Taskfarm Documentation

Release 0.0

Magnus Hagdorn

Jan 19, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

The taskfarm is a server-client application that tracks tasks to be completed. The server is provides a REST API to create and update runs. This is the python server documentation.

This package solves the problem of managing a loosely coupled taskfarm where there are many tasks and the workers are entirely independent of each other. Instead of using a farmer process a database is used to hand out new tasks to the workers. The workers contact a web application via http(s) to get a new task.

You can use the taskfarm-worker python package to connect to the taskfarm service.

The server is implemented using the flask web framework and uses flask-sqlalchemy to connect to a relational database.

CHAPTER

TWO

INSTALLATION

2.1 Local Installation

You can install the package from source after cloning the repository

```
git clone https://github.com/mhagdorn/taskfarm.git
cd taskfarm
python3 setup.py install
```

or simply using `pip`

```
pip install taskfarm
```

2.1.1 Setup

After installing the python package you need to connect to a database. For testing purposes you can use sqlite. However, sqlite does not allow row locking so if you use parallel workers a single task may get assigned to multiple workers. For production use you should use a postgres database instead.

You can set the environment variable `DATABASE_URL` to configure the database connection (see the SQLAlchemy documentation). For example

```
export DATABASE_URL=sqlite:///app.db
```

or

```
export DATABASE_URL=postgresql://user:pw@host/db
```

You then need to create the tables by running

```
adminTF --init-db
```

You can then create some users

```
adminTF -u some_user -p some_password
```

These users are used by the worker to connect to the service.

2.1.2 Running the Taskfarm Server

The taskfarm server is a flask web application. For testing you can run it locally using

```
export FLASK_APP=taskfarm
flask run
```

You can check the service is running by browsing to http://localhost:5000/ or running

```
curl http://localhost:5000/
```

For a production setup you need to deploy the flask application using a WSGI server such as gunicorn. The flask documentation lists the various options for self-hosting or hosting in the cloud a flask application.

2.2 Containerised Installation

Instead of installing the taskfarm server locally and managing the flask webapplication service you can run the taskfarm server as a containerised service. You need a working docker setup and docker compose. The taskfarm service is built using Ubuntu containers, one for the web application, one for the postgres database and one for the web server. You can build and start the containers using

```
docker-compose build
```

You need to initialise the database and create a user, ie

```
docker-compose run web  adminTF --init-db
docker-compose run web  adminTF -u taskfarm -p hello
```

You can now start the service

```
docker-compose up -d
```

and you can reach the taskfarm server on port 80. You can check the service is running by browsing to http://localhost/ or running

```
curl http://localhost/
```

CHAPTER

THREE

TASKFARM REST API

Resource	Operation	Description
	GET /	
run	*POST /api/run*	create a new run
runs	*GET /api/runs*	get a list of all runs
	POST /api/runs/(string:uuid)/restart	restart all tasks of a run
	GET /api/runs/(string:uuid)/tasks/(int:taskID)	information about a particular task
	PUT /api/runs/(string:uuid)/tasks/(int:taskID)	update a particular task
	POST /api/runs/(string:uuid)/task	request a task for run
	GET /api/runs/(string:uuid)	get information about a particular run
	DELETE /api/runs/(string:uuid)	delete a particular run
token	*GET /api/token*	get the authentication token
worker	*POST /api/worker*	create a worker

GET /

print info about taskfarm server

POST /api/run

create a new run

Request JSON Object

- **numTasks** (*int*) – the number of tasks of the run

Response JSON Object

- **id** (*int*) – run ID
- **uuid** (*string*) – run UUID
- **numTasks** (*int*) – the number of tasks

Status Codes

- 400 Bad Request – when numTask is missing

GET /api/runs

get a list of all runs

Response JSON Array of Objects

- **id** (*int*) – run ID
- **uuid** (*string*) – run UUID
- **numTasks** (*int*) – the number of tasks

`DELETE /api/runs/(string:` *uuid*`)`

delete a particular run

Parameters

- **uuid** (*string*) – uuid of the run

Status Codes

- 404 Not Found – when the run does not exist
- 204 No Content – when the run was successfully deleted

`GET /api/runs/(string:` *uuid*`)`

get information about a particular run

Parameters

- **uuid** (*string*) – uuid of the run

Query Parameters

- **info** – request particular information about the run.

Status Codes

- 404 Not Found – when the run does not exist
- 404 Not Found – when unkown information is requested
- 200 OK – the call successfully returned a json string

The `info` query parameter can be one of `percentDone`, `numWaiting`, `numDone`, `numComputing` to get particular information of the run. By default `info` is the empty string and call returns a json object containing all those pieces of information.

`POST /api/runs/(string:` *uuid*`)/restart`

restart all tasks of run

Parameters

- **uuid** (*string*) – uuid of the run

Query Parameters

- **all** (*string*) – can be True/False (default). When set to to True restart all tasks otherwise restart only partially completed tasks

Status Codes

- 400 Bad Request – when run with uuid does not exist
- 404 Not Found – when parameter all has wrong value
- 204 No Content – success

`POST /api/runs/(string:` *uuid*`)/task`

request a task for run

Parameters

- **uuid** (*string*) – uuid of the run

Request JSON Object

- **worker_uuid** (*string*) – uuid of worker requesting a task

Response JSON Object

- **id** (*int*) – task ID
- **task** (*int*) – task number
- **percentCompleted** (*float*) – percentage compelted of task
- **status** (*string*) – task status, can be one of `waiting`, `computing`, `done`

Status Codes

- 400 Bad Request – when worker_uuid is not present
- 404 Not Found – when worker does not exist
- 404 Not Found – when run does not exist
- 204 No Content – no more tasks
- 201 Created – new tasks

GET /api/runs/(string: *uuid*)/tasks/(int: *taskID*)

get information about a particular task

Parameters

- **uuid** (*string*) – uuid of the run
- **taskID** (*int*) – the task's ID

Query Parameters

- **info** – request particular information about the task

Status Codes

- 404 Not Found – when the run does not exist
- 404 Not Found – when the taskID < 0 or when taskID is larger than the number of tasks
- 404 Not Found – when unkown information is requeste
- 200 OK – the call successfully returned a json string

The `info` query parameter can be one of `status` or `percentDone` to get particular information of the task. By default `info` is the empty string and call returns a json object containing all those pieces of information.

PUT /api/runs/(string: *uuid*)/tasks/(int: *taskID*)

update a particular task

Parameters

- **uuid** (*string*) – uuid of the run
- **taskID** (*int*) – the task's ID

Request JSON Object

- **percentCompleted** (*float*) – percentage of task completed
- **status** (*string*) – status of task, can be `waiting`, `computing`, `done`

Status Codes

- 400 Bad Request – an error occurred updating the task
- 204 No Content – the task was successfully updated

`GET /api/token`

get the authentication token

Response JSON Object

- **token** (*string*) – the authentication token

`POST /api/worker`

create a worker

Request JSON Object

- **uuid** (*string*) – worker uuid
- **hostname** (*string*) – hostname where worker is running
- **pid** (*int*) – process ID (PID) of worker

Response JSON Object

- **uuid** (*string*) – worker uuid
- **id** (*int*) – worker ID

Status Codes

- 400 Bad Request – when input json is incomplete
- 201 Created – when worker was created successfully

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

HTTP ROUTING TABLE

/

/api