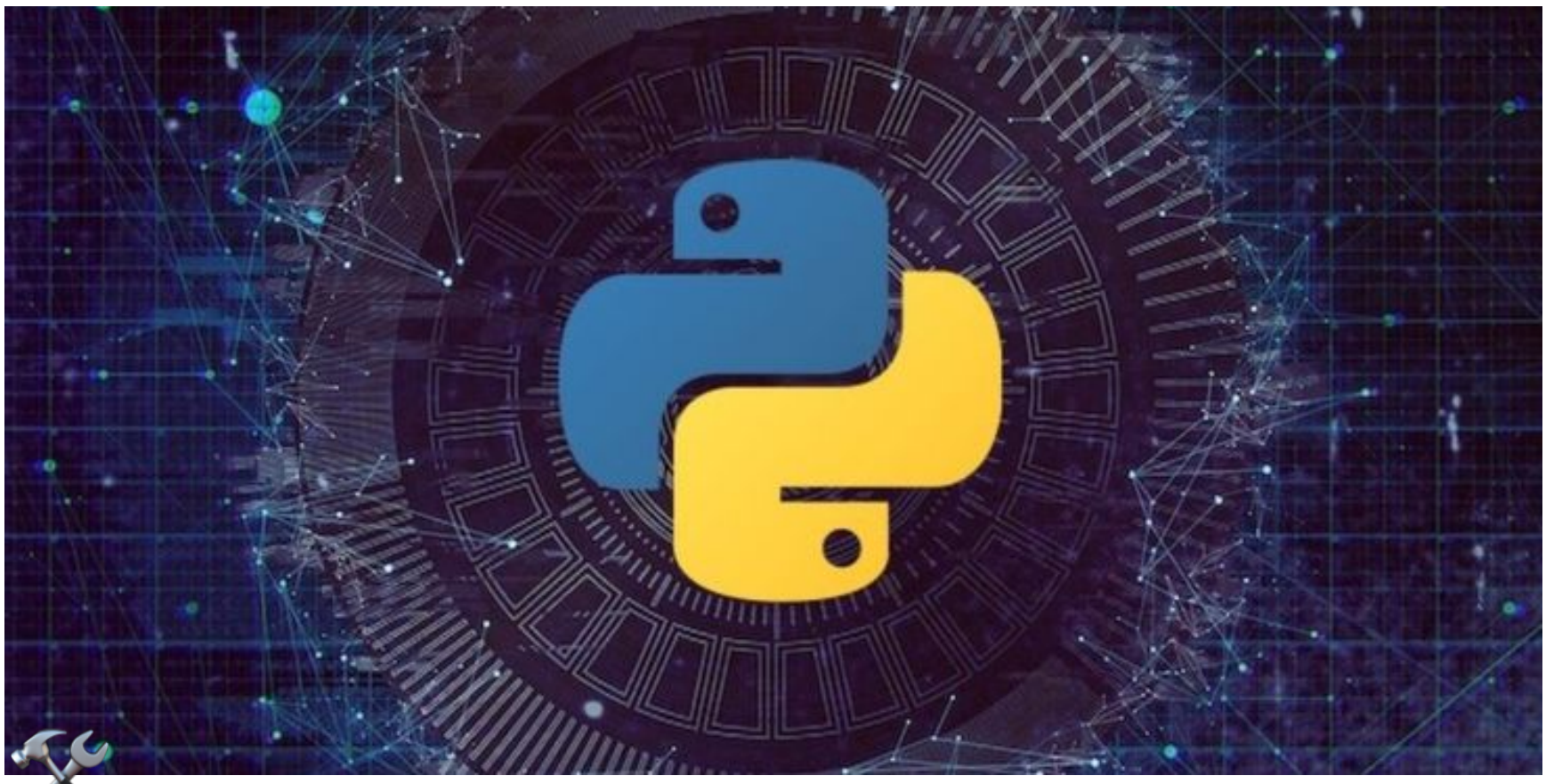

# Automatización de Backups

## 📌 Descripción General

Este proyecto permite automatizar el respaldo de la configuración de un dispositivo de red (SRX) mediante SSH. El script guarda el archivo de configuración localmente y lo transfiere de manera segura a un servidor SFTP utilizando autenticación basada en claves. Posteriormente, elimina el archivo local para mantener el entorno limpio y organizado.

