AWAKELAB

Desarrollo Web Python con Django

Desarrollador de aplicaciones Full Stack

Python Trainee

6.1.- Contenido 1: Describir las características fundamentales del framework Django para el desarrollo de aplicaciones empresariales acorde al entorno Python

Objetivo de la jornada

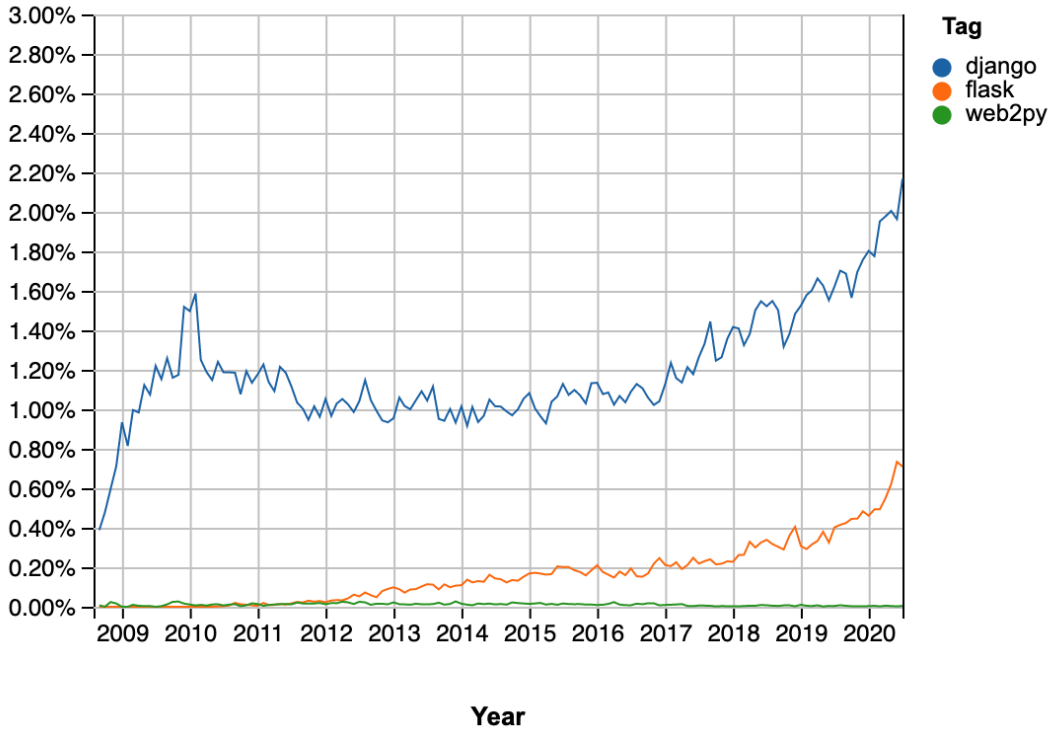
1. Reconoce las características, ventajas y potencialidades de Django y su utilidad para el desarrollo de aplicaciones empresariales bajo el entorno Python.
2. Distingue la aplicación de tareas con Python puro versus Django integrado para el desarrollo de aplicaciones Web
3. Reconoce el procedimiento de Incorporación de Django como integrador de un proyecto Web
4. Instala los componentes Django utilizando el utilitario PIP para la preparación del entorno
5. Aplica procedimiento de creación de un nuevo proyecto Django utilizando el utilitario manage.py para generar la estructura inicial del proyecto

6.1.1.- Introducción a Django

1.6.1.1. Qué es Django

A partir de su documentación oficial:
Django es un framework web Python de alto nivel que fomenta el desarrollo rápido y un diseño limpio y pragmático. Creado por desarrolladores experimentados, se encarga de gran parte de la molestia del desarrollo web, por lo que puede concentrarse en escribir su aplicación sin necesidad de reinventar la rueda. Es gratis y de código abierto.

Naturalmente existen otras alternativas a Django como Flask o web2py que, aunque no son exactamente similares (full-stack) son también populares, pero definitivamente django es el framework más popular dentro de los frameworks que usan Python. Abajo se ve un gráfico de tendencias de uso en el mundo de los frameworks en todos los lenguajes. Django claramente lleva la delantera y va en subida.



En 2005 los desarrolladores web de Lawrence Journal World crearon Django para ayudar a los periodistas a poner historias en la Web rápidamente. Ahora, se utiliza en una amplia variedad de sitios web y aplicaciones, como Instagram, Pinterest y el Washington Times.

1.6.1.2. Por qué usar Django

A partir de su documentación oficial:

- Es rápido
- Seguro
- Escalable

Algunas ventajas de usar Django que se pueden enumerar aquí:

- **Soporte Object-Relational Mapping (ORM):** Django proporciona un puente entre el modelo de datos y el motor de base de datos, y admite un gran conjunto de sistemas de bases de datos, incluidos MySQL, Postgres, SQLite, MariaDB, Oracle. Django también admite bases de datos NoSQL a través de la bifurcación Django-nonrel. Por ahora, las únicas bases de datos NoSQL compatibles son MongoDB, cassandraDB y SimpleDB y otros.

- **Soporte multilingüe:** Django admite sitios web multilingües a través de su sistema de internacionalización integrado. Para que pueda desarrollar un sitio web, que admita varios idiomas.

- **Soporte del framework:** Django tiene soporte integrado para Ajax, RSS, almacenamiento en caché y varios otros framworks.

- **GUI de administración:** Django proporciona una agradable interfaz de usuario lista para su uso en actividades administrativas.

- **Entorno de desarrollo:** Django viene con un servidor web ligero para facilitar el desarrollo y las pruebas de aplicaciones de un extremo a otro.

1.6.1.3. Python y los entornos virtuales. Por qué usar entornos virtuales

Python, como la mayoría de los otros lenguajes de programación modernos, tiene su propia forma única de descargar, almacenar y resolver paquetes (o módulos). Si bien esto tiene sus ventajas, se tomaron algunas decisiones interesantes sobre el almacenamiento y la resolución de paquetes, lo que ha dado lugar a algunos problemas, particularmente con cómo y dónde se almacenan los paquetes.
Hay algunas ubicaciones diferentes donde estos paquetes se pueden instalar en su sistema. Por ejemplo, la mayoría de los paquetes del sistema se almacenan en un directorio secundario de la ruta almacenada en **sys.prefix**.

Por ejemplo, en MacOSx

```
>>> import sys
>>> sys.prefix
'/usr/local/Cellar/python@3.8/3.8.5/Frameworks/Python.framework/Versions/3.8'
```

O en debian

```
>>> import sys
>>> sys.prefix
'/usr'
```

De forma más relevante para nuestros propósitos, los paquetes de terceros instalados usando **`easy_install`** o **`pip`** (compruebe que tiene alguna de estas dos utilidades; de preferencia **`pip python3-pip`**; **`easy_install`** ya está en desuso) generalmente se colocan en uno de los directorios señalados por **`site.getsitepackages`**:

```
>>> import site
>>> site.getsitepackages()
['/usr/local/Cellar/python@3.8/3.8.5/Frameworks/Python.framework/Versi
ons/3.8/lib/python3.8/site-packages']
```

1.6.1.4. Instalaciones en el entorno global

Es importante saber esto que acabamos de ver porque, de forma predeterminada, todos los proyectos de su sistema utilizarán estos mismos directorios para almacenar y recuperar **paquetes (site packages** bibliotecas de terceros). A primera vista, esto puede no parecer un gran problema, y realmente no lo es, para los paquetes del sistema (**system packages** paquetes que son parte de la biblioteca estándar de Python), pero sí es importante para los **site packages**. Nota: utilizaremos **packages** en lugar de paquetes para que, entre otros, la información que se muestra efectivamente en pantalla, resulte más entendible.

Considere el siguiente escenario donde tiene dos proyectos: ProyectoA y ProyectoB, los cuales tienen una dependencia en la misma biblioteca, ProyectoC. El problema se hace evidente cuando comenzamos a requerir diferentes versiones de ProyectoC. Quizás ProyectoA necesite la v1.0.0, mientras que ProyectoB requiera la nueva v2.0.0, por ejemplo.

Este es un problema real para Python, ya que no puede diferenciar entre versiones en el directorio **site-packages**. Entonces, tanto la v1.0.0 como la v2.0.0 residirán en el mismo directorio con el mismo nombre:

/System/Library/Frameworks/Python.framework/Versions/3.5/Extras/lib/python/ProyectoC

Dado que los proyectos se almacenan solo de acuerdo con su nombre, no hay diferenciación entre versiones. Por lo tanto, se requeriría que ambos proyectos, ProyectoA y ProyectoB, usen la misma versión, lo cual es inaceptable en muchos casos.

