# MVC

Modelo–vista–controlador (MVC) es un patrón de arquitectura de software, que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones.

* El Modelo: Es la representación de la información con la cual el sistema opera, por lo tanto gestiona todos los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (lógica de negocio).
* El Controlador: Responde a eventos (frecuentemente acciones del usuario) e invoca peticiones al ‘modelo’ cuando se hace alguna solicitud sobre la información (por ejemplo, editar un documento o un registro en una base de datos). También puede enviar comandos a su ‘vista’ asociada si se solicita un cambio en la forma en que se presenta el ‘modelo’, por tanto se podría decir que el ‘controlador’ hace de intermediario entre la ‘vista’ y el ‘modelo’.
* La Vista: Presenta el ‘modelo’ (información y lógica de negocio) en un formato adecuado para interactuar (usualmente la interfaz de usuario) por tanto requiere de dicho ‘modelo’ la información que debe representar como salida.

# WSGI

WSGI (Web Server Gateway Interface) es una especificación de una interface simple y universal entre los servidores web y las aplicaciones web o frameworks desarrolladas con python.

Mientras que en PHP el código se ejecuta al acceder al fichero que lo contiene y genera el HTML, en Python, cada petición a un servidor web que contiene una aplicación en Python la recibe el fichero WSGI. Las rutas de las URI's no corresponden con los ficheros físicos del servidor, si no que la wsgi application hace de enrouter.

El WSGI, que será un único fichero, es la aplicación responsable de manejar las peticiones, y de devolver la respuesta adecuada según la URI solicitada. En esta aplicación tendremos que definir una función, que actúe con cada petición del usuario. Esta función, deberá ser una función WSGI aplicación válida. Esto significa que:

* Deberá llamarse *application*
* Deberá recibir dos parámetros:
  + *environ*, del módulo *os*, que provee un diccionario de las peticiones HTTP estándar y otras variables de entorno que se envían al servidor.
  + la función *start\_response*, encargada de entregar la respuesta HTTP al client.

El diccionario *environ* que se recibe con cada pedido HTTP, contiene las variables estándar de la especificación CGI, entre ellas:

* REQUEST\_METHOD: método “GET”, “POST”, …
* SCRIPT\_NAME: parte inicial de la “ruta”, que corresponde a la aplicación
* PATH\_INFO: 2ª parte de la “ruta”, determina la “ubicación” virtual en la aplicación.
* QUERY\_STRING: la porción de la URL que sigue al “?”, si existe
* CONTENT\_TYPE, CONTENT\_LENGTH de la petición HTTP
* SERVER\_NAME, SERVER\_PORT, combinadas con SCRIPT\_NAME y PATH\_INFO dan la URL
* SERVER\_PROTOCOL: la versión del protocolo (“HTTP/1.0” or “HTTP/1.1”)

# Flask

<https://wiki.python.org/moin/WebFrameworks> https://github.com/OpenWebinarsNet/curso\_flask

Este framework resulta ideal para construir servicios web (como APIs REST) o aplicaciones de contenido estático. Flask es Open Source y está amparado bajo una licencia BSD.

<http://flask.pocoo.org/mailinglist/>  
<https://github.com/OpenWebinarsNet/curso_flask/tree/master/ejemplos/u4>

Flask no tiene ORMs, wrappers o configuraciones complejas, eso lo convierte en un candidato ideal para aplicaciones ágiles o que no necesiten manejar ninguna dependencia. Si necesitas trabajar con base de datos sólo tenemos que utilizar una extensión.

Flask soporta la ejecución una batería de pruebas unitarias. Flask es compatible con el estándar wsgi. Flask nos proporciona un objeto que en realidad es una aplicación wsgi, por ello será muy sencillo en un servidor web que tenga servidor de aplicaciones wsgi. Flask soporta de forma íntegra el uso de cookies y sesiones.

Puedes ver el código en Github, la documentación es muy completa y te puedes suscribir a su lista de correos para mantenerte al día de las actualizaciones. Flask es totalmente compatible con el patrón MVC. Aunque por defecto no tiene un ORM, podemos usar una extensión de Flask para definir el modelo de datos. Esta característica nos abstrae del uso del motor de Base de Datos y lo hace independiente. Con Flask vamos a definir un controlador, que es capaz de determinar las rutas con las que accedemos a la aplicación, procesar la información necesaria y mostrar la información necesaria en cada momento.

