

Servidores web y backend

Introducción al Internet de las Cosas



Marco Teran
Universidad Sergio Arboleda

2023

Contenido

1 Introducción a backend

- Framework de desarrollo backend

2 Introducción a Flask

3 Flask aplicado

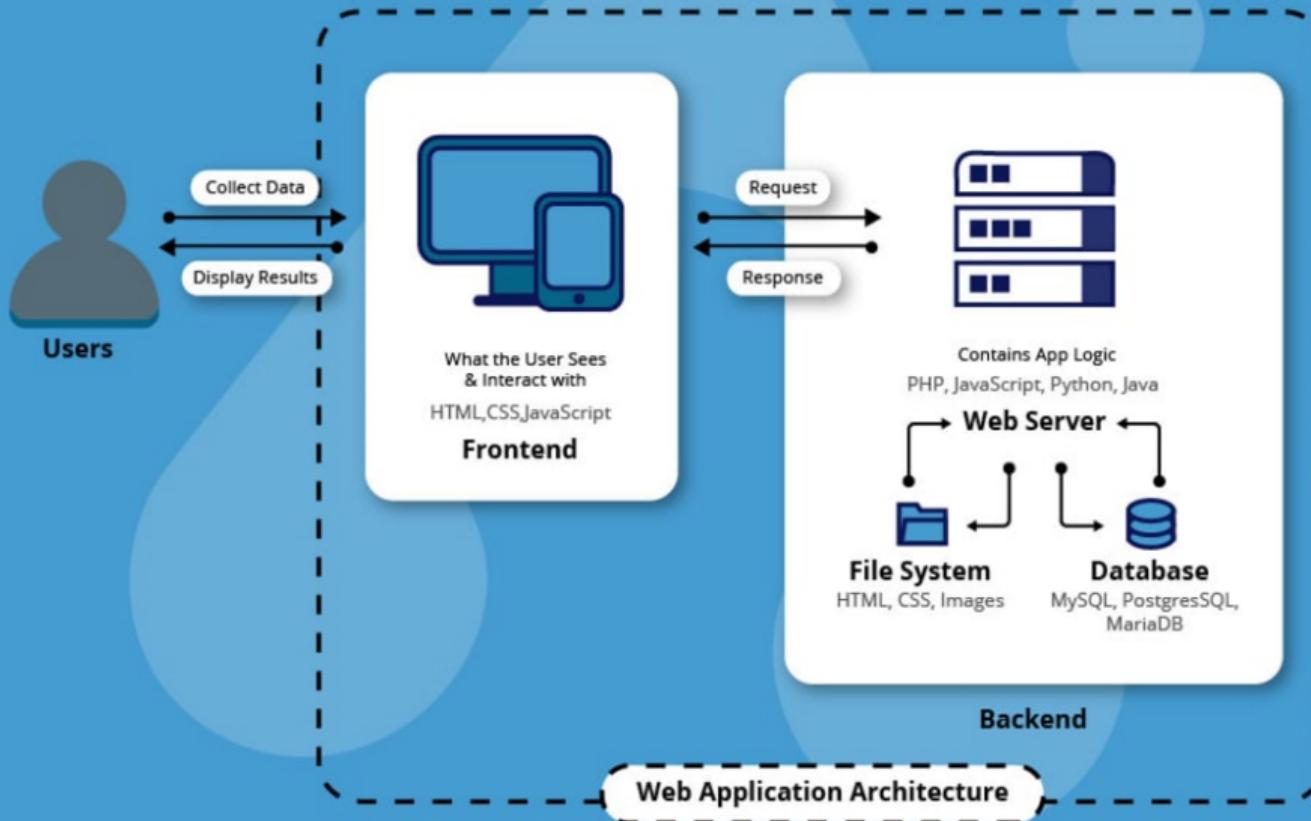
- El hardware
- Controlando los GPIO
- Controlando los GPIO: La plantilla
- Integración de sensores y actuadores
- Más allá con las plantillas

Introducción a backend



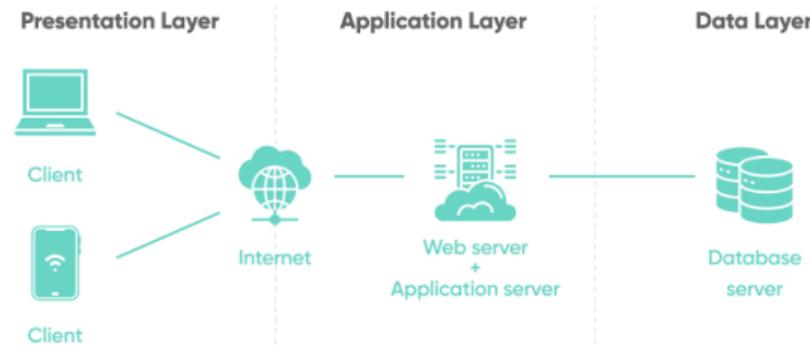
WHAT IS BACK-END DEVELOPMENT?

Web Application Architecture



Introducción a backend

- El **backend** se refiere a la parte del desarrollo web que se encarga del procesamiento y almacenamiento de datos en el servidor.
- Es responsable de la lógica de negocio, la comunicación con bases de datos y otros servicios, y la generación de respuestas para enviar al cliente.
- En el contexto de **Internet de las Cosas**, el backend juega un papel fundamental al recibir y procesar los datos provenientes de los dispositivos conectados.



Backend

- El backend se construye utilizando diferentes tecnologías y frameworks.
- Proporciona herramientas y funcionalidades para manejar solicitudes HTTP, realizar operaciones en la base de datos y generar respuestas dinámicas.
- Con Flask, es posible implementar la lógica de negocio de una aplicación web de manera eficiente y escalable.



Data science analysis



Business logic development



Secure data storage



Web and mobile app
backend development

BACKEND DEVELOPER RESPONSIBILITIES



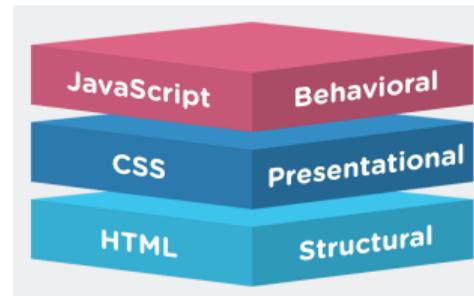
System architecture
building



Payment process
configuration

Frontend

- El frontend se refiere a la parte del desarrollo web que se encarga de la interfaz de usuario y la interacción con el usuario en el navegador.
- Utiliza tecnologías como HTML, CSS y JavaScript para crear y diseñar la apariencia visual de una aplicación web.
- El frontend se comunica con el backend a través de solicitudes HTTP para obtener o enviar datos.
- En el contexto de Internet de las Cosas, el frontend puede mostrar información relevante de los dispositivos conectados y permitir al usuario interactuar con ellos.



FRONT END TOP SKILLS

HTML

CSS

JS



AngularJS
Bootstrap

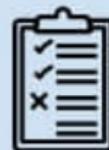
Backbone



Frameworks

Ember

ReactJS



Testing /
Debugging



What's the Difference?



HTML

Hypertext Markup Language

Create the structure

- Controls the layout of the content
- Provides structure for the web page design
- The fundamental building block of any web page



CSS

Cascading Style Sheet

Stylize the website

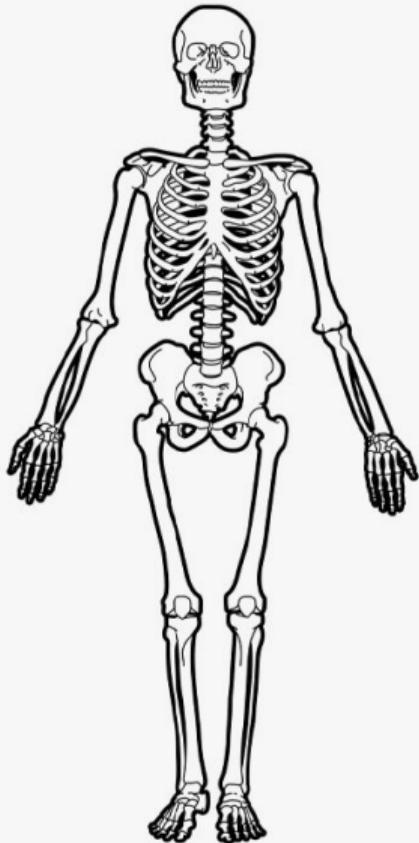
- Applies style to the web page elements
- Targets various screen sizes to make web pages responsive
- Primarily handles the "look and feel" of a web page



Javascript

Increase interactivity

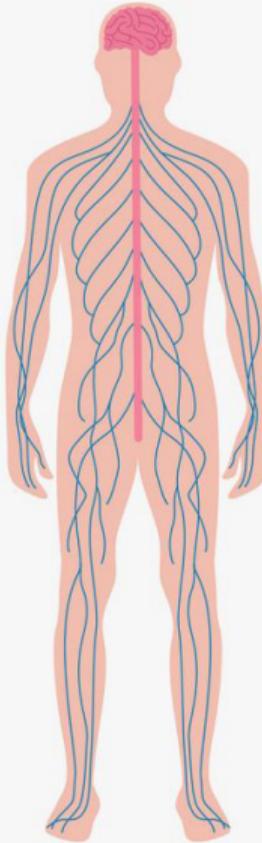
- Adds interactivity to a web page
- Handles complex functions and features
- Programmatic code which enhances functionality



