**📖 MANUAL SCRIPT PYTHON v2 - Para Programadores Java**

**🎯 ¿QUÉ HACE EL SCRIPT? (En términos Java)**

**Es como un programa Java que:**

// Pseudocódigo Java

HashMap<String,Object> datos = cargarYAMLs();

Template plantilla = seleccionarPlantilla();

String configuracion = plantilla.render(datos);

escribirArchivo(configuracion);

**Pero en Python, más simple:**

datos = cargar\_yamls()

plantilla = seleccionar\_plantilla()

configuracion = plantilla.render(datos)

guardar\_archivo(configuracion)

**🏗️ ARQUITECTURA DE CLASES (Como Java)**

class GeneradorConfiguraciones: # ← Tu clase principal

def \_\_init\_\_(self): # ← Constructor

self.global\_config = None # ← Variables de instancia

self.sede\_config = None

self.dispositivo\_config = None

self.jinja\_env = None

def cargar\_configuraciones(self): # ← Métodos (como Java)

def configurar\_jinja(self):

def validar\_configuracion(self):

def generar\_configuracion(self):

def guardar\_configuracion(self):

**Es exactamente como:**

public class GeneradorConfiguraciones {

private Object globalConfig;

private Object sedeConfig;

private Object dispositivoConfig;

private JinjaEnvironment jinjaEnv;

public GeneradorConfiguraciones() { }

public boolean cargarConfiguraciones() { }

public boolean configurarJinja() { }

// etc...

}

**🔄 FLUJO DE EJECUCIÓN (Como main() de Java)**

**1. PUNTO DE ENTRADA (main)**

if \_\_name\_\_ == "\_\_main\_\_": # ← public static void main(String[] args)

main()

def main():

parser = argparse.ArgumentParser() # ← Scanner o similar para argumentos

args = parser.parse\_args() # ← Leer argumentos línea comandos

generador = GeneradorConfiguraciones() # ← new GeneradorConfiguraciones()

generador.cargar\_configuraciones() # ← generador.cargarConfiguraciones()

**2. PROCESAMIENTO (Como métodos de clase)**

# Es como tener estos métodos en Java:

public boolean cargarConfiguraciones(String sede, String tipo) {

// Leer 3 archivos YAML (como leer Properties o JSON)

this.globalConfig = leerYAML("global\_config.yaml");

this.sedeConfig = leerYAML("sedes/" + tipo + ".yaml");

this.dispositivoConfig = leerYAML("dispositivos/" + dispositivo + ".yaml");

return true;

}

**📂 VARIABLES DE INSTANCIA (Como atributos Java)**

class GeneradorConfiguraciones:

def \_\_init\_\_(self):

# Estas son como private variables en Java:

self.global\_config = None # ← private HashMap globalConfig;

self.sede\_config = None # ← private HashMap sedeConfig;

self.dispositivo\_config = None # ← private HashMap dispositivoConfig;

self.jinja\_env = None # ← private TemplateEngine jinjaEnv;

**Uso posterior:**

def cargar\_configuraciones(self):

self.global\_config = datos # ← this.globalConfig = datos;

def generar\_configuracion(self):

plantilla = self.jinja\_env.get\_template() # ← this.jinjaEnv.getTemplate()

**🔍 ANÁLISIS FUNCIÓN POR FUNCIÓN (Como métodos Java)**

**🏭 cargar\_configuraciones() - Como método DAO**

def cargar\_configuraciones(self, sede\_nombre, tipo\_sede, tipo\_dispositivo):

# Es como un método que lee 3 archivos:

try:

with open('global\_config.yaml', 'r') as f: # ← FileInputStream

self.global\_config = yaml.safe\_load(f) # ← JSON.parse() o similar

archivo\_sede = f'sedes/{tipo\_sede}.yaml' # ← String concatenation

with open(archivo\_sede, 'r') as f:

self.sede\_config = yaml.safe\_load(f)

