Práctica 3.4. Despliegue de una aplicación Flask (Python)

1. Introducción

1.1 ¿Qué es un *framework*?

Actualmente en el desarrollo moderno de aplicaciones web se utilizan distintos *Frameworks* que son herramientas que nos dan un esquema de trabajo y una serie de utilidades y funciones que nos facilita y nos abstrae de la construcción de páginas web dinámicas.

En general los *Frameworks* están asociado a lenguajes de programación (Ruby on Rails (Ruby), Symphony (PHP)), en el mundo de Python el más conocido es Django pero Flask es una opción que quizás no tenga una curva de aprendizaje tan elevada, pero nos posibilita la creación de aplicaciones web igual de complejas de las que se pueden crear en Django.

1.2 Flask

En la actualidad existen muchas opciones para crear páginas web y muchos lenguajes (PHP, JAVA), y en este caso Flask nos permite crear de una manera muy sencilla aplicaciones web con Python.

Flask es un “micro” Framework escrito en Python y concebido para facilitar el desarrollo de Aplicaciones Web bajo el patrón MVC.

La palabra “micro” no designa a que sea un proyecto pequeño o que nos permita hacer páginas web pequeñas, sino que al instalar Flask tenemos las herramientas necesarias para crear una aplicación web funcional, pero si se necesita en algún momento una nueva funcionalidad hay un conjunto muy grande extensiones (*plugins*) que se pueden instalar con Flask que le van dotando de funcionalidad.

En la instalación no se tienen todas las funcionalidades que se pueden necesitar, pero de una manera muy sencilla se pueden extender el proyecto con nuevas funcionalidades por medio de *plugins*.

El patrón MVC es una manera o una forma de trabajar que permite diferenciar y separar lo que es el modelo de datos (los datos que van a tener la App que normalmente están guardados en BD), la vista (página HTML) y el controlador (donde se gestiona las peticiones de la app web).

1.3 Gunicorn

Cuando se implementa una aplicación web basada en Python, normalmente se tienen estas tres piezas:

- Servidor web (Nginx, Apache)
- Servidor de aplicaciones WSGI (Gunicorn, uWSGI, mod_wsgi, Waitress)
- Framework utilizado para el desarrollo (Django, Flask, Pyramid, FastAPI)

Los servidores web procesan y distribuyen las solicitudes de los navegadores y otros clientes y envían respuestas a los mismos.

WSGI (*Web Server Gateway Interface*) proporciona un conjunto de reglas para estandarizar el comportamiento y la comunicación entre servidores web y aplicaciones web. Mediante el uso de servidores y aplicaciones web compatibles con WSGI, los desarrolladores pueden concentrar su tiempo y energía en el desarrollo de aplicaciones web en lugar de administrar la comunicación entre la aplicación y el servidor web.

Finalmente, Gunicorn, que es la abreviatura de *Green Unicorn*, es un servidor de aplicaciones WSGI que se encuentra entre el servidor web y su aplicación web, gestionando la comunicación entre los dos. Acepta solicitudes del servidor y las traduce (a través de WSGI) en algo que la aplicación web puede entender antes de pasarla a la aplicación web real. Envía respuestas desde la aplicación web al servidor. También se encarga de ejecutar varias instancias de la aplicación web, reiniciándolas según sea necesario y distribuyendo solicitudes a instancias saludables.

1.4 Gestor de paquete pip

`pip` es el comando para instalar paquetes de Python integrados en las fuentes desde la versión 3.4.

Este comando automatiza la conexión al sitio https://pypi.org/, la descarga, la instalación e incluso la compilación del módulo solicitado. Además, se ocupa de las dependencias de cada paquete.

1.5 Entornos virtuales en Python

Un entorno virtual es una forma de tener múltiples instancias paralelas del intérprete de Python, cada una con diferentes conjuntos de paquetes y diferentes configuraciones. Cada entorno virtual contiene una copia independiente del intérprete de Python, incluyendo copias de sus utilidades de soporte.

