LIEGEOIS-MORALES Grégory RT0805 COMPTE RENDU DE PROJET

# Partie 1 : Environnement technologique

Toutes les actions réalisées dans ce qui suit ont été réalisé sur un OS Windows. Pour un système Linux, ce serait différent au niveau des commandes utilisées, mais le principe reste le même.

## Réaliser l’installation de Maven

Pour la procédure d’installation de Maven, nous partons ici du principe que nous n’avons aucun JDK Java. Commençons par installer Java :

* Étape 1: Installation de Java
  + Allez sur le site officiel d'Oracle ([Oracle (Java 11)](https://www.oracle.com/java/technologies/downloads/#java11)) pour télécharger le Java Development Kit (JDK). On choisit JAVA 11 pour la suite.
  + Installer le JDK.
  + Une fois l'installation terminée, nous devons définir la variable d'environnement JAVA\_HOME sur notre système pour qu'il sache où chercher le JDK. Pour ce faire, on suit ces étapes :
    - Pour le nom de la variable, entrez JAVA\_HOME. Pour la valeur, entrez le chemin d'accès du répertoire où le JDK a été installé. Par défaut, cela devrait être quelque chose comme C:\Program Files\Java\jdk-11
    - On ajoute également à la variable PATH : %JAVA\_HOME%\bin
    - Pour vérifier que Java est correctement installé et que JAVA\_HOME est correctement défini, on ouvre un cmd et on exécute les commandes :
      * java -version
      * echo %JAVA\_HOME%.

Les deux commandes doivent retourner la version de Java et le chemin d'accès du JDK, respectivement.

* Etape 2 : Installation de Maven
  + On va sur le site officiel de Maven ([Maven](https://maven.apache.org/download.cgi)) pour télécharger la dernière version de Maven. On télécharge le fichier .zip sous "Binary zip archive".
  + On extrait le contenu du fichier .zip dans le répertoire de notre choix. On a décidé de le placer ici : C:\Program Files\Java\Maven.
  + Tout comme pour Java, on doit définir la variable d'environnement M2\_HOME et ajouter Maven au PATH de notre système. Pour ce faire, suivez ces étapes :
    - On ajoute M2\_HOME aux variables d’environnement, avec la valeur, le chemin d'accès du répertoire spécifié précédemment.
    - On ajoute un élément dans la variable PATH, qui a pour attribut %M2\_HOME%\bin
    - Pour vérifier que Maven est correctement installé
      * Exécuter « mvn -version » dans un CMD. La commande doit retourner la version de Maven et les informations sur l'environnement.

Ici, Maven est installé, et on peut l’utiliser tel quel. Cependant, on va ajouter des éléments pour pouvoir utiliser JakartaEE (il nous faut un serveur d’applications, ici on choisit GlassFish). En déclarant la dépendance JakartaEE dans le fichier pom.xml du projet Maven, il téléchargera automatiquement les bibliothèques nécessaires.

(J’étais parti pour utiliser GlassFish initialement, mais j’utilise finalement le serveur d’applications TomCat.)

* Etape 3 : Installation de GlassFish, pour le serveur d’application
  + Allez sur le site officiel de GlassFish ([GlassFish](https://projects.eclipse.org/projects/ee4j.glassfish/downloads)) et on prend le dernier package.
  + Extraire le contenu du fichier .zip dans le répertoire de Java (Pour centraliser)
  + Pour faciliter l'utilisation de GlassFish, on ajoute le chemin du répertoire bin de GlassFish à notre variable d'environnement PATH.
* Etape 4 : Création d’un projet JakartEE avec Maven
  + Ouvrir un cmd et se placer dans le repertoire où l’on souhaite avoir le projet.
  + Exécuter la commande suivante pour créer un nouveau projet Jakarta EE avec Maven:
    - mvn archetype:generate -DarchetypeArtifactId=maven-archetype-webapp (Pour un projet sans interface web, on utilisera quickstart à la place de webapp).
    - Cette commande générera un nouveau projet dans le répertoire qui sera spécifié ensuite, puisque le mode interactif tourne encore. On peut ouvrir ce répertoire dans un IDE Java et commencer à développer notre application. A noter que l’on utilisera l’IDE EclipseEE.
* Étape 5 : Démarrer GlassFish et déployer l'application
  + Démarrez GlassFish en ouvrant une nouvelle fenêtre de commande et en exécutant la commande : asadmin start-domain.

Cela va démarrer le serveur. Pour l’arrêter, on utilisera stop-domain.

