

Analyser des projets Spring Boot

Pour analyser un projet Spring Boot (qui peut consister en plusieurs modules) et extraire toutes les API définies avec les annotations `@GetMapping` ou `@PostMapping`, nous devons parcourir le répertoire du projet, identifier les classes de contrôleur, extraire les chemins des API et les imprimer dans un format spécifié. Voici un script Python complet qui accomplit cette tâche, inspiré de la structure du code fourni mais adapté aux nouvelles exigences. La sortie listera chaque contrôleur avec ses chemins GET et POST associés au format "XXController: chemin get chemin post".

Approche

1. **Parcourir le Répertoire** : Trouver de manière récursive tous les fichiers `.java` dans le répertoire du projet et ses sous-répertoires.
2. **Identifier les Contrôleurs** : Vérifier chaque fichier Java pour les annotations `@Controller` ou `@RestController` pour confirmer qu'il s'agit d'une classe de contrôleur.
3. **Extraire les Chemins au Niveau de la Classe** : Rechercher les annotations `@RequestMapping` au niveau de la classe pour déterminer les chemins de base.
4. **Extraire les API au Niveau de la Méthode** : Trouver les méthodes annotées avec `@GetMapping` ou `@PostMapping`, extraire leurs chemins et les combiner avec les chemins de base s'ils sont présents.
5. **Organiser et Imprimer** : Collecter les chemins GET et POST par contrôleur et les imprimer dans le format spécifié.

Script Python

```
import os
import sys
import re
from collections import defaultdict

def find_java_files(root_dir):
    """
    Trouver de manière récursive tous les fichiers .java dans le répertoire donné et ses sous-répertoires.

    Args:
        root_dir (str): Le répertoire racine à partir duquel commencer la recherche.

    Yields:
        str: Le chemin complet de chaque fichier .java.
    """
    for dirpath, dirnames, filenames in os.walk(root_dir):
```

```

for filename in filenames:
    if filename.endswith('.java'):
        yield os.path.join(dirpath, filename)

def extract_paths(line, annotation_type):
    """
    Extraire les valeurs de chemin d'une annotation Spring (@GetMapping, @PostMapping, ou @RequestMapping).

    Args:
        line (str): La ligne contenant l'annotation.
        annotation_type (str): Le type d'annotation ('GetMapping', 'PostMapping', ou 'RequestMapping').

    Returns:
        list: Une liste de chaînes de chemins extraites de l'annotation.
    """
    if annotation_type in ['GetMapping', 'PostMapping']:
        match = re.search(rf'@{annotation_type}\((.*)\)', line)
        if match:
            content = match.group(1)
            # Extraire toutes les littérales de chaîne entre guillemets
            paths = re.findall(r'"([^\"]*)"', content)
            return paths
        return []
    elif annotation_type == 'RequestMapping':
        match = re.search(r'@RequestMapping\((.*)\)', line)
        if match:
            content = match.group(1)
            # Rechercher l'attribut 'value' ou 'path'
            value_match = re.search(r'(value|path)\s*=\s*({[^\}]*}|"([^\"]*)"')', content)
            if value_match:
                value = value_match.group(2)
                if value.startswith('{'):
                    paths = re.findall(r'"([^\"]*)"', value)
                else:
                    paths = [value.strip('"')]
                return paths
            # Si pas de 'value' ou 'path', supposer une spécification de chemin directe
            paths = re.findall(r'"([^\"]*)"', content)
            return paths
    return []

```

```

if __name__ == '__main__':
    # Analyser les arguments de la ligne de commande
    if len(sys.argv) != 2:
        print("Utilisation: python script.py <répertoire_racine>")
        sys.exit(1)

    root_dir = sys.argv[1]
    if not os.path.isdir(root_dir):
        print(f"[ERREUR] Le chemin spécifié n'est pas un répertoire: {root_dir}")
        sys.exit(1)

    print(f"[INFO] Début de l'analyse du répertoire: {root_dir}")

    # Initialiser un dictionnaire pour stocker les mappages des contrôleurs
    controllers = defaultdict(lambda: {'GET': [], 'POST': []})
    total_files = 0
    error_files = 0

    # Traiter tous les fichiers Java
    for java_file in find_java_files(root_dir):
        try:
            with open(java_file, 'r', encoding='utf-8') as f:
                lines = f.readlines()

            # Vérifier si le fichier est un contrôleur
            if any('@Controller' in line or '@RestController' in line for line in lines):
                controller_name = os.path.basename(java_file).replace('.java', '')

                # Trouver la ligne de déclaration de classe pour séparer les annotations au niveau de la classe
                class_line_index = None
                for i, line in enumerate(lines):
                    if re.search(r'public\s+(class|abstract\s+class|interface)\s+\w+', line):
                        class_line_index = i
                        break

                if class_line_index is None:
                    continue

                # Extraire les chemins de base @RequestMapping au niveau de la classe
                base_paths = []