Finalmente, esta automatización fue configurada con tareas programadas (`cron`) para ejecutarse de forma diaria, asegurando que los respaldos se realicen automáticamente sin intervención manual.

---

**Puedes acceder al código completo en mi GitHub:**

👉 https://github.com/pedroecheverria/SRX-Backup-Automation-.git

---

## 📄 Requisitos Principales

1. **Para el Equipo de Red (SRX):**
   - Usuario y contraseña creados en el dispositivo.
   - Permisos suficientes para ejecutar el comando de respaldo (`show configuration`).
2. **Para el Servidor SFTP:**
   - Usuario y clave pública configurados para autenticación.
   - Directorio remoto accesible donde se almacenarán los respaldos.
   - Permisos para crear y modificar archivos/directorios.
3. **Para el Entorno Local:**
   - Python instalado (versión 3.x).
   - Librerías necesarias:
     - `paramiko`
     - `dotenv`

- Acceso al sistema de archivos para guardar temporalmente los respaldos.

4. **Para Automatización (Cron Job):**
   - Acceso al sistema operativo para configurar tareas programadas (cron en Linux).
   - Un script Bash que ejecute el script Python.
5. **Opcional:**
   - Correo SMTP configurado (si se implementa la notificación por correo).
   - Un servidor de logs centralizado o local para registrar eventos del script.

## 📚 Estructura Recomendada del Proyecto

Antes de empezar, es importante describir cómo organizar el proyecto tanto para la **fase de pruebas** como para **producción.**

### 1. Fase de Pruebas

Durante el desarrollo, se prioriza la simplicidad para facilitar los ajustes y la depuración. Todos los archivos necesarios pueden mantenerse en el mismo directorio.

**Estructura recomendada para pruebas:**

```
/mi_proyecto/
    ├── script.py           # Código principal del proyecto
    ├── .env                # Archivo con credenciales y configuraciones
    ├── private_key.pem     # Clave privada para conexión SFTP
```

### 2. Producción

En un entorno de producción, la seguridad y la organización son prioridades. Los archivos sensibles deben estar separados del código y protegidos con permisos adecuados.

**Estructura recomendada para producción:**

```
/opt/mi_aplicacion/
    ├── script.py           # Código principal del proyecto
/etc/mi_aplicacion/
    ├── .env                # Archivo con credenciales y configuraciones
    ├── private_key.pem     # Clave privada para conexión SFTP
```

#### 💡 Observación

**Separación de responsabilidades:** Los archivos sensibles (`.env` y `private_key.pem`) se almacenan fuera del directorio del código.

**Seguridad:** Configurar permisos restrictivos para los archivos sensibles:

```
sudo chmod 600 /etc/mi_aplicacion/.env
sudo chmod 600 /etc/mi_aplicacion/private_key.pem
```

---

## ⌨️ Código Python del Proyecto

### PASO 1 : Creación del Archivo `.env` para Almacenar Variables Sensibles.

En un entorno de pruebas, puedes usar VS Code o cualquier otro ambiente, donde creas este archivo de texto simple que contiene pares clave-valor para almacenar configuraciones y variables sensibles (como credenciales). Este archivo no debe incluirse en repositorios públicos ni compartirse, ya que contiene información confidencial.

¿Por qué usar un archivo `.env`?

1. **Separación de configuración y código:** Mantener las credenciales fuera del código hace que este sea más seguro y fácil de compartir o migrar.
2. **Facilidad de configuración:** Puedes cambiar credenciales o configuraciones simplemente editando el archivo, sin necesidad de modificar el código.

Aquí te dejo un ejemplo de como seria un archivo `.env` tomando como ejemplo un dispositivo SRX:

```
# Credenciales del dispositivo SRX
SRX_IP=IP_Equipo
SRX_USER=admin
SRX_PASS=contraseña_secreta

# Credenciales del servidor SFTP
SFTP_HOST=sftp.example.com
SFTP_USER=sftp_user
SFTP_KEY_PATH=/ruta/a/clave_privada.pem

# Configuración del servidor SMTP para enviar correos (OPCIONAL)
SMTP_SERVER=smtp.example.com
SMTP_PORT=587
EMAIL_USER=tu_correo@example.com
EMAIL_PASS=contraseña_email
EMAIL_TO=destinatario@example.com
```

> 💡 **Observación**
>
> Cuando pases tu script a un servidor de producción, guarda el archivo `.env` en una ubicación segura y con permisos restringidos. Esto ayuda a proteger tus credenciales de accesos no autorizados.

