PTOS antes de comenzar:

Instalar librerías en cmd

* Pip install urllib3
* Pip install requests

Instalar extensión Chrome json viewer.

Una URL se compone de los siguientes elementos

HTTPS://API.ipgeolocation.io/ipgeo? + apiKey=12344567 + ip=8.8.8.8

URL BASE + PARAMETRO1 + PARAMETRO2

Aunque también hay algunas que vienen completas y solo se le agrega el parámetro como es la de Pokémon.

Ejemplo de script con 3 parámetros (api de ipgeolocation):

Texto

Descripción generada automáticamente

Aquí se puede ver que importamos librerías urllib.parse y requests

Quien ocupe el código deberá ingresar una ip y el código le responderá con información seleccionada sobre ella

La variable:

req = urlbase + urllib.parse.urlencode({"apiKey":api, "ip":ip})

nos permite el armado del URL, donde se ven los 3 parámetros; **urlbase+apikey+ip**. Pero es ***urllib.parse.urlencode*** quien realiza la tarea.

La variable :

consulta = requests.get(req).json()

nos permitirá consultar al URL creado sobre información de la IP ingresada. Como vemos todo se le consulta a la variable “consulta” ya que es quien tendrá la información y la mostrara en formato json.

Para sacar información detallada de un archivo json ocupamos lo siguiente:

dst= consulta["time\_zone"]["dst\_end"]["utc\_time"]

aquí estamos creando la variable ***dst*** la cual le pedirá a Consulta (quien puede obtener info del url) sobre el dato de “utc\_time”. Se ingresan antes time\_zone y dst\_end entre [] ya que “utc\_time” está contenido en ellos, ósea para llegar a este dato debo pasar por estas dos keys.

CASO API POKEMON:

Texto

Descripción generada automáticamente

Se importan las mismas librerías

Pero no se ocupa ***urllib.parse.urlencode*** ya que no es necesario, porque a esta api solo debemos agregarle el parámetro, que en este caso seria el nombre del Pokémon, ósea la misma URL trae consigo misma la API.

Por lo tanto solo necesitamos sumar las variables URL (dada por la pagina de la api) + nombre (quien tendrá el input para ingresar el Pokémon). Esta suma se ve en :

req = url + nombre

la variable consulta se hace del mismo modo anterior, ocupando la librería ***requests***: ***requests.get(req).json()*** . donde req sería el URL creado. A la variable consulta se le harán todas las peticiones de datos [].

Hay una variable diferente que es:

habilidad = [ hab["ability"]["name"] for hab in  consulta["abilities"] ]

print("Habilidades", ", " .join(habilidad))

aquí podemos ver que dentro de la variable habilidad se crea una variable ***hab*** que contiene los parámetros de dato que buscamos, además se le inluye un bucle FOR el cual busca los datos de variable ***hab*** en (IN) la variable ***consulta*** que contiene el dato *abilities*. A su ves *abilities* contiene a ability y name. Esto se hace porque hay mas de una habilidad presente en los Pokémon.

Los demás valores, al ser únicos, se les recopila de forma normal, consultando a variable ***consulta[“parámetro a buscar”]***. Cabe destacar que este parámetro debe estar escrito dentro de los corchetes tal como esta en el diccionario json.

DOCKERS

**Mkdir Docker**

**Cd Docker**

**docker -h** -> ver comandos de Docker

Docker images en Google 🡪 vere las imágenes existentes

**Vim index.html**

<html>

<head>

<title>HOLITA</title>

</head>

<body>

<h1> ENSAYOO </h1>

</body>

</html>

:x -> guardar y salir

**Vim Dockerfile**

FROM Ubuntu:latest 🡪 imagen de Ubuntu:ultima versión

RUN apt-get update && \

apt-get install -y apache2 && \

rm -rf /var/lib/apt/lists/\*

COPY index.html /var/www/html

EXPOSE 80

CMD [“/usr/sbin/apache2ctl”, “-D”, “FOREGROUND”] 🡪 comando para correr apache en primer palno

