





# Atelier 9 – Threads, DI, Factory...

## Table des matières

1	Objectifs.....	2
2	Théorie .....	2
3	Introduction.....	2
4	Exercices.....	3
4.1	Maven.....	3
4.2	Requête HTTP .....	5
4.3	Lecture au clavier.....	5
4.4	Threads, domaine, architecture .....	5
4.5	Factory.....	6
4.6	Injection de dépendances.....	6
4.7	 Packaging de l'application.....	7
4.8	 Blacklist .....	7
4.9	 Analyse de contenu.....	8
4.10	 Injection de dépendances plus complexe.....	8

## 1 Objectifs

ID	Objectifs
AJ01	Mettre en pratique les Threads dans un cas réel
AJ02	Utiliser les classes internes
AJ03	Mettre en pratique les factory
AJ04	Utiliser l'injection de dépendances

## 2 Théorie

La théorie sur l'injection de dépendances, les factory... a été vue en cours de COO. N'hésitez pas à vous référer aux slides, à vos notes, et aux exemples de code écrits en cours.

## 3 Introduction

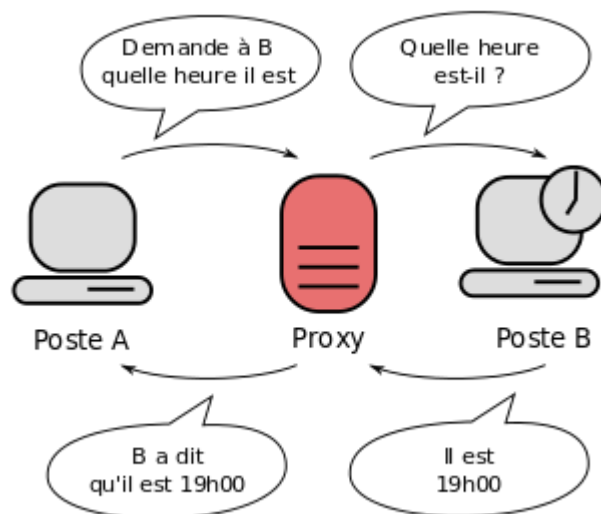
Un proxy est un composant logiciel informatique qui joue le rôle d'intermédiaire en se plaçant entre deux hôtes pour faciliter ou surveiller leurs échanges.

Dans le cadre plus précis des réseaux informatiques, un proxy est alors un programme servant d'intermédiaire pour accéder à un autre réseau, généralement internet. Par extension, on appelle aussi « proxy » un matériel comme un serveur mis en place pour assurer le fonctionnement de tels services.

Source Wikipedia <https://fr.wikipedia.org/wiki/Proxy>

Dans le diagramme ci-contre, le proxy sert bien d'intermédiaire entre le poste A et le poste B pour leurs discussions. Dès lors, on imagine bien que le proxy est bien placé pour vérifier ce que le poste A demande au poste B.

Il existe des proxys web ; ce sont des proxys qui sont programmés pour vérifier les sites web que les clients visitent. Ils permettent de bloquer certains sites, ou certains contenus. Dans ce cas, le client demande au proxy le site web qu'il souhaite visiter. Le proxy vérifie



qu'il a bien ce droit. Si oui, alors il fait lui-même la requête, et transmet la réponse au client. Sinon, il répond au client qu'il n'a pas le droit de visiter le site.

Nous allons aujourd'hui programmer un tel proxy, en version simplifiée. Nous allons uniquement programmer la partie « Proxy -> Poste B » telle que sur le schéma, et nous allons faire une interface en console (`System.in`) pour dialoguer avec notre proxy.

Voici le flux global de notre application :

- On démarre l'application, une console est disponible et nous demande quel site on veut visiter ;
- On écrit dans la console l'adresse du site, par exemple : <https://duckduckgo.com/>;
- Le proxy lit l'adresse dans la console et démarre un nouveau thread pour traiter la demande ;
- Le thread va vérifier dans un fichier de blacklist si le site qui est visité est autorisé. Sinon, il écrit un message d'erreur dans la console ;
- Ce thread va demander sur internet le site à l'adresse qu'on a lue au clavier grâce à un client web ;
- Le client web va nous donner la réponse ;
- Le thread va vérifier si le contenu du site contient des mots, ou des patterns interdits, si oui, il écrit un message d'erreur dans la console ;
- Le thread va écrire la réponse dans la console : le « status code » (200, 400...), ainsi que le « content » qui sera généralement du HTML.

