TP : Serveurs HTTP

Compétences visées :
- Mettre en œuvre un serveur HTTP
- Comprendre les différents éléments qui constituent une requête et une réponse HTTP.

1. Configuration de l'environnement de travail

1.1. JDK et JRE pour Java

Dans un premier temps, vous devez vous assurer que votre environnement Java est bien configuré ; c'est-à-dire que les variables d'environnement JAVA_HOME et PATH sont bien configurées.

VARIABLES D'ENVIRONNEMENT	SOUS LINUX (A L'ESISAR)	SOUS WINDOWS
JAVA_HOME	/opt/java/jdk-21.0.3+9	C:\Java
PATH	$JAVA_HOME/bin:$PATH	Ajouter au **début** du PATH : %JAVA_HOME%\bin

Remarque : Si vous travaillez sur votre machine personnelle, vous pouvez installer l'OpenJDK ou bien le JDK d'Oracle ! **Attention à la configuration et au numéro de version !**

Pour vérifier que votre configuration est bonne, vous devez tester en ligne de commandes ces deux instructions :

- `javac -version` (pour s'assurer que le JDK est bien le bon)
- puis `java -version` (pour s'assurer que vous compilez dans la même version que celle que vous exécutez).

Vous devez obtenir le même outil (OpenJDK) et la même version pour les deux commandes.

1.2. Apache Maven

1.2.1. Fonctionnement de Apache Maven à l'Esisar

L'architecture utilisée pour Maven à l'Esisar est une configuration « d'entreprise » avec un dépôt local à l'Esisar présent sur un serveur Sonatype Nexus Repository. L'utilisation de Maven respecte donc un certain nombre de contraintes liées au réseau. La recherche d'une dépendance ne se fait pas directement, elle passe par le dépôt Sonatype Nexus Repository. Voici les différentes étapes pour la recherche d'une dépendance :

1. la machine en salle de TP interroge le serveur Sonatype Nexus Repository en passant par le pare-feu ;
2. si la dépendance est présente, elle est récupérée dans votre dépôt local ;

3. si la dépendance est absente, une requête est faite pour accéder à un dépôt distant (généralement : *Maven Central*), cette requête passe alors par le proxy avant d'aller sur Internet récupérer la dépendance dans le dépôt distant.
4. la réponse à la requête contenant la dépendance est alors retournée au Sonatype Nexus Repository en passant par le proxy, puis elle est transmise à votre dépôt local.

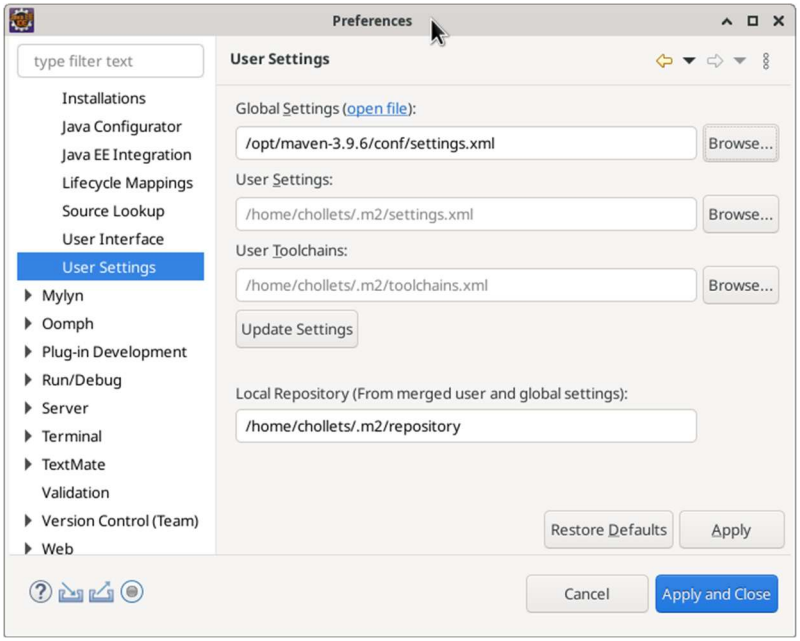
L'intérêt d'utiliser un serveur Sonatype Nexus Repository est de diminuer le nombre d'accès aux dépôts distants. Seules les dépendances absentes sur le serveur Nexus sont téléchargées.