HTML



CSS

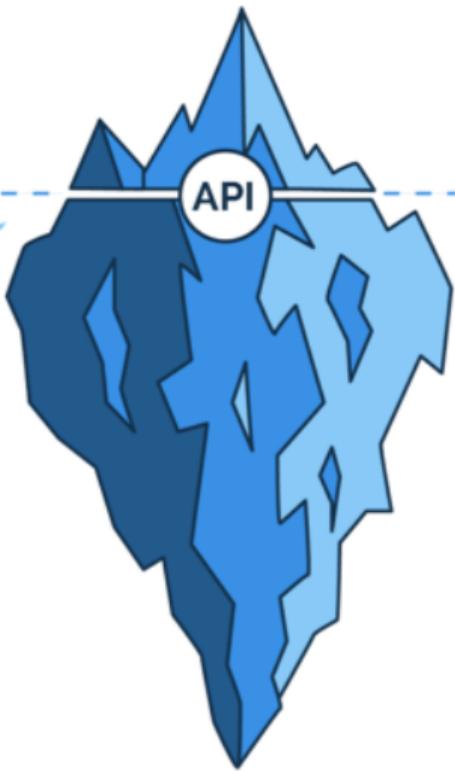


JavaScript



Backend

- 👤 Users don't see
- ⚙️ 80% of total effort
- 📋 Repetitive



Frontend

- 👤 Users see
- 📊 20% of total effort



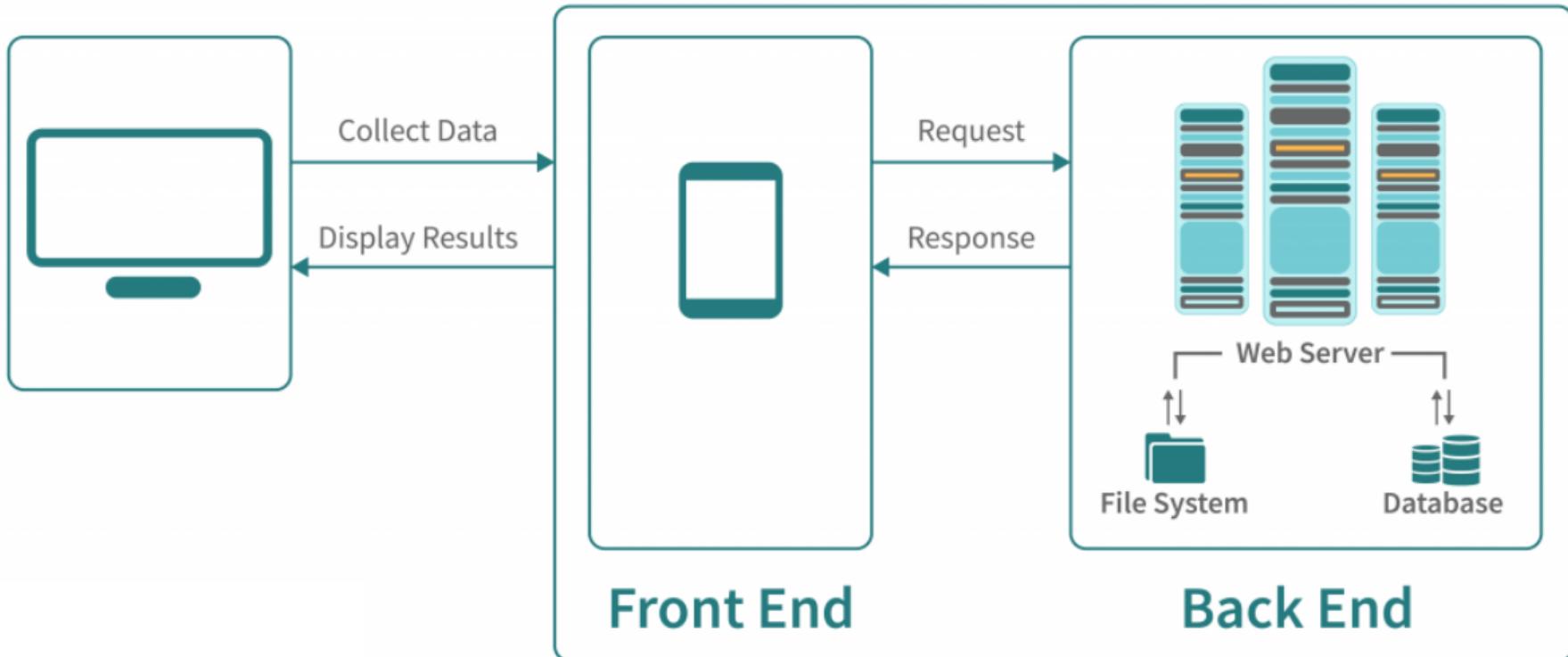


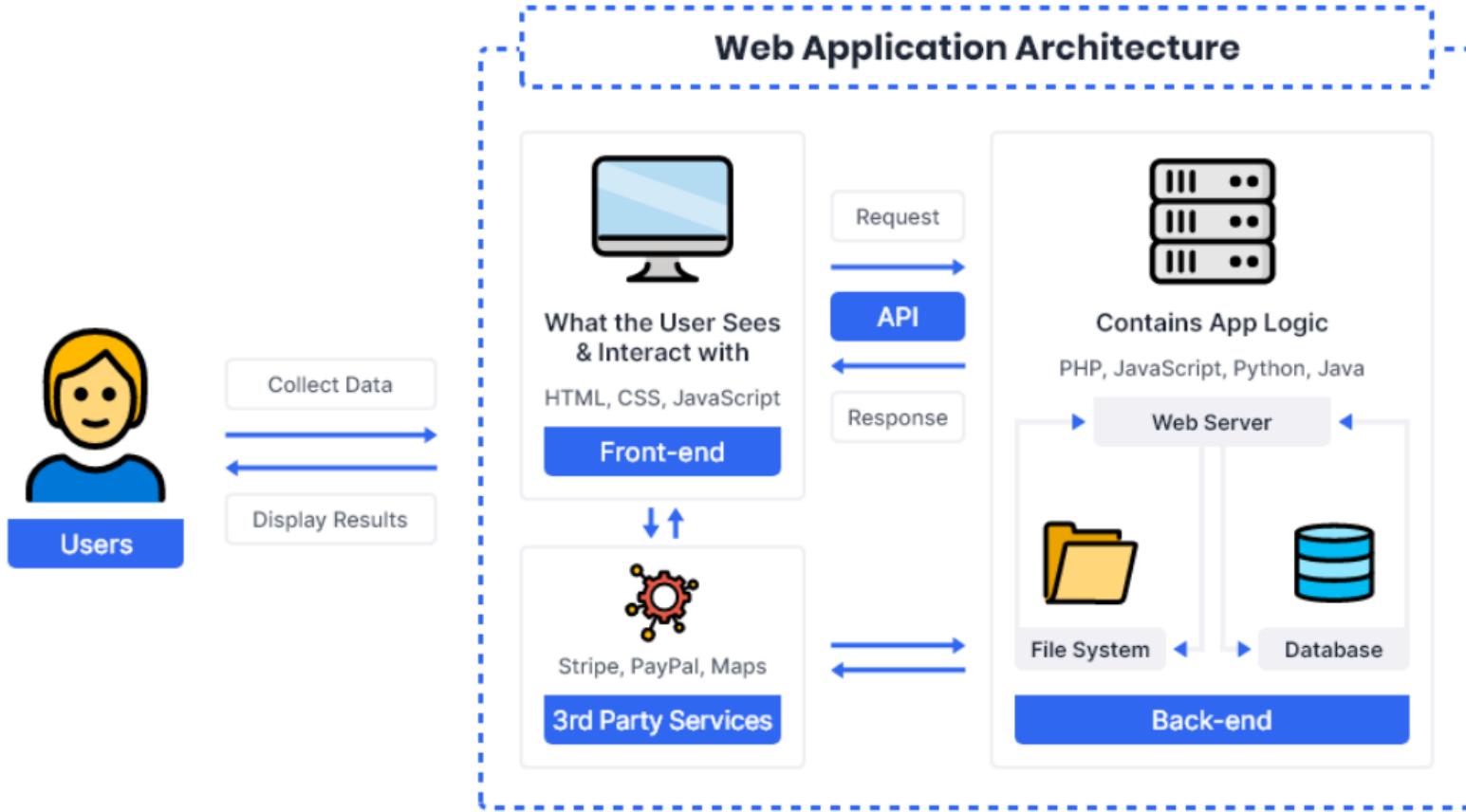
Front End

- Markup and web languages such as HTML, CSS and Javascript
- Asynchronous requests and Ajax
- Specialized web editing software
- Image editing
- Accessibility
- Cross-browser issues
- Search engine optimisation