## 4 Exercices

### 4.1 Maven

Pour pouvoir faire des requêtes web, on a besoin d'un client HTTP. Apache en fournit un bon :

<https://hc.apache.org/httpcomponents-client-4.5.x/index.html>

Comme vous le savez, on pourrait importer le JAR dans notre projet. Pour cela, il faudrait :

- Télécharger le JAR
- L'importer dans notre projet (configurer le build path)
- Vérifier régulièrement qu'il n'y a pas des mises à jour
- S'il y en a, les télécharger, mettre à jour le build path

Bref, ce n'est pas facile à gérer, surtout quand on a beaucoup de dépendances...

Maven permet de gérer cela : [https://fr.wikipedia.org/wiki/Apache Maven](https://fr.wikipedia.org/wiki/Apache_Maven)

Cet outil est très complet et complexe. Nous allons aujourd'hui l'utiliser uniquement pour la gestion des packages externes dont notre application dépend. Maven contient un repository centralisé qui contient plein de projets que des développeurs comme vous ont voulu y mettre. Par exemple, vous avez développé un super « Logger » qui pourrait être utile à d'autres, n'hésitez pas à l'enregistrer dans le repository de Maven. N'importe quel autre développeur pourra alors l'installer dans son projet de manière automatique.

C'est ce que nous allons faire ici pour HTTPClient de Apache. Créons donc un projet Maven :

- Dans IntelliJ, cliquez sur **New Project** (File -> new -> Project si vous n'êtes pas dans la fenêtre de départ).
- Donnez le nom **AJ\_atelier09** à votre projet dans **Name**.
- Choisissez l'endroit où vous voulez créer votre projet dans **Location**.
- Choisissez **Maven** (et non IntelliJ !) comme **Build system**.
- Dans **Advanced Settings**, donnez comme **GroupId** **be.vinci.aj**.
- Cliquez sur **Create**.

Un nouveau projet est créé. Maven est lancé, et est en train de télécharger les dépendances, et tout ce qui est nécessaire pour le projet (pas grand-chose pour un projet vide).

Une fois terminé, vous remarquerez plusieurs choses :

- Maven crée plusieurs dossiers pour mettre vos sources. N'y touchons pas, vous mettrez vos sources uniquement dans le dossier `src/main/java`
- Maven crée un fichier `pom.xml` qui décrit le projet. L'artifact ID, combiné au group ID est l'identifiant unique de notre projet. C'est grâce à lui qu'on peut différencier notre projet des autres.
- Parfois, Maven met automatiquement comme JRE java 1.5. Changeons cela en ajoutant dans le fichier `pom.xml`, dans la balise `project`, en dessous de `version` :

```
<properties>
  <maven.compiler.source>17</maven.compiler.source>
  <maven.compiler.target>17</maven.compiler.target>
</properties>
```

Remplacez la **version de java** par celle qui est installée sur votre machine.

- Cliquez droit sur votre projet, dans la partie « Maven », cliquez sur « Reload project ». Maven va alors lire le fichier `pom.xml`, et répercuter les modifications sur votre projet. Cependant, dans certains cas, vous devrez quand même manuellement signaler à IntelliJ qu'il doit utiliser la bonne version de Java. Si vous avez des soucis, n'hésitez pas à vérifier dans "Projet Structure" que vous avez la même version de Java que dans le `pom.xml`.

Pour installer HTTPClient, on va voir dans la documentation : <https://hc.apache.org/httpcomponents-client-5.1.x/quickstart.html>

1. Dans le premier point, on nous dit de suivre un lien si on souhaite l'installer à l'aide d'un gestionnaire de dépendances (maven par exemple). Cliquez dessus.
2. Sur cette nouvelle page, on peut cliquer sur HTTPClient qui va nous rediriger sur Maven Central, qui contient toutes les dépendances qu'on peut installer dans nos projets, ainsi que toutes les versions.
3. On constate que la dernière version semble être la 5.1.4 (au 24/11/22). Cliquez sur le numéro de version pour arriver sur la page de description de la librairie. Constatez le bout d'XML dans la partie en haut à droite, commençant par la balise `<dependency>`, avec un bouton "copier".
4. Retournons dans notre IDE, et ouvrons notre `pom.xml`. Dans la section `<dependencies>` de notre fichier, ajoutez le bout d'XML représentant la dépendance. Si la balise `<dependencies>` n'existe pas, n'hésitez pas à l'ajouter.