1.2.2. Configuration de Apache Maven

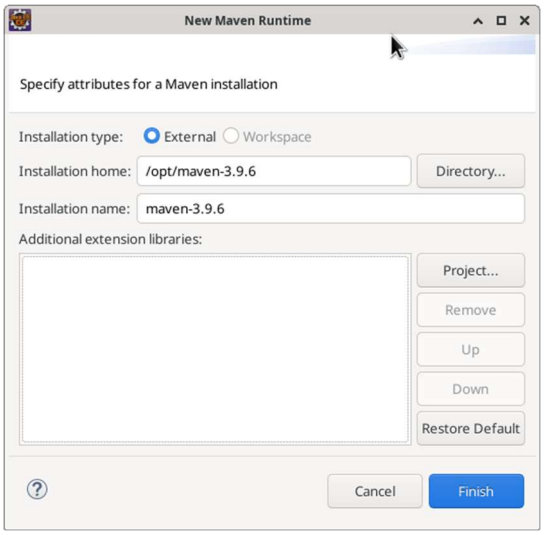
Dans un premier temps, vous devez vous assurer que Maven est bien installé en testant la commande suivante : `mvn -version`.

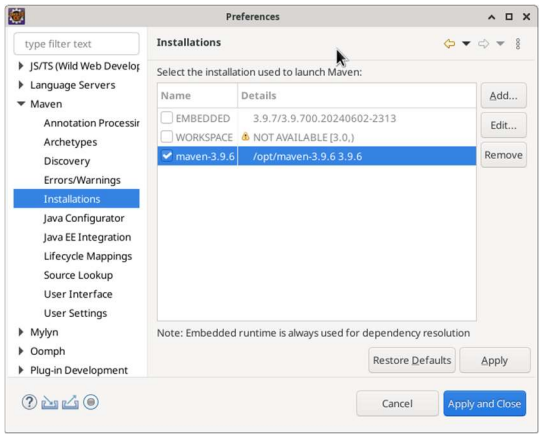
Il est possible de compiler et de packager ses applications directement à partir de l'environnement de développement Eclipse. Ce dernier est livré et configuré, par défaut, avec une version de Maven embarqué dans l'environnement de développement, qui ne correspond pas à la configuration de l'Esisar. Dans le cadre d'une configuration d'entreprise comme à l'Esisar, vous devez impérativement modifier la configuration de l'IDE :

- dans *Window > Preferences > Maven > User Settings*, vous devez remplir la partir *Global Settings* avec le chemin du fichier suivant : */opt/maven-3.9.6/conf/settings.xml*. Puis *Apply*.



- dans *Window > Preferences > Maven > Installations*, vous devez ajouter l'environnement Maven (*Add > Directory > /opt/maven 3.9.6*) et le cocher. Puis *Apply*.





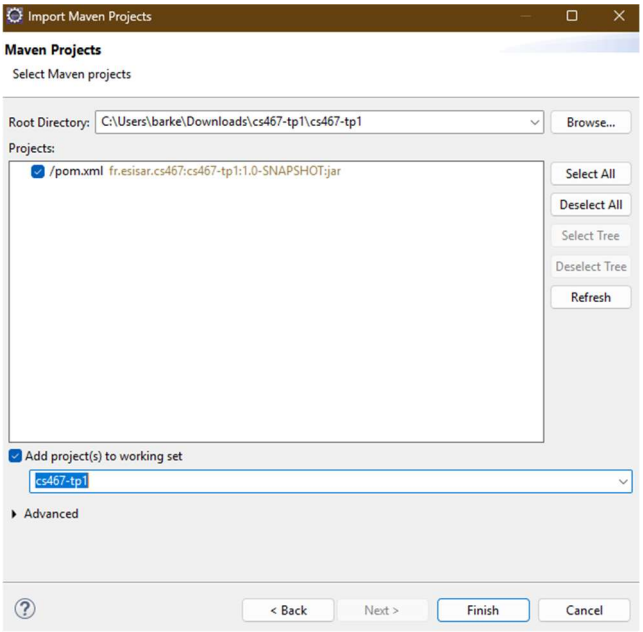
Cette configuration est à faire à chaque changement de workspace !