**docker build -t web1** . 🡪 para construir el Docker

**docker run -d -p 80:80 –name web1 web 1** 🡪 ejecutar el docker

**docker ps** 🡪 debe mostrame el Docker

**docker logs web1** 🡪 muestra errores

**docker container stop web1** 🡪 para el Docker, cuando hay error y hay que solucionar.

**docker container rm web1** 🡪 elimino el Docker

para buscar: localhost

Docker rmi -f “IDimagen” – eliminar imagen docker

DOCKER CON PYTHON

Debemos tener 3 archivos para este Docker: el archivo Python, index y Dockerfile.

vim ejemplo.py

from flask import Flask

from flask import request

from flask import render\_template

var = Flask(\_\_name\_\_)

@var.route("/")

def main():

return render\_template("index.html")

if \_\_name\_\_ == "\_\_main\_\_":

var.run(host="0.0.0.0", port=8080)

Texto

Descripción generada automáticamente\*\*\*\* el (if \_\_name\_\_ == "\_\_main\_\_":) es solo una igualdad que siempre será verdadera.

Por ejemplo, definir peso=1

Y luego poner

If peso ==1 :

var.run(host="0.0.0.0", port=8080)

vim Dockerfile

FROM python

RUN apt-get update -y

RUN apt-get install -y python3-pip

RUN pip install flask

COPY index.html /home/myapp/templates/

COPY ejemplo.py /home/myapp

EXPOSE 8080

CMD python3 /home/myapp/ejemplo.py

Vim index.html

<html>

<head>

<title>HOLITA</title>

</head>

<body>

<h1> ENSAYOO </h1>

</body>

</h

:x -> guardar y salir

Docker build -t web2 .

Docker run -d -p 8080:8080 --name web2 web2

\*\*\*hay que recordar que toda palabra **python** es en minúscula.

Para buscar: local host:8080

Texto

Descripción generada automáticamenteEXPLICACION CODIGO:

***FLASK***

1. **\*\*Instalación de Flask\*\*:** Primero, necesitas instalar Flask en tu entorno de desarrollo. Puedes hacerlo usando pip, el administrador de paquetes de Python. Ejecuta el siguiente comando en tu terminal:

Patrón de fondo

Descripción generada automáticamente

2. **\*\*Crea tu aplicación\*\*:** Una vez que hayas instalado Flask, puedes comenzar a construir tu aplicación. Crea un nuevo archivo Python para tu aplicación y comienza importando Flask:



3. **\*\*Define las rutas\*\*:** Las rutas en Flask son los puntos de acceso a tu aplicación web. Puedes definir rutas utilizando el decorador `@app.route()`. Por ejemplo:



En este ejemplo, la ruta `'/'` es la raíz de tu aplicación, y la función `index()` se ejecutará cuando alguien acceda a esa ruta. La función devuelve `'¡Hola, mundo!'`.

4. **\*\*Ejecuta la aplicación\*\***: Para ejecutar tu aplicación Flask, simplemente agrega el siguiente código al final de tu archivo Python:



Esto arrancará el servidor de desarrollo de Flask. Ahora puedes acceder a tu aplicación en tu navegador visitando la dirección `http://localhost:5000`.

Estos son solo los pasos básicos para comenzar con Flask. A medida que avances, aprenderás sobre aspectos más avanzados, como el manejo de plantillas, el acceso a bases de datos, la implementación de autenticación y autorización, entre otros. La documentación oficial de Flask es una excelente fuente de información para aprender más sobre cómo utilizar todas las características que ofrece.