5. Pour installer les dépendances, faites à nouveau un « Reload project ». Vous remarquerez qu'il y a, à présent, dans la partie "External Libraries" dans l'explorateur de votre projet, des librairies supplémentaires, et notamment « HTTPClient ».

## 4.2 Requête HTTP

Maintenant que notre projet est prêt, utilisez la [documentation QuickStart du HTTPClient](#) pour faire une requête GET vers l'URL <https://duckduckgo.com/> et afficher le status code ainsi que le HTML. Ceci nous permettra de vérifier que nous arrivons à faire des requêtes web et récupérer les informations que nous voulons.

Ecrivez votre code dans une classe Main dans un package main.

Remarquez l'utilisation d'une factory HttpClients pour obtenir une instance de HttpClient.

### Astuces :

- C'est important de mettre toujours le protocole dans l'URL de la requête : [duckduckgo.com](https://duckduckgo.com/) ne fonctionne pas, alors que [https://duckduckgo.com](https://duckduckgo.com/) fonctionne.
- Une fois que vous avez lancé votre application, vous remarquerez qu'il y a un message rouge qui apparaît. Ne vous inquiétez pas, c'est juste un avertissement. Vous pouvez l'ignorer.
- Pour afficher le HTML de la page, consultez encore une fois la [documentation QuickStart du HTTPClient](#) pour voir comment traiter le "response body" qui est matérialisé par une HttpEntity qui peut être traité à l'aide des méthodes de EntityUtils. Regardez les méthodes disponibles dans cette classe.

## 4.3 Lecture au clavier

Modifiez votre code pour lire au clavier l'adresse à laquelle doit être faite la requête web.

Pour rappel, voici comment lire au clavier :

```
Scanner scanner = new Scanner(System.in);  
String url = scanner.nextLine();
```

Utilisez un try-with-resource pour ne pas oublier de fermer le scanner.

## 4.4 Threads, domaine, architecture

Comme nous avons maintenant la certitude de pouvoir faire les requêtes web correctement, et de lire au clavier les URL, attaquons notre application plus en profondeur.

Créez une classe Query dans le package domaine. Cette classe contient 2 attributs :

- l'URL de la requête (String)
- la "HTTP method" (GET, POST) de type QueryMethod (classe interne énuméré)

Créez une classe QueryHandler dans le package server qui possède un attribut de type Query. Cette classe s'occupe de faire la requête avec le HttpClient, et écrire la réponse à l'écran. Cette classe représente des Threads. Effectivement, comme vous le savez, les requêtes réseau sont particulièrement lentes, comparées à d'autres opérations locales à la machine. Il est donc important de ne pas bloquer le thread principal le temps que la requête du client soit terminée. Cela permet également de paralléliser les traitements. C'est pourquoi on fait la requête HTTP dans un thread à part.

**Remarque** : pour simplifier, le QueryHandler ne gère que des query avec la méthode GET.

Créer une classe ProxyServer dans le package server qui va s'occuper de lire au clavier, créer les Query et démarrer les QueryHandler. Ce comportement est implémenté dans une méthode startServer. N'hésitez pas à faire une boucle infinie de type `while (true) { }` pour lire indéfiniment au clavier, tant que l'application est lancée.

Enfin, la méthode main de la classe Main ne s'occupe maintenant plus que de créer un ProxyServer, et appeler sa méthode startServer.

Vérifiez que votre projet fonctionne correctement. Si vous voulez tester le multi-threading, vous pouvez simuler un temps de requête HTTP plus long, en ajoutant un `Thread.sleep(10000)` dans votre thread. Vous pourrez ainsi faire 2 requêtes, et observer qu'elles sont bien traitées simultanément.

## 4.5 Factory

Maintenant que nous avons une application fonctionnelle et bien structurée, appliquons les concepts vus en COO.

On observe une dépendance concrète très forte entre les classes du package server, et la classe Query. Pour rendre notre code indépendant aux changements, divisons la classe Query, en une interface Query, et une classe QueryImpl.

Pour cela, renommez votre classe Query en QueryImpl, et utilisez l'outil « Refactor -> Extract interface » que vous trouverez en cliquant droit sur la classe QueryImpl.

Vous pouvez à présent changer la visibilité de QueryImpl en « package-friendly ».

Créez dès lors une QueryFactory dans le package domaine, avec une méthode statique getQuery. Utilisez-la dans le package server, au lieu de faire le `new Query` (dépendance concrète).

### **Remarques :**

- Résolvez le problème de visibilité de la classe interne énuméré QueryMethod.
- La factory renvoie des objets « vides », et l'utilisateur de l'objet peut l'initialiser avec les setters.

## 4.6 Injection de dépendances

Mettez en place le système d'injection de dépendances tel que vu en COO pour la QueryFactory. Version simple, par le constructeur, avec « new » dans le `main()`, sans introspection, sans fichier properties...

### **!!! SPOILER ALERT !!!**

Vous n'avez pas d'idée par où commencer ? Vous avez regardé le cours de COO, les exemples de code, et malgré tout vous n'y arrivez pas ? Voici ce à quoi le `main()` devrait ressembler :

```
public static void main(String[] args) {
    QueryFactory queryFactory = new QueryFactoryImpl();
    ProxyServer proxyServer = new ProxyServer(queryFactory);
    proxyServer.startServer();
}
```

**C'est volontairement écrit tout petit, pour vous inciter à le faire vous-même !**

## 4.7 Packaging de l'application

Nous allons créer ensemble un jar exécutable qui va contenir toutes les dépendances téléchargées par Maven (HTTPClient...), et que nous allons ensuite exécuter facilement.

- Ajoutez ce code dans votre pom.xml. Il définit que pour faire un build de l'application, maven doit créer un jar avec les dépendances contenues dedans ;

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>
        <archive>
          <manifest>
            <mainClass>fully.qualified.MainClass</mainClass>
          </manifest>
        </archive>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </plugin>
  </plugins>
</build>
```

- Remplacez `fully.qualified.MainClass` par le nom complet de votre classe main (packages compris) ;
- Demandez à votre IDE de builder l'application. Pour cela, appuyez 2 fois sur la touche CTRL, et entrez `mvn clean compile assembly:single`.
- On vient de demander à Maven de supprimer le contenu du dossier `target`, compiler l'application, et faire un jar ;
- Dans le dossier `target`, vous avez maintenant un jar. Exécutez-le dans un terminal (powershell sur Windows) avec la commande `java -jar le-nom-du-jar`.

Petite astuce : en cliquant droit sur le dossier `target`, vous avez une option "Show in" (ou "open in" pour IntelliJ) -> "Terminal". Ceci vous ouvre un terminal directement dans le dossier `target`.

Si vous avez un souci avec le plugin `maven-assembly-plugin` qui n'est pas trouvé, ajoutez la dépendance suivante :

```
<dependency>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>3.4.2</version>
</dependency>
```

## 4.8 Blacklist

Mettez en place un fichier `properties` qui va contenir une clef `blacklistedDomains` qui aura comme valeur une liste de domaines interdits (ex : `google.be, facebook.com...`), séparés par des « ; ».

Créez un service `BlacklistService` qui va lire ce fichier `properties`, une et une seule fois au chargement de la classe, et qui va s'occuper de vérifier si le site visité est autorisé ou non. Injectez ce service là où on a besoin de lui (`ProxyServer`).

Ce service contiendra un attribut static `blacklistedDomains` de type `Set<String>`. Celui-ci sera initialisé au chargement de la classe (`clinit`). Il contiendra également une méthode d'instance (non static) : `public boolean check(Query query)`. C'est elle qui s'occupera de vérifier si le site est autorisé ou non. Pour cela, vérifiez juste si l'URL de la requête contient le domaine. Petite astuce : il est possible de faire cette vérification en une ligne grâce aux stream ;-)

#### 4.9 Analyse de contenu

Dans le fichier `properties`, ajoutez une clef `unauthorizedContent` qui aura comme valeur une liste d'expressions régulières, séparées par des « ; ».

Ajoutez dans le service qui s'occupe de la blacklist, une méthode qui s'occupe de vérifier si le contenu du site contient des éléments définis dans les expressions régulières.

#### 4.10 Injection de dépendances plus complexe

Mettez en place l'injection de dépendances plus complexe vue au cours de COO : avec annotation, map, introspection...