Flask utiliza Jinja2 como motor de plantillas, con lo que es muy fácil diseñar las vistas que vamos a mostrar a los usuarios en cada momento.

De la lista de extensiones que nos permite ampliar la funcionalidad de Flask, vamos a utilizar:

* Flask-Script: La extensión flask-script nos proporciona la posibilidad de gestionar nuestra aplicación flask desde una comando (Interfaz de línea de comando).
* Flask-Bootstrap: si queremos trabajar con plantillas que utilicen como hoja de estilos y javascript el framework bootstrap podemos ulizar la extansión Flask-Bootstrap.
* Flask-WTF: Flask-WTF es una extensión de Flask que nos permite trabajar con la librería WTForm de python, que nos facilita la generación y validación de formularios HTML.
* Flask-Sqlalchemy: Usaremos la extensión Flask-SQLAlchemy que nos provee un wrapper para el proyecto SQLAlchemy, el cual es un Object Relational Mapper o ORM.
* Flask-Login: Flask-Login es una librería que nos proporciona la posibilidad de gestionar las sesiones de nuestros usuarios.

<https://medium.com/@madumalt/flask-app-deployment-in-windows-apache-server-mod-wsgi-82e1cfeeb2ed>

<https://uwsgi-docs.readthedocs.io/en/latest/WSGIquickstart.html>

<https://www.fullstackpython.com>

# Instalación en Debian

Vamos a realizar la instalación de Flask utilizando la herramienta pip en un entorno virtual creado con virtualenv. La instalación de Flask depende de dos paquetes: [**Werkzeug**](http://werkzeug.pocoo.org/), una librería WSGI para Python y [**jinja2**](http://jinja.pocoo.org/docs/2.9/) como motor de plantillas. El entorno virtual es un espacio dentro del usuario de la máquina que permite realizar instalaciones de paquetes Pyrhon sin que entre en conflicto con los paquetes Python previamente instalados en el sistema.

Como Flask es compatible con python3 vamos a crear un entorno virtual compatible con la versión 3 del interprete python. Para ello nos aseguremos que tenemos la utilidad instalada:

# apt-get install python-virtualenv

Y creamos el entorno virtual:  
$ virtualenv -p /usr/bin/python3 flask

Para activar nuestro entorno virtual:  
$ source flask/bin/actívate  
(flask)$

Y a continuación instalamos Flask:  
(flask)$ pip install Flask

Si nos aparece el siguiente aviso durante la instalación: WARNING: The C extensión could not be compiled, speedups are not enabled. Failure information, if any, is above. Retrying the build without the C extensión now.

La instalación se realiza bien, pero no se habilita el aumento de rendimiento de jinja2. Puedes volver a realizar la instalación después de instalar el siguiente paquete:  
# apt-get install python3-dev

Al finalizar podemos comprobar los paquetes Python instalados:

* (flask)$ pip freeze
* Flask==versión
* Jinja2==versión
* MarkupSafe==version
* Werkzeug==versión
* click==versión
* itsdangerous==version

Podemos guardar las dependencias en un fichero requirements.txt:  
# pip freeze > requirements.txt La utilización del fichero ˋrequirements.txtˋ, donde vamos a ir guardando los paquetes python (y sus versiones) de nuestra instalación, nos va a posibilitar posteriormente poder crear otro entrono virtual con los mismos paquetes:  
 # pip install -r requirements.txt

Y finalmente comprobamos la versión de flask que tenemos instalada:  
(flask)$ flask –version

Para desactivar el entorno virtual ejecutamos el comando 🡪 deactivate

Para activar el entorno virtual ejecutamos el comando 🡪 source ~/virtualenv/flask/bin/activate

# Instalación en Windows

Un entorno virtual dentro del usuario de la máquina nos permite instalar paquetes Python sin que entren en conflicto con los paquetes instalados en el sistema. También facilita que al desplegar al entorno de Producción, la aplicación mantenga las mismas dependencias que en entorno de Desarrollo. Si durante la instalación aparece un warning de que la extensión de C no está compilada, para evitar problemas de rendimiento instalamos las librerías de desarrollo de Python 🡪 **pip uninstall flask** 🡪 **pip install python-dev** 🡪 **pip install flask**

Una vez instalado Flask se recomienda comprobar los paquetes instalados mediante el comando **pip freeze** y dado que estamos trabajando con entornos virtuales, podemos guardar la salida de dicho comando en un fichero 🡪 **pip freeze > requirements.txt**