Aquí es donde entran en juego los entornos virtuales y las herramientas **virtualenv/venv.**

En esencia, el propósito principal de los entornos virtuales de Python es crear un entorno aislado para los proyectos de Python. Esto significa que cada proyecto puede tener sus propias dependencias, independientemente de las dependencias que tengan los demás proyectos.

En nuestro pequeño ejemplo anterior, solo necesitaríamos crear un entorno virtual separado para ProyectoA y ProyectoB, y estaríamos listos para comenzar. Cada entorno, a su vez, podría depender de la versión de ProyectoC que elijan, independientemente del otro.

Lo mejor de esto es que no hay límites para la cantidad de entornos que puede tener, ya que son solo directorios que contienen algunos scripts. Además, se crean fácilmente con las herramientas de línea de comandos **virtualenv** o con el switch **-m env** en Python mismo.

1.6.1.5. El entorno virtual

El comando venv; Iniciando el entorno virtual; Saliendo del entorno virtual

Vamos a hacer un poco de repaso de lo que vimos en la primera clase del módulo 2 en relación a este tema. (sección VIRTUALENV)

Si está utilizando Python 3, entonces ya debería tener instalado el módulo **venv** de la biblioteca estándar.

Comience creando un nuevo directorio para trabajar, con el nombre que usted desee:

```
$ mkdir <directorio para trabajar>
$ cd <directorio para trabajar>
```

Cree un nuevo entorno virtual dentro del directorio:

```
$ python3 -m venv env
```

El nombre **env** puede ser reemplazado, pero se ha transformado en una especie de convención implícita.

Nota: De forma predeterminada, el entorno virtual no incluirá ninguno de sus paquetes de sitio existentes.

El enfoque **venv** de Python 3 tiene la ventaja de obligar a elegir una versión específica del intérprete de Python 3 que se debe utilizar para crear el entorno virtual. Esto evita cualquier confusión sobre en qué instalación de Python se basa el nuevo entorno.

Desde Python **3.3** a **3.4**, la forma recomendada de crear un entorno virtual era utilizar la herramienta de línea de comandos **pyvenv** que también viene incluida con su instalación de Python 3 de forma predeterminada. Pero en **3.6** y superior, **python3 -m venv** es el camino a seguir.

En el ejemplo anterior, este comando crea un directorio llamado env, que contiene una estructura de directorio similar a esta:

```
├── bin
│   ├── Activate.ps1
│   ├── activate
│   ├── activate.csh
│   ├── activate.fish
│   ├── easy_install
│   ├── easy_install-3.8
│   ├── pip
│   ├── pip3
│   ├── pip3.8
│   ├── python -> python3
│   └── python3 -> /usr/local/bin/python3
├── include
├── lib
│   └── python3.8
│       └── site-packages
└── pyvenv.cfg
```

Esto es lo que contiene cada carpeta:

- **bin**: archivos que interactúan con el entorno virtual
- **include**: encabezados C que compilan los paquetes de Python
- **lib**: una copia de la versión de Python junto con una carpeta de paquetes del sitio donde se instala cada dependencia

Además, hay copias de, o enlaces simbólicos a, algunas herramientas de Python diferentes, así como a los propios ejecutables de Python. Estos archivos se utilizan para garantizar que todo el código y los comandos de Python se ejecuten dentro del contexto del entorno actual, que es cómo se logra el aislamiento del entorno global. Explicaremos esto con más detalle más adelante.

Más interesantes son los scripts de activación en el directorio bin. Estos scripts se utilizan para configurar su shell para que utilice el ejecutable Python del entorno y sus **site-packages** de forma predeterminada.

Para utilizar los paquetes/recursos de este entorno de forma aislada, debe "activarlo". Para hacer esto, simplemente ejecute lo siguiente:

```
$ source env/bin/activate
(env) $
```

Observe cómo su mensaje ahora tiene como prefijo el nombre de su entorno (env, en nuestro caso). Este es el indicador de que **env** está actualmente activo, lo que significa que el ejecutable de Python solo usará los paquetes y la configuración de este entorno.

Para volver al contexto del "sistema" ejecutamos deactivate:

```
$ source env/bin/activate
```

```
(env) $
```

1.6.1.6. Flexibilidad de instalación y configuración

Para nuestros fines vamos a dar por sentado que usted tiene instalado Python 3.x y trabajaremos con la versión Django 3.1

Dentro de su directorio de trabajo (creado para estos propósitos **<directorio para trabajar>**) y con el entorno virtual **activado** ejecutaremos lo siguiente:

```
$ pip install Django==3.1
```

Esto basta para instalar Django y algunas de sus dependencias:

```
$ pip install Django==3.1
...
Successfully installed Django-3.1 asgiref-3.2.10 pytz-2020.1
sqlparse-0.3.1
```

1.6.1.7. El entorno de desarrollo Django

El entorno de desarrollo es una instalación de Django en su computadora local que puede utilizar para desarrollar y probar aplicaciones Django antes de implementarlas en un entorno de producción.

Las principales herramientas que proporciona el propio Django son un conjunto de scripts de Python para crear y trabajar con proyectos de Django, junto con un servidor web de desarrollo simple que puede usar para probar aplicaciones web de Django locales (es decir, en su computadora, no en un servidor web externo) en el navegador web de su computadora.

Hay otras herramientas periféricas, que forman parte del entorno de desarrollo, que no cubriremos específicamente en este apartado, ya que, las hemos visto con anterioridad. Estos incluyen cosas como un editor de texto o IDE para editar código y una herramienta de administración de control de fuente como Git para administrar de manera segura diferentes versiones de su código.

1.6.1.8. Opciones de configuración de Django

Django es extremadamente flexible en términos de cómo y dónde se puede instalar y configurar. Django puede ser:

- Instalado en diferentes sistemas operativos.

- Instalado desde la fuente, desde el índice de paquetes de Python (**PyPi**) y en muchos casos desde la aplicación del administrador de paquetes de la computadora host.
- Configurado para usar una de varias bases de datos, que también pueden necesitar instalarse y configurarse por separado.
- Ejecutarse en el entorno Python del sistema principal o dentro de entornos virtuales Python independientes.

Cada una de estas opciones requiere una inicialización y configuración ligeramente diferentes. Para el resto de esta clase, le mostraremos cómo configurar Django en una pequeña cantidad de sistemas operativos, y esa configuración se asumirá en el resto de la clase/módulo.

Primero que todo, para ratificar que estamos en condiciones de trabajar con django, ejecutemos

```
(env) <user>:<folder>$ python -m django –version
3.1
```

Si obtuvo **3.1** estamos listos para continuar. Nota: ejecutamos **python** dentro del **virtualenv** lo que significa en nuestro caso que estamos ejecutando **python3** (**3.8** para ser más específicos, aunque eso no debería afectar mayormente lo que vamos a hacer mientras se trate de **python3**)

Ahora probaremos la instalación, si la prueba anterior funciona, una prueba más interesante es crear un proyecto esqueleto y verlo funcionando. Para hacer esto, usaremos el terminal, recuerde que estamos en la carpeta donde desea almacenar esta aplicación de Django.

