
But I never wanted to do DevOps!

Mar 20, 2020

Contents

1	Who is this for?	3
2	Before you start	5
3	Workshop steps	7
4	Further reading	29

You can be a Python web programmer and not have to do anything with DevOps, by making use of containerisation technology and Platform-as-a-service systems.

This hands-on workshop will expand your repertoire and put new components in your developer's toolbox - and will help ensure you never find yourself having to answer to a pager in the middle of the night.

The examples and exercises in the workshop will be based on a workflow using:

- [Docker](#), with the
- [Divio Cloud platform](#) and
- [Amazon Web Services](#).

We will also work with:

- [Django](#)
- Django reusable applications including [Django Axes](#) and [Django Photologue](#)
- [LogDNA](#) and [Azure cognitive services](#)
- system-level libraries for image handling and processing

... and other tools, but it's about **skills**, not **tools** - and *all the skills are transferable*. The purpose of the workshop is to help you become familiar with a mindset and way of working that:

- **allows you to stop worrying about servers and deployment**
- **... by making use of services that take care of that for you.**

CHAPTER 1

Who is this for?

This workshop has been written for the Python developer who likes programming, but would like to give the responsibility for configuration and maintenance of hardware, operating systems, databases, web servers and other important things that a website needs to someone else.

Containerisation is an excellent way to do that - but then you have to learn some new tricks, and abandon some old habits and ways of working. This workshop is intended to ease you along that path.

It's what I wish I had discovered several years ago, when I was exactly in that position.

2.1 Prerequisites

2.1.1 Required software

You will need to have the following installed or configured, and know at least the basics of using them, before proceeding:

- Git ([GitHub's set up Git guide](#))
- SSH, so that you can provide your public key to a server ([GitHub's guides to setting up SSH](#))

2.1.2 Hardware requirements

Please check the Docker hardware requirements below. It *is* possible to use older hardware and software, using Virtual Box and Docker Toolbox - but it can require quite a bit of extra work to set up.

Macintosh users

- a 2010 or newer model. Run the command `sysctl kern.hv_support` to check for the required hardware virtualisation support (the response should be 1).
- macOS Sierra 10.12 or newer

Windows users

- virtualisation enabled in BIOS
- Windows 10 64bit Pro, Enterprise or Education

Linux users 

You'll probably be OK. . .

CHAPTER 3

Workshop steps

The pages in this section cover what you will do. In this workshop, you will learn by doing. Understanding always comes after doing. I will not explain much in this section, but the exercises are designed to help you discover key principles. Occasionally, links will be provided to *Further reading* sections. These are optional, and it's recommended to use them *after* you have successfully *done* the relevant tasks.

3.1 The workshop

Note: Before starting, please see *Before you start*.

3.1.1 Installation

You will need to install:

Docker and Docker Compose

- Macintosh users: [Docker for Mac](#)
- Windows users: [Docker for Windows](#)
- Linux users: [Docker CE](#) and [Docker Compose](#)

The Divio CLI

(You can install the Divio CLI in a virtual environment if you prefer.)

- the [Divio CLI](#): `pip install divio-cli`

3.1.2 Create a project to work with

Note: Before starting, please see *Before you start*.

Using the Divio Control Panel (you will need an account, it's free), [create a new project](#).

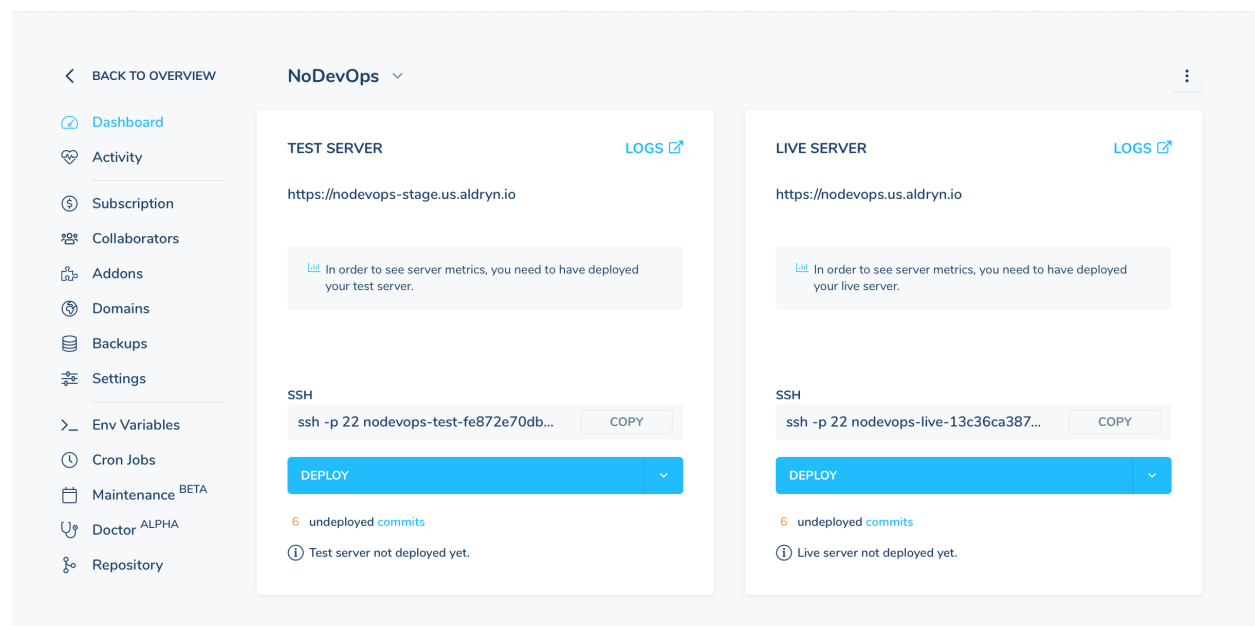
Select the defaults:

- *Platform:* Python 3.6
- *Type:* Django

You can use Divio's Git server (also the default).

Hit **Skip** in the *Subscription* view.

After a few moments, you'll be presented with its dashboard, which will look something like this:

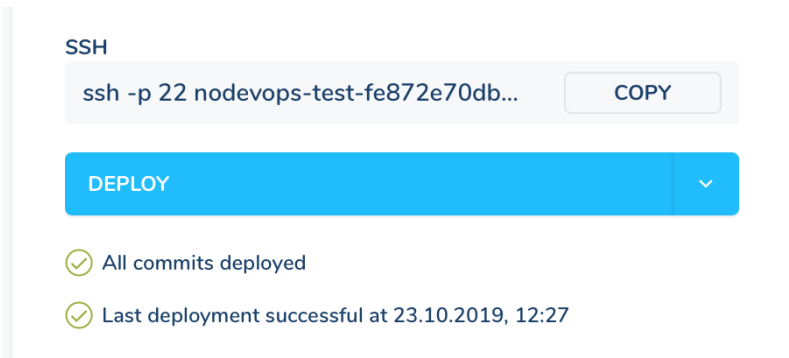


Explanation

Further reading: *What exactly happens on Divio when a project is created?*

3.1.3 Deploy your project

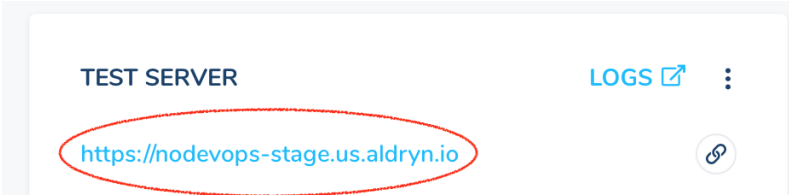
The project exists, but isn't actually running yet. Let's deploy it: hit **Deploy** for the test server.