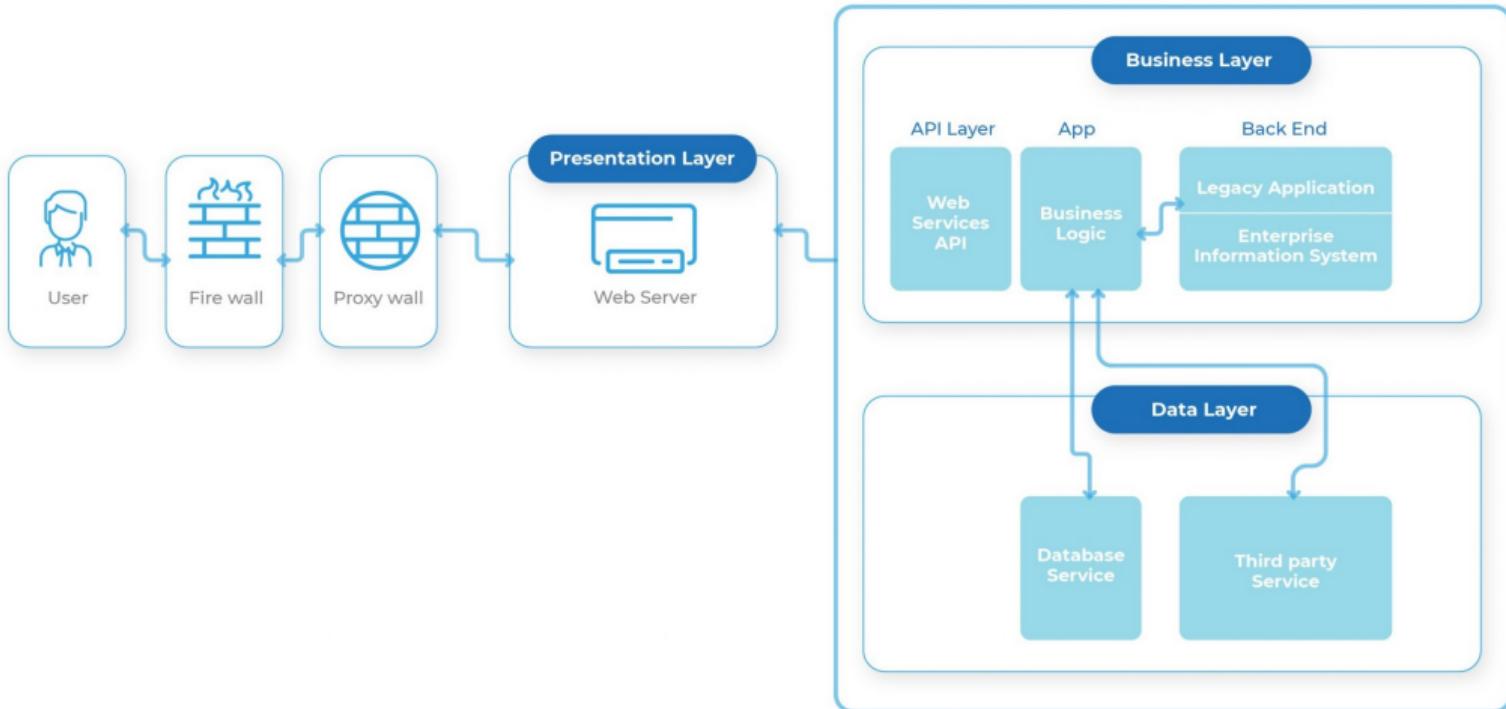
Back End

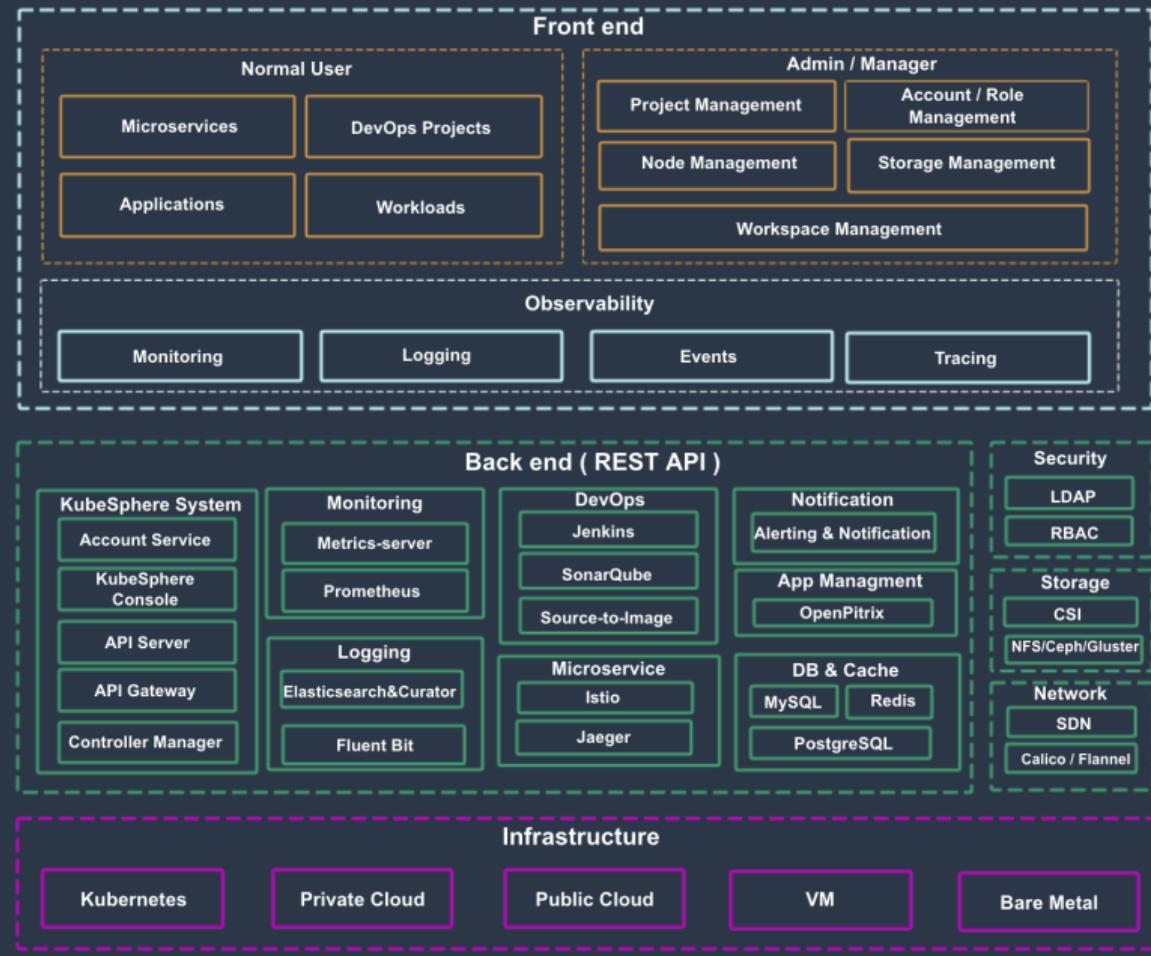
- Programming and scripting such as Python, Ruby and/or Perl
- Server architecture
- Database administration
- Scalability
- Security
- Data transformation
- Backup





Web Application Architecture Layers





Importancia de un servidor web para Internet de las Cosas

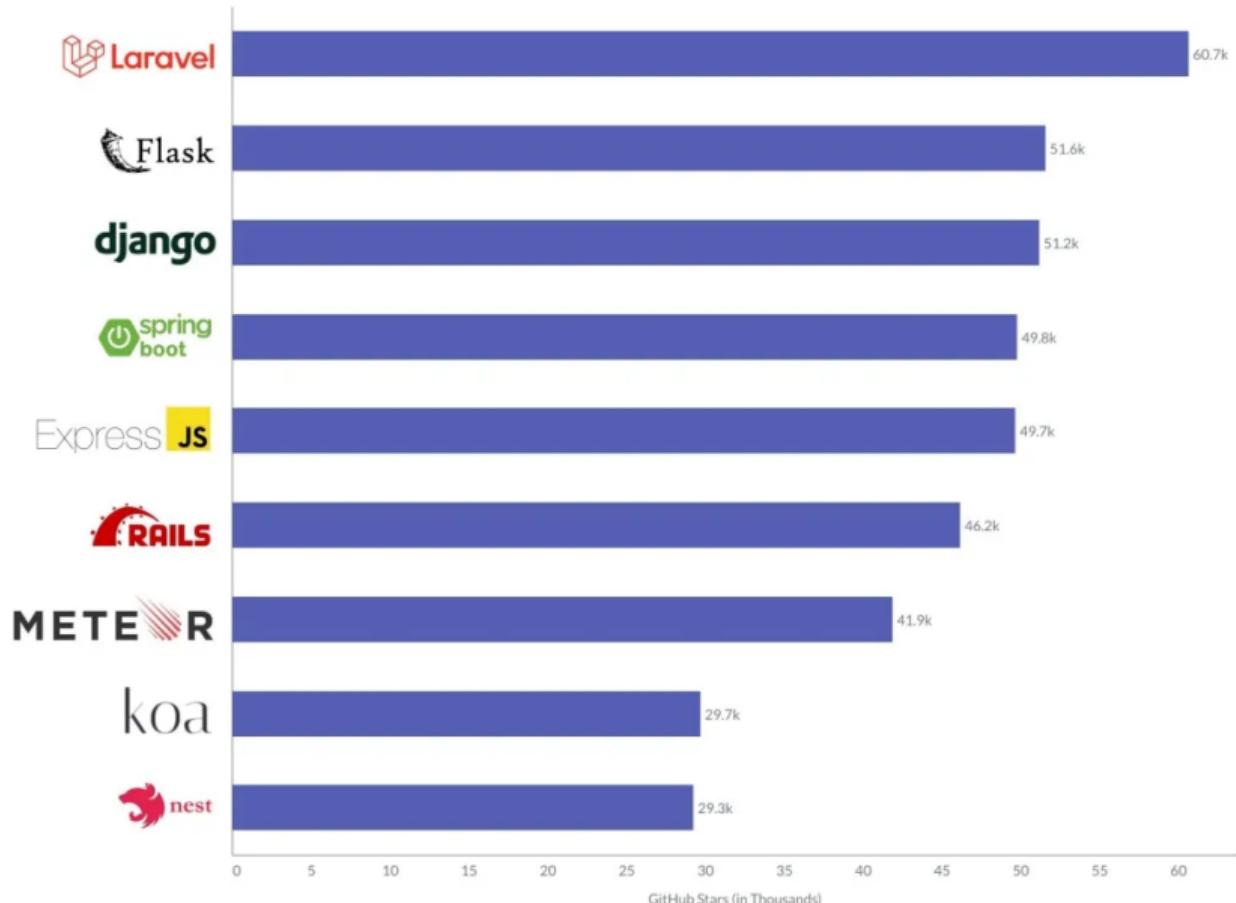
- Un servidor web es fundamental en el contexto de Internet de las Cosas, ya que actúa como el intermediario entre los dispositivos y los usuarios.
- Permite recibir y procesar los datos provenientes de los dispositivos conectados, realizar operaciones de control y enviar respuestas y visualizaciones relevantes a los usuarios.
- El servidor web puede realizar análisis de datos en tiempo real, almacenar información importante y facilitar la comunicación entre los diferentes componentes de un sistema de IoT.
- Además, un servidor web seguro y confiable es crucial para proteger la integridad y privacidad de los datos en un entorno de IoT.

