Introducción

Para la solución del caso, se requirió desarrollar dos aplicaciones: una para el agente quien recopila la información del sistema y la envía a la API; y otra para la API, que es quien recibe la información y la almacena en un archivo CSV y un JSON.
Ambas aplicaciones fueron desarrolladas en el lenguaje de programación Python.

Agente

Será quien recopile la información del sistema. Para ello, se utilizaron librerías de Python que permiten acceder a la misma.

Supuestos

- ☐ El agente se ejecutará en una máquina virtual con Linux.
- ☐ Se deberá de disponer de conexión a internet para poder enviar la información a la API.

Problemas

- ☐ Será necesario determinar cómo enviar la información desde el agente a la API.

Soluciones

- ☐ Se utilizará una solicitud HTTP POST para enviar la información del sistema a la API.

Desarrollo

El agente utilizará las siguientes librerías:

- ☐ **“request”** para realizar solicitudes HTTP a la API.
- ☐ **“platform” p**ara obtener información del sistema operativo comoe le procesador , nombre del SO y version.
- ☐ **“psutils”** para obtener información del sistema, como procesos y usuarios.

API

Será quien escuche solicitudes HTTP en un endpoint especifico y almacene la información recibida en un archivo CSV y JSON con el formato <IP de servidor>_<AAAA-MM-DD>. Para ello se utilizarán las librerías que permitan el manejo de solicitudes HTTP y archivos CSV y JSON.

Supuestos

- ☐ La API se ejecutará en una instancia de AWS EC2.
- ☐ Deberá disponer de conexión a internet para recibir solicitudes del agente.
- ☐ Se considera que Docker ya se encuentra instalado.

Problemas

- ☐ Será necesario determinar cómo manejar las solicitudes HTTP y extraer la información del cuerpo de la solicitud..
- ☐ Es necesario establecer como almacenar la información en un archivo CSV.

Soluciones

- ☐ Se utilizará el paquete **request** para manejar las solicitudes entrantes y un handler para el endpoint **/data**.

Desarrollo

La API utilizará las siguientes librerías:

- ☐ **“flask”** para crear y ejecutar la aplicación Flask y manejar las rutas y solicitudes HTTP entrantes.
- ☐ **“request”** para acceder a los datos de las solicitudes HTTP entrantes.
- ☐ **“csv”** para el manejo de archivos CSV
- ☐ **“json”** para guardar la info en un archivo json.
- ☐ **“time”**para obtener la fecha actual.

Instrucciones para la Ejecución

Requisitos previos:

- ☐ Tener instalado Python en la maquina Linux.
- ☐ Configurar el entorno de desarrollo con la documentación oficial de Python.

Repositorios:
https://github.com/marianovanini/challenge-meli-seg-py

Para la **API**:

- Considerando que la instancia de AWS EC2 ya fue creada junto con la configuración previa de Key Pair y su posterior conexión por SSH:
  - Abrir la terminal por SSH según la documentación de AWS
  - Instalar Python en caso de no contar con el.
  - Dirigirse al directorio donde se quiera clonar el repositorio
  - Clonar el repositorio con **git clone**
  - Dentro del directorio, lanzar los siguientes comandos:
    - `$ docker build -t flaskapp .` esto creara la imagen con el nombre flaskapp(modificable a criterio) siguiendo las descripciones detalladas en el DOCKERFILE.
    - `$ docker run -it -p 7000:8080 -d flaskapp` de esta manera se correrá la imagen recientemente creada, dentro del contenedor correrá en el puerto 8080 y 7000 será el puerto definido de forma publica, utilizaremos -d (--detach) para que el contenedor quede corriendo como un proceso
    - `dfa6d8d469ba6988b619ccf478f3d98f74d037d9f48e6eb58895e7d5efe10261` obtendremos un ID del proceso del container corriendo.
  - Así la API quedará escuchando en el puerto,en este caso se utilizó el **7000**.
- El endpoint definido para la API es **/data** el cual acepta solicitudes HTTP POST para recibir la información del agente.
- Para enviar una solicitud al endpoint **/data** se debe hacer una solicitud HTTP **POST** a la **dirección IP** de la instancia de AWS EC2, en el puerto **7000**. A continuación se deja un body a modo de ejemplo:

```
{
  "hostname": "my-server",
  "processor": "Intel Core i7",
  "processes": "PID PPID USER %CPU %MEM CMD\n123 1 user 0.0 0.1 process1\n456
1 user 0.1 0.2 process2",
  "users": "user1\nuser2",
  "os_name": "Linux",
  "os_ver": "4.19.0-14-amd64"
}
```

