serpent_tracker

Brian Hopkins

Mar 23, 2020

CONTENTS:

CHAPTER

ONE

GETTING UP AND RUNNING LOCALLY

1.1 Setting Up Development Environment

Make sure to have the following on your host:

- Python 3.7
- PostgreSQL.
- Redis, if using Celery

First things first.

1. Create a virtualenv:

```
$ python3.7 -m venv <virtual env path>
```

2. Activate the virtualenv you have just created:

```
$ source <virtual env path>/bin/activate
```

3. Install development requirements:

```
$ pip install -r requirements/local.txt
$ pre-commit install

 .. note::

    the `pre-commit` exists in the generated project as default.
    for the details of `pre-commit`, follow the [site of pre-commit](https://pre-
↪commit.com/).
```

4. Create a new PostgreSQL database using createdb:

```
$ createdb <what you have entered as the project_slug at setup stage> -U postgres␣
↪--password <password>
```

Note: if this is the first time a database is created on your machine you might need an initial PostgreSQL set up to allow local connections & set a password for the `postgres` user. The postgres documentation explains the syntax of the config file that you need to change.

5. Set the environment variables for your database(s):

```
$ export DATABASE_URL=postgres://postgres:<password>@127.0.0.1:5432/<DB name␣
↪given to createdb>
# Optional: set broker URL if using Celery
$ export CELERY_BROKER_URL=redis://localhost:6379/0
```

Note: Check out the *Settings* page for a comprehensive list of the environments variables.

See also:

To help setting up your environment variables, you have a few options:

- create an `.env` file in the root of your project and define all the variables you need in it. Then you just need to have `DJANGO_READ_DOT_ENV_FILE=True` in your machine and all the variables will be read.
- Use a local environment manager like direnv

6. Apply migrations:

```
$ python manage.py migrate
```

7. See the application being served through Django development server:

```
$ python manage.py runserver 0.0.0.0:8000
```

1.2 Setup Email Backend

1.2.1 MailHog

Note: In order for the project to support *MailHog* it must have been bootstrapped with `use_mailhog` set to `y`.

MailHog is used to receive emails during development, it is written in Go and has no external dependencies.

For instance, one of the packages we depend upon, `django-allauth` sends verification emails to new users signing up as well as to the existing ones who have not yet verified themselves.

1. Download the latest MailHog release for your OS.
2. Rename the build to `MailHog`.
3. Copy the file to the project root.
4. Make it executable:

```
$ chmod +x MailHog
```

5. Spin up another terminal window and start it there:

```
./MailHog
```

6. Check out http://127.0.0.1:8025/ to see how it goes.

Now you have your own mail server running locally, ready to receive whatever you send it.

1.2.2 Console

Note: If you have generated your project with `use_mailhog` set to `n` this will be a default setup.

Alternatively, deliver emails over console via `EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'`.

In production, we have Mailgun configured to have your back!

1.3 Celery

If the project is configured to use Celery as a task scheduler then by default tasks are set to run on the main thread when developing locally. If you have the appropriate setup on your local machine then set the following in `config/settings/local.py`:

```
CELERY_TASK_ALWAYS_EAGER = False
```

1.4 Sass Compilation & Live Reloading

If you'd like to take advantage of live reloading and Sass compilation you can do so with a little bit of preparation, see sass-compilation-live-reload.

1.5 Summary

Congratulations, you have made it! Keep on reading to unleash full potential of Cookiecutter Django.

CHAPTER

TWO

GETTING UP AND RUNNING LOCALLY WITH DOCKER

The steps below will get you up and running with a local development environment. All of these commands assume you are in the root of your generated project.

Note: If you're new to Docker, please be aware that some resources are cached system-wide and might reappear if you generate a project multiple times with the same name (e.g. this issue with Postgres).

2.1 Prerequisites

- Docker; if you don't have it yet, follow the installation instructions;
- Docker Compose; refer to the official documentation for the installation guide.

2.2 Build the Stack

This can take a while, especially the first time you run this particular command on your development system:

```
$ docker-compose -f local.yml build
```

Generally, if you want to emulate production environment use `production.yml` instead. And this is true for any other actions you might need to perform: whenever a switch is required, just do it!

2.3 Run the Stack

This brings up both Django and PostgreSQL. The first time it is run it might take a while to get started, but subsequent runs will occur quickly.

Open a terminal at the project root and run the following for local development:

```
$ docker-compose -f local.yml up
```

You can also set the environment variable `COMPOSE_FILE` pointing to `local.yml` like this:

```
$ export COMPOSE_FILE=local.yml
```

And then run:

```
$ docker-compose up
```

To run in a detached (background) mode, just:

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

2.4 Execute Management Commands

As with any shell command that we wish to run in our container, this is done using the `docker-compose -f local.yml run --rm` command:

```
$ docker-compose -f local.yml run --rm django python manage.py migrate
$ docker-compose -f local.yml run --rm django python manage.py createsuperuser
```

Here, `django` is the target service we are executing the commands against.

2.5 (Optionally) Designate your Docker Development Server IP

When `DEBUG` is set to `True`, the host is validated against `['localhost', '127.0.0.1', '[::1]']`. This is adequate when running a `virtualenv`. For Docker, in the `config.settings.local`, add your host development server IP to `INTERNAL_IPS` or `ALLOWED_HOSTS` if the variable exists.