Pour rappel, à l'Esisar, vous devez travailler dans le répertoire Documents, puis à la fin de la séance, sauvegarder votre travail dans le répertoire Documents_distants. Le répertoire Documents permet de travailler dans de meilleures conditions (pas d'accès distants aux serveurs), par contre, il n'est pas sauvegardé.

2. Exercice 1 : Création d'un serveur HTTP Vert.x

Après avoir téléchargé l'archive cs467-tp1.zip sur Chamilo, vous devez le dézipper, puis l'importer en tant que projet dans Eclipse IDE :

- dans *File > Import > Maven > Existing Maven Projects.*
- puis sélectionnez le répertoire du fichier dézippé.



Pour nettoyer, compiler et exécuter le code de l'application, vous pouvez le faire depuis un terminal de commandes à partir du répertoire du projet avec la commande suivante :

```
mvn clean compile exec:java -DskipTests
```

2.1. Quel jour et quelle heure est-il ?

L'objectif de ce premier exercice est que votre serveur réponde la date et l'heure actuelles à une requête GET envoyée sur le port 8080 :

Requête :

```
curl --get -i http://localhost:8080
```

Réponse attendue :

```
HTTP/1.1 200 OK
Content-Type: text/plain
content-length: 27

2025-01-26T11:43:14.714293
```

A faire :

- Complétez la classe HttpServerRequestHandler avec une méthode getCurrentDateTime() qui prend en paramètre un objet de type HttpServerRequest, qui est une requête HTTP et qui retourne la date et l'heure[1].
- Complétez le code de la méthode handle() de la classe HttpServerRequestHandler pour que la méthode écrite précédemment soit appelée lors d'un appel GET avec le chemin « / » et avec les éléments attendus.
- Testez que votre application est bien fonctionnelle.

2.2. Ping

Vous allez compléter le code de votre serveur pour ajouter une méthode permettant de vous assurer que votre serveur réponde ou non. Le serveur doit répondre positivement le message « pong » à une requête utilisant la méthode POST à l'adresse /ping ayant dans son corps le message « ping », sinon il doit répondre négativement (400 Bad Request).

Requête :

```
curl -i -X POST -H "Content-Type: text/plain" http://localhost:8080/ping --data "ping"
```

Réponse attendue :

```
HTTP/1.1 200 OK
Content-Type: text/plain
content-length: 5

pong
```

Requête :

```
curl -i -X POST -H "Content-Type: text/plain" http://localhost:8080/ping --data "hi"
```

Réponse attendue :

```
HTTP/1.1 400 Bad Request
content-length: 38

Bad request: body doesn't contain ping
```

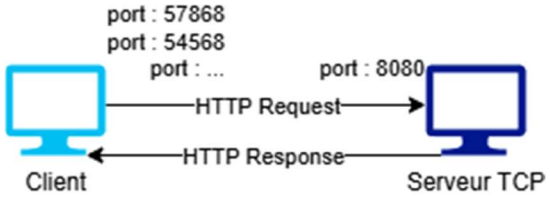
[1] Utilisez la classe LocalDateTime disponible dans Java.

A faire :

- Complétez la classe HttpServerRequestHandler avec une méthode postPing() qui prend en paramètre un objet de type HttpServerRequest, qui est une requête HTTP et qui retourne la réponse attendue selon la requête.
- Complétez le code de la méthode handle() de la classe HttpServerRequestHandler pour que la méthode écrite précédemment soit appelée lors d'un appel POST avec le chemin « /ping » contenant les éléments attendus .
- Testez que votre application est bien fonctionnelle.