Los paquetes instalados en cada entorno virtual sólo se ven en ese entorno virtual y en ningún otro. Incluso los paquetes grandes y complejos con binarios dependientes de la plataforma pueden ser acorralados entre sí en entornos virtuales.

De esta forma, tendremos entornos independientes entre sí, parecido a como ocurría con los directorios de los proyectos de Node.js. De este modo, los **entornos virtuales** de Python nos **permiten instalar un paquete de Python en una ubicación aislada en lugar de instalarlo de manera global.**

PIPENV

Pipenv es una herramienta que apunta a traer todo lo mejor del mundo de empaquetado (*bundler, composer*, npm, cargo, yarn, etc.) al mundo de Python.

Automáticamente crea y maneja un entorno virtual para tus proyectos, también permite agregar/eliminar paquetes desde tu Pipfile así como como instalar/desinstalar paquetes. También genera lo más importante, el archivo *Pipfile.lock*, que es usado para producir determinado *build*.

2. Procedimiento completo para el despliegue

1. Instalamos el gestor de paquetes de Python pip:

```
sudo apt-get update

sudo apt-get install python3-pip
```

2. instalamos el paquete *pipenv* para gestionar los entornos virtuales:

```
sudo apt install pipenv
```

3. Y comprobamos que está instalado correctamente mostrando su versión:

```
pipenv --version
```

4. Creamos el directorio en el que almacenaremos nuestro proyecto:

```
sudo mkdir /var/www/nombre_mi_aplicacion
```

5. Al crearlo con sudo, los permisos pertenecen a *root*. Hay que cambiarlo para que el dueño sea nuestro usuario (*sergio* en mi caso) y pertenezca al grupo `www-data`, el usuario usado por defecto por el servidor web:

```
sudo chown -R $USER:www-data /var/www/mi_aplicacion
```

6. Establecemos los permisos adecuados a este directorio, para que pueda ser leído por todo el mundo:

```
chmod -R 775 /var/www/mi_aplicacion
```

7. Dentro del directorio de nuestra aplicación, creamos un archivo oculto `.env` que contendrá las variables de entorno necesarias:

```
touch .env
```

8. Editamos el archivo y añadimos las variables, indicando cuál es el archivo *.py* de la aplicación y el entorno, que en nuestro caso será producción:

```
sergio@Debian-DAW:/var/www/ejemplo_flask$ cat .env
FLASK_APP=wsgi.py
FLASK_ENV=production
```

9.Iniciamos ahora nuestro entorno virtual. *Pipenv* cargará las variables de entorno desde el fichero .env de forma automática:

```
pipenv Shell
```

Veremos que se nos inicia el entorno virtual, cosa que comprobamos porque aparece su nombre al inicio del *prompt*:

```
sergio@Debian-DAW:/var/www/ejemplo_flask$ pipenv shell
Loading .env environment variables...
Loading .env environment variables...
Creating a Pipfile for this project...
Launching subshell in virtual environment...
sergio@Debian-DAW:/var/www/ejemplo_flask$  . /home/sergio/.local/share/virtualenvs/ejemplo_flask-tlSrtaXB/bin/activate
(ejemplo_flask) sergio@Debian-DAW:/var/www/ejemplo_flask$
```

10. Usamos *pipenv* para instalar las dependencias necesarias para nuestro proyecto (al tener activo el entorno virtual *ejemplo_flask* todas las dependencias se instalarán en ese entorno):

```
pipenv install flask gunicorn
```

11. Vamos ahora a crear la aplicación *Flask* más simple posible, a modo de *PoC* (*proof of concept*). El archivo que contendrá la aplicación propiamente dicha será **application.py** y **wsgi.py** se encargará únicamente de iniciarla y dejarla corriendo:

```
touch application.py wsgi.py
```

Y tras crear los archivos, los editamos para dejarlos así:

File: application.py

```
from flask import Flask

app = Flask(__name__)


@app.route('/')
def index():
    '''Index page route'''

    return '<h1>Aplicacion desplegada</h1>'
```

File: wsgi.py

```
from application import app

if __name__ == '__main__':
    app.run(debug=False)
```

12. Corramos ahora nuestra aplicación a modo de comprobación **con el servidor web integrado de Flask**. Si especificamos la dirección 0.0.0.0 lo que le estamos diciendo al servidor es que escuche en todas sus interfaces, si las tuviera:

```
(ejemplo_flask) sergio@Debian-DAW:/var/www/ejemplo_flask$ flask run --host '0.0.0.0'
 * Tip: There are .env files present. Install python-dotenv to use them.
 * Serving Flask app 'wsgi.py'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a prod
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.1.39:5000
Press CTRL+C to quit
```

Ahora podremos acceder a la aplicación desde nuestro ordenador, nuestra máquina anfitriona, introduciendo en un navegador web: http://IP-maq-virtual:5000:

Si lo haces en AWS EC2 habrás de abrir el puerto correspondiente en el grupo de seguridad

13. Comprobemos ahora que Gunicorn funciona correctamente también. Si os ha funcionado el servidor de desarrollo de Flask, podéis usar el siguiente comando para probar que la aplicación funciona correctamente usando Gunicorn, accediendo con vuestro navegador de la misma forma que en el paso anterior:

```
gunicorn --workers 4 --bind 0.0.0.0:5000 wsgi:app
```

Donde:

- ***--workers N***. Establece el número de *workers* o hilos que queremos utilizar, como ocurría con Node Express. Dependerá del número de *cores* que le hayamos dado a la CPU de nuestra máquina virtual.
- ***--bind 0.0.0.0:5000***. Hace que el servidor escuche peticiones por **todas** sus interfaces de red y en el puerto 5000
- ***wsgi:app.*** Es el nombre del archivo con extensión .py y app es la instancia de la aplicación Flask dentro del archivo.

14. Todavía **dentro de nuestro entorno virtual**, debemos tomar nota de cuál es el ***path*** o ruta desde la que se ejecuta *gunicorn* para poder configurar más adelante un servicio del sistema. Podemos averiguarlo así:

```
(ejemplo_flask) sergio@Debian-DAW:/var/www/ejemplo_flask$ which gunicorn
/home/sergio/.local/share/virtualenvs/ejemplo_flask-tlSrtaXB/bin/gunicorn
```

Y tras ello debemos salir de nuestro entorno virtual con el sencillo comando `deactivate`

15. Puesto que ya debemos tener instalado Nginx en nuestro sistema, lo iniciamos y comprobamos que su estado sea activo.

```
sudo systemctl start nginx

sudo systemctl status nginx
```

16. Ya fuera de nuestro entorno virtual, crearemos un archivo para que systemd[1] corra Gunicorn como un **servicio del sistema más**:

```
(ejemplo_flask) sergio@Debian-DAW:/var/www/ejemplo_flask$ sudo cat /etc/systemd/system/flask_app.service
[Unit]
Description=Flask app service - ejemplo_flask con Gunicorn
After=network.target

[Service]
# Usuario/grupo que ejecutan la app
User=sergio
Group=www-data

# Carpeta del proyecto
WorkingDirectory=/var/www/ejemplo_flask/

# Activamos el venv en el entorno
Environment="PATH=/home/sergio/.local/share/virtualenvs/ejemplo_flask-tlSrtaXB/bin"

# Gunicorn con socket UNIX y 3 workers
ExecStart=/home/sergio/.local/share/virtualenvs/ejemplo_flask-tlSrtaXB/bin/gunicorn \
  --workers 3 \
  --bind unix:/var/www/ejemplo_flask/ejemplo_flask.sock \
  wsgi:app

# Opcionales recomendados
Restart=on-failure
RestartSec=5
ExecReload=/bin/kill -s HUP $MAINPID
KillSignal=SIGQUIT
TimeoutStopSec=30

[Install]
WantedBy=multi-user.target
```