2.6 Configuring the Environment

This is the excerpt from your project's `local.yml`:

```
# ...

postgres:
  build:
    context: .
    dockerfile: ./compose/production/postgres/Dockerfile
  volumes:
    - local_postgres_data:/var/lib/postgresql/data
    - local_postgres_data_backups:/backups
  env_file:
    - ./.envs/.local/.postgres

# ...
```

The most important thing for us here now is `env_file` section enlisting `./.envs/.local/.postgres`. Generally, the stack's behavior is governed by a number of environment variables (*env(s)*, for short) residing in `envs/`, for instance, this is what we generate for you:

```
.envs
├── .local
│   ├── .django
│   └── .postgres
└── .production
```

(continues on next page)

(continued from previous page)

```
    ├── .django
    └── .postgres
```

By convention, for any service `sI` in environment `e` (you know `someenv` is an environment when there is a `someenv.yml` file in the project root), given `sI` requires configuration, a `.envs/.e/.sI` *service configuration* file exists.

Consider the aforementioned `.envs/.local/.postgres`:

```
# PostgreSQL
# ------------------------------------------------------------------------------
POSTGRES_HOST=postgres
POSTGRES_DB=<your project slug>
POSTGRES_USER=XgOWtQtJecsAbaIyslwGvFvPawftNaqO
POSTGRES_PASSWORD=jSljDz4whHuwO3aJIgVBrqEml5Ycbghorep4uVJ4xjDYQu0LfuTZdctj7y0YcCLu
```

The three envs we are presented with here are `POSTGRES_DB`, `POSTGRES_USER`, and `POSTGRES_PASSWORD` (by the way, their values have also been generated for you). You might have figured out already where these definitions will end up; it's all the same with `django` service container envs.

One final touch: should you ever need to merge `.envs/production/*` in a single `.env` run the `merge_production_dotenvs_in_dotenv.py`:

```
$ python merge_production_dotenvs_in_dotenv.py
```

The `.env` file will then be created, with all your production envs residing beside each other.

2.7 Tips & Tricks

2.7.1 Activate a Docker Machine

This tells our computer that all future commands are specifically for the dev1 machine. Using the `eval` command we can switch machines as needed.:

```
$ eval "$(docker-machine env dev1)"
```

2.7.2 Debugging

ipdb

If you are using the following within your code to debug:

```
import ipdb; ipdb.set_trace()
```

Then you may need to run the following for it to work as desired:

```
$ docker-compose -f local.yml run --rm --service-ports django
```

django-debug-toolbar

In order for `django-debug-toolbar` to work designate your Docker Machine IP with `INTERNAL_IPS` in `local.py`.

2.7.3 Mailhog

When developing locally you can go with MailHog for email testing provided `use_mailhog` was set to `y` on setup. To proceed,

1. make sure `mailhog` container is up and running;
2. open up `http://127.0.0.1:8025`.

2.7.4 Celery tasks in local development

When not using docker Celery tasks are set to run in Eager mode, so that a full stack is not needed. When using docker the task scheduler will be used by default.

If you need tasks to be executed on the main thread during development set CELERY_TASK_ALWAYS_EAGER = True in config/settings/local.py.

Possible uses could be for testing, or ease of profiling with DJDT.

2.7.5 Celery Flower

Flower is a "real-time monitor and web admin for Celery distributed task queue".

Prerequisites:

- `use_docker` was set to `y` on project initialization;
- `use_celery` was set to `y` on project initialization.

By default, it's enabled both in local and production environments (`local.yml` and `production.yml` Docker Compose configs, respectively) through a `flower` service. For added security, `flower` requires its clients to provide authentication credentials specified as the corresponding environments' `.envs/.local/.django` and `.envs/.production/.django` `CELERY_FLOWER_USER` and `CELERY_FLOWER_PASSWORD` environment variables. Check out `localhost:5555` and see for yourself.

CHAPTER

THREE

SETTINGS

This project relies extensively on environment settings which **will not work with Apache/mod_wsgi setups**. It has been deployed successfully with both Gunicorn/Nginx and even uWSGI/Nginx.

For configuration purposes, the following table maps environment variables to their Django setting and project settings:

Environment Variable	Django Setting	Development Default	Production Default
DJANGO_READ_DOT_ENV_FILE	READ_DOT_ENV_FILE	False	False

Environment Variable	Django Setting	Development Default	Production Default
DATABASE_URL	DATABASES	auto w/ Docker; postgres://project_slug w/o	raises error
DJANGO_ADMIN_URL	n/a	'admin/'	raises error
DJANGO_DEBUG	DEBUG	True	False
DJANGO_SECRET_KEY	SECRET_KEY	auto-generated	raises error
DJANGO_SECURE_BROWSER_XSS_FILTER	SECURE_BROWSER_XSS_FILTER	n/a	True
DJANGO_SECURE_SSL_REDIRECT	SECURE_SSL_REDIRECT	n/a	True
DJANGO_SECURE_CONTENT_TYPE_NOSNIFF	SECURE_CONTENT_TYPE_NOSNIFF	n/a	True
DJANGO_SECURE_FRAME_DENY	SECURE_FRAME_DENY	n/a	True
DJANGO_SECURE_HSTS_INCLUDE_SUBDOMAINS	SECURE_HSTS_INCLUDE_SUBDOMAINS	n/a	True
DJANGO_SESSION_COOKIE_HTTPONLY	SESSION_COOKIE_HTTPONLY	n/a	True
DJANGO_SESSION_COOKIE_SECURE	SESSION_COOKIE_SECURE	n/a	False
DJANGO_DEFAULT_FROM_EMAIL	DEFAULT_FROM_EMAIL	n/a	"your_project_name <noreply@your_domain_name>"
DJANGO_SERVER_EMAIL	SERVER_EMAIL	n/a	"your_project_name <noreply@your_domain_name>"
DJANGO_EMAIL_SUBJECT_PREFIX	EMAIL_SUBJECT_PREFIX	n/a	"[your_project_name] "
DJANGO_ALLOWED_HOSTS	ALLOWED_HOSTS	['*']	['your_domain_name']