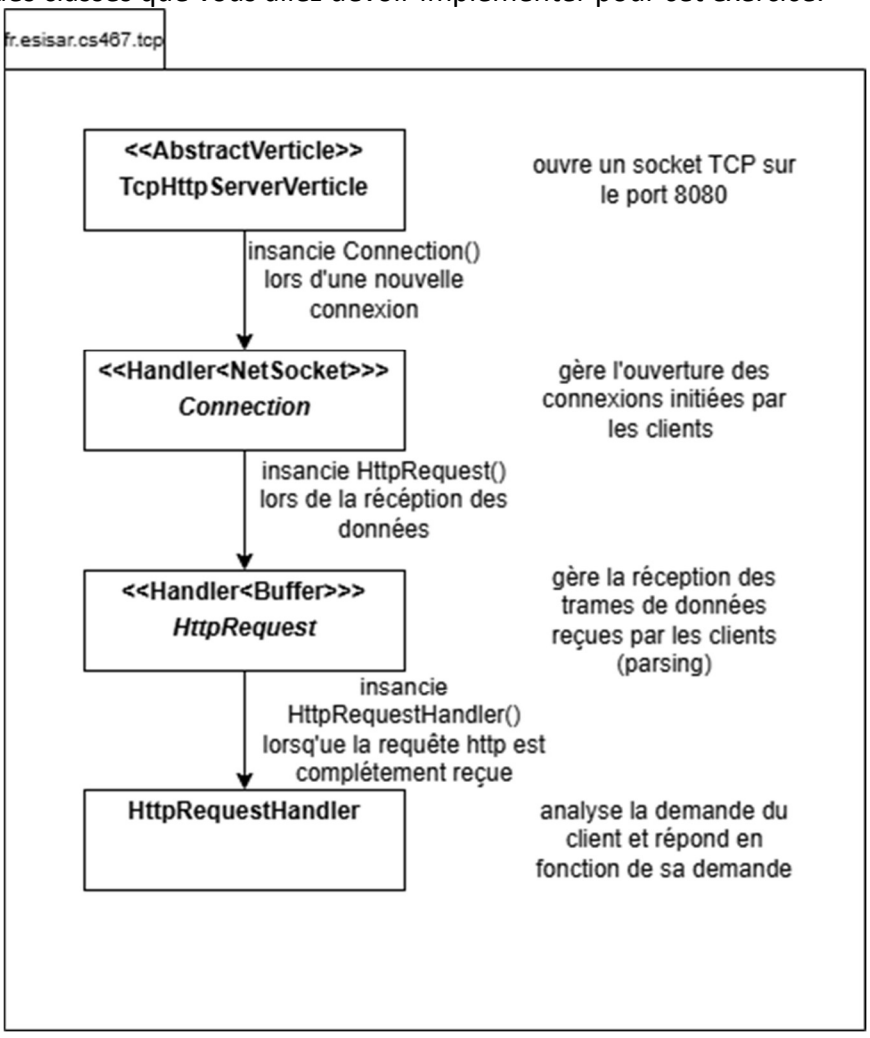
3. Exercice 2 : Serveur HTTP Vert.x basé sur TCP

