MASTER DÉVELOPPEMENT LOGICIEL

UNIVERSITÉ TOULOUSE III PAUL SABATIER

M1 DL

# Réalisation d'un service web « Toulouse Vélo/Piéton »

## Interopérabilité des Applications et introduction aux Web Services

MOUGEOT Matteo
LACHERAY Benjamin

2015/2016

# Table des matières

# Structure du projet

L'utilisation d'Apache Maven nous a permis de gérer simplement les dépendances de notre projet et également permis de le diviser en trois modules.

## Module projet-root

Ce module est un simple conteneur pour les deux modules principales du projet : le module métier et le module web-services.

## Module projet-ws

Ce module est celui qui implémente nos différents web-services, implantés en suivant l'approche Contract-First à l'aide du framework Spring-WS. Il est composé de 5 Endpoints qui correspondent aux 5 stories décrient dans le sujet. Ce module est dépendant de projet-metier.

## Module projet-metier

Le module projet-metier contient le cœur du projet : la consommation des différentes APIs REST : ArcGis, JCDeceaux, Open Weather Map, Open Street Map, ainsi que tous les algorithmes pour traiter ces informations.

# Détails sur projet-metier

## Package “mdl.iaws”

Le package principale du module s’appelle **mdl.iaws**. Il contient toutes les classes de service des API (une classe par API consommée) ainsi que la classe **Services** qui regroupe toutes les méthodes pour répondre aux user stories demandé.

Nous avons également divisé ce module en deux sous-packages : **types** et **weatherUtil**.

## Package “types”

Le package **types** contient les structures de donnée du projet.
Il contient une classe **GeoCoordinate** qui représente une coordonnée géographique ainsi qu’une classe **Station** qui représente une station de vélo et qui permet de connaître sa position géographique, son nombre de vélos disponibles ainsi quel le nombre de places disponibles dans la station.

## Package “weatherUtil”

Le package **weatherUtil** contient plusieurs types relatifs au climat et aux véhicules, ainsi que des constantes relatives au climat et à la vitesse d’un vélo ou d’un piéton. Il contient également une méthode permettant d’associer une vitesse en fonction du type de véhicule et du type de précipitation.

Les différents types énumérés:

- Le type des précipitations **TypePrecipitation** : NO, RAIN, SNOW
- Le risque de chaussée mouillée **TypeShodRisk** : NUL, FAIBLE, MODERE, FORT
- Le type des véhicules **TypeVehicule** : PIED, VELO

Les différentes constantes utilisées pour la vitesse sont récapitulées dans le tableau ci-dessous. Elles permettent de prendre en compte la météo dans les calculs de temps des itinéraires, et sont données en km/h.

| TypePrecipitation / TypeVehicule | NO | RAIN | SNOW |
|---|---|---|---|
| PIED | 5 | 5 | 4 |
| VELO | 18 | 16 | 12 |

Une map permet représenter ce tableau et une méthode en donne un accès facile.
Exemple : getVitesse(TypeVehicule.VELO, TypePrecipitation.RAIN)

Le choix de ces vitesses n'a pas été fait au hasard, elles sont issues de "recherches", trouvées entre autre à ces différents liens :

- http://www.wesaw.it/2013/05/quelle-est-la-vitesse-moyenne-a-velo/
- http://sante.lefigaro.fr/mieux-etre/sports-activites-physiques/marche/quels-types-marche

## Classe OpenStreetMap

Cette classe permet d'interroger l'API du même nom. Elle permet de convertir une adresse en une coordonnée géographique. La méthode interrogeant l'API s'appelle **addressToGeoCoor**, elle prend en paramètre une adresse (String) et retourne une coordonnée (**GeoCoordinate**).
La consommation de l'API se fait en XML.

## Classe ArcGis

Cette classe permet d'interroger l'API du même nom. Elle permet de calculer la distance entre deux coordonnées géographique ou plusieurs à la fois. La méthode **distance** prend deux coordonnées (**GeoCoordinate**) en paramètre et retourne la distance (float en km) entre les deux. La méthode **length** prend une coordonnée G en paramètre ainsi qu'un tableau de coordonnées et retourne un tableau de distances entre G et chaque point du tableau de coordonnées.
La consommation de cette API se fait en JSON (XML non disponible).