Framework de desarrollo backend



Frameworks de desarrollo backend

- Los frameworks de desarrollo backend son herramientas que facilitan la creación de aplicaciones web y servicios en el lado del servidor.
- Estos frameworks ofrecen una estructura y conjunto de funcionalidades predefinidas para agilizar el proceso de desarrollo y promover buenas prácticas.
- Estos frameworks proporcionan una arquitectura robusta y modular para construir aplicaciones escalables y de alto rendimiento.
- Simplifican tareas comunes como el enrutamiento de URLs, el manejo de solicitudes y respuestas HTTP, el acceso a bases de datos y la gestión de sesiones de usuario.
- Al utilizar un framework, los desarrolladores pueden ahorrar tiempo y esfuerzo al aprovechar las características y soluciones implementadas por la comunidad de desarrollo.



Ejemplos de frameworks de desarrollo backend

- Flask es un ejemplo de framework de desarrollo backend ampliamente utilizado en el contexto de diseño de servicios web.
- Flask es un framework de desarrollo web ligero y flexible que permite crear aplicaciones web y servicios con Python.
- Otros frameworks populares incluyen Django (utilizando Python), Ruby on Rails (utilizando Ruby) y Express.js (utilizando JavaScript).

Introducción a Flask



Flask

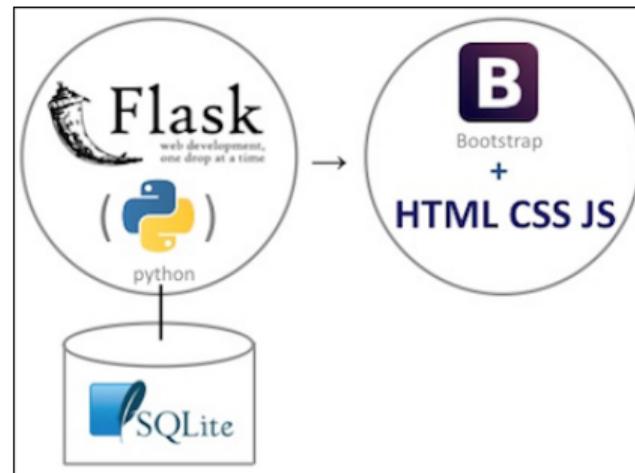
Introducción a Flask

- Flask es un framework web ligero y flexible escrito en Python.
- Permite crear aplicaciones web rápidas y escalables utilizando el lenguaje de programación Python.
- Es una herramienta popular para el desarrollo de servicios web y aplicaciones de Internet de las Cosas.



Descripción general del framework Flask

- Flask sigue el principio de "microframework", lo que significa que proporciona solo las funcionalidades esenciales para construir aplicaciones web sin imponer restricciones innecesarias. No requiere herramientas o bibliotecas particulares.
- Es fácil de aprender y usar, lo que lo hace ideal para proyectos pequeños y medianos.
- A pesar de ser minimalista, Flask es altamente personalizable y permite la incorporación de extensiones para agregar funcionalidades adicionales según sea necesario.



Descripción general del framework Flask

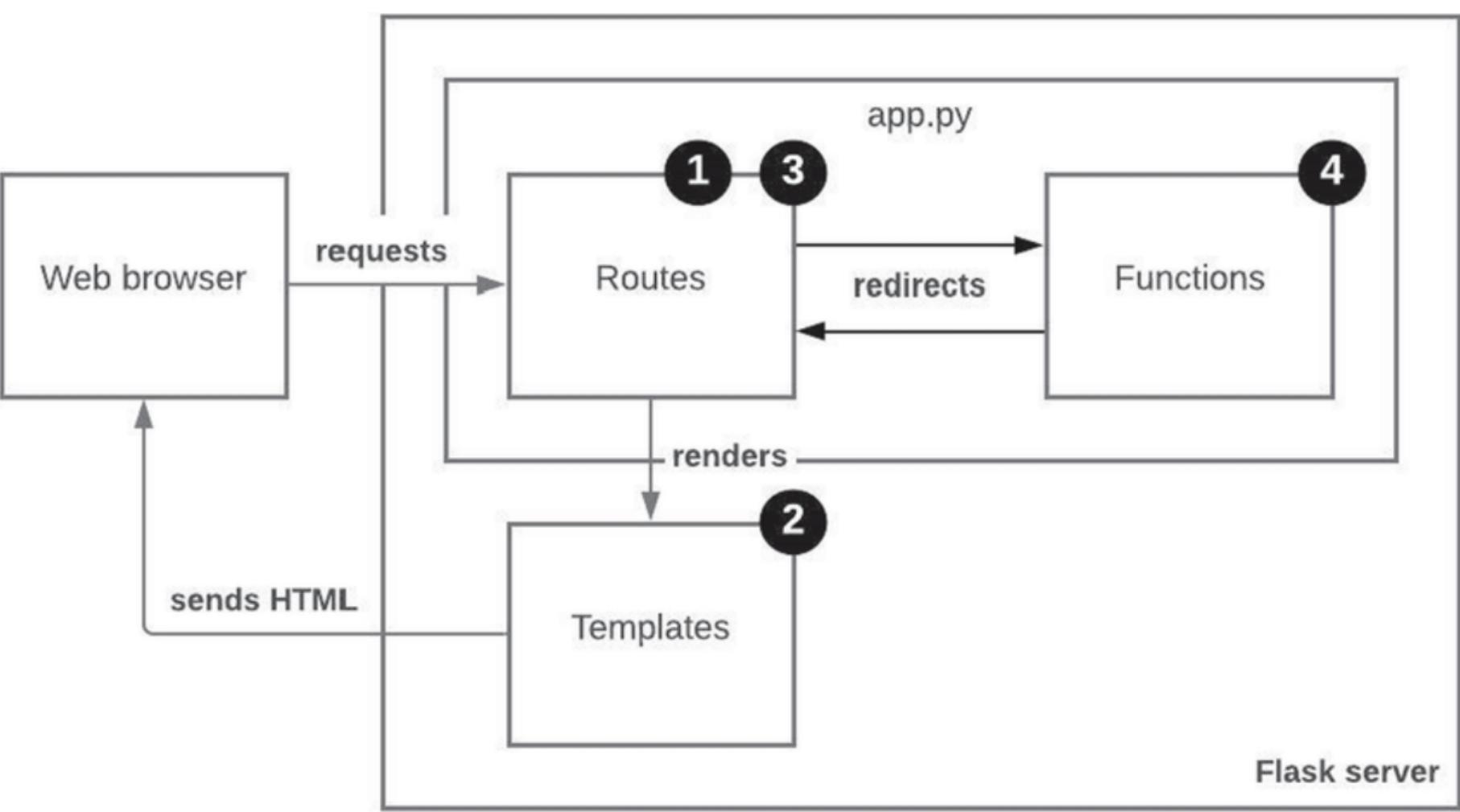
- No cuenta con una capa de abstracción de base de datos, validación de formularios ni otros componentes en los que bibliotecas de terceros proporcionen funciones comunes.
- Sin embargo, Flask admite extensiones que pueden agregar características a la aplicación como si estuvieran implementadas en Flask mismo.