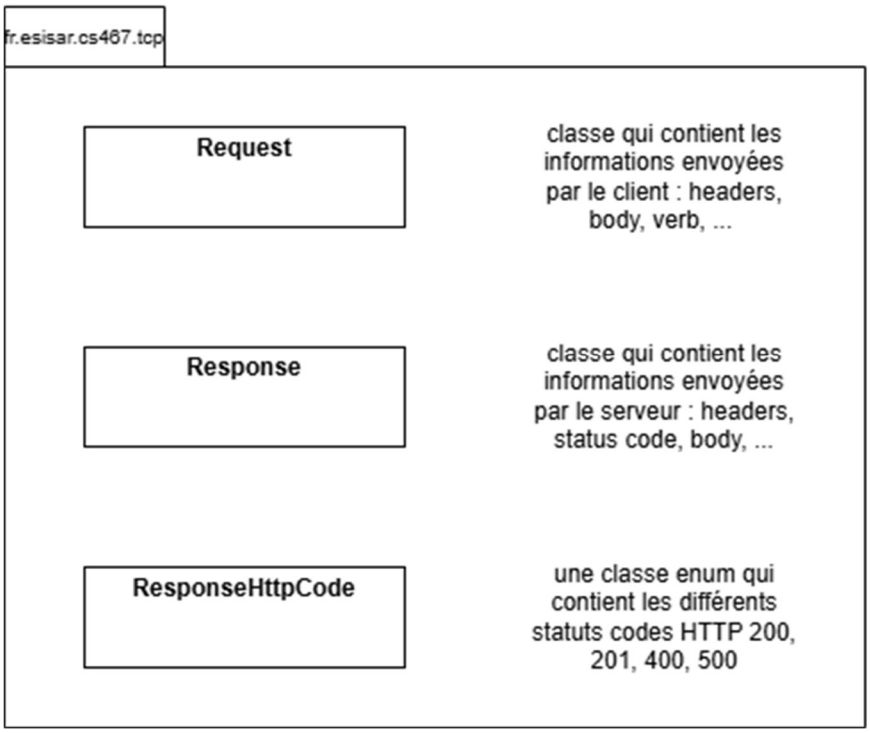
Dans l'exercice précédent, vous avez bénéficié des facilités offertes par Eclipse Vert.x. L'objectif de ce deuxième exercice est de mettre en place un serveur HTTP avec un socket TCP.



Le code que vous devez réaliser, doit être dans un package nommé fr.esisar.cs467.tcp dans le projet cs467-tp1.

Voici la liste des classes que vous allez devoir implémenter pour cet exercice.





Pour commencer, vous allez créer la classe Request, qui va servir à stocker les informations envoyées par le client. Pour vous aider à compléter ce fichier vous devez vous interroger sur quels sont les éléments que l'on retrouve dans une requête HTTP ?

```java
package fr.esisar.cs467.tcp;

import io.vertx.core.http.HttpMethod;

public class Request {

    private String httpVersion;
    private HttpMethod method;
    /*
     * ...other request fields that you know
     * getters, setters, and generic constructor
     */

    public void parseRequestHeaders(String request) {
        // TODO: headers decoding
    }

}
```

A faire :

Dans la fonction parseRequestHeaders(), vous devez décoder la partie en-tête (headers) de la requête HTTP. Pour ce faire, vous devez récupérer notamment la version de HTTP utilisée, le type de méthode (GET, POST…), les en-têtes, le chemin et le corps de la requête.

Attention, les headers dans le protocole HTTP sont insensibles à la casse, autrement dit, content-length == Content-Length, ne pas oublier de prendre en compte cette particularité dans votre application.

A faire :

Ensuite, créez l'énumération ResponseHttpCode, qui contient les différents codes HTTP, envoyé par le serveur. *Oui, c'est le même statut de code 404 Not found, que vous recevez quand vous vous trompez en allant sur un site Web non existant !*

```
package fr.esisar.cs467.tcp;

public enum ResponseHttpCode {
    OK(200, "OK"),
    CREATED(201, "Created"),
    BAD_REQUEST(400, "Bad Request"),
    NOT_FOUND(404, "Not Found"),
    INTERNAL_SERVER_ERROR(500, "Internal Server Error");

    private final int code;
    private final String description;

    ResponseHttpCode(int code, String description) {
        this.code = code;
        this.description = description;
    }

    public int getCode() {
        return code;
    }

    public String getDescription() {
        return description;
    }

    @Override
    public String toString() {
        return code + " " + description;
    }

}
```

A faire :

Vous devez poursuivre avec l'implantation de la classe Response, qui représente une réponse HTTP envoyée par le serveur. Cette classe contient toutes les informations qui définissent une réponse http : comme la version HTTP, le code réponse HTTP, les en-têtes et le corps de la réponse. La méthode toString() doit correctement retourner une chaîne de caractères conforme au protocole HTTP. Pour rappel, la première ligne contient le statut, ensuite les

entêtes (une par ligne), puis un saut-de-ligne, puis le corps de la réponse. Utilisez la classe StringBuilder[2,3] pour construire cette chaîne de caractères.

```
package fr.esisar.cs467.tcp;

public class Response {

    private String httpVersion = "HTTP/1.1";
    private ResponseHttpCode responseHttpCode;

    /*
     * other fields that you need to send in a response, any guesses?
     * getters, setters, constructor with fields
     */


    @Override
    public String toString() {
        // TODO: how does the client expect to receive the message?
        // you have to prepare the response and put the elements in order as
per the http rfc standard
        /*
         *  Response = Status-Line
                *(( general-header
                | response-header
                | entity-header ) CRLF)
                CRLF
                [ message-body ]
         */
    }
}
```