```

```

for line in lines[:class_line_index]:
    if re.search(r'\s*@RequestMapping', line):
        base_paths = extract_paths(line, 'RequestMapping')
        break
if not base_paths:
    base_paths = ['']

# Extraire les chemins de méthode @GetMapping et @PostMapping
get_paths = []
post_paths = []
for line in lines[class_line_index:]:
    if re.search(r'\s*@GetMapping', line):
        paths = extract_paths(line, 'GetMapping')
        for base in base_paths:
            for path in paths:
                full_path = base + path
                get_paths.append(full_path)
    elif re.search(r'\s*@PostMapping', line):
        paths = extract_paths(line, 'PostMapping')
        for base in base_paths:
            for path in paths:
                full_path = base + path
                post_paths.append(full_path)

# Stocker les chemins uniques
get_paths = sorted(list(set(get_paths)))
post_paths = sorted(list(set(post_paths)))

if get_paths or post_paths:
    controllers[controller_name]['GET'] = get_paths
    controllers[controller_name]['POST'] = post_paths

total_files += 1
except Exception as e:
    print(f"[ERREUR] Impossible de lire le fichier {java_file}: {e}")
    error_files += 1

# Imprimer le résumé
print(f"[INFO] Total des fichiers Java tentés: {total_files + error_files}")
print(f"[INFO] Traités avec succès: {total_files}")

```

```

print(f"[INFO] Fichiers avec erreurs: {error_files}")
print(f"[INFO] Total des contrôleurs trouvés: {len(controllers)}")

# Imprimer les résultats dans le format spécifié
for controller, mappings in sorted(controllers.items()):
    print(f"{controller}:")
    for path in mappings['GET']:
        print(f"get {path}")
    for path in mappings['POST']:
        print(f"post {path}")

```

Explication

- **Imports** : Nous utilisons `os` pour le parcours des répertoires, `sys` pour les arguments de la ligne de commande, `re` pour les expressions régulières et `defaultdict` pour organiser les données des contrôleurs.
- `find_java_files` : Cette fonction trouve de manière récursive tous les fichiers `.java` dans le répertoire du projet, gérant plusieurs modules naturellement via `os.walk`.
- `extract_paths` : Cette fonction analyse les annotations `@GetMapping`, `@PostMapping` OU `@RequestMapping` pour extraire les valeurs de chemin. Elle gère :
 - Les chemins uniques (par exemple, `@GetMapping("/chemin")`).
 - Les chemins multiples (par exemple, `@GetMapping({" /chemin1", " /chemin2"})`).
 - Les attributs nommés (par exemple, `@RequestMapping(value = " /chemin")`).
- **Logique principale** :
 - **Gestion des arguments de la ligne de commande** : Prend un répertoire racine en entrée, similaire au script fourni.
 - **Détection des contrôleurs** : Vérifie la présence de `@Controller` ou `@RestController` dans le contenu du fichier.
 - **Chemins de base** : Extrait les chemins `@RequestMapping` au niveau de la classe avant la déclaration de la classe.
 - **Extraction des API** : Traite les lignes après la déclaration de la classe pour `@GetMapping` et `@PostMapping`, combinant les chemins avec les chemins de base.
 - **Sortie** : Imprime chaque contrôleur avec ses chemins GET et POST, un par ligne, précédé de “get” ou “post”.
- **Gestion des erreurs** : Intercepte les exceptions pendant la lecture des fichiers et les rapporte, reflétant la robustesse du code fourni.

Exemple de Sortie

Pour un projet Spring Boot avec un fichier `UserController.java` :

```

@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/all")
    public ResponseEntity<?> getAllUsers() { ... }

    @PostMapping("/create")
    public ResponseEntity<?> createUser() { ... }
}

```

L'exécution de `python script.py /chemin/vers/projet` pourrait produire :

```

[INFO] Début de l'analyse du répertoire: /chemin/vers/projet
[INFO] Total des fichiers Java tentés: 10
[INFO] Traités avec succès: 10
[INFO] Fichiers avec erreurs: 0
[INFO] Total des contrôleurs trouvés: 1
UserController:
get /users/all
post /users/create

```

Notes

- **Simplifications** : Le script suppose que les annotations sont sur une seule ligne et que les chemins sont des littérales de chaîne sans expressions complexes. Pour une utilisation en production, un analyseur Java pourrait être plus robuste.
- **GET et POST uniquement** : Comme demandé, il se concentre sur `@GetMapping` et `@PostMapping`, ignorant les autres méthodes comme PUT ou DELETE (bien que la trace de pensée ait considéré `@RequestMapping` avec des méthodes).
- **Concatenation des chemins** : Combine directement les chemins au niveau de la classe et de la méthode (par exemple, `/users + /all = /users/all`). Spring normalise les multiples barres obliques, donc cela est suffisant pour des fins d'affichage.

Ce script répond aux exigences de la requête tout en maintenant une structure similaire au code fourni, assurant ainsi la familiarité et la fiabilité.