django crash course

from start to production

May 7th, 2019
By Karol Horosin

What are you going to learn?

How to build a **complete** django application and bring it to production.

What do you need to know

Basic knowledge of

- Python
- Html, CSS, JavaScript
- Databases
- Web development concepts

This talk

github.com/horosin/django-crash-course

(slides, code, etc.)

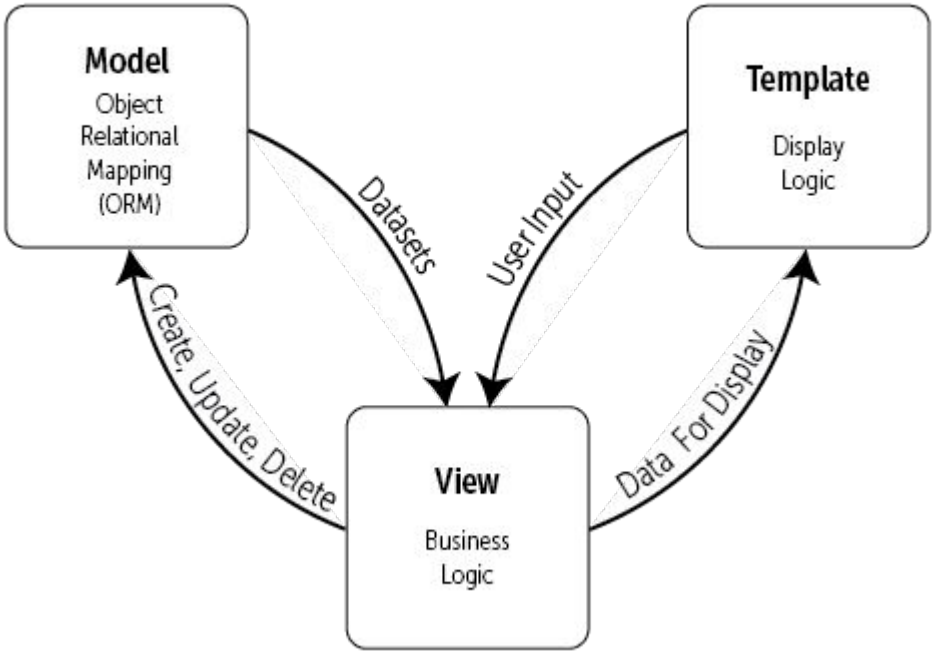
The plan

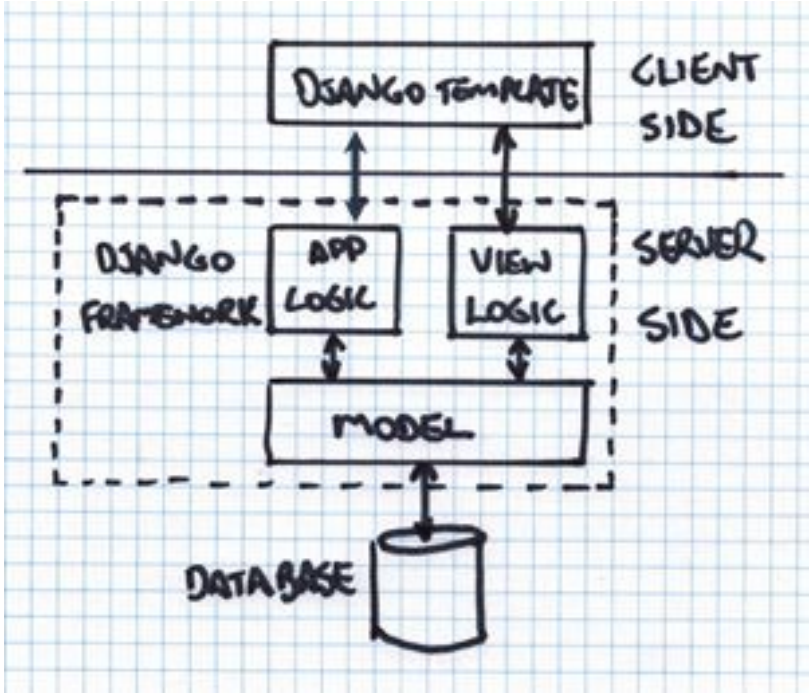
1. Briefly about docker and django
2. First steps with docker
3. Starting a project
4. Implementation
5. Production-ready configuration