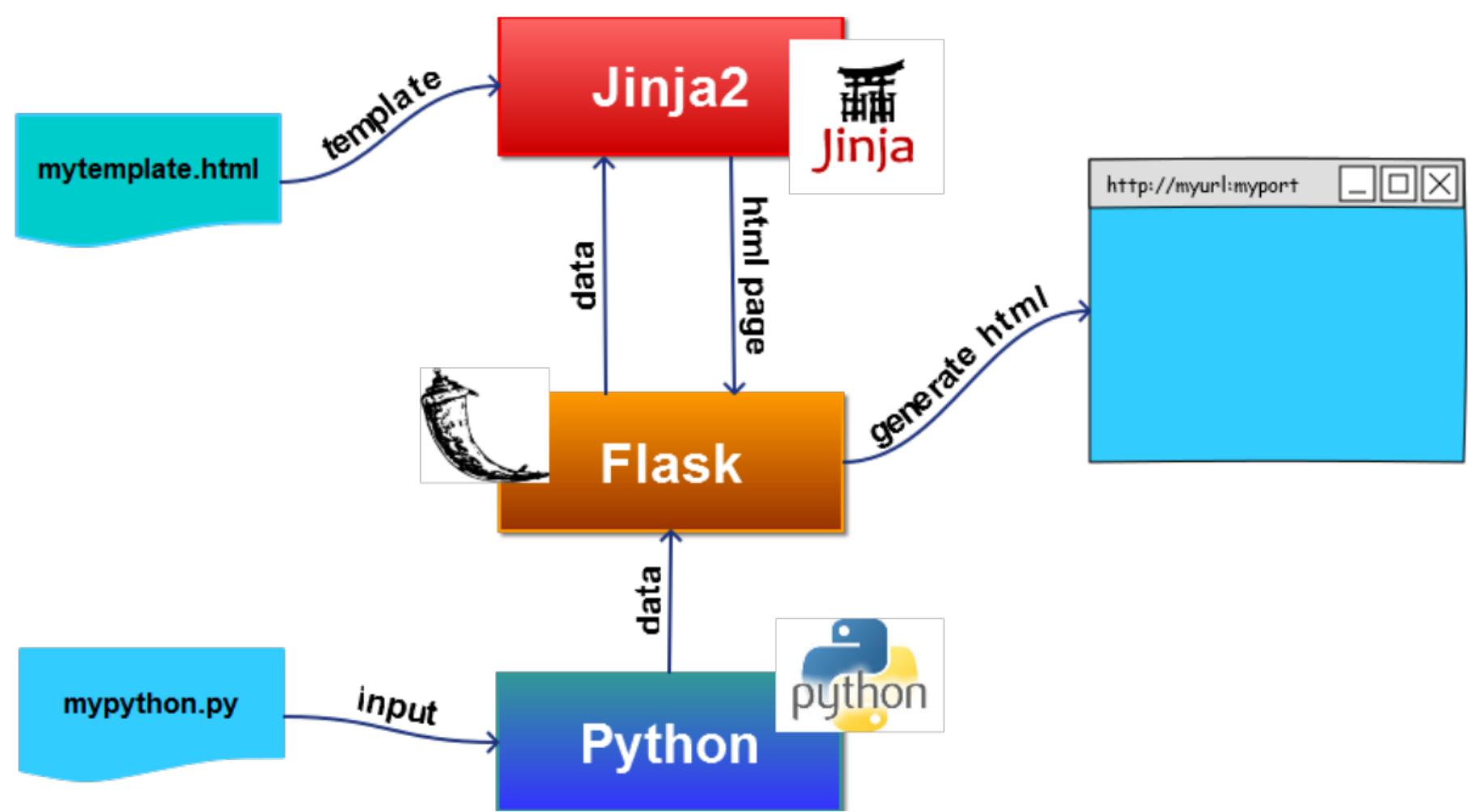
Conceptos básicos de Flask

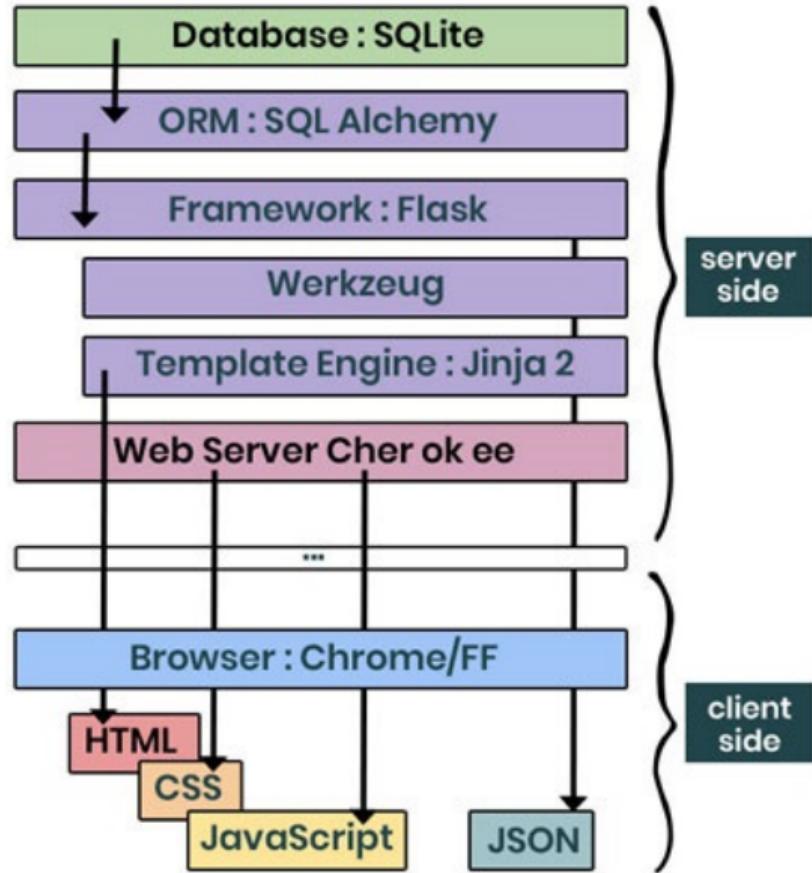
- Las rutas en Flask son URLs que el usuario puede visitar en la aplicación. Cada ruta se mapea a una función de Python llamada vista.
- Las vistas son responsables de manejar las solicitudes HTTP y generar las respuestas correspondientes.
- Las plantillas son archivos HTML con marcadores especiales que permiten la generación dinámica de contenido.
- Los controladores son funciones que realizan operaciones específicas, como el manejo de formularios o la interacción con bases de datos.

Herramientas y bibliotecas relacionadas con Flask

- Jinja2 es un sistema de plantillas utilizado por Flask para generar respuestas HTML dinámicas. Permite la inclusión de variables, estructuras de control y bloques de contenido reutilizable.
- WTForms es una biblioteca de manejo de formularios que simplifica la validación y procesamiento de datos enviados a través de formularios HTML. Facilita la creación de formularios seguros y robustos en aplicaciones Flask.







Trabajo con bases de datos: SQLite

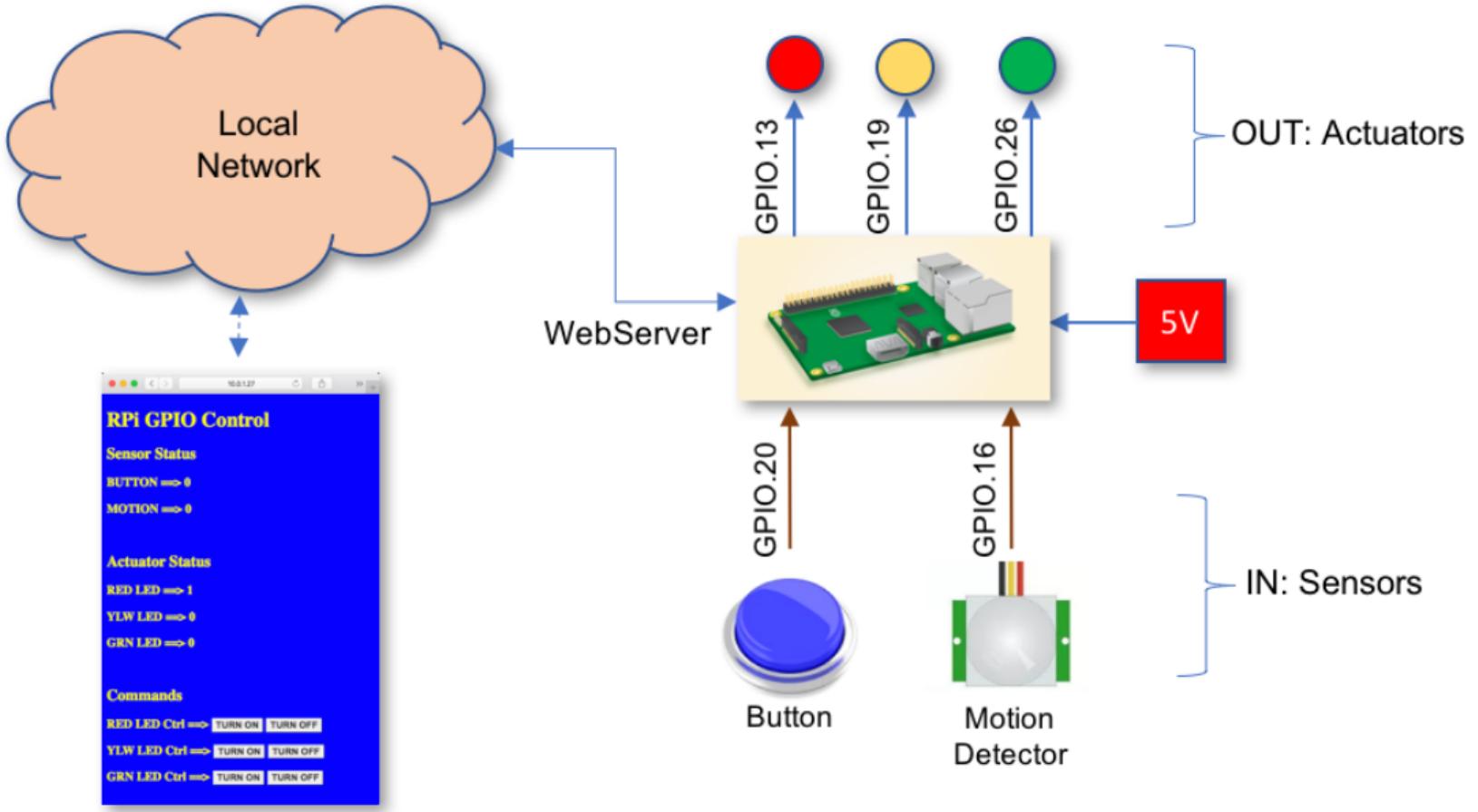
- SQLite es una base de datos relacional liviana y fácil de usar que se integra bien con Flask.
- Permite almacenar y recuperar datos de manera eficiente en aplicaciones web.
- Flask proporciona extensiones como Flask-SQLAlchemy para interactuar con bases de datos SQLite y simplificar las operaciones de CRUD (Crear, Leer, Actualizar y Eliminar).