* + Compiler l’application avec Maven en exécutant « mvn package » dans le répertoire du projet, là où se trouve le fichier pom.xml. Cela générera un fichier .war dans le répertoire target (Attention, war car on a créé un projet Web, sinon c’est un jar de base)
  + Pour déployer l'application sur GlassFish, on utilise la commande asadmin de GlassFish. Dans la fenêtre de commande, on exécute la commande suivante :
    - « asadmin deploy chemin\_vers \_war\_file »

en remplaçant « chemin\_vers \_war\_file » par le chemin d'accès complet du fichier .war que l’on a généré avec Maven.

* + GlassFish déploiera l'application et on pourra y accéder en ouvrant un navigateur et en naviguant vers <http://localhost:8080/myapp/>.

A noter qu’il est également possible d’utiliser l'interface d'administration web de GlassFish pour gérer nos applications déployées. On peut y accéder en naviguant vers <http://localhost:4848/>.

Nous pouvons désormais passer à la suite du projet : aucune autre installation ne devrait être nécessaire (en dehors des packages nécessaires dans le fichier pom.xml, qui seront installer automatiquement).

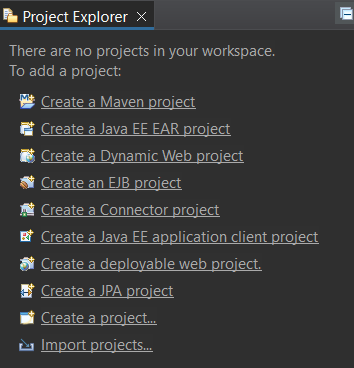
Tous les codes de chaque parties sont disponibles dans les dossiers calculatrice, programmeTV, servlets pour les projets d’exemple ;et activite pour le projet.

## Faire un projet d’exemple avec 1 projet sans dépendance

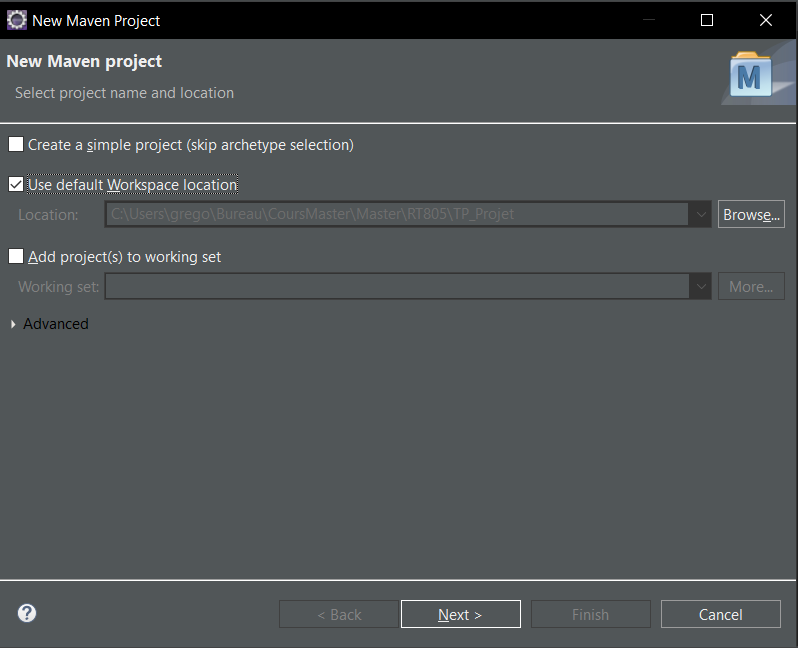
On lance la commande :

mvn archetype:generate -DarchetypeArtifactId=maven-archetype-quickstart

Sinon, dans l’IDE EclipseEE, on peut faire nouveau projet. C’est ce que l’on va utiliser ici :



« Create a Maven Project »



Pour un projet simple, on peut passer l’« archetype selection » (projet de base donc).

On va réaliser une simple calculatrice, utilisant les symboles +-\*/. Le code est disponible dans le dossier calculatrice.

Au moment de la génération du fichier jar, il a fallu ajouter au fichier pom :

<manifest>

<mainClass>calculatrice.Calculatrice</mainClass>

</manifest>

Dans la partie build, plugins,plugin.

Ensuite on peut faire la commande « mvn clean compile package » et on obtient le fichier jar lançable :



Des erreurs d’arrondis surviennent à cause de la représentation binaire des nombres, mais ce n’est pas le problème ici, on peut passer à l’utilisation d’une dépendance dans notre projet.

## Faire un projet d’exemple avec au moins une dépendance

Pour ce projet, nous allons utiliser Jsoup, afin de donner le programme TV. Après plusieurs tests sur différents site, j’ai utilisé le site <https://tv-programme.com/> qui me paraissait le plus simple à utiliser.