Donde:

- **User**: Establece el usuario que tiene permisos sobre el directorio del proyecto (el que pusisteis en el paso 5)
- **Group**: Establece el grupo que tiene permisos sobre el directorio del proyecto (el que pusisteis en el paso 5)
- **Environment**: Establece el directorio bin (donde se guardan los binarios ejecutables) dentro del entorno virtual (lo vistéis en el paso 14)
- **WorkingDirectory**: Establece el directorio base donde reside nuestro proyecto
- **ExecStart**: Establece el *path* donde se encuentra el ejecutable de *gunicorn* dentro del entorno virtual, así como las opciones y comandos con los que se iniciará[2].

Debéis cambiar los valores para que coincidan con los de vuestro caso particular.

[1] `/etc/systemd/system` es el lugar recomendado para *units personalizados*
`/usr/lib/systemd/system` es el lugar donde se instalan los units que vienen con los paquetes de la distro (por ejemplo: `nginx.service` o `ssh.service`)

[2] El –bind le dice a Gunicorn en qué dirección debe escuchar las peticiones entrantes. Esta vez, en lugar de una IP_puerto, lo hará a través de un archivo socket.

Ahora, como cada vez que se crea un servicio nuevo de *systemd*, se habilita y se inicia:

```
systemctl enable nombre_mi_servicio

systemctl start nombre_mi_servicio

systemctl status nombre_mi_servicio
```

```
(ejemplo_flask) sergio@Debian-DAW:/var/www/ejemplo_flask$ systemctl status flask_app.service
● flask_app.service - Flask app service - ejemplo_flask con Gunicorn
     Loaded: loaded (/etc/systemd/system/flask_app.service; enabled; preset: enabled)
     Active: active (running) since Fri 2025-09-12 19:10:50 CEST; 6min ago
 Invocation: 256ba5f733d7452999e70e5b52d9c63a
   Main PID: 11455 (gunicorn)
      Tasks: 4 (limit: 4619)
     Memory: 62M (peak: 62.7M)
        CPU: 476ms
```

1. Pasemos ahora a configurar **Nginx**, que es algo que ya deberíamos tener dominado de capítulos anteriores.

Creamos un archivo con el nombre de nuestra aplicación y dentro estableceremos la configuración para ese sitio web. El archivo, como recordáis, debe estar en `/etc/nginx/sites-available/nombre_aplicacion` y tras ello lo editamos para que quede de la siguiente forma. **¿Qué es el .sock[3]?**

```
server {

    listen 80;

    server_name mi_aplicacion www.mi_aplicacion;

    access_log /var/log/nginx/mi_aplicacion.access.log;

    error_log /var/log/nginx/mi_aplicacion.error.log;

    location / {

            include proxy_params;

            proxy_pass http://unix:/var/www/nombre_aplicacion/nombre_aplicacion.sock;

    }

}
```

[3] El socket (`/var/www/ejemplo_flask/ejemplo_flask.sock`) es un archivo especial en el sistema de ficheros que actúa como canal de comunicación entre Nginx y Gunicorn. Se trata de un *Unix Domain Socket* (UDS); Se comporta como un puerto TCP, pero en vez de usar una dirección IP y un número de puerto, usa una ruta en el sistema de ficheros.
Se usa en este socket en vez de redirigir por IP:puerto porque es la comunicación es más rápida ya que se queda dentro del *kernel* y no pasa por la pila TCP/IP y es más seguro ya que no expones un puerto abierto

2. Recordemos que ahora debemos crear un link simbólico del archivo de sitios webs disponibles al de sitios web activos:

```
sudo ln -s /etc/nginx/sites-available/nombre_aplicacion /etc/nginx/sites-enabled/
```

Y nos aseguramos de que se ha creado dicho *link* simbólico:

```
ls -l /etc/nginx/sites-enabled/ | grep nombre_aplicacion
```

3. Nos aseguramos de que la configuración de Nginx no contiene errores, reiniciamos Nginx y comprobamos que se estado es activo:

```
nginx -t

sudo systemctl restart nginx

sudo systemctl status nginx
```

4. Ahora nos queda comprobar que podemos acceder. Puesto que aún no hemos tratado con el DNS, vamos a editar el archivo `/etc/hosts` de nuestra máquina anfitriona para que asocie la IP de la máquina virtual, a nuestro `server_name`.