Para el **Agente**:

- Clonar el repositorio, y acceder desde VS Code o similar
- Modificar la URL con la URL de la instancia de la API en EC2, siendo el cambio de forma similar al siguiente ejemplo:
  - ```
    url = 'http://<localhost>:7000/data'
    ```
  - de forma tal que quede la URL de la instacia
  - ```
    url = 'http://<URL_EC2>:7000/data'
    ```
  - mantener el puerto 7000 ya que tal puerto fue asignado de manera publica en la creación de la imagen de Docker de la API.(en caso de querer modificarlo, realizarlo primero en la creación del docker de la API)
- Abrir una terminal en el directorio donde se tenga el repositorio.
- Ejecutar el comando para enviar la informacion a la API:
  - ```
    python src/app.py
    ```
- Finalmente verificar la respuesta:
  - `Información enviada exitosamente` Así el agente recopilará la información y la enviará a la API cada vez que se ejecute tal comando.
  - `Ocurrió un error de conexión: HTTPConnectionPool(host='localhost', port=8000):` este error ocurrirá en caso de haber modificado el puerto, (en el ejemplo a 8000 y corriendo de manera local)

Verificar información enviada a la API:

- Desde la consola de EC2 si ejecutamos:
  - `docker container exec -it elastic_williams /bin/sh` nos dejará el promp listo para realizar las tareas que consideremos necesarias (elastic_williams es el nombre que tomó nuestro contenedor dentro de la instancia de EC2).
  - `/app # ls` de esta manera listaremos los archivos dentro del directorio /app
  - ```
    172.17.0.1_2023-06-01.csv   172.17.0.1_2023-06-01.json  Docker
    file                        requirements.txt            src
    ```
    finalmente podemos chequear que se crearon ambos archivos con la información del agente.

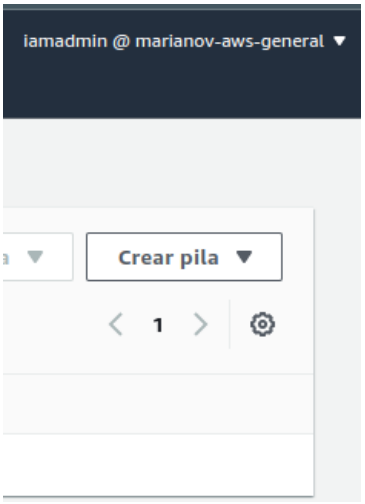
Código de CloudFormation

Se adjunta el mismo en formato .yaml . En el mismo se definió dentro de las propiedades de la instancia la siguiente:

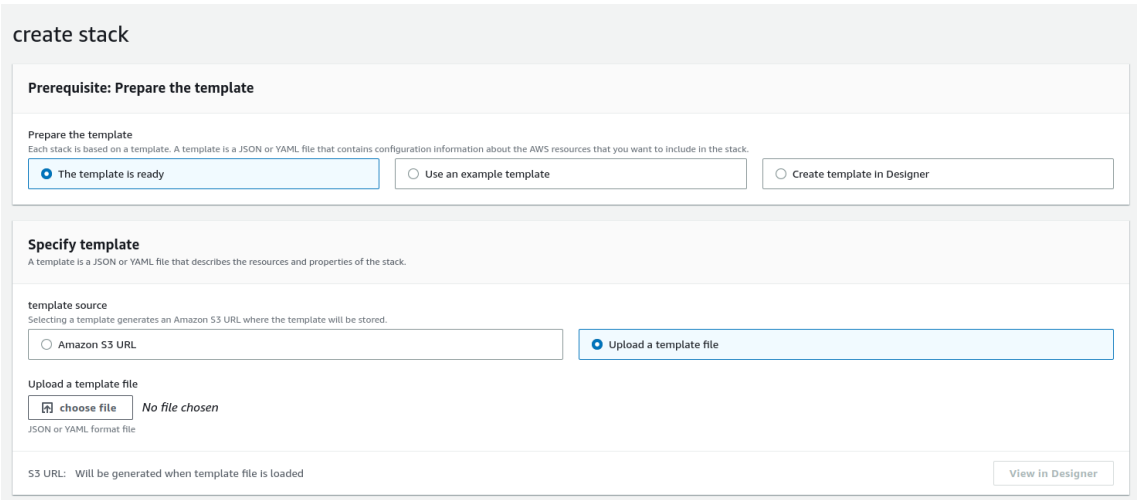
KeyName: **MyKeyPair**

Tal nombre debe coincidir con el nombre de la clave creada desde la pestaña “EC2 → Red y Seguridad → Pares de claves” para que posteriormente no existan problemas en la conexión por SSH.

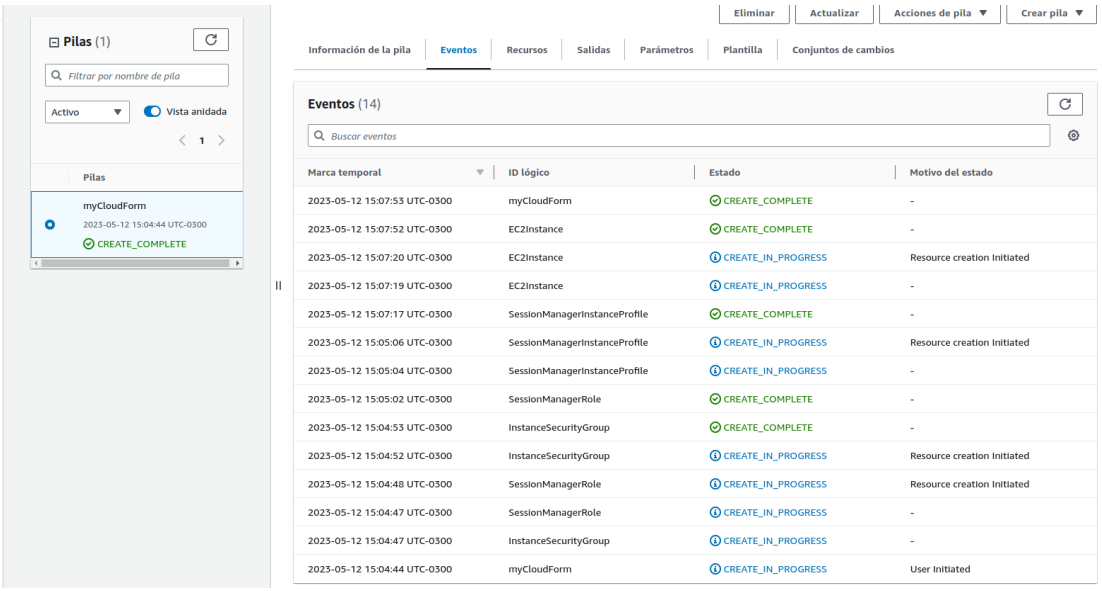
Pasos para levantar la infraestructura:

1. Loguearse en la cuenta de AWS
2. Dirigirse a CloudFormation
3. Seleccionar Create Stack
   a. 

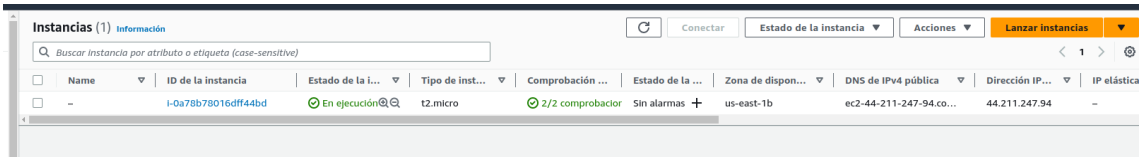
4. Subir archivo adjunto como “template file”



5. Darle un nombre al Stack y luego en **Submit** y esperar a que termine el proceso. Se brinda captura del Stack creado.



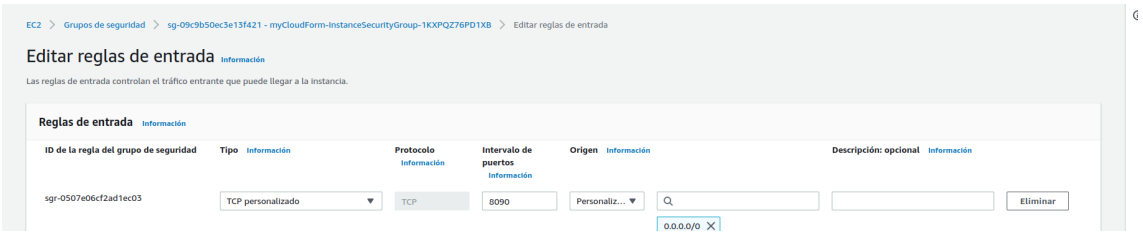
6. Una vez finalizado el proceso, dirigirse a **EC2** y verificar que la instancia esté en **ejecución.**



7. Conectarse a la misma por SSH desde la maquina con Linux siguiendo los pasos indicados en la pestaña “SSH cliente”.

Security Group

Con la instancia ya levantada, se deben editar las reglas de entrada del SecurityGroup y añadir un TCP personalizado con el puerto **7000**, permitiendo todas las redes. Para ello dirigirse a EC2 → Grupos de Seguridad → <seleccionar_SG> → Editar reglas de entrada



De esta manera se le permite a la API, que se le pueda relevar información a través de dicho puerto.

Docker

Se incluye en la API un Dockerfile para la creación de la imagenes y su posterior dockerización. A continuacion se muestra la construccion del mismo

```
FROM alpine:3.10

#añado python3 y pip al alpine
RUN apk add --no-cache python3-dev \
   && pip3 install --upgrade pip

#defino directorio de trabajo
WORKDIR /app

COPY . /app

RUN pip3 install --no-cache-dir -r requirements.txt

CMD ["python3", "src/app.py"]
```

1. **FROM alpine:3.10**: Especifica la imagen base que se utilizará para construir el contenedor. En este caso, se utiliza la imagen de Alpine Linux versión 3.10.
2. **RUN apk add --no-cache python3-dev** : Ejecuta un comando en el contenedor durante el proceso de construcción. En este caso, se está instalando python3-dev en el contenedor Alpine para proporcionar soporte para Python 3.
3. **&& pip3 install --upgrade pip**: Ejecuta otro comando para actualizar pip, el gestor de paquetes de Python, en el contenedor.
4. **WORKDIR /app**: Establece el directorio de trabajo dentro del contenedor como /app.
5. **COPY . /app**: Copia todos los archivos y directorios del directorio actual (donde se encuentra el Dockerfile) al directorio de trabajo del contenedor (/app).
6. **RUN pip3 install --no-cache-dir -r requirements.txt**: Ejecutado para instalar las dependencias del proyecto. Lee el archivo requirements.txt y utiliza pip3 para instalar las bibliotecas y paquetes especificados.
7. **CMD ["python3", "src/app.py"]:** Establece el comando predeterminado a ejecutar cuando se inicie el contenedor. En este caso, se ejecuta el archivo app.py ubicado en la carpeta src utilizando el intérprete de Python 3.

Este Dockerfile se utiliza para construir una imagen de Docker que incluye Python 3, instala las dependencias necesarias y configura el comando para ejecutar la aplicación Flask del agente.

Funcionalidad a implementar a futuro

Tomando como base el código de cloud formation se podría incorporar la creación de un S3 para poder almacenar la información en un bucket y en otra instancia poder acceder a través de un endpoint a dichos archivos. Para lo cual se puede utilizar el cdk de AWS para Python. A su vez, la creación de una base de datos permitirá guardar la url de donde se almacenaría dicho archivo y de esa manera acceder a través de la misma.