Introducción a Flask

- Flask es un microframework para Python basado en Werkzeug que permite crear aplicaciones web de todo tipo rápidamente.
- Dash está basado en flask.
- Flask es muy versátil, puedes hacer prototipos y APIs muy rápido.
- Para la realización de APIs veremos otra librería llamada FASTAPI.

Preparando el entorno de programación

- Creamos un directorio “tutorial-flask”. Una vez dentro, inicializaremos nuestro entorno Python con:

```
python -m venv env
```

- Activamos el venv:
  - En linux con:

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

  - En windows:

```
env\Scripts\activate.bat
```

- Podemos también configurarlo en vscode.
- Usaremos este venv para toda la clase.

- Sabremos que el entorno está activo porque el prompt comienza con la palabra **(env)**
- Para instalar Flask (versión 1.x) escribiremos en el terminal el siguiente comando:

```
pip install Flask
```

- Podemos ver todas las dependencias de nuestra aplicación si ejecutamos el siguiente comando:

```
pip freeze
```

Creando la primera aplicación Flask

- Nuestra primera app será un helloword.
- Crearemos un fichero llamado run.py.
- Añadimos el siguiente código:

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
    return 'Hello, World!'
```

```python
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
    return 'Hello, World!'
```

- Toda aplicación Flask es una instancia WSGI de la clase Flask.
- Importamos dicha clase y creamos una instancia, que en este caso he llamado app.
- Para crear dicha instancia, debemos pasar como primer argumento el nombre del módulo o paquete de la aplicación. Para estar seguros de ello, utilizaremos la palabra reservada **name**
- Esto es necesario para que Flask sepa donde encontrar las plantillas de nuestra aplicación o los ficheros estáticos.

```python
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
    return 'Hello, World!'
```

- Tendremos métodos asociados a las distintas URLs que componen nuestra aplicación.
- Flask se encarga de que partir de una petición a una URL se ejecute finalmente nuestra rutina. Lo único que tendremos que hacer nosotros es añadir un decorador a nuestra función.
- La función hello_world será invocada cada vez que se haga una petición a la URL raíz de nuestra aplicación.

```python
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
    return 'Hello, World!'
```

- El decorador route de la aplicación (app) es el encargado de decirle a Flask, qué URL debe ejecutar su correspondiente función.
- El nombre que le demos a nuestra función será usado para generar internamente URLs a partir de dicha función.
- La función debe devolver la respuesta, que será mostrada en el navegador del usuario.

Probando la aplicación

- Flask viene con un servidor interno que nos facilita mucho la fase de desarrollo.
- No debemos usar este servidor en un entorno de producción ya que no es este su objetivo.
- Debemos indicarle al servidor qué aplicación debe lanzar declarando la variable de entorno `FLASK_APP`

- Declaramos la varaible FLASK_APP de la siguiente manera:
  - En Linux/Mac o con git bash:

```
export FLASK_APP="run.py"
```

  - En Windows:

```
set "FLASK_APP=run.py"
```

- Una vez que hemos definido dónde puede el servidor de Flask encontrar nuestra aplicación, lo lanzamos ejecutando:

```
flask run
```

- Esto lanzará el servidor de pruebas:

```
* Serving Flask app "run.py"
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

- Para comprobar que nuestra aplicación funciona, podemos entrar al navegador y en la barra de direcciones introducir:

```
localhost:5000
```

- Esto nos mostrará el mensaje «Hello world!» de nuestra función hello_world().
- Por defecto, el servidor escucha en el puerto 5000 y solo acepta peticiones de nuestro propio ordenador.
- Si queremos cambiar el puerto por cualquier motivo lo podemos hacer de dos formas distintas:
  - Estableciendo la variable de entorno `FLASK_RUN_PORT` en un puerto diferente.
  - Indicando el puerto al lanzar el servidor flask run `--port 6000`.
- Para aceptar peticiones de otros ordenadores de nuestra red local lanzaremos el servidor de la siguiente manera:

```
flask run --host 0.0.0.0
```

Modo debug

- El modo debug que es muy útil usar mientras estamos desarrollando.
- Cada vez que hagamos un cambio en nuestro código reiniciará el servidor y no tendremos que hacerlo manualmente para que los cambios se tengan en cuenta.
- Para activar el modo debug simplemente hay que añadir la variable de entorno FLASK_ENV y asignarle el valor development:
  - En Linux/Mac:

```
export FLASK_ENV="development"
```

  - En Windows:

```
set "FLASK_ENV=development"
```

- El modo debug hace que:
  - Se active un depurador.
  - Se tengan en cuenta los cambios después de guardarlos sin tener que reiniciar manualmente el servidor.
  - Activa el modo debug, de manera que si se produce una excepción o error en la aplicación veremos una traza de los mismos.