Este archivo, en Linux, está en: `/etc/hosts`

Y en Windows: `C:\Windows\System32\drivers\etc\hosts`

Y deberemos añadirle la línea:

```
192.168.X.X myproject www.myproject
```

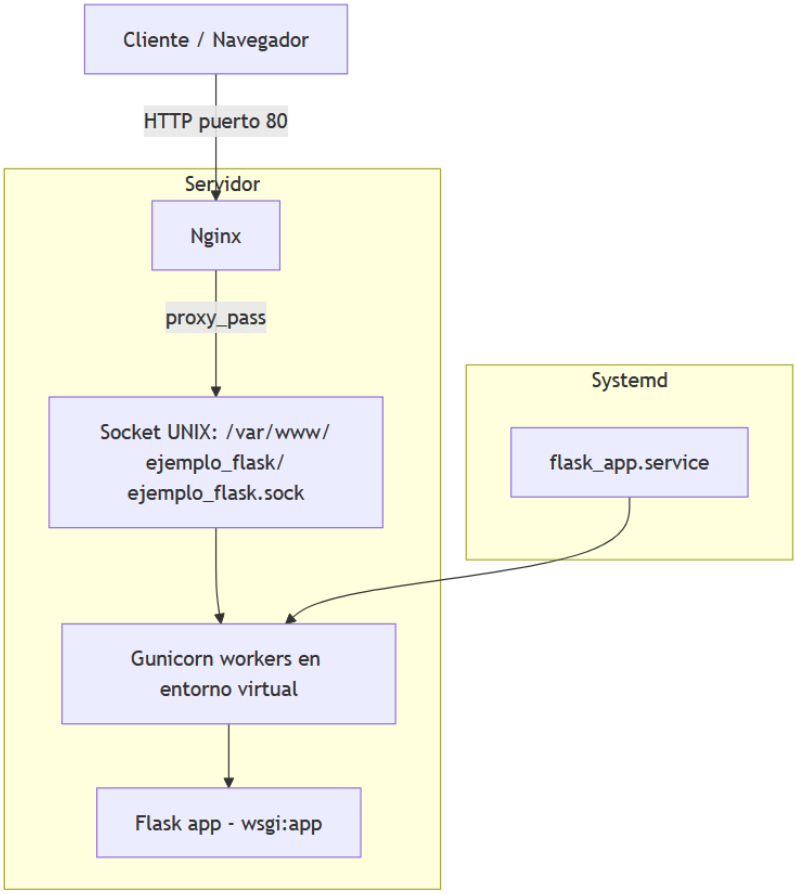
El último paso es comprobar que todo el despliegue se ha realizado de forma correcta y está funcionando, para ello accedemos desde nuestra máquina anfitrión a:

```
http://nombre_aplicacion
```

O:

```
http://www.nombre_aplicacion
```

Así quedaría el diagrama de la aplicación:



TAREA

1. ¿Qué es un *unit* en el contexto de Linux?

2. Explica cada línea del archivo de configuración del *unit* que has creado

3. Busca, lee, entiende y explica qué es y para qué sirve un servidor *WSGI*

4. Documenta adecuadamente con explicaciones y capturas de pantalla los procesos de despliegue de ambas aplicaciones en Flask, así como las respuestas a las cuestiones planteadas.

Acuérdate de realizar el vídeo en que se muestre que el servicio creado está activo y su correcto funcionamiento.