## Classe OpenWeatherMap

Cette classe permet de récupérer des informations sur la météo d'une adresse donnée en interrogeant l'API du même nom. Les ressources consommées se font en XML.

## Classe JCDecaux

Cette classe permet de récupérer des informations sur les stations de vélos en libre-service en utilisant l'API du même nom. Les ressources consommées se font en JSON.

## Classe Service

La classe **Service** regroupe toutes les méthodes pour répondre aux user stories demandé. C'est cette classe qui sera utilisé pour utiliser nos services. Le constructeur par défaut ne prend pas de paramètre mais la ville peut-être stipulé (ville où JCDecaux est présent).

# User Stories

## User story 1a

*"A partir d'une adresse, le service doit pouvoir afficher les 3 stations de vélos non-vides les plus proches, avec le nombre de vélos restants à ces stations."*

Le cœur de cette story est la classe **JCDecaux** grâce à la méthode **getNearestNonEmptyStations**. La classe interroge tout d'abord l'API de JCDecaux, et récupère toutes les stations disponibles dans la ville. On peut ensuite calculer les distances de chaque station par rapport au point courant grâce aux classes ci-dessous :

- **OpenStreetMap** : pour récupérer les coordonnées correspondant à l'adresse courante.
- **ArcGis** : pour calculer les distances à vol d'oiseau entre l'adresse courante et chaque station.

Il nous reste plus qu'à trier les stations, on garde les 3 plus proches tout en éliminant les stations vides et on les retourne.

## User story 1b

*"A partir d'une adresse, le service doit pouvoir afficher les 3 stations de vélos non-complètes les plus proches, avec le nombre de places disponibles pour déposer un vélo."*

Cette story est implémentée de la même manière que la précédente. Seul le filtrage change, c'est maintenant les stations pleines qui sont éliminées. Elle est accessible via la méthode **getNearestNonFullStations**.

Note : en réalité les stories 1a et 1b utilisent toutes les deux la méthode **getNearestStationsConditional** qui permet de retourner les X stations en fonction d'un filtre (non vide ou non plein).

## User story 2

*"A partir d'une adresse de départ, le service indiquera le risque de chaussée mouillée en fonction de la pluviométrie observée dans les 3 dernières heures."*

C'est la méthode **riskShod** de notre classe **Services** qui satisfait cette user story. Elle utilise notre classe **OpenWeatherMap** pour accéder aux types et au niveau de précipitation.
En fonction de ces informations, notre méthode **riskShod** retourne le risque (classe enum **TypeShodRisk**).

## User story 3

*"A partir d'une adresse de départ et d'une adresse d'arrivée, le servie calculera une estimation du temps de trajet à pied en tenant compte de l'incidence de la météo sur la vitesse du piéton"*

C'est la méthode **computeTravelWalk** de notre classe **Services** qui satisfait cette user story. Elle prend en paramètre l'adresse de départ et l'adresse d'arrivée. On convertie ensuite ces adresses en coordonnées grâce à **OpenStreetMap**, on calcule la distance entre les deux grâce à **ArcGis**, et enfin on calcule le temps grâce à la bonne constante de vitesse en fonction des précipitations retournées par **OpenWeatherMap**.

## User story 4

*"A partir d'une adresse de départ de d'une adresse d'arrivée, le service calculera une estimation du temps de trajet en vélo en tenant compte du temps du trajet pour rejoindre à pied la station de vélo la plus proche possédant un vélo libre, du temps de parcours en vélo jusqu'à la station de vélo non-complète la plus proche de l'adresse d'arrivée ainsi que du temps pour finalement rejoindre cette adresse d'arrivée à pied. La météo aura une incidence sur la vitesse de déplacement."*

C'est la méthode **computeTravelTime** de notre classe **Services** qui satisfait cette user story. Elle utilise la classe implémentant l'user story précédente pour calculer le temps nécessaire pour rejoindre la première station, et pour aller de la station d'arrivée à l'adresse d'arrivée. Pour le temps en vélo de la première station à la deuxième, on s'y prend de la même manière qu'à la story précédente, sauf que le type véhicule est VELO et plus PIED.

