UWSGI_TASKMANAGER_USE_FILTER_COLLAPSE = True
UWSGI_TASKMANAGER_SAVE_LOGFILE = False
```

6. Configure the notifications, following the *How to enable notifications* guide *(optional)*.

2.3 How to add django-uwsgi-taskmanager to a dockerized stack

This documentation is for **developers**, that want to add this application to an existing django application, within a *dockerized* stack.

The following `docker-compose.yml` shows parts of a stack where an API service is provided. Note the **web.command** value, invoking the uwsgi server in the container.

That invocation generates 4 processes able to fullfill the http(s) request-response cycle, and 2 processes checking and running processess added to the spooler.

The `/var/lib/uwsgi` directory is defined as a persistent volume and contains the spooler files used by the app. This ensures that the processes keep being executed at scheduled times even after a container's restart.

Note: The yml file is partial and is only shown for illustration purposes.

```
version: "3.5"

services:
  web:
    container_name: service_web
    restart: always
    image: acme/project/service:latest
    expose:
      - "8000"
    links:
      - postgres:postgres
    environment:
      - DATABASE_URL=postgis://${POSTGRES_USER}:${POSTGRES_PASS}@postgres/${POSTGRES_DB}
      - DEBUG=${DEBUG}
      ...
      - UWSGI_TASKMANAGERN_OTIFICATIONS_SLACK_TOKEN=${UWSGI_TASKMANAGER_NOTIFICATIONS_
↪SLACK_TOKEN}
```

(continues on next page)

(continued from previous page)

```
      - UWSGI_TASKMANAGER_NOTIFICATIONS_SLACK_CHANNELS=${UWSGI_TASKMANAGER_NOTIFICATIONS_
↪SLACK_CHANNELS}
      - CI_COMMIT_SHA=${CI_COMMIT_SHA}
    volumes:
      - public:/app/public
      - uwsgi_spooler:/var/lib/uwsgi
      - weblogs:/var/log
    command: /usr/local/bin/uwsgi --socket=:8000 --master \
        --env DJANGO_SETTINGS_MODULE=config.settings
        --pythonpath=/app --module=config.wsgi --callable=application \
        --processes=4 --spooler=/var/lib/uwsgi --spooler-processes=2

  ...

volumes:
  public:
    name: service_public
  uwsgi_spooler:
    name: service_uwsgi_spooler
  weblogs:
    name: service_weblogs

networks:
  default:
    external:
      name: webproxy
```

2.4 How to enable notifications

The `notifications system` enables `django-uwsgi-taskmanager` to send custom notifications at the end of tasks execution. Tasks may be sent according to the specified `level` parameter in the handler:

- `failed`: whenever failures are trapped during the execution,
- `errors` or `warnings`: when the execution terminates correctly, but errors or warnings are detected,
- `ok`: when everything runs smoothly, just to know.

From release 2.1.0, the notifications system has been refactored into a pluggable system. The subsystems ready to be plugged are: **Slack** and **email**. Development of a custom subsystem is possible, and a small developer guide is present in the last paragraph of this section.

To enable the Slack notifications subsystem, you have to first install the required packages, which are not included by default. To do that, just:

```
pip install django-uwsgi-taskmanager[notifications]
```

This will install the `django-uwsgi-taskmanager` package from PyPI, including the optional slackclient dependency required to make Slack notifications work.

Email notifications are instead handled using Django django.core.mail module, so no further dependencies are needed and they should work out of the box, given you have at least one email backend properly configured.

Then, you have to configure the `UWSGI_TASKMANAGER_NOTIFICATION_HANDLERS` setting variable as a dictionary with the chosen handlers.

For example, to set up the slack notification handler:

```
UWSGI_TASKMANAGER_NOTIFICATION_HANDLERS = {
    "slack": {
        "class": "taskmanager.notifications.SlackNotificationHandler",
        "level": "errors",
        "token": env("UWSGI_TASKMANAGER_NOTIFICATIONS_SLACK_TOKEN", default=""),
        "channel": env("UWSGI_TASKMANAGER_NOTIFICATIONS_SLACK_CHANNELS", default=""),
    },
}
```

with the following env variables set:

- `UWSGI_TASKMANAGER_NOTIFICATIONS_SLACK_TOKEN`, the Slack token as string.
- `UWSGI_TASKMANAGER_NOTIFICATIONS_SLACK_CHANNELS`, a list of strings representing the names or ids of the channels which will receive the notifications.

For the email notification handler:

```
UWSGI_TASKMANAGER_NOTIFICATION_HANDLERS = {
    "mail": {
        "class": "taskmanager.notifications.MailNotificationHandler",
        "level": "errors",
        "from_email": env("UWSGI_TASKMANAGER_NOTIFICATIONS_EMAIL_FROM", default=""),
        "recipients": env("UWSGI_TASKMANAGER_NOTIFICATIONS_EMAIL_RECIPIENTS", default="
↪"),
    },
}
```

with the following env variables:

- `UWSGI_TASKMANAGER_NOTIFICATIONS_EMAIL_FROM`, the "from address" you want your outgoing notification emails to use.
- `UWSGI_TASKMANAGER_NOTIFICATIONS_EMAIL_RECIPIENTS`, a list of strings representing the recipients of the notifications.

More than one handler can be added. Notifications will be sent to all parties defined.

2.4.1 Developing a custom handler

The basic notification handler is defined in `taskmanager.notifications.NotificationHandler`, as an abstract class. All handlers subclass this one.

Handlers class can be created anywhere in the python import path. If found, they will be imported by the taskmanager application, during the app startup, and registered as active handler.

In order to setup the handler in the settings, a custom dictionary must be created, just like the two examples above. The dictionary needs to be created, with the `class` and `level` keys, at least.

The `class` key will be popped out of the dictionary and used to instantiate the handler, with the others keys passed as arguments.

The `emit_notifications` method of the `Report` class will call all registered handlers and emit the notifications. It is called at the end of `taskmanager.tasks.exec_command_task`.

Dependencies, should they be needed, must be installed separately.

Feel free to create a pull request if you want to add a notification handler directly in the package.

2.5 How to contribute to the project

2.5.1 Documentation

This documentation is written using sphinx. It follows the guidelines on writing technical documentation set by Daniele Procida, and is contained in the `docs` directory of the project.

In order to contribute to the documentation, the following packages should be added to the virtualenv on the developer machine:

```
sphinx
sphinx-django-command
sphinx-rtd-theme
sphinx-autobuild
pyembed-rst
```

Then, from inside the `docs` directory:

```
make clean
make build html
```

The `makefile` has been customised with respect to the original one generated by the `sphinx-quickstart` script, and it contains a `livehtml` target, that allows to rebuild the html output each time the rst source files are changed and saved.

```
make livehtml
```

2.5.2 Development

Source code is available on https://github.com/openpolis/django-uwsgi-taskmanager.

Tests can be launched with

```
python demo/manage.py test
```

The source code is tested for syntax and format using black.

2.6 How to debug tasks

Since the uwsgi uses the spooler processes, debugging the task execution in these process requires a hack through remote debugging.

The following procedure works in pyCharm IDE.

1. `pip install pydevd-pycharm==191.6605.12` (versions must be upgraded, see preferences/about)
2. open a shell in the virtual environment and prepare this command with the following set of arguments:

```
uwsgi --http=:8000 --master \
  --chdir=/Users/gu/Workspace/django-uwsgi-taskmanager/demo \
  --static-map /static=./static \
  --module=demo.wsgi --callable=application \
  --pythonpath=/Users/gu/Workspace/django-uwsgi-taskmanager/demo \
  --processes=2 \
  --spooler=./spooler --spooler-processes=1
```

3. define a python remote debug configuration on pycharm, using localhost:4444 as host:port
4. add this snippet of code right before the point you want the execution to break

