

ANEXO A

Nota Técnica

Diseña de una aplicación web para la automatización de configuraciones de administración de redes de datos en un ISP

Introducción

El objetivo de esta aplicación es migrar configuraciones entre dispositivos intermediarios de un proveedor de servicio de internet con sistemas operativos Cisco IOS/IOS XE a Cisco IOS XR o Huawei AR debido a que los protocolos de enrutamiento de los sistemas operativos más actuales han sido optimizados por los fabricantes.

Se construye una aplicación web basada en microservicios, el Front-End está basado en el entorno de trabajo de Angular con alojamiento y base de datos en el servicio en la nube de Firebase, por otro lado, el Back-End está desarrollado en Python, el cual puede alojarse localmente con Docker o con el servicio en la nube de Cloud Run.

Materiales

- Sistema operativo de Linux / Sistema operativo virtual de Linux
- Visual Studio Code
- GNS3
- Docker
- Repositorio de proyecto: <https://github.com/ESPOL-NETMI>

Procedimiento

Paso 1: Creación de programa base de Back-End

- a) Instalación de WSL2 para sistemas operativo de Windows 10, omitir este ítem en caso de tener sistema operativo Linux o MacOS para el desarrollo del proyecto.

cmd	dism.exe /online/enable-feature/featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
	dism.exe /online/enable-feature/featurename:VirtualMachinePlatform/all /norestart
	<i>wsl –set-default-version 2.</i>

Tabla 1 comandos de instalación de wsl

Instalar versión de Linux de la tienda de Microsoft, se recomienda instalar Ubuntu 18.04 LTS.



Ilustración 1 instalación de versión de WSL

- b) Habilitar WSL2 en Visual Studio Code para Windows 10 instalando la extensión Remote – WSL

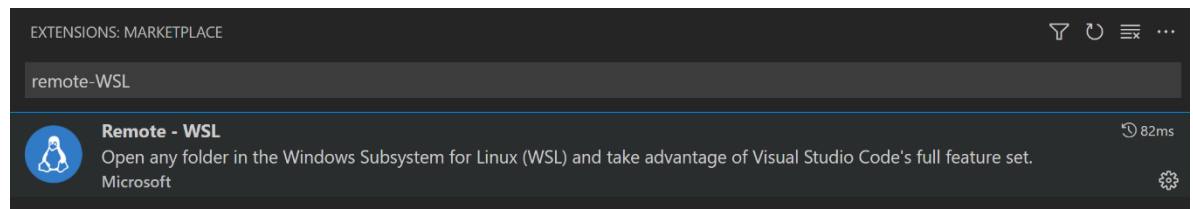



Ilustración 2 Instalación de WSL remoto para VSC

- c) Crear la carpeta base del proyecto de manera local, luego abrir la carpeta en VS, por siguiente dar clic en el ícono  de abrir una ventana remota y por último reabrir la carpeta en modo WSL.

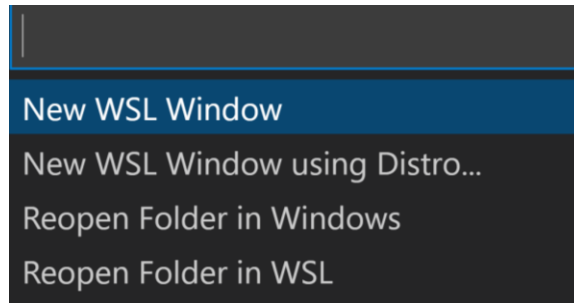


Ilustración 3 Acceso a ambiente de WSL en VSC

- d) En la carpeta crear los siguientes archivos y carpeta:
- El archivo main.py donde se declararán los microservicios.
 - El archivo gpapi.py donde se declara funciones para conexión con los dispositivos.
 - El archivo genipyats.txt donde se declararán las librerías de cisco con sus versiones.
 - El archivo microservice.txt donde se declararán las librerías de microservicios y de plantillas con sus versiones.
 - La carpeta plantillas contiene los archivos base para cada método de migración.
- e) Declarar en genipyats.txt las siguientes librerías.
- pyats[full] == 21.8
 - genie == 21.8
- f) Declarar en microservice.txt las siguientes librerías.
- pyyaml == 5.4.1
 - Jinja2==3.0.1
 - requests==2.26
 - flask == 2.0.1
 - flask-cors == 3.0.10

g) Crear un entorno virtual de Python e instalar las librerías.

Bash	apt-get install python3-venv
	pip install --upgrade pip
	python3 -m venv envnetmi
	source envnetmi/bin/actívale
	python3 -m pip install --upgrade setuptools
	python3 -m pip install -r geniepyats.txt
	python3 -m pip install -r microservice.txt

Tabla 2 Instalación de ambiente de trabajo y librerías

Paso 2: Desarrollo de Back-End

Se realiza la creación de funciones en gpapi.py tal para recibir los parámetros de autenticación con dispositivos intermediarios y construcción de un diccionario para la conexión remota, así mismo, obtener las configuraciones y cerrar la conexión con el dispositivo.

Los parámetros para autenticación con los dispositivos intermediarios son los siguientes:

- Nombre del equipo.
- Dirección IP.
- nombre de usuario.
- Contraseña.
- sistema operativo.
- contraseña de privilegios.
- puerto (si es requerido).
- Encriptación.

```

gapi.py
1  from genie import testbed
2
3  class gpapi(object):
4      def __init__(self, json_data=None):
5
6          name = json_data["name"]
7          ip = json_data["ip"]
8          protocol = json_data["protocol"]
9          password = json_data["password"]
10         username = json_data["username"]
11         os = json_data["os"]
12         enable = json_data["enable"]
13         port = json_data["port"]
14         encriptacion = json_data["encryp"]
15         if(protocol=="ssh"):
16             protocol=protocol+" -c "+encriptacion
17         testbeddict={"devices":{
18             name:{
19                 "connections":{
20                     "cli":{
21                         "ip":ip,
22                         "protocol": protocol,
23                         "port":port
24                     }
25                 },
26                 "credentials":{
27                     "default":{
28                         "username": username,
29                         "password": password
30                     }, "enable":{
31                         "password": enable
32                     }
33                 },
34                 "os": os
35             }
36         }}
37         test = testbed.load(testbeddict)
38         self.device = test.devices[name]
39
40     def connect(self):
41         self.device.connect(init_exec_commands=[], init_config_commands=[], log_stdout=False)
42
43     def disconnect(self):
44         self.device.disconnect()
45
46     def showconfig(self, show=None):
47         self.connect()
48         parse = self.device.parse(show)
49         return parse

```

Ilustración 4 Funciones para acceso a dispositivos intermediarios

El formato del diccionario de autenticación es representado como en el siguiente ejemplo:

```
! testbed.yaml
1  devices:
2    GYE_PRIMAX:
3      connections:
4        cli:
5          ip: 172.25.192.101
6          port: 443
7          protocol: ssh -c aes128-cbc
8      credentials:
9        default:
10         password: cisco
11         username: admin
12         enable:
13           password: cisco123
14     os: iosxe
```

Ilustración 5 Ejemplo de formato de diccionario creado en código

La creación de microservicios en main.py conlleva un conjunto de funciones con método POST para recibir datos de autenticación y entregar la configuración de los dispositivos, para declarar los servicios se requiere el uso de Flask, Json, Request, Cors, Environment, YAML y entre otras librerías que se detalla en la ilustración 6, para mayor detalle de las funciones de microservicios acceder al repositorio del proyecto.

```
main.py
1  from flask import Flask,json,request
2  import logging
3  from flask_cors import CORS, cross_origin
4  from jinja2 import Environment, FileSystemLoader
5  import yaml
6  from gpapi import gpapi
7  app = Flask(__name__)
8  cors = CORS(app)
9  app.config['CORS_HEADERS'] = 'Content-Type'
10
11  @app.route('/show',methods=['POST'])
12  @cross_origin()
13  > def show(): ...
```

Ilustración 6 Declaración de librerías y ejemplo de microservicio "show" para pruebas

```

60 @app.route('/getparsingcfg',methods=['POST'])
61 @cross_origin()
62 def getparsingcfg():
63     try:
64         json_data = request.get_json()
65         show = json_data["show"]
66         plantilla = json_data["plantilla"]
67         connection = gpapi(json_data)
68         parse = connection.showconfig(show)
69         data = {"data":parse}
70         connection.disconnect()
71     try:
72         show2 = json_data["show2"]
73         parse2 = connection.showconfig(show2)
74         data = {"data":parse,"data2":parse2}
75     except:
76         data = {"data":parse}
77         connection.disconnect()
78         env = Environment(loader = FileSystemLoader('.'), trim_blocks=True, lstrip_blocks=True)
79         template = env.get_template(plantilla)
80         doc = template.render(data)
81         s=200
82     except:
83         doc="No data"
84         s=400
85     response = app.response_class(response=doc,
86                                   status=s,
87                                   mimetype='text/cfg')

```

Ilustración 7 Ejemplo de función para obtener plantilla de configuración

Para poder agregar más funciones de microservicio, la documentación de los diccionarios con la información de los dispositivos intermediarios se puede acceder a <https://pubhub.devnetcloud.com/media/genie-feature-browser/docs/#/parsers>.

Paso 3: Creación de programa base de Front-End

La base del proyecto pertenece al panel de trabajo Argon de Creative Tim, para más información visitar <https://www.creative-tim.com/product/argon-dashboard-angular>. Se procede a construir el Front-End con interfaces para obtener información de dispositivos y obtener las plantillas de configuración, así mismo, el repositorio del proyecto se puede utilizar como base del Front-End.

Paso 4: Desarrollo de Front-End

En esta sección se explicará información necesaria que involucra las funcionalidades del proyecto.

- a) Se establece una estructura que pertenece a información esencial de los dispositivos.

```
1  export class Device {  
2    $key: string;  
3    name: string;  
4    os: string;  
5    ip: string;  
6    model: string;  
7    protocol: string;  
8    encryp:string;  
9  }
```

Ilustración 8 Estructura de base de datos

- b) Se utiliza la librería file-saver para poder descargar los archivos de configuración.

```
npm install file-saver --save
```

- c) Se utiliza la librería AngularFireAuth como método de autenticación para acceder a la plataforma web.
- d) Se utiliza las librerías AngularFireDatabase y AngularFireList para las funciones de base de datos en la nube, tal como, obtener, insertar, actualizar y eliminar datos.

```
npm install --save firebase
```

- e) Los archivos que definen la estructura se detallan a continuación:

El archivo `app.module.ts` define el archivo principal de la plataforma web indicando estructura, enrutamiento y conexión con los servicios.

```
18 @NgModule({
19   imports: [
20     BrowserModule,
21     FormsModule,
22     HttpClientModule,
23     ComponentsModule,
24     NgbModule,
25     RouterModule,
26     AppRoutingModule,
27     AngularFireModule.initializeApp(environment.firebase),
28     AngularFireDatabaseModule,
29   ],
30   declarations: [
31     AppComponent,
32     AdminLayoutComponent,
33     AuthLayoutComponent
34   ],
35   providers: [AngularFireAuth, AuthGuard, AuthService],
36   bootstrap: [AppComponent]
37 })
38 export class AppModule { }
```

Ilustración 9 archivo `app.module.ts`

El archivo `environment.ts` define la estructura con la información de autenticación con el servicio en la nube de Firebase.

```
5 export const environment = {
6   production: false,
7   firebase :{
8     apiKey: "xxxxxxxxxxxxx",
9     authDomain: "netmi-frontend.firebaseio.com",
10    projectId: "netmi-frontend",
11    storageBucket: "netmi-frontend.appspot.com",
12    messagingSenderId: "xxxxxxxxx",
13    appId: "xxxxxxxxxxxxx",
14    measurementId: "G-xxxxxxxxx"
15  }
16 };
```

Ilustración 10 archivo `environment.ts`

El archivo `app.routing.ts` define el enrutamiento general de la plataforma web, de modo que divide la aplicación en `AdminLayoutComponent` y `AuthLayoutComponent`.

```
8  const routes: Routes = [
9    {
10     path: '',
11     redirectTo: 'dashboard',
12     pathMatch: 'full',
13   }, {
14     path: '',
15     component: AdminLayoutComponent,
16     children: [
17       {
18         path: '',
19         loadChildren: './layouts/admin-layout/admin-layout.module#AdminLayoutModule'
20       }
21     ],
22   }, {
23     path: '',
24     component: AuthLayoutComponent,
25     children: [
26       {
27         path: '',
28         loadChildren: './layouts/auth-layout/auth-layout.module#AuthLayoutModule'
29       }
30     ]
31   }, {
32     path: '**',
33     redirectTo: 'dashboard'
34   }
35 ];
```

Ilustración 11 `app.routing.ts`

El archivo `auth-layout.routing.ts` define el enrutamiento de la plataforma web cuando se requiere autenticación de usuario, por lo cual, solo se puede acceder a la ruta de 'login'.

```
5  export const AuthLayoutRoutes: Routes = [
6    { path: 'login', component: LoginComponent }
7  ];
```

Ilustración 12 `auth-layout.routing.ts`

El archivo `admin-layout.routing.ts` define el enrutamiento de la plataforma web cuando existe autenticación de usuario.

```
7  export const AdminLayoutRoutes: Routes = [
8    { path: 'dashboard', component: DashboardComponent, canActivate : [AuthGuard] },
9    { path: 'configuration', component: DevicesComponent, canActivate : [AuthGuard] },
10   { path: 'tables', component: TablesComponent, canActivate : [AuthGuard] },
11 ];
```

Ilustración 13 `admin-layout.routing.ts`

El archivo global.ts define la ruta que conecta el Front-End con los microservicios.

```
1 export const ruta = "http://xxxxxxxxxx:5000/";
```

Ilustración 14 global.ts

Paso 5: Despliegue de aplicación

a) Despliegue local y simulación de GNS3

Despliegue de Back-End

ng serve

Despliegue de Front-End

python3 main.py

b) Despliegue en la nube

Despliegue de Back-End en Cloud Run

o	Abrir SDK de Gcloud, clonar repositorio
< / >	cd src
	gcloud app deploy
¿?	Desplegar: yes
< / >	gcloud app browse

Tabla 3 Pasos para despliegue en Cloud Run

Despliegue de Front-End en Firebase

< / >	firebase login
¿?	Compartir datos: y/n [opcional]
< / >	ng build
	Firebase init
Opción	Hosting
¿?	Elegir directorio público: yes
	Sobreescribir index: no
< / >	Firebase deploy

c) Despliegue en contenedores.

Se recomienda crear una carpeta destinada a la virtualización de contenedores, en ella se definirá un archivo docker-compose.yml y dos carpetas para división del Back-End y Front-End con archivos Dockerfile.

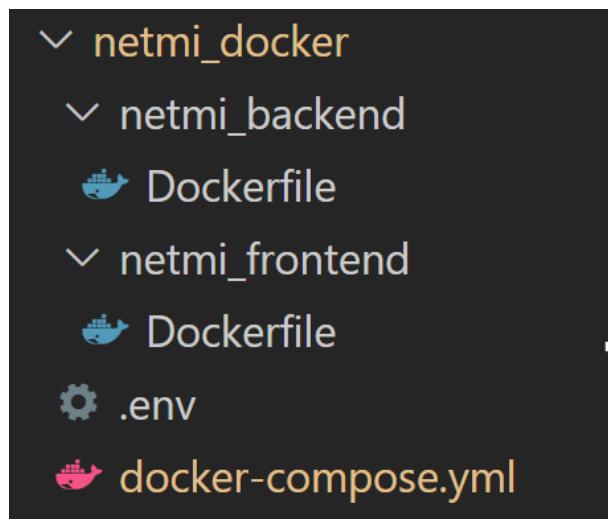


Ilustración 15 Estructura de carpeta para docker

En docker-compose.yml se definirá como servicio al Back-End y Front-End definiendo una red, nombre de contenedor, puertos de acceso, ubicación de Dockerfile y entre otras configuraciones. En cada Dockerfile se indicará el tipo de contenedor con sus respectivas instrucciones y ubicación de archivos para crear la virtualización. Un ejemplo de estas configuraciones personalizadas de contenedores se encuentra en la ruta netmi_docker del repositorio del proyecto. Para crear los contenedores, ubicarse en la carpeta principal donde se encuentran estos y proceder con el comando:

```
docker-compose up --build -d
```

Almacenamiento de código fuente

El código desarrollado de todo el servicio web será colocado en una organización privada de Github creada por nosotros. En él se crearán los repositorios destinados a cada parte del proyecto realizado, además se agregará al cliente y los respectivos tutores para su constancia.

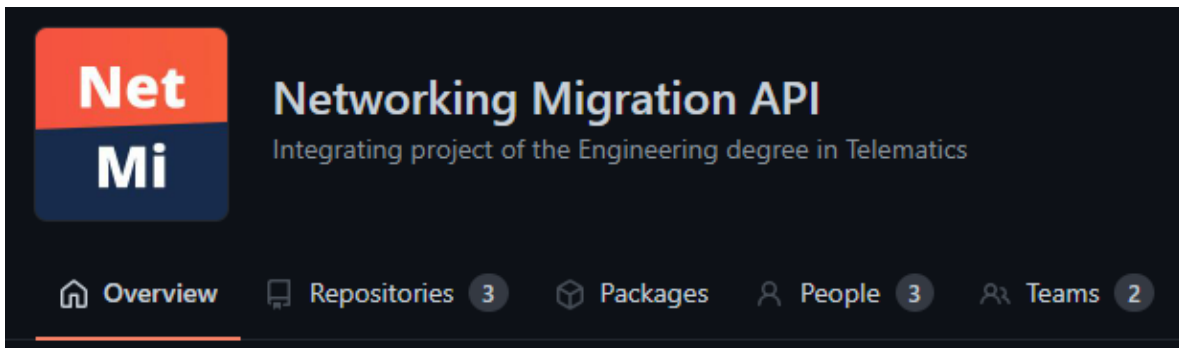


Ilustración 16 Organización en Github de la plataforma web NetMi

En esta organización se han creado tres repositorios donde se manejarán individualmente los cambios e interacciones del Back-End y Front-End, el ultimo repositorio contendrá los archivos Docker que realizaran la respectiva instalación de todos los contenedores necesarios para su ejecución.



Ilustración 17 Estructura del proyecto con Back-End, Front-End y Docker

Dentro de este último repositorio se encuentra un instructivo de instalación desarrollado en un archivo README.MD que se puede visualizar de la siguiente forma.

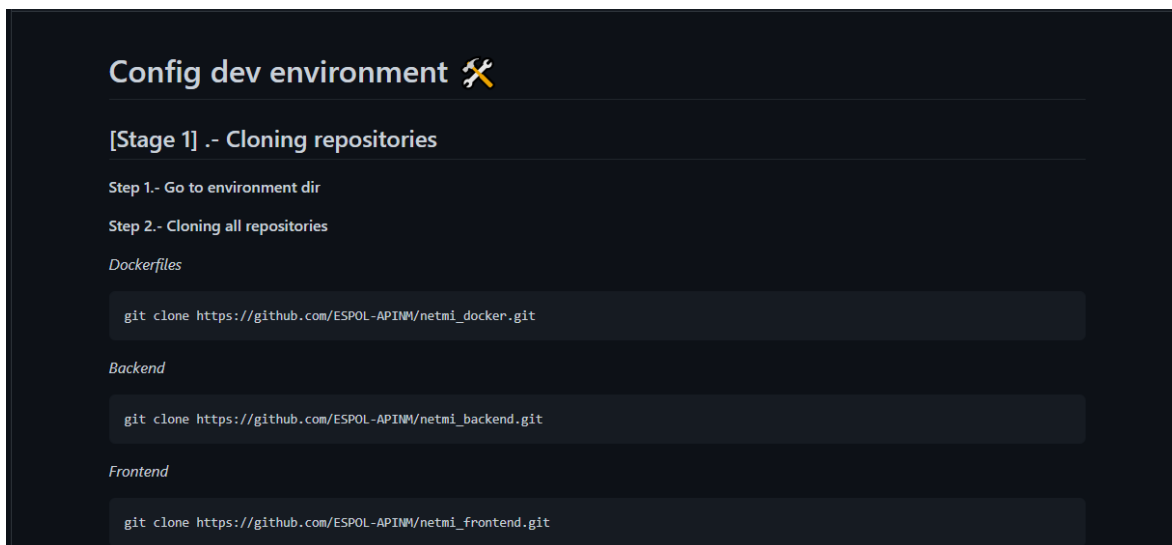


Ilustración 18 archivo readme.md para clonar el proyecto