Django eztaskmanager

Release 0.1.0

G. Celata

Apr 09, 2024

CONTENTS

django-eztaskmanager is a Django application designed to initiate standard Django management tasks *asynchronously*. This is done through a conventional Django administrative interface, using either RQ or, in the near future, Celery.

django-eztaskmanager is both an evolution and an upgrade from our previous django-uwsgi-taskmanager; it comes with these key **features**:

- usage of standard Django management commands as task *templates*;
- capability to import existing management commands through a meta-management command;
- manual starting and stopping of tasks via an administrative interface;
- ability to schedule singular and periodic tasks using the Django admin system;
- compatibility with RQ (rq + rq-scheduler) or Celery (celery + celery-beat) for queue management;
- verification or download of generated reports/logs;
- live log streaming display, with error and warning filters for task debugging;
- notification capabilities via email or Slack on task completion or failure.

Note: Right now, **django-eztaskmanager** is built to play nice with RQ (Redis Queue). It does the job, and does it well. But I know some of you out there swear by Celery, and I hear you. It's on my radar and I'm knee-deep in code working to get it integrated.

So, keep an eye out for updates!

CHAPTER

ONE

GET STARTED

Following the demo tutorial, it will be possible to install, configure and use **eztaskmanager** for a simple demo django project running in developer mode, working with Redis Queue.

Further knowledge, with detailed admin description, use in production deployments, enabling existing notifications plugins or developing a custom one, can be found in the *How-to guides*.

1.1 The demo tutorial (RQ)

Clone the project from github onto your hard disk:

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

There is a basic Django project under the `demoproject` directory, set to use `eztaskmanager`.

```
demoproject/
├── demoproject/
│   ├── __init__.py
│   └── asgi.py
│   ├── settings.py
│   ├── test_settings.py
│   ├── urls.py
│   └── wsgi.py
├── manage.py
├── static/
└── docker-compose-local.yml
```

1.1.1 Installation and setup

As a **pre-requisite**, a Redis server should be up and running on the default port 6379. Follow the instructions, or if you use docker, just run `docker compose -f docker-compose-local.yml`.

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

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

Install **eztaskmanager**, this will install all needed dependencies (django, redis, django-rq, rq-scheduler,...):

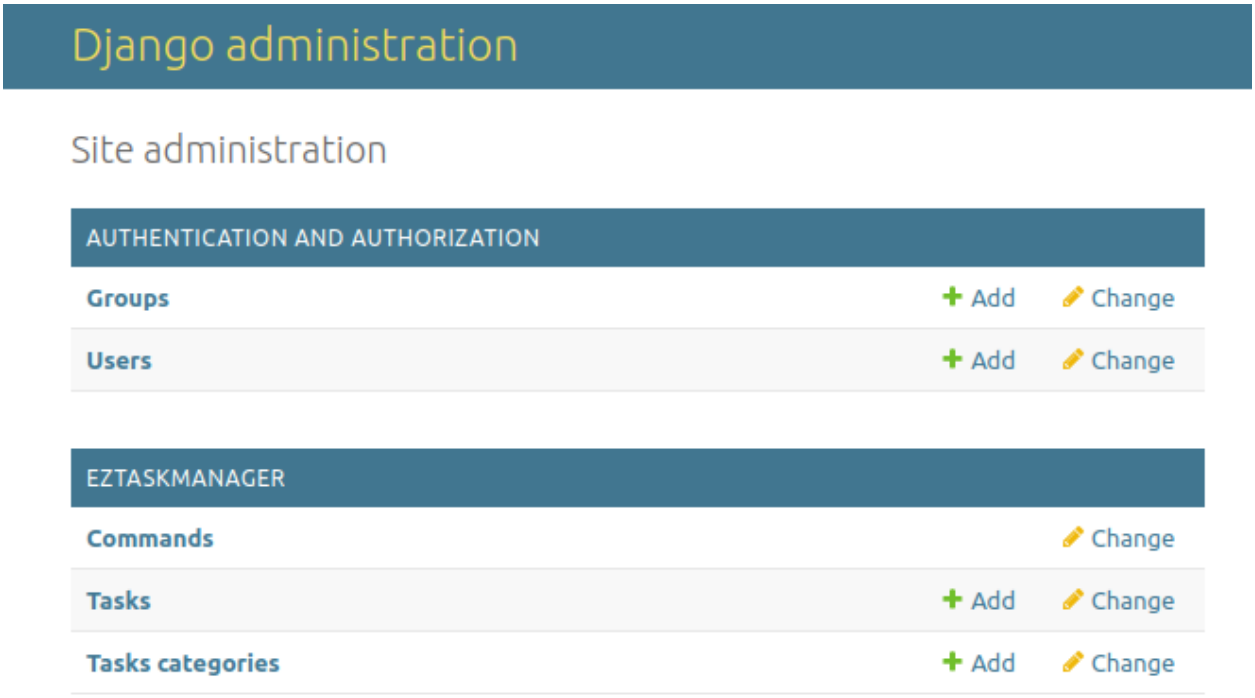
```
(venv) $ pip install django-eztaskmanager
```

Then execute this commands to setup the server in development mode, the rq worker and the scheduler:

```
(venv) $ python manage.py migrate  # create tables in the DB (default sqlite will do)
(venv) $ python manage.py createsuperuser # take note of username and password for login
(venv) $ python manage.py collectcommands --excludecore  # collect basic commands from␣
↪the eztaskmanager package
(venv) $ python manage.py runserver  # django app server on port 8000
(venv) $ python manage.py rqworker  # rq worker to execute enqueued tasks
(venv) $ python manage.py rqscheduler --verbosity=3  # rq scheduler to enqueue periodic␣
↪tasks
```

1.1.2 Usage

Visit http://127.0.0.1:8000/admin/ and login with the credentials set in the `createsuperuser` task. a new **EZ-TASKMANAGER** section appears.



3 administration sections are available:

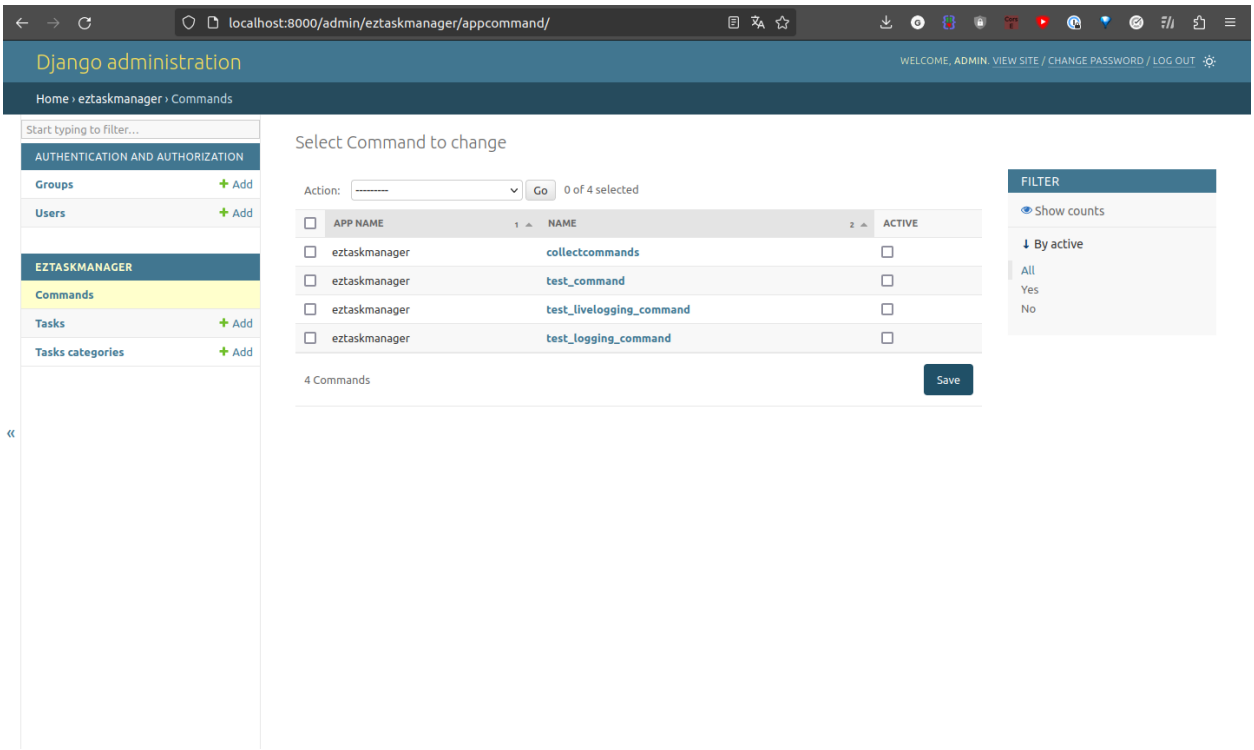
- **Commands**: to see and select which commands are available for tasks creation.
- **Tasks**: to manage tasks, usual CRUD operations, and start and stop
- **Task categories**: to manage task groups

Create and Launch a Task

A task is an invoked management command with specific arguments.

You have already launched and imported commands using `python manage.py collectcommands --excludecore` during the setup phase.

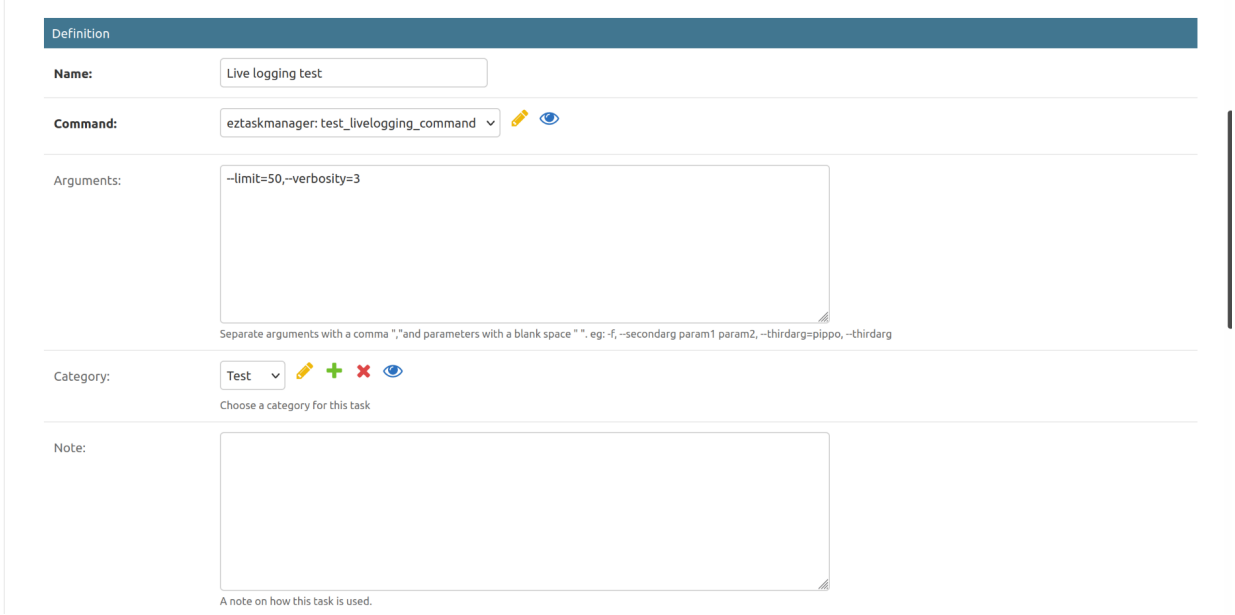
Navigate to the Commands section of the admin interface; here, you will find a list of commands from all your Django applications. In this demo project, only commands within **eztaskmanager** will be available.



Ensure to activate the following commands, as only **active** commands will be available for task creation:

- **test_livelogging_command**
- **collectcommands**

Now, in the Tasks section, click the **Add task** button in the top-right corner.



Provide a name, select the desired command, and enter the arguments as: `--limit=200,--verbosity=3`. Leave all other fields untouched.

Click **Save and continue editing**, and then **Start**.

This will enqueue the task, and it will be executed immediately within the queue manager context (RQ worker).

Note: Django management commands usually extend the *`django.core.management.base.BaseCommand`* and log to the STDERR stream. As such, log messages can be seen in the RQ worker logs.

To have log messages stored within the database for future inspection or live viewing, the management command must extend *`eztaskmanager.services.logger.LoggerEnabledCommand`*.

Monitor Task Execution in Live-viewer Window

Setting the `limit` task option to 200 provides ample time for observation of log messages in the live viewer.

Upon pressing the **Start** button, the task status next to its title will shift to *Started*.

Scroll down to the *Launch Reports* section where a new report will be generated. Click "Show the log messages," which will open a new tab in your browser where log messages will appear as they are generated by the executing task.

Schedule a Task for Future Execution

To **schedule** a task to start at a specific time, set the **Scheduling** fields accordingly:

Execute Periodic Tasks

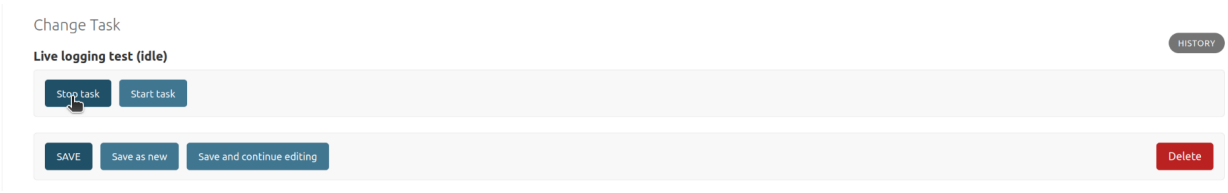
To execute a task **repeatedly**, schedule both fields for a future date and set the **Repetition rate** and **Repetition period** to desired values:

Note: In order to confirm task execution, observe the following (refresh the page as necessary):

- The **Last datetime** and **Next** read-only fields update over time.
- New reports are generated and shown in the Reports section (only the latest five are retained).
- The RQ worker console displays a stream of messages corresponding with the `verbosity` arguments.

Terminate Future Task Executions

To stop a future scheduled task, click the **Stop task** button and verify that executions cease.



CHAPTER

TWO

HOW-TO GUIDES

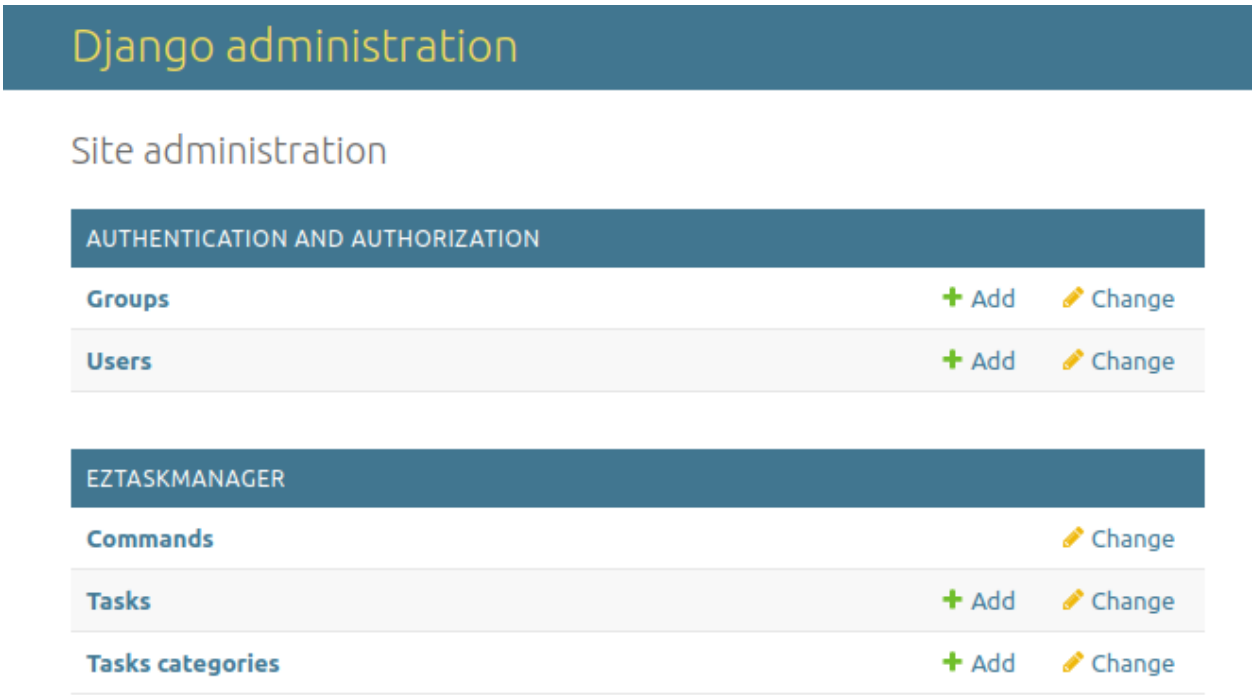
2.1 Managing Tasks in Django Admin Site

This guide is intended for **users** who want to manage tasks using Django's built-in administrative interface.

It is assumed that these users are familiar with the basic functionality of a Django admin interface, therefore CRUD (Create, Read, Update, Delete) operations won't be covered here.

Upon successful login to the admin site of your application, a **Task Manager** section will be visible, providing you the ability to manage your tasks.

Within the Django admin site, the **Task Manager** section will include the application's views.

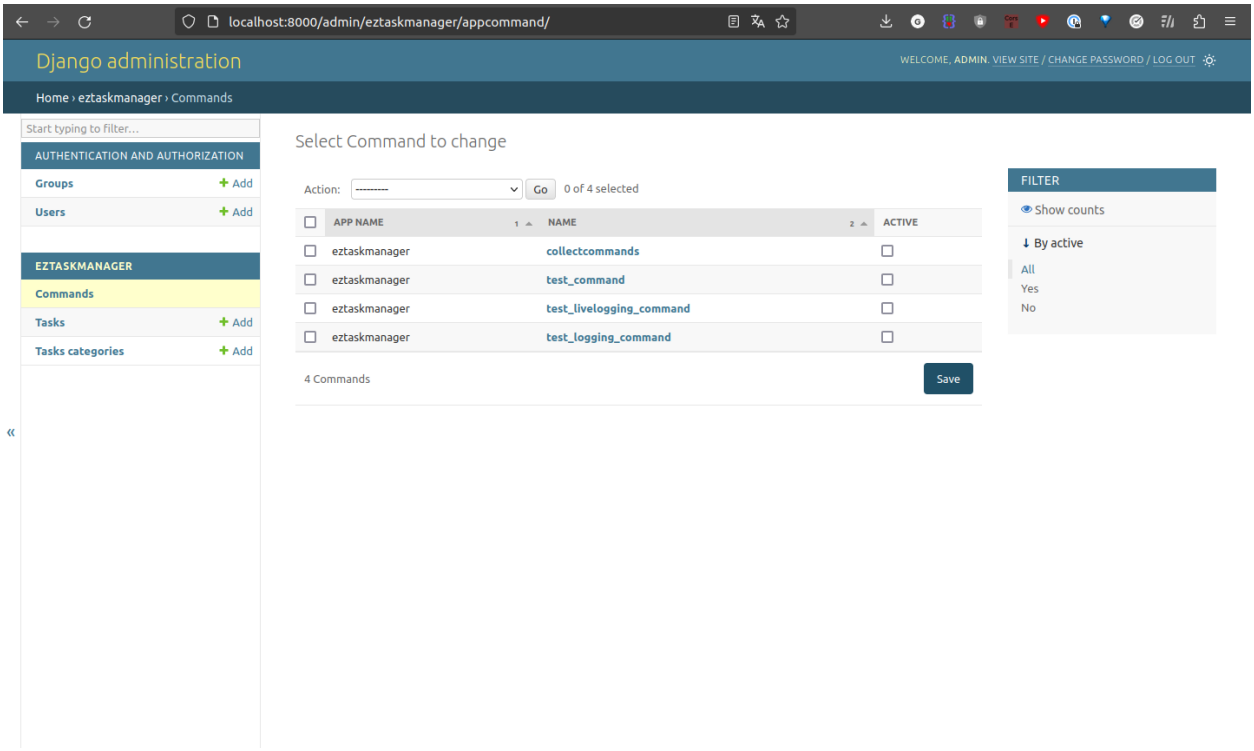


2.1.1 Gathering commands

Commands for tasks should be gleaned from the host project's applications, specifically among those tasks that have been defined for management. This will enable them to be available as *launchable* commands.

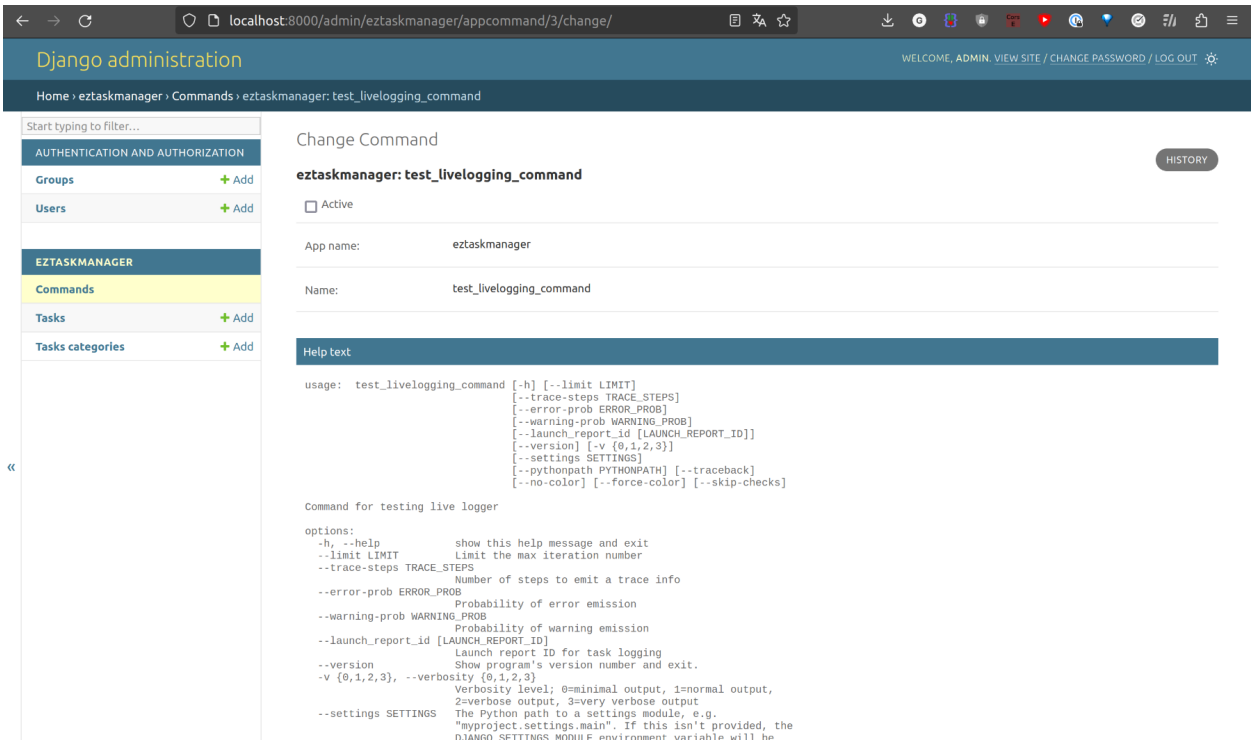
This process can be achieved using the `collectcommands` management task[1].

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



Complete syntax of a command can be found on the command details page, which is accessible by clicking on the application name in the command's row.

[1] The *excludecore* parameter is used to prevent the fetching of core Django tasks.



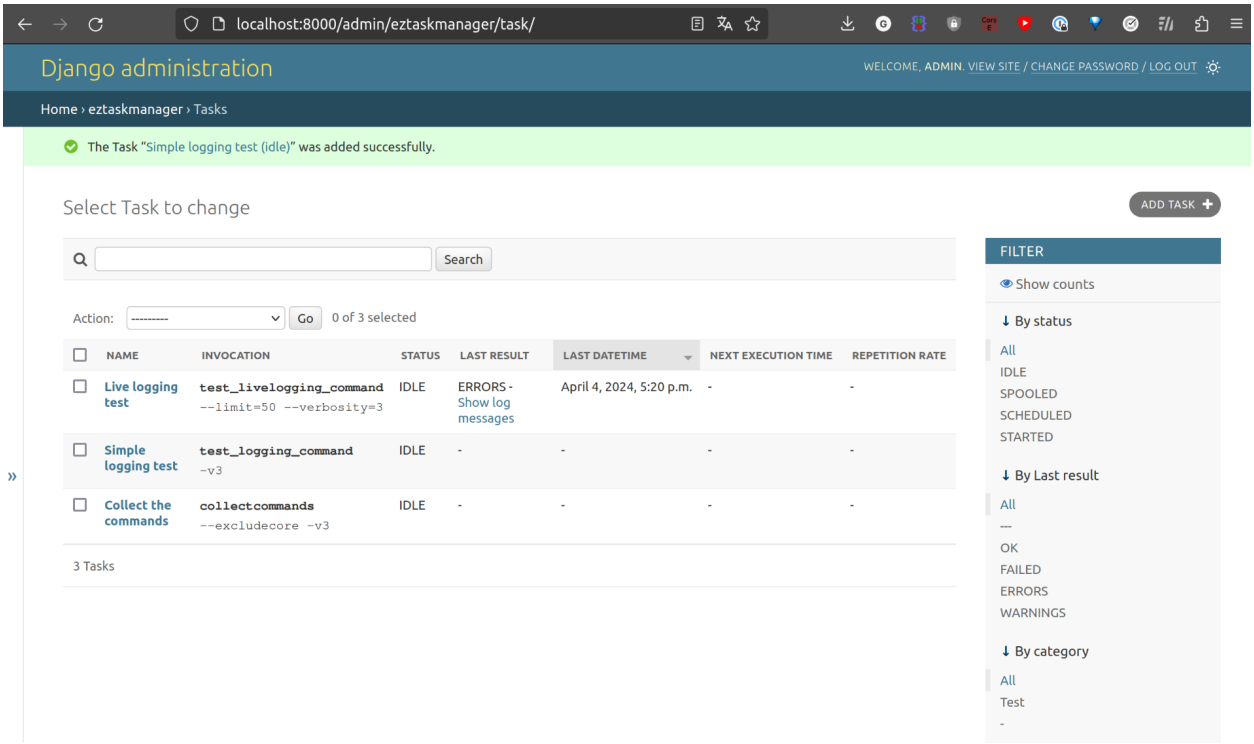
Commands are removable. To recreate tasks from these deleted commands, re-running the `collectcommands` task will be necessary.

Only those commands flagged as `active` can be utilised to generate tasks. Hence, to prevent a command from being used to create tasks, simply turn its `active` status to false.

You can also generate a task using the `collectcommands` command. This allows you to launch the collection of available commands directly through django-eztaskmanager.

2.1.2 Tasks Overview

The *Tasks* section serves as the central administration view where every operation takes place. Tasks can be listed, filtered, searched, created, modified, and removed using Django-admin's standard CRUD processes.



You have the capabilities to start or stop a task both in the *list view* and the *detail view*.

By default, tasks are sorted according to their latest launch time. This ensures that the most frequently used tasks are displayed upfront, avoiding clutter by infrequently used tasks. Additional sorting criteria can be applied by clicking the column headers.

The outcomes of the tasks are indicated both color-coded and with detailed notes of errors/warnings, if any. Tasks with warnings or errors (yellow and orange color codes) might still be functioning as expected as sometimes the errors can be attributed to issues with the data source. Tasks that fail (red code) require immediate attention as it suggests there are issues within the task's code or logic itself.

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

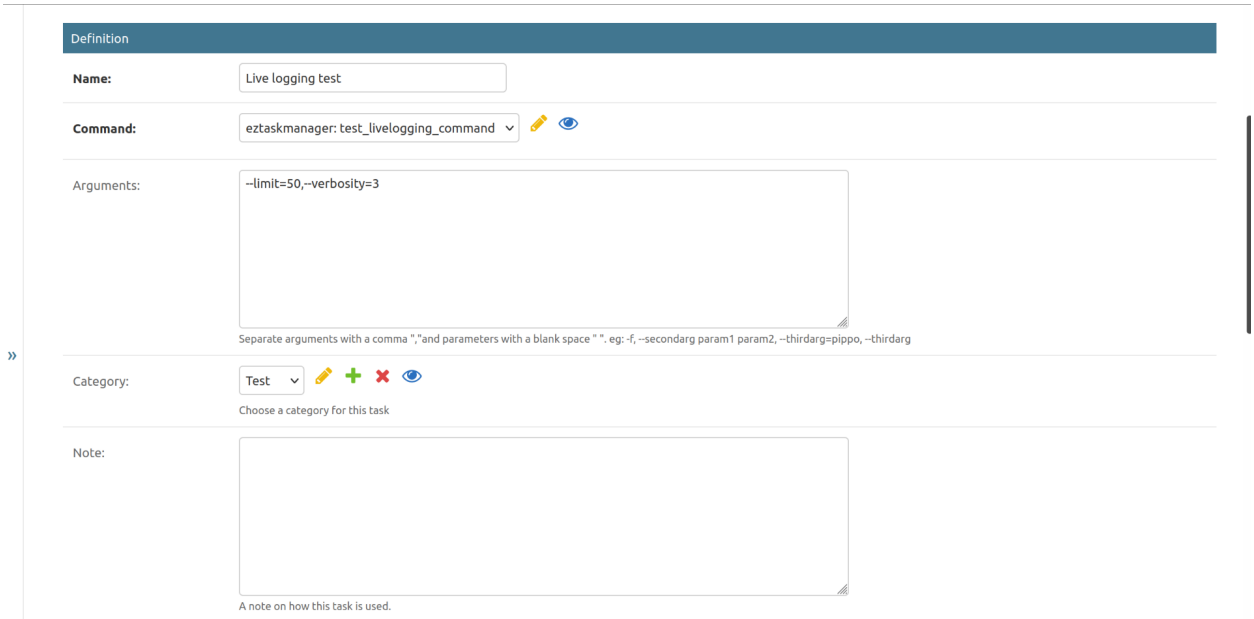
Hovering over the task name reveals a descriptive note, given that the task authors have added one. This note can provide insight into different aspects of the task instance and highlight any peculiarities of the arguments needed.

2.1.3 Task Structure

A task is comprised of four main sections:

- **Definition**: Contains the task name, command, arguments, category, and notes.
- **Scheduling**: Specifies the start time and recurrence rate.
- **Last Execution**: Shows the queued job id, status, last execution datetime, last result, next execution, and the count of warnings or errors.
- **Reports**: Every execution of a task generates a **Report**. Only the last five reports are stored and shown in each task's detail view.

2.1.4 Task Definition



The **Definition** section contains the following fields:

- **Name**: This is where you provide a unique name for the task. Using unique names with prefixes can facilitate easy visual identification of tasks.

Note: Remember that one command can be applied to multiple tasks with different arguments. Ensure that you give distinct **names** and describe the differences in detail in the **note** field. This will help other users make informed decisions about which task to use.

- **Command**: Select the appropriate command from the list available in the command popup.
- **Arguments**: Here, you enter the arguments the command requires using a specific syntax:

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

For example: `-f, --secondarg param1 param2, --thirdarg=pippo, --thirdarg`

- **Category**: Choose an existing category or create a new one for the task.
- **Note**: This field is for a descriptive note explaining how the command or its arguments are used.

2.1.5 Task Categories

Task categories are an efficient way of managing tasks when their quantity starts to increase. You can assign a category to each task and then filter the tasks list by category.

Note: Keep your category names simple and short. Try to limit the total number of categories to less than ten to avoid any confusion for other users.

2.1.6 Task Scheduling

The **Scheduling** process involves the following fields:

- **Scheduling**: Specify a date and time for the task's initial launch.
- **Repetition Period**: Select a frequency for the task to repeat: *minute*, *hour*, *day*, or *month*.
- **Repetition Rate**: Set a numerical value for the task's repetition rate.
- To **schedule a one-time future task**: Set the scheduling field to a future time and press the start button.
- To **schedule a recurring future task**: Set both scheduling and repetition fields, then press the start button.
- To **cancel a scheduled start**: Press the stop button.

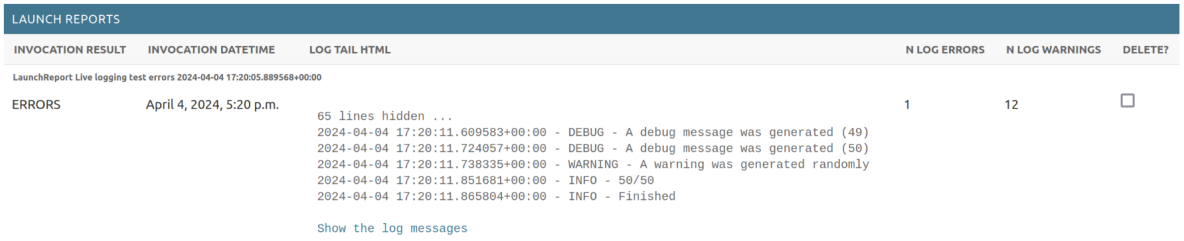
2.1.7 Understanding the Task's Last Execution Status

The fields in this section are *read-only* and display information about the task's last execution.

- **Status**: This can show one of the following: - `IDLE`: The task has either never started or it was stopped. - `STARTED`: The task is currently running. - `SCHEDULED`: The task is set to start at some point in the future.
- **Last Datetime**: This shows the date and time of the last execution.
- **Last Result**: This shows the result of the last execution:
  - `OK`: The task ran without any errors or warnings.
  - `WARNINGS`: The task ran correctly, but with warnings. Refer to the report for details.
  - `ERRORS`: The task ran correctly, but with errors. Refer to the report for details.
  - `FAILED`: There was a runtime error. Refer to the report for details.
- **Errors**: This shows the number of detected errors from the last execution.
- **Warnings**: This shows the number of detected warnings from the last execution.

Note: Before a task starts for the first time, it is put in the spooler. Therefore, the task's status may show as `SPOOLED`. A few moments later, after refreshing the page, the status will change to `STARTED`. This is to be expected.

2.1.8 Reading the Task's Reports

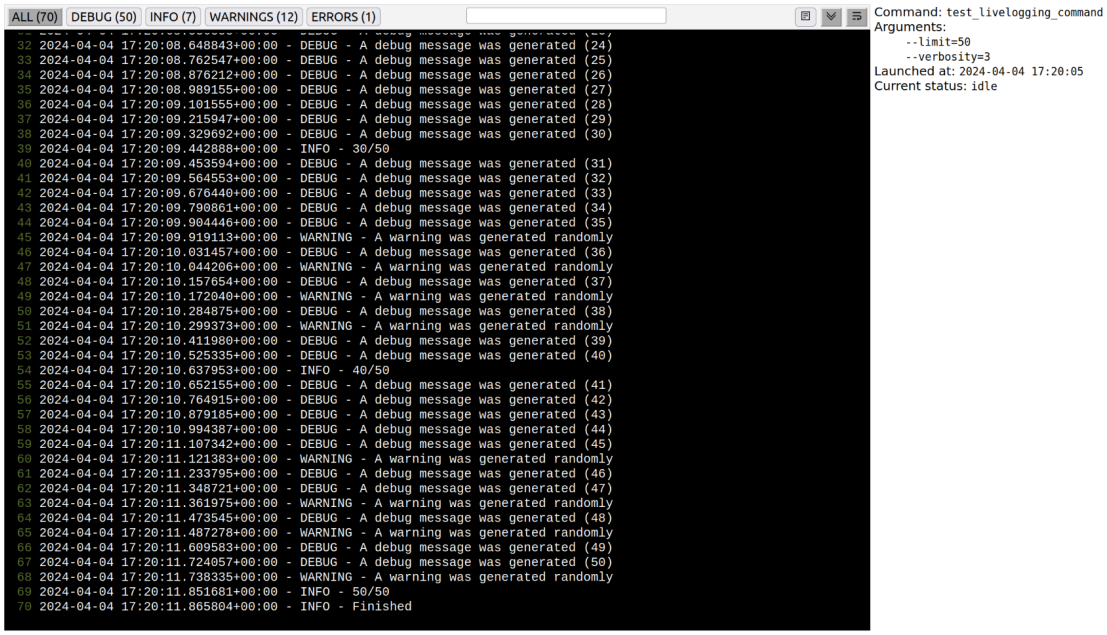


After a task is complete, a report is generated and added to the **reports** section. To conserve space, only the last 5 reports remain accessible for users.

Each report includes the **result** and **invocation datetime** fields, plus the last 10 log lines from the execution.

Clicking on the *show the log messages* link opens a new page containing the log messages.

Live logging test



If the task is still running, the page will refresh to display new messages as they're added.

At the top of the page is a **toolbar** divided into three sections:

- **Levels Buttons** (`ALL`, `DEBUG`, `INFO`, `WARNING`, `ERROR`): These function as filters. Clicking one only shows messages of that type. The numbers next to each button indicate the amount of messages per type. A button only appears after a message of its type has been added to the log file.
- **Search Field**: This helps in filtering messages by a specific string. Only messages containing this string are listed. Clicking on the 'x' button next to the search field will reset all filters (equivalent to pressing the `ALL` button).
- Commands on the right side of the toolbar: - The **raw logs** button opens a new page displaying the log files in raw text format. - The **sticky mode** button toggles the auto-scrolling of message displays. This can be used to focus on a specific part of the log messages.

Note: The entire list of log messages is rendered on a single page. This can cause long rendering times for lengthy lists. The recommended solution is to implement tasks that do not log excess messages… rubric:: Footnotes

---

2.2 Integrating django-eztaskmanager into an existing project

This guide is designed for **developers** who wish to incorporate *django-eztaskmanager* into their existing Django project.

0. Install the application using *pip*:

   Via **PyPI**:

   ```
   pip install django-eztaskmanager
   ```

   Or directly from **GitHub**:

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

1. Include "eztaskmanager" in your *INSTALLED_APPS* setting as follows:

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

2. Install and configure 'django-rq' and 'django-scheduler' or 'celery' and 'celery-beats' if they are not already included in your project.

3. Run `python manage.py migrate` to generate the eztaskmanager tables.

4. Execute the `collectcommands` management task to create taskmanager commands[1]:

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

5. Include the *eztaskmanager* URL configuration in your project's *urls.py* file:

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('django-rq/', include('django_rq.urls')),
    path('eztaskmanager/', include('eztaskmanager.urls'))
]
```

6. Configure parameters in your settings file as delineated below *(optional)*. Uncommented settings represent default values:

```
# eztaskmanager
# EZTASKMANAGER_QUEUE_SERVICE_TYPE = 'RQ'
# EZTASKMANAGER_N_LINES_IN_REPORT_LOG = 10
# EZTASKMANAGER_N_REPORTS_INLINE = 10
# EZTASKMANAGER_SHOW_LOGVIEWER_LINK = True
# EZTASKMANAGER_USE_FILTER_COLLAPSE = True
# EZTASKMANAGER_NOTIFICATION_HANDLERS = {}
# EZTASKMANAGER_BASE_URL = None
EZTASKMANAGER_NOTIFICATION_HANDLERS = {
    "email-errors": {
        "class": "eztaskmanager.services.notifications.
↪EmailNotificationHandler",
        "level": "failure",
        "from_email": "admin@example.com",
        "recipients": ["admin@example.com", ],
    },
}

EZTASKMANAGER_N_LINES_IN_REPORT_LOG = 5
# EZTASKMANAGER_SHOW_LOGVIEWER_LINK = False
```

[1] Using *excludecore* prevents fetching of core Django tasks.

7. Follow the *How to enable notifications* guide to set up notifications *(optional)*.

2.3 Integrating Django-eztaskmanager into a Dockerized Stack

This guide is intended for **developers** aiming to integrate **eztaskmanager** into their Django application within a *dockerized* setup. This specific case applies to an **rq** queue manager.

The illustrated `docker-compose.yml` depicts portions of a stack that operates an app via the **web** service.

The integration requires two more services: **rqworker** and **rqscheduler**

These three —together: the web, rqworker, and rqscheduler-, need to spring from the same Django app image while holding different start-up commands. Consequently, a consolidated shared environment gets defined, reused in the `docker_compose.yml` stack blueprint:

```
version: "3.7"

# Define anchor for reusable parts
x-shared-environment: &shared-environment
  environment:
    - POSTGRES_HOST
    - POSTGRES_PORT
    - POSTGRES_DB
    - POSTGRES_USER
    - POSTGRES_PASSWORD
    - DATABASE_URL=postgres://${POSTGRES_USER}:${POSTGRES_PASSWORD}@${POSTGRES_HOST}:$
↪{POSTGRES_PORT}/${POSTGRES_DB}
    - DJANGO_DEBUG
    - REDIS_URL
    - CI_COMMIT_SHA
    - ...

services:
  web:
    <<: *shared-environment
    container_name: ${COMPOSE_PROJECT_NAME}_web
    restart: always
    build:
      context: .
      dockerfile: ./compose/production/django/Dockerfile
    image: acme/${COMPOSE_PROJECT_NAME}_production_django
    deploy:
      replicas: 4
    expose:
      - "5000"
    depends_on:
      - postgres
      - redis
    command: /start
    networks:
      - default
      - gw
```

(continues on next page)

(continued from previous page)

```
rqworker:
  <<: *shared-environment
  restart: unless-stopped
  build:
    context: .
    dockerfile: ./compose/production/django/Dockerfile
  image: acme/${COMPOSE_PROJECT_NAME}_production_django
  deploy:
    replicas: 2
  depends_on:
    - postgres
    - redis
  command: /start-rqworker
  networks:
    - default
    - gw

rqscheduler:
  <<: *shared-environment
  restart: unless-stopped
  build:
    context: .
    dockerfile: ./compose/production/django/Dockerfile
  image: acme/${COMPOSE_PROJECT_NAME}_production_django
  deploy:
    replicas: 2
  depends_on:
    - postgres
    - redis
  command: /start-rqscheduler
  networks:
    - default
    - gw

postgres:
  container_name: ${COMPOSE_PROJECT_NAME}_postgres
  restart: unless-stopped
  build:
    context: .
    dockerfile: ./compose/production/postgres/Dockerfile
  image: acme/${COMPOSE_PROJECT_NAME}_production_postgres
  volumes:
    - postgres_data:/var/lib/postgresql/data
    - postgres_data_backups:/backups
  environment:
    - POSTGRES_HOST=${POSTGRES_HOST}
    - POSTGRES_PORT=${POSTGRES_PORT}
    - POSTGRES_DB=${POSTGRES_DB}
    - POSTGRES_USER=${POSTGRES_USER}
    - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}

redis:
```

(continues on next page)

(continued from previous page)

```
    container_name: ${COMPOSE_PROJECT_NAME}_redis
    volumes:
      - redis_data:/data
    restart: always
    image: redis:latest
```

Note: The YAML file is only partially shown here for explanatory purposes. Adjustments might be needed based on your specific application.

Note: In the `docker_compose.yml` example, a reference to `./compose/production/django` indicates the residence of the django image's `Dockerfile`, paired with the bash scripts launching the server, the worker, and the scheduler operations.

The `start` command would resemble:

```
#!/bin/bash
exec /usr/local/bin/gunicorn config.wsgi --bind 0.0.0.0:5000 --chdir=/app
```

The `start-rqworker` command would be:

```
#!/bin/bash
python manage.py rqworker default --with-scheduler
```

And, for `start-rqscheduler`:

```
#!/bin/bash
python manage.py rqscheduler --verbosity=2
```

As for **Celery**, the same logic would apply, only the starting commands would change, using something similar to:

```
#!/bin/sh

celery -A proj worker -l info --concurrency 2
celery -A proj beat -l info ..concurrency 2
```

2.4 How to enable notifications

The `notifications system` enables `django-eztaskmanager` 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.

Thanks to the work of Gabriele Lucci in django_uwsgi_taskmanager, the notifications system is *pluggable*. It comes with **email** and **slack**. 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-eztaskmanager[slack]
```

This will install the `django-eztaskmanager` package from PyPI, including the optional slack_sdk 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 `EZTASKMANAGER_NOTIFICATION_HANDLERS` setting variable as a dictionary with the chosen handlers.

For example, to set up the slack notification handler:

```
EZTASKMANAGER_NOTIFICATION_HANDLERS = {
    "slack": {
        "class": "taskmanager.notifications.SlackNotificationHandler",
        "level": "errors",
        "token": env("EZTASKMANAGER_NOTIFICATIONS_SLACK_TOKEN", default=""),
        "channel": env("EZTASKMANAGER_NOTIFICATIONS_SLACK_CHANNELS", default=""),
    },
}
```

with the following env variables set:

- `EZTASKMANAGER_NOTIFICATIONS_SLACK_TOKEN`, the Slack token as string.
- `EZTASKMANAGER_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:

```
EZTASKMANAGER_NOTIFICATION_HANDLERS = {
    "mail": {
        "class": "taskmanager.notifications.MailNotificationHandler",
        "level": "errors",
        "from_email": env("EZTASKMANAGER_NOTIFICATIONS_EMAIL_FROM", default=""),
        "recipients": env("EZTASKMANAGER_NOTIFICATIONS_EMAIL_RECIPIENTS", default=""),
    },
}
```

with the following env variables:

- `EZTASKMANAGER_NOTIFICATIONS_EMAIL_FROM`, the "from address" you want your outgoing notification emails to use.
- `EZTASKMANAGER_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 `eztaskmanager.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 `LaunchReport` class will call all registered handlers and emit the notifications. It is called at the end of `eztaskmanager.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 Contribute to the Project

2.5.1 Documentation

The project documentation, present in the `docs` directory, is created using sphinx. We adhere to the guidelines for creating technical documentation proposed by Daniele Procida.

To contribute to the documentation, ensure that your virtual environment has the following packages installed:

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

To build the documentation, navigate to the `docs` directory and run the following commands:

```
make clean
make html
```

Our `makefile` has been customized from the original produced by the `sphinx-quickstart` script, and includes a `livehtml` target. This facilitates the automatic rebuilding of HTML output whenever changes to the rst source files are made.

```
make livehtml
```

2.5.2 Development

The source code is hosted on https://github.com/openpolis/django-eztaskmanager.

There's a suite of unit tests. Run them with:

```
python demoproject/manage.py test
```

Source code syntax and formatting are validated using flake8.

2.6 Debugging Tasks

Debugging can be complex with multiple running components.

Prior to running the development components, a Docker stack must be executed because it's necessary for Redis to be operational (and Mailhog helpfully tests email notifications).

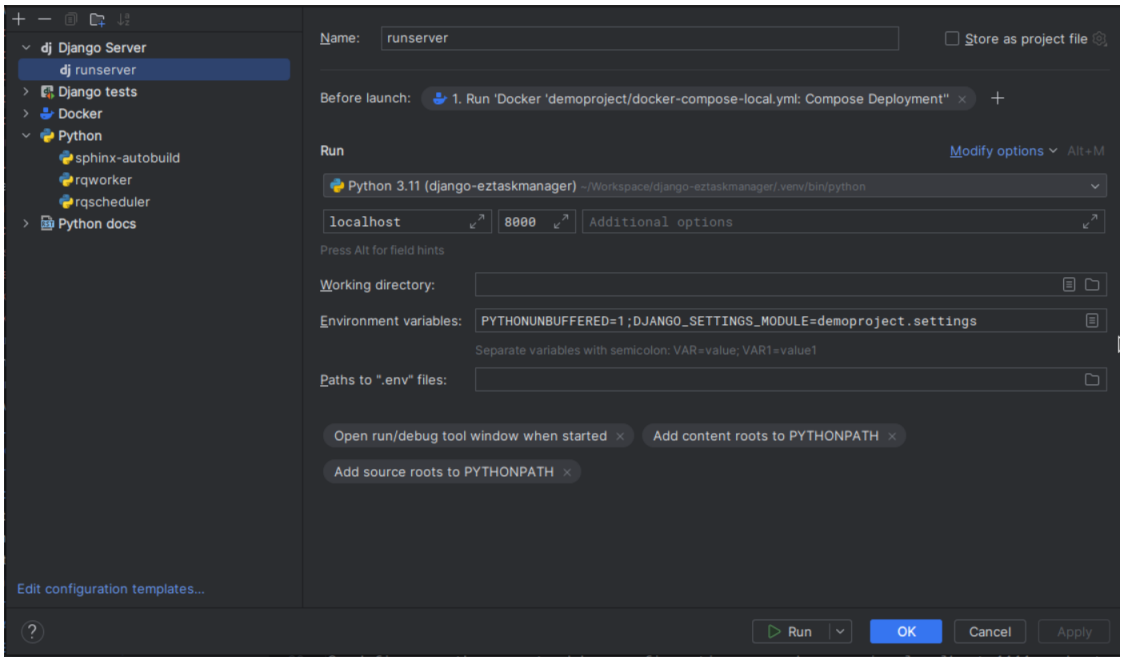
Two components can be debugged during development:

- The `runserver`, used for testing the admin application and the livelogviewer view.
- The `rqworker`, used for testing the wrapper that initiates the tasks.

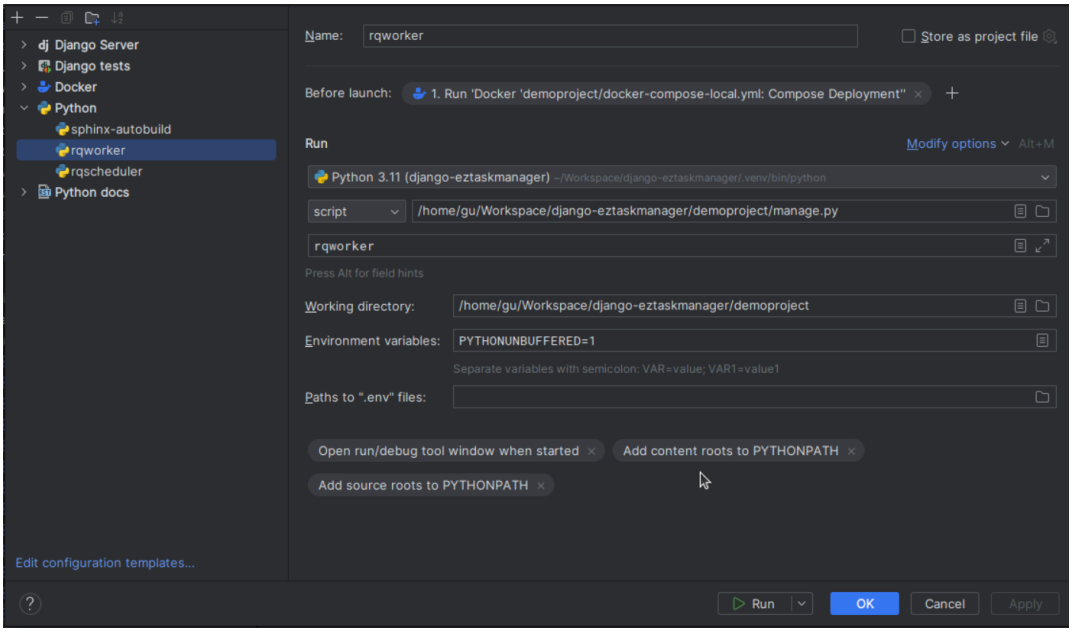
In the PyCharm IDE, you need to establish three run/debug configurations:

- The Django server configuration, which controls the `runserver`. A *Before launch* task

is added to this that runs the Docker stack before the *runserver*. This ensures Redis and Mailhog are operational.

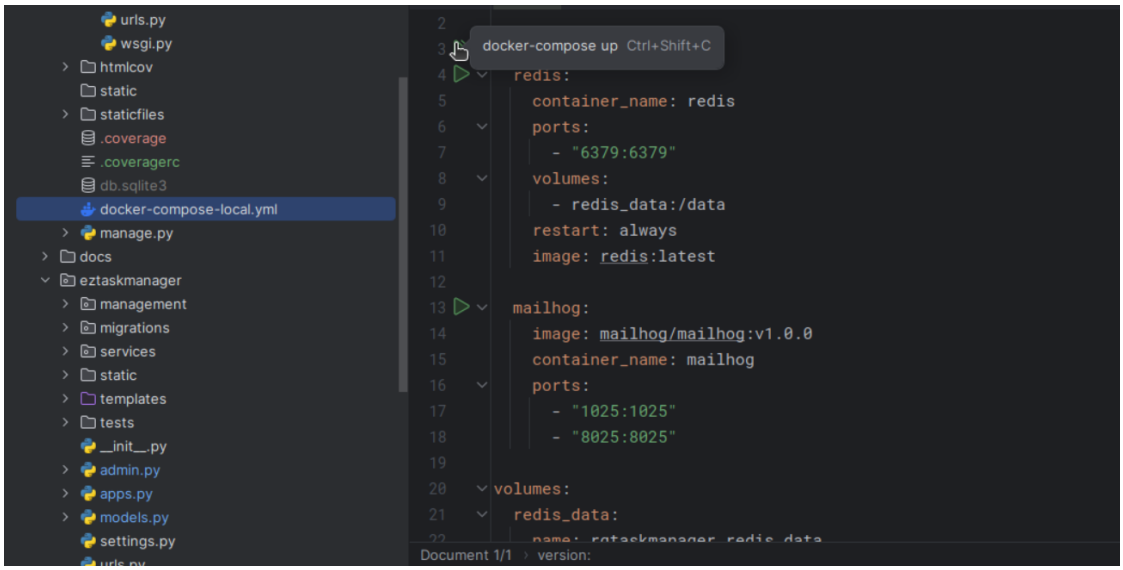


- The *rqworker* configuration.



Note: To make a configuration for local Docker compose execution, simply open the file in

PyCharm and click on the double green arrows. This actions *docker compose up* and creates a temporary run/debug configuration. Save this configuration to reuse it later.



CHAPTER

THREE

REFERENCE

Classes and functions are documented here thoroughly for the developer.

Information is fetched from the comments in the source code.

3.1 models

class eztaskmanager.models.**AppCommand**(**args*, ***kwargs*)

An application command representation.

Parameters

- **id** (*BigAutoField*) – Id
- **name** (*CharField*) – Name
- **app_name** (*CharField*) – App name
- **active** (*BooleanField*) – Active

name

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

app_name

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

active

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

get_command_class()

Get the command class.

property help_text

Get the command help text.

exception DoesNotExist

exception MultipleObjectsReturned

id

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

objects = <django.db.models.manager.Manager object>

task_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

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

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

class eztaskmanager.models.**LaunchReport**(**args*, ***kwargs*)

A report of a task execution with log.

Parameters

- **id** (*BigAutoField*) – Id
- **task_id** (*ForeignKey*) – Task
- **invocation_result** (*CharField*) – Invocation result
- **invocation_datetime** (*DateTimeField*) – Invocation datetime

RESULT_NO = ''

RESULT_OK = 'ok'

RESULT_FAILED = 'failed'

RESULT_ERRORS = 'errors'

RESULT_WARNINGS = 'warnings'

RESULT_CHOICES = (('', '---'), ('ok', 'OK'), ('failed', 'FAILED'), ('errors', 'ERRORS'), ('warnings', 'WARNINGS'))

task

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

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

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

invocation_result

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

invocation_datetime

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

classmethod get_notification_handlers()

Get the list of notification handlers to send the report to.

get_log_lines()

Format the log entries, here is an example.

read_log_lines(*offset: int*)

Use an offset to read lines of the llog related to the report (self) starting from the offset.

param
offset lines to start from

return
2-tuple (list, int) - list of lines of log records from offset - the number of total lines

log_tail(*n_lines=10*)

Return the last lines of the logs of a launch_report.

property n_log_lines

Return the number of log lines for this report.

property n_log_errors

Return the number of errors in this report.

property n_log_warnings

Return the number of warnings in this report.

delete(**args*, ***kwargs*)

Refresh the task cache after deleting this report.

exception DoesNotExist

exception MultipleObjectsReturned

get_invocation_result_display(*, *field=<django.db.models.fields.CharField: invocation_result>*)

get_next_by_invocation_datetime(*, *field=<django.db.models.fields.DateTimeField: invocation_datetime>*, *is_next=True*, ***kwargs*)

get_previous_by_invocation_datetime(*, *field=<django.db.models.fields.DateTimeField: invocation_datetime>*, *is_next=False*, ***kwargs*)

id

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

logs

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

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

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

objects = <django.db.models.manager.Manager object>

task_id

class eztaskmanager.models.**Log**(**args*, ***kwargs*)

The log generated by a report.

Parameters

- **id** (*BigAutoField*) – Id
- **launch_report_id** (*ForeignKey*) – Launch report
- **timestamp** (*DateTimeField*) – Timestamp
- **level** (*CharField*) – Level
- **message** (*TextField*) – Message

launch_report

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

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

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

timestamp

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

level

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

message

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

exception DoesNotExist

exception MultipleObjectsReturned

get_next_by_timestamp(**, field=<django.db.models.fields.DateTimeField: timestamp>, is_next=True, **kwargs*)

get_previous_by_timestamp(**, field=<django.db.models.fields.DateTimeField: timestamp>, is_next=False, **kwargs*)

id

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

launch_report_id

objects = <django.db.models.manager.Manager object>

class eztaskmanager.models.**TaskCategory**(**args*, ***kwargs*)

A task category, used to group tasks when numbers go up.

Parameters

- **id** (*BigAutoField*) – Id
- **name** (*CharField*) – Name

name

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

exception DoesNotExist

exception MultipleObjectsReturned

id

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

objects = <django.db.models.manager.Manager object>

task_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

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

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

class eztaskmanager.models.**Task**(**args*, ***kwargs*)

A command related task.

Represents a management command with a defined set of arguments (

Parameters

- **id** (*BigAutoField*) – Id
- **name** (*CharField*) – Name
- **command_id** (*ForeignKey*) – Command
- **arguments** (*TextField*) – Separate arguments with a comma ","and parameters with a blank space " ". eg: -f, –secondarg param1 param2, –thirdarg=pippo, –thirdarg
- **category_id** (*ForeignKey*) – Choose a category for this task
- **status** (*CharField*) – Status
- **scheduling** (*DateTimeField*) – Initial scheduling
- **repetition_period** (*CharField*) – Repetition period
- **repetition_rate** (*PositiveSmallIntegerField*) – Repetition rate
- **note** (*TextField*) – A note on how this task is used.
- **scheduled_job_id** (*CharField*) – A unique identifier for the scheduled job, if any

- **cached_last_invocation_datetime** (*DateTimeField*) – Last datetime
- **cached_last_invocation_result** (*CharField*) – Last result
- **cached_last_invocation_n_errors** (*PositiveIntegerField*) – Errors
- **cached_last_invocation_n_warnings** (*PositiveIntegerField*) – Warnings
- **cached_next_ride** (*DateTimeField*) – Next execution time

REPETITION_PERIOD_MINUTE = 'minute'

REPETITION_PERIOD_HOUR = 'hour'

REPETITION_PERIOD_DAY = 'day'

REPETITION_PERIOD_WEEK = 'week'

REPETITION_PERIOD_MONTH = 'month'

REPETITION_PERIOD_CHOICES = (('minute', 'MINUTE'), ('hour', 'HOUR'), ('day', 'DAY'), ('month', 'MONTH'))

STATUS_IDLE = 'idle'

STATUS_SPOOLED = 'spooled'

STATUS_SCHEDULED = 'scheduled'

STATUS_STARTED = 'started'

STATUS_CHOICES = (('idle', 'IDLE'), ('spooled', 'SPOOLED'), ('scheduled', 'SCHEDULED'), ('started', 'STARTED'))

name

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

command

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

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

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

arguments

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

category

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

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

Child.parent is a ForwardManyToOneDescriptor instance.

status

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

scheduling

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

property scheduling_utc

Sho the scheduling time, in UTC.

repetition_period

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

repetition_rate

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

property is_periodic

A periodic task is such only if both repetition period and rate are set.

note

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

scheduled_job_id

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

cached_last_invocation_datetime

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

cached_last_invocation_result

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

cached_last_invocation_n_errors

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

cached_last_invocation_n_warnings

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

cached_next_ride

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

property interval_in_seconds

Returns the interval in seconds based on the repetition period and rate.

Returns
The interval in seconds.

Return type
int

Example

Create an instance of the class obj = MyClass()

Set the repetition period and rate obj.repetition_period = 'day' obj.repetition_rate = 2

Calculate the interval in seconds result = obj.interval_in_seconds() # Returns 2 * 24 * 60 * 60

property args

Get the task args.

property options

Get the task options.

property complete_args

Returns a list containing all the non-null values from the dictionary of arguments.

Get all task args in order to avoid problems with required options.

Returns
A list containing non-null argument values.

As suggested here: https://stackoverflow.com/questions/32036562/call-command-argument-is-required

compute_cache()

Compute cached values for this task.

prune_reports(*n: int = 5*)

Delete all Task's LaunchReports except latest *n*.

exception DoesNotExist

exception MultipleObjectsReturned

category_id

command_id

get_cached_last_invocation_result_display(*, *field=<django.db.models.fields.CharField: cached_last_invocation_result>*)

get_repetition_period_display(*, *field=<django.db.models.fields.CharField: repetition_period>*)

get_status_display(*, *field=<django.db.models.fields.CharField: status>*)

id

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

launchreport_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

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

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

objects = <django.db.models.manager.Manager object>

3.2 views

Define Django views for the taskmanager app.

class eztaskmanager.views.**LogViewerView**(***kwargs*)

Class LogViewerView displays a log viewer page with log information for a specific report.

template_name = 'log_viewer.html'

static get_report_lines(*report*)

Return the log lines for this report.

get_context_data(***kwargs*)

Return the context data for the view.

class eztaskmanager.views.**LiveLogViewerView**(***kwargs*)

A template view to view the rolling report log.

template_name = 'live_log_viewer.html'

get_context_data(***kwargs*)

Return the context data for the view, removing the offset file, to allow reading log lines from the start.

class eztaskmanager.views.**AjaxReadLogLines**(***kwargs*)

Read log lines starting from an offset, as JsonResponse.

New log size and task status are included in the response.

render_to_response(*context*, ***response_kwargs*)

Render a response with JSON data.

Parameters

- **self** – The instance of the class calling the method.
- **context** (*dict*) – A dictionary containing the context data.
- ****response_kwargs** – Additional keyword arguments used for the response.

Returns

A response object with JSON data containing the following keys:

- new_log_lines (list): List of log lines.
- task_status (str): The status of the task.
- log_size (int): The size of the log.

Return type
JsonResponse

Raises
None. –

3.3 services

eztaskmanager.services.**run_management_command**(*task_id: int*)

Execute a management command.

Creates a LaunchReport for this execution.

Parameters
task_id (*int*) – The task object representing the management command to be executed.

Returns
None

3.3.1 queues

Task queue services.

These are the class that implement the interface to the queue managers.

The abstract TaskQueueService class is the interface each class has to implement.

- RQTaskQueueService implements the service with Redis Queue.
- CeleryTaskQueueService implements the service with Celery (TBD)

class eztaskmanager.services.queues.**TaskQueueService**

Abstract base class for managing task queues.

abstract add(*task*)

To be implemented in concrete subclasses.

abstract remove(*task*)

To be implemented in concrete subclasses.

eztaskmanager.services.queues.**get_task_service**()

Fetch the correct queue service, based on settings.

exception eztaskmanager.services.queues.**TaskQueueException**

Dedicated exception for TaskQueue classes.

class eztaskmanager.services.queues.**RQTaskQueueService**

A subclass of TaskQueueService that manages tasks using RQ (Redis Queue).

queue

The default RQ queue.

Type
Queue

- add(task, at=None)

Enqueues a task to be executed either immediately or at a specific time.

- remove(task)

Cancels a task if it is currently in the queue.

add(*task: Task*)

Add the task to the Redis queue.

Parameters
task – The task to be added.

Returns
The job created for the task, either scheduled or enqueued for immediate execution.

Raises
Exception – If there is an error while launching the task.

fetch_job_with_next_time(*task*)

Fetch the next job in the queue, with its execution time.

remove(*task*)

Remove the job from the queue and updates the tasks' values.

class eztaskmanager.services.queues.**CeleryTaskQueueService**

A subclass of TaskQueueService that manages tasks using Celery.

add(*task:* Task)

Add a task to the Celery queue.

remove(*task*)

Remove a job from the Celery queue.

3.3.2 logger

eztaskmanager.services.logger.**verbosity2loglevel**(*verbosity*)

Map verbosity level to logging level.

class eztaskmanager.services.logger.**LoggerEnabledCommand**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)

This class is a subclass of BaseCommand that adds logging functionality to the execute method.

logger = None

execute(**args*, ***kwargs*)

Override the BaseCommand method, adding stream and Database handlers, if not existing.

create_parser(*prog_name*, *subcommand*, ***kwargs*)

Create a parser.

class eztaskmanager.services.logger.**DatabaseLogHandler**(*launch_report_id*)

A handler class that logs messages to a database.

This class extends the logging.Handler class and provides functionality to log messages to a database. Each log message is saved as a Log object in the database with the launch report level, and message attributes.

Usage:
log_handler = DatabaseLogHandler("launch_report_1") logger.addHandler(log_handler) logger.error("An error occurred")

emit(*record*)

Implement the method to send the log message to the DB.

3.3.3 notifications

eztaskmanager.services.notifications.**get_base_url**()

Retrieve the base URL for the current site.

Returns
The base URL for the current site.

Return type
str

Raises
None. –

Examples

```
>>> get_base_url()
'example.com'
```

class eztaskmanager.services.notifications.**NotificationHandler**(*level: int | str = 0*)

An abstract base class for handling notifications.

handle(*report*)

Check the result of the report against the established lo level before emitting notifications.

abstract emit(*report*)

Abstract method. To be implemented in concrete classes.

class eztaskmanager.services.notifications.**SlackNotificationHandler**(*token*, *channel*, *level*)

This class is a notification handler that sends notifications to a specified Slack channel using the Slack API.

Params:
client (slack_sdk.WebClient): The Slack WebClient object used to interact with the Slack API. channel (str): The Slack channel to which the notifications will be sent. level (int): The log level at which notifications will be sent.

emit(*report*)

Sends an email notification based on the given report.

emit(*report*)

Send a Slack notification based on the given report's result.

class eztaskmanager.services.notifications.**EmailNotificationHandler**(*from_email*, *recipients*, *level*)

A class for handling email notifications.

Inherits from NotificationHandler.

from_email

The email address to use as the sender of the notification.

Type
str

recipients

A list of email addresses to send the notification to.

Type
list[str]

level

The level of the notification.

Type
int

emit(*report*)

Sends an email notification based on the given report.

emit(*report*)

Send an email notification based on the given level.

eztaskmanager.services.notifications.**emit_notifications**(*report*)

Emit notifications for the given report.

Parameters
report – The report object containing the invocation result.

Returns
None

3.4 django.core.management.base

Base classes for writing management commands (named commands which can be executed through `django-admin` or `manage.py`).

exception django.core.management.base.**CommandError**(**args*, *returncode=1*, ***kwargs*)

Exception class indicating a problem while executing a management command.

If this exception is raised during the execution of a management command, it will be caught and turned into a nicely-printed error message to the appropriate output stream (i.e., stderr); as a result, raising this exception (with a sensible description of the error) is the preferred way to indicate that something has gone wrong in the execution of a command.

exception django.core.management.base.**SystemCheckError**(**args*, *returncode=1*, ***kwargs*)

The system check framework detected unrecoverable errors.

class django.core.management.base.**CommandParser**(*, *missing_args_message=None*, *called_from_command_line=None*, ***kwargs*)

Customized ArgumentParser class to improve some error messages and prevent SystemExit in several occasions, as SystemExit is unacceptable when a command is called programmatically.

parse_args(*args=None*, *namespace=None*)

error(*message: string*)

Prints a usage message incorporating the message to stderr and exits.