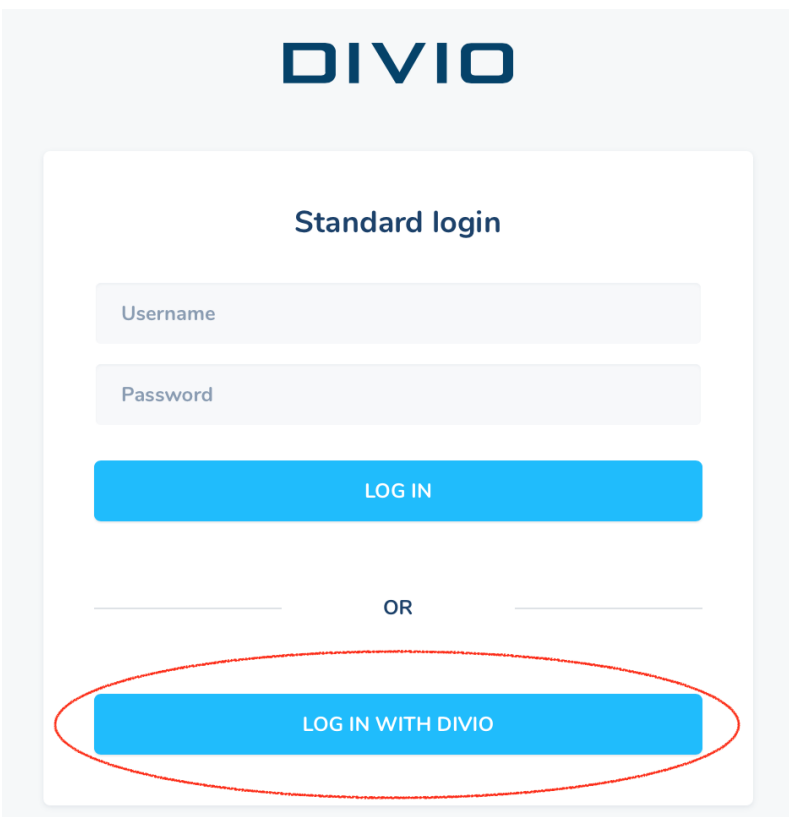
This will take about three or four minutes.

Visit the *Test* site

You can use the link in the Control Panel to access the *Test* server of your project.



You can now log in to the Django admin.



But I never wanted to do DevOps!

Explanation

Further reading: *What exactly happens on Divio when a project is deployed?*

3.1.4 Set up and run your project up locally

Introducing

- `divio login`
 - `divio project list`
 - `divio project setup`
 - `docker-compose up`
-

The next step is to set up your project on your own computer, for development purposes and also to understand better how the system works.

For this, we will use the Divio CLI that you installed in the *Installation* section.

Log in with the CLI

Run:

```
divio login
```

This will fetch a token from <https://control.divio.com/account/desktop-app/access-token/>.

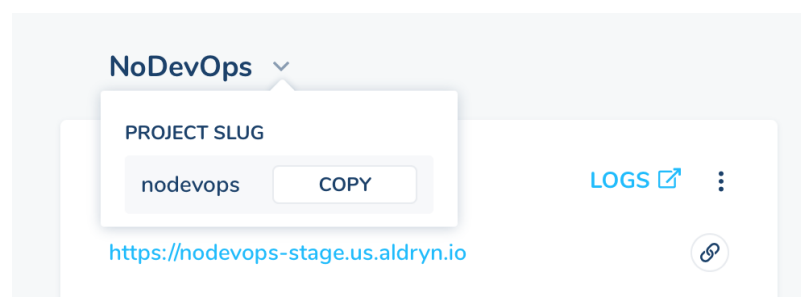
Upload your public key to the Divio Control Panel

Upload your public key to <https://control.divio.com/account/ssh-keys/>

Note: If you need help with setting up public keys, see [GitHub's excellent guide](#).

Set up your project locally

Get your project's *slug* from the Control Panel:



Or you can see it by listing your projects:

```
divio project list
```

Set up the project:

```
divio project setup <project slug>
```

This will *build* your project.

Explanation

Optional further reading: *What happens during the set-up process of a Divio project?*

Run the local site with `docker-compose up`

cd into the project directory, and to get the site up and running, execute:

```
docker-compose up
```

You should be able to log in to your site at <http://localhost:8000>.

`docker-compose up` is one of the key commands for working with your Docker project locally.

Stop the project

If you hit `control-c` in the console when the project is running, it will stop it.

3.1.5 Control the local project with `docker-compose`

Introducing

- `docker-compose stop`
- `docker-compose build`

`docker-compose` is a command that uses the project's `docker-compose.yml` file to control the local project. The `docker-compose.yml` file contains information about how the project should be built, what containers should be created, how they should be linked to each other, and so on.

`docker-compose up` and `stop`

You already know `docker-compose up`, which runs your project. If you hit `control-c` in the console, it will stop it.

Another way to stop it is by executing `docker-compose stop` in the same directory.

`docker-compose build`

When you created the project the first time, you *built* it locally. Sometimes you need to rebuild a project (for example, if you made a change to its requirements) with `docker-compose build`.

Try it.

Look at the output:

```
docker-compose build
db uses an image, skipping
Building web
[...]
```

It builds the `web` container, but skips building the `db` container. That's because it doesn't need to. The `db` container, as you will see from the `docker-compose.yml` file, uses an off-the-shelf image. In Docker, images can be built and saved for re-use, saving time and making things repeatable.

Now that you know how to stop, start and build a local project, we are going to enter the local development cycle.

Explanation

Further reading: *[Understanding the docker-compose.yml file](#)*

3.1.6 Explore the local environment

Introducing

- `docker-compose run`
 - the *statelessness* of containerised environments
-

We're going to dive in further to the local environment, and get used to some of the ways available to interact with it.

So far you have used two `docker-compose` commands:

- `docker-compose up`, to run the project
- `docker-compose build`, to make it rebuild the image

For this exercise, we will assume that you plan to install a reusable Django application into your project that requires [Pillow](#), the [Python imaging library](#). The steps you will follow here are ones you might follow in a real-life example.

Use `docker-compose run` to run a command

First, is Pillow already installed and available in the environment? We can check:

```
docker-compose run web pip list
```

This runs the command `pip list` inside a web container.

You will find yourself using `docker-compose run` a lot. In this example, we're using it as a "one-off" command. The command:


- launched a container
- ran `pip list` in it

- shut down the container

(it created the container from the image just to run `pip list`).

Use `docker-compose` run to work inside a container

As you may already know, Pillow requires some third-party system libraries to support various image types. You will find two sample images included in this documentation (note that your web browser may not support the WebP format):

- a JPEG image  - every browser should render this file
- a WebP image - not all browsers can render this; download it anyway

Download them, and copy them to your project. Let's check which images are supported, by starting a shell in the environment and trying to open them:

```
docker-compose run web bash
Starting nodevops_db_1 ... done
root@59d5a381b0ff:/app#
```

In this case, the container is launched and remains running while we work in it. We can open a Python shell:

```
/app# python
Starting example_db_1 ... done
Python 3.6.8 (default, Mar 27 2019, 08:53:45)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

Now in the Python shell, let's see if it has JPEG support ready:

```
>>> from PIL import Image
>>> Image.open("test.jpg")
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=1024x768 at 0x7FA2A2E1C4A8>
```

That indicates that it does.

And WebP support:

```
>>> Image.open("test.webp")
/usr/local/lib/python3.6/site-packages/PIL/Image.py:2817: UserWarning: image file
↳could not be identified because WEBP support not installed
  warnings.warn(message)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.6/site-packages/PIL/Image.py", line 2818, in open
    raise IOError("cannot identify image file %r" % (filename if filename else fp))
OSError: cannot identify image file 'test.webp'
```

So, our environment can support JPEG images but not WebP. For that, we need to install the `webp-dev` system library. Exit the Python shell, and use `apt` for the installation:

```
/app# apt-get update
Ign:1 http://deb.debian.org/debian stretch InRelease
[...]
Reading package lists... Done

/app# apt-get install libwebp-dev
```

(continues on next page)

(continued from previous page)

```
Reading package lists... Done
[...]
Processing triggers for libc-bin (2.24-11+deb9u4) ...
```

You can start up Python in the shell again, and try once more:

```
>>> from PIL import Image
>>> Image.open("test.webp")
<PIL.WebPImagePlugin.WebPImageFile image mode=RGB size=540x405 at 0x7FADB5085A58>
```

Success!

And now you know that in order to use WebP images with Pillow, you're going to need `libwebp-dev` installed in the environment.

Discover the statelessness of containerised environments

However, try this:

1. Exit the Python shell.
2. Exit the Bash shell (this will exit the container).
3. Start the environment again with Bash (`docker-compose run web bash`).
4. Try opening the Python shell, and opening the image as you did before with `Image.open("test.webp")`.

You may be surprised (or annoyed) to find that it fails, as if you had never installed `libwebp-dev`. This is because every time your environment is launched, it is created anew from the image. Nothing you do to the container *persists*. The container is **stateless**.

If we need something to persist in the environment, it will need to be baked into the image itself.

We will do this in the next section by editing the `Dockerfile`.

3.1.7 Edit the Dockerfile

Introducing

- the `Dockerfile`
 - using `RUN` in the `Dockerfile`
-

Your `Dockerfile` will look much like this:

```
FROM divio/base:4.15-py3.6-slim-stretch

ENV PIP_INDEX_URL=${PIP_INDEX_URL:-https://wheels.aldryn.net/v1/aldryn-extras+pypi/$
↪{WHEELS_PLATFORM:-aldryn-baseproject-py3}/+simple/} \
    WHEELSPROXY_URL=${WHEELSPROXY_URL:-https://wheels.aldryn.net/v1/aldryn-
↪extras+pypi/${WHEELS_PLATFORM:-aldryn-baseproject-py3}/}
COPY requirements.* /app/
COPY addons-dev /app/addons-dev/
RUN pip-reqs compile && \
    pip-reqs resolve && \
    pip install \
```

(continues on next page)

(continued from previous page)

```
--no-index --no-deps \  
--requirement requirements.urls
```

```
COPY . /app
```

```
RUN DJANGO_MODE=build python manage.py collectstatic --noinput
```

It contains a series of instructions that are used to build the project's image. For example, the highlighted line above runs a Django command to collect static files.

Use RUN to install a package in the image

We want to add the commands we used (`apt-get update` and `apt-get install libwebp-dev` to install `libwebp-dev`. Add two lines as indicated:

```
FROM divio/base:4.15-py3.6-slim-stretch
```

```
RUN apt-get update
```

```
RUN apt-get install libwebp-dev -y
```

```
ENV PIP_INDEX_URL=${PIP_INDEX_URL:-https://wheels.aldryn.net/v1/aldryn-extras+pypi/$  
↪{WHEELS_PLATFORM:-aldryn-baseproject-py3}/+simple/} \  

```

By making a change to the `Dockerfile`, you've made a change to the instructions that build it. To use these new instructions, you will need to rebuild the project:

```
docker-compose build
```

And now you can try once more to open a Python shell and open a WebP image as you did in the previous section. In fact you can launch a Python shell directly with:

```
docker-compose run web python
```

and then run:

```
>>> from PIL import Image  
>>> Image.open("test.webp")
```

And this time it will work - because you have baked support for WebP into the environment.

Explanation

Further reading: *Understanding the Dockerfile*

3.1.8 Install a new Python package

Introducing

- the `requirements.in` file
-

Next, we're going to install a new package, [Django Axes](#), into the project (Django Axes keeps track of log-in attempts).

But I never wanted to do DevOps!

Maybe you are used to using pip to install Python packages into your environment. You can try it - launch a bash shell in a container as you did previously, and run

```
pip install django-axes==3.0.3
```

It will work - but what do you think will happen after you exit the container and launch it again? Will `django-axes` still be there? You can check with:

```
docker-compose run web pip list
```

As you will realise, pip installation (just like other operations on the environment) doesn't survive the statelessness of Docker containers.

Build the pip installation into your Docker image

You have already used the `RUN` command in the `Dockerfile`, so you could do the same again, adding:

```
RUN pip install django-axes==3.0.3
```

However, this is a bit crude and won't look very nice when we have to add many packages. In fact, there are already some lines in the `Dockerfile` that take care of pip installation for you:

```
ENV PIP_INDEX_URL=${PIP_INDEX_URL:-https://wheels.aldryn.net/v1/aldryn-extras+pypi/$
↪{WHEELS_PLATFORM:-aldryn-baseproject-py3}/+simple/} \
    WHEELSPROXY_URL=${WHEELSPROXY_URL:-https://wheels.aldryn.net/v1/aldryn-
↪extras+pypi/${WHEELS_PLATFORM:-aldryn-baseproject-py3}/}
COPY requirements.* /app/
COPY addons-dev /app/addons-dev/
RUN pip-reqs compile && \
    pip-reqs resolve && \
    pip install \
        --no-index --no-deps \
        --requirement requirements.urls
```

These lines process the `requirements.in` file, in which you will already find some packages listed. Add:

```
django-axes==3.0.3
```

right at the end of the file. Now you will need to rebuild the project because you have made a change. Run:

```
docker-compose build
```

Now the *image* for the project contains the new requirement - this time you *will* see it if you run:

```
docker-compose run web pip list
```

because (like the WebP support in the previous step) you have baked it into the image, and it will be available in any environment created from that image, from now on.

Configure the Django application for django-axes

The only configuration required for Django Axes is to add it to the `INSTALLED_APPS` in `settings.py`. This project uses Divio's optional Django project with auto-configured settings, so the way to do it is with:


```
# all django settings can be altered here

INSTALLED_APPS.extend([
    "axes",
])
```

You will need to run migrations:

```
docker-compose run web python manage.py migrate
```

And here's Django Axes in the admin, which you will see if you start the project up again (`docker-compose up`):

AXES	
Access attempts	 Change
Access logs	 Change

3.1.9 Deploy your changes to the cloud

Introducing

- `divio project deploy`
 - SSH access to the server
-

We've completed some local development. Let's deploy it to the cloud.


We made changes to three files (`Dockerfile`, `requirements.in`, `settings.py`). So:

```
git add Dockerfile
git commit -m "Added WebP support"
git add requirements.in settings.py
git commit -m "Added Django Axes"
git push
```

On the project Dashboard, you will see that your new commits are listed:

SSH

ssh -p 22 nodevops-test-fe872e70db... [COPY](#)

DEPLOY 

2 undeployed commits

You can deploy using the Control Panel, or by running:

```
divio project deploy
```

But I never wanted to do DevOps!

When it has finished deploying, you should check the Test server to see that all is as expected. Once you're satisfied that it works correctly, you can deploy the Live server too:

```
divio project deploy live
```

You have now worked through the complete workflow: project creation, local development, deployment. We'll continue to work in a cycle of development and deployment, introducing new concepts and techniques.

Explanation

Now that you have worked through a complete cycle, it would be a good time to pause and consider some of the questions that have arisen.

Further reading: *Ways of thinking about containerised development and deployment*

3.1.10 Further development

Introducing

- `divio project push db`
 - `divio project push media`
-

In this section we will continue to develop the project. We will install [Django Photologue](#), a popular image gallery application. Some of these steps are familiar, but it is valuable to reinforce them.

Install Django Photologue in the project

We will more or less follow the directions in the [Django Photologue](#) installation and configuration instructions, and use the templates and styling from the [Django Photologue demo project](#).

Requirements

Add the package to your `requirements.in` (always remembering to pin it):

```
django-photologue==3.10
```

(Check its [dependencies](#) - though in fact we don't need to do anything here.)

Edit `settings.py`

Add the necessary applications to `INSTALLED_APPS`:

```
INSTALLED_APPS.extend([
    'axes',
    'photologue',
    'sortedm2m',
])
```

Edit `urls.py`

```
# -*- coding: utf-8 -*-
from django.conf.urls import url, include
from django.views.generic import TemplateView
from aldryn_django.utils import i18n_patterns
import aldryn_addons.urls

urlpatterns = [
    url(r'^$', TemplateView.as_view(template_name="homepage.html"), name='homepage'),
    url(r'^photologue/', include('photologue.urls', namespace='photologue')),
] + aldryn_addons.urls.patterns() + i18n_patterns(
    # add your own i18n patterns here
    *aldryn_addons.urls.i18n_patterns() # MUST be the last entry!
)
```

Copy the templates and static files from the demo project

The templates and static files are at https://github.com/richardbarran/django-photologue/tree/master/example_project/example_project.

Copy the templates to your project's `templates` directory, and the static files to `static`.

Build and migrate

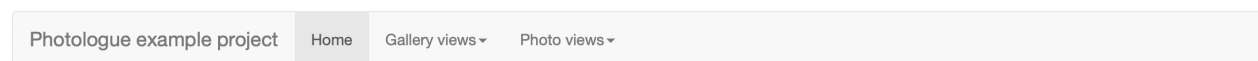
As before, you will need to build and run Django migrations:

```
docker-compose build
docker-compose run web python manage.py migrate
```

And now you are ready to try out the project:

```
docker-compose up
```

At <http://localhost:8000>, you should now find the site:



Welcome to the Photologue example project

This is a quick demo of the Photologue application - just click on the menu options above.

It uses the built-in [Bootstrap](#)-compatible templates that you can use to get you quickly up and running, or completely replace with your own templates.

In the admin, create a gallery or two at <http://localhost:8000/en/admin/photologue/gallery/>, and add some images to them.

We installed WebP support earlier - so make sure that it can handle a WebP image (again, note that your browser may not support WebP images).

But I never wanted to do DevOps!

Commit and deploy your changes

Once the local project is working as it should, commit the changes:

```
git add requirements.in settings.py urls.py static templates
git commit -m "Installed Django Photologue"
git push
```

And finally:

```
divio project deploy
```

You can now check the project on the Test server.

Push database and media

You will notice that the project on the cloud doesn't contain any of your photos. That's because although you pushed your code, your content hasn't been transferred.

To push the database to the cloud:

```
divio project push db
```

And the media files:

```
divio project push media
```

(in both cases you could add `live` to push them to the Live server).

3.1.11 Explore file handling

Introducing

- devolved media storage
-

In this section we will explore an aspect of containerised deployment that often takes people by surprise. It's related to the *statelessness* and *lack of persistence* of the containerised environment, and has various implications, not just for deployment, but also for how your code handles files.

Disable volume mapping locally

For convenience, the `docker-compose.yml` file in your project maps some directories *inside* the container environment to directories accessible from outside it. That's how you can make changes to code and configuration using your normal development tools, rather than having to work inside the container itself.

This is achieved with the `volumes` directive:

```
services:
  [...]
  web:
    [...]
    volumes:
      - "../app:rw"
      - "../data:/data:rw"
```


In order to understand the behaviour of containers better, we need to disable this.

Comment out the highlighted section.

Explore file behaviour

Launch a new bash shell inside the container:: `docker-compose run web bash`

In the container, create a new file (`touch test_file.txt`).

You'll see it there if you run `ls`. You will be able to open it from a Python application too.

However, you won't see it in your own environment - it stays inside the container. Similarly, if you add a file to the project directory in your environment, you won't see inside the container. Exit the container, and start one up again: the file won't be there. It disappeared when the container disappeared.

Create a file once more in the container, and now, leaving your container running, start up another bash session, in a new terminal window. You won't see the file in that session either. It's another container for the same project, but the changes you make in the one container *will not appear in any others*.

Containers are their own little universe, as far as local file storage is concerned.

This has significant implications for your code. It means that you can't use local file storage locally (what would happen in a project that used two containers running in parallel as a way of increasing performance?).

Instead, as Django Photologue does, you have to ensure that your code doesn't make assumptions about how images are handled. In a Django project like this one, that means you should not use Django's `FileSystemStorage`. Instead, you need to use `django.core.files.storage.default_storage`, which devolves handling to an appropriate backend.

In this project, Aldryn Django takes care of setting up storage, on Amazon's S3 system.

Further reading

Containerisation has implications for other aspects of coding too.

For some of them, see *Some practical implications of containerisation*.

3.1.12 Using an external service

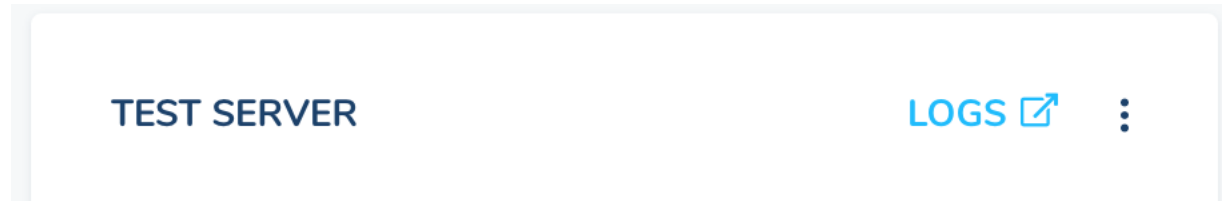
We'll continue with the development cycle, this time adding an *external service* to the project. The example we'll use is just a simple one: setting up logging with [LogDNA](#). All the same, it demonstrates some key aspects of configuring external services - you will most likely need to:

- install some additional software specific to that service and the interfaces it provides
- obtain some sort of key to authenticate with the external service
- configure your settings appropriately

Explore your project's logs

Your project emits logs. You can see them in the console when running locally. In addition, you can use the *Logs* link on the cloud server to see logs from the Test and Live environments.

The cloud server logs



These logs have some limitations.

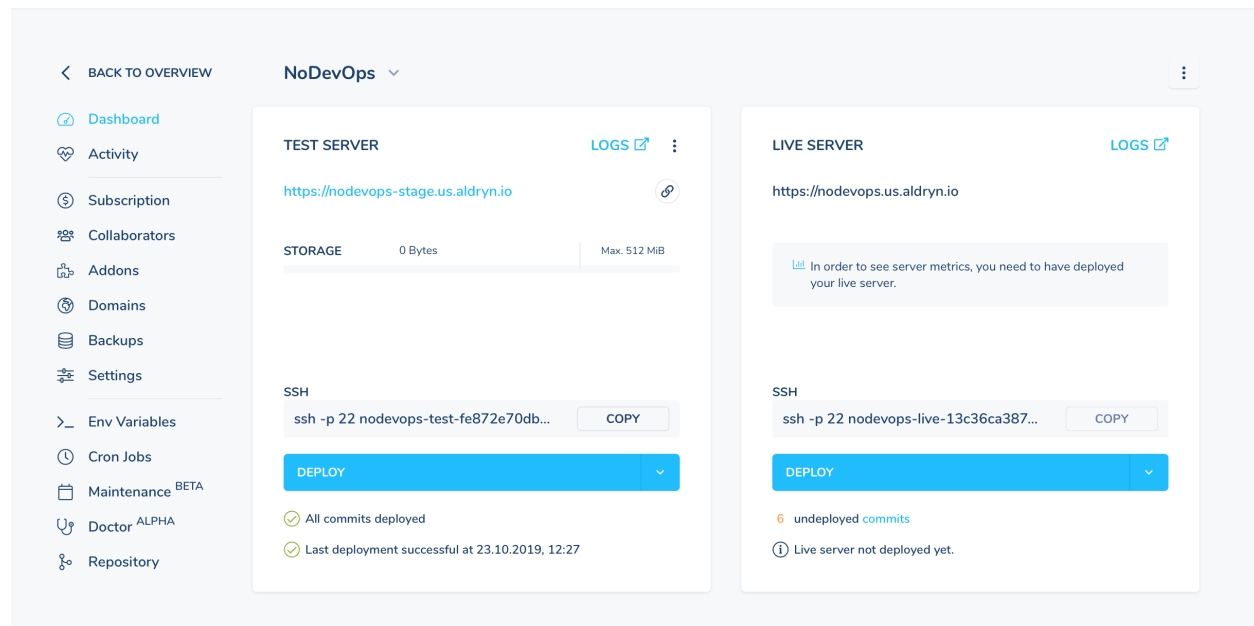
- they are limited to 1000 lines
- they are undifferentiated (everything appears in one place)
- they do not update in real time

They're adequate for basic purposes, but not really sophisticated enough for use in production for a complex, high-traffic project.

Server logs from a shell session

Perhaps you are used to SSHing into a server to run `tail` on a log file. This is quick and convenient, and perhaps you'd like to do this with your containerised project. However, it's not going to work well in a containerised environment. When your project is running inside a container, you won't have the same kind of access. For example, *you can't rely on being able to SSH into it in the usual way.*, and your project may not even have a persistent filesystem that it can save its logs to.

You can try it if you like, if your project has SSH access enabled:



- but you will soon find that the results are not satisfactory.

Configure logging as an external service

Subscribe to the service

If you don't already have a LogDNA account, visit <https://logdna.com> and register for a free account. LogDNA will provide you with an *ingestion key*.

Install the logdna package

The `logdna` package provides a new logging handler (`logdna.LogDNAHandler`) that will forward log messages to LogDNA.

Add `logdna` to your project's requirements and rebuild the project (`docker-compose build`).

Configure logging in `settings.py`

It's beyond the scope of this document to go into details of logging configuration in Django. In summary, however:

- the Aldryn Django project you're using a [default logging configuration](#) already
- this makes the `LOGGING` dictionary available in `settings.py`
- we will extend that, by adding a new logger and handler configuration.

The LogDNA library provides a logging handler, via the class `logdna.LogDNAHandler`. Add the new configuration to the *end* of your `settings.py`:

```
LOGGING["handlers"]["logdna"] = {
    'class': 'logdna.LogDNAHandler',
    'key': '<insert your ingestion key here>',
    'options': {
        'hostname': 'your-website-name',
        'index_meta': True
    }
}
```

What we have done here is added the new handler, `logdna` (the name doesn't actually matter) as a key to the dictionary.

Next, we need to configure the existing loggers, that actually produce the logs, to use the handler. In this example, we will append the `logdna` handler to the configuration of:

- the unnamed root logger `""`
- the `django` logger
- the `django.request` logger