Le code est relativement simple lorsque la page internet est bien structurée. Après une analyse de cette dernière, j’ai pu extraire les informations que je souhaitais.

package programmeTV;

import org.jsoup.Jsoup;

import org.jsoup.nodes.Document;

import org.jsoup.nodes.Element;

import org.jsoup.select.Elements;

public class ProgrammeTV {

public static void main(String[] args) {

try {

// Récupérer le document HTML à partir de l'URL

Document doc = Jsoup.*connect*("https://tv-programme.com/programme-tv-tnt/").get();

// Sélectionner l'élément contenant les informations

Element containerFluid = doc.selectFirst("body div.container-fluid");

Element row = containerFluid.selectFirst("div.row");

Element colLg9 = row.selectFirst("div.col-lg-9");

// Récupérer les informations dans chaque ligne (tr) du tbody

Elements trElements = colLg9.select("tbody tr");

for (Element trElement : trElements) {

// Récupérer le nom de la chaîne

Element channelElement = trElement.selectFirst("td > a.stretched-link > img");

String channel = channelElement.attr("alt");

System.***out***.println("Chaîne : " + channel);

// Récupérer l'heure de début

Element startTimeElement = trElement.selectFirst("big");

String startTime = startTimeElement.text();

System.***out***.println("Heure de début : " + startTime);

// Récupérer le titre de l'émission

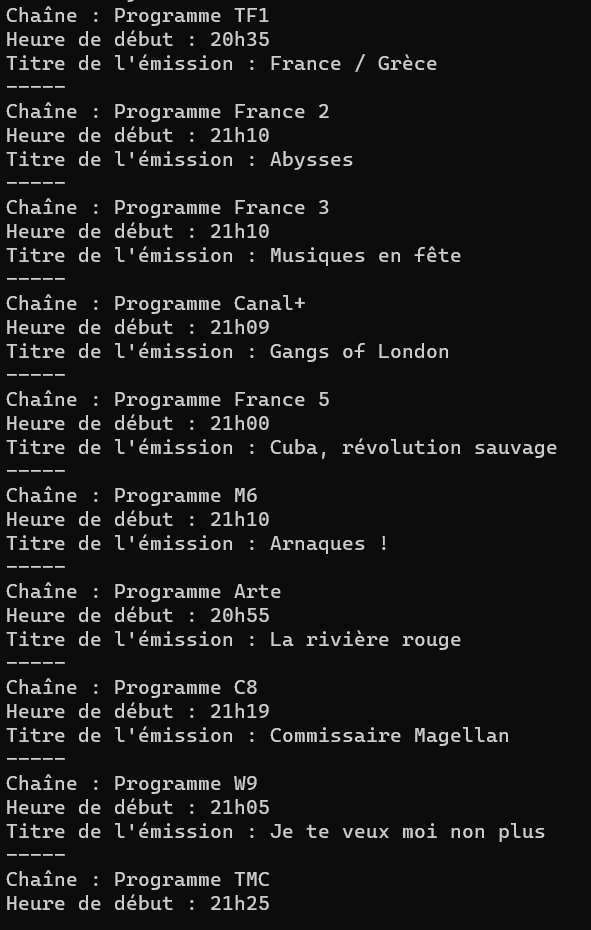
Element titleElement = trElement.selectFirst("h2 > a.stretched-link > strong");

String title = titleElement.text();

System.***out***.println("Titre de l'émission : " + title);

System.***out***.println("-----");

}

 } catch (Exception e) {

e.printStackTrace();

}

}

}

Pour lancer le Jar, il faut normalement inclure toutes les dépendances.

On peut aussi faire le choix de générer le Jar avec toutes les dépendances, et ce, en ajoutant le plugin maven-assembly-plugin dans le fichier pom.xml.

## Faire un projet d’exemple en Jakarta EE, exploitant JSP (ou JSF) et servlet

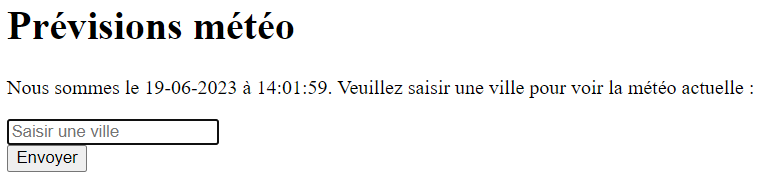
Pour l’utilisation de servlet, nous allons réaliser une page qui présente la météo actuelle dans un lieu précis. Pour cela, on utilise le site :