The following table lists settings and their defaults for third-party applications, which may or may not be part of your project:

Environment Variable	Django Setting	Development Default	Production Default
CELERY_BROKER_URL	CELERY_BROKER_URL	auto w/ Docker; raises error w/o	raises error
DJANGO_AWS_ACCESS_KEY_ID	AWS_ACCESS_KEY_ID	n/a	raises error
DJANGO_AWS_SECRET_ACCESS_KEY	AWS_SECRET_ACCESS_KEY	n/a	raises error
DJANGO_AWS_STORAGE_BUCKET_NAME	AWS_STORAGE_BUCKET_NAME	n/a	raises error
DJANGO_AWS_S3_REGION_NAME	AWS_S3_REGION_NAME	n/a	None
DJANGO_GCP_STORAGE_BUCKET_NAME	GS_BUCKET_NAME	n/a	raises error
GOOGLE_APPLICATION_CREDENTIALS	n/a	n/a	raises error
SENTRY_DSN	SENTRY_DSN	n/a	raises error
DJANGO_SENTRY_LOG_LEVEL	SENTRY_LOG_LEVEL	n/a	logging.INFO
MAILGUN_API_KEY	MAILGUN_API_KEY	n/a	raises error
MAILGUN_DOMAIN	MAILGUN_SENDER_DOMAIN	n/a	raises error
MAILGUN_API_URL	n/a	n/a	"https://api.mailgun.net/v3"

3.1 Other Environment Settings

DJANGO_ACCOUNT_ALLOW_REGISTRATION (=True) Allow enable or disable user registration through *django-allauth* without disabling other characteristics like authentication and account management. (Django Setting: ACCOUNT_ALLOW_REGISTRATION)

CHAPTER

FOUR

TESTING

We encourage users to build application tests. As best practice, this should be done immediately after documentation of the application being built, before starting on any coding.

4.1 Pytest

This project uses the Pytest, a framework for easily building simple and scalable tests. After you have set up to develop locally, run the following commands to make sure the testing environment is ready:

```
$ pytest
```

You will get a readout of the *users* app that has already been set up with tests. If you do not want to run the *pytest* on the entire project, you can target a particular app by typing in its location:

```
$ pytest <path-to-app-in-project/app>
```

If you set up your project to develop locally with docker, run the following command:

```
$ docker-compose -f local.yml run --rm django pytest
```

Targeting particular apps for testing in `docker` follows a similar pattern as previously shown above.

4.2 Coverage

You should build your tests to provide the highest level of **code coverage**. You can run the `pytest` with code `coverage` by typing in the following command:

```
$ docker-compose -f local.yml run --rm django coverage run -m pytest
```

Once the tests are complete, in order to see the code coverage, run the following command:

```
$ docker-compose -f local.yml run --rm django coverage report
```

Note: At the root of the project folder, you will find the *pytest.ini* file. You can use this to customize the `pytest` to your liking.

There is also the *.coveragerc*. This is the configuration file for the `coverage` tool. You can find out more about configuring `coverage`.

See also:

For unit tests, run:

```
$ python manage.py test
```

Since this is a fresh install, and there are no tests built using the Python unittest library yet, you should get feedback that says there were no tests carried out.