### Ruta recomendada en Linux:

Guarda el archivo `.env` fuera del directorio público o de acceso al código, por ejemplo:

- `/etc/tu_aplicacion/.env`
- `/opt/tu_aplicacion/.env`

En este caso, `tu_aplicacion` es un nombre genérico que puedes personalizar según el nombre de tu proyecto o aplicación. Sirve para organizar los archivos de configuración de manera estructurada y específica dentro del sistema operativo. Por ejemplo, si tu proyecto se llama "BackupSRX", la ruta final podría ser `/etc/BackupSRX/.env`.

## PASO 2: Importación de Librerías

En Python, usamos librerías para realizar tareas específicas sin necesidad de programarlas desde cero. Por ejemplo, en este script:

```
import paramiko
from dotenv import load_dotenv
import os
from datetime import datetime
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
```

- `paramiko`: Biblioteca de Python para conectarse a dispositivos y servidores mediante SSH o SFTP.

- `dotenv`: Permite cargar variables sensibles (como credenciales) desde un archivo `.env` de forma segura.
- `os`: Se utiliza para acceder al sistema operativo, como rutas de archivos.
- `datetime`: Para obtener la fecha y generar nombres únicos para los backups.
- `smtplib` **y** `email`: Para enviar correos electrónicos con notificaciones.

## PASO 3: Carga del Archivo `.env` y Definición de Variables para Credenciales

Este paso permite al programa obtener las credenciales necesarias (como IPs, usuarios y contraseñas) de un archivo externo `.env`

Las variables se llaman utilizando la función `os.getenv("NOMBRE_DE_LA_VARIABLE")`. Esto busca la variable en el archivo `.env` (que ya fue cargado con `load_dotenv("ruta_de_tu_archivo")`), o en las variables de entorno del sistema.

Por ejemplo:

```
SRX_IP = os.getenv("SRX_IP")
```

- `SRX_IP`: Es la variable en Python donde se almacena el valor.
- `"SRX_IP"`: Es el nombre exacto de la variable definida en el archivo `.env`.

Aquí se muestra como se cargaron cada una de las variables en el código:

```
from dotenv import load_dotenv
import os

# Cargar las variables desde el archivo .env
load_dotenv('ruta_de_tu_archivo')

# Credenciales del dispositivo SRX
SRX_IP = os.getenv("SRX_IP")
SRX_USER = os.getenv("SRX_USER")
SRX_PASS = os.getenv("SRX_PASS")

# Credenciales del servidor SFTP
SFTP_HOST = os.getenv("SFTP_HOST")
SFTP_USER = os.getenv("SFTP_USER")
SFTP_KEY_PATH = os.getenv("SFTP_KEY_PATH")

# Configuración para el servidor de correo
SMTP_SERVER = os.getenv("SMTP_SERVER")
SMTP_PORT = int(os.getenv("SMTP_PORT"))
EMAIL_USER = os.getenv("EMAIL_USER")
EMAIL_PASS = os.getenv("EMAIL_PASS")
EMAIL_TO = os.getenv("EMAIL_TO")
```

> 💡 **Observación**
>
> ### 🔐 En producción:
>
> La clave privada debe almacenarse en un directorio seguro, como `/etc/mi_aplicacion/`, con permisos restrictivos que solo permitan acceso al usuario que ejecuta el script.
>
> ### 🛠️ En pruebas:
>
> Para facilitar el desarrollo, tanto el archivo `.env` como la clave privada pueden mantenerse en el mismo directorio que el script. Esto simplifica el manejo de rutas y evita configuraciones adicionales durante la fase inicial del proyecto.

# Paso 4: Establecer la Conexión SSH y Ejecutar el Comando en el Dispositivo SRX

En este paso, el script se conecta al dispositivo SRX utilizando SSH mediante la librería `paramiko`. Luego, ejecuta un comando para extraer su configuración y la guarda como texto en la variable `config`.

## Explicación Detallada de Cada Parte del Código

Aquí desglosamos cada línea del código del paso 4 y explicamos cómo funciona.