- Flask asocia una URL con un método de nuestro código.
- Tenemos que añadir el decorador route() a la función que queramos ejecutar cuando se hace una petición a una determinada URL.
- En Flask, por convención, a las funciones que están asociadas a una URL se les llama «vistas».

Nuestra página principal.

- La URL de esta página será "/" y en ella se mostrará el listado de posts de nuestro blog.
- Los posts se almacenarán en una lista en memoria.

- modifica el fichero run.py borra la función hello_world() y añade:

```
posts = []
@app.route("/")
def index():
    return "{} posts".format(len(posts))
```

- En una aplicación web no todas las páginas tienen una URL definida de antemano, como es el caso de la página principal.
- Cada post tendrá una URL única que se generará dinámicamente.
- Sin embargo, no vamos a tener una vista por cada URL que se genere.
- Vamos a tener una única vista que se encargará de recoger un parámetro que identifique al post que queremos mostrar y, en base a dicho parámetro, recuperar el post para finalmente mostrarlo al usuario.

- Para hacer esto, a una URL le podemos añadir secciones variables o parametrizadas con <param>. La vista recibirá <param> como un parámetro con ese mismo nombre.
- Opcionalmente se puede indicar un conversor (converter) para especificar el tipo de dicho parámetro así converter:param.
- Por defecto, en Flask existen los siguientes conversores:
    - string: Es el conversor por defecto. Acepta cualquier cadena que no contenga el carácter ‘/’.
    - int: Acepta números enteros positivos.
    - float: Acepta números en punto flotante positivos.
    - path: Es como string pero acepta cadenas con el carácter ‘/’.
    - uuid: Acepta cadenas con formato UUID.

- Vamos a crear una vista para mostrar un post a partir del slug del título del mismo.
- Un slug es una cadena de caracteres alfanuméricos (más el carácter ‘-‘), sin espacios, tildes ni signos de puntuación.

```
@app.route("/p/<string:slug>/")
def show_post(slug):
    return "Mostrando el post {}".format(slug)
```

- Al definir una URL acabada con el carácter ‘/’, si el usuario accede a esa URL sin dicho carácter, Flask lo redirigirá a la URL acabada en ‘/’.
- En cambio, si la URL se define sin acabar en ‘/’ y el usuario accede indicando la ‘/’ al final, Flask dará un error HTTP 404.

Renderizando una página HTML

- Flask trae por defecto un motor de renderizado de plantillas llamado Jinja2 que te ayudará a crear las páginas dinámicas de tu aplicación web.
- Para renderizar una plantilla creada con Jinja2 simplemente hay que hacer uso del método render_template().
- A este método debemos pasarle el nombre de nuestra plantilla y las variables necesarias para su renderizado como parámetros clave-valor.

- Flask buscará las plantillas en el directorio templates de nuestro proyecto.
- En el sistema de ficheros, este directorio se debe encontrar en el mismo nivel en el que hayamos definido nuestra aplicación.
- En nuestro caso, la aplicación se encuentra en el fichero run.py.
- Crear este directorio y añadir las páginas index.html, post_view.html y admin/post_form.html.

- Ahora modifiquemos el cuerpo de las vistas index(), show_post() y post_form(), para que muestren el resultado de renderizar las respectivas plantillas.
- Importar el método render_template() del módulo flask:

```
from flask import render_template:

@app.route("/")
def index():
    return render_template("index.html", num_posts=len(posts))
@app.route("/p/<string:slug>/")
def show_post(slug):
    return render_template("post_view.html", slug_title=slug)
@app.route("/admin/post/")
@app.route("/admin/post/<int:post_id>/")
def post_form(post_id=None):
    return render_template("admin/post_form.html", post_id=post_id)
```