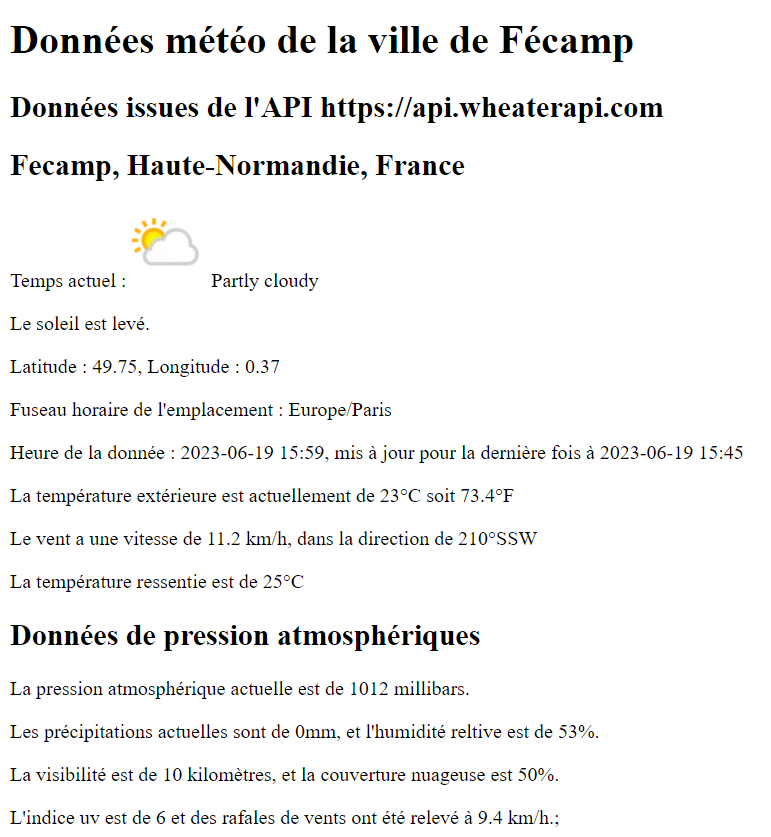
https://api.weatherapi.com/v1/current.xml?key="+***cleAPI***+"&q="+***ville***+"&aqi=no

Après la création d’un compte, je peux obtenir ma clé d’Api.

Via un formulaire, on demande la saisie de la ville à l’utilisateur. Cela est accessible à l’adresse :

<http://localhost:8080/servlet/>



La servlet envoi une requête post vers la page /servlet/weather, en envoyant la ville dans les données. Pour cela, on a alors deux servlets, et deux pages JSP : une pour le formulaire, avec l’affichage de la date et de l’heure actuelle, et une pour afficher les résultats de notre requête.

L’API nous répond en envoyant des données XML. On s’occupe de le parser pour l’affichage.

Au niveau du travail effectué, après réalisation des codes, on a placé le fichier war obtenu dans le dossier webapps de notre serveur d’application Tomcat.

# Partie 2 : Développement

# Sujet 1 : Tracking sportif

## Présentation des idées de développement

Pour développer cette application, il est nécessaire d’avoir 3 principaux acteurs :

* ApplicationSportive, qui sera une application Java qui simulera l’activité et qui l’enverra en temps réels au ServeurSportif. Au lancement de l’activité (choix du sport parmi une liste), l’utilisateur est amené à s’identifier (Par le biais d’un choix d’un compte existant, ou pas sa création).
* ServeurSportif, qui sera le récepteur des informations émises par ApplicationSportive. Ces données seront stockées dans un fichier XML (à la fin de l’activité), dont on précisera plus loin la structure. On aura un fichier XML par utilisateur de l’ApplicationSportive.
* Visualisateur, qui va lire et interpréter le fichier XML contenant l’historique d’activité de l’utilisateur. La comparaisons / statistiques sur les activités pourront être implémenté en utilisant XPATH. En revanche, cela ne se fera qu’en affichage d’historique et non en temps réel.

A noter que le ServeurSportif est un serveur WEB (ici en local). L’ApplicationSportive communiquera avec le ServeurSportif à l’aide de servlet Java.

Chacun de ces morceaux s’appuieront sur des classes utilitaires, telles que les classes java Activite, ou PositionsGPS.

## Fichiers XML, la structure :

On aura un fichier XML par utilisateur (pour éviter de le surcharger). J’ai réalisé un schéma avant que tous les fichiers respectent la structure imposée. Ce schéma est disponible dans le dossier ./src/main/resources/ sous le nom Schema.xsd.