Para después llevarnos dicho fichero a otro servidor 🡪 **pip install –r requirements.txt**

Acceder al directorio que vamos a utilizar como entorno virtual: **cd entornovirtual**

* Instalar pip en ese directorio: **pip install pipenv**
* **pipen**
* Arrancar la consola del entorno virtual: **pipenv shell**
* Instalamos Flask en el entorno virtual: **pip install Flask**
* Para ver lo que acaba de instalar: **pip freeze**
* Podemos guardar las dependencias en un fichero requirements.txt:  
  **pip freeze > requirements.txt**

La utilización del fichero ˋrequirements.txtˋ, donde vamos a ir guardando los paquetes python (y sus versiones) de nuestra instalación, nos va a posibilitar posteriormente poder crear otro entrono virtual con los mismos paquetes: **pip install -r requirements.txt**

Para entrar y salir del entorno virtual: **exit() 🡪 pipenv shell.** Para ver los comandos posibles y su significado: **pip**

<http://timmyreilly.azurewebsites.net/python-flask-windows-development-environment-setup/> <http://flask.pocoo.org/>

<https://www.stuartellis.name/articles/python-development-windows/>

<https://docs.python-guide.org/starting/install/win/>

<https://www.stuartellis.name/articles/python-getting-started/>

<https://docs.python-guide.org/dev/virtualenvs/>

<https://www.stuartellis.name/articles/python-learning-resources/>

# Aplicación

<http://flask.pocoo.org/docs/dev/tutorial/factory/>

Basicamente en lo que respecta al corrido de la aplicacion, export FLASK\_APP, export FLASK\_ENV, export FLASK\_DEBUG. Existen tres formas para ejecutar las aplicaciones flask:

1. Utilizando variables de entorno.
   * FLASK\_APP: Donde se indica el nombre del fichero que se va a ejecutar
   * FLASK\_DEBUG: Para indicar el modo de depuración
   * FLASK\_ENV: Para indicar el modo de desarrollo

En este caso no es necesario que el código de la aplicación tenga la función **run()** en el programa principal:

$ export FLASK\_APP=my\_application

$ export FLASK\_ENV=development

flask run

1. Utilizando la función **run()**en el código:

**from flask import Flask** # Constructor  
**app = Flask(\_\_name\_\_)** # Aplicación WSGI

**@app.route('/')** # Decorador **def hello\_world():**

**return 'Hello Matrix!'**

**if \_\_name\_\_ == '\_\_main\_\_':**

**app.run('0.0.0.0', 8080, debug=True)**

1. Utilizando el plugin Flask-Script

Dentro de nuestro entorno virtual, tendremos un fichero vacío ***\_\_init\_\_.py*** que indica que la aplicación es un módulo de Pyhon. El app.run() se ejecuta con ***el manage.py***

**pip install flask-script  
pip freeze>requirements.txt**

* Creamos el fichero manage.py
* Arrancamos el entorno virtual 🡪 **pipenv shell**
* Ejecutamos el archivo 🡪
  + **py manage.py runserver**
  + **py manage.py shell**
* La segunda opción nos da una línea de intérprete Python donde la aplicación wsgi app.py ya está importada 🡪 **app.config**

De este modo, en lugar de utilizar la instrucción **run()** tenemos a nuestra disposición el comando para ejecutar la aplicación en nuestro servidor de desarrollo. Posteriormente iremos incrementando funcionalidades al manager de nuestra app.

Si queremos ejecutar el servidor de aplicaciones en alguna IP/puerto específico utilizamos la opción -h / -p 🡪 **py manage.py runserver –h 0.0.0.0 –p 5000**

La estructura de la aplicación web será:

* manage.py 🡪 script que gestiona la aplicación (flask-script)
* requirements.txt 🡪 fichero con los módulos de la aplicación (dependencias)
* Aplicación 🡪 Paquete Python con el módulo principal (app.py) y otros
  + app.py
  + \_\_init\_\_.py

**from flask\_script import Manager** # Objeto Manager  
**from aplicacion.app import app**# Aplicacion WSGI del fichero fuente módulo aplicación

**manager = Manager (app)**# Objeto creado pasándole al constructor la aplicación WSGI

**@app.config['DEBUG'] = True** # Ensure debugger will load

**if \_\_name\_\_ == '\_\_main\_\_':**

**manager.run()**

# CONTROLADOR: enrutamiento, peticiones y respuestas

Al utilizar el manage.py cada vez que se realiza un cambio en el código se recarga el servidor.