Briefly about technologies used

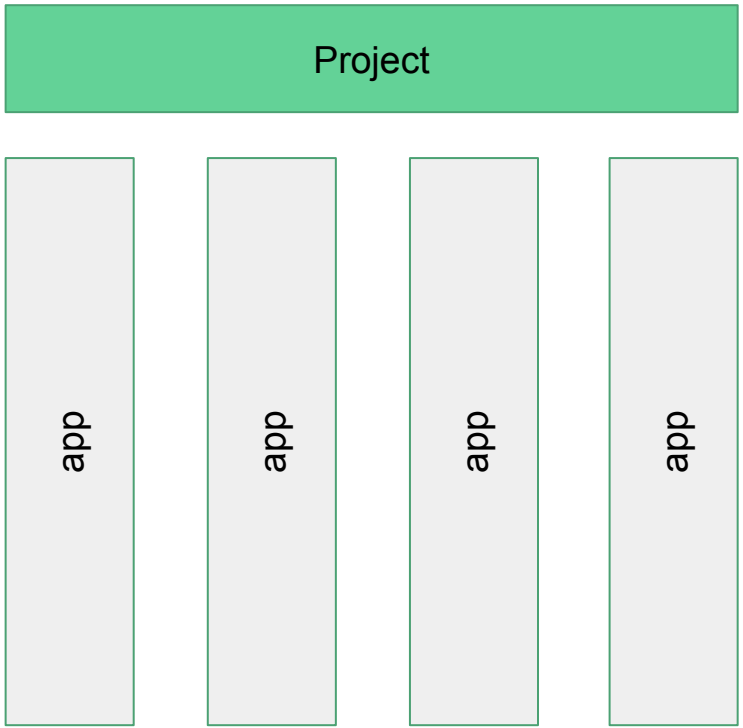
1. Most popular Python web framework
2. Big community
3. Functionalities out of the box
4. So many answers on stack overflow!
5. Ton of community packages

MVC?


Model
Object
Relational
Mapping
(ORM)
Template
Display
Logic
Datasets
User Input
Create, Update, Delete
Data For Display
View
Business
Logic


DJANGO TEMPLATE
CLIENT SIDE
DJANGO FRAMEWORK
APP LOGIC
VIEW LOGIC
SERVER SIDE
MODEL
DATABASE

Project structure

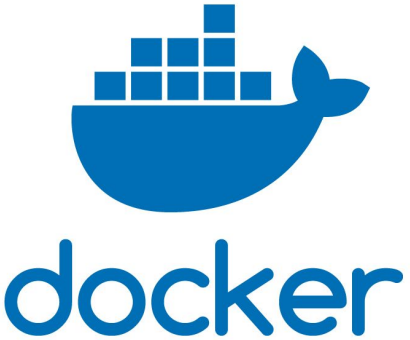


Directory structure:

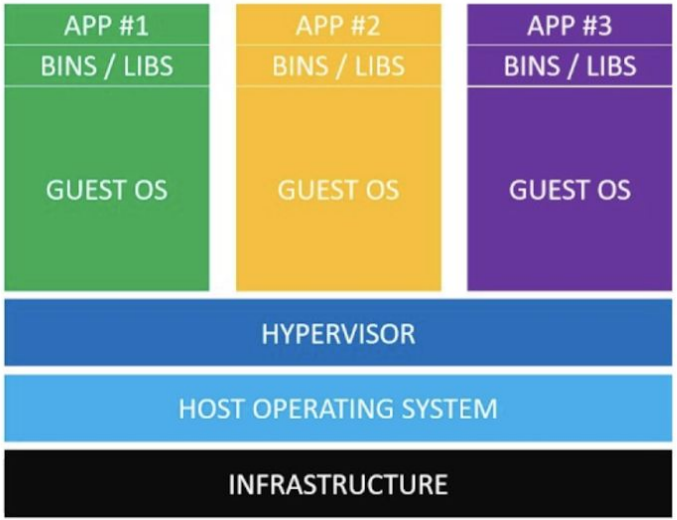
> project_name
>> project_name
>> app1
>> app2

Why docker?