## 1. Configuración inicial y mensaje de depuración

```
print("Directorio de trabajo actual:", os.getcwd())
```

- **¿Qué hace?**
  - Muestra el directorio desde donde se ejecuta el script. Esto es útil para asegurarte de que el programa esté buscando archivos en el lugar correcto.
- **Función usada:**
  - `os.getcwd()`: Retorna la ruta del directorio de trabajo actual.

## 2. Crear el cliente SSH

```
ssh = paramiko.SSHClient()
```

- **¿Qué hace?**
  - Crea un objeto cliente que se usará para establecer una conexión SSH con el dispositivo SRX.
- **Librería usada:**
  - `paramiko.SSHClient()`: Parte de la librería `paramiko`, esta clase maneja la comunicación SSH.

## 3. Configurar política para aceptar claves no conocidas

```
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
```

- **¿Qué hace?**
  - Configura el cliente SSH para aceptar automáticamente claves del host remoto que no están en la lista de claves conocidas (`known_hosts`).
- **Política usada:**
  - `paramiko.AutoAddPolicy()`: Permite que el cliente SSH confíe en hosts nuevos automáticamente. **Nota:** En producción, es mejor usar una política más estricta.

## 4. Conectar al dispositivo SRX

```
ssh.connect(SRX_IP, username=SRX_USER, password=SRX_PASS)
```

- **¿Qué hace?**
  - Establece una conexión SSH al dispositivo usando las credenciales cargadas desde el archivo `.env`.
- **Parámetros:**
  - `SRX_IP`: Dirección IP del dispositivo.
  - `SRX_USER`: Usuario para la conexión.
  - `SRX_PASS`: Contraseña del usuario.

## 5. Ejecutar el comando remoto

```
stdin, stdout, stderr = ssh.exec_command(command)
```

- **¿Qué hace?**

- Envía el comando especificado al dispositivo SRX y ejecuta el proceso en el host remoto.

- **Parámetros:**
  - `command`: Contiene el comando que queremos ejecutar:

    Este comando:

    ```
    command = "show configuration | display set | no-more"
    ```

    - Muestra la configuración del dispositivo.
    - La formatea como un conjunto de comandos (`display set`).
    - Evita paginación (`no-more`).
- **Salida:**
  - `stdin`: Se usa para enviar datos al comando en el dispositivo remoto (no se usa aquí).
  - `stdout`: Contiene la salida estándar del comando (lo que normalmente verías en pantalla si ejecutaras el comando manualmente).
  - `stderr`: Contiene la salida de errores (si el comando falló o generó advertencias).

### 6. Leer la salida del comando

```
config = stdout.read().decode()
```

- **¿Qué hace?**
  - Lee todo el contenido de la salida (`stdout`) y lo convierte en texto legible.
- **Explicación técnica:**
  - `stdout.read()`: Obtiene la salida como datos binarios.
  - `.decode()`: Convierte esos datos binarios en texto, usando la codificación predeterminada (`UTF-8`).

### 7. Imprimir la configuración

```
print("=== Configuración del SRX ===")
print(config)
```

- **¿Qué hace?**
  - Muestra la configuración extraída en la terminal para confirmar que el comando se ejecutó correctamente.

### 8. Cerrar la conexión SSH

```
ssh.close()
```

- **¿Qué hace?**
  - Termina la conexión SSH con el dispositivo para liberar recursos.

## Paso 5: Guardar la Configuración Extraída en un Archivo Local

En este paso, el script toma la configuración obtenida del dispositivo SRX (paso 4) y la guarda en un archivo local. Esto es útil para realizar respaldos y tener un registro de las configuraciones a lo largo del tiempo.

- **Nombre del archivo:** Basado en la fecha actual, como `SRX_backup_DD_MM_AAAA.txt`.
- **Ubicación del archivo:** La ruta completa se genera automáticamente para evitar confusiones.

```
# Obtener la fecha actual para el nombre del archivo
current_date = datetime.now().strftime("%d_%m_%Y")
backup_filename = f"SRX_backup_{current_date}.txt"
backup_filepath = os.path.abspath(backup_filename)
```

```
# Guardar la configuración en un archivo
with open(backup_filepath, "w") as backup_file:
    backup_file.write(config)

print(f"Backup guardado en {backup_filepath}")
```

## Paso 6: Enviar el Archivo al Servidor SFTP

En este paso, el script transfiere el archivo de configuración recién guardado a un servidor SFTP. Esto permite almacenar el respaldo en una ubicación centralizada y segura, útil para cumplir con políticas de backup y recuperación de desastres.

### Explicación Detallada de Cada Parte del Código

Aquí desglosamos cada línea del código del paso 5 y explicamos cómo funciona.

### 1. Verificar el Archivo Local

```
if not os.path.exists(local_filepath):
    print("Error: El archivo local no existe.")
    return
```

- **¿Qué hace?**

  Confirma que el archivo que queremos transferir existe con el fin de evitar intentar subir un archivo inexistente, lo que generaría errores.

### 2. Cargar la Clave Privada

```
private_key = paramiko.RSAKey.from_private_key_file(SFTP_KEY_PATH)
```

- **¿Qué hace?**

  Necesario para autenticar la conexión al servidor SFTP.

### 3. Establecer Conexión SFTP

```
transport = paramiko.Transport((SFTP_HOST, 22))
transport.connect(username=SFTP_USER, pkey=private_key)
sftp = paramiko.SFTPClient.from_transport(transport)
```

- **¿Qué hace?**

  Inicia la conexión segura con el servidor SFTP utilizando la clave privada.

### 4. Verificar o Crear la Carpeta Remota

```
try:
    sftp.chdir(remote_dir)
except IOError:
    sftp.mkdir(remote_dir)
    sftp.chdir(remote_dir)
```

- **¿Qué hace?**

  Asegura que el directorio remoto donde se almacenará el archivo exista. Si no, lo crea.

### 5. Subir el Archivo

```
remote_filepath = f"{remote_dir}/{os.path.basename(local_filepath)}"
sftp.put(local_filepath.replace("\\", "/"), remote_filepath)
```

- **¿Qué hace?**

Envía el archivo desde el sistema local al servidor SFTP.

### 6. Cerrar la Conexión SFTP

```
sftp.close()
transport.close()
```

- **¿Qué hace?**

  Libera los recursos asociados a la conexión.

### 7. Eliminar el Archivo Local

```
os.remove(local_filepath)
print(f"Archivo local eliminado: {local_filepath}")
```

- **¿Qué hace?**

  Elimina el archivo local después de haber sido enviado exitosamente con el fin de mantener el entorno limpio, y evita la acumulación de respaldos en el servidor local.

---

## ⏰ Código Bash para Configurar el Cron Job

## Funcionalidad de `cron`

El `cron` es una herramienta en sistemas Linux que permite programar tareas para que se ejecuten automáticamente en momentos específicos. En este caso, configuraremos el script para que se ejecute todos los días a las 2:00 a.m.

### Paso 1: Crear un Script Bash que Ejecute el Código Python

1. Crea un archivo llamado `ejecutar_respaldo.sh`:

```
#!/bin/bash

# Activar el entorno virtual (si usas uno)
source /ruta/a/tu/entorno/virtual/bin/activate

# Ejecutar el script Python
python3 /ruta/a/tu/script.py
```

1. Haz el archivo ejecutable:

```
chmod +x ejecutar_respaldo.sh
```

### Paso 2: Configurar el Cron Job

1. Abre el editor de tareas programadas con:

```
crontab -e
```

1. Agrega la siguiente línea al final del archivo para programar la tarea a las 2:00 a.m. todos los días:

```
0 2 * * * /ruta/a/ejecutar_respaldo.sh >> /ruta/a/tu/logs/backup.log 2>&1
```

**Explicación de la línea:**

- `0 2 * * *`**:** Indica la hora de ejecución (2:00 a.m. todos los días).
- `/ruta/a/ejecutar_respaldo.sh`**:** La ruta al script Bash que ejecuta el código Python.

- `>> /ruta/a/tu/logs/backup.log`: Guarda la salida estándar del script en un archivo de log para seguimiento.
- `2>&1`: Redirige los errores a la misma salida estándar para registrarlos en el log.

---

## ⚙️ Puntos de Mejora

1. **Enviar Notificación por Correo:**
   - Añadir funcionalidad para enviar un correo con el estado del respaldo (exitoso o fallido) al equipo encargado.
2. **Vaciar el Servidor SFTP Cada 60 Días:**
   - Implementar un script que elimine respaldos antiguos en el servidor SFTP para evitar acumulación y mantener espacio disponible.
3. **Logs Avanzados:**
   - Centralizar los logs en un archivo rotativo para gestionar mejor el historial y evitar archivos de log demasiado grandes.