Ahora ejecute

```
$ django-admin startproject sitiodeprueba
```

Verá que se creó la sub-carpeta **sitiodeprueba**, diríjase a esa sub-carpeta

```
$ cd sitiodeprueba
```

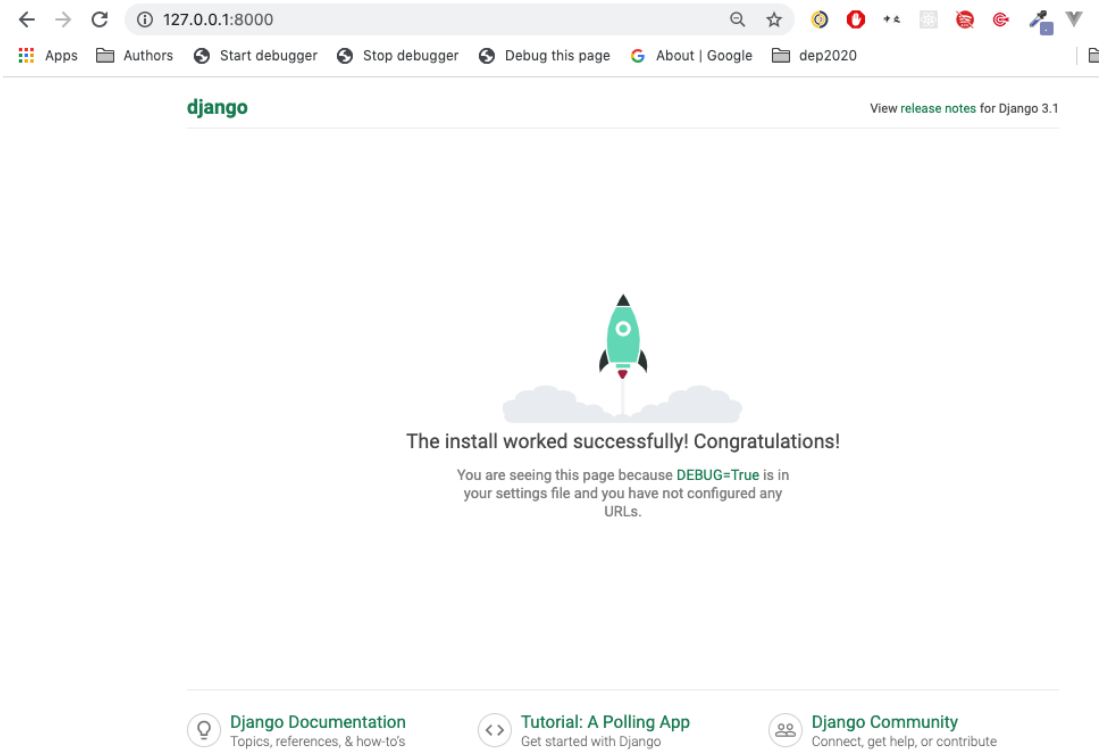
Podemos ejecutar el "servidor web de desarrollo" desde esta carpeta usando **manage.py** y el comando **runserver**, como se muestra:

```
$ python manage.py runserver

Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
```

```
You have 18 unapplied migration(s). Your project may not work properly until
you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
August 27, 2020 - 15:20:38
Django version 3.1, using settings 'sitiodeprueba.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Nota: por lo pronto no se preocupe de las **migrations**.

Una vez que el servidor se está ejecutando, puede ver el sitio navegando a la siguiente URL en su navegador web local: http://127.0.0.1:8000/. Debería ver un sitio que se parece a esto:



A la vez en el terminal podrá ver un registro de eventos y mensajes, mientras mantenga el servidor funcionando.

1.6.1.9. Entorno desarrollo v/s producción

Lo que vimos anteriormente nos garantiza un entorno de desarrollo, lo usaremos para desarrollar nuestra aplicación en nuestra máquina de forma local.
Pero una vez que su sitio esté terminado (o terminado "lo suficiente" para comenzar las pruebas públicas), tendrá que alojarlo en un lugar más público y accesible que su computadora de desarrollo personal.

Supongamos que hemos estado trabajando en un entorno de desarrollo, utilizando el servidor web de desarrollo Django para compartir su sitio con el navegador/red local

y ejecutando su sitio web con configuraciones de desarrollo (inseguras) que exponen la depuración (debug) y otra información privada. Antes de poder alojar un sitio web de forma externa, primero tendrá que:

- Realizar algunos cambios en la configuración de su proyecto.
- Elegir un entorno para alojar la aplicación Django.
- Elegir un entorno para alojar archivos estáticos.
- Configurar una infraestructura de nivel de producción para servir su sitio web.

El entorno de producción es el entorno proporcionado por un computador "servidor" donde ejecutará su sitio web para consumo externo. El entorno incluye:

- Hardware en el que se ejecuta el sitio web.
- Sistema operativo (por ejemplo, Linux, Windows).
- Tiempo de ejecución para el lenguaje de programación y bibliotecas del framework sobre las cuales está escrito el sitio web.
- Servidor web utilizado para servir páginas y otro contenido (por ejemplo, Nginx, Apache).
- Eventualmente un servidor de aplicaciones pasa solicitudes "dinámicas" entre su sitio web Django y el servidor web.
- Bases de datos de las que depende su sitio web.

Nota: Dependiendo de cómo esté configurada su producción, también puede necesitar proxy inverso (reverse proxy), un equilibrador de carga (load balancer), etc.

El servidor podría estar ubicado en sus instalaciones y conectado a Internet mediante un enlace rápido, pero es mucho más común usar un servidor alojado "en la nube". Lo que esto significa en realidad es que su código se ejecuta en una computadora remota (o posiblemente una computadora "virtual") en los centros de datos de una empresa de alojamiento. El servidor remoto generalmente ofrecerá algún nivel garantizado de recursos informáticos (por ejemplo, CPU, RAM, memoria de almacenamiento, etc.) y conectividad a Internet por un precio determinado.

Este tipo de hardware informático accesible vía redes remotamente se denomina Infraestructura como servicio (IaaS - Infrastructure as a Service). Muchos proveedores de IaaS ofrecen opciones para preinstalar un sistema operativo en particular, en el que debe instalar los otros componentes de su entorno de producción. Otros proveedores le permiten seleccionar entornos con más funciones, tal vez incluyendo una configuración completa de Django y servidor web.

Nota: Los entornos prediseñados pueden facilitar la puesta en marcha de su sitio web porque reducen la configuración, pero las opciones disponibles pueden limitarlo a un servidor desconocido (u otros componentes) y pueden estar basadas en una versión anterior del sistema operativo. A menudo, es mejor instalar los componentes usted mismo, de modo que obtenga los que desee, y cuando necesite actualizar partes del sistema, tenga una idea de por dónde empezar.

Otros proveedores de alojamiento (hosting) admiten Django como parte de una oferta de plataforma como servicio (PaaS - Platform as a Service). En este tipo de hosting, no necesita preocuparse por la mayor parte de su entorno de producción (servidor web, servidor de aplicaciones, balanceadores de carga), ya que la plataforma de host se encarga de ellos por usted (junto con la mayor parte de lo que necesita hacer en orden para escalar su aplicación). Eso hace que la implementación sea bastante fácil, porque solo necesita concentrarse en su aplicación web y no en el resto de la infraestructura del servidor.

Algunos desarrolladores elegirán la mayor flexibilidad proporcionada por IaaS sobre PaaS, mientras que otros aprecian la reducción de la sobrecarga de mantenimiento y el escalado más fácil de PaaS. Cuando se está comenzando, configurar su sitio web en un sistema PaaS es mucho más fácil.

1.6.1.10. Django v/s Python

Recapitulando hemos visto que Django es un framework para el desarrollo de aplicaciones web que está escrito en Python.

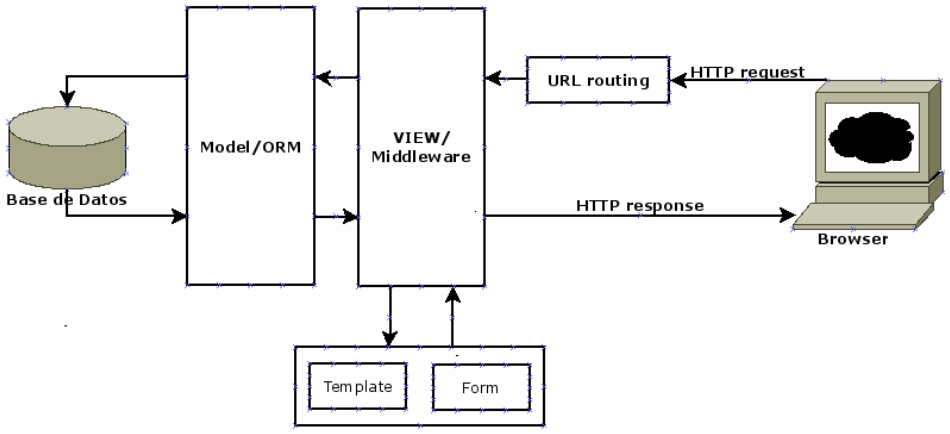
Existen bastantes versiones del framework (y del lenguaje Python también) pero para nuestros propósitos usaremos **Django 3.1** y **Python 3.x** (de preferencia **3.7** o **3.8**)

1.6.1.11. Django y la estructura Web para el desarrollo

Django usa objetos de solicitud (request) y respuesta (response) para comunicarse entre el cliente y el servidor. Como Django es un framework web, estamos hablando de objetos de solicitud y respuesta HTTP.

Cada vez que escribe una URL, su navegador realiza una solicitud a un servidor, que enviará archivos, generalmente HTML, y su navegador los procesará y mostrará. Al final del día, no importa cuán complicada sea su aplicación web, todo lo que hace es enviar archivos debido a una solicitud.

Un framework web proporciona un conjunto de herramientas de componentes reutilizables que permite a los desarrolladores concentrarse en crear su aplicación rápidamente sin reinventar la rueda. En cada aplicación web surgen problemas como cómo se asigna una determinada URL a un conjunto de código o cómo se crea una página de forma dinámica de acuerdo con algunos datos de su base de datos. Un framework web le permitirá hacerlo rápidamente para que pueda concentrarse en escribir el código para la URL dada o en el contenido de una página determinada. Estos dos problemas principales se denominan enrutamiento (routing) y plantillas (templates), respectivamente, y son componentes principales en muchos marcos web.



Nota: los frameworks web front-end son completamente diferentes.

Django es uno de los frameworks web de código abierto más populares y grandes. Está escrito en Python y compañías como Instagram, Doordash y Disqus lo usan en sus sistemas, lo que con suerte es suficiente para demostrar que Django es robusto y escalable.

1.6.1.12. Soporte para bases de datos

A partir de su documentación oficial:

Django admite oficialmente las siguientes bases de datos:

- PostgreSQL
- MariaDB
- MySQL