Flask aplicado

Flask aplicado

Objetivo del tutorial:

En este tutorial, se utilizará una Raspberry Pi como un servidor web local, donde se controlarán tres de sus GPIOs programados como salidas (actuadores) y se monitorearán dos de sus GPIOs programados como entradas (sensores) a través de una página web sencilla.



Instalación de Flask

```
# Instalación de Flask en Raspberry Pi  
sudo apt-get install python3-flask
```

Para mantener tus archivos organizados, es recomendable crear una carpeta para tu proyecto.
Por ejemplo:

```
mkdir rpiWebServer
```

El comando anterior creará una carpeta llamada rpiWebServer donde guardaremos nuestros archivos Python (aplicación):

```
/home/pi/Documents/rpiWebServer
```

Instalación de Flask

Dentro de esta carpeta, crearemos otras dos subcarpetas: **static** para archivos **CSS** y eventualmente archivos **JavaScript**, y **templates** para archivos **HTML** (o más precisamente, plantillas **Jinja2**). Ve a la carpeta que acabas de crear:

```
cd rpiWebServer
```

Y crea las dos nuevas subcarpetas:

```
mkdir static  
mkdir templates
```

La estructura final de carpetas se verá así:

```
/rpiWebServer  
  /static  
  /templates
```

Aplicación Python WebServer

Para crear nuestro primer servidor web Python con Flask, sigue los siguientes pasos:

- 1 Abre tu entorno de desarrollo de **Python 3**, como VSC o Sublime.
- 2 Copia el siguiente código Hello World en tu entorno y guárdalo, por ejemplo, como `helloWorld.py`:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello world'

if __name__ == '__main__':
    app.run(debug=True, port=5005)
```

Aplicación Python WebServer

El código anterior realiza las siguientes acciones:

- Carga el módulo Flask en tu script de Python:

```
from flask import Flask
```

- Crea un objeto Flask llamado app:

```
app = Flask(__name__)
```

- Ejecuta la función index() cuando alguien acceda a la URL raíz ('/') del servidor. En este caso, solo envía el texto "Hello World" al navegador web del cliente mediante el comando return:

```
def index():
    return 'Hello world'
```

- Una vez que este script se esté ejecutando desde la línea de comandos en la terminal, el servidor comenzará a escuchar en el puerto 80, informando cualquier error:

```
if __name__ == '__main__':
    app.run(debug=True, port=5005)
```

Dirección IP de Raspberry Pi

Si no estás seguro acerca de la dirección IP de tu Raspberry Pi, puedes ejecutar el siguiente comando en tu terminal:

```
ifconfig
```

En la sección `wlan0:`, encontrarás la dirección IP. Por ejemplo:

```
127.168.0.18
```

Ejecución de la Aplicación

Ahora, ejecuta el programa de aplicación anterior:

```
sudo python3 helloWorld.py
```

- La aplicación se ejecutará a menos que escribas [CTRL] + [C].
- Ahora debes abrir cualquier navegador web que esté conectado a la misma red wifi que tu Raspberry Pi e ingresar su dirección IP, como se muestra en la segunda imagen anterior
- El Hello World debería aparecer en tu navegador.



10.0.1.27



>>



Hello World

File Edit Tabs Help

```
> ^C
pi@raspberrypi:~/Documents/rpiWebServer $ 
pi@raspberrypi:~/Documents/rpiWebServer $ ifconfig
eth0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    ether b8:27:eb:10:da:5f txqueuelen 1000 (Ethernet)
      RX packets 0 bytes 0 (0.0 B)
      RX errors 0 dropped 0 overruns 0 frame 0
      TX packets 0 bytes 0 (0.0 B)
      TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
        loop txqueuelen 1 (Local Loopback)
        RX packets 231833 bytes 40189187 (38.3 MiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 231833 bytes 40189187 (38.3 MiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.1.27 netmask 255.255.255.0 broadcast 10.0.1.255
    inet6 fe80::8f1a:c0ba:3679:708e prefixlen 64 scopeid 0x20<link>
        ether b8:27:eb:45:8f:0a txqueuelen 1000 (Ethernet)
        RX packets 543880 bytes 108830029 (103.7 MiB)
        RX errors 0 dropped 467 overruns 0 frame 0
        TX packets 606367 bytes 579127740 (552.2 MiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

pi@raspberrypi:~/Documents/rpiWebServer $ sudo python3 helloWorld.py
* Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger pin code: 355-015-716
10.0.1.27 - - [04/Jan/2018 15:40:48] "GET / HTTP/1.1" 200 -
10.0.1.10 - - [04/Jan/2018 16:14:12] "GET / HTTP/1.1" 200 -
```

Creando una Página Web de Servidor Apropriada

Vamos a mejorar nuestra aplicación Hello World creando una plantilla HTML y un archivo CSS para dar estilo a nuestra página. Esto es importante, de lo contrario, complicaríamos el script de Python al poner todo en él.

Plantillas

- Se crea un archivo HTML que estará ubicado en la subcarpeta /templates, podemos utilizar archivos separados con marcadores de posición donde deseamos insertar datos dinámicos.
- Crearemos un archivo llamado index.html, que se guardará en /templates. Puedes usar cualquier editor de texto para crear tu archivo HTML, como VSC, Sublime, o nano en la terminal o el editor de texto de la Raspberry Pi (LeafPad) que se encuentra en el menú principal Accesorios.
- VSC puede ser utilizado para trabajar con todos los archivos del proyecto al mismo tiempo (.py, .html y .css). Es una buena opción para proyectos más grandes y complejos.

Creando una Página Web de Servidor Apropriada

Creemos /templates/index.html:

```
<!DOCTYPE html>
<head>
<title>{{ title }}</title>
</head>
<body>
<h1>Hello, World!</h1>
<h2>The date and time on the server is: {{ time }}</h2>
</body>
</html>
```

Todo lo que está entre llaves dobles dentro de la plantilla HTML se interpreta como una variable que será pasada desde el *script* de Python mediante la función `render_template`.

Creando una Página Web de Servidor Apropriada (continuación)

Se crea un nuevo script de Python. LO llamaremos helloWorldTemplate.py:

```
from flask import Flask, render_template
import datetime

app = Flask(__name__)
@app.route("/")

def hello():
    now = datetime.datetime.now()
    timeString = now.strftime("%Y-%m-%d %H:%M")
    templateData = {
        'title' : 'HELLO!',
        'time': timeString
    }
    return render_template('index.html', **templateData)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80, debug=True)
```

Creando una Página Web de Servidor Apropriada (continuación)

- Observa que creamos una cadena de formato (`timeString`) utilizando la fecha y hora del objeto `now`, que almacena la hora actual.
- Otra cosa importante en el código anterior es que creamos un diccionario de variables (un conjunto de claves, como el título que se asocia con valores, como `HELLO!`) para pasar a la plantilla. En el `return`, devolveremos la plantilla `index.html` al navegador web utilizando las variables en el diccionario `templateData`.

Ejecuta el script de Python:

```
sudo python3 helloWorldTemplate.py
```

- Abre cualquier navegador web e ingresa la dirección IP de tu Raspberry Pi. La imagen anterior muestra el resultado.
- Observa que el contenido de la página cambia dinámicamente cada vez que la actualizas con los datos de variable actualizados pasados por el script de Python. En nuestro caso, `title` es un valor fijo, pero `time` cambia cada segundo.

Creando una Página Web de Servidor Apropriada (continuación)

Ahora, agreguemos estilo a nuestra página creando un archivo CSS y guárdalo en /static/style.css:

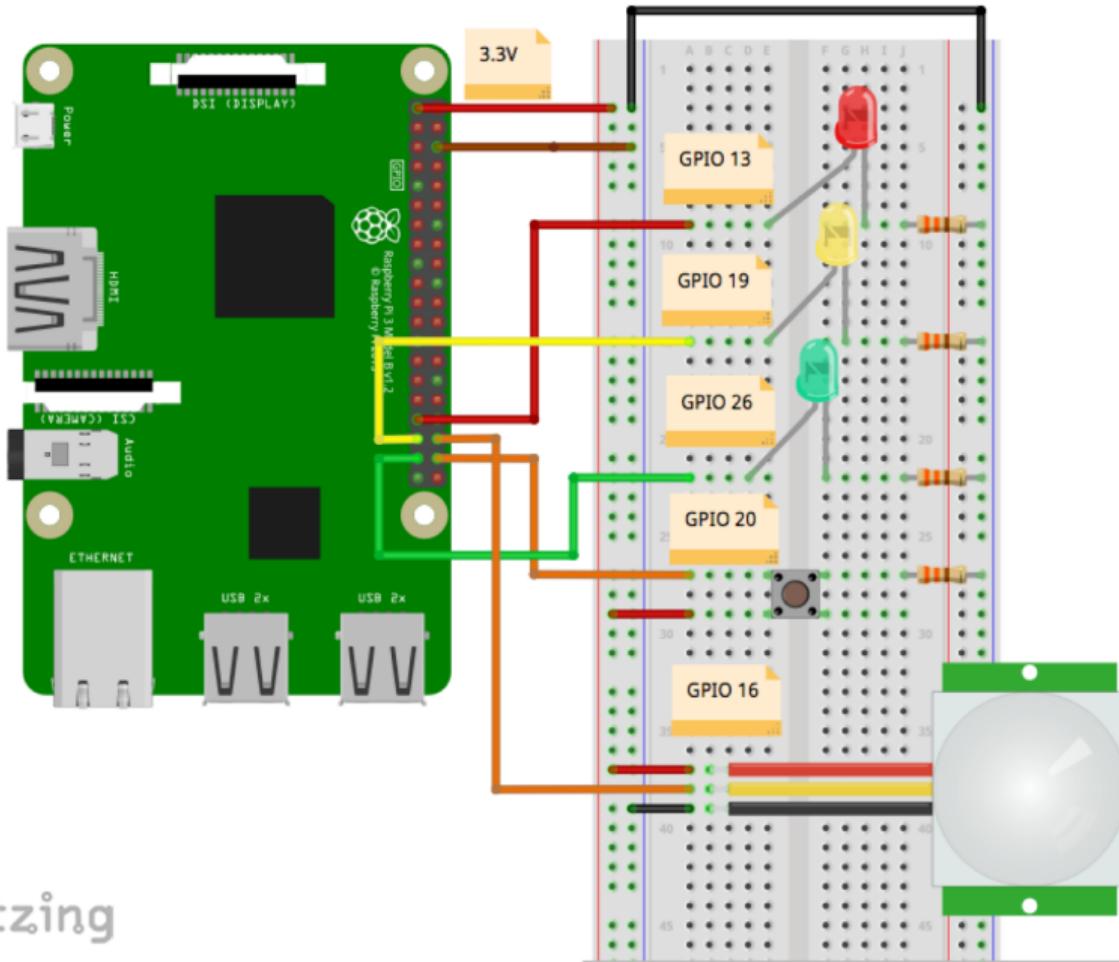
```
body{  
    background: blue;  
    color: yellow;  
}
```

También debes modificar el archivo index.html para indicarle que busque el archivo style.css. Haz esto insertando el enlace link en head:

```
<!DOCTYPE html>  
<head>  
<title>{{ title }}</title>  
<link rel="stylesheet" href="../static/style.css/">  
</head>  
<body>  
<h1>Hello, World!</h1>  
<h2>The date and time on the server is: {{ time }}</h2>  
</body>  
</html>
```

Recuerda que index.html está dentro de /template y style.css está dentro de /static, por lo tanto, debes indicarle al HTML que suba y baje nuevamente para buscar la subcarpeta

EI hardware



fritzing

Leyendo el Estado de los GPIO

Vamos a leer el estado de nuestros **sensores** mediante la monitorización de sus GPIO.

El script de Python Creemos un nuevo script de Python y llamémoslo app.py:

```
'''  
Raspberry Pi GPIO Status and Control  
'''  
  
import RPi.GPIO as GPIO  
from flask import Flask, render_template  
  
app = Flask(__name__)  
  
GPIO.setmode(GPIO.BCM)  
GPIO.setwarnings(False)  
button = 20  
senPIR = 16  
buttonSts = GPIO.LOW  
senPIRSts = GPIO.LOW  
  
# Configura los pines del botón y el sensor PIR como entrada  
GPIO.setup(button, GPIO.IN)  
GPIO.setup(senPIR, GPIO.IN)
```

Leyendo el Estado de los GPIO

```
@app.route("/")
def index():
    # Leer el estado de los sensores
    buttonSts = GPIO.input(button)
    senPIRSts = GPIO.input(senPIR)
    templateData = {
        'title' : 'Estado de Entrada GPIO',
        'button' : buttonSts,
        'senPIR' : senPIRSts
    }
    return render_template('index.html', **templateData)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80, debug=True)
```

Leyendo el Estado de los GPIO (continuación)

- Se definen los GPIO 20 y 16 como entrada, leer sus valores y almacenarlos en 2 variables: `buttonSts` y `senPIRSts`.
- Dentro de la función `index()`, pasaremos esos valores a nuestra página web a través de las variables `button` y `senPIR`, que forman parte de nuestro diccionario de variables: `templateData`.

Una vez que hayas creado `app.py`, ejecútalo en tu terminal:

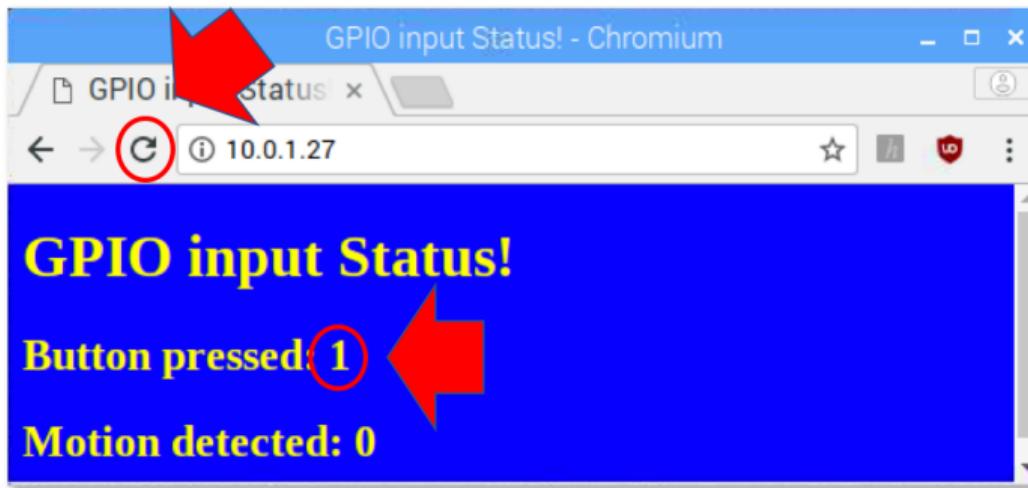
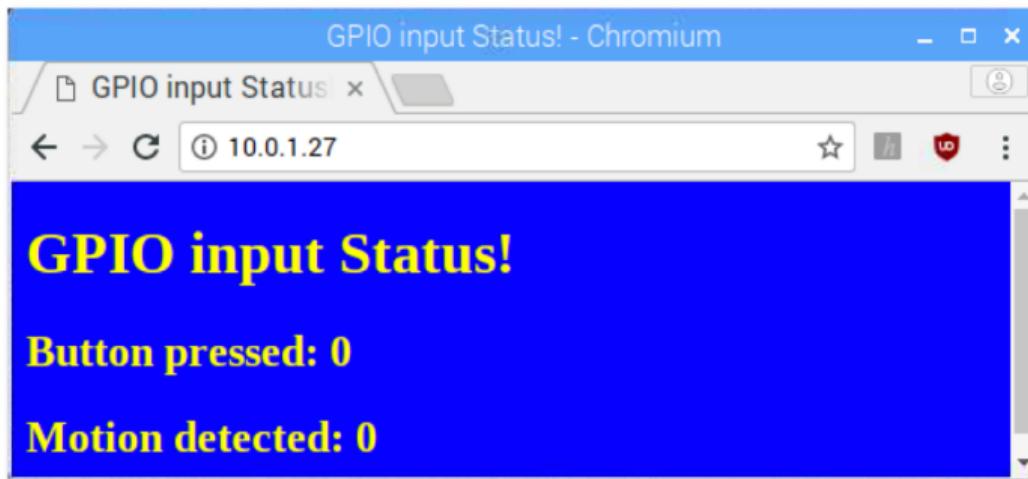
```
python3 app.py
```

Leyendo el Estado de los GPIO (continuación)

La plantilla También creamos un nuevo archivo index.html para mostrar el estado de los GPIO de ambos sensores:

```
<!DOCTYPE html>
<head>
<title>{{ title }}</title>
<link rel="stylesheet" href='../static/style.css' />
</head>
<body>
<h1>{{ title }}</h1>
<h2>Botón presionado: {{ button }}</h2>
<h2>Detección de movimiento: {{ senPIR }}</h2>
</body>
</html>
```

Presiona el botón o realiza un movimiento frente al sensor PIR y actualiza la página.



Controlando los GPIO

Controlando los GPIO

- Ahora que sabemos cómo **leer** el estado de los GPIO, cambiemos su estado.
- Lo que haremos es **comandar** los **actuadores** a través de una página web
- Tenemos 3 LEDs conectados a los GPIO de la Raspberry.
- Al controlarlos remotamente, cambiaremos su estado de **BAJO (LOW)** a **ALTO (HIGH)** y viceversa
- En lugar de LEDs, podríamos tener relés controlando la lámpara y/o el ventilador de una habitación, por ejemplo.

Controlando los GPIO: El script de Python

Creemos un nuevo script de Python y llamémoslo app.py:

```
'''  
Raspberry Pi GPIO Status and Control  
'''  
  
import RPi.GPIO as GPIO  
from flask import Flask, render_template, request  
  
app = Flask(__name__)  
  
GPIO.setmode(GPIO.BCM)  
GPIO.setwarnings(False)  
  
# Define actuators GPIOs  
ledRed = 13  
ledYlw = 19  
ledGrn = 26  
  
# Initialize GPIO status variables  
ledRedSts = 0  
ledYlwSts = 0  
ledGrnSts = 0
```

Controlando los GPIO: El script de Python

```
# Define led pins as output
GPIO.setup(ledRed, GPIO.OUT)
GPIO.setup(ledYlw, GPIO.OUT)
GPIO.setup(ledGrn, GPIO.OUT)

# Turn leds OFF
GPIO.output(ledRed, GPIO.LOW)
GPIO.output(ledYlw, GPIO.LOW)
GPIO.output(ledGrn, GPIO.LOW)
```

Controlando los GPIO: El script de Python

```
@app.route("/")
def index():
    # Read Sensors Status
    ledRedSts = GPIO.input(ledRed)
    ledYlwSts = GPIO.input(ledYlw)
    ledGrnSts = GPIO.input(ledGrn)

    templateData = {
        'title': '|Estado de Salida GPIO!',
        'ledRed': ledRedSts,
        'ledYlw': ledYlwSts,
        'ledGrn': ledGrnSts,
    }
    return render_template('index.html', **templateData)
```

