TP N° 6: Tests de performance (Synchronous programming :Platform threads vs. Virtual threads)

Rappel :

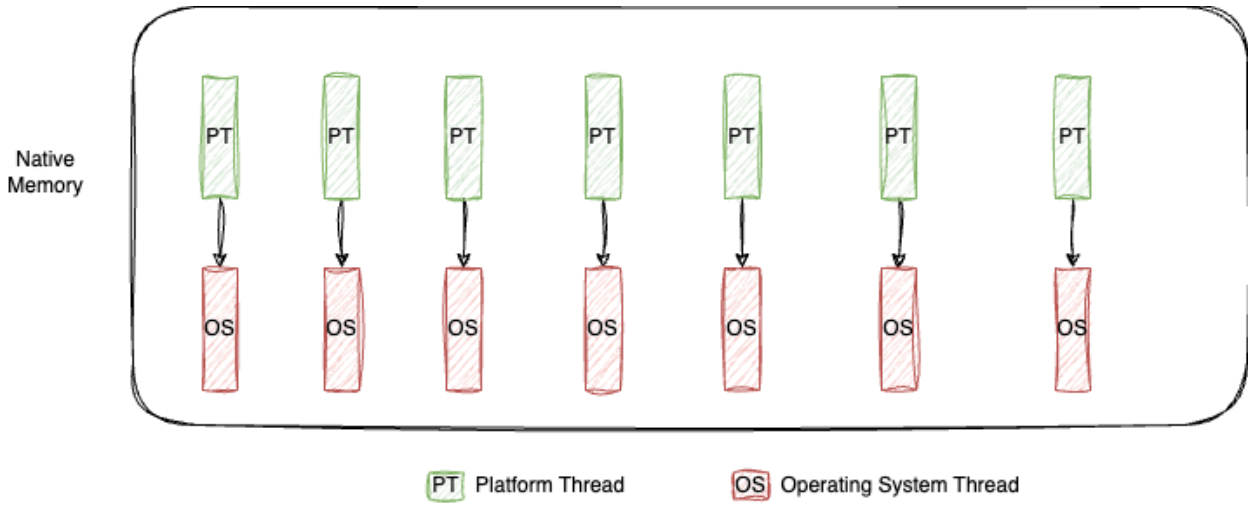


Fig 1 : Platform Thread

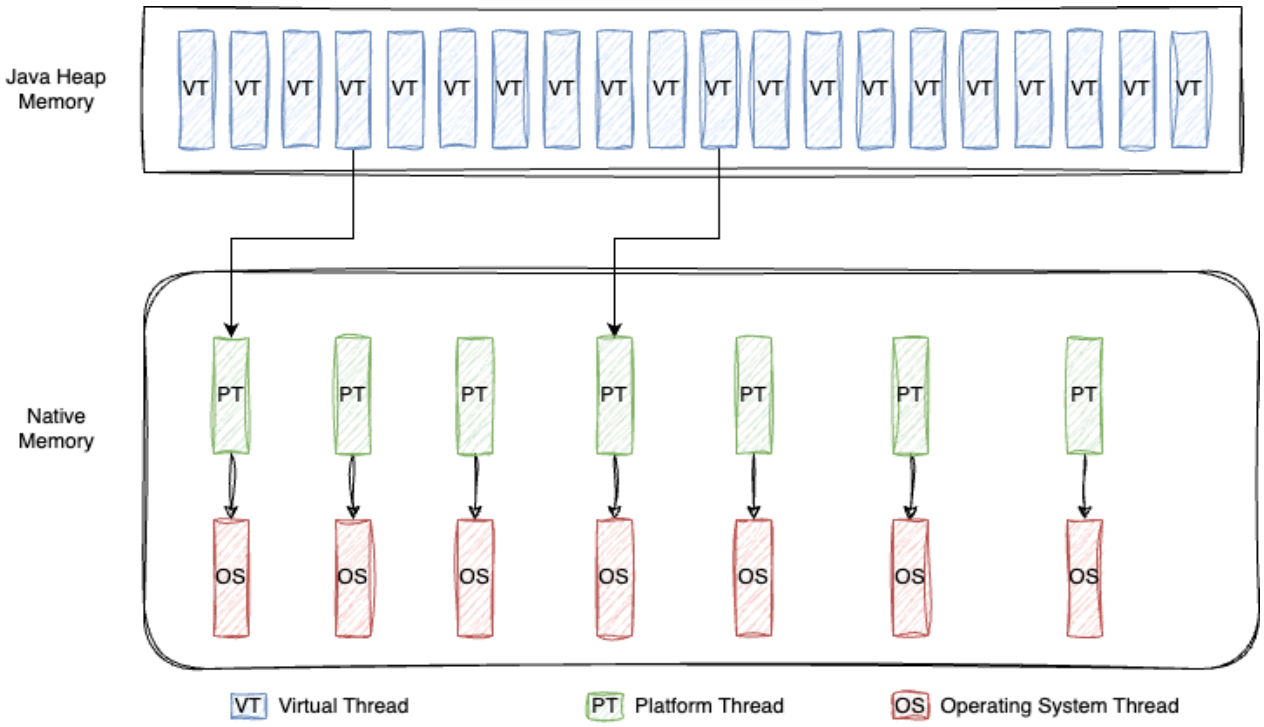


Fig. 2 : Virtual threads

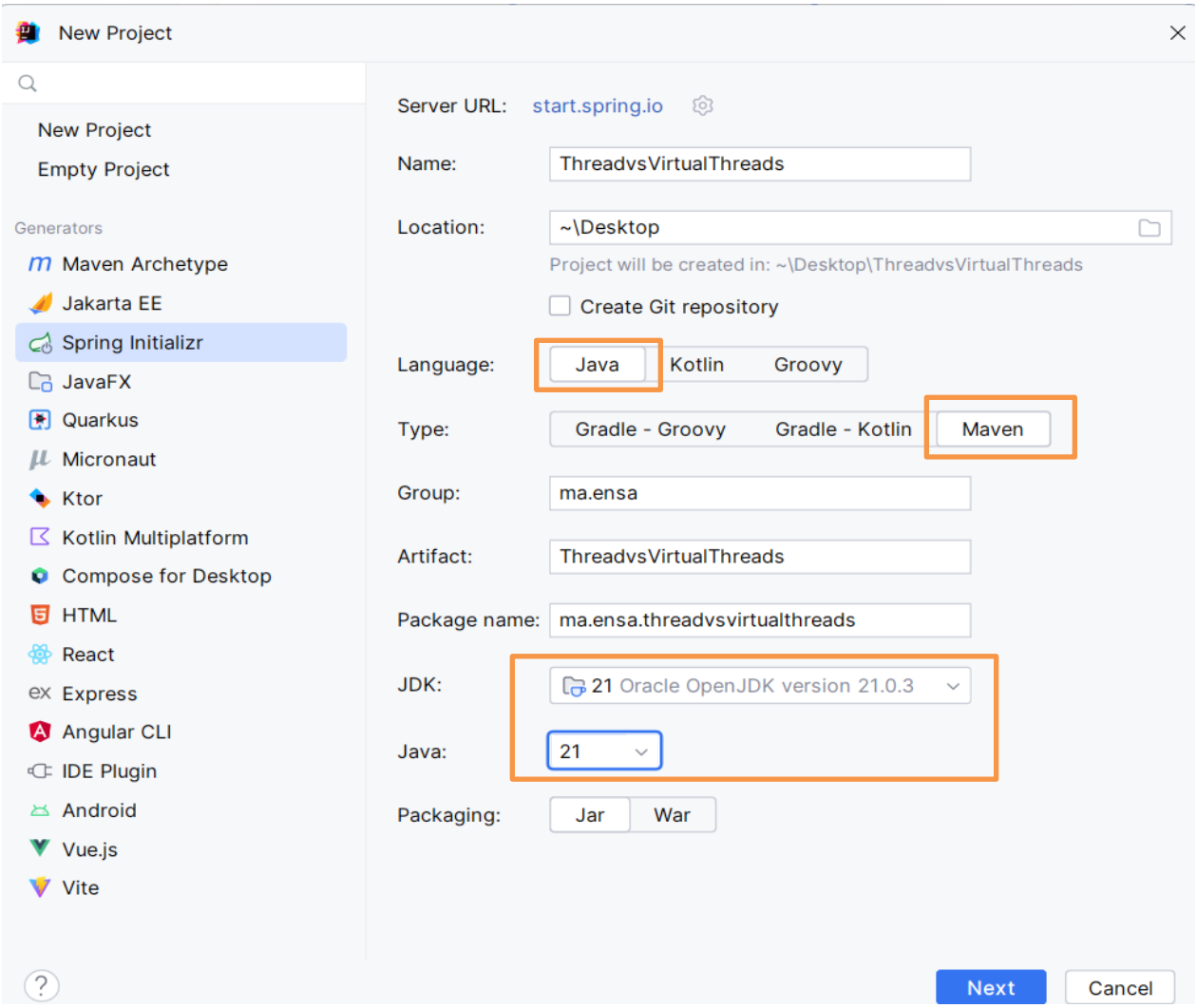
Activité 1 : contexte de programmation synchrone

Dans cette activité, nous souhaitons réaliser une étude benchmark dans le contexte de la programmation synchrone pour comparer la performance des threads classiques (Platform threads) avec les threads virtuels (Virtual threads). Pour illustrer cette étude nous développons une API CRUD à l'aide du Framework Spring Boot afin de gérer les produits d'une boutique virtuelle. Cette API va subir par la suite des tests de performance à l'aide de l'outil Apache Jmeter et Apache Benchmark dans le cas des threads classique et le cas des threads virtuelles.

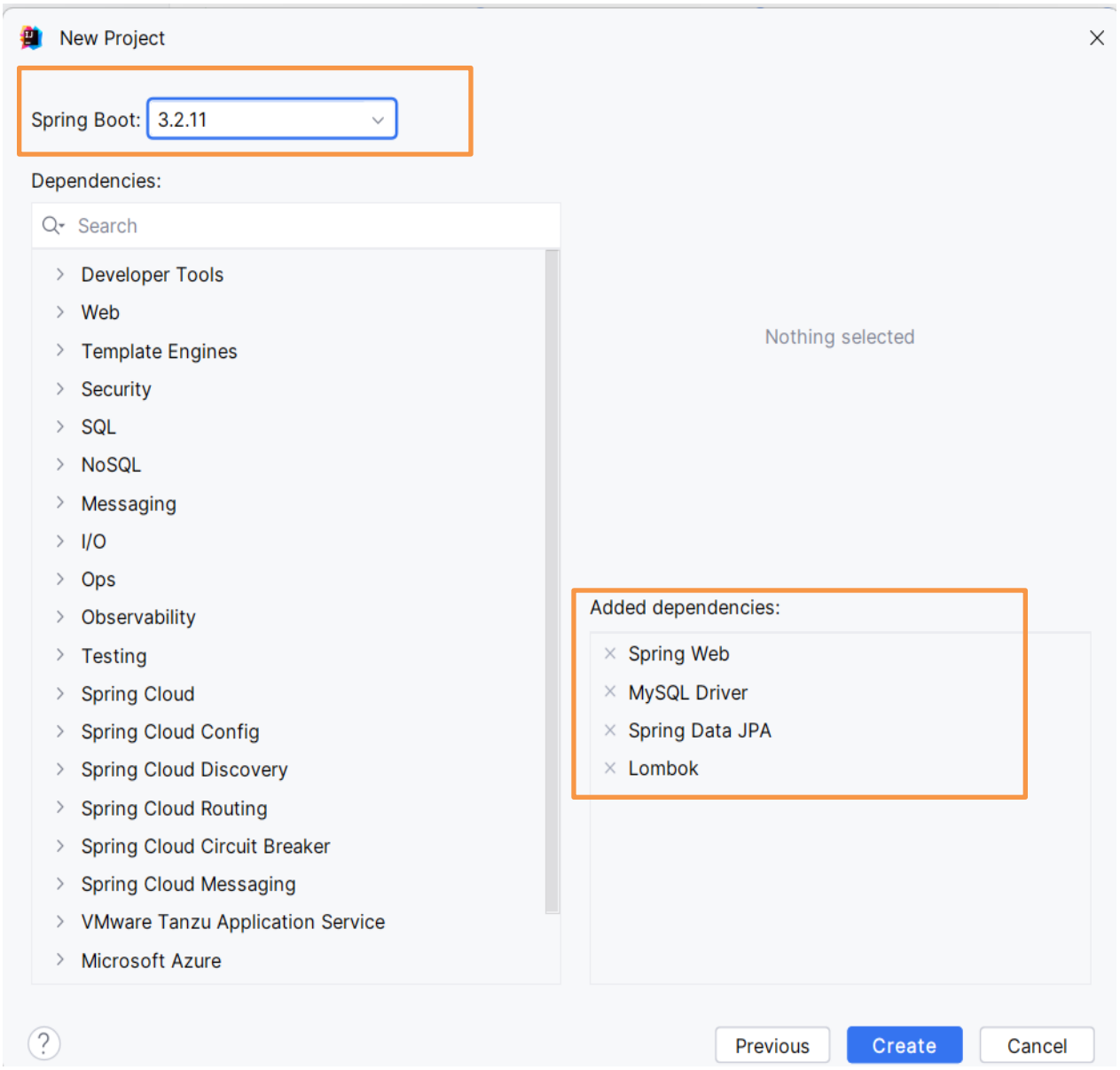
N.B : Le dossier Tools comporte les différents outils nécessaires pour réaliser le TP

Travail à réaliser :

- Etape 1 : Créer un nouveau projet Spring Boot à l'aide d'IntelliJ IDEA comme suit :



- Etape 2 : Choisir la version du Spring Boot (3.2.x) et ajouter les dépendances comme suit :



- Etape 3 : Après la création du projet, ajouter dans le fichier pom.xml la dépendance suivante :

```
<dependency>
   <groupId>org.apache.commons</groupId>
   <artifactId>commons-lang3</artifactId>
   <version>3.12.0</version>
</dependency>
```

The Apache Commons Lang 3 library is a popular, full-featured package of utility classes, aimed at extending the functionality of the Java API.The library's repertoire is pretty rich, ranging from string, array and number manipulation,

reflection and concurrency, to the implementations of several ordered data structures, like pairs and triples (generically known as tuples).

- Etape 4 : Créer l'API CRUD pour gérer les produits de la boutique virtuelle
  a) Créer l'entité Product comme suit :

```
-  package ma.ensa.threadvsvirtualthreads;
   import jakarta.persistence.Entity;
   import jakarta.persistence.GeneratedValue;
   import jakarta.persistence.GenerationType;
   import jakarta.persistence.Id;
   import lombok.Getter;
   import lombok.Setter;
   @Entity
   @Getter
   @Setter
   public class Product {
           @Id
           @GeneratedValue(strategy = GenerationType.IDENTITY)
           private Long id;
           private String productName;
           private Long price;
       }
```

  b) Créer l'interface ProductRepository

```
package ma.ensa.threadvsvirtualthreads;

import org.springframework.data.jpa.repository.JpaRepository;

public interface ProductRepository extends JpaRepository<Product,Long> {
}
```

  c) Créer la classe ProductsService :

```
package ma.ensa.threadvsvirtualthreads;
import org.apache.commons.lang3.RandomStringUtils;
import org.apache.commons.lang3.RandomUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class ProductsService {
    @Autowired
    ProductRepository productRepository;
    public List<Product> getProducts() throws InterruptedException {
        Thread.sleep(1000); // simuler la latence réseau
        return productRepository.findAll();
    }
    public String bulkSaveProduct() throws InterruptedException {
        for(int i=0; i< 1000; i++) {
            Product product = new Product();
```

```
product.setProductName(RandomStringUtils.randomAlphanumeric(5));
            product.setPrice(RandomUtils.nextLong(10,1000));
            product.setPrice(1L);
            saveOrUpdate(product);
        }
        return "finished";
    }
    public Product getProductsById(Long id)
    {
        return productRepository.findById(id).get();
    }
    public void saveOrUpdate(Product product)
    {
        productRepository.save(product);
    }
    //deleting a specific record by using the method deleteById() of
CrudRepository
    public void delete(long id)
    {
        productRepository.deleteById(id);
    }
    //updating a record
    public void update(Product product, long productid)
    {
        productRepository.save(product);
    }

}
```

d) Créer la classe Contrôleur

```
package ma.ensa.threadvsvirtualthreads;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;
@RestController
public class Controller {
    @Autowired
    ProductsService productsservice;

    //creating a get mapping that retrieves all the books detail from
the database
    @GetMapping("/products")
    private List<Product> getAllProducts() throws InterruptedException
{
        return productsservice.getProducts();
    }

    //creating a get mapping that retrieves the detail of a specific
product
    @GetMapping("/products/{productid}")
    private Product getProduct(@PathVariable("productid") long id) {
        return productsservice.getProductsById(id);
    }
```

```
    //creating a delete mapping that deletes a specified product
    @DeleteMapping("/products/{productid}")
    private void deleteProduct(@PathVariable("productid") int id) {
        productsservice.delete(id);
    }

    //creating post mapping that post the product detail in the
database
    @PostMapping("/products")
    private long saveProduct(@RequestBody Product product) {
        productsservice.saveOrUpdate(product);
        return product.getId();
    }

    //creating put mapping that updates the product detail
    @PutMapping("/products/{productid}")
    private Product update(@RequestBody Product product,
@PathVariable("productid") long id) {
        productsservice.update(product,id);
        return product;
    }
    //creating post mapping that save 1000 products detail in the
database
    @PostMapping("/bulksave")
    public String BulkProduct() throws InterruptedException {
        return productsservice.bulkSaveProduct();
    }
}
```

e) Lancer l'installation de MySQL puis créer un compte avec un mot de passe.

f) Créer une base de données avec le compte déjà créé.

g) Dans le fichier application.proprieties ajouter la configuration mysql suivante :

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/nom_de_votre_BD
spring.datasource.username=nom_utistaeur
spring.datasource.password=votre_password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.show-sql: true
```

h) Lancer l'exécution de l'application, puis utiliser l'outil Postman pour ajouter **1000 produits** en lançant une seule requête de type POST avec l'URI suivant : localhost:8080/bulksave

i) Lancer une requête de type GET avec l'URI suivant : localhost:8080/products pour vérifier l'existence de 1000 produits dans la table product de la base de données.

Activité 2 : l'utilisation de l'outil de test de performance « Apache Jmeter»

JMeter est un outil open source gratuit utilisé pour analyser et mesurer les performances des applications, des différents services logiciels et des sites Web. Entièrement écrit en Java, **JMeter**

peut être utilisé pour effectuer des tests de performance, de charge et fonctionnels de nombreuses applications Web et protocoles de serveur différents.

Apache JMeter peut simuler de lourdes charges sur un serveur en créant simultanément plusieurs utilisateurs virtuels, également appelés utilisateurs simultanés. JMeter peut également être utilisé pour tester des applications Web et FTP, des API SOAP et REST, ainsi que des protocoles de messagerie tels que SMTP, POP3, IMAP et bien plus encore.

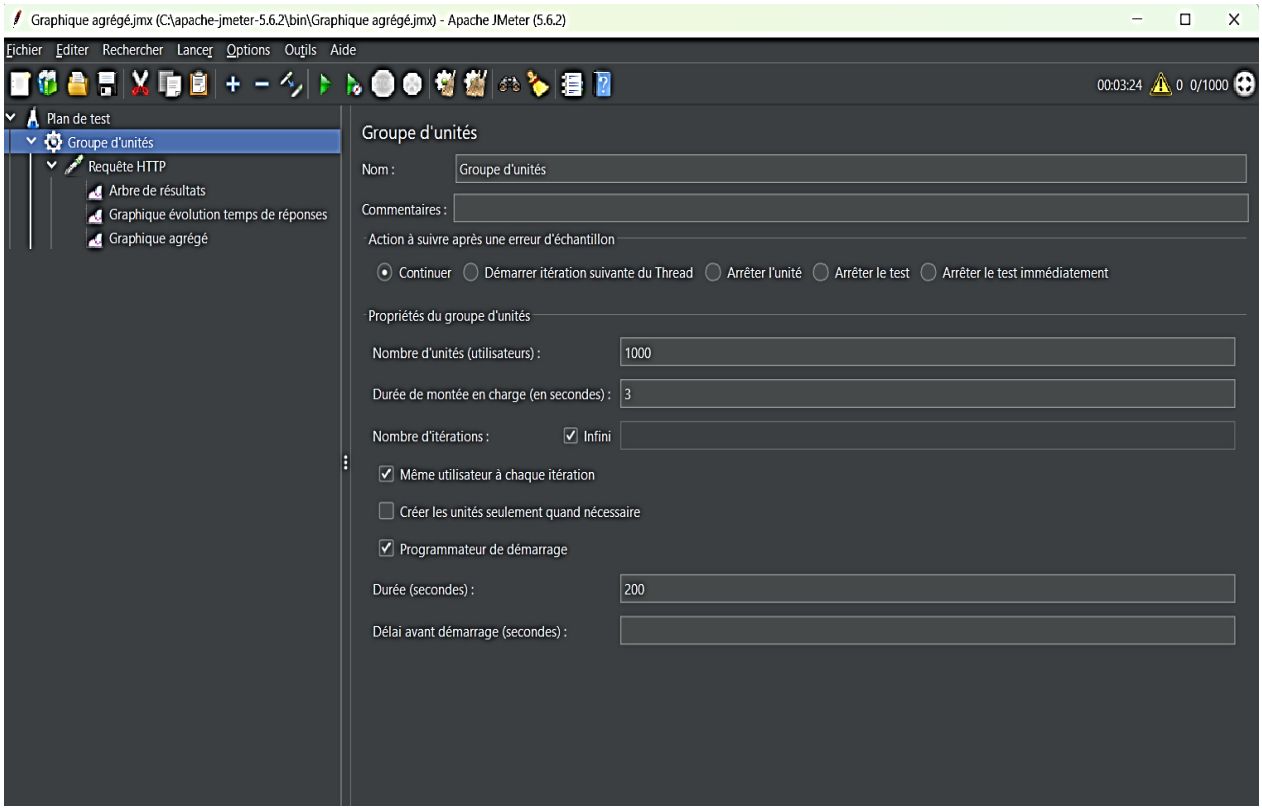
Travail à réaliser

a) Ouvrir le dossier « apache-jmeter-5.6.2 », puis accéder au sous-dossier **bin** puis lancer le jar exécutable de « ApacheJMeter»

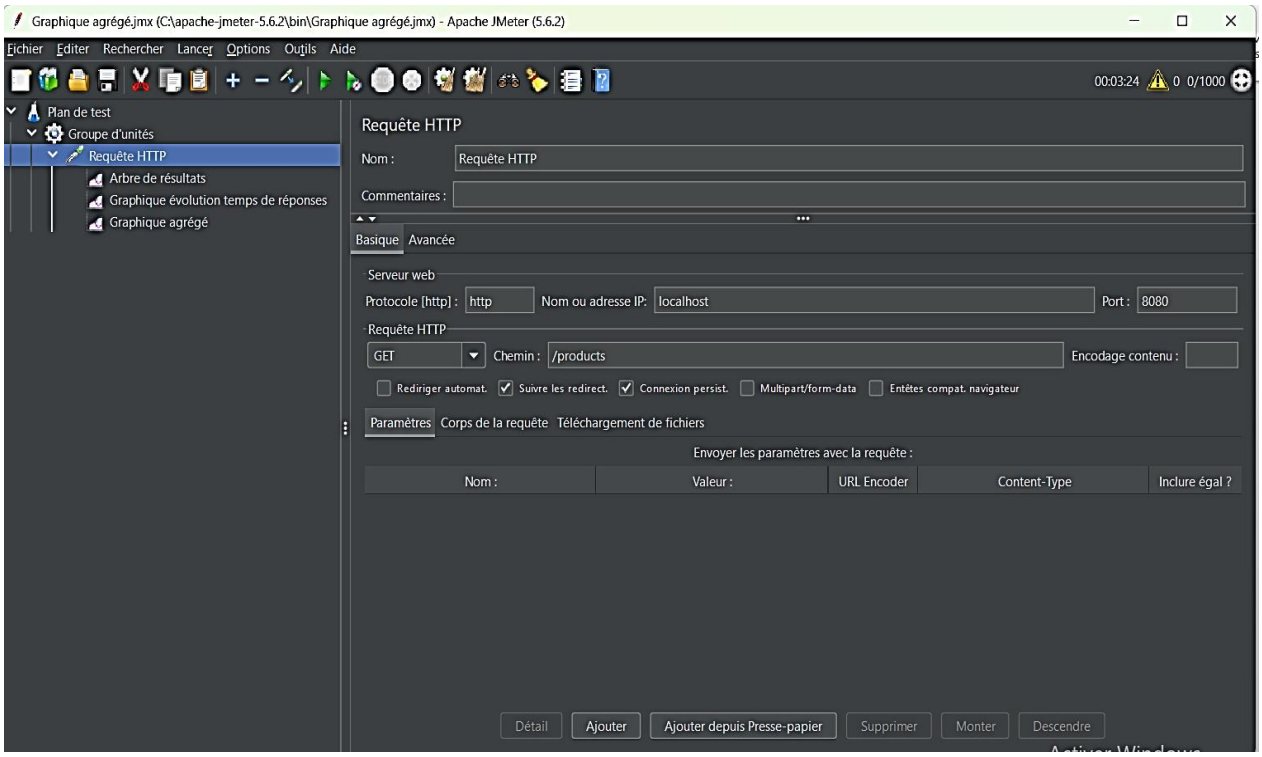
b) Après avoir lancé Jmeter, suivez les étapes suivantes pour lancer un plan de test qui va simuler la concurrence entre utilisateurs pour consommer la ressource suivante : localhost:8080/products

Cas 1 : Threads classiques

i. Créer un groupe d'unité (un **groupe** de threads représente un **groupe** d'**utilisateurs** virtuels effectuant un ensemble d'opérations): clic droit sur le Plan de test puis le remplir par les valeurs comme suit :



ii. Créer une requête http de test lié au groupe déjà crée d'unité comme suit :



iii. Ajouter ensuite les récepteurs de résultats de la requête http : **Arbre de résultats ; Graphique évolution temps de réponse ; graphique de résultats ; graphique agrégé**; **Tableau de résultats**

iv. Démarrer le test puis remplir le tableau suivant à la fin du test :

platform threads				
Débit moy	**Temps de réponse moy**	**% Erreur**	**Ko/sec reçus**	**Ko/sec émis**

Cas 2 : Threads virtuels

Pour activer les threads virtuels Java dans Spring Boot en ajoutant cette configuration dans le fichier « application.properties » :

```
spring.threads.virtual.enabled=true
```

Remarque : Cette configuration nécessite JDK 21 ou supérieur et Spring Boot 3.2 ou supérieur.
Refaire le test pour le cas des threads virtuels puis remplir le tableau suivant :

Virtual threads				
Débit moy	**Temps de réponse moy**	**% Erreur**	**Ko/sec reçus**	**Ko/sec émis**

Activité 3 : l'utilisation de l'outil de test de performance « Apache Benchmark »

Apache Benchmark est un outil de benchmark HTTP permettant de mesurer la performance d'un serveur Web, en particulier le nombre de requête qu'il peut servir par seconde.

Travail à réaliser

a) Ouvrir le dossier « httpd-2.4.62-240904-win64-VS17», puis accéder au sous-dossier bin où se trouve l'outil **ab**.
b) Ajouter le chemin vers **ab** dans la variable d'environnement système PATH.
c) Dans le terminal, lancer l'outil de test en tapant **ab** pour vérifier avant de lancer des tests.

Cas 1 : Threads classique

Pour désactiver les threads virtuels Java dans Spring Boot et revenir au cas des threads classiques changer le fichier « application.properties » par :

```
spring.threads.virtual.enabled=false
```

- **Test 1 (1000 concurrent requests)** :

Dans l'invite de commande ou terminal taper :

```
ab -n 20000 -c 1000 http://localhost:8080/products
```

```
-n requests  :   Number of requests to perform
-c concurrency : Number of multiple requests to make (concurrent requests)
```

Ensuite, remplir le tableau suivant :

platform threads					
Time taken for tests	Requests per second	Time per request	Failed requests	Transfer rate	Ko/sec émis

Cas 2 : Threads virtuels

Réactiver les threads virtuels puis refaire le test1 et remplir le tableau suivant :

virtual threads					
Time taken for tests	Requests per second	Time per request	Failed requests	Transfer rate	Ko/sec émis

- **Test 1 (2000 concurrent requests)** :

Dans l'invite de commande ou terminal taper :

ab -n 20000 -c 2000 http://localhost:8080/products

Comme pour le test précédent, lancer le test2 pour les deux cas (Platform threads et virtual threads) puis remplir les tableaux suivants :

platform threads					
Time taken for tests	Requests per second	Time per request	Failed requests	Transfer rate	Ko/sec émis

virtual threads					
Time taken for tests	Requests per second	Time per request	Failed requests	Transfer rate	Ko/sec émis

Question : Que peut-on déduire après l'analyse des résultats obtenus dans ce TP?