Curso Avanzado de Django

Artículo

Tutorial de despliegue de la aplicación

Pablo Trinidad 23 de Enero de 2019

Introducción

En esta lectura seguiremos todos los pasos necesarios para lanzar nuestra aplicación
a producción. El proceso de deployment es extremadamente sencillo por lo que, aunque
en este tutorial estaremos usando AWS, migrar nuestro stack a cualquier otra plataforma
en la nube con cualquier otro OS no implica ningún esfuerzo sustancial.

Recuerda que todo el objetivo de usar docker es tener un contrato claro con el OS que
nos permitiera liberar la aplicación sin tener que realizar ningún cambio al código fuente.
Por otro lado, y como vimos en los 12 principios de una app, la
configuración es el único elemento de nuestra aplicación
que cambiará dependiendo el entorno de ejecución.

Setup

AWS (EC2)

1. Crear una instancia.
   Durante la creación de la instancia debes:
   - Asignarle un rol
     que tenga el permiso **`AmazonS3FullAccess`**
   - Asegurarte que el *security group* de la instancia tenga abiertos los puertos 22 (SSH), 80 (HTTP) y 443 (HTTPS).

2. Contectarte a la instancia
   y actualizar los paquetes del servidor con los comandos:
   - `sudo apt-get update -y`
   - `sudo apt-get upgrade -y`

AWS (S3 Bucket)

El stack que tenemos está listo para manejar la media (archivos subidos por los usuarios) desde un
servicio externo llamada S3. AWS S3 prometee la distribución óptima de archivos, así como su simple
administración.

Empezar a utilizar S3 es bastante sencillo:

1. Crea un bucket.

2. Agrega una política
   al bucket como la siguiente:

```
{
   "Id": "Policy1548096183802",
   "Version": "2012-10-17",
```

```
            "Action": [
              "s3:GetObject"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:s3:::BUCKET-NAME/*",
            "Principal": "*"
          }
        ]
      }
```

Recuerda reemplazar **`BUCKET_NAME`** con el nombre de tu bucket. También puedes obtener el
ARN de tu bucket
revisando la descripción general.

Docker

Docker es el verdadero héroe en todo esto. El tutorial se puede resumir en: 1) Insertar las variables
de entorno. 2) docker-compose build y 3) docker-compose up.

Es necesario que instales los siguientes paquetes en el servidor:

- Docker 1.10+
- Docker Compose 1.6+

Dominio

La configuración específica de los registros DNS del dominio varía según el proveedor pero
realmente se traduce a crear los siguientes 2 registros:

- `A | www | IP`
- `A | @ | IP`

Lo cual quiere decir que crearás dos registros de tipo ADDRESS para tu dominio desnudo (domain.com)
y tu subdominio www (www.domain.com) y los dos apuntarán a la IP de la instancia.
Cuando estés realizando la configuración ten en mente que este cambio puede tomar entre 5 minutos y 48 horas. Personalmente nunca he tenido que esperar más de 5 minutos pero tenlo en mente.

El valor de **`IP`** lo puedes obtener en el detalle de tu instancia en el dashboard de AWS EC2, aparecerá
como *`IPv4 Public IP`*.

Si la administración de los registros DNS de tu dominio se vuelve un *pain in the ass*, este
puede ser un muy buen punto para que comiences a explorar servicios de Name Servers. Te recomiendo
mirar Cloudflare y AWS Route 53.

Código

Para esta lectura tengo preparado un "bundle" final con los cambios que hemos realizado en
el curso. Aquí también puede ser un buen momento para explorar múltiples herramientas que
automaticen la carga del código fuente, así como su instalación en el servidor. Estoy hablando
de CI y CD. Por el momento lo descargaremos de la siguiente manera:

Ubicados en el home de ubuntu (`/home/ubuntu/`) corremos los siguientes comandos:

1. Descargamos el archivo:

   ```
   curl -o bundle.tar.gz
   https://codeload.github.com/pablotrinidad/cride-
   platzi/tar.gz/Deployment-bundle
   ```

2. Lo descomprimimos: `tar -xzf bundle.tar.gz`
3. Lo renombramos: `mv cride-platzi-Deployment-bundle/ cride`

Al terminar esta sección contarás con docker instalado en el servidor, tu dominio apuntando
a la IP de la instancia, un bucket de S3 listo para recibir y servir achivos, y el código fuente ubucado en **`/home/ubuntu/cride/`**

Instalación de la app

Antes de comenzar, revisa el archivo `production.yml` en la raíz del proyecto. Recuerda
que este archivo tiene toda la configuración necesaria para iniciar la aplicación e incluye
los siguientes servicios:

- **`django`**: La aplicación corriendo bajo `Gunicorn`.
- **`postgres`**: Base de datos de PostgreSQL con los datos de la aplicación.
- **`redis`**: Una instancia de Redis para cache.
- **`caddy`**: Un servidor HTTP que se encargará de recibir las peticiones del exterior y usa HTTPS por default.
- **`celeryworker`**: Corre el worker de Celery.
- **`celerybeat`**: Corre el beat de Celery.
- **`flower`**: Corre Flower, una herramienta para visualizar celery en la web.

Futuros pasos y cierre del cur...

La mayoría de los servicios anteriores son configruados a través de variables de entorno por lo que
tenemos que comenzar creándolas dentro de nuestro servidor.