index.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Tutorial Flask: Miniblog</title>
</head>
<body>
    {{ num_posts }} posts
</body>
</html>
```

post_view.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ slug_title }}</title>
</head>
<body>
    Mostrando el post {{ slug_title }}
</body>
</html>
```

post_form.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>
        {% if post_id %}
            Modificando el post {{ post_id }}
        {% else %}
            Nuevo post
        {% endif %}
    </title>
</head>
<body>
{% if post_id %}
    Modificando el post {{ post_id }}
{% else %}
    Nuevo post
{% endif %}
</body>
</html>
```

- El aspecto de estas páginas es similar a una página html estática con la excepción de {{ num_posts }} y {{ slug_title }} y los caracteres {% y %}.
- Dentro de las llaves se usan los parámetros que se pasaron al método render_template().
- El resultado de ello es que durante el renderizado se sustituirán las llaves por el valor de los parámetros. De este modo podemos generar contenido dinámico en nuestras páginas.

Para crear una plantilla tenemos que tener en cuenta:

- Cualquier expresión contenida entre llaves dobles se mostrará como salida al renderizarse la página.
- Es posible usar estructuras de control, como sentencias if o bucles for, entre los caracteres {% y %}.

Ficheros estáticos

- Además de las plantillas que generan contenido dinámico, una página web también se compone de ficheros CSS para definir estilos, imágenes y código javascript.
- Todo este tipo de ficheros se conocen como recursos estáticos, ya que su contenido no cambia a lo largo del ciclo de ejecución de una aplicación web.
- Para que nuestro código CSS o Javascript sea servido, debemos ubicar estos ficheros en un directorio llamado static situado al mismo nivel que el directorio templates. Este directorio estará accesible en la URL /static .

- Crear un fichero llamado base.css dentro del directorio static en el que definiremos el estilo de nuestro blog:

```
body {
    margin: 0;
    padding: 0;
    font-size: 100%;
    line-height: 1.5
}
h1, h2, h3, h4 {
    margin: 1em 0 .5em;
    line-height: 1.25
}
h1 {
    font-size: 2em
}
h2 {
    font-size: 1.5em
}
h3 {
    font-size: 1.2em
}
ul, ol {
    margin: 1em 0;
    padding-left: 40px
}
p, figure {
    margin: 1em 0
}
a img {
    border: none
}
sup, sub {
    line-height: 0
}
```

Plantillas

- Como has podido comprobar, las páginas que por el momento forman nuestro blog index, post_view y post_form comparten la misma estructura.
- Podemos evitar repetir código si creamos una plantilla base que pueda ser reutilizada.
- Para crear plantillas y bloques de código reutilizables, Jinja2 pone a nuestra disposición la etiqueta {% block %}{% endblock %}

- Dentro del directorio templates crea un fichero llamado base_template.html y pega el siguiente código:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}{% endblock %}</title>
    <link rel="stylesheet" href="{{ url_for("static", filename="base.css") }}">
</head>
<body>
{% block content %}{% endblock %}
</body>
</html>
```

- En ella hay definidos dos bloques: {% block title %}{% endblock %} para el título y {% block content %}{% endblock %} como un contenedor para el contenido en sí de la página.
- Además, hemos enlazado el fichero con nuestros estilos base.css haciendo uso de la función url_for().
- Ahora podemos modificar el resto de plantillas para que hagan uso de ella:

index.html

```
{% extends "base_template.html" %}
{% block title %}Tutorial Flask: Miniblog{% endblock %}
{% block content %}
    {{ num_posts }} posts
{% endblock %}
```

post_view.html

```
{% extends "base_template.html" %}
{% block title %}{{ slug_title }}{% endblock %}
{% block content %}
    Mostrando el post {{ slug_title }}
{% endblock %}
```

post_form.html

```
{% extends "base_template.html" %}
{% block title %}
    {% if post_id %}
        Modificando el post {{ post_id }}
    {% else %}
        Nuevo post
    {% endif %}
{% endblock %}
{% block content %}
    {% if post_id %}
        Modificando el post {{ post_id }}
    {% else %}
        Nuevo post
    {% endif %}
{% endblock %}
```

Formularios

- Los formularios son una parte muy importante de cualquier aplicación.
- El protocolo HTTP define una serie de métodos de petición, conocidos como «verbos HTTP», para indicar la acción que se desea realizar sobre un recurso determinado.
- Cada uno de estos verbos tiene una semántica diferente, siendo los más comunes los métodos GET y POST:
  - GET debe emplearse para solicitar un recurso, por lo tanto, las peticiones realizadas con este método solo deben recuperar datos (y no modificarlos).
  - POST se utiliza para enviar una entidad a un recurso específico, de manera que se envían datos al servidor donde son procesados. Es por ello que, por regla general, siempre que necesitemos enviar datos a un servidor para ser manipulados lo haremos a través del método POST.