```
LOGGING["loggers"][""]["handlers"].append('logdna')
LOGGING["loggers"]["django"]["handlers"].append('logdna')
LOGGING["loggers"]["django.request"]["handlers"].append('logdna')
```

More information about configuring Django logging can be found in [Django's logging documentation](#).

Note: The precise way of doing this shown here is specific to Aldryn Django projects. For another service or a project not using Aldryn Django, the principle would be the same, but the details would be different.

Test the new configuration

You can test your logging configuration locally, and once you have done that, deploy the site on the cloud once more. Now the logs emitted by your application will be sent to LogDNA, where you can manage them better.

3.1.13 A complete example workflow

To help you practise the skills you have learned, here is an example of a complete workflow, using Wagtail (a very popular Django-based content management system), Azure’s Cognitive Services API, OpenCV and more.

In this exercise, you will:

- create a project
- add and deploy some custom code
- integrate OpenCV for feature and face detection in images
- use a third-party Wagtail application that includes an API for image content recognition, via Microsoft Azure.

Create the project, on the cloud and locally

Create a new Wagtail project at <https://control.divio.com/control/project/create/> (skip the Subscription page). Deploy the Test server.

Set the project up locally, with `divio project setup <project id>`.

Check that it runs as expected: `cd` into the project, and run:

```
docker-compose up
```

You can log into it at <http://localhost:8000/django-admin>, and see the Wagtail admin at <http://localhost:8000/admin>.

Build a new application

Create the application

Next, we’ll add a new “home” application to the project, by extending Wagtail’s `Page` model.

Run:

```
docker-compose run --rm web python manage.py startapp home
```

This issues the Django `startapp` command, inside the web container. You will find the files for the new `home` application in the project.

Add the new HomePage model

In the new application (at `home/models.py`), add a new `HomePage` model (this example is copied straight from the [Wagtail documentation](#)):

```
from django.db import models

from wagtail.core.models import Page
from wagtail.core.fields import RichTextField
from wagtail.admin.edit_handlers import FieldPanel

class HomePage(Page):
    body = RichTextField(blank=True)

    content_panels = Page.content_panels + [
        FieldPanel('body', classname="full"),
    ]
```

Add a template for it

Create a new file at `home/templates/home/home_page.html`:

```
{% extends "base.html" %}

{% load wagtailcore_tags %}

{% block body_class %}template-homepage{% endblock %}

{% block content %}
    {{ page.body|richtext }}
{% endblock %}
```

Add a base template for the whole project

As you can see, the template above extends `"base.html"`. So, create `templates/base.html` in the root of the project (not the home application):

```
<h1>{{ page.title }}</h2>

{% block content %}{% endblock %}
```

Configure Django settings

In the project's `settings.py`, edit the `INSTALLED_APPS` list, to add the home application:

```
INSTALLED_APPS.extend([
    'home',
])
```

Create and run migrations

The project has new models, so we need to create and run migrations for them:

```
docker-compose run --rm web python manage.py makemigrations home
```

But I never wanted to do DevOps!

followed by:

```
docker-compose run --rm web python manage.py migrate
```

Check that you can now create new pages with a body field.

Deploy your changes

Git add, commit and push your changes to the Cloud.

Then you can either deploy the Test server using the Control Panel, or run:

```
divio project deploy
```

Implement image feature detection

Upload an image to Wagtail. You will find that you can draw a ‘focal point’ around the important part of an image, such as the subject’s face. This is used when cropping automatically. But, Wagtail has a nice feature: automatic detection of key areas of an image.

To enable the feature, add:

```
WAGTAILIMAGES_FEATURE_DETECTION_ENABLED = True
```

to the `settings.py`. Before it will work though, we need to add OpenCV to the project.

Install a package via `requirements.in`

In your project’s `requirements.in`, add:

```
https://files.pythonhosted.org/packages/7b/d2/  
↪a2dbf83d4553ca6b3701d91d75e42fe50aea97acdc00652dca515749fb5d/opencv_python-4.1.0.25-  
↪cp36-cp36m-manylinux1_x86_64.whl
```

(This is from <https://pypi.org/project/opencv-python/>.)

Install system libraries

We also need some system libraries for OpenCV. In the `Dockerfile`, add:

```
RUN apt-get update && apt-get install -y libsm6 libxrender1 libxext6
```

(after the `# <DOCKER_FROM>...# </DOCKER_FROM>` is a good place).

Rebuild

The changes you have made require that the project be rebuilt:

```
docker-compose build
```

Once completed, you should be able to upload an image and see the automatic feature detection.

Implement image content detection

The third-party `wagtailaltgenerator` is a nice addon for Wagtail.

Before you use it, you need to obtain a key for [Azure cognitive services](#) - you can get a free account and key. Once you have your key:

- add `wagtailaltgenerator` to the `requirements.in`
- add `wagtailaltgenerator` to the `INSTALLED_APPS`

Then add key to the `setting.py`, for example:

```
COMPUTER_VISION_API_KEY = '99265265444f147baecb737f38074bca'  
COMPUTER_VISION_REGION = 'centralus'
```

Once again:

- `docker-compose build`
- test locally
- push to cloud
- deploy on cloud

and finally, once you're happy with it on the Test server, you can also deploy to Live:

```
divio project deploy live
```


CHAPTER 4

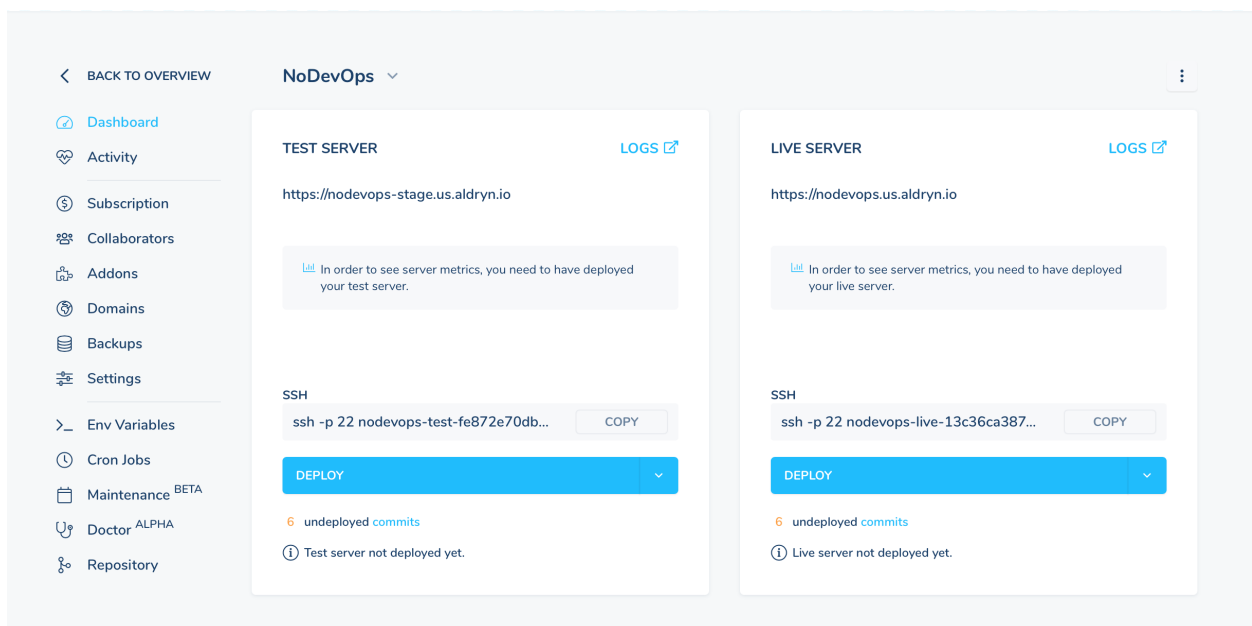
Further reading

The pages in this section contain optional, additional explanation. The secret of learning is that you don't need to understand the thing you are learning - *doing* is far more valuable. All the same, once you have learned some basic things you will be ready for deeper explanation. The pages in this section are all linked to from pages in the previous section.

4.1 Further reading

4.1.1 What happens when a project is created

When a new project is created, you will see something like this on the project dashboard:



But I never wanted to do DevOps!

As implied by the *6 undeployed commits* message, the new project has a repository. This is what the infrastructure does:

A repository is created

The system creates a new repository. As an *application developer*, you could in fact have created that repository and everything in it yourself, but in this case, it's created for you. You might even prefer to, and on some platforms you would be required to. For now, since the Divio Control Panel can do this for us, let's work with it and consider alternatives later.

This Git repository contains all the instructions needed to build a *Docker image* for the project. It contains a `Dockerfile`, describing the commands that must be run to build it, a `requirements.in` file, listing the Python packages that will need to be installed, and so on.

This repository defines the *application*, the part that you as an application developer will be responsible for and will want to get your hands on. To the greatest extent possible, this application needs to be *platform-independent*. It should not care whether it is to be deployed on Divio or on some other platform (just like your skills, this application should be transferable with minimum extra work).

Divio Cloud developer handbook links

- [Version control for your Divio projects](#)
 - [How to configure a Git remote for your project](#)
-

Services are provisioned

The Control Panel has provisioned services for the new application. (As an application developer who doesn't want to be doing DevOps, you should be thrilled to discover the Cloud Platform has taken care of this for you.)

The database, media storage, web server and other services have not only been set up for you, your project is ready to use them right away.

It has done this for two *environments*, *Test* and *Live*. On Divio, your *Test* server is a private staging environment, while *Live* is the public server.

Other Cloud providers may offer a similar arrangement, but in any case the important point is that given a *single repository*, you can deploy it in multiple environments, and all Cloud hosting providers will make this easy.

Environment variables are configured

You may be used to including things like database credentials in settings files, or perhaps in multiple settings files for different environments. It's a good idea to keep those things **out** of your repositories. They aren't *code*, they are *configuration*, and they belong to *environments*.

So, we store them in environment variables, and each environment has its own set of variables. All the repository contains is a way to look for the variables and turn them into the appropriate Django settings.

Divio developer handbook links

[Divio CLI reference](#)

4.1.2 What happens during the Docker deployment process

This is a two-part process.

1. First an **image is built**. You can think of an image as a *template*.
2. Next, a **container is deployed**. You can think of containers as *instances that are created from the template*.

Building the image

- The Control Panel clones the Git repository.
- Docker follows the instructions in the `Dockerfile` to build the *image*:
 - it uses a pre-built Docker image to install system components (DevOps work, in other words)
 - it uses pip to install the Python requirements
 - it runs the Django `collectstatic` command to copy static files to the right place.

Deploying the container

- The image is now built, and the Control Panel takes over again. It creates a Docker *container* (an instance of the image), and provides it with the required environment variables
- Inside the container:
 - it launches Django
 - runs any database migrations that need to be run.
- Then the Control Panel checks that the new container is running as it should be, and if so it declares the deployment successful.

The site is now available, connected to its web server, database and media storage.

Deployment on other platforms

On different platforms, the deployment process can be a little different, but the basic principle is the same: the system builds an image from the repository, creates a container from the image and launches it in an environment, wired up to its backing services.

4.1.3 What happens during the local set-up process

What happens when you set up the project locally is almost the same as what happens when it's *deployed on the Cloud*, and you can watch it happening in the console.

- The Divio CLI uses Git to clone the repository.
- Docker uses the instructions in the `Dockerfile` to build the *image*:
 - it uses a pre-built Docker image to install system components
 - it uses pip to install the Python requirements
 - it runs the Django `collectstatic` command to copy static files to the right place.
- Finally, the Divio CLI will pull down the database and media from your Test server.

The image is now built, and is waiting to be used.

Docker Compose and `docker-compose.yml`

Cloud deployments are handled (*orchestrated*) by the Control Panel. There are various orchestration systems - as the name suggests, orchestration means co-ordinating all the various components of project in the right order so that they add up to a functional whole.

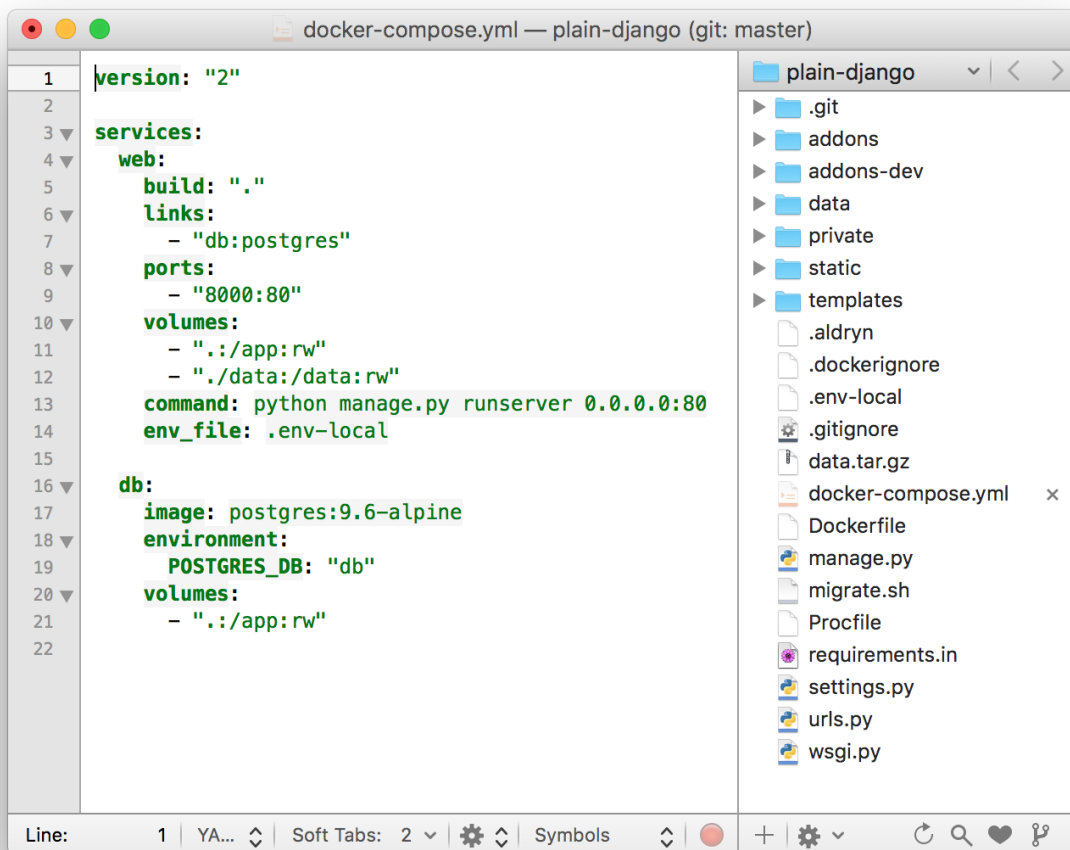
In your local environment, there are a number of Cloud components that you **don't** have:

- a Control Panel
- a database cluster
- a media host
- a public web server and load-balancer

So we have to do things quite differently locally.

Instead of the Control Panel, Docker Compose orchestrates the deployment. Instead of a database cluster, Docker Compose sets up a Postgres database inside another Docker container on your computer. And by running the Django project in `DEBUG` mode, Django itself takes care of publishing media and providing a web server (more on this later).

4.1.4 The `docker-compose.yml` file



When you run your project (with `docker-compose up` or other commands), Docker Compose will use this file to orchestrate its components.

Below we've highlighted the relevant parts of the file - you don't need to go through it all right now, but you will probably want to come back to it later.

Services or containers

```
services:
  web:
    [...]
  db:
    [...]
```

The file defines two *services* - in effect, two containers that Docker Compose will launch. One is the `web` service or container, containing your Django project. The other is the `db` container, our local stand-in for the database cluster that would be available on the Cloud.

The web service

```
build: "."
```

Docker Compose needs to know where to find the instructions to build the image for this container - the answer is right here, because that's where the `Dockerfile` is.

```
command: python manage.py runserver 0.0.0.0:80
```

When the container starts up, start the Django runserver on port 80.

```
ports:
  - "8000:80"
```

... and map that port 80 *inside* the container to port 8000 *outside*.

```
env_file: .env-local
```

Read the file `.env-local` for any environment variables.