El objeto Flask **app** nos proporciona un decorador **router** que es capaz de filtrar la función que se va ejecutar analizando la petición HTTP, fundamentalmente por la ruta y el método que se hace la petición. Se comprueba la ruta de la petición HTTP, y cuando coincide con alguna indicada en las rutas, se ejecuta la función correspondiente devolviendo una respuesta HTTP.

Si declaramos rutas terminando en / son consideradas como un directorio de un sistema de fichero, en este caso si se accede a la ruta sin la barra final se producirá una redirección a la ruta correcta. Si declaramos las rutas sin / al final, se consideran como un fichero el sistema y al acceder con / nos devuelve una respuesta Not Found 404.

## Rutas Dinámicas

Son rutas variables que corresponden a un determinado patrón o expresión regular: La parte dinámica de la ruta la podemos obtener como variable que recibe la función correspondiente. Además, varias rutas pueden ejecutar una misma función. Aunque no es obligatorio podemos especificar el tipo de la variable capturada como forma de verificación, sin embargo Python la casteará a una cadena de caracteres ya que está concatenada a la ruta url:

* string: Acepta cualquier texto sin barras (por defecto)
* int: Acepta enteros
* float: Acepta números reales
* path: Acepta cadena de caracteres con barras

## Construcción de rutas

Podemos importar la función url\_for que nos permite construir rutas a partir del nombre de la función asociada. Lo cual es muy útil a la hora de crear vistas si queremos incorporar enlaces que vayan a una ruta determinada. La función url\_for nos permite escribir el nombre de una función declarada en el archivo manage.py y nos devuelve la ruta de dicha función:

1. **py manage.py shell**
2. **>>> from flask import url\_for**
3. **print(url\_for("articulos"))**
4. **print(url\_for(“hola”,nombre=”Pepe”, edad=20))**

De esta manera no tenemos que poner a mano las rutas en una plantilla html, ya que sí en un momento dado se deciden cambiar esas rutas las plantillas se actualizarán automáticamente.

1. **py manage.py runserver**
2. abrimos otra consola
3. **curl** [**http://localhost:5000/login**](http://localhost:5000/login)
4. **curl -X POST** [**http://localhost:5000/login**](http://localhost:5000/login)

## El objeto request

En muchas ocasiones es deseable acceder a una URL con los dos métodos, de tal manera que haremos una cosa cuando acedemos con GET y haremos otra cuando se acceda con POST. Por defecto las rutas indicadas en la funciones route sólo son accesibles utilizando  
el método GET. El objeto request es un tipo de dato inmutable multi-dict y se diferencia de un diccionario en que determinados atributos del objeto pueden guardar más de un valor.

**curl** [**http://localhost:5000/**](http://localhost:5000/login)**info?id=10**

**curl -X POST** [**http://localhost:5000/**](http://localhost:5000/login)**info -d id=55**

## Respuestas Http

El decorador router gestiona la petición HTTP recibida y crea un objeto response con la respuesta HTTP: el código de estado, las cabeceras y los datos devueltos. Esta respuesta la prepara a partir de lo que devuelve la función *vista* ejecutada con cada route. Estas funciones pueden devolver tres tipos de datos:

* Una cadena, o la generación de una plantilla (que veremos posteriormente). Por defecto se indica un código 200 y las cabeceras por defecto.
* Un objeto de la clase response generado con la función **make\_repsonse()**, que recibe los datos devueltos, el código de estado y las cabeceras.
* Una tupla con los mismos datos: datos, cabeceras y código de respuesta.

Siempre que queramos devolver una respuesta con un código determinado, podemos utilizar en cualquier punto del código la función **abort()**. Esta función termina con el flujo de ejecución del código. Si queremos personalizar una respuesta para un código de error determinado, utilizaríamos el decorador **errorhandler()** acompañando una función que recibe un parámetro error.

## Contenido estático

Ampliamos la estructura de nuestra aplicación web:

* manage.py 🡪 script que gestiona la aplicación (flask-script)
* requirements.txt 🡪 fichero con los módulos de la aplicación (dependencias)
* aplicación 🡪 Paquete Python con el módulo principal (app.py) y otros
  + app.py
  + \_\_init\_\_.py
  + static 🡪 Directorio donde guardar el contenido estático
    - css
    - js
    - media

Para hacer referencia desde nuestra aplicación web a las rutas que corresponden a los ficheros estáticos de nuestro sistema de archivos utilizamos la función **url\_for()** con:

* primer parámetro el nombre del directorio del contenido estático 🡪 static
* segundo parámetro el fichero

# VISTA: plantillas y formularios

<http://jinja.pocoo.org/docs/2.10/>

jinja2 es independiente de Flask, luego podemos utilizarlo en cualquier proyecto de Python que necesitemos generar algún fichero cuya estructura se repite frecuentemente.