```
import pydevd
pydevd.settrace('localhost', port=4444, stdoutToServer=True,␣
↪stderrToServer=True)
```

   use `wsgi.py` to debug the request/response processes and `taskmanager/models.py` or `taskmanager/tasks.py`, to debug the command execution

5. add breakpoints
6. launch the **uwsgi** command in terminal
7. launch the debugger in pycharm
8. navigate the admin UI, create and launch the task
9. debug!

When no debugger is activated, this can be used to test the uwsgi-spooler in a local development environment. Just remove the code snippets and launch the uwsgi command from the terminal.

You'll be able to manage tasks and execute the commands using the uwsgi spooler processes.

CHAPTER

THREE

REFERENCE

Classes and functions are documented here automatically, extracting information from the comments in the source code.

3.1 taskmanager.models

Define Django models for the taskmanager app.

Classes

`AppCommand`(*args, **kwargs)	An application command representation.
`Report`(*args, **kwargs)	A report of a task execution with log.
`Task`(*args, **kwargs)	A command related task.
`TaskCategory`(*args, **kwargs)	A task category, used to group tasks when numbers go up.

3.2 taskmanager.management.base

Base classes for writing management commands.

Classes

`LoggingBaseCommand`([stdout, stderr, ...])	A subclass of BaseCommand that logs messages using the django logging system.

3.3 taskmanager.logging

Define utils for logging.

Classes

NoTerminatorStreamHandler([stream])	A stream handler.

3.4 taskmanager.tasks

Define uWSGI exec command tasks for the taskmanager app.

taskmanager.tasks.**exec_command_task**(*curr_task*, **args*, ***kwargs*)

Execute the command of a Task.

Parameters

- **curr_task** (*Task*) – instance of the task to execute
- **args** – unnamed arguments
- **kwargs** – named arguments

CHAPTER

FOUR

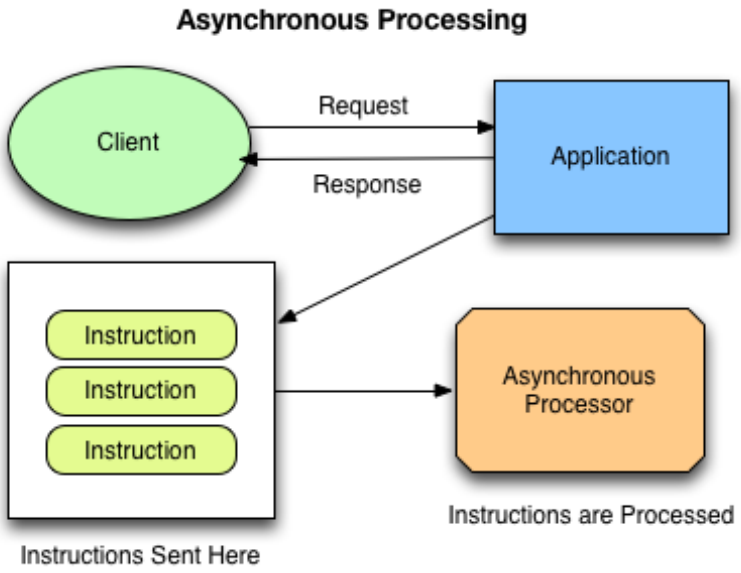
DISCUSSIONS

Although Celery is the most used solution to execute distributed asynchronous tasks in python and django-channels is the new hype, this project offers a solution based on uWSGI spooler, which requires no additional components, is particularly easy to setup, and has a straight learning curve.

4.1 Pre-requisites

uWSGI is normally used ad an application server, to accept requests, transfer control to the python web application using the wsgi protocol, and send the response back.

If configured as shown in this documentation, it can spawn some processes to handle asynchronous tasks, reading the queue from a specified *spool* directory.



The following snippet of code starts a uWSGI server able to process both HTTP requests and asynchronous tasks[1]:

```
uwsgi --check-static=./static --http=:8000 --master \
  --module=wsgi --callable=application \
  --pythonpath=./ \
  --processes=4 --spooler=./uwsgi-spooler --spooler-processes=2
```

[1] Setting up uWSGI in production usually involves some sort of frontend proxy, but this is not the place to discuss it.

- 4 processes will accept HTTP requests and send HTTP responses;
- 2 processes will check the spooler and execute tasks there;
- 1 master process will superintend all other processes.
- the `./uwsgi-spooler` path is the physical location on disk where the spooled tasks will be kept

PYTHON MODULE INDEX

t

INDEX

M

T