**EXPLICACION DE RUTAS**

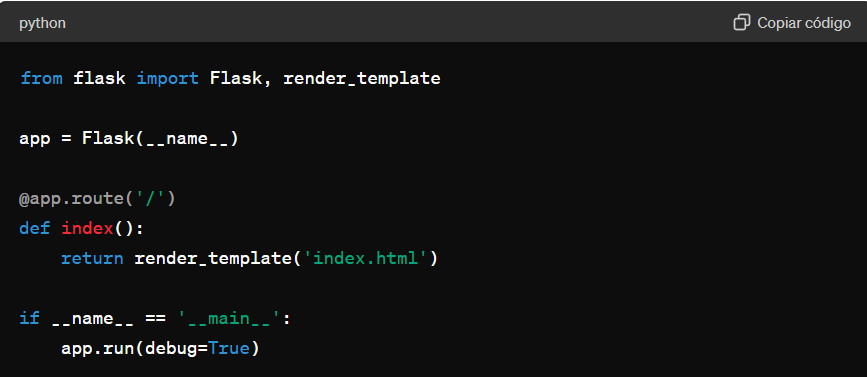
La ruta `home/myapp/templates/` parece ser una ruta en un sistema de archivos, y por la estructura de la ruta, parece estar relacionada con una aplicación Flask.

Texto

Descripción generada automáticamenteEn el contexto de una aplicación Flask, la carpeta `templates` es donde generalmente se almacenan las plantillas HTML que la aplicación utiliza para renderizar las páginas web dinámicas. Cuando creas una aplicación Flask, es común organizar tu proyecto de la siguiente manera:

En esta estructura, `app.py` es el archivo principal de tu aplicación Flask, `templates` es la carpeta donde se almacenan las plantillas HTML y `static` es donde se almacenan los archivos estáticos como hojas de estilo CSS, scripts JavaScript, imágenes, etc.

Cuando defines una ruta en Flask que renderiza una plantilla, generalmente utilizas la función `render\_template` para renderizar la plantilla desde la carpeta `templates`. Por ejemplo:



En este ejemplo, cuando alguien accede a la ruta raíz `'/'`, Flask renderizará la plantilla `index.html` desde la carpeta `templates`. Por lo tanto, en el contexto de una aplicación Flask, la ruta `home/myapp/templates/` se refiere a la ubicación en el sistema de archivos donde se almacenan las plantillas HTML de la aplicación.

**JENKINS**

**--Git**

CREAR REPO EN GIT y copiar el link en txt

Sacar user y pegar en txt

Sacar token y pegar en txt

git config --global user.name "Sample User"

git config --global user.email [sample@example.com](mailto:sample@example.com)

git init

git remote add origin [https://github.com/*github-username*/sample-app.git](https://github.com/github-username/sample-app.git)

git add \*

git status

git commit -m "Committing sample-app files."

git push origin master

**--Intalacion Jenkins**

docker pull jenkins/jenkins:lts

docker run --rm -u root -p 8080:8080 -v jenkins-data:/var/jenkins\_home -v $(which docker):/usr/bin/docker -v /var/run/docker.sock:/var/run/docker.sock -v "$HOME":/home --name jenkins\_server jenkins/jenkins:lts

**fijarse en esto “**77dc402e31324c1b917f230af7bfebf2 <--Your password will be different”

--recuperación contraseña (cuando ya fue instalado debo iniciar con admin y esta password)

**docker exec -it jenkins\_server /bin/bash**

**cat /var/jenkins\_home/secrets/initialAdminPassword**

**-EN JENKINS**

---BUILD

-new item

- poner nombre con build

-freestyle project

- general tab

- source code management : git

Credentials: add

Pongo el user y token de github

Add

* La casilla de credentials que dice “none” cambiarla a la creada recientemente.
* Build steps: execute Shell (pongo comandos para iniciar Docker o proyecto)

Docker build -t app4 .

Docker run -d -p 80:80 –name web app4

-save

-Dar a Build Now – para hacer funcionar

--TEST

- parar y eliminar el Docker creado (docker stop / Docker rm)

-new item

- name con TEST

- freestyle Project

-ok

- source code management : NONE

- build trigers: Build after other projects are built

projects to watch: nombrearchivoBUILD

* Build steps: execute Shell (pongo ip y puerto Docker y algo que este en el index)

if curl http://172.17.0.1:80/ | grep "ALGO DEL INDEX"; then

exit 0

else

exit 1

fi

--PIPELINE

-new item

- Nombre pipeline

- seleccionar pipeline

-ok

- al final cargar este código solamente (pipeline)

node {

stage('Preparation') {

catchError(buildResult: 'SUCCESS') {

sh 'docker stop aplication'

sh 'docker rm aplication'