1. Entramos en el intérprete de Python 🡪 desde el cmd de Windows accedemos a la ruta del entorno virtual y ejecutamos **py**
2. **from jinja2 import Template**
3. Almacenamos la plantilla en una variable 🡪 **temp1=’Hola {{nombre}}’**
4. Generamos la plantilla 🡪 **print(Template(temp1).render(nombre=”Pepe”))**

Utilizamos el método render() de la clase Template 🡪 **temp2=’<a href=”{{ url }}”>{{enlace}}</a>’**

**print( Template(temp2).render(url=”http://www.flask.com”, enlace=”Flask”))**

A las plantillas, además de variables también podemos enviarles listas y diccionarios python.

**temp3=’<a href=”{{ datos[0] }}”>{{ datos[1] }}</a>’  
print(Template(temp3).render(datos=[http://www.flask.com”,”Flask”))**

**temp4=’<a href=”{{ datos.url }}”>{{ datos.enlace }}</a>’  
print(Template(temp4).render(datos={“url”:”http://www.flask.com”, “enlace”:”Flask”}))**

Los filtros son funciones que se ejecutan para modificar el dato enviado a la plantilla:

* striptags 🡪 hace un trim al principio y fina de la variable
* title 🡪 Muestra el primer carácter en Mayúscula

**temp5=’Hola {{nombre|striptags|title}}’  
print( Template(temp5).render(nombre=” pepe ”) )**

* separa los elementos de la lista concatenados con el carácter ,

**temp6=”los datos son {{ lista|join(‘, ’) }}”  
print( Template(temp6).render(lista=[“amarillo”, “Verde”, “rojo”]) )**

* cuenta el número de caracteres del último elemento de la lista

**temp7=”El ultimo elemnto tiene {{ lista|last|length }}”  
print( Template(temp7).render(lista=[“amarillo”, “verde”, “rojo”]) )**

Aquellos caracteres que poseen un significado determinado en HTML habrá que escaparlos:

**temp8=”La siguiente cadena muestra todos los caracteres: {{ info|e }}”  
print( Template(temp8).render(info=”<hola&que&tal>”) )**

Para generar una plantilla utilizamos la función render\_template() que recibe como parámetro el fichero html y las variables que se pasan a esta. Las platntillas las vamos a guardar en ficheros html en la carpeta Templates, dentro del directorio aplicación.

La herencia de plantillas utiliza una plantilla base de la cual heredan todas las demás, indicando los bloques que las plantillas hijas pueden sobreescribir.

## Bootstrap myCSS & myJS

Podemos utilizar la hoja de estilo que queramos, la podemos guardar en nuestro contenido estático o acceder por medio de una url. Sin embargo si queremos trabajar con plantillas que utilicen como hoja de estilos y javascript el framework bootstrap podemos utilizar la extensión Flask-Bootstrap. <https://pythonhosted.org/Flask-Bootstrap/>

1. Con nuestro entrono virtual activado 🡪 **pip install Flask-Bootstrap4==4.0.2**
2. pip freeze 🡪 **pip freeze > requirements.txt**

La plantilla base de bootstrap ofrece los siguientes bloques que podemos sobrescribir:

* **html** 🡪 Contiene el contenido completo de la etiqueta <html>
* **html\_attribs** 🡪 Atributos para la etiqueta <html>
* **head** 🡪 Contiene el contenido completo de la etiqueta <head>
* **body** 🡪 Contiene el contenido completo de la etiqueta <body>
* **body\_attribs** 🡪 Atributos para la etiqueta <body>
* **title** 🡪 Contiene el contenido completo de la etiqueta <title>
* **styles** 🡪 Contiene todos los estilos CSS de la etiqueta <link>
* **metas** 🡪 Contiene los <meta> de la cabecera
* **navbar** 🡪 Un bloque vacío encima del contenido
* **scripts** 🡪 Contiene todos los scripts en la etiqueta <script> al final del body

**from flask\_bootstrap import Bootstrap  
app = Flask(\_\_name\_\_)  
Bootstrap(app)**

El Framework de Flask genera una plantilla base que trabaja con Bootstrap, de manera que a partir de ahora, todas las plantillas tendrán la hoja de estilos del Framework de Bootstrap.