- Oracle
- SQLite

También hay una serie de backends de bases de datos proporcionados por terceros.

1.6.1.12. Velocidad de desarrollo

Django está construido con Python, sabemos que está familiarizado con este hecho. Solo estamos aprovechando la oportunidad para destacar algunos de los beneficios clave de Django que heredó de Python, naturalmente entre ellos la velocidad.

- Python es uno de los lenguajes más populares y en crecimiento del mundo. Según fuentes como: Análisis de datos de Indeed.com Jobs por Coding Dojo; GitHub Octoverse; Encuesta para desarrolladores de Stack Overflow 2018.

- Python es fácil de aprender. Suele ser el primer idioma elegido por los desarrolladores.

- No deje que la declaración anterior lo engañe al creer que es solo para principiantes. Python también se utiliza en tecnología de vanguardia. Muchos gigantes, incluido Google, usan Python en su tecnológica de manera extensiva.

- Python es un lenguaje muy recomendado para desarrollar web scrapers.

- Funciona bien con otros lenguajes.

    - Desarrollar con Python no significa que tenga que ceñirse a todo lo que se creó solo con Python. Puede utilizar bibliotecas creadas con muchos otros lenguajes, incluido C/C++/Java

- Python es portátil y fácil de leer

- Python incluso puede ejecutarse en JVM. Eche un vistazo a Jython.

- Python se utiliza y se admite ampliamente en tecnología de vanguardia como Big Data, Machine Learning, etc.

- Acceso a una enorme biblioteca PyPI.

“El desarrollo de programas con Python es de 5 a 10 veces más rápido que con C/C++ y de 3 a 5 veces más rápido que con Java”. - fuente: https://www.python.org/doc/essays/omg-darpa-mcc-position/

1.6.1.13. Estructura minimalista

Las "Baterías están incluidas" significa que Django viene con la mayoría de las bibliotecas y herramientas necesarias para casos de uso comunes, listas para usar. Django ORM, Middlewares, autenticación, bibliotecas HTTP, soporte para múltiples sitios, i18n, Django Admin, motor de plantillas, etc. son algunas de las "baterías". Otros frameworks no ofrecen tanto soporte listo para usar. Entonces, ¿Por qué no usar algo que está preparado, probado en batalla, que alimenta algunos de los sitios

web más grandes del planeta, desarrollado activamente y respaldado por la comunidad?

Si no necesita la mayoría de las funciones que ofrece Django, debería considerar el uso de un micro framework. No reinvente la rueda. Dedique su tiempo a las cosas que importan, deje que Django se encargue del resto.

1.6.1.14. Estructura Flexible

Hay algo importante que necesita más atención. Muchos frameworks, como Laravel, Yii, etc. han intentado facilitar el trabajo con el panel de administración. Pero ninguno de ellos está ni remotamente cerca del framework Django en términos de trabajar con el panel de administración.

Algunas personas argumentan que Django Admin no es lo suficientemente flexible y que personalizar cualquier parte requiere mucho esfuerzo. Uno puede estar de acuerdo con esta declaración durante los primeros días de usar Django, pero finalmente, cuando usted se familiarice con el framework, descubrirá que esto es incorrecto. Sí, hay una curva de aprendizaje, pero vale la pena cada segundo. Django Admin está realmente muy bien estructurado. En algunos proyectos puede usado el administrador de Django tal cual, y en otros se puede reemplazar por completo con plantillas personalizadas desarrolladas desde cero. En cualquier caso, no toma más tiempo del que se tarda si lo desarrollara con cualquier otro framework que conociera.

¿La mejor parte? Obtiene el módulo de permisos y autenticación listo para usar. Si se construye desde cero, esto podría llevar semanas (o días) al menos.

1.6.1.15. Librerías propias de cada proyecto

Al ser de código abierto e increíblemente popular, Django ha creado una gran comunidad. Sabemos las ventajas del software de código abierto. Django tiene las mismas ventajas. La documentación oficial de Django es más que suficiente para los desarrolladores. Puede encontrar soluciones fácilmente si se queda atascado. Además de eso, Stack Overflow está inundado de preguntas y respuestas relacionadas con Django.

Django tiene muchas bibliotecas propias y paquetes que puede descargar de inmediato para resolver algún problema o funcionalidad, por mencionar algunos:

1-. **sentry-sdk:** Sentry es un monitor de aplicaciones multiplataforma, con un enfoque en la notificación de errores. Resuelve esas situaciones en la que el servidor arroja un error y no hay que investigarlo. Por ejemplo, detecta cualquier excepción no detectada (lo que generalmente significa que un código 500 regresa desde el

servidor sin más información) y nos enviará detalles completos sobre el problema inmediatamente después de que suceda.

2.- **djangorestframework:** Si desea escribir una API con restricciones arquitectónicas REST, este paquete lo hará por usted, junto con la documentación adecuada generada automáticamente que respalda la implementación del paquete en proyectos.

3.- **django-extensions:** Un conjunto de herramientas que le ayudarán en su trabajo diario. JSONField se introdujo por primera vez en este paquete antes de que se convirtiera oficialmente en parte de Django 1.9 (para PostgreSQL).

4.- **django-rest-framework-jwt:** adds JWT token authorization. Jason Wen Tokens. No tendrá que implementar y administrar el token de autenticación para la API por su cuenta. Proporciona todo lo que necesita para la autenticación JWT: el punto final de inicio de sesión y la clase de autenticación Django.

5.- **django-rest-swagger:** El DRF (django rest framework) ofrece documentación generada automáticamente, y este paquete hace que la documentación sea más agradable para sus proyectos.

6.- **easy-thumbnails:** Cualquier servicio web que permita cargar y ver fotos necesita la función de miniaturas. Este paquete lo proporciona.

7.- **django-simple-history:** Este paquete mantiene un historial de cambios de registros. Si un cliente tiene acceso a la base de datos a través del panel de administración, es posible que cambie algo por error o quiera volver a la versión anterior del texto.

8.- **django-adminactions:** Una simple exportación de datos a tipos de archivos conocidos como CSV o XLS. También puede exportar datos como un dispositivo al servidor de prueba, y este dispositivo puede contener claves externas. El paquete también permite generar gráficos en el panel de administración.

9.- **django-model-utils:** Un conjunto de utilidades útiles para modelos Django. Agrega algunos modelos y campos que ya ayudan con algunos problemas comunes.

10.- **django-storages:** Permite usar cualquier servicio de almacenamiento en la nube como almacenamiento de archivos predeterminado. Eso se vuelve importante cuando desea pasar lo que los usuarios nos envían a otro almacenamiento. Por ejemplo, un usuario puede enviarnos una gran cantidad de datos importantes como fotos, películas o incluso copias de seguridad. Para que estos datos estén disponibles para ellos sin ralentizar la aplicación, es una buena idea utilizar soluciones de almacenamiento en la nube como Google Cloud.

Si quiere encontrará más utilidades para usar con django puede ver el sitio https://djangopackages.org/

Tal como en cualquier ambiente virtual de Python, los paquetes Django se instalan de la siguiente forma, por ejemplo para el paquete **djangorestframework**:

```
(venv)$ pip install djangorestframework

Collecting djangorestframework
  Downloading djangorestframework-3.11.1-py3-none-any.whl (911 kB)
     |████████████████████████████████| 911 kB 3.6 MB/s
Requirement already satisfied: django>=1.11 in
./venv3.8/lib/python3.8/site-packages (from djangorestframework) (3.1)
Requirement already satisfied: pytz in ./venv3.8/lib/python3.8/site-packages (from
django>=1.11->djangorestframework) (2020.1)
Requirement already satisfied: sqlparse>=0.2.2 in
./venv3.8/lib/python3.8/site-packages (from django>=1.11->djangorestframework)
(0.3.1)
Requirement already satisfied: asgiref~=3.2.10 in
./venv3.8/lib/python3.8/site-packages (from django>=1.11->djangorestframework)
(3.2.10)
Installing collected packages: djangorestframework
Successfully installed djangorestframework-3.11.1
```

Con lo que un nuevo comando **pip freeze** para consultar el estado de nuestro paquetes instalados daría como resultado:

```
(venv)$ pip freeze
asgiref==3.2.10
Django==3.1
djangorestframework==3.11.1
pytz==2020.1
sqlparse==0.3.1
```

Lo que, como hemos mencionado anteriormente, nos permite replicar el estado actual de nuestro desarrollo en cualquier máquina alternativa, para desarrollo o despliegue.

1.6.1.16. Aislación del entorno Python y proyectos Django

De acuerdo a lo explicado en la sección de entornos virtuales, cada uno de los paquetes mencionados más arriba podrá estar definido para un proyecto en particular con una versión distinta, según las características de éste, y los requerimientos de compatibilidad con otros paquetes que estén siendo utilizado en el mismo desarrollo.

Esto permite tener múltiples implementaciones de sus proyectos, incluso usando distintas versiones, tanto de Python como de Django. Esto sirve para probar conceptos o soluciones sin tener que interferir un proyecto, o código, que está en uso y no presenta problemas.

Un archivo básico **requirements.txt** con dependencias de Django tiene un aspecto como el que se muestra a continuación:

requirements.txt

```
asgiref==3.2.10
Django==3.1
pytz==2020.1
sqlparse==0.3.1
```

Al igual que en cualquier proyecto Python, este archivo se genera utilizando la utilidad pip, estando en el entorno virtual correspondiente al proyecto, de la siguiente forma:

```
$ pip freeze > requirements.txt
```

En caso de desear replicar las mismas dependencias en otra máquina, para trabajar paralelamente en el mismo proyecto o para desplegarlo en producción, utilizamos:

```
$ pip install -r requirements.txt
```

1.6.1.17. Compatibilidad de versiones Python y Django

Según la documentación oficial de Django:

¿Qué versión de Python puedo usar con Django?

- Django version: 1.11
  - Python versions: 2.7, 3.4, 3.5, 3.6, 3.7 (added in 1.11.17)
- Django version: 2.0
  - Python versions: 3.4, 3.5, 3.6, 3.7
- Django version: 2.1
  - Python versions: 3.5, 3.6, 3.7
- Django version: 2.2
  - Python versions: 3.5, 3.6, 3.7, 3.8 (added in 2.2.8)
- Django version: 3.0, 3.1
  - Python versions: 3.6, 3.7, 3.8

Django está comprometido con la estabilidad de la API (de Django) y la compatibilidad con versiones posteriores. En pocas palabras, esto significa que el código que desarrolle con una versión de Django seguirá funcionando con versiones futuras. Es posible que deba realizar cambios menores al actualizar la versión de Django que utiliza su proyecto: consulte la sección lanzamiento (release notes) para la versión o versiones a las que está actualizando.

1.6.1.18. El enrutador de Django

Vamos a volver un poco atrás para retomar ciertos conceptos. Recuerde que cuando ejecutó

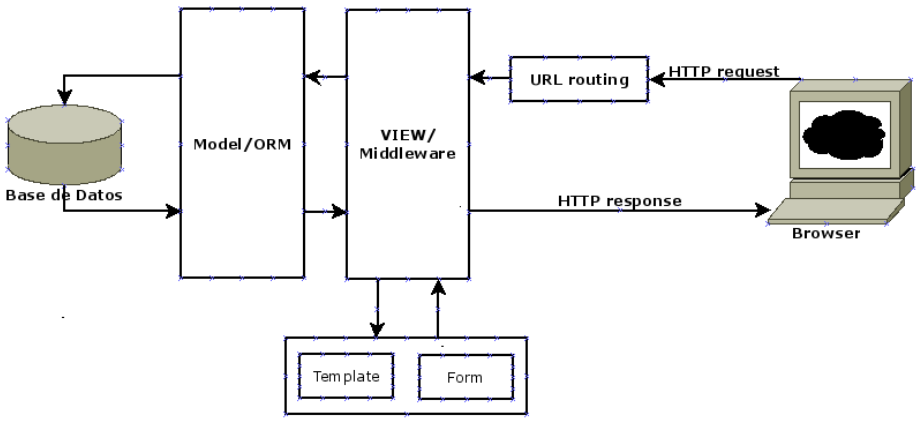
```
$ django-admin startproject sitiodeprueba
```

Se creó una (sub)carpeta llamada **sitiodeprueba**, dentro de esa carpeta tenemos la siguiente estructura

```
.
├── db.sqlite3
├── manage.py
└── sitiodeprueba
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Existe una otra carpeta **sitiodeprueba** anidada, veremos de qué se trata más adelante.

Si examinamos los archivos creados por **django-admin startproject** veremos que existe uno llamado **urls.py**, este archivo está directamente relacionado con el enrutamiento en Django. El script creó un **URLconf** automáticamente. Este módulo Python es un mapeo entre expresiones de ruta de URL a funciones Python (sus vistas **- views**).



Como vemos en la figura una vez que la solicitud (request) se produce es necesario dirigirla para que sea procesada por el código apropiado, eso es precisamente el enrutamiento.

El archivo **urls.py** debería contener algo similar a esto

```
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Lo principal a tener en cuenta aquí es la variable **urlpatterns**, que Django espera encontrar en su módulo **ROOT_URLCONF**. Esta variable define el mapeo entre las URL y el código que maneja esas URL.

La definición de tal variable se encuentra en **settings.py** del módulo mismo **(sitiodeprueba/sitiodeprueba/settings.py)**