Ces classes vont vous permettre de plus facilement manipuler les requêtes HTTP entrantes et sortantes.
La classe HttpRequestHandler va rediriger les requêtes HTTP entrantes aux méthodes correspondantes. Par exemple, lorsque l'on effectue une requête GET sur le chemin /, c'est la méthode getCurrentDateTime() qui va être appelée, et c'est cette méthode qui se charge de construire une réponse HTTP (avec la classe Response), puis de l'envoyer via le socket (NetSocket).

A faire :

Complétez le code de la classe suivante pour obtenir le comportement souhaité.

```
package fr.esisar.cs467.tcp;

import java.time.LocalDateTime;
import io.vertx.core.http.HttpMethod;
```

[2] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/StringBuilder.html
[3] https://dev.to/bassaoudev/les-differences-cles-entre-string-stringbuilder-et-stringbuffer-en-java-674

```
import io.vertx.core.net.NetSocket;

public class HttpRequestHandler {

    private NetSocket socket;
    private Request request;

    public HttpRequestHandler(NetSocket socket, Request request) {
        this.socket = socket;
        this.request = request;
    }

    public void handle() {
        switch (request.getPath()) {
            case "/":
                ...
        }
    }

    private void getCurrentDateTime() {
    }

    private void postEcho() {
    }

    private void postPingPong() {
    }

}
```

A faire :

La classe qui va gérer la réception et le décodage des trames de données est HttpRequest. Dans cette classe, vous devez coder la logique dans la méthode handle(), qui est exécutée à chaque réception de trame. Dans cette méthode handle(), vous recevez les trames TCP, récupérez les données reçues via le *buffer*, pour avoir les headers et le corps.
L'attribut body permet de concaténer le corps des trames reçues, celui-ci doit être réinitialisé après avoir répondu au client body.setLength(0). L'attribut request permet de conserver les informations de la requête, il faut également le réinitialiser après avoir répondu. La variable socket représente la connexion entre le client et le serveur (voir la classe Connexion donnée ci-après), c'est à travers cette connexion que vous pouvez répondre au client.

```
package fr.esisar.cs467.tcp;

import io.vertx.core.Handler;
import io.vertx.core.buffer.Buffer;
import io.vertx.core.net.NetSocket;

public class HttpRequest implements Handler<Buffer> {
```

```
    private StringBuilder body = new StringBuilder();
    private Request request = new Request();
    private NetSocket socket;

    public HttpRequest(NetSocket socket) {
        this.socket = socket;
    }

    @Override
    public void handle(Buffer buffer) {
        // TODO: implement parsing and calling the corresponding methods
    }
}
```

La classe qui va gérer les connexions est Connection.java

```
package fr.esisar.cs467.tcp;

import io.vertx.core.Handler;
import io.vertx.core.net.NetSocket;

public class Connection implements Handler<NetSocket>{


    @Override
    public void handle(NetSocket socket) {
        socket.handler(new HttpRequest(socket));
    }
}
```

A faire :

La classe TcpHttpServerVerticle doit créer votre serveur TCP : c'est elle qui initie le socket TCP. Cette classe ressemble beaucoup à HttpServerVerticle, mais vous devez ici instancier un NetServer à la place d'un serveur HTTP.

```
package fr.esisar.cs467.tcp;

import io.vertx.core.AbstractVerticle;

public class TcpHttpServerVerticle extends AbstractVerticle {

    @Override
    public void start() {
        // TODO: create new connection similar to HttpServerVerticle
        // create a NetServer instead
    }

}
```

A faire :
N'oubliez pas de mettre à jour votre classe principale en appelant votre nouvelle classe TcpHttpServerVerticle à la place du serveur HTTP.