```
links:
  - "db:postgres"
```

Resolve the hostname `postgres` to the `db` container.

The db service

```
image: postgres:9.6-alpine
```

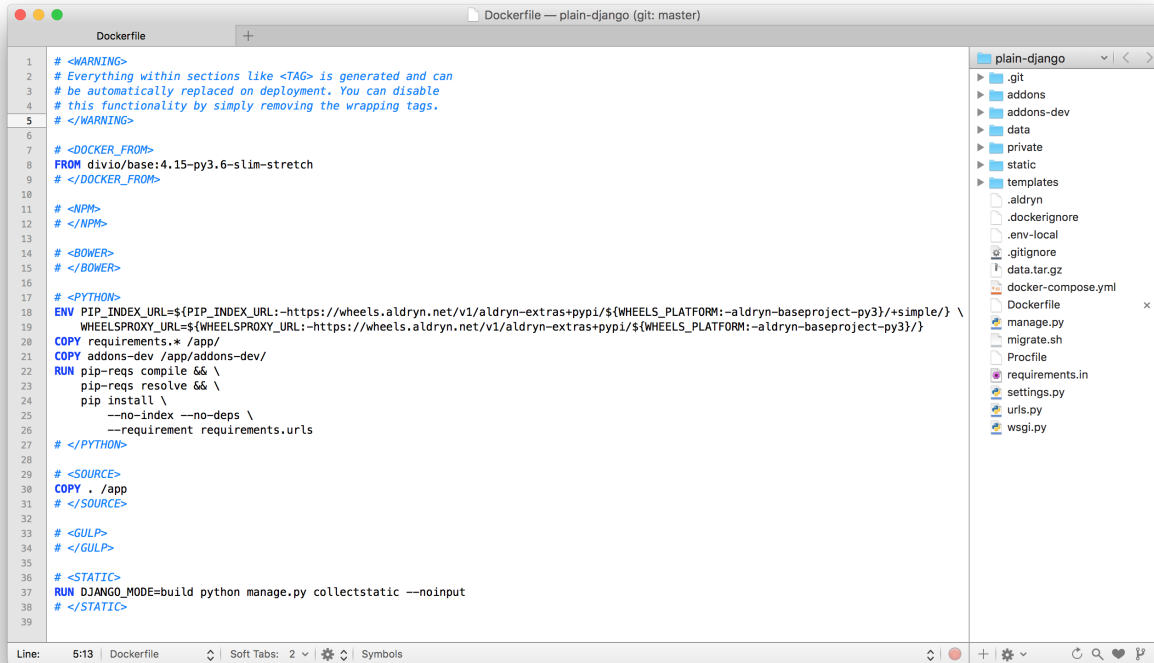
This container is based on a pre-built, off-the-shelf image, one specifically created to run Postgres 9.6 on Alpine Linux.

```
environment:
  POSTGRES_DB: "db"
```

When the container is launched, its environment should include the `POSTGRES_DB` variable with the value `db`.

4.1.5 The Dockerfile

Let's take a look at the `Dockerfile`. This is the file that defines your web application *image*. (The `db` image doesn't have a `Dockerfile` associated with it - it's a pre-built image.)



The `Dockerfile` in this project is defined when the project is created, using the options you selected. (It's one of the Divio standard `Dockerfiles`, so inevitably it contains some Divio-specific components - but we can ignore those, and look at ones you might use with any Docker host.)

The `FROM` instruction

```
FROM divio/base:4.15-py3.6-slim-stretch
```

This is the pre-built image that our project is based on. It's getting it from the [Docker Hub](#), where you can see that the codebase for this image is [published on GitHub](#).

Not only that, you can also see that it, in turn, is based on another image: `python:3.6.8-slim-stretch`. These images form the layers of the Docker project, and they go down all the way to the operating system components. It's a very handy system for building what you need in a modular and repeating way.

Python package installation

First, `ENV` declares a couple of environment variables:

```
ENV PIP_INDEX_URL=${PIP_INDEX_URL:-https://wheels.aldryn.net/v1/aldryn-extras+pypi/$
↪ {WHEELS_PLATFORM:-aldryn-baseproject-py3}/+simple/} \
    WHEELSPROXY_URL=${WHEELSPROXY_URL:-https://wheels.aldryn.net/v1/aldryn-
↪ extras+pypi/${WHEELS_PLATFORM:-aldryn-baseproject-py3}/}
```

Then we copy some files into a working directory within the image:

```
COPY requirements.* /app/
COPY addons-dev /app/addons-dev/
```

And run pip instructions to install the requirements:

```
RUN pip-reqs compile && \
    pip-reqs resolve && \
    pip install \
        --no-index --no-deps \
        --requirement requirements.urls
```

Copy miscellaneous files

Various files need to be copied into the image:

```
COPY . /app
```

Run collectstatic

collectstatic will do the same for Django's static files:

```
RUN DJANGO_MODE=build python manage.py collectstatic --noinput
```

4.1.6 Ways of thinking

While you have been following the tutorial steps, you will have encountered some key aspects of containerised development and deployment that make a difference to how you need to work.

In short, when you install something into a system, say using `apt` or `pip`, it will persist - remain installed - for the lifetime of the system. That could be indefinitely.

At the same time, one rarely rebuilds a server - or even a virtual server or a virtualenv - from scratch. Once it has been created, it is added to, maintained, upgraded and so on.

However, containers are by their nature short-lived and impermanent, **and** they are recreated from scratch very often.

Here I will recapitulate some of the implications of this and introduce some new ways to help think about them.

Persistence and non-persistence

The environment of containerised project, unlike that of a virtualenv for example, does not *persist*.

If you activate a virtualenv, install something in it, and come back to activate it again later, you will find your installed package still in it, because the environment persists.

The changes will last forever.

The changes in a containerised environment *do not persist*. If you activate the environment, install something in it and come back to activate it again later, you will find that what you installed is nowhere to be seen.

The changes to a container last only as long as the lifetime of that particular container; as soon as the container is shut down, the changes will be lost.

Understanding this is absolutely key to understanding how to work with containerisation.

Traditional and containerised development and deployment

In ‘traditional’ development and deployment, you’re building a system that you will eventually deploy. This is your work of art, your precious construction. It’s the expensive final end of your labours, that you’ve worked hard for. You started with the basics, added on to them, made multiple refinements, until finally your creation was complete and ready to face the world.

In containerised deployment, your final product is cheap and disposable. It’s not precious at all, it’s just an instance, infinitely reproducible, of an image, and what’s precious are the *instructions* that are followed to assemble that image.

What matters is *not the thing we build*, but *the instructions we build it with*.

So: **stop working on the construction; work on instruction.**

In a way, this is the central lesson of this workshop, and once you have internalised it, you will find working with containerisation much easier.

Containers are cheap

Containers are:

- cheap: it costs almost nothing (time, computing resources, money) to create a new one
- disposable: throw them away when you finish using them, as soon as you finish using them; don’t keep them around
- reproducible: if you can build one, you can build a thousand new ones with the same effort, and they’ll all be exactly the same
- unchanging: once created, they stay the same until they are destroyed; no important changes are made to them or in them that must be retained for the future

Instructions are expensive

On the other hand, whereas otherwise you could start with the basics and then refine and massage them into shape - you can’t take that attitude any more. What you start with, when you build your containers, must be **perfect**. You must come up with the perfect set of instructions, because once you set the build process in motion, what you get will be what you asked for, and you won’t have the opportunity to fix things up as you go along.

It’s the difference between preparing for a night camping at the bottom of your own garden, and getting a rocket ready for a trip to the moon.

One bit of good news is that Platform-as-a-service systems provide most of the instructions (well-tested, expertly-written) for you - you only need to make your amendments.

Advantages and disadvantages for the developer

You may feel that this way of working doesn’t only bring advantages, and that’s true - you do lose something.

The disadvantage: this is a slower and less streamlined, less interactive way to work. There’s initially more overhead from Docker and its set-up.

The advantage: you’ll quickly gain back the little extra effort you spend, and more.

In fact you very quickly get to used to not thinking:

I will make the changes on the Test server [or perhaps, in the development environment].

and instead start thinking:

I'll write the change to the project repository and rebuild.

Practical implications

Until you are fully used to these ways of thinking, some of their implications will occasionally take you by surprise.

For some of them, see *Some practical implications of containerisation*

4.1.7 Some practical implications of containerisation

In *Ways of thinking*, we discuss some general implications for thinking about development and deployment when using containerisation.

Here, I will discuss some more practical implications, in some cases specifically for the Aldryn Django project you are working with, that you will already have encountered, but are worth considering again.

There are many advantages to containerisation, such as knowing that the site running on your own computer is running in the same environment (as far as possible) as it will be in deployment. But this means that it's not as directly accessible to you may expect - it's not running directly on your computer, but in another environment on your computer.

This can take some getting used to, and it does mean that often an extra step will be required to interact with the project.

For example...

Debugging - *what* are you looking at?

Suppose that the application in our container is misbehaving, and we need to do some interactive debugging in the Django shell. Once again, instead of doing `python manage.py shell`, we need to do:

```
docker-compose run web python manage.py shell
```

That's fine - but remember, the shell you have just invoked is a shell *in a brand new instance*, not in the actual instance that was exhibiting the troublesome behaviour. The new shell is not receiving the same traffic or load as the "real" one. It's not like SSHing into a live server to find out what it's doing. This is why making good use of logging is the approach to take.

You can't just SSH into your server any more

It's very nice to be able to SSH into a live server to see what's going on, to inspect a Python process or even see what the web server or operating system are doing. It's very reassuring, and a quick way to get insights and maybe fix things.

You can't do that any more, because you don't really have a server any more. All you have is a containerised environment that is dedicated to running your application and all the other components are elsewhere.

For example, on Divio, although there is a link to SSH into a live container, it's not the same thing as being able to SSH into your server. It's just a new container that has been spun up specifically for you to SSH into, just as was the case when you executed `docker-compose run` commands.

That means you will never be able to use it to see the requests that are hitting your site - they are going elsewhere.

You might ask, "Why can't I SSH into the container that *is* receiving those requests?" In fact there's no guarantee that there is a single container that's receiving them. In many projects, multiple, parallel containers handle the requests allocated to them by a load-balancer.

Instead, you have to get used to gaining insights in a different way. The web containers report to a log; you can also make use of services such as Sentry.

These are less immediate and interactive - but in the long run, they are more valuable tools.

Interacting with the database

On the other hand, suppose you do a `.save()` on an object in the shell. Now this makes a change to the *database*. The database, unlike the environment in a container (but like, for example your media storage) is and needs to be *persistent*. So that change will be picked up by every container that connects to the database. To achieve this, the database stores its data outside the container.

As you can see, containerisation obliges us to think not about “the server” but about “the services”. A database is a different kind of thing from an application container; it behaves differently and has different needs. Once again, it can be difficult to start thinking in the new way, but when you do, it’s liberating.

You may be used to connecting to your live database to run queries using the tool of your choice, or even to make changes directly. That’s no longer possible; the database is **only** accessible from within your application. Added security, reduced convenience.

You can SSH into a container and use the `dbshell` or `psql` client - but the price of containerisation is more distance between you and the services you’re relying on.

You can of course also use other tools to interact with the database, such as Django’s `dbshell`:

```
docker-compose run web python manage.py dbshell
```

Or the `psql` tool (Django projects use Postgres, but the same principle applies for other databases):

```
docker-compose run web psql -h postgres -U postgres db
```

Note that the tools these commands launch are tools *inside* the `web` container, talking to a service inside a `db` container. What if you have a favourite GUI tool on your own computer, that you want to connect to the database running inside its container?

This becomes bit more complex. First you must:

Expose the database’s port

The database is tucked away inside its own container. In order to connect to the database from a tool running directly on your own machine, you will need to expose its port (5432) on that container, by adding a `ports` section to the `db` service in `docker-compose.yml` that maps the port to your host:

```
db:
  image: postgres:9.4
  ports:
    - 5432:5432
```

This means that external traffic reaching the container on port 5432 will be routed to port 5432 internally.

The ports are `<host port>:<container port>` - you could choose another host port if you are already using 5432 on your host.

Then restart the `db` container with: `docker-compose up -d db`

Connect to the database

You will need then to provide the connection details to the Postgres client. The connection port will be 5432 of course.

For a Divio project, the other values will be:

- username: postgres
- password: not required
- database: db
- address: 127.0.0.1

Now you can access the database using your Postgres tool of choice. For example, if you're using the `psql` command line tool, you can connect to the project database with:

```
psql -h 127.0.0.1 -U postgres db
```

Divio Cloud developer handbook links

- [How to interact with your project's database](#)
-

You can't store media files the way you once did

Although container instance running your application will have its own local file storage, this will be **independent** of each of the others, and it won't persist - once that instance ceases to exist, so will the files. That storage will also be inaccessible to any other instances of the application.

This means a project's applications, cron jobs or other process can't expect to save files to its local storage, and then expect to find them again: **if you save files to local storage you will lose them.**

Instead, use the media storage - Amazon S3 - that is provided as a backing service to your project.

You can do this by using Django's storage APIs. The storage defined by Django's `django.core.files.storage.default_storage` will do the right thing.

See [Working with your project's media storage in Python applications](#) and [How to interact with your project's media storage](#) for more.

Interact with a running container

In all the examples so far, we have fired up new containers with `docker-compose run ...` whenever we needed to use one, with the caveat that each time it's a brand new container with no memory of what has happened in or to other containers.

You can in fact interact directly with a container that is already running. First, you need to know its name; run:

```
docker ps
```

and look for the name, which might be something like `example_web`. Now you can do:

```
docker exec -i example_web python manage.py shell
```

(The `-i` flag gives you an interactive console.)

This gives you *some* persistent access into to a container, and can be useful when you do need that persistence while developing or debugging - but it only persists for the lifetime of that particular container.

Common operations

When do you need to...

If you:

- add a new module that hasn't yet been loaded
- change your local environment variables (in `.env-local`)

you will need to restart the local server.

It's also necessary to restart the local sever when you make a change to existing Python code, but the Django runserver restarts (just as usual) to reload it.

If you make a change to:

- your Python requirements
- your `Dockerfile`
- your `docker-compose.yml` file

you will (in some cases, may) need to run `docker-compose build` to build them into the image.

4.1.8 Some Divio-specific notes

The addons system

If you have a look at your `settings.py` file, you will find that it looks a bit different.

This is because in pre-packaged Divio project (recommended) full use is made of the *Divio addons system*.

Your Django project contains Django (a completely standard Django installed via pip), but it also contains *Aldryn Django*. Aldryn Django is a wrapper for Django. It installs Django, and provides some convenience code to configure it automatically for the Divio environments.

For example, you will see that there is no `DATABASES` setting in the file. In fact, all the settings are there in the module; try this:

```
docker-compose run --rm web python manage.py shell
>>> from django.conf import settings
>>> settings.DATABASES
{'default': {'NAME': 'db', 'USER': 'postgres', 'PASSWORD': '', 'HOST': 'postgres',
↪ 'PORT': 5432, 'CONN_MAX_AGE': 0, 'ENGINE': 'django.db.backends.postgresql_psycopg2',
↪ 'ATOMIC_REQUESTS': False, 'AUTOCOMMIT': True, 'OPTIONS': {}, 'TIME_ZONE': None,
↪ 'TEST': {'CHARSET': None, 'COLLATION': None, 'NAME': None, 'MIRROR': None}}}
```

How does this work?

In an application that has been installed via an addon (a wrapper), at least some of its configuration can be taken care of by the wrapper, in its `aldryn_config.py` file. For example, the **'Aldryn Django DATABASES setting** <https://github.com/divio/aldryn-django/blob/support/1.11.x/aldryn_config.py#L110-L129 > '_.

This looks for environment variables providing the database credentials, allowing the cloud Test and Live environments to supply them knowing that Django will pick them up.

Locally, DATABASES is set in `.env-local`, and functions in just the same way.

The lines:

```
import aldryn_addons.settings
aldryn_addons.settings.load(locals())
```

in your `settings.py` import all the settings that are automatically configured by the addons system.

You don't have to use this

It's important to note that **using this is entirely optional**. You don't need to use Aldryn Django, or the addons system, and you can configure all your settings manually or otherwise if you prefer.

However, not only can you use it in the case of packages such as Aldryn Django, you can even create wrappers for your own applications.

Some advantages of using the addons system

Managing settings this way does have some advantages:

Seamless operation across different environments

Whether in the local, Test or Live environments, Django will have the correct value, *provided by the environment*. You - as an application programmer who doesn't want to do DevOps - now longer need to care whether your database or media settings are right, you can just get on with your application programming.

Settings are optimised for the platform

The settings in the Divio addons are optimised for the platform - they've been configured expertly and on the basis of experience. *You* don't need to be responsible for them.

Settings can be exposed and managed via the Control Panel

In the project dashboard on the Control Panel, take a look at *Addons > Aldryn Django > Configure*. You'll see that some settings can be exposed in the GUI.

Settings can be configured programmatically

Some settings need to be added in just the right place. You can ensure that for example a `MIDDLEWARE` for an application will be inserted just where it needs to go.