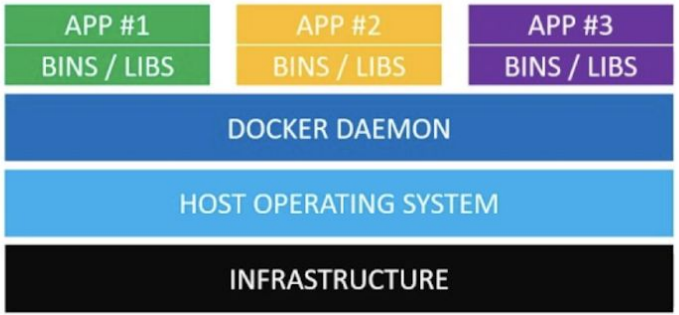
1. Automated development and production environment creation and packaging.
2. Encapsulation.
3. No performance overhead.
4. Easy way to share your own and run systems.
5. Ready to use, pre-configured tools available (DBs, etc.)
6. Environment emulation.



How it works?



Virtual Machines



Docker Containers

Glossary

Image - lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Comprises of layers.

Container - in principle a running image.

Dockerfile - a file describing an image (“recipe”), in general, each command is a layer.

Let’s code!

Prerequisites

Install docker

https://docs.docker.com/install/

If you’re running linux, instal docker-compose

https://docs.docker.com/compose/install/

Quick docker intro (postgresql)

Up to speed with docker

Let’s use postgres as an example.

1. Search: `postgres docker` and go to docker hub
2. Based on the documentation run a command

```
docker run --name mypg -e POSTGRES_PASSWORD=postgres \
-p 5432:5432 -d --rm postgres
```

3. List running containers `docker ps`
4. Stop the container `docker stop mypg`
5. What happens?

Command breakdown

```
docker run \
    --name mypg \
    -e POSTGRES_PASSWORD=postgres \
    -p 5432:5432 \
    -d
    --rm
    postgres
```

base command
name the container
set postgres password
export the port
run detached
remove after running
what are we running

Too much options to type in? - try compose

docker-compose - a tool to define and run multi-container configurations

```
version: '3'
services:
  db:
    image: postgres:alpine
    ports:
      - "5432:5432"
    volumes:
      - ./local-folder:/var/lib/postgresql/data/
    environment:
      POSTGRES_PASSWORD: postgres
```