Primero necesitamos un folder para almacenarlas, el cuál podemos conseguir corriendo (dentro de `/home/ubuntu/cride`)

```
mkdir -p .envs/.production
```

Para que nuestra aplicación funcione necesitamos los siguientes archivos de variables:

```
touch .envs/.production/.django
touch .envs/.production/.postgres
touch .envs/.production/.caddy
```

Y dentro de cada uno, con tu editor de texto favorito (vi o nano), escribir las siguientes variables

```
# .postgres

# PostgreSQL
POSTGRES_HOST=postgres
POSTGRES_PORT=5432
POSTGRES_DB=cride
POSTGRES_USER=SECURE_USER
POSTGRES_PASSWORD=SECURE_PASSWORD


# .django

# Django
DJANGO_SETTINGS_MODULE=config.settings.production
DJANGO_DEBUG=False
DJANGO_SECRET_KEY=SECURE_SECRET_KEY
```

```
# Admin
DJANGO_ADMIN_URL=SECURE_ADMIN_URL

# E-Mail
MAILGUN_API_KEY=SECRET_KEY
MAILGUN_DOMAIN=SECRET_KEY


# Redis
REDIS_URL=redis://redis:6379/0

# Flower
CELERY_FLOWER_USER=SECURE_USER
CELERY_FLOWER_PASSWORD=SECURE_PASSWORD



# .caddy

DOMAIN_NAME=comparteride.com
```

Es **MUY IMPORTANTE** que notes que algunas de las variables no tienen valores y es tu trabajo poner valores reales y seguros. Como comentario adicional, tu ADMIN_URL
debe terminar en `/`, es decir, si tu URL es `supersecretadminurl`, entonces el valor de `DJANGO_ADMIN_URL` debe ser `supersecretadminurl/`.

Adicionalmente tengo un script de Python que puedes usar para generar secretos de manera
segura, este script tiene múltiples opciones de generación pero por default escupe un
hash de SHA256 codificado en base64. Puedes consultar el script aquí: https://gist.github.com/pablotrinidad/857eb828fd54ecee636a957e8219e8ac, es muy fácil de usar.

Launch the thing already!

Como mencioné anteriorment, gracias a que el proyecto usa Docker podemos simplificar
mucho el proceso de deployment. Al igual que en desarrollo, los comandos que tenemos
que correr son los de construcción (build) y el de ejecución (up). Vayamos al grano:

1. La primer tarea que ejecuta Django es juntar los estáticos, pero como nosotros no hemos
   utilizado ninguno, tenemos que crear un folder vacío por el momento: `mkdir /home/ubuntu/cride/static`
2. Después construimos el stack con `sudo docker-compose -f production.yml build`
3. Ahora podemos inicializar la base de datos con `sudo docker-compose -f production.yml run --rm django python manage.py migrate`
4. Aprovechamos para crear un super usuario con `sudo docker-compose -f production.yml run --rm django python manage.py createsuperuser`
5. Corremos la aplicación con `sudo docker-compose -f production.yml up`
6. Listo (casi)!

Ahora que el servidor está arriba te recomiendo probar que funcione correctamente
realizando lo siguiente:

1. Inicia sesión en el admin
2. Crea un usuario llamando al API (`POST /users/signup/`)
3. Verifica que el email se haya enviado usando Mailgun. (Tu cuenta tiene que estar ACTIVADA)
4. Completa la verificación de cuenta de Comparte Ride y actualiza la foto de

Cerrando la sesión SSH

Como ya sabes, en el momento en el que cierres la sesión SSH con el servidor el proceso
que mantiene a Docker corriendo será terminado. Para evitar esto podríamos usar `docker-compose -f production.yml up -d` lo cuál ejecuta el stack en modo *"detached"*, pero
realmente no resuelve el problema. Si el proceso termina su ejecución necesitamos poder estar
seguros de que se va a poder recuperar sin necesidad de iniciarlo manualmente. Para lograr esto
usaremos una herramienta llamada supervisor que nos permitirá registrar nuestra aplicación
como un servicio de Systemd. Para realizar esto tenemos que seguir los siguientes pasos:

1. Instalar supervisor: `sudo apt-get install supervisor -y`
2. Iniciar sesión como super user: `sudo su -`
3. Crear el archivo `cride.conf` dentro de `/etc/supervisor/conf.d/` con el siguiente contenido:

```
[program:cride]
command=docker-compose -f production.yml up
directory=/home/ubuntu/cride
redirect_stderr=true
autostart=true
autorestart=true
priority=10
```

4. Registrar el servicio en supervisor con `supervisorctl reread` y luego
   `supervisorctl update`

6. Verificar que el servicio esté corriendo con `supervisorctl status`

Podemos verificar que todo esté funcionando correctamente reiniciando el servidor desde AWS.

> al terminar el arranque del OS tu sitio está arriba, terminaste!

onclusión

52 ealmente el proceso de configuración del servidor es ajeno al proyecto,
53 nalmente lo único
ue tuvimos que hacer para lanzar nuestra aplicación es correr `docker-compose`
54 `uild` y `docker-compose up`.

55 hora que ya tienes tu servicio corriendo te recomiendo muchísimo leer la
56 ocumentación del proyecto base
)jango Cookiecutter) sobre deployment para fortalecer tu conocimiento sobre el
57 ack.

58 ecuerda que cada que hagas una actualización y después de haber descargado el
59 ódigo lo único
ue tienes que hacer es `docker-compose build` y `supervisorctl restart cride`.
60

61 Escribe aquí tu pregunta +2

ibacrea Estudiante • hace 9 meses

al cambiar el user y pass de posgresql, me da error, hay otro archivo con esos datos?

1

Futuros pasos y cierre del cur...