```
...
ROOT_URLCONF = 'sitiodeprueba.urls'
...
```

1.6.1.19. MTV en Django para aplicaciones monolíticas

El patrón MVC (modelo - vista - controlador) es un patrón de arquitectura de software que separa la presentación de datos de la lógica de manejo de las interacciones del usuario (en otras palabras, le ahorra estrés :), ha existido como un concepto por un tiempo e invariablemente ha experimentado un crecimiento exponencial en su uso. desde su concepción. También se ha descrito como una de las mejores formas de crear aplicaciones cliente-servidor, todos los mejores frameworks para la web se basan en el concepto MVC.

Para desglosarlo, aquí hay una descripción general del concepto MVC;

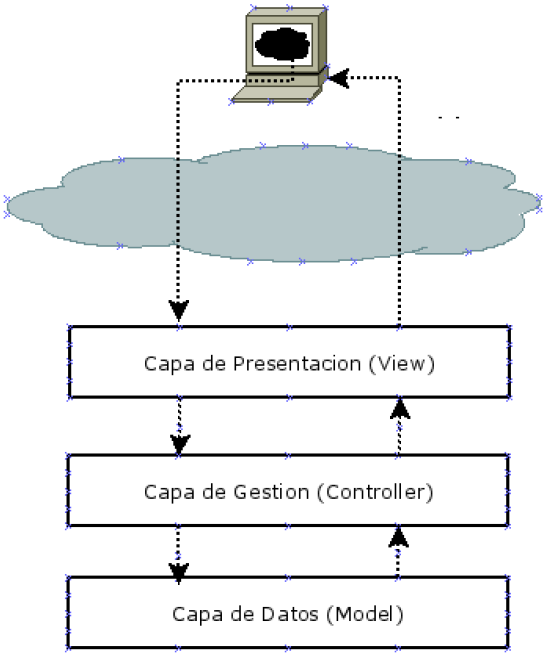
- **Modelo:** esto maneja la representación de sus datos, sirve como una interfaz para los datos almacenados en la base de datos misma y también le permite interactuar con sus datos sin tener que perturbarse con todas las complejidades de la base de datos subyacente.

- **Vista:** como su nombre lo indica, representa lo que ve en su navegador para una aplicación web o en la interfaz de usuario para una aplicación de escritorio.

- **Controlador:** proporciona la lógica para manejar el flujo de presentación en la vista o actualizar los datos del modelo, es decir, usa lógica programada para averiguar qué se extrae de la base de datos a través del modelo y se pasa a la vista, también obtiene información del usuario a través de la vista e

implementa la lógica dada ya sea cambiando la vista o actualizando los datos a través del modelo. Para hacerlo más simple, considérelo como la sala de máquinas.

Sabiendo ahora lo que es el patrón de diseño MVC, también sabemos que es una arquitectura común a muchos frameworks en distintos lenguajes. Más detalles en: https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador



MVC v/s MTV

Ahora veremos las variaciones que presenta este patrón de diseño MVC en Django particularmente. Los creadores de Django lo diseñaron para resolver un conjunto particular de problemas en una organización de noticias, originalmente se utilizó para administrar tres sitios web de noticias: The Lawrence Journal-World, lawrence.com y KUsports.com. En el centro de este conjunto de problemas había tres necesidades muy diferentes:

- La gente de datos necesitaba una interfaz común para trabajar con fuentes de datos, formatos y software de base de datos dispares.

- Los equipos de diseño necesitaban gestionar la experiencia del usuario con las herramientas que ya tenían (HTML, CSS, JavaScript, etc.).

- Los desarrolladores de código necesitaban un framework que les permitiera desplegar cambios de sistema rápidamente y mantenerlos a todos contentos.

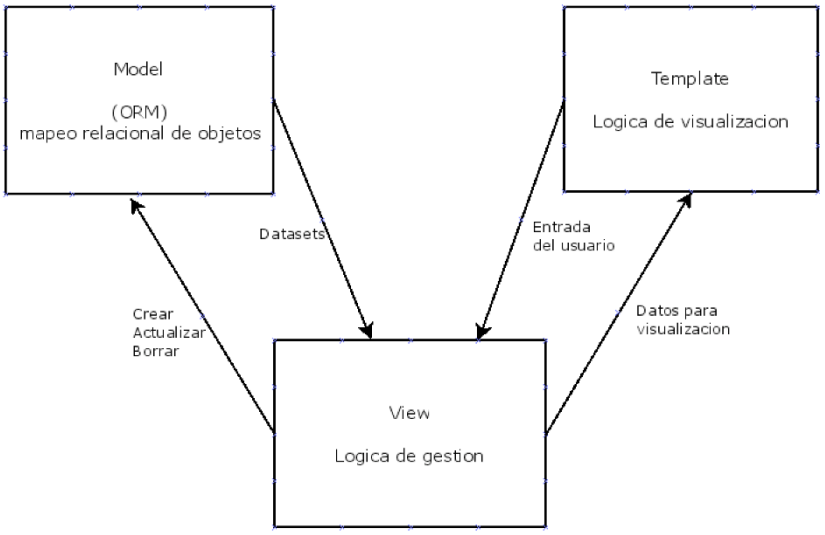
Para hacer que todo esto funcionara, era fundamental garantizar que cada uno de estos componentes básicos (datos, diseño y lógica) se pudiera gestionar de forma independiente o, para utilizar un término informático más correcto, el framework tenía que emplear un acoplamiento flexible (loose coupling).
La arquitectura de Django consta de tres partes principales:

- La **Parte 1** es un conjunto de herramientas que facilitan mucho el trabajo con datos y bases de datos.

- La **Parte 2** es un sistema de plantilla (template) de texto plano adecuado para no programadores; y

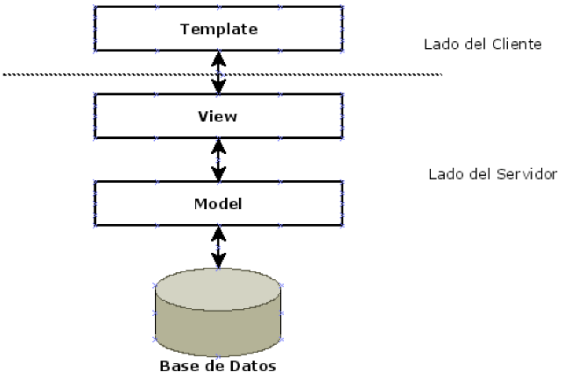
- La **Parte 3** es un framework que maneja la comunicación entre el usuario y la base de datos y automatiza muchas de las partes más difíciles de administrar en un sitio web complejo.

Las partes 1 y 2 son claramente distinguibles en la figura de abajo:

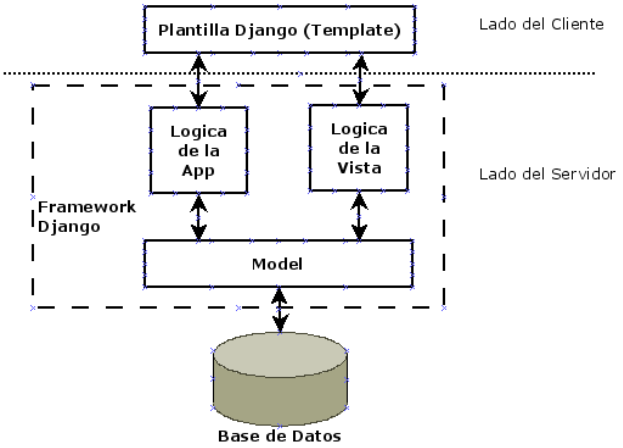


Una forma común de explicar la arquitectura de Django en términos de MVC es describirla como Modelo-Plantilla-Vista (MTV) o Modelo-Vista-Plantilla (MVT). No hay diferencia entre MTV y MVT; son dos formas diferentes de describir lo mismo, lo que aumenta la confusión.

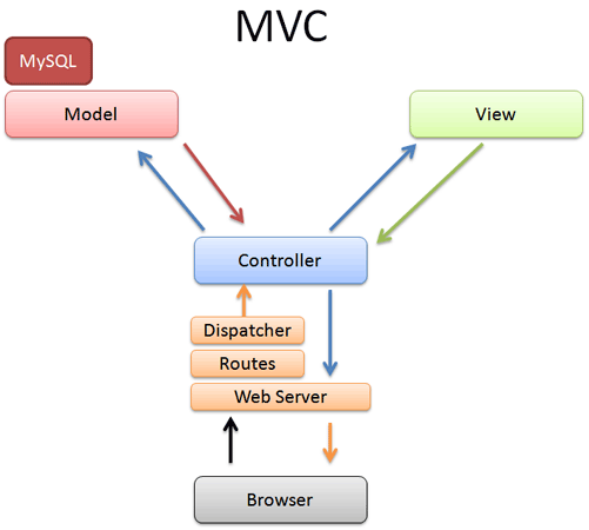
La parte engañosa de este diagrama es la vista (view). La vista en Django se describe con mayor frecuencia como equivalente al controlador en MVC, pero no lo es, sigue siendo la vista. Abajo la figura muestra una vista ligeramente diferente del MTV de Django.



Django usa objetos de solicitud y respuesta para comunicarse entre el cliente y el servidor. Como Django es un framework web, estamos hablando de objetos de solicitud y respuesta HTTP. Entonces, en este proceso simplificado, la vista recupera datos de la base de datos a través del modelo, los formatea, los agrupa en un objeto de respuesta HTTP y los envía al cliente (navegador). En otras palabras, la vista (view) presenta el modelo al cliente como una respuesta HTTP. Esta es también la definición exacta de lo que es una vista en MVC: "La vista significa la presentación del modelo en un formato particular".
Abajo tenemos un esquema que quizás describe mejor la arquitectura de Django:



Este es un esquema MVC más típico, por ejemplo usado por Ruby on Rails (fuente: https://betterexplained.com/articles/intermediate-rails-understanding-models-views-and-controllers/)



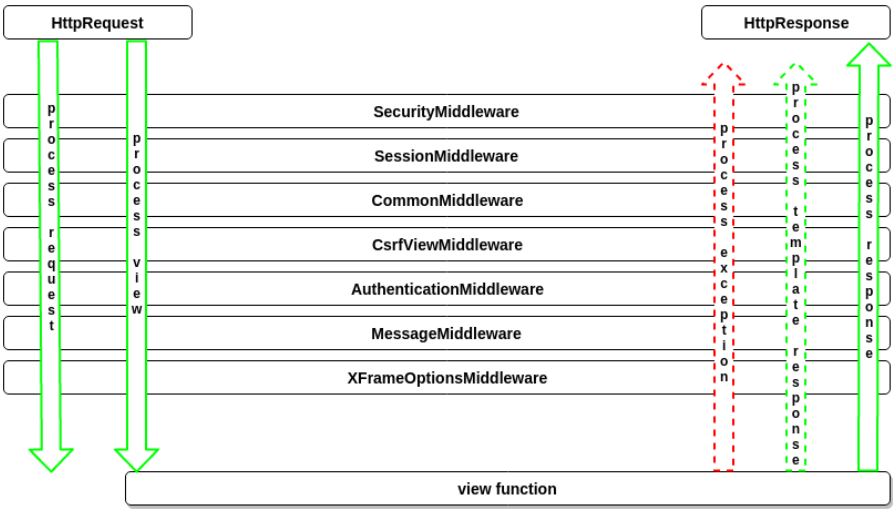
El primer punto de confusión que podemos aclarar en el caso de Django es: dónde poner una función o clase en particular:

¿La función/clase devuelve una respuesta?

- SÍ, es una vista. Ponerlo en el módulo de vistas (**views.py**).

- NO, no es una vista, es la lógica de la aplicación. Ponerlo en otro lugar (en algún **otro_lugar.py**).

El siguiente punto a tener en cuenta es que el framework Django encapsula el modelo, la lógica de vista y la lógica de la app (gestion/business). En algunos tutoriales, se dice que el framework de Django es el controlador, pero eso tampoco es cierto: el framework Django puede hacer mucho más que responder a la entrada del usuario e interactuar con los datos.
Un ejemplo perfecto de este poder adicional es el middleware de Django, que se encuentra entre la vista y el lado del cliente. El middleware de Django realiza comprobaciones críticas de seguridad y autenticación antes de enviar la respuesta al navegador. Fuente: https://medium.com/@adamdonaghy/why-django-middleware-is-so-darn-cool-dd1b79a268ad



El middleware está descrito en la variable **MIDDLEWARE** dentro del archivo **settings.py**

1.6.1.20. El principio DRY y su aplicación en el entorno Python/Django

Hay muchos frameworks que se jactan de ser compatibles con "DRY" (don't repeat yourself - no se repita usted mismo). Al trabajar con muchos de ellos, uno se da cuenta de que ninguno lo hace bien. Lamentablemente, a la mayoría de los frameworks no les importa lo suficiente como para seguir realmente el principio "DRY". En nuestra opinión, si está creando una aplicación que actualizará regularmente (la mayoría de las aplicaciones en la actualidad), debe seguir DRY para evitar problemas.

En el framwork Laravel, por ejemplo, se requiere que escriba validaciones para cada ruta por separado. Este es el caso de la mayoría de los demás frameworks. Para que su código sea compatible con DRY, debe esforzarse. Es difícil mantener un control, especialmente cuando se trabaja en equipo. El marco de Django, por otro lado, está diseñado de tal manera que debe hacer todo lo posible para violar el principio DRY.
Es como debería ser, veamos un ejemplo:

Así es como funcionan las validaciones y las migraciones de bases de datos en Django:

1.- Cree una clase de modelo con los campos obligatorios. Especifique las validaciones y restricciones adicionales según sea necesario.

2.- Las migraciones se generan con un solo comando CLI: **python manage.py makemigrations**.

3.- La base de datos se actualiza con los cambios con un solo comando CLI: **python manage.py migrate.**

4.- Las validaciones y restricciones se verifican automáticamente en cada operación CRUD, ya sea en Django Admin o Django REST Framework. No es necesario que vuelva a escribir validaciones.

5.- La misma clase modelo se utiliza para generar vistas CRUD de Django Admin. No se requiere HTML/CSS personalizado.

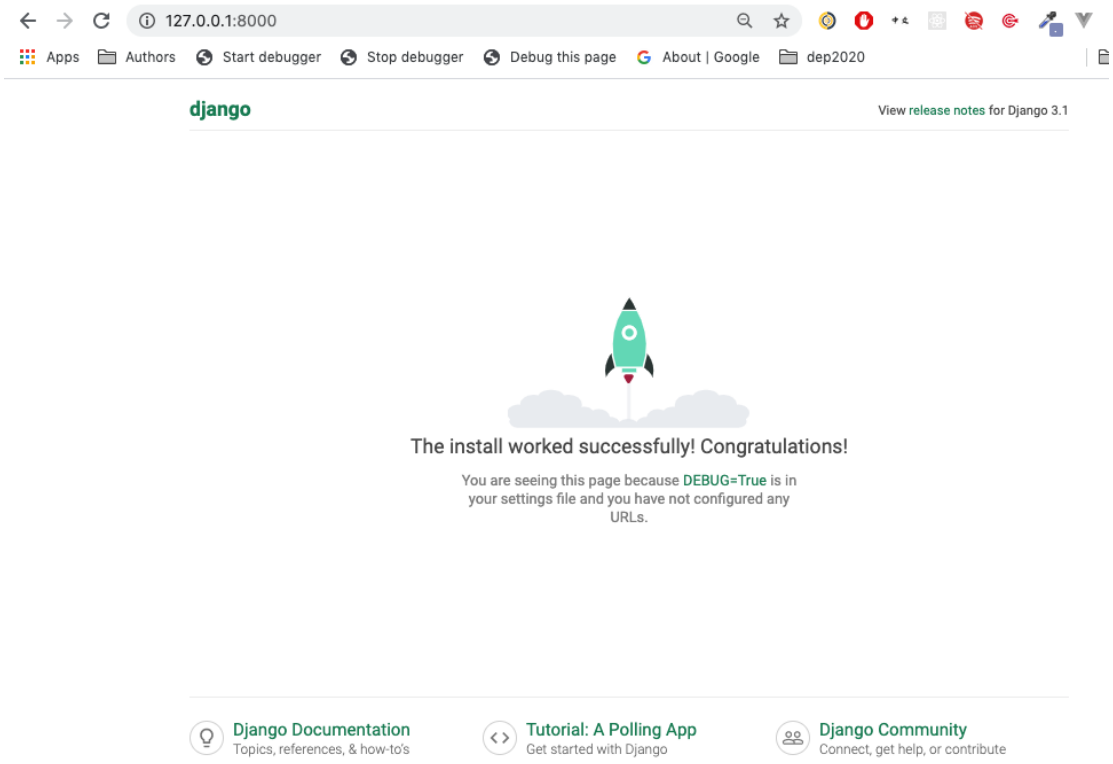
Si compara estas características con cualquier otro framework, es difícil que pueda hacer todo esto con solo las siguientes líneas de código:

```
class Empleado(models.Model):
    nombre = models.CharField(max_length=127)
    email = models.EmailField(null=True, blank=True)
    creado_en = models.DateTimeField(blank=True, null=True, auto_now_add=True)
    actualizado_en = models.DateTimeField(blank=True, null=True, auto_now=True)
```

No se trata solo de "no repetirse". Le ayuda a evitar errores en el futuro. Todos hemos estado en situaciones en las que cambió algo en algún lugar, pero se olvidó de hacerlo en otro lugar y lo descubrimos solo después de que muchos clientes se encontraron con el problema.

En Django, refiriéndose al código anterior, si alguna vez necesita cambiar el **max_length** de un campo a otra cosa, hágalo aquí. Se aplicará automáticamente a la base de datos y la validación de todas las rutas.

1.6.1.21. Qué son los Templates de Django



Tal contenido (template/plantilla) se encuentra dentro del código interno de django **.../views/templates/default_urlconf.html código del framework** que nosotros no debiéramos alterar o modificar bajo ningún punto de vista.

Vamos a modificar nuestra aplicación para que en lugar de mostrar el mensaje de bienvenida Django, muestre algún contenido nuestro.

Como sabemos, lo primero es enrutar, en **sitiodeprueba/sitiodeprueba/urls.py** vamos a agregar nuestra ruta raíz:

```
urlpatterns = [
    path('', views.index, name='index'),
    path'admin/', admin.site.urls),
]
```

Pero si se da cuenta no tenemos **views** definidos en ninguna parte, por lo tanto, lo vamos a crear.

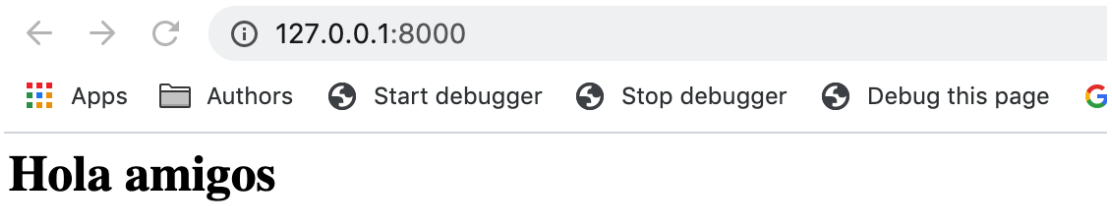
archivo: **sitiodeprueba/sitiodeprueba/views.py**