Essential compose commands

docker-compose up

docker-compose stop

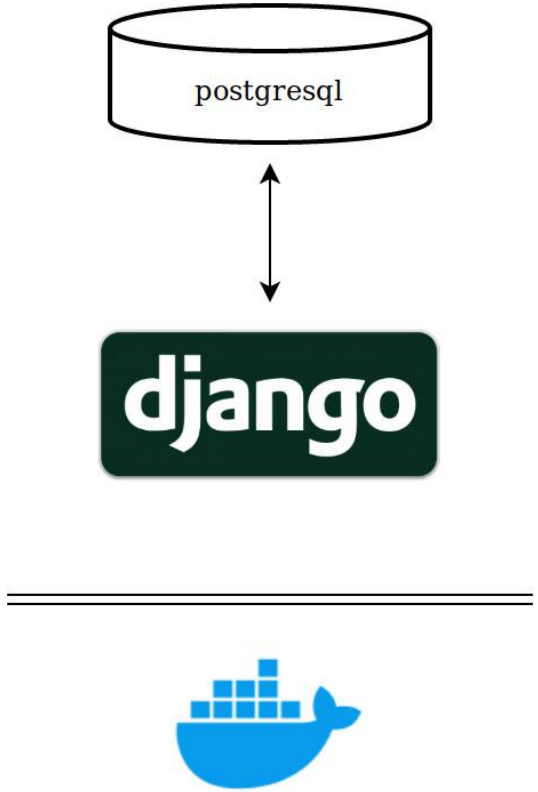
docker-compose run NAME

docker-compose build

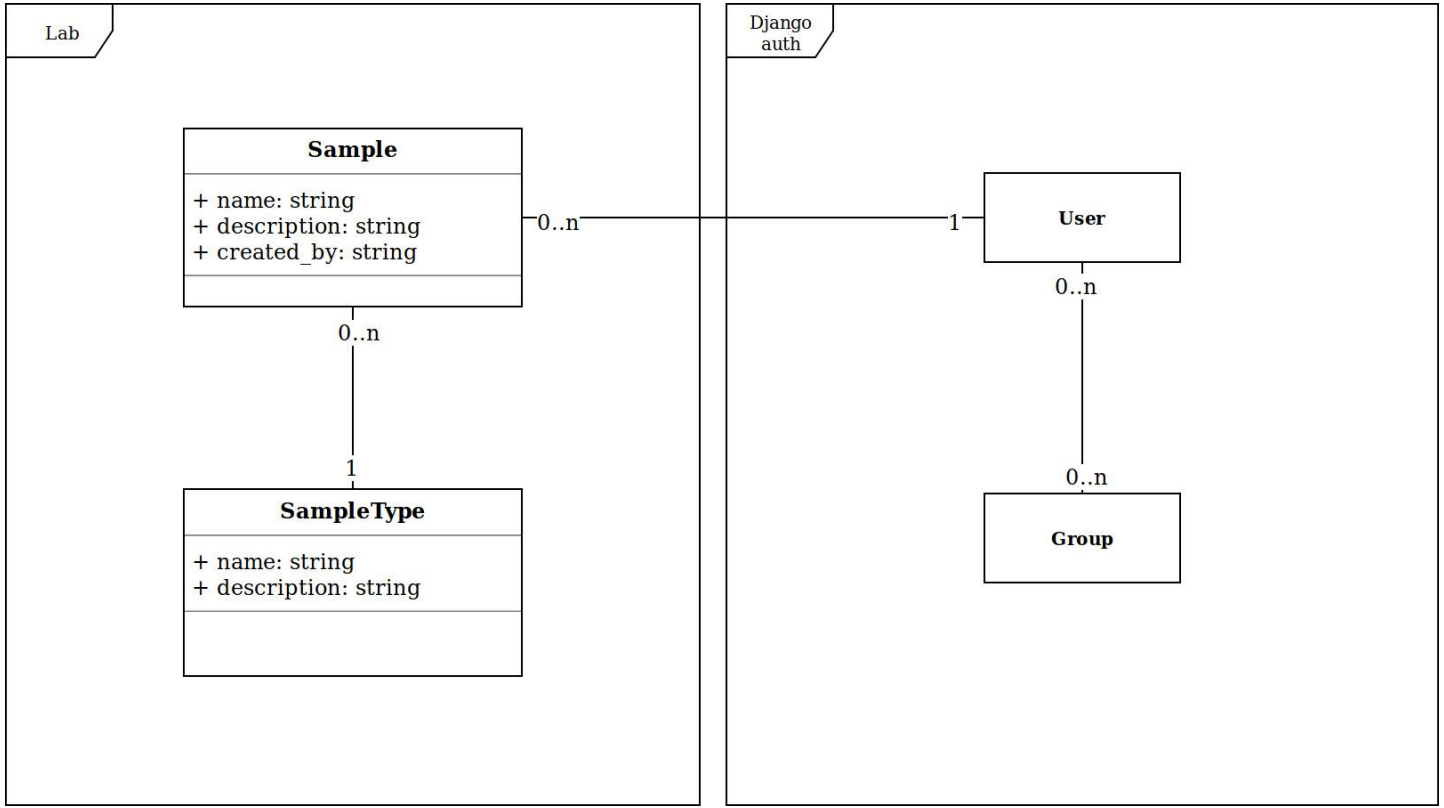
Starting django project

The app

1. Managing “samples”
2. User management
3. Authentication
4. Run on top of docker ->
5. Use as many out of the box django features a we can



Database schema



Let’s get started

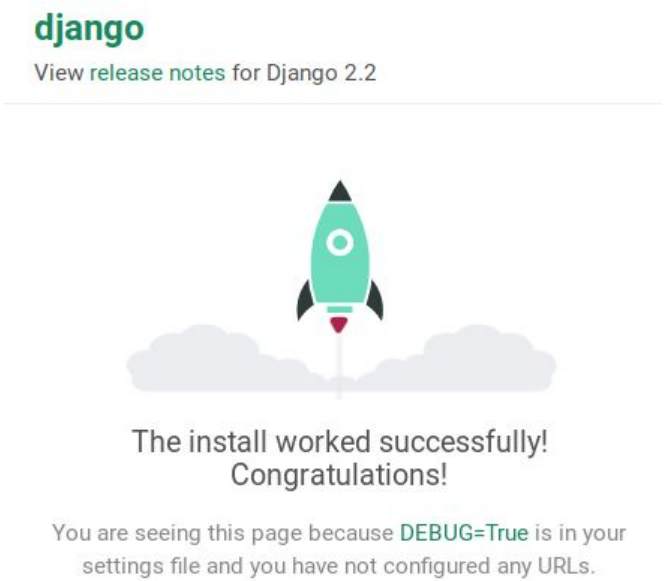
Search: docker-compose django

Go to:
https://docs.docker.com/compose/django/

Follow the steps adjusting them to your taste.