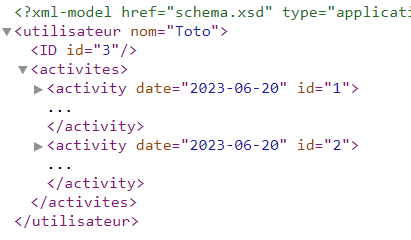
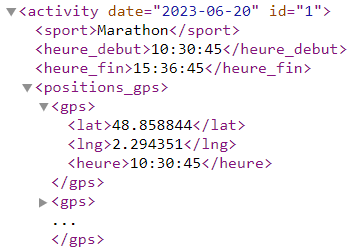
### Le schéma

Pour ce schéma, j’ai réalisé plusieurs types personnalisés pour les éléments suivants :

* L’heure, avec un format hh :mm :ss
* La date, avec un format yyyy-mm-jj
* Les coordonnées GPS, sous la forme DD.DDDDDD
* La vitesse, sous la forme DD.DD
* La distance, sous la forme DD.DD

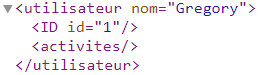
### Les fichiers XML

Ensuite, on retrouve les différentes balises, avec parfois des attributs imposés. Le tout se retrouve ensuite sous la forme qui suit :



Le champ <ID> est un champ indiquant combien d’activités ont été pratiqués, mais c’est aussi un champ qui permet de marquer une activité. En effet, j’ai réalisé une fonction qui me permet de supprimer une activité, par rapport à son ID qui est unique (attribut id).

De même, l’attribut id de la balise ID est incrémenté à chaque nouvelle activité. Ainsi, il est possible de repérer absolument toutes les activités.

A noter que lorsqu’un utilisateur rentre ses données, un fichier XML est créer à son nom : Si le fichier existe déjà, c’est qu’un autre utilisateur a le même nom, et il est invité à changer.

### Le choix du XML

Pourquoi un fichier XML par utilisateur plutôt qu’une base de données ? Ici c’est une application scolaire qui sera utilisé par pas plus de 5 personnes. Il n’est donc pas nécessaire d’utiliser l’outil puissant de la base de données pour ces cas ci.

En revanche, pour une application déployée, il serait nécessaire de le faire.

### Présentation des codes utilitaires

La vitesse moyenne sera calculée comme la moyenne des vitesses entre 2 coordonnées GPS. Ce traitement sera fait sur le serveur d’application, au moment de l’enregistrement : on a la distance parcouru entre les 2 coordonnées GPS, ainsi que le temps nécessaire à ce parcours.

En revanche, on a besoin d’au moins 2 points GPS, puis on aura X1\_X2, X2\_X3 …. Xn-1\_Xn

Au niveau des échanges entre les acteurs, nous nous enverrons des messages en XML.

### Simulation de déplacement

Pour simuler une activité, j’utilise le code suivant :

//une simule une activité pour une durée donnée

public void simulerActivite(String sport,double lat,double lng,LocalDate date,LocalTime heure,long duree) throws TransformerException {

Activite activite=Activite.*demarrerActivite*(sport, date, heure, lat, lng);

//une fois l'activité démarré, on simule les déplacements

double bearing,distance;

LocalTime endTime=heure;

double[] newCoord = {lat,lng};

//on va faire 100 points de repère

do {

Random random = new Random();

bearing = random.nextDouble() \* 180;// entre 0 et 180 deg

distance = 0.5+random.nextDouble()\*0.5;//entre 0.5 et 1 km

newCoord=PositionsGPS.*getNewCoordinates*(newCoord[0], newCoord[1], bearing, distance);

int minutesToAdd = 3+random.nextInt(8); //next int entre 0 et 7 Nombre de minutes à ajouter entre 3 et 10 minutes du km

endTime = endTime.plusMinutes(minutesToAdd);

if(endTime.isBefore(heure.plusMinutes(duree)))

activite.ajouterPointRepere(newCoord[0], newCoord[1], endTime);

}while((endTime.isBefore(heure.plusMinutes(duree))));

//on arrête l'activité

activite.terminerActivite(newCoord[0], newCoord[1], endTime);

//on enregistre l'activité

this.ajouterActivite(activite);

}

Comme je voulais quelque chose de fonctionnel, j’ai simulé des déplacements sur une carte à partir des données GPS, à partir d’une orientation (bearing), et d’une distance. Je vais dans cette direction et j’obtiens une nouvelle paire de coordonnées. A partir de celle-ci, je dis que j’ai parcouru cette distance en un temps aléatoire.

C’est ce qui me permets d’avoir ces données : 

## Architecture de l’Application

L’application possède cette architecture (après un mvn clean) :

. Main et Visualisateur ont été des classes de tests. Elles sont inutiles ici.

. Activite.java représente sous forme de classe ce qu’est une activité.

. On retrouve 4 servlets associées aux 4 fichier JSP. La servlet DemarrerActivite ne s’appuie pas sur un JSP. Elle va simplement être appelé et rediriger ensuite le flux vers infoUtilisateur.jsp.

. ManipulationXML.java sera la classe pour ajouter/supprimer/créer des utilisateurs/des activités.