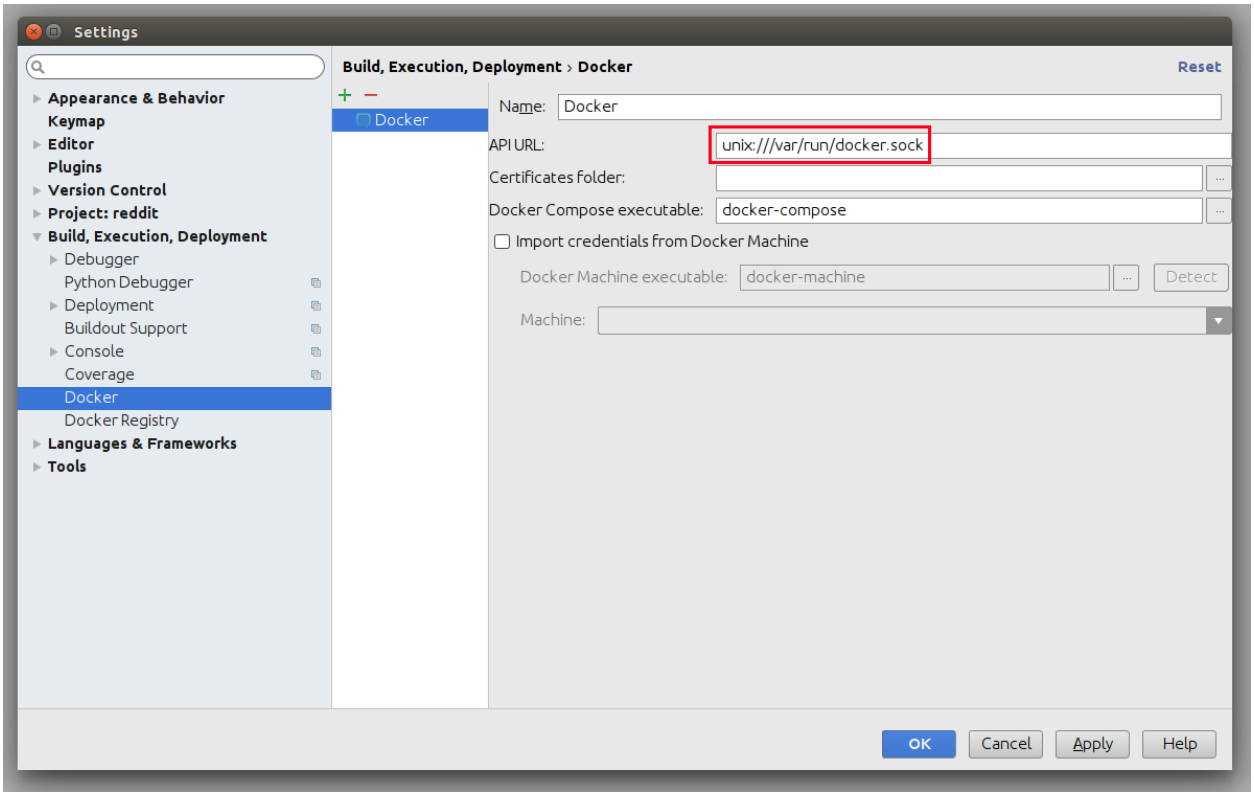
CHAPTER

FIVE

DOCKER REMOTE DEBUGGING

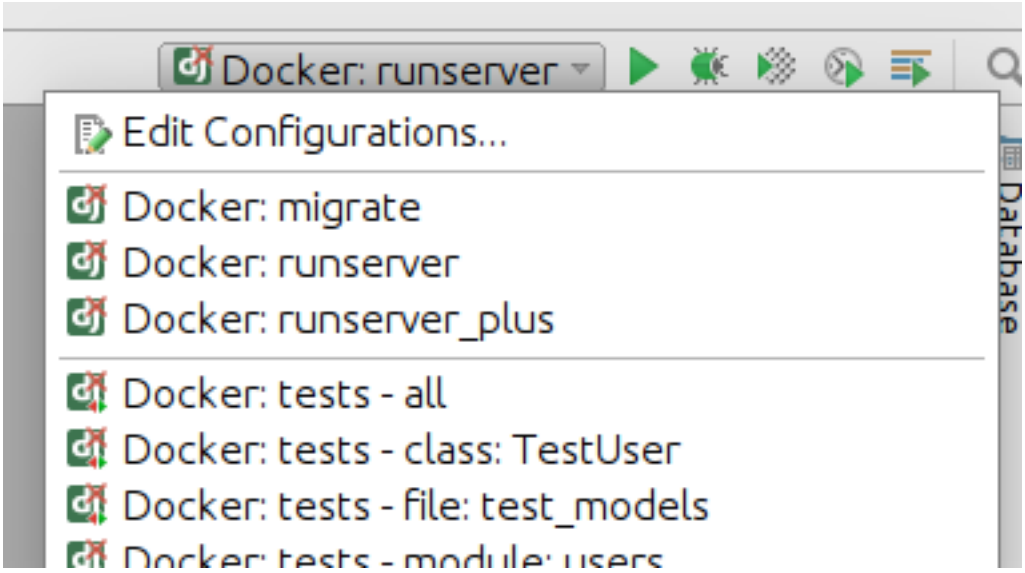
To connect to python remote interpreter inside docker, you have to make sure first, that Pycharm is aware of your docker.

Go to *Settings* > *Build, Execution, Deployment* > *Docker*. If you are on linux, you can use docker directly using its socket *unix:///var/run/docker.sock*, if you are on Windows or Mac, make sure that you have docker-machine installed, then you can simply *Import credentials from Docker Machine*.