```
@app.route("/<deviceName>/<action>")
def action(deviceName, action):
    if deviceName == 'ledRed':
        actuator = ledRed
    if deviceName == 'ledYlw':
        actuator = ledYlw
    if deviceName == 'ledGrn':
        actuator = ledGrn

    if action == "on":
        GPIO.output(actuator, GPIO.HIGH)
    if action == "off":
        GPIO.output(actuator, GPIO.LOW)

    ledRedSts = GPIO.input(ledRed)
    ledYlwSts = GPIO.input(ledYlw)
    ledGrnSts = GPIO.input(ledGrn)

    templateData = {
        'ledRed': ledRedSts,
        'ledYlw': ledYlwSts,
        'ledGrn': ledGrnSts,
    }
    return render_template('index.html', **templateData)
```

Controlando los GPIO: El script de Python

```
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80, debug=True)
```

Controlando los GPIO (continuación)

Lo nuevo en el código anterior es la nueva **ruta** definida:

```
@app.route("/<deviceName>/<action>")
```

Desde la página web, se generarán llamadas con el siguiente formato:

`http://10.0.1.27/ledRed/on`

`http://10.0.1.27/ledRed/off`

En el ejemplo anterior, ledRed es el `deviceName` y **on** u **off** son ejemplos de posibles **actions** (acciones).

Controlando los GPIO (continuación)

Esas rutas serán identificadas y *procesadas* adecuadamente. Los pasos principales son:

- Convertir la cadena ledRed, por ejemplo, en su pin GPIO equivalente. La variable entera ledRed es equivalente a **GPIO13**. Almacenaremos este valor en la variable actuator.
- Para cada actuador, analizaremos la **acción** o **comando** y actuaremos en consecuencia. Si la **acción** es **on**, por ejemplo, debemos usar el comando: `GPIO.output(actuator, GPIO.HIGH)`.
- Actualizar el estado de cada actuador.
- Actualizar la biblioteca de variables.
- Devolver los datos a `index.html`

Controlando los GPIO: La plantilla

Ahora creamos un archivo index.html para mostrar el estado de los GPIO de cada actuador y, lo que es más importante, crear **botones** para enviar los comandos:

```
<!DOCTYPE html>
<head>
<title>Control de GPIO</title>
<link rel="stylesheet" href='../static/style.css' />
</head>
<body>
<h1>Actuadores</h1>
<h2>Estado</h2>
<h3>LED ROJO ==> {{ ledRed }}</h3>
<h3>LED AMARILLO ==> {{ ledYlw }}</h3>
<h3>LED VERDE ==> {{ ledGrn }}</h3>
<br>
<h2>Comandos</h2>
<h3>
Control LED ROJO ==>
<a href="/ledRed/on" class="button">ENCENDER</a>
<a href="/ledRed/off" class="button">APAGAR</a>
</h3>
<h3>
Control LED AMARILLO ==>
<a href="/ledYlw/on" class="button">ENCENDER</a>
<a href="/ledYlw/off" class="button">APAGAR</a>
</h3>
<h3>
Control LED VERDE ==>
```

Controlando los GPIO: La plantilla

A continuación, se muestra el archivo style.css:

```
body {  
    background: blue;  
    color: yellow;  
}  
  
.button {  
    font: bold 15px Arial;  
    text-decoration: none;  
    background-color: #EEEEEE;  
    color: #333333;  
    padding: 2px 6px 2px 6px;  
    border-top: 1px solid #CCCCCC;  
    border-right: 1px solid #333333;  
    border-bottom: 1px solid #333333;  
    border-left: 1px solid #CCCCCC;  
}
```



10.0.1.27



>>



Actuators

Status

RED LED ==> 0

YLW LED ==> 1

GRN LED ==> 0

Commands

RED LED Ctrl ==> **TURN ON** **TURN OFF**

YLW LED Ctrl ==> **TURN ON** **TURN OFF**

GRN LED Ctrl ==> **TURN ON** **TURN OFF**

Integración de sensores y actuadores



Controlando GPIOs: El script de Python final

Ahora debemos unir las 2 partes que desarrollamos anteriormente:

```
'''  
Estado y control de GPIO de Raspberry Pi  
'''  
  
import RPi.GPIO as GPIO  
from flask import Flask, render_template, request  
  
app = Flask(__name__)
```

Controlando GPIOs: El script de Python final

```
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
# Definir GPIOs de los sensores
button = 20
senPIR = 16
# Definir GPIOs de los actuadores
ledRed = 13
ledYlw = 19
ledGrn = 26
# Inicializar variables de estado de los GPIOs
buttonSts = 0
senPIRSts = 0
ledRedSts = 0
ledYlwSts = 0
ledGrnSts = 0
```

Controlando GPIOs: El script de Python final

```
# Configurar los pines de los botones y el sensor PIR como entrada
GPIO.setup(button, GPIO.IN)
GPIO.setup(senPIR, GPIO.IN)

# Configurar los pines de los LEDs como salida
GPIO.setup(ledRed, GPIO.OUT)
GPIO.setup(ledYlw, GPIO.OUT)
GPIO.setup(ledGrn, GPIO.OUT)

# Apagar los LEDs
GPIO.output(ledRed, GPIO.LOW)
GPIO.output(ledYlw, GPIO.LOW)
GPIO.output(ledGrn, GPIO.LOW)
```

Controlando GPIOs: El script de Python final

```
@app.route("/")
def index():
    # Leer el estado de los GPIOs
    buttonSts = GPIO.input(button)
    senPIRSts = GPIO.input(senPIR)
    ledRedSts = GPIO.input(ledRed)
    ledYlwSts = GPIO.input(ledYlw)
    ledGrnSts = GPIO.input(ledGrn)
    templateData = {
        'button': buttonSts,
        'senPIR': senPIRSts,
        'ledRed': ledRedSts,
        'ledYlw': ledYlwSts,
        'ledGrn': ledGrnSts,
    }
    return render_template('index.html', **templateData)
```

```
@app.route("/<deviceName>/<action>")
def action(deviceName, action):
    if deviceName == 'ledRed':
        actuator = ledRed
    if deviceName == 'ledYlw':
        actuator = ledYlw
    if deviceName == 'ledGrn':
        actuator = ledGrn

    if action == "on":
        GPIO.output(actuator, GPIO.HIGH)
    if action == "off":
        GPIO.output(actuator, GPIO.LOW)
```

```
buttonSts = GPIO.input(button)
senPIRSts = GPIO.input(senPIR)
ledRedSts = GPIO.input(ledRed)
ledYlwSts = GPIO.input(ledYlw)
ledGrnSts = GPIO.input(ledGrn)

templateData = {
    'button': buttonSts,
    'senPIR': senPIRSts,
    'ledRed': ledRedSts,
    'ledYlw': ledYlwSts,
    'ledGrn': ledGrnSts,
}
return render_template('index.html', **templateData)
```

Controlando GPIOs: El script de Python final

```
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80, debug=True)
```

RPi GPIO Control

Sensor Status

BUTTON ==> 0

MOTION ==> 0

Actuator Status

RED LED ==> 1

YLW LED ==> 0

GRN LED ==> 0

Commands

RED LED Ctrl ==> [TURN ON](#) [TURN OFF](#)

YLW LED Ctrl ==> [TURN ON](#) [TURN OFF](#)

GRN LED Ctrl ==> [TURN ON](#) [TURN OFF](#)

Más allá con las plantillas

Más allá con las plantillas

- Como discutimos brevemente antes, la función `render_template()` invoca el motor de plantillas **Jinja2** que viene incluido con el framework Flask.
- **Jinja2** sustituye los bloques `{{...}}` con los valores correspondientes, dados por los argumentos proporcionados en la llamada a `render_template()`.
- Pero esta sustitución no es lo único que **Jinja2** puede hacer. Por ejemplo, las plantillas también admiten declaraciones de control, que se colocan dentro de bloques `{% ... %}`.
- Podemos cambiar nuestra plantilla `index.html` para agregar declaraciones condicionales y mostrar un botón específico dependiendo del valor real del actuador. En otras palabras, **alternar** la condición del actuador.

Más allá con las plantillas

Vamos a reescribir la parte de estado y comando de index.html, utilizando declaraciones condicionales:

```
<!DOCTYPE html>
<head>
<title>GPIO Control</title>
<link rel="stylesheet" href='../../static/master.css' />
</head>
```

```
<body>
<h1>RPi GPIO Control</h1>
<h2> Sensor Status </h2>
<h3> BUTTON ==> {{ button }}</h3>
<h3> MOTION ==> {{ senPIR }}</h3>
<br>
```

```
<h2> Actuator Status & Control </h2>
<h3> RED LED ==> {{ ledRed }} ==>
{% if ledRed == 1 %}
<a href="/ledRed/off" class="button">TURN OFF</a>
{% else %}
<a href="/ledRed/on" class="button">TURN ON</a>
{% endif %}
</h3>
<h3> YLW LED ==> {{ ledYlw }} ==>
{% if ledYlw == 1 %}
<a href="/ledYlw/off" class="button">TURN OFF</a>
{% else %}
<a href="/ledYlw/on" class="button">TURN ON</a>
{% endif %}
</h3>
<h3> GRN LED ==> {{ ledGrn }} ==>
{% if ledGrn == 1 %}
<a href="/ledGrn/off" class="button">TURN OFF</a>
{% else %}
<a href="/ledGrn/on" class="button">TURN ON</a>
{% endif %}
</h3>
</body>
</html>
```

¡Muchas gracias por su atención!

¿Preguntas?



Contacto: Marco Teran
webpage: marcoteran.github.io/
e-mail: marco.teran@usa.edu.co