Dans les ressources, on retrouve des fichiers XML qui sont importés au début dans l’application. Mais sur le serveur, on retrouve ceci :

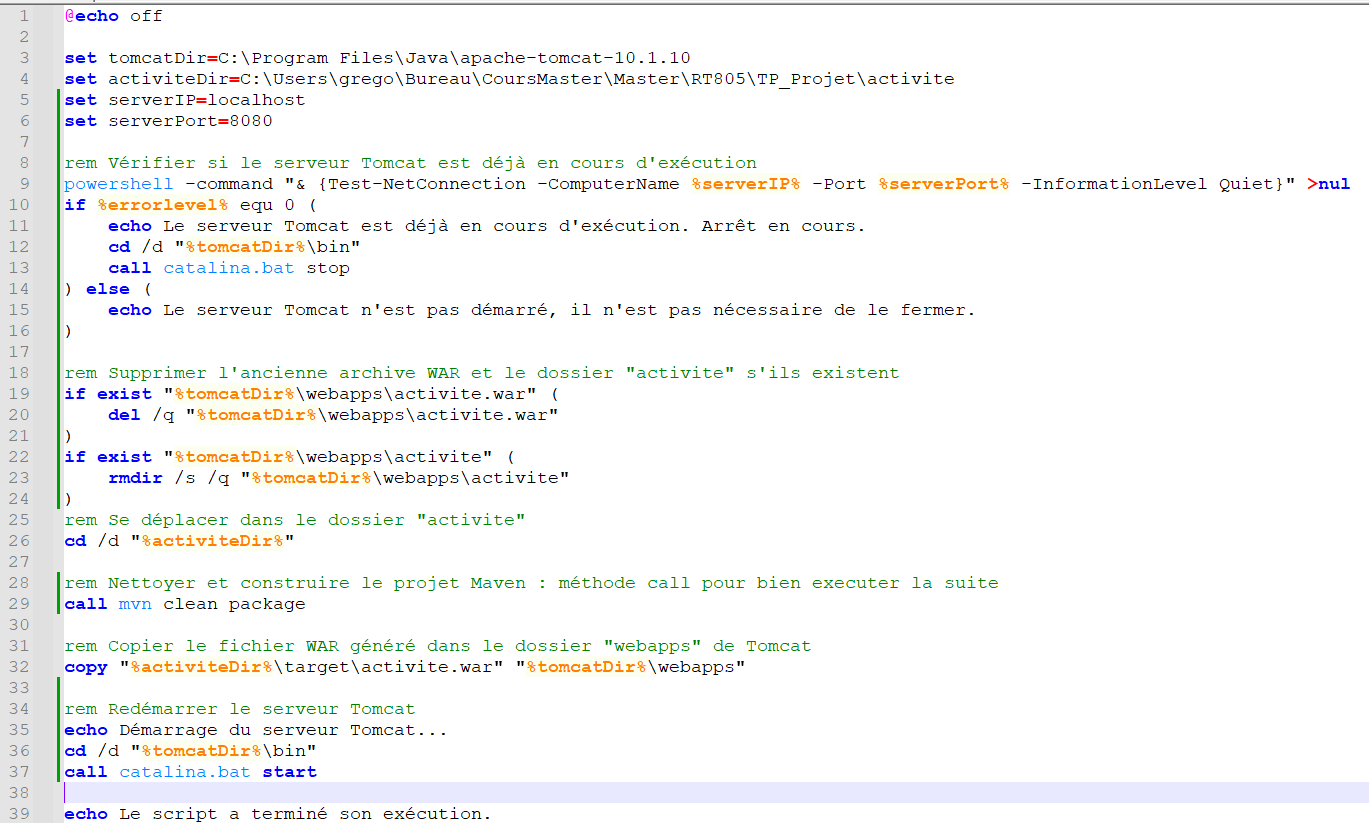
Les fichiers se sauvegardes XML se font dans le dossiers classes du répertoire WEB-INF.

Au déplacement de l’archive WAR, le serveur d’applications Tomcat a réalisé cette structure.

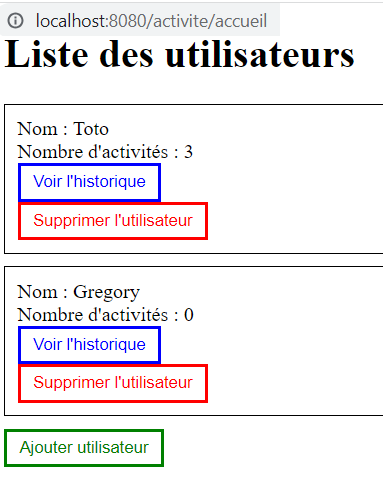
## Le traitement des informations

Une fois tous les petits morceaux de codes utilitaires réalisé, je vais pouvoir réaliser les pages internet qui vont être visible côté client. Tout le backend est résumé précédemment. On va maintenant s’occuper de la partie frontend.