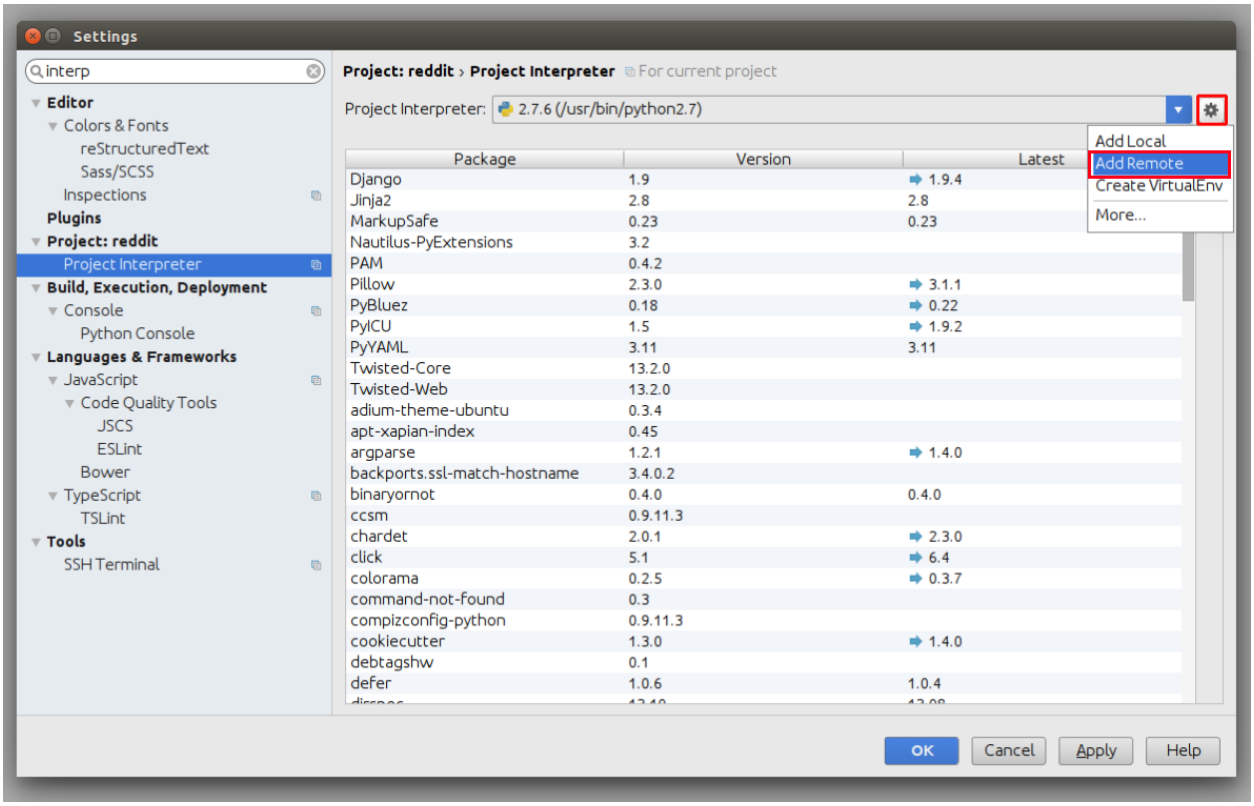
5.1 Configure Remote Python Interpreter

This repository comes with already prepared "Run/Debug Configurations" for docker.

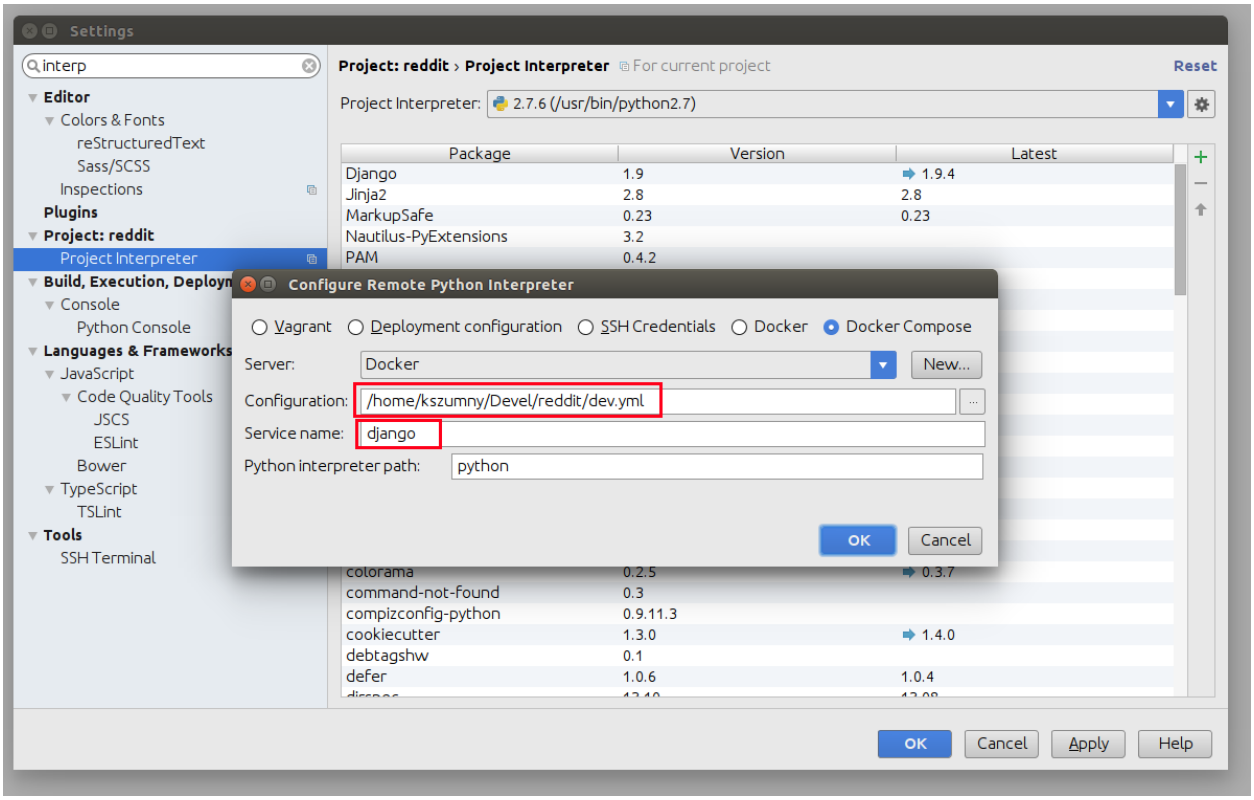


But as you can see, at the beginning there is something wrong with them. They have red X on django icon, and they cannot be used, without configuring remote python interpteter. To do that, you have to go to *Settings* > *Build, Execution, Deployment* first.