# Personalizar nombre (como setter)

self.sede\_config['sede']['nombre'] = sede\_nombre # ← sede.setNombre()

except FileNotFoundError as e: # ← catch (FileNotFoundException)

print(f"[ERROR] Archivo no encontrado: {e}")

return False

**Equivalente Java:**

public boolean cargarConfiguraciones(String sedeNombre, String tipoSede, String tipoDispositivo) {

try {

// Cargar global config

this.globalConfig = leerYAML("global\_config.yaml");

// Cargar config de sede

String archivoSede = "sedes/" + tipoSede + ".yaml";

this.sedeConfig = leerYAML(archivoSede);

// Personalizar nombre

((HashMap)sedeConfig.get("sede")).put("nombre", sedeNombre);

return true;

} catch (FileNotFoundException e) {

System.out.println("[ERROR] Archivo no encontrado: " + e.getMessage());

return false;

}

}

**⚙️ configurar\_jinja() - Como inicializar TemplateEngine**

def configurar\_jinja(self):

# Como inicializar un motor de plantillas en Java:

self.jinja\_env = Environment( # ← new TemplateEngine()

loader=FileSystemLoader('plantillas') # ← engine.setTemplateDirectory()

)

# Registrar funciones helper (como registrar utils)

self.jinja\_env.filters['cidr\_to\_netmask'] = cidr\_to\_netmask

**✅ validar\_configuracion() - Como método validator**

def validar\_configuracion(self):

errores = [] # ← ArrayList<String> errores

if self.sede\_config['sede']['bloques\_activos']['cifrado']: # ← if (config.isCifradoActivo())

if 'cifrado' not in self.sede\_config: # ← if (!config.hasCifrado())

errores.append("Cifrado activo pero sin configuracion")

if errores: # ← if (!errores.isEmpty())

for error in errores: # ← for (String error : errores)

print(f" - {error}")

return False

return True

**🎯 determinar\_plantilla() - Como método selector**

def determinar\_plantilla(self, tipo\_dispositivo, modulo\_cifrado, modulo\_gestion):

# Como un switch/case en Java:

if modulo\_cifrado: # ← if (moduloCifrado)

return "cifrado\_modulo.j2"

elif modulo\_gestion: # ← else if (moduloGestion)

if tipo\_dispositivo.startswith('router'): # ← if (tipoDispositivo.startsWith("router"))

return "gestion\_router\_modulo.j2"

elif tipo\_dispositivo.startswith('switch'): # ← else if (tipoDispositivo.startsWith("switch"))

return "gestion\_switch\_modulo.j2"

else:

return f"{tipo\_dispositivo}.j2" # ← return tipoDispositivo + ".j2";

**🏗️ generar\_configuracion() - Como método processor principal**

def generar\_configuracion(self, tipo\_dispositivo, modulos...):

# 1. Determinar plantilla (como factory pattern)

plantilla\_nombre = self.determinar\_plantilla(...)

plantilla = self.jinja\_env.get\_template(plantilla\_nombre) # ← factory.createTemplate()

# 2. Preparar datos (como crear DTO)

contexto = { # ← HashMap<String, Object> contexto

'global\_config': self.global\_config, # ← contexto.put("globalConfig", this.globalConfig)

'sede': self.sede\_config['sede'], # ← contexto.put("sede", this.sedeConfig.get("sede"))

'direccionamiento': self.sede\_config['direccionamiento'],

'interfaces': self.dispositivo\_config['interfaces']

}

# 3. Generar (como template.process())

configuracion = plantilla.render(\*\*contexto) # ← String config = template.process(contexto)

return configuracion

**Equivalente Java:**

public String generarConfiguracion(String tipoDispositivo, boolean modulos...) {

// 1. Factory pattern

String plantillaNombre = determinarPlantilla(...);

Template plantilla = this.jinjaEnv.getTemplate(plantillaNombre);

// 2. Crear DTO/Map

HashMap<String, Object> contexto = new HashMap<>();

contexto.put("globalConfig", this.globalConfig);

contexto.put("sede", ((HashMap)this.sedeConfig).get("sede"));

contexto.put("direccionamiento", ((HashMap)this.sedeConfig).get("direccionamiento"));

// 3. Procesar template

String configuracion = plantilla.render(contexto);

return configuracion;

}

**🎮 JINJA2 = Motor de Plantillas (Como Velocity/Freemarker)**

**Plantilla .j2 (Como .vm de Velocity):**

hostname {{ sede.nombre | replace(" ", "\_") }}\_ppal <!-- ${sede.nombre}\_ppal -->

!