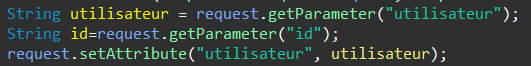
Pour générer le fichier .war et le déplacer au bon endroit dans le repertoire /webapps/ de Tomcat, j’ai réalisé un petit script afin de réaliser automatiquement cette opération. On coupe le serveur Tomcat s’il est démarré, on génère le fichier .war, on le déplace en supprimer les occurrences précédentes, et on relance le serveur :



## Présentation du site WEB :

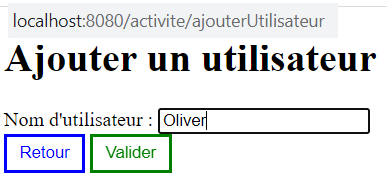
Sur la page d’accueil, on retrouve les Sportifs qui ont déjà une session :

Ici, il n’y a pas de login et de mot de passe. On garanti simplement l’unicité de l’utilisateur avec son nom (On est sensible à la casse, l’utilisateur Grégory n’est pas le même que grEgOrY).

A noter que pour conserver la « session », entre chaque page, j’ai effectué :

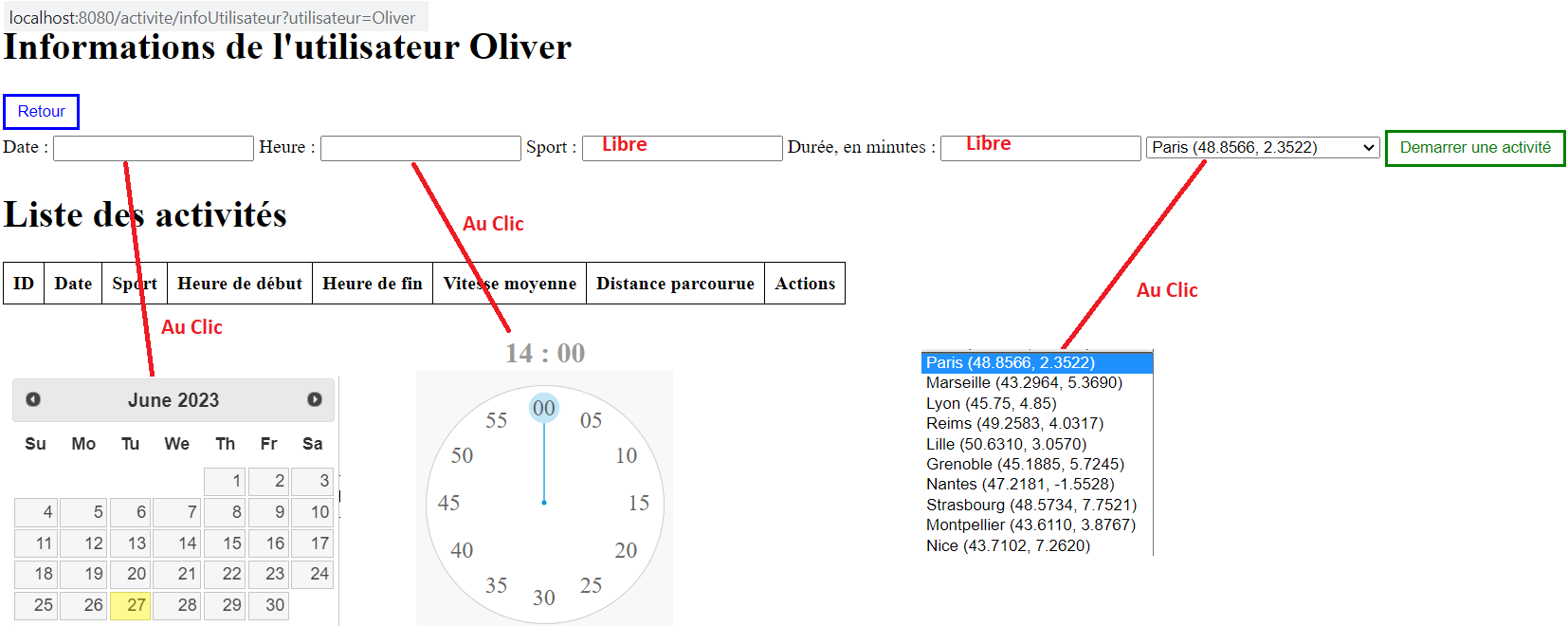
C’est forcément assez lourd et peu adapté ici donc, car entre chaque page JSP, je recharge complétement le fichier XML, au lieu de le conserver d’une étape à l’autre.