Next, you have to add new remote python interpreter, based on already tested deployment settings. Go to *Settings* > *Project* > *Project Interpreter*. Click on the cog icon, and click *Add Remote*.

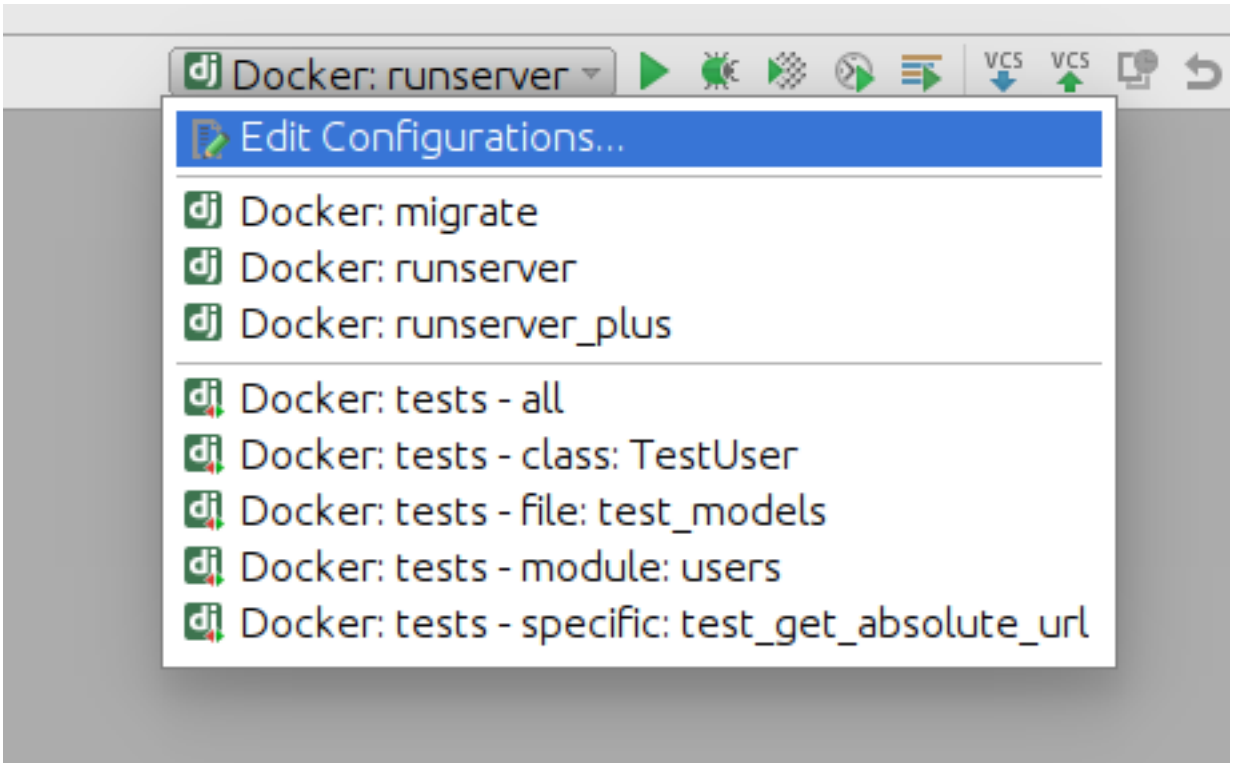


Switch to *Docker Compose* and select *local.yml* file from directory of your project, next set *Service name* to *django*



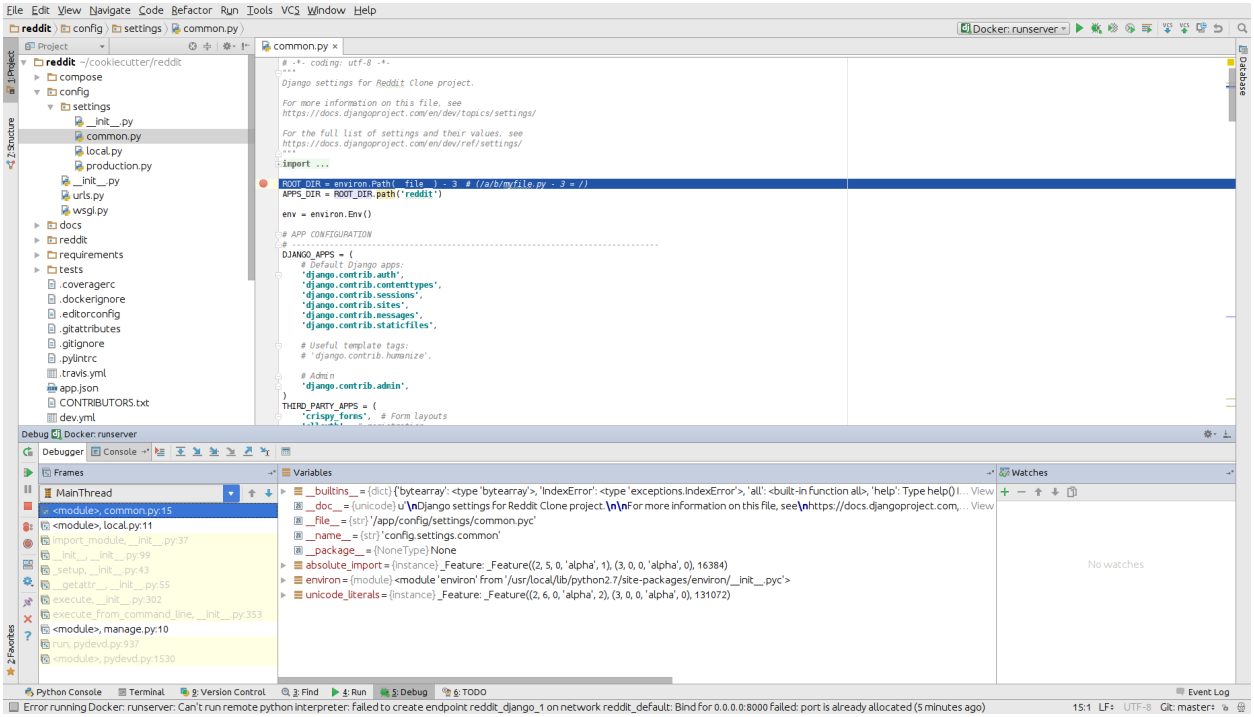
Having that, click *OK*. Close *Settings* panel, and wait few seconds...