- Creamos una nueva plantilla en el directorio templates.
- Llamaremos a esta plantilla signup_form.html y su contenido será el siguiente:

```
{% extends "base_template.html" %}
{% block title %}Registro de usuarios{% endblock %}
{% block content %}
    <form action="" method="post">
        <label for="name">Nombre: </label>
        <input type="text" id="name" name="name" /><br>
        <label for="email">Email: </label>
        <input type="text" id="email" name="email" /><br>
        <label for="password">Contraseña: </label>
        <input type="password" id="password" name="password" /><br>
        <input type="submit" id="send-signup" name="signup" value="Registrar" />
    </form>
{% endblock %}
```

- La plantilla hereda de la plantilla base de nuestro proyecto.
- Para crear un formulario en HTML se usa la etiqueta form.
- En el atributo action se indica la URL a la que se enviarán los datos del formulario.
- Si este campo se deja vacío la URL será la misma desde la que se descargó el recurso.
- En el atributo method indicamos el método a usar al enviar el formulario. En este caso POST, como se mencionó en el apartado anterior.
- Para procesar posteriormente los campos del formulario, se debe indicar el nombre con el que los identificará el servidor a través del atributo name.

- Para mostrar el formulario y procesarlo añadiremos una vista al final del fichero run.py que estará asociada a la URL /signup/:

```
@app.route("/signup/")
def show_signup_form():
    return render_template("signup_form.html")
```

- Si accedemos a la dirección http://localhost:5000/signup/, veremos el formulario en el navegador:
- Si intentamos enviar el formulario pulsando sobre el botón Registrar, nos encontraremos con un error.

- Cualquier vista solo responde ante peticiones GET. Si queremos responder ante otro tipo de peticiones, debemos indicarlo en el parámetro methods del decorador route:

```
@app.route("/signup/", methods=["GET", "POST"])
def show_signup_form():
    return render_template("signup_form.html")
```

- Ahora nuestra vista acepta peticiones POST.
- Por el momento, en cualquiera de los dos casos siempre se mostrará el formulario de registro.

Accediendo a los datos de una petición en Flask

- El objeto request contiene toda la información que el cliente (por ejemplo el navegador web) envía al servidor.
- Entre esta información se encuentran las cabeceras HTTP, los parámetros de la URL, la codificación preferida y, cómo no, los datos que se envían a través de un formulario.

- Si tenemosuna URL con parámetros como /?page=1&list=10.
- Podemos acceder al atributo args del objeto request para obtener estos parámetros:

```
from flask import request
@app.route("/")
def index():
    page = request.args.get('page', 1)
    list = request.args.get('list', 20)
    ...
```

- Los campos enviados a través de un formulario se acceden por medio del atributo form del objeto request.
- Vamos a modificar la vista show_signup_form() para que dependiendo de si es GET o POST realice una acción o otra:

```
from flask import render_template, request, redirect, url_for
@app.route("/signup/", methods=["GET", "POST"])
def show_signup_form():
    if request.method == 'POST':
        name = request.form['name']
        email = request.form['email']
        password = request.form['password']
        next = request.args.get('next', None)
        if next:
            return redirect(next)
        return redirect(url_for('index'))
    return render_template("signup_form.html")
```

- Si se ha enviado el formulario, se recupera cada uno de los campos del mismo por medio del diccionario form del objeto request.
- Recuerda que el nombre de los campos es aquel que indicamos con el atributo name en la plantilla del formulario de registro.
- Luego comprobamos si se pasó por la URL el parámetro next. Este parámetro lo usaremos para redirigir al usuario a la página que se indica en el mismo. Si no se especifica, simplemente lo redirigimos a la página de inicio.
- En caso de que no se haya enviado el formulario, se devuelve como respuesta la página que muestra el formulario de registro.