# Analizar POMs de Maven

Para abordar su consulta, proporcionaré una solución para generar un grafo de dependencias en formato DOT al encontrar recursivamente todos los archivos `pom.xml` en un directorio y analizar sus dependencias, sin suponer una estructura de proyecto Maven multimodular fija como la del ejemplo (`project/module1/pom.xml`, etc.). El script se centrará en recolectar primero todos los archivos `pom.xml`, luego en analizarlos para determinar las dependencias entre módulos y generar el grafo.

---

## Descripción General de la Solución

Aquí está lo que hará el script: 1. **Recolección Recursiva de Archivos `pom.xml`**: Recorrerá el directorio raíz especificado para encontrar todos los archivos `pom.xml`, independientemente de su ubicación o estructura. 2. **Análisis de Información del Módulo**: Extraerá `groupId`, `artifactId` y dependencias de cada `pom.xml`. 3. **Manejo de Herencia de `groupId`**: Si un `pom.xml` no especifica un `groupId`, lo heredará de una declaración `<parent>`, si está presente. 4. **Identificación de Dependencias Intra-Proyecto**: Asociará dependencias a módulos dentro del proyecto en función de `groupId` y `artifactId`. 5. **Generación de Salida en Formato DOT**: Creará un grafo dirigido de dependencias entre módulos.

Este enfoque no depende de una jerarquía de directorios predefinida ni asume que los archivos `pom.xml` padres estén en directorios padres.

---

## Script en Python

A continuación se muestra el script en Python adaptado a sus requisitos:

```
import os
import sys
import xml.etree.ElementTree as ET

# Definir el espacio de nombres de Maven POM
NS = "{http://maven.apache.org/POM/4.0.0}"

# Caché para groupId para evitar análisis redundantes
group_id_cache = {}

def get_group_id(pom_path, pom_map):
    """
```

```
    Extraer el groupId de un archivo pom.xml, considerando la herencia del padre.

    Args:
        pom_path (str): Ruta al archivo pom.xml.
        pom_map (dict): Mapeo de rutas de archivos pom.xml a sus datos analizados.

    Returns:
        str: El groupId del módulo.
    """
    if pom_path in group_id_cache:
        return group_id_cache[pom_path]

    tree = ET.parse(pom_path)
    root = tree.getroot()
    group_id_elem = root.find(NS + 'groupId')

    if group_id_elem is not None:
        group_id = group_id_elem.text.strip()
    else:
        # Verificar la declaración del padre
        parent = root.find(NS + 'parent')
        if parent is not None:
            parent_group_id = parent.find(NS + 'groupId').text.strip()
            parent_artifact_id = parent.find(NS + 'artifactId').text.strip()
            parent_relative_path = parent.find(NS + 'relativePath')
            if parent_relative_path is not None and parent_relative_path.text:
                parent_pom_path = os.path.normpath(
                    os.path.join(os.path.dirname(pom_path), parent_relative_path.text)
                )
            else:
                # Predeterminar al directorio padre si relativePath está omitido
                parent_pom_path = os.path.join(os.path.dirname(pom_path), '..', 'pom.xml')
                parent_pom_path = os.path.normpath(parent_pom_path)

            if parent_pom_path in pom_map:
                group_id = get_group_id(parent_pom_path, pom_map)
            else:
                raise ValueError(f"POM padre no encontrado para {pom_path}: {parent_pom_path}")
        else:
            raise ValueError(f"No groupId o padre especificado en {pom_path}")
```

```
    group_id_cache[pom_path] = group_id
    return group_id

def get_artifact_id(pom_path):
    """
    Extraer el artifactId de un archivo pom.xml.

    Args:
        pom_path (str): Ruta al archivo pom.xml.

    Returns:
        str: El artifactId del módulo.
    """
    tree = ET.parse(pom_path)
    root = tree.getroot()
    artifact_id_elem = root.find(NS + 'artifactId')

    if artifact_id_elem is None:
        raise ValueError(f"pom.xml debe especificar artifactId: {pom_path}")

    return artifact_id_elem.text.strip()

def get_dependencies(pom_path):
    """
    Extraer la lista de dependencias de un archivo pom.xml.

    Args:
        pom_path (str): Ruta al archivo pom.xml.

    Returns:
        list: Lista de tuplas (groupId, artifactId) para cada dependencia.
    """
    tree = ET.parse(pom_path)
    root = tree.getroot()
    dependencies = []

    for dep in root.findall(NS + 'dependencies/' + NS + 'dependency'):
        dep_group_id_elem = dep.find(NS + 'groupId')
        dep_artifact_id_elem = dep.find(NS + 'artifactId')
```

```
        if dep_group_id_elem is not None and dep_artifact_id_elem is not None:
            dep_group_id = dep_group_id_elem.text.strip()
            dep_artifact_id = dep_artifact_id_elem.text.strip()
            dependencies.append((dep_group_id, dep_artifact_id))

    return dependencies

if __name__ == '__main__':
    # Verificar argumentos de la línea de comandos
    if len(sys.argv) != 2:
        print("Uso: python script.py <directorio_raiz>")
        sys.exit(1)

    root_dir = sys.argv[1]
    if not os.path.isdir(root_dir):
        print(f"Error: {root_dir} no es un directorio")
        sys.exit(1)

    # Paso 1: Encontrar recursivamente todos los archivos pom.xml
    pom_files = [
        os.path.join(root, file)
        for root, _, files in os.walk(root_dir)
        for file in files if file == 'pom.xml'
    ]

    if not pom_files:
        print(f"No se encontraron archivos pom.xml en {root_dir}")
        sys.exit(1)

    # Paso 2: Construir un diccionario de todos los POMs para búsquedas de padres
    pom_map = {pom_file: None for pom_file in pom_files}

    # Paso 3: Extraer información del módulo
    modules = {}  # (groupId, artifactId) -> pom_path
    for pom_file in pom_files:
        try:
            group_id = get_group_id(pom_file, pom_map)
            artifact_id = get_artifact_id(pom_file)
            modules[(group_id, artifact_id)] = pom_file
        except ValueError as e:
```

```
        print(f"Advertencia: Saltando {pom_file} debido a error: {e}")
        continue

# Paso 4: Analizar dependencias
dependencies = set()
for pom_file in pom_files:
    try:
        importer_group_id = get_group_id(pom_file, pom_map)
        importer_artifact_id = get_artifact_id(pom_file)
        importer_key = (importer_group_id, importer_artifact_id)
        deps = get_dependencies(pom_file)
        for dep_group_id, dep_artifact_id in deps:
            dep_key = (dep_group_id, dep_artifact_id)
            if dep_key in modules and dep_key != importer_key:
                # Agregar dependencia como tupla (importador, importado) usando artifactId para simplicid
                dependencies.add((importer_artifact_id, dep_artifact_id))
    except ValueError as e:
        print(f"Advertencia: Error al procesar dependencias en {pom_file}: {e}")
        continue

# Paso 5: Salida en formato DOT
print('digraph G {')
for from_module, to_module in sorted(dependencies):
    print(f'  "{from_module}" -> "{to_module}";')
print('}')
```