Start django project

docker-compose run web django-admin startproject lab .



Configure the app to use our db

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'postgres',
        'USER': 'postgres',
        'PASSWORD': 'postgres',
        'HOST': 'db',
        'PORT': 5432,
    }
}
```

Migrate database (create base schema)

docker exec -it django-workshop-app_web_1 ./manage.py migrate

Admin panel

Create admin user

docker exec -it django-workshop-app_web_1 bash

./manage.py createsuperuser

Your first django app (module)

Creating an app

./manage.py startapp samples

Take a look at what was created

...

First view

[samples/views.py]

```
from django.http import HttpResponse

def index(request):
   return HttpResponse("Hello, world. Index page.")
```

Setting up urls

[lab/urls.py]

```
path('samples/', include('samples.urls')),
```

[samples/urls.py]

```
from django.urls import path
from . import views
urlpatterns = [
   path('', views.index, name='index'),
]
```

Templates pt. 1

[settings.py]

```
INSTALLED_APPS = [
  ...
  'samples.apps.SamplesConfig',
]
```

[samples/views.py]

```
def other_page(request):
   context = {
       'test': 'passing a value to the template'
   }
   return render(request, 'samples/other.html', context)
```

Templates pt. 2

[samples/templates/samples/other.html]

```
<!DOCTYPE html>
<html>
<head></head>
<body>
   <h1>Some page</h1>
   <p>Some paragrapgh</p>
   <p>{{ test }}</p>
</body>
</html>
```

Template inheritance

Real-life templates

- Template inheritance
- We do not want plain html
- Let’s throw in some bootstrap
  https://getbootstrap.com/docs/4.3/getting-started/introduction/
- Loading static files
  - `{% load static %}`
- Let’s be different a bit?
  https://bootswatch.com/pulse/

Models

Models

- Models - database layout with metadata and operations
  - ‘Definitive source of truth about the data’

```
class Sample(models.Model):
   name = models.CharField(max_length=200)
   description = models.TextField(blank=True)
   type = models.ForeignKey(SampleType, on_delete=models.CASCADE)
   created_by = models.ForeignKey(User, on_delete=models.CASCADE)
```

Migrating schema

- ./manage.py makemigrations
- ./manage.py migrate

Register models in admin

```
admin.site.register(Model)
```

Change admin site name

```
admin.site.site_name = ‘Lab admin’
```

Display models properly in admin

```
class Sample(models.Model):
  ...

   def __str__(self):
       return self.name
```

Model API

Play around with commands

```
./manage.py shell
from samples.models import Sample, SampleType
Sample.objects.all()
stype = SampleType.objects.all().first()
from django.contrib.auth.models import User
usr = User.objects.all().first()
sample = Sample(name='Heart tissue', type=stype, created_by=usr)
sample.save()
sample.created_by
sample.created_by_id
sample.name = 'Liver tissue'
sample.save()
sample = Sample.objects.get(pk=2)
sample.delete()
```

Writing simple views

List view (1)

```
from .models import Sample


def index(request):
    samples = Sample.objects.all()
    return render(request, 'samples/index.html', {"samples": samples})
```

Writing simple views (2)

```
{% for sample in samples %}
<tr>
    <td>{{sample.id}}</td>
    <td>{{sample.name}}</td>
    <td>{{sample.type}}</td>
    <td>{{sample.created_by}}</td>
</tr>
{% endfor %}
```

Datatables: https://datatables.net/examples/styling/bootstrap4

Detail view

```
def detail(request, sample_id):
   sample = get_object_or_404(Sample, pk=sample_id)
   return render(request, 'samples/detail.html', {'sample': sample})
```

```
path('<int:sample_id>/', views.detail, name='detail'),
```

Inserting data

New django elements

1. Cross site request forgery protection

   ```
   {% csrf_token %}
   ```

2. Checking request method

   ```
   if request.method == "POST":
   ```

3. Getting POST data

   ```
   name = request.POST['name']
   ```

4. Redirecting

   ```
   return redirect('index')
   ```

Class based views

1. Import class to inherit from
   from django.views import View
2. Subclass
   class CreateSampleAltView(View):
3. Get support
   def get(self, request):
4. Post support
   def post(self, request):

Generic views

Generic views - TemplateView