Si queremos añadir otro fichero de hoja de estilos:  
{% block styles %}  
{{super()}}  
<link rel="stylesheet" href="{{url\_for('.static', filename='mystyle.css')}}">  
{% endblock %}

Si queremos añadir otro fichero JavaScript:  
{% block scripts %}  
<script src="{{url\_for('.static', filename='myscripts.js')}}"></script>  
{{super()}}  
{% endblock %}

Añadir el idioma español en los atributos de <html>:  
{% block html\_attribs %} lang="es"{% endblock %}

La estructura de la aplicación web será:

* manage.py 🡪 **from aplicacion.app\_bootstrap import app**
* requirements.txt 🡪 Bootstrap
* Aplicación 🡪 Paquete Python con el módulo principal (app.py) y otros
  + app.py
  + \_\_init\_\_.py
  + Static
  + Templates 🡪 tempboot1234.html

## Formularios

Las plantillas de nuestro programa van a heredar de base\_form.html que a su vez hereda de bootstrap/base.html. Creamos una plantilla con el bloque que nos permite poner una barra de menú navegación navbar. Después, en el block content creamos un nuevo block dynamic. El resto de nuestras plantillas heredarán de esta plantilla base y en ellas reescribiremos el block dynamic.<https://www.howtogeek.com/261591/how-to-create-and-run-bash-shell-scripts-on-windows-10/> <http://flask.pocoo.org/docs/1.0/blueprints/>

* manage.py 🡪 **from aplicacion.app\_form import app**

## Flask-WTF

Esta extensión permite trabajar con la librería WTForm de Python tratando los formularios como una clase <https://flask-wtf.readthedocs.io/en/stable/>   
<https://flask-wtf.readthedocs.io/en/stable/> https://wtforms.readthedocs.io/en/latest/  
<https://flask-wtf.readthedocs.io/en/stable/quickstart.html#creating-forms>

Dado que WTForm es una librería independiente de Flask, utilizamos la extensión para instalarla por dependencia 🡪 pip install Flask-WTF De manera que por un lado se instala la extensión Flask-WTF y por otro lado la librería WTForms 🡪 pip freeze

Por defecto Flask-WTF protege los formularios contra el ataque CSRF ([Cross-Site Request Forgery](https://es.wikipedia.org/wiki/Cross-site_request_forgery)). Al crear el formulario podemos inicializarlo con datos, normalmente con parámetros recibidos en la vista (request.form): El objeto FormCalculadora(request.form) nos ofrece algunos atributos y métodos para su gestión:

* form.validate\_on\_submit(): Comprueba y valida el envío del formulario.
* form.data: Nos ofrece un diccionario con los datos del formulario y sus valores.
* form.errors: Si el formulario no es válido, devuelve un diccionario con los errores.
* form.num1.data: Para cada campo (en este ejemplo num1) nos devuelve su valor.
* form.num1.errors: Tupla con los errores de validación del campo determinado.
* form.num1(): Nos devuelve el código HTML para generar este campo.
* form.num1.label(): Devuelve el código HTML para general la etiqueta del campo.

Para implementar la protección CSRF, Flask-WTF necesita que configuremos una clave de cifrado, para generar tokens encriptados que se utilizarán para verificar la autenticidad de la petición.🡪 **app.secret\_key = 'clave de cifrado lo más robusta posible'**

Cada vez que generemos un formulario se incluirá un campo oculto que contendrá el token cifrado para verificar si el envío del formulario ha sido lícito. Para generar este campo oculto utilizamos el método **form.csrf\_token()** que mostrará un código HTML parecido a este:  
<input id="csrf\_token" name="csrf\_token" type="hidden" value="I3bM...">

**from aplicacion.formularios import FormCalculadora  
app = Flask(\_\_name\_\_)  
app.secret\_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'**

Abrimos el intérprete de Python desde el contexto de la aplicación 🡪 py manage.py Shell  
>>> from aplicacion.formularios import FormCalculadora  
>>> form=FormCalculadora() 🡪 >>> form.data 🡪 >>> form.errors

## Subida de Ficheros

Utilizamos el tipo de campo FileField para crear nuestro formulario para subir un fichero. Cuando se manda un fichero al servidor la información del fichero la encontramos en request.file. <http://flask.pocoo.org/docs/0.12/patterns/fileuploads/>

Cuando creamos un formulario con el constructor UploadForm(), no lo inicializa en blanco sino que automáticamente inicializa el formulario con los datos mandados en POST, y por lo tanto, ya no es necesario utilizar el request.forms que recoge los datos del método POST, ni tampoco será necesario recuperar el objeto request.file.