---

## Cómo Funciona

### 1. Entrada de Línea de Comandos

- Toma un solo argumento: `<directorio_raiz>`, el punto de partida para la búsqueda recursiva.
- Valida que sea un directorio.

### 2. Encontrar Archivos `pom.xml`

- Usa `os.walk` para recorrer recursivamente el árbol de directorios y recolectar todos los archivos `pom.xml` en una lista.

### 3. Análisis de Información del Módulo

- **Group ID (`groupId`)**:
  - Extraído de cada `pom.xml`.
  - Si no está presente, busca una sección `<parent>` y resuelve el `groupId` del POM padre referenciado usando `relativePath` (o predetermina al directorio padre si está omitido).
  - Cachéa resultados para evitar reanálisis.
- **Artifact ID (`artifactId`)**: Debe estar presente en cada `pom.xml`.
- **Dependencias**: Extrae pares `(groupId, artifactId)` de la sección `<dependencies>`.

### 4. Análisis de Dependencias

- Construye un mapa de `(groupId, artifactId)` a `pom_path` para todos los módulos.
- Para cada `pom.xml`, verifica sus dependencias contra el mapa de módulos para encontrar dependencias intra-proyecto.
- Excluye dependencias de sí mismo (donde un módulo depende de sí mismo).
- Registra dependencias como pares `(importer_artifactId, imported_artifactId)`.

### 5. Salida en Formato DOT

- Genera un grafo dirigido en formato DOT, usando `artifactId` como etiquetas de nodos para simplicidad.

---

## Ejemplo de Uso

Supongamos que su estructura de directorios es irregular:

```
myproject/
  app/pom.xml (groupId="com.myapp", artifactId="app", depende de "core")
  libs/core/pom.xml (groupId="com.myapp", artifactId="core")
  tools/util/pom.xml (groupId="com.myapp", artifactId="util", depende de "core")
```

Ejecute el script:

```
python script.py myproject
```

**Salida:**

```
digraph G {
  "app" -> "core";
  "util" -> "core";
}
```

Esto muestra que tanto `app` como `util` dependen de `core`.

---

## Notas y Limitaciones

- **Sin Agrupación por Directorio**: A diferencia del script original en el rastro de pensamiento, esta versión no agrupa módulos por estructura de directorios ya que no lo solicitó. Usa `artifactId` directamente en el grafo.
- **Resolución de Padre**: Depende de `relativePath` en la sección `<parent>` o asume que el POM padre está en el directorio padre si está omitido. Si el POM padre no está en el conjunto recolectado, lanza un error.
- **Coincidencia Simple**: Coincide dependencias solo por `groupId` y `artifactId`, ignorando versiones.
- **Sin Sustitución de Propiedades**: No maneja propiedades de Maven como `${project.groupId}`.
- **Manejo de Errores**: Salta archivos `pom.xml` problemáticos con una advertencia en lugar de fallar por completo.

Este script cumple con su requisito de “iterar recursivamente, obtener todos los POMs primero y luego analizar”sin suponer una estructura de archivos fija. ¡Hágamelo saber si necesita ajustes!