ALOM - TP 2 - Handcrafting a webserver

Table of Contents

1. Présentation et objectifs 1
2. Création d'un projet Maven 1
3. Jouons avec les Threads Java 2
4. Threads & Sockets 3

1. Présentation et objectifs

Dasn ce TP, nous allons manipuler les Threads, Sockets et autres objets bas-niveau, dans le but d'implémenter un web-server qui ressemble à ce que fait Tomcat.

2. Création d'un projet Maven

Créez un projet Maven avec les paramètres suivants :

- groupId : com.miage.alom.tp
- artificatId : java-threads
- version : 1.0.0-SNAPSHOT
- package : com.miage.alom.tp

Pour créer votre projet, utilisez la commande `mvn archetype:generate`.

Maven va vous proposer une liste d'archetypes qui sont autant de modèles de projets et il va vous demander de sélectionner celui que vous souhaitez utiliser :

```
Choose a number or apply filter (format: [groupId:]artifactId, case sensitive
contains)
```

Pour ce TP, nous allons utiliser le modèle nommé `org.apache.maven.archetypes:maven-archetype-simple`. Maven vous demande de confirmer votre sélection en vous affichant le résultat de votre recherche :

```
Choose archetype:
1: remote -> org.apache.maven.archetypes:maven-archetype-simple (An archetype which
contains a simple Maven project.)
```

L'outil vous demande ensuite de saisir la version de l'archetype (e.g., 1.4), un groupId (e.g., com.miage.alom.tp), un artefactId (java-threads) ainsi qu'un numéro de version (1.0.0-SNAPSHOT) et un package (laissez-celui par défaut qui correspond au groupId). Un répertoire portant le nom de l'artefactId a dû être créé dans votre répertoire courant.

3. Jouons avec les Threads Java

Importez le projet dans un IDE de votre choix. Lorsque le projet est importé, allez dans le package de source com.miage.alom.tp, créez la classe Helloworld et collez le code suivant dedans :

```
package com.miage.alom.tp;

import java.util.HashMap;
import java.util.Map;
import java.util.Random;

public class Helloworld extends Thread {
  private static final int NB_BOUCLE = 100;
  private Random random = new Random();
  private Map logs = new HashMap();
  private Long startTime;
  private String name;
  public Helloworld(Long startTime, String name) {
    this.startTime=startTime;
    this.name = name;
  }
  public Map getLogs() {
    return this.logs;
  }
  public void run() {
    for(int i=0;i<NB_BOUCLE;i++) {
      this.logs.put((System.nanoTime()-startTime),"Thread["+name+"] count : ["+i+"]");
      if(random.nextInt(100) < 10) {
        this.logs.put((System.nanoTime()-startTime),"Thread["+name+"] yield");
        Thread.yield();
        try {
          Thread.sleep(10);
        }
        catch(InterruptedException e){
            // nothing to do :)
        }
      }
    }
  }
}
```

Puis, remplacez le code de la classe App créé par Maven par le code suivant :

```
public static void main( String[] args ) {
  long startTime = System.nanoTime();
  Helloworld h1 = new Helloworld(startTime,"A");
  Helloworld h2 = new Helloworld(startTime,"B");
  h1.start();
  h2.start();
  while(h1.isAlive() && h2.isAlive()) {
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  Map result = h1.getLogs();
  result.putAll(h2.getLogs());

  for(Object l : new TreeSet(result.keySet())) {
    System.out.println("["+l+"] "+result.get(l));
  }
  System.out.println("End of execution");
}
```

Lancez l'application en utilisant la classe App (étant donné qu'elle contient une méthode main) à plusieurs reprises, afin de constater que l'ordonnancement n'est pas toujours le même.

Puis modifiez ces deux classes afin que :

- La classe App ne boucle plus tant que h1 et h2 sont alive, mais attendent simplement 100 ms puis arrête h1 et h2
- La classe Helloworld utilise une boucle infinie au démarrage du thread, jusqu'à ce qu'elle soit arrêtée proprement (par un appel extérieur de méthode influant sur la boucle par exemple ? cf cours ...)

Lorsque vous aurez terminé la partie ci-dessus (manipulation des threads et de la classe HelloWorld), vous pouvez passer à la suite.

4. Threads & Sockets

Pour (re)commencer, nous allons recréer un projet java simple avec les paramètres suivants :

- groupId : com.miage.alom.tp
- artificatId : java-webserver
- version : 1.0.0-SNAPSHOT
- package : com.miage.alom.tp

Téléchargez les sources fournies en pièce jointe du TP (java-webserver.zip) et désarchivez les fichiers dans votre répertoire projet/src/main/java. Importez ensuite le projet dans votre IDE.

Votre objectif est de créer un mini serveur web en utilisant les classes fournies (principalement un analyseur de requête HTTP, un handler de fichier et un handler de Servlet), notamment en :

- Créant une écoute sur un port 8080
- Acceptant les connexions sur ce port
- En gérant ces connexions au travers d'un thread dédié (gestion d'une requête à chaque connexion, possibilité d'utiliser une ConcurrentLinkedQueue pour se passer les requêtes d'un thread à un autre)
- Fermant la socket lorsque la requête a été traitée

Pour le rendu du TP, il est attendu un repo GitLab contenant :

- Un fichier README.md
- L'ensemble du répertoire de votre projet java-webserver, notamment le fichier pom.xml, le répertoire src et son contenu, tout autre fichier que vous jugerez utile, mais pas le répertoire target contenant les fichiers compilés

La génération de votre repo GitLab ce fait en ouvrant ce lien : https://gitlab-classrooms.cleverapps.io/assignments/5bc75b86-fdad-44b3-a5b6-f820f2570056/accept

Faites un mvn clean pour vous débarasser du répertoire `target/` juste avant de pusher votre code. Vous pouvez aussi créer un fichier `.gitignore`.

Dans le fichier README.md, n'hésitez pas à détailler ce qui vous a posé problème, les blocages que vous avez eu (que vous ayez réussi à les résoudre ou non), et toute autres informations que vous jugerez utile, en lien avec ce TP.