LP SIL – Concepteur-Développeur
en Environnement Distribué
2017 - 2018

IUT Robert Schuman
Institut universitaire de technologie
Université de Strasbourg

DOCUMENTATION DU PROJET

ARCHITECTURE MATERIELLE & LOGICIELLE

GROUPE « OVERKILL LEGACY »

COUTY Robin
CROCHEMORE Thomas
KOEHL Dylan
PÉQUIGNOT Ysaline
WEIBEL Lucas

Table des matières

API HTTP serveur

Architecture générale

L'API du projet a été réalisée avec Java 1.8. Afin de gérer les dépendances multiples de ce projet, Maven a été utilisé en tant que gestionnaire de dépendances, mais aussi de compilation.

Pour respecter les conventions de Maven au sujet de l'architecture d'un projet utilisant cette technologie, celui-ci est découpé en deux parties principales : d'un côté, le code source dans `src.main`, et de l'autre les tests dans `src.test`.

L'architecture de la partie `src.main` est disponible à droite. Un découpage du code source a été réalisé selon les fonctionnalités des classes faisant partie du projet. On retrouve notamment des catégories apparaissant dans la plupart des projets comme les DAOs (Data Access Objects) ou les services, ceci indiquant la séparation des couches « métier », « accès aux données », « traitement » et « présentation ».

On retrouve aussi un répertoire spécial `resources`, qui contient les fichiers de configuration Hibernate et les propriétés Log4J. Tous les fichiers contenus dans ce répertoire sont automatiquement chargés dans l'archive lors du déploiement.

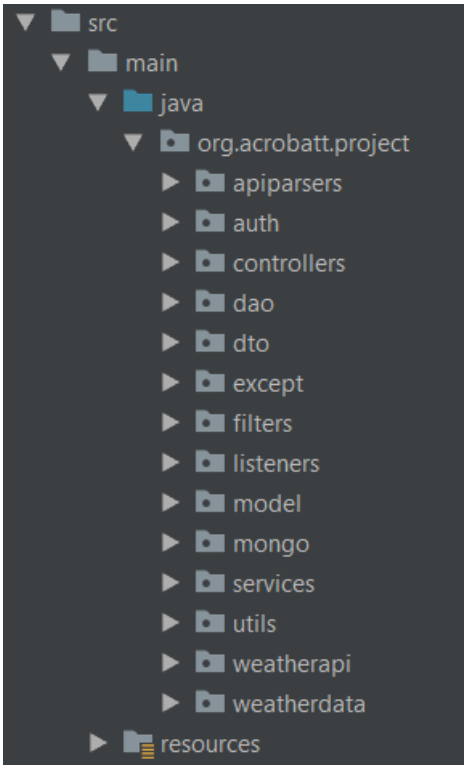


Figure 1 - Architecture générale

Un descriptif général du contenu de chaque package est disponible ci-dessous (les packages marqués avec ★ seront détaillés plus précisément dans la section suivante) :

`apiparsers`★	Contient les classes servant à normaliser les données MongoDB sous MySQL
`auth`	Contient les classes nécessaires à l'authentification d'un utilisateur
`controllers`★	Contient les classes de ressource Jersey/JAX-RS, c'est-à-dire les routes REST/HTTP
`dao`★	Contient les classes gérant l'accès aux données de la base MySQL
`dto`	Contient les classes gérant les Query Strings (semi-DTOs)
`except`	Contient des exceptions personnalisées pour certains cas spécifiques
`filters`	Contient un filtre CORS pour les réponses serveur
`listeners`	Contient un utilitaire chargeant la configuration Hibernate au premier appel à l'API
`model`★	Contient le modèle de données
`mongo`	Contient des classes de configuration du client MongoDB
`services`★	Contient les classes faisant le lien entre les contrôleurs et les DAOs
`utils`	Contient des classes utilitaires diverses facilitant le travail des développeurs
`weatherapi`	Contient le portage du script Python utilisé au début du projet (voir plus loin)
`weatherdata`	Contient une classe servant à charger les données MongoDB dans MySQL

Détails des packages principaux

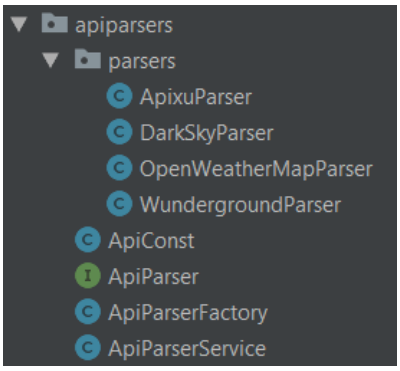


Figure 2 - Package "apiparsers"

Package « apiparsers »

Ce package contient l'un des cœurs de l'application, à savoir les « *parsers* », des classes de normalisation des données MongoDB vers MariaDB/MySQL. On retrouve le design pattern « *factory* », qui renvoie le *parser* à utiliser selon certains détails prédéterminés.

Ce choix technologique n'est cependant pas extensible facilement. Des détails concernant cela sont disponibles dans le dossier-annexe « Compléments de projet ».

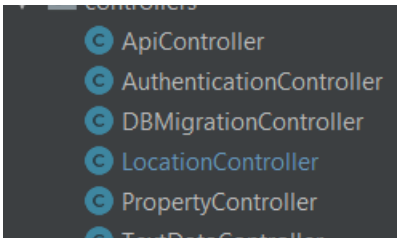


Figure 3 - Package "controllers"

Package « controllers »

Ce package contient les classes de ressources Jersey/JAX-RS, qui sont par extension les différents points de connexion pour les clients. Chaque contrôleur sert un nombre et un type défini de ressources, ce dernier étant indiqué dans son nom.

Le contrôleur « `DBMigrationController` » est spécial, dans le sens où il gère la migration des données de base nécessaires au bon fonctionnement de l'application.

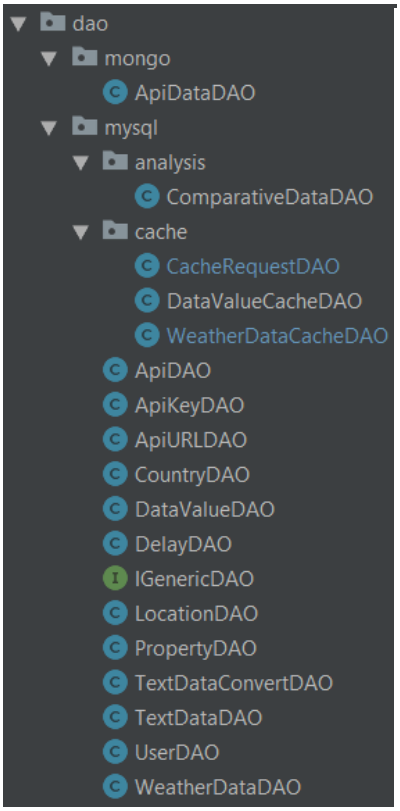


Figure 4 - Package "dao"

Package « dao »

Ce package est relativement conséquent en nombre de classes, puisqu'il couvre la totalité des fonctions métier du projet. On retrouve plusieurs sections :

- `mongo` : contient une seule classe servant à récupérer les données brutes de MongoDB sous un format temporaire.
- `mysql` : contient la plupart des classes du package parent. Ces classes gèrent toutes les tables du modèle commun de base, sans prendre en compte les tables supplémentaires pour stocker des résultats de calculