yourlabs Documentation

Release 0.0.0

is_null

November 30, 2011

CONTENTS

This repo contains various Django *super simple* applications we use internally.

You want to read this page before using the following application specific documentation shortcuts:

CHAPTER

ONE

YOURLABS.RUNNER

It is frequent for projects to need commands to be executed continuously. When cron or spoolers aren't the way to go, runner provides a simple way to create background threads which chains commands continuously.

1.1 Install

This app just provides a command: add *yourlabs.runner* in your project's *settings.INSTALLED_APPS*.

Also, runner expects the following settings:

- *settings.LOGGING['loggers']['runner']*: your logger config
- *settings.RUN_ROOT*: the path where it should create its pidfiles

1.2 Usage

Example usage for command *run_functions*:

```
./manage.py run_functions tasks.send_mail
```

This would continuously run the *send_mail* command if, for example, your project root contained such a *tasks.py* file:

```
from django.core.management import call_command

def send_mail():
    call_command('send_mail')
```

1.2.1 Task chains

It can continuously run any number of tasks in the specified order:

```
./manage.py run_functions tasks.send_mail tasks.retry_deferred
```

The advantage of splitting tasks is monitoring and error reporting.

1.2.2 Cooldown time

After a function is run, runner can wait a little: the *cooldown*. For example, for your database server to flush and so on.

A *cooldown* time should be adjusted in each task, on each server, to balance between getting the job done and being resource-reasonnable, is easy with the runner.task decorator:

```
import time
from datetime import timedelta as td

from django.core.management import call_command

from yourlabs import runner

# wait 15 minutes after a successfull execution
# and wait 30 minutes after a failed execution
@runner.task(success_cooldown=td(minutes=15), fail_cooldown=td(minutes=30))
def send_mail():
    call_command('send_mail')
```

1.2.3 Customize privileges

Running the tasks under a particular user is easy in bash, for example:

```
su $username -c "source /srv/$domain/env/bin/activate && \
    nice -n 5 /srv/$domain/main/manage.py run_functions \
        tasks.send_mail \
        tasks.retry_deferred \
    &>> /srv/$domain/log/runner_debug_0 & disown"
```

1.2.4 Customize process priority

This example shows how to give priority to the runner of *gsm_sync_live* over the *send_mail_retry_deferred* runner:

```
su $username -c "source /srv/$domain/env/bin/activate && \
    nice -n 5 /srv/$domain/main/manage.py run_functions \
        tasks.gsm_sync_live \
&>> /dev/null & disown"
su $username -c "source /srv/$domain/env/bin/activate && \
    nice -n 15 /srv/$domain/main/manage.py run_functions \
        tasks.send_mail \
        tasks.retry_deferred \
&>> /dev/null & disown"
```

To know more about process priorities and scheduling configuration, read the manual of the nice command used in this example.

1.3 Maintenance

One of the main goals of *yourlabs.runner* is to require as low maintenance as possible.

1.3.1 Admin Emails

When a task started failing every time, i received around 750 emails because it was the weekend. So I've been very carefull to make email notifications throttlable.

Because a failure can come from a code update: the admin is emailed on the first execution failure. The admin also receives an email when a **new** exception is thrown. An exception is **new** if this process hasn't notified the admin about it yet.

After a failure, it will notify the admin after the process downtime is superior to the *non_recoverable_downtime* option. Is is important to set this option according to possible network errors that would cause a task to fail.

If a process is stuck in a failure, then it will notify the admin everytime *non_recoverable_downtime* is reached, to make sure there is an email stuck at the top of his inbox, without spamming it. In practice, 6 or 12 hours is a reasonnable setting for *non_recoverable_downtime*.

1.3.2 Example

An example task, *yourlabs.runner.tasks.divide_by_zero*, is configured for a *fail_cooldown* of 1 second, and a *non_recoverable_downtime* of 3 seconds:

```
>>> ./manage.py run_functions yourlabs.runner.tasks.divide_by_zero
[yourlabs] Could not find your project root, not setting up
[yourlabs] Setting PROJECT_ROOT: /srv/bet.yourlabs.org/main
DEBUG Found pidfile divide_by_zero containing: 13698
DEBUG Could not find /proc/13698, wiping pidfile divide_by_zero
DEBUG Wrote pidfile divide_by_zero
DEBUG [divide_by_zero] Execution failed
DEBUG [divide_by_zero] Sent email to admins: First exception caught: integer division or modulo by ze
DEBUG [divide_by_zero] Sleeping 1 seconds
DEBUG [divide_by_zero] Execution failed
DEBUG [divide_by_zero] Sleeping 1 seconds
DEBUG [divide_by_zero] Execution failed
DEBUG [divide_by_zero] Sleeping 1 seconds
DEBUG [divide_by_zero] Execution failed
DEBUG [divide_by_zero] Sent email to admins: Non recoverable downtime reached
DEBUG [divide_by_zero] Sleeping 1 seconds
DEBUG [divide_by_zero] Execution failed
DEBUG [divide_by_zero] Sleeping 1 seconds
DEBUG [divide_by_zero] Execution failed
DEBUG [divide_by_zero] Sleeping 1 seconds
DEBUG [divide_by_zero] Execution failed
DEBUG [divide_by_zero] Sent email to admins: Non recoverable downtime reached again
DEBUG [divide_by_zero] Sleeping 1 seconds
DEBUG [divide_by_zero] Execution failed
DEBUG [divide_by_zero] Sleeping 1 seconds
DEBUG [divide_by_zero] Execution failed
DEBUG [divide_by_zero] Sleeping 1 seconds
DEBUG [divide_by_zero] Execution failed
DEBUG [divide_by_zero] Sent email to admins: Non recoverable downtime reached again
DEBUG [divide_by_zero] Sleeping 1 seconds
DEBUG [divide_by_zero] Execution failed
DEBUG [divide_by_zero] Sleeping 1 seconds
DEBUG [divide_by_zero] Execution failed
DEBUG [divide_by_zero] Sleeping 1 seconds
```

1.3.3 Concurrency handling

Each runner will create a pidfile in *RUN_ROOT*, for example *PROJECT_ROOT/var/run/send_mail_retry_deferred.pid* for *run_functions tasks.send_mail tasks.retry_deferred* if *RUN_ROOT* is set to *PROJECT_ROOT + '/var/run/*

The runner doesn't even attempt to delete its pidfile on exit. It keeps in mind that a dead pidfile might be left for example after a power outage.

When a runner starts, it checks if a pidfile exists and unless option *killconcurrent* is set to False, it will attempt to kill the existing process if any. Anyway, it will delete and re-create the pidfile with the actual pid.

This is implemented in the *runner.Runner.concurrency_security* method.

Danger: If a concurrent runner checks for the pidfile **before** the other one writes it, then it will result in concurrent processes which has no pidfile.

1.3.4 Upgrading processes

Starting the same queues again and waiting a few seconds results in a process upgrade, a feature from concurrency handling. The queues will naturally be replaced by the new code (from your tasks or in runner itself).

Example process upgrade using a shell script:

```
<<< 22:50.31 Sun Sep 11 2011!~bet_prod/main
<<< root@tina!12456 E:130 S:1 G:master bet_prod_env
>>> source ../local && start_runner
Starting run_functions tasks.gsm_sync tasks.update_index
Starting run_functions tasks.gsm_sync_live
Starting run_functions tasks.send_mail tasks.retry_deferred
<<< 22:50.33 Sun Sep 11 2011!~bet_prod/main
<<< root@tina!12462 S:1 G:master bet_prod_env
>>> ps aux | grep run_functions
bet_prod 24499  2.3  1.2  33744 25644 pts/3    SN   22:46   0:05 python /srv/bet_prod/main/manage.py
bet_prod 24502  7.5  1.2  34128 26092 pts/3    SN   22:46   0:18 python /srv/bet_prod/main/manage.py
bet_prod 24505  0.7  1.2  32568 24412 pts/3    SN   22:46   0:01 python /srv/bet_prod/main/manage.py
bet_prod 24626 18.0  0.3  12328  7072 pts/3    RN   22:50   0:00 python /srv/bet_prod/main/manage.py
bet_prod 24629 57.0  0.6  17536 12380 pts/3    RN   22:50   0:00 python /srv/bet_prod/main/manage.py
bet_prod 24632  2.0  0.1   6624  2920 pts/3    RN   22:50   0:00 python /srv/bet_prod/main/manage.py
root     24639  0.0  0.0   4408   836 pts/3    S+   22:50   0:00 grep run_functions
<<< 22:50.34 Sun Sep 11 2011!~bet_prod/main
<<< root@tina!12463 S:1 G:master bet_prod_env
>>> ps aux | grep run_functions
bet_prod 24626 15.1  1.2  32868 24808 pts/3    RN   22:50   0:02 python /srv/bet_prod/main/manage.py
bet_prod 24629 17.6  1.2  33804 25876 pts/3    SN   22:50   0:02 python /srv/bet_prod/main/manage.py
bet_prod 24632 13.8  1.2  32564 24412 pts/3    SN   22:50   0:01 python /srv/bet_prod/main/manage.py
root     24663  0.0  0.0   4408   836 pts/3    S+   22:50   0:00 grep run_functions
```

1.4 Historical context

A project needs to continuously run tasks (duh!). Several chains of API calls must to be done with different intervals, to ensure a balance between data freshness and performance. Needless to say, this is a mission critical task.

The first attempt was using threads but it turned out everything had to be done to have sane monitoring. First i implemented exception handling in one task. Then, refactored it to use it in another task.

runner.Runner was born. However, it did not make sense to carry the weight of thread management anymore. The command run_functions was born. It really looked handy and it was a sunny day so it was open sourced.

CHAPTER

TWO

APPLICATIONS

yourlabs.runner It is frequent for projects to need commands to be executed continuously. When cron or spoolers aren't the way to go, runner provides a simple way to create background threads which chains commands continuously. For what it's worth, it's partly documented.

yourlabs.smoke High level tests, like testing if a view returns status 200, are called "smoke tests". This application makes creating complete smoke tests for all possible urls in your project easy. It's not really ready for a end user.

yourlabs.setup Helps moving away the boilerplate from a django project's settings.py. We use it but it's not documented.

CHAPTER

THREE

INSTALL

There is one small repo for all apps, which is install easy to install:

```
pip install -e git+git@github.com:yourlabs/yourlabs.git#egg=yourlabs
```

This will clone the repo in *your-python-env/src/yourlabs* and install the yourlabs module. You can then install any application you like. For example, install the "runner" applications by adding to *settings.INSTALLED_APPS*: *'yourlabs.runner'*.

Note: you can hack directly in that repo.

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*