{% for vrf\_name, vrf\_config in global\_config.segmentacion.vrfs.items() %}

ip vrf {{ vrf\_name }} <!-- #foreach($vrf in $vrfs) -->

rd {{ vrf\_config.rd }} <!-- rd ${vrf.rd} -->

{% endfor %} <!-- #end -->

**Datos YAML = Properties/JSON:**

sede:

nombre: "alcorcon" # sede.setNombre("alcorcon")

global\_config:

segmentacion:

vrfs:

main:

rd: "65000:1" # vrf.setRd("65000:1")

**Resultado final:**

hostname alcorcon\_ppal

!

ip vrf main

rd 65000:1

**Es exactamente como usar Velocity o Freemarker en Java.**

**🚨 MANEJO ERRORES (Try/Catch como Java)**

try:

archivo = open("datos.yaml") # ← FileInputStream file = new FileInputStream()

datos = yaml.safe\_load(archivo) # ← procesar archivo

except FileNotFoundError as e: # ← catch (FileNotFoundException e)

print(f"[ERROR] Archivo no encontrado: {e}") # ← System.out.println()

return False # ← return false;

except yaml.YAMLError as e: # ← catch (YAMLException e)

print(f"[ERROR] Error al procesar YAML: {e}")

return False

**Idéntico a Java, solo cambia la sintaxis.**

**🎛️ ARGUMENTOS LÍNEA COMANDOS (Como Apache Commons CLI)**

parser = argparse.ArgumentParser() # ← CommandLineParser parser = new DefaultParser()

parser.add\_argument('--sede', required=True) # ← options.addOption("sede", true, "Nombre sede")

parser.add\_argument('--dispositivo', required=True) # ← options.addOption("dispositivo", true, "Tipo")

args = parser.parse\_args() # ← CommandLine cmd = parser.parse(options, args)

sede\_nombre = args.sede # ← String sede = cmd.getOptionValue("sede")

**Funciona igual que cualquier parser de argumentos en Java.**

**💡 PECULIARIDADES PYTHON vs JAVA**

**1. \*\*contexto = "Expandir HashMap"**

contexto = {'sede': 'madrid', 'tipo': 'simple'}

plantilla.render(\*\*contexto)

# Es lo mismo que:

plantilla.render(sede='madrid', tipo='simple')

**En Java sería:**

HashMap contexto = new HashMap();

contexto.put("sede", "madrid");

contexto.put("tipo", "simple");

template.render(contexto); // Pasar el HashMap completo

**2. f"string {variable}" = String.format()**

nombre = "alcorcon"

mensaje = f"Procesando sede: {nombre}" # ← String.format("Procesando sede: %s", nombre)

**3. List comprehensions = Stream API**

activos = [k for k, v in bloques.items() if v] # ← bloques.stream().filter(v -> v).map(k -> k).collect()

**🎯 RESUMEN EJECUTIVO PARA JAVASTICOS**

**✅ LO QUE YA SABES:**

* **Es una clase normal** con constructor, métodos, variables de instancia
* **Try/catch** funciona igual (solo se llama try/except)
* **Lógica de programación idéntica** - mismos patrones (factory, DAO, etc.)
* **Colecciones similares** - HashMap=dict, ArrayList=list
* **Main() funciona igual** (solo diferente sintaxis)

**🆕 LO NUEVO:**

* **No declaras tipos** - variable = valor (en lugar de String variable = valor)
* **self en lugar de this**
* **Indentación en lugar de {}**
* **Sintaxis más concisa** - menos verboso
* **Diccionarios y listas integrados** - no necesitas importar Collections

**🏆 CONCLUSIÓN:**

**Tu script es un programa Java normal, escrito en Python.** Si entiendes Java, entiendes este script perfectamente.

**La lógica es 100% familiar - solo la sintaxis es más simple.**