After few seconds, all *Run/Debug Configurations* should be ready to use.

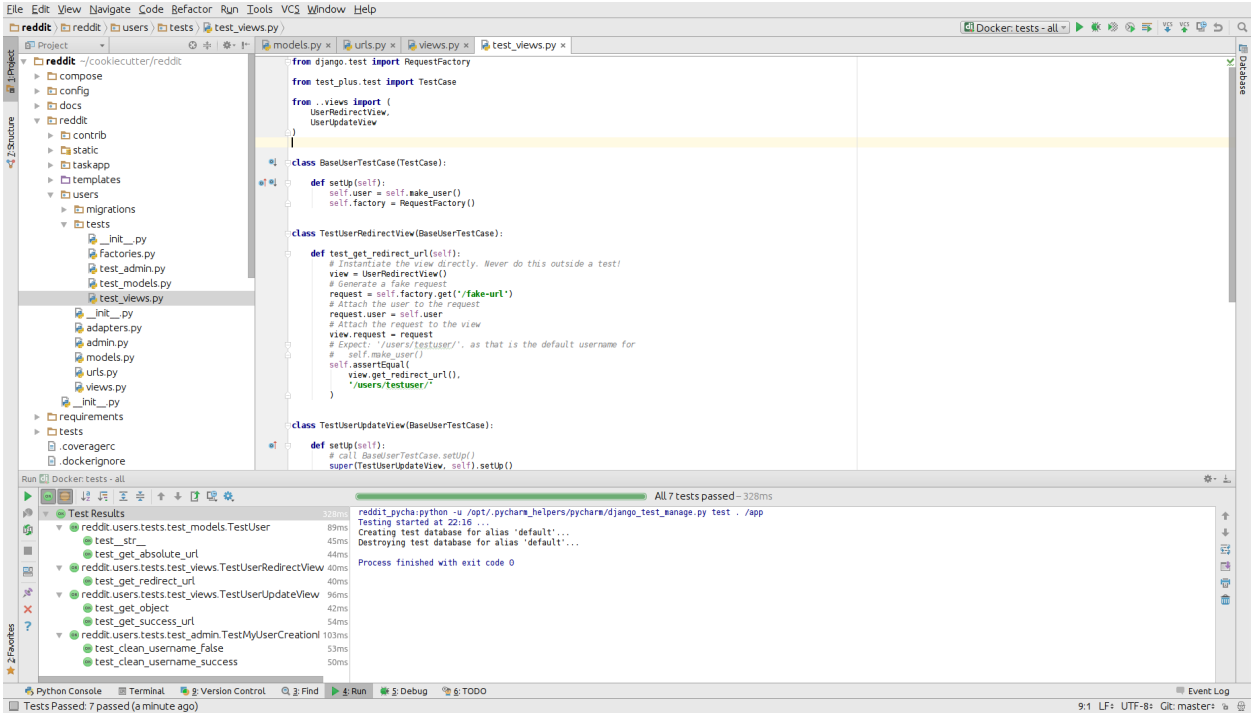


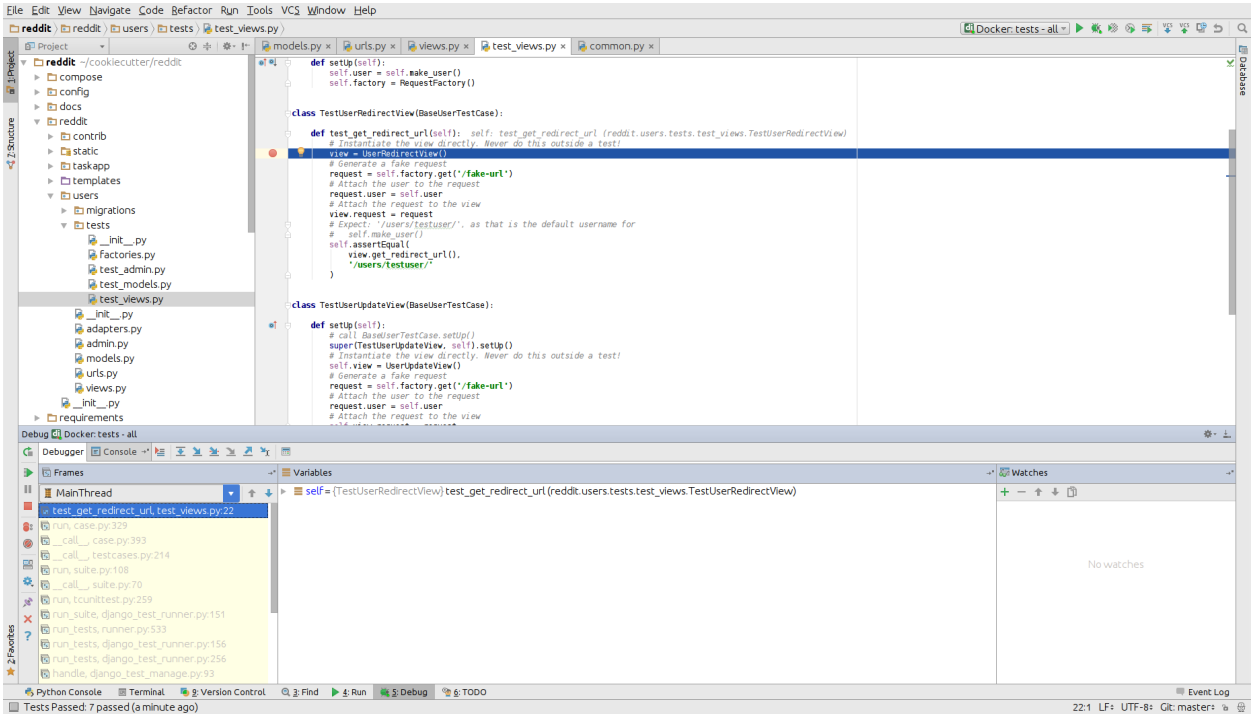
Things you can do with provided configuration:

- run and debug python code

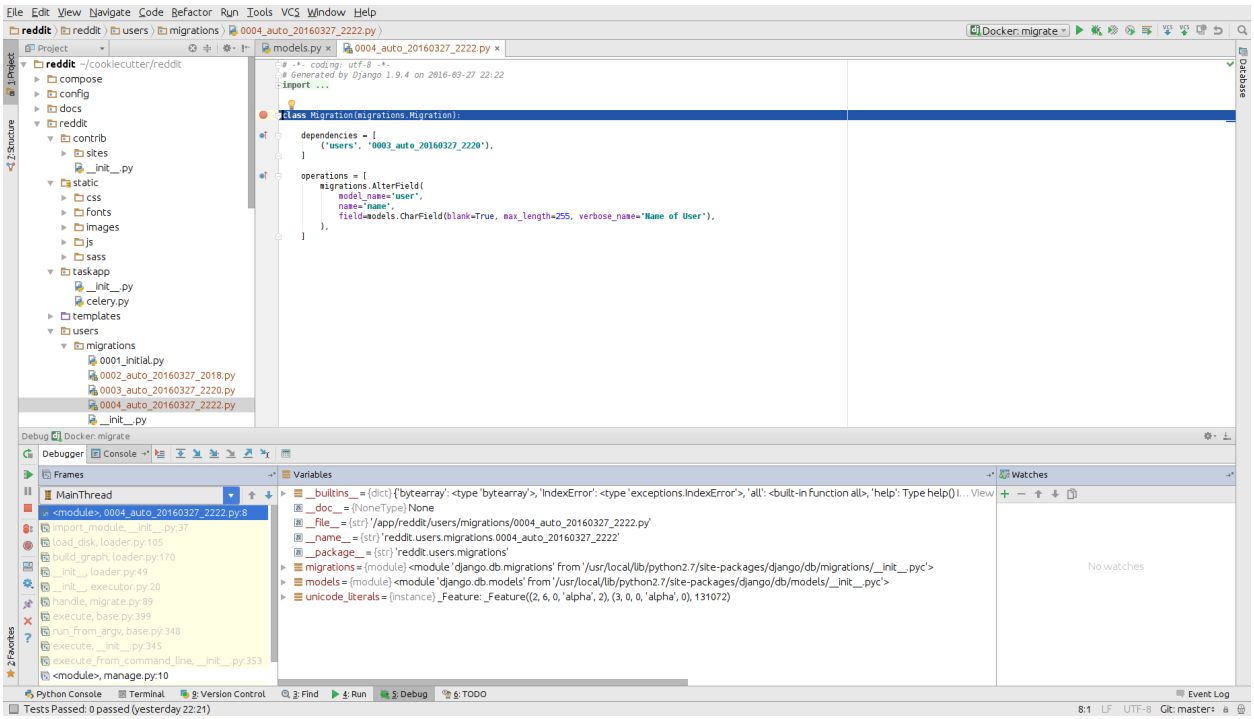


- run and debug tests





- run and debug migrations or different django management commands



- and many others..

5.2 Known issues

- Pycharm hangs on “Connecting to Debugger”

This might be fault of your firewall. Take a look on this ticket - https://youtrack.jetbrains.com/issue/PY-18913

- Modified files in *.idea* directory

Most of the files from *.idea/* were added to *.gitignore* with a few exceptions, which were made, to provide “ready to go” configuration. After adding remote interpreter some of these files are altered by PyCharm:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   .idea/project_name.iml

no changes added to commit (use "git add" and/or "git commit -a")
```

In theory you can remove them from repository, but then, other people will lose a ability to initialize a project from provided configurations as you did. To get rid of this annoying state, you can run command:

```
$ git update-index --assume-unchanged serpent_tracker.iml
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

INDEX