Note : en réalité l'user story 3 et 4 utilisent toutes les deux la méthode **computeTravelType** qui calcule le temps d'un point A à un point B en fonction du type de véhicule et de la météo.

# Détails sur projet-ws

## Implémentation

Ce module est celui qui implémente nos différents web-services. Il est dépendant de projet-métier puisqu'il se sert de notre classe **Services**. Il est composé de 5 classes Endpoints qui correspondent aux 5 stories décrient dans le sujet. Pour chaque classe, un fichier XSD a été créé et contient le schéma d'une requête et celui d'une réponse. Chaque requête est analysée en suivant la structure de ce schéma, et chaque réponse est construite en suivant ce schéma en fonction des données retournées par **Services**.

## WSDL

Le WSDL regroupant tous les services a été généré. D'abord dynamiquement via tomcat puis ensuite gelé. Il est visible dans WEP-INF/wsdl/final.wsdl.

# Politique de tests

Nous avons effectué des tests unitaires dans le module **projet-metier** et des tests d'intégrations dans le module **projet-ws**.

## Politique de qualité

Pour le module projet-metier, nous avons utilisé différents plugins :

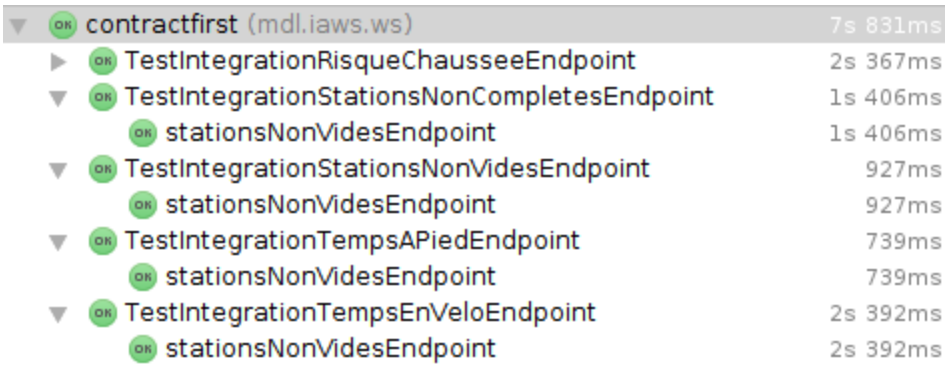
- Utilisation de **Checkstyle** : 0 erreur
- Utilisation de **Cobertura** : 100% de couverture (pour les classes testées)
- Utilisation de **FindBugs** : 0 bugs

## Tests unitaires

Il y a des tests unitaires pour les classes **GeoCoordinate**, **Station** et pour l'intégralité du package **weatherUtil**, les tests passent avec une couverture de 100%. Il n'y a pas de tests pour les classes interrogeant les APIs car les résultats ne sont pas prévisibles.

### Tests d'intégrations

Il y a un test d'intégration par classe Endpoint créé soit 5 tests.
Pour chaque test on vérifie que la réponse de l'Endpoint est cohérente par rapport à la structure attendue (XSD). Les 5 tests passent.




# Difficultés rencontrées

Nous avons eu beaucoup de problèmes pour faire fonctionner la partie Spring et faire passer les tests d'intégrations. Nous avons finalement réussi en recherchant des informations dans le cours, dans le TP3 et sur internet (dont beaucoup de forum).
Nous avons eu des difficultés avec l'application-context, notre bean n'était pas valide car la classe utilisé avait un constructeur avec paramètre. Après avoir compris que l'erreur venait d'ici, nous avons résolu le problème en mettant un constructeur par défaut sans paramètre.

# Conclusion

Notre solution arrive à un aboutissement de 100% par rapport à ce qui a été demandé en étant fonctionnel. Nous avons également généré le WSDL.
Cependant il serait intéressant d'améliorer le code pour gérer plus profondément les erreurs en cas d'adresse non valide (géré actuellement avec des sys.err) ou d'API non disponible.