If you override this in a subclass, it should not return – it should either exit or raise an exception.

add_subparsers(***kwargs*)

django.core.management.base.**handle_default_options**(*options*)

Include any default options that all commands should accept here so that ManagementUtility can handle them before searching for user commands.

django.core.management.base.**no_translations**(*handle_func*)

Decorator that forces a command to run with translations deactivated.

class django.core.management.base.**DjangoHelpFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Customized formatter so that command-specific arguments appear in the –help output before arguments common to all commands.

show_last = {'--force-color', '--no-color', '--pythonpath', '--settings', '--skip-checks', '--traceback', '--verbosity', '--version'}

add_usage(*usage*, *actions*, **args*, ***kwargs*)

add_arguments(*actions*)

class django.core.management.base.**OutputWrapper**(*out*, *ending='\n'*)

Wrapper around stdout/stderr

property style_func

flush()

Flush write buffers, if applicable.

This is not implemented for read-only and non-blocking streams.

isatty()

Return whether this is an 'interactive' stream.

Return False if it can't be determined.

write(*msg=''*, *style_func=None*, *ending=None*)

Write string to stream. Returns the number of characters written (which is always equal to the length of the string).

class django.core.management.base.**BaseCommand**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)

The base class from which all management commands ultimately derive.

Use this class if you want access to all of the mechanisms which parse the command-line arguments and work out what code to call in response; if you don't need to change any of that behavior, consider using one of the subclasses defined in this file.

If you are interested in overriding/customizing various aspects of the command-parsing and -execution behavior, the normal flow works as follows:

1. `django-admin` or `manage.py` loads the command class and calls its `run_from_argv()` method.
2. The `run_from_argv()` method calls `create_parser()` to get an `ArgumentParser` for the arguments, parses them, performs any environment changes requested by options like `pythonpath`, and then calls the `execute()` method, passing the parsed arguments.
3. The `execute()` method attempts to carry out the command by calling the `handle()` method with the parsed arguments; any output produced by `handle()` will be printed to standard output and, if the command is intended to produce a block of SQL statements, will be wrapped in `BEGIN` and `COMMIT`.
4. If `handle()` or `execute()` raised any exception (e.g. `CommandError`), `run_from_argv()` will instead print an error message to `stderr`.

Thus, the `handle()` method is typically the starting point for subclasses; many built-in commands and command types either place all of their logic in `handle()`, or perform some additional parsing work in `handle()` and then delegate from it to more specialized methods as needed.

Several attributes affect behavior at various steps along the way:

help
A short description of the command, which will be printed in help messages.

output_transaction
A boolean indicating whether the command outputs SQL statements; if `True`, the output will automatically be wrapped with `BEGIN;` and `COMMIT;`. Default value is `False`.

requires_migrations_checks
A boolean; if `True`, the command prints a warning if the set of migrations on disk don't match the migrations in the database.

requires_system_checks
A list or tuple of tags, e.g. [Tags.staticfiles, Tags.models]. System checks registered in the chosen tags will be checked for errors prior to executing the command. The value '__all__' can be used to specify that all system checks should be performed. Default value is '__all__'.

To validate an individual application's models rather than all applications' models, call `self.check(app_configs)` from `handle()`, where `app_configs` is the list of application's configuration provided by the app registry.

stealth_options
A tuple of any options the command uses which aren't defined by the argument parser.

help = ''

output_transaction = False

requires_migrations_checks = False

requires_system_checks = '__all__'

base_stealth_options = ('stderr', 'stdout')

stealth_options = ()

suppressed_base_arguments = {}

get_version()
Return the Django version, which should be correct for all built-in Django commands. User-supplied commands can override this method to return their own version.

create_parser(*prog_name*, *subcommand*, ***kwargs*)
Create and return the `ArgumentParser` which will be used to parse the arguments to this command.

add_arguments(*parser*)
Entry point for subclassed commands to add custom arguments.

add_base_argument(*parser*, **args*, ***kwargs*)
Call the parser's add_argument() method, suppressing the help text according to BaseCommand.suppressed_base_arguments.

print_help(*prog_name*, *subcommand*)
Print the help message for this command, derived from `self.usage()`.

run_from_argv(*argv*)
Set up any environment changes requested (e.g., Python path and Django settings), then run this command. If the command raises a `CommandError`, intercept it and print it sensibly to stderr. If the `--traceback` option is present or the raised `Exception` is not `CommandError`, raise it.

execute(**args*, ***options*)

Try to execute this command, performing system checks if needed (as controlled by the `requires_system_checks` attribute, except if force-skipped).

check(*app_configs=None*, *tags=None*, *display_num_errors=False*, *include_deployment_checks=False*, *fail_level=40*, *databases=None*)

Use the system check framework to validate entire Django project. Raise CommandError for any serious message (error or critical errors). If there are only light messages (like warnings), print them to stderr and don't raise an exception.

check_migrations()

Print a warning if the set of migrations on disk don't match the migrations in the database.

handle(**args*, ***options*)

The actual logic of the command. Subclasses must implement this method.

class django.core.management.base.**AppCommand**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)

A management command which takes one or more installed application labels as arguments, and does something with each of them.

Rather than implementing `handle()`, subclasses must implement `handle_app_config()`, which will be called once for each application.

missing_args_message = 'Enter at least one application label.'

add_arguments(*parser*)

Entry point for subclassed commands to add custom arguments.

handle(**app_labels*, ***options*)

The actual logic of the command. Subclasses must implement this method.

handle_app_config(*app_config*, ***options*)

Perform the command's actions for app_config, an AppConfig instance corresponding to an application label given on the command line.

class django.core.management.base.**LabelCommand**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)

A management command which takes one or more arbitrary arguments (labels) on the command line, and does something with each of them.

Rather than implementing `handle()`, subclasses must implement `handle_label()`, which will be called once for each label.

If the arguments should be names of installed applications, use `AppCommand` instead.

label = 'label'

missing_args_message = 'Enter at least one label.'

add_arguments(*parser*)

Entry point for subclassed commands to add custom arguments.

handle(**labels*, ***options*)

The actual logic of the command. Subclasses must implement this method.

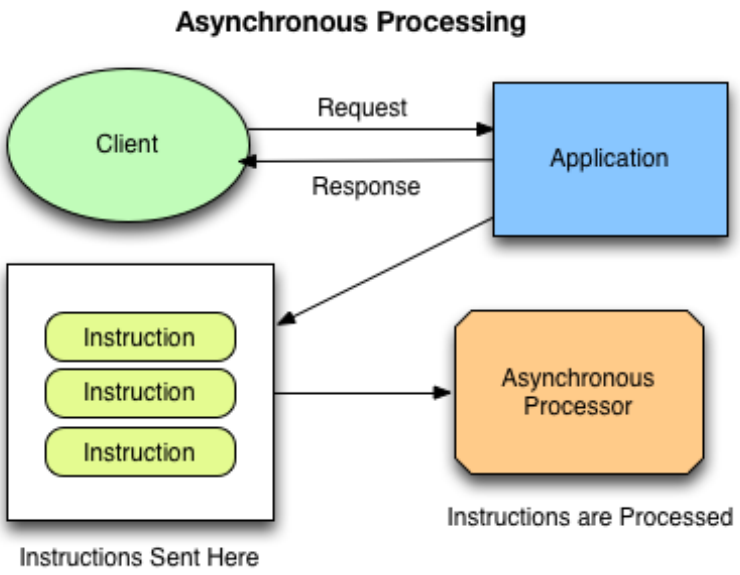
handle_label(*label*, ***options*)

Perform the command's actions for `label`, which will be the string as given on the command line.

CHAPTER

FOUR

DISCUSSIONS



4.1 Interface for complex tasks Django ecosystem

Managing and maintaining a Django-based application involves executing complex tasks such as, for example, Extract, Transform, and Load (ETL) operations, cleanup operations, indexing, ... While these tasks are fundamental for data management/warehousing, their manual execution can be laborious and prone to errors.

Providing a dedicated interface for such operations not only streamlines the process but also enhances efficiency and reduces potential mistakes that can arise from manual intervention.

An automated, seamless and robust system is thus crucial in handling such operations. This offers benefits in terms of task efficiency, reduction of manual intervention, and elimination of potential errors that can arise from manual process steps.

4.2 Integration into Django admin site

Equally crucial is the requirement for the said interface to be integrated within the native Django admin site. The rationale behind this is to bring about ease of use and accessibility particularly to non-technical personnel who need to interact with the system.

The Django admin site provides an out-of-the-box, user-friendly GUI that simplifies the management tasks. Therefore, integrating the custom interface into the Django admin site will offer non-technical users a familiar and intuitive environment to execute complex tasks without the need to write or understand code.

4.3 Use of a Queue Manager

A key component in managing and executing complex tasks is the use of a job or task queue manager. In the context of Python and Django, robust queue managers namely RQ (Redis Queue) and Celery stand out as the most popular options. These tools facilitate the management and processing of asynchronous tasks which are queued and executed based on priority, scheduled time, or other custom logic.

The need for such a setup arises from the inherent complexity of managing multiple long-running tasks, each possibly varying in computational requirements. Without a queue manager, the system risks running into resource allocation issues, redundancy, and failure in task execution. In contrast, utilizing a queue manager provides control over resource allocation, task prioritization and consequently, a more efficient and reliable system. Choosing between RQ and Celery will depend on specific application requirements, although either will contribute significantly to streamline task management.

In conclusion, integrating an interface for complex operations into the Django admin site, and utilizing a reliable queue manager, are essential steps towards efficient and reliable Django application management, particularly for non-technical users.

PYTHON MODULE INDEX

d

e

INDEX

I

L

M

N

O

P

Q

R

S

T

V

W