```
from django.urls import path

from django.views.generic import TemplateView


urlpatterns = [

    path('about/', TemplateView.as_view(template_name="about.html")),

]
```

Generic views for models

Your basic CRUDs are taken care of:
(Create, Retrieve, Update, Delete)

- **C:** CreateView
- **R:** DetailView, ListView
- **U:** UpdateView
- **D:** DeleteView

Create View

Class

```
class SampleCreateView(generic.CreateView):
   model = Sample
   fields = '__all__'
   success_url = reverse_lazy('index')
```

FORM

```
   <form method="post">{% csrf_token %}
       {{ form.as_p }}
       <input type="submit" value="Save">
   </form>
```

Create form - bootstrap

```
pip install django-crispy-forms

INSTALLED_APPS = [
    # [...]
    'crispy_forms'
]

CRISPY_TEMPLATE_PACK = 'bootstrap4'

{% load crispy_forms_tags %}

{{ form|crispy }}
```

Update View

```
class SampleUpdateView(generic.UpdateView):
   model = Sample
   fields = '__all__'

   def get_success_url(self):
       return reverse_lazy('detail', args=[self.kwargs['pk']])
```

More generic views

Guide:

https://docs.djangoproject.com/en/2.2/topics/class-based-views/

List:

https://docs.djangoproject.com/en/2.2/ref/class-based-views/

Django forms - check it out

Authorization

Docs

https://docs.djangoproject.com/en/2.2/topics/auth/

User-related views

```
path('accounts/', include('django.contrib.auth.urls')),
```

http://localhost:8000/accounts/login/

```
...missing template
```

Create new folder:

app/templates/registration

And file:

app/templates/registration/login.html

Login template

Official docs is the source:

https://docs.djangoproject.com/en/2.2/topics/auth/default/#django.contrib.auth.views.LoginView

```
LOGIN_REDIRECT_URL = '/samples/'
LOGOUT_REDIRECT_URL = '/accounts/login'
```

User-dependent links

```
{% if user.is_superuser %}
<li class="nav-item">
  <a class="nav-link" href="{% url 'admin:index' %}">Admin</a>
</li>
{% endif %}


<ul class="nav navbar-nav ml-auto">
  {% if user.is_authenticated %}
  <li class="nav-item mr-1">Witaj, {{ user.get_username }} </li>
  <li class="nav-item"><a href="{% url 'logout'%}?next={{request.path}}">wyloguj</a></li>
  {% else %}
  <li class="nav-item"><a href="{% url 'login'%}?next={{request.path}}">zaloguj</a></li>
  {% endif %}
</ul>
```

Add middleware to secure all views

https://stackoverflow.com/a/46976284

```
LOGIN_REQUIRED_URLS = (
    r'(.*)',
)
LOGIN_REQUIRED_URLS_EXCEPTIONS = (
    r'/admin(.*)$',
    r'/accounts(.*)$',
)
```

Testing

Production?

What makes a production deployment?

1. Production grade database
2. Production settings
3. Production grade server
4. Security
5. Efficiency

start.sh

```
#!/bin/bash

# Start Gunicorn processes
echo Starting Gunicorn.
exec gunicorn clinicaldb.wsgi:application \
    --bind 0.0.0.0:8000 \
    --workers 2
```

Serving static files

Whitenoise

http://whitenoise.evans.io/en/stable/

```
'whitenoise.middleware.WhiteNoiseMiddleware',
```

Nginx

Self signed certificates/add your own:

https://www.digitalocean.com/community/tutorials/how-to-create-a-self-signed-ssl-certificate-for-nginx-in-ubuntu-18-04

Automatic certificates?

Automate getting https certificates with nginx and let’s encrypt’s certbot.

https://medium.com/@pentacent/nginx-and-lets-encrypt-with-docker-in-less-than-5-minutes-b4b8a60d3a71

Next steps?

Resources

1. Official django tutorial (and docs in general) https://docs.djangoproject.com/en/2.2/intro/tutorial02/
2. Django crash course by Traversy https://www.youtube.com/watch?v=D6esTdOLXh4
3. Next part of this tutorial

Q&A