```
from django.http import HttpResponse


def index(request):
    returnHttpResponse("<h1>Hola amigos</h1>")
```

Ahora necesitamos que **urls.py** sepa que tenemos definido **views**

```
from django.contrib import admin
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index, name='index'),
    path('admin/', admin.site.urls),
]
```

Si se dirige una vez más a http://127.0.0.1:8000/ debería ver:



Hemos sobreescrito la ruta raíz.

1.6.1.22. Renderización en Django

El ejemplo anterior solo nos mostró cómo mostrar una frase ("Hola amigos") pero ¿qué sucede si queremos desplegar páginas más complejas? En ese caso debemos escribir templates.

Creamos un nuevo directorio **templates/sitiodeprueba** de forma que nuestra estructura de carpetas queda así:

```
.
├── db.sqlite3
├── manage.py
└── sitiodeprueba
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── templates
    │   └── sitiodeprueba
    │       └── index.html
    ├── urls.py
    ├── views.py
    └── wsgi.py
```

Note que creamos el directorio **templates/sitiodeprueba** y no solamente una carpeta **templates**, esto porque Django elegirá la primera plantilla que encuentre cuyo nombre coincida, y si tuviera una plantilla con el mismo nombre en una aplicación diferente, Django no podría distinguir entre ellas. Necesitamos decirle a Django cuales la correcta, y la mejor manera de asegurar esto es mediante **namespaces**. Es decir, colocando esas plantillas dentro de otro directorio con el nombre de la propia aplicación.

Usaremos un template de las primeras clases, lo nombraremos 'index.html' dentro de **.../templates/sitiodeprueba**

```
<!DOCTYPE html>
<html lang="es">
<head>
    <title>Landing Page</title>
    <meta charset="utf-8"/>
    <style>
        #imagen {
            float:left;
            width:60%;
            text-align: center;
        }
        #formulario-beneficios {
            float:right
            width:40%; }
        header footer {
            clear:both
            text-align: center;
        } </style>
</head>
<body>
<header>
```

```
   <h1>Nuestro Producto Estrella</h1> <h2>Regi´strate dentro de los 100
primeros
    y recibira´s un descuento</h2>
</header>
<section id="imagen">
    <img src="https://www.cfg.com/images/cfg1a_new.gif"
         height="250px">
    <p>Image copyrights to https://www.cfg.com/</p>
</section>
<section id="formulario-beneficios">
    <form>
        <p>Nombre</p>
        <input type="text">
        <p>Email</p>
        <input type="email"/>
        <br>
        <br>
        <button type="submit">Regi´strate</button>
    </form>
    <br>
    <h3>¡Nuestros Beneficios!</h3>
    <ul>
        <li>Rapidez</li>
        <li>Seguridad</li>
        <li>Gran Precio</li>
        <li>Flexibilidad</li>
    </ul>
</section>
<footer>
    <br>
    <p>
         <a href="">Like Red Social 1</a> |
         <a href="">Like Red Social 2</a> |
         <a href="">Like Red Social 3</a> |
         <a href="">Like Red Social 4</a>
    </p>
    <br>
    <p>Copyright &copy; 2019-2020 -
       Desarrollos de Landing Pages Inc.</p>
</footer>
</body>
</html>
```

Necesitamos especificarle a Django donde buscar las templates, eso lo hacemos en el archivo **sitiodeprueba/sitiodeprueba/settings.py**, donde agregaremos las siguientes líneas al comienzo, y donde corresponde a la variable **TEMPLATES.**

```
import os
PROJECT_APP_PATH = os.path.dirname(os.path.abspath(__file__))
...
TEMPLATES = [
    {
```

```
        ...
        'DIRS': [os.path.join(PROJECT_APP_PATH,'templates')],
        ...
        }
]
```

La variable **PROJECT_APP_PATH** contiene la ruta hacia **.../sitiodeprueba/sitiodeprueba**; tenemos duplicado el nombre lo cual no es raro en muchos proyectos Django que hemos visto, pero el primer nombre de la carpeta más externa no tiene por qué llamarse de la misma forma que la interna.

Arreglemos nuestra estructura de carpeta para que quede de esta manera:

```
.
└── sitio
    ├── db.sqlite3
    ├── manage.py
    └── sitiodeprueba
        ├── __init__.py
        ├── asgi.py
        ├── settings.py
        ├── templates
        │   └── sitiodeprueba
        │       └── index.html
        ├── urls.py
        ├── views.py
        └── wsgi.py
```

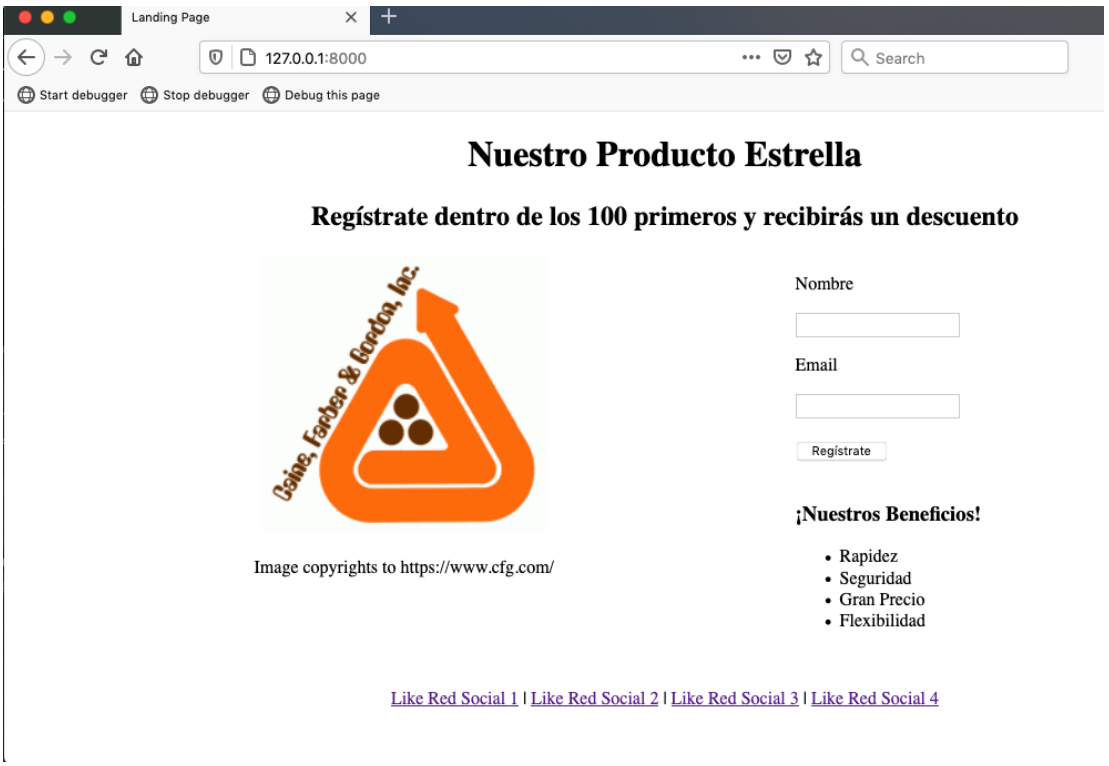
Por claridad renombramos la carpeta externa **sitio**

```
mv sitiodeprueba sitio
```

Para ejecutar el servidor local ahora debemos estar dentro de **sitio**

```
.../sitio$ python manage.py runserver
```

Si todo ha salido bien deberíamos ver nuestra antigua template en http://127.0.0.1:8000/



6.1.2.- Referencias

[1] Django Documentation
https://docs.djangoproject.com/en/3.1/

[2] D. Feldroy & A. Feldroy , Two Scoops of Django 3.x, Best Practices for the Django Web Frame-work, 2020.

[3] William S. Vincent, Django for Beginners Build websites with Python & Django, 2020.

[4] Nigel George, Build a Website with Django 3, 2019.

[5] Django Tutorial
https://www.geeksforgeeks.org/django-tutorial/