}

}

stage('Build') {

build 'BUILT'

}

stage('Results') {

build 'TEST'

}

}

-SAVE

\*\*me permite poder levantar test y built a la vez para ver donde esta el error

EXPLICACIONES:

if curl http://172.17.0.1:5050/ | grep "You are calling me from 172.17.0.1"; then

exit 0

else

exit 1

fi

Este código es un script de shell que utiliza el comando `curl` para hacer una solicitud HTTP a `http://172.17.0.1:5050/` y luego utiliza `grep` para buscar una cadena específica en la respuesta recibida. Aquí está el desglose del código:

1. `curl http://172.17.0.1:5050/`: Este comando utiliza `curl` para realizar una solicitud HTTP GET a la URL `http://172.17.0.1:5050/`. Esto intentará obtener el contenido de esa URL.

2. `| grep "You are calling me from 172.17.0.1"`: El símbolo `|` se utiliza para redirigir la salida del comando anterior (`curl`) al comando siguiente (`grep`). `grep` es una herramienta que busca patrones en el texto. En este caso, busca la cadena "You are calling me from 172.17.0.1" en la salida de `curl`.

3. `then`: Esta es una palabra clave que indica el inicio de un bloque condicional. Indica que si el comando `grep` tiene éxito (es decir, encuentra la cadena deseada en la salida de `curl`), se ejecutará el código dentro del bloque `then`.

4. `exit 0`: Esto es una instrucción para salir del script con un código de salida 0, lo que generalmente indica que el script se ejecutó correctamente.

5. `else`: Esta es otra palabra clave que indica el inicio de un bloque condicional alternativo. Indica que si el comando `grep` no tiene éxito (no encuentra la cadena deseada en la salida de `curl`), se ejecutará el código dentro del bloque `else`.

6. `exit 1`: Esto es una instrucción para salir del script con un código de salida 1, lo que generalmente indica un error o un estado no exitoso.

En resumen, este script verifica si la respuesta de la solicitud HTTP a `http://172.17.0.1:5050/` contiene la cadena "You are calling me from 172.17.0.1". Si lo hace, el script sale con un código de salida 0, lo que indica éxito. Si no lo hace, el script sale con un código de salida 1, lo que indica un fallo.

------------------------------------------------------------------------------------------------------------

node {

stage('Preparation') {

catchError(buildResult: 'SUCCESS') {

sh 'docker stop aplication'

sh 'docker rm aplication'

}

}

stage('Build') {

build 'BUILT'

}

stage('Results') {

build 'TEST'

}

}

Este fragmento de código parece ser un pipeline de Jenkins escrito en Groovy. Aquí está la explicación:

1. `node { ... }`: Esto indica que todo el código dentro del bloque `node` se ejecutará en un agente de Jenkins. Un agente es una máquina esclava que realiza el trabajo definido en el pipeline.

2. `stage('Preparation') { ... }`: Esto define una etapa en el pipeline llamada "Preparation" (Preparación). Las etapas son divisiones lógicas del pipeline que ayudan a organizar y visualizar el flujo de trabajo.

3. `catchError(buildResult: 'SUCCESS') { ... }`: Esto envuelve un bloque de código y captura cualquier error que ocurra dentro de él. Si se produce un error, el resultado de la construcción se establecerá en "SUCCESS" para que Jenkins no lo marque como un fallo completo del pipeline.

4. `sh 'docker stop samplerunning'`: Este comando utiliza el paso de shell (`sh`) para detener un contenedor Docker llamado "samplerunning". Esto detendrá la ejecución del contenedor Docker.

5. `sh 'docker rm samplerunning'`: Este comando utiliza el paso de shell (`sh`) para eliminar el contenedor Docker llamado "samplerunning". Esto eliminará completamente el contenedor Docker después de detenerlo.

En resumen, este fragmento de código de pipeline de Jenkins detiene y elimina un contenedor Docker llamado "samplerunning" como parte de la etapa de "Preparation". Si hay algún error durante este proceso, Jenkins registrará la construcción como un éxito (`SUCCESS`) en lugar de un fracaso.