Je n’ai malheureusement pas eu le temps de réfléchir à une autre solution beaucoup plus propre.

Concernant l’ajout d’utilisateur, on arrive sur la page :

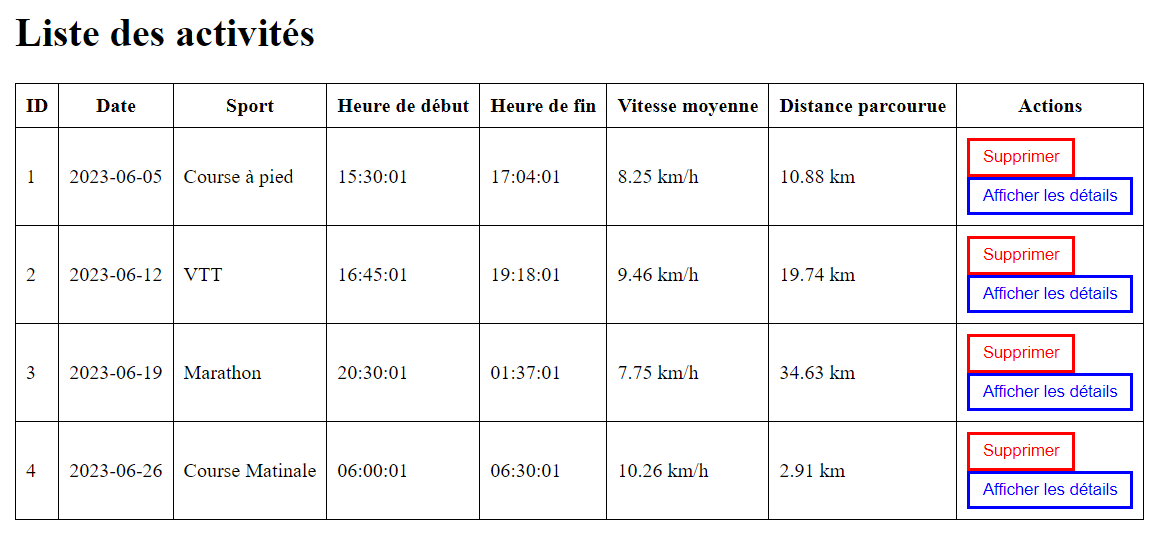
Une fois l’utilisateur ajouté, par exemple Olivier, il apparait sur la page d’accueil, et on peut maintenant commencer à saisir des activités.

Une fois encore, je ne suis pas parti du principe que j’allais lancer l’activité, simuler un mouvement, et arrêter l’activité. J’ai simplement décidé de saisir la date de l’activité, l’heure de début de celle-ci, le nom du sport, ainsi que la durée pratiquée et le point de départ (Un choix parmi une liste). Cela se présente comme suit :



Une fois l’activité choisie (génération aléatoire), on peut démarrer l’activité. Elle se lance et s’arrête dès qu’elle trouve un point de coordonnées qui a dépassé le temps demandé.

Il n’y a pas de gestion d’erreur ici. Par exemple, si l’utilisateur saisie « TOTO » dans la durée en minutes, on a une erreur côté serveur. C’est quelque chose qui devrait être fait.

L’utilisateur peut ensuite voir une récapitulatif des activités effectuées, et constaté à quel point il est grand sportif :

Il peut décider de supprimer une activité qu’il ne souhaite plus voir. C’est ici que l’intérêt de l’ID a lieu, afin de pouvoir rentre unique chaque activité.

Il peut également décidé d’afficher le détail. A partir des coordonnées GPS présent dans le fichier, et de l’outil JavaScript Leaflet, il est possible de visualiser le trajet effectué :



## Conclusion :

Au final pour ce projet, on respecte plus ou moins les consignes, à la différence qu’il n’y a pas de réception en temps réel des coordonnées de l’utilisateur.

Pour cela, j’avais pensé à des envois régulier de message en XML, avec à chaque fois les coordonnées GPS associées au lieu d’envoi, l’heure d’envoi, mais aussi des « token » précisant si c’est un début d’activité, une fin d’activité, ou une activité en cours.

Cependant, hormis le point sur la comparaison des activités, et les statistiques dessus (par exemple le nombre de kilomètres parcouru en tout, le temps d’activités etc.), le reste a été implémenté.

Pour ces statistiques, on pourrait également réaliser une somme des kilomètres parcourus ou du temps d’activités, puisque ce sont des informations que l’on a dans la vue utilisateur.

Tous les codes, ainsi que les archives, seront disponibles sur git, à l’adresse :

<https://github.com/Halakazam/ProjetRT0808>

Il seront également déposé à l’endroit disponible pour le rendu du projet.