* manage.py 🡪 **from aplicacion.app\_file import app**

# Modelo: Gestionando los datos

Python ofrece diversos módulos para conectarnos a distintos motores de base de datos:

* Mysql: [MySql-Python](https://pypi.python.org/pypi/MySQL-python), [PyMySQL](https://pypi.python.org/pypi/PyMySQL), [Mysql Conector](https://dev.mysql.com/downloads/connector/python/)
* [postgreSQL](https://wiki.postgresql.org/wiki/Python)
* [sqlite3](https://docs.python.org/2/library/sqlite3.html)

Usaremos la extensión [Flask-SQLAlchemy](http://flask-sqlalchemy.pocoo.org/2.1/) para manejar el modelo de datos de nuestra aplicación. Esta extensión provee un wrapper para el proyecto [SQLAlchemy](https://www.sqlalchemy.org/), el cual es un [Object Relational Mapper](https://es.wikipedia.org/wiki/Mapeo_objeto-relacional) o ORM. Los ORM permiten a las aplicaciones con base de datos trabajar con objetos en vez de tablas y SQL. Las operaciones realizadas en los objetos son traducidos a comandos de base de datos transparente para el ORM. Por lo tanto se abstrae el gestor de base de datos utilizado, si cambiamos de SGBD no tendremos que cambiar nuestra aplicación. De este modo, por ejemplo, podemos estar trabajando con sqlite3 en nuestro entorno de desarrollo, y con una base de datos PostgreSQL en nuestro entorno de producción.

## Tienda Virtual

Los datos que guardamos en nuestra base de datos serán representados por una colección de clases que son referidas como modelos de base de datos. La capa ORM hará las traducciones necesarias y convertirá los objetos creados a partir de estas clases a filas en la base de datos.



* Categorías: Tabla para guardar las categorías de los artículos
* Artículos: Datos del artículo. Cada artículo corresponde a una categoría
* Usuarios: Datos del usuario. Un usuario puede tener perfil de administrador
* Carrito: Podríamos guardarlo en una tabla, pero en nuestro ejemplo vamos a usar cookies para gestionar los artículos que va comprando un usuario
* Se puede buscar artículos por categorías
* Los clientes se pueden dar de alta en nuestra tienda virtual
* El usuario administrador puede dar de alta, modificar o eliminar artículos de la tienda
* Cuando un usuario compra un artículo se guarda en un “carro de la compra”. El carro de la compra no lo vamos a guardar en una tabla de la base de datos, vamos a utilizar cookies para guardar dicha información
* Los usuario pueden acceder a la tienda para realizar las compras, vamos a utilizar sesiones para simular la persistencia de la conexión

La extensión Flask-sqlalchemy nos posibilita usar el módulo sqlalchemy en nuestra aplicación FLask. Para instalarla, en nuestro entono virtual activo: pip install Flask-Sqlalchemy

<https://docs.sqlalchemy.org/en/13/core/type_basics.html>  
<https://docs.sqlalchemy.org/en/13/core/constraints.html>

Cuando trabajábamos con formularios pusimos una variable de configuración con la cadena de caracteres para generar los token de seguridad. En este caso también tendremos que realizar una serie de configuraciones, como por ejemplo, indicar a Sqlalchemy qué gestor de base de datos vamos a utilizar. A fin de no tener que ir estableciendo variables de entorno en cada módulo creamos el fichero config.py

**import os**

**SECRET\_KEY = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT’  
PWD = os.path.abspath(os.curdir)**

**DEBUG = TRUE  
SQLALCHEMY\_DATABASE\_URI = 'sqlite:///{}/dbase.db’.format(PWD)  
SQLALCHEMY\_TRACK\_MODIFICATIONS = FALSE**

Cuando se crea un modelo, si hemos importado explícitamente from sqlalchemy import Boolean, Column, ForeingKey podemos utilizar los tipos directamente. Si no ponemos esa línea y nos quedamos solo con la importación from aplicacion import db tendremos que atacarlos con db.Column

Probamos nuestro código en la consola: Importamos el objeto db que representa la base de datos y posteriormente los modelos que vamos a utilizar. Desde el directorio del entorno virtual:

py

>>> from aplicacion.app import db

>>> from aplicacion.model import Categorias,Articulos

>>> db.create\_all()

Automáticamente se genera la base de datos en el directorio. Está instrucción no actualiza la estructura de la base de datos si cambiamos el modelo, por lo tanto en esa circunstancia tenemos que borrar las tablas y crearlas de nuevo:

db.drop\_all()

db.create\_all()

**SQLite Manager** es una extension/complemento de Firefox para poder ver la db

Podemos utilizar también la migración de base de datos que me permite, al cambiar el modelo actualizar la estructura de la base de datos. Para trabajar con migraciones podemos usar la extensión [Flask-Migrate](https://flask-migrate.readthedocs.io/en/latest/)

A continuación añadimos una categoría:

>>> cat=Categorias(nombre="Arcade")

>>> db.session.add(cat)

>>> db.session.commit()

Y dos artículos de esa categoría (la FK se puede escribir a pelo o utilizando la relación):

>>> art1=Articulos(nombre="PAC-MAN", precio=12, descripcion="juego de fantasmitas", stock=1, CategoriaId=1)

>>> art2=Articulos(nombre="Super Mario Bros", precio=25, descripcion="juego de platoformas", stock=10, categoria=cat)

>>> db.session.add\_all([art1,art2])  
 >>> db.session.commit()

Podemos cambiar el valor de un campo y volver a añadirlo:

>>> art1.precio=15

>>> db.session.add(art1)

>>> db.session.commit()

Borrado de registros:  
 >>> db.session.delete(art2)

>>> db.session.commit()

Podemos realizar [diferentes operaciones](http://docs.sqlalchemy.org/en/latest/orm/query.html) para obtener un conjunto de registros. Por ejemplo podemos obtener el primer registro: >>> row1=Articulos.query.first()  
O el registro cuyo identificador sea el 2: >>> row2=Articulos.query.get(2)  
O todos los registros: >>> articulos=Articulos.query.all()

**for art in Articulos.query.all():  
 print(art.nombre)**

Obtener el número los registros: >>> Articulos.query.count()

Filtrado de registros por un campo y por dos campos:

Articulos.query.filter\_by(precio=25).all()

Articulos.query.filter\_by(precio=25).filter\_by(iva=21).all()

Ordenar por un campo:

Articulos.query.order\_by(“precio”).all()

Articulos.query.filter\_by(precio=25).filter\_by(iva=21).all()

Trabajando con las relaciones podemos obtener los datos de la categoría a partir de un artículo o los artículos a partir de una categoría: **art1.categoria.nombre**

**for art in cat.articulos:  
 print(art.nombre)**

La estructura de nuestra aplicación es la siguiente:

* manage.py 🡪 script que gestiona la aplicación (flask-script)
* requirements.txt 🡪 fichero con los módulos de la aplicación (dependencias)
* **static**
  + **css**
    - myStyleSheet.css
* **templates**
  + parent.html
  + notFound.html
  + index.html
* **aplicacion** 🡪 Paquete Python con el módulo principal (app.py) y otros
  + \_\_init\_\_.py
  + app.py
  + config.py
  + forms.py
  + model.py

En cuanto al ORM SQLAlchemy podemos destacar:

* En la variable \_\_tablename\_\_ indicamos el nombre de la tabla a la que corresponde esta clase.
* Vamos indicando los distintos campos del modelo utilizando el constructor db.Column e indicando el tipo de datos que van a guardar. Podemos indicar los [siguientes tipos de datos](http://docs.sqlalchemy.org/en/latest/core/type_basics.html).
* Además del tipo de datos podemos indicar [los atributos de cada campo](http://docs.sqlalchemy.org/en/latest/core/constraints.html) (primary\_key, unique, ForeignKey,…)
* Hemos indicado una relación con el constructor relationship, esto nos permite relacionar objetos de una clase (registros de una tabla) con los objetos de otra clase que están relacionados. En nuestro caso es una relación uno a uno entre un artículo y su categoría. (En el modelo de categoría puedes ver una relación 1 a N, una categoría tiene varios artículos).

Activamos el entorno virtual 🡪 pipenv shell

py manage.py drop\_tables

py manage.py create\_tables  
py manage.py add\_data\_tables  
py manage.py runserver

Activamos el entorno virtual 🡪 pipenv shell