3.1. Testez votre nouveau serveur

Après avoir créé toutes les classes et avoir implémenté les méthodes *getCurrentDateTime()* et *postPingPong()* en TCP, exécutez votre serveur, puis testez dans un nouveau terminal, si votre programme fonctionne avec les commandes suivantes.

Tout d'abord la requête GET qui renvoie la date et l'heure, avec :

mvn test -Dtest=TcpServerTest#testGetCurrentDateTime

Ensuite la requête POST sur le chemin /ping qui renvoie pong quand le corps est bon, génère une erreur sinon, avec :

mvn test -Dtest=TcpServerTest#testPingPongSuccess
et
mvn test -Dtest=TcpServerTest#testPingPongFailure

3.2. Echo

Pour vérifier si votre serveur TCP qui gère les requêtes HTTP fonctionne bien, vous devez créer une nouvelle méthode *postEcho()* dans la classe HttpRequestHandler qui renvoie au client le même corps que celui qu'il a envoyé avec une requête POST à l'adresse « / ».

Requête :

curl -i -X POST http://localhost:8080/ --data "hi"

Réponse attendue :

```
HTTP/1.1 200 OK
Content-Type: text/plain
content-length: 2

Hi
```

Pour savoir si l'implémentation de votre serveur est robuste, testez l'envoi de données de grande taille.

Requête :

curl -i -X POST http://localhost:8080/ -d @- < lots_of_data.txt

Réponse attendue :

```
HTTP/1.1 200 OK
content-length: 38894
content-type: text/plain

12345678910111213141516171810111213141516171819999910000
```

Que se passe-t-il si on envoie un corps très long à votre serveur actuel ? Comment contourner ce problème ?
Testez si cette fonction fonctionne bien avec la commande suivante :

mvn test -Dtest=TcpServerTest#testEcho

Testez l'intégralité de votre application avec la commande suivante :

mvn test

4. Pour aller plus loin…

Dans cette dernière partie, vous devez créer une nouvelle application Java, qui a comme rôle d'être le client de votre serveur.

4.1. Créer un nouveau projet

Depuis le terminal de commande, à partir du répertoire workspace, exécutez la commande suivante, qui permet de créer un nouveau projet Maven :

mvn archetype:generate -DgroupId=fr.esisar.cs467 -DartifactId=cs467-tp1-client -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

Importez ce nouveau projet dans Eclipse (Fichier > Import > Maven > Existing Maven Projects), puis Remplacez votre fichier pom.xml par le suivant :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.esisar.cs467</groupId>
  <artifactId>cs467-tp1-client</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>cs467-tp1-client</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>io.vertx</groupId>
      <artifactId>vertx-web-client</artifactId>
      <version>4.5.12</version>
    </dependency>
  </dependencies>
```

```xml
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.11.0</version>
        <configuration>
          <source>21</source>
          <target>21</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>exec-maven-plugin</artifactId>
        <version>3.5.0</version>
        <configuration>
          <mainClass>fr.esisar.cs467.App</mainClass>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>java</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

4.2. Créer un client HTTP

Ecrivez le code du client dans la classe App.java dont le squelette est donné ci-après. Votre client doit envoyer les requêtes GET et POST données dans l'exercice précédent. La réponse de ces requêtes devra être affichée dans la console.

```java
package fr.esisar.cs467;

import io.vertx.core.Vertx;
import io.vertx.ext.web.client.WebClient;

public class App {
    public static void main( String[] args )     {
        Vertx vertx = Vertx.vertx();
        WebClient client = WebClient.create(vertx);

        // GET /
        client.get(8080, "localhost", "/")
              .send()
              .onSuccess(buffer -> {
                 ...
              });
```

```
        // POST /ping

        // POST /
    }
}
```

Testez votre application cliente avec le serveur développé dans l'exercice précédent. Exécutez votre serveur que vous avez développé dans l'exercice précèdent.
Pour exécuter votre application cliente, exécutez la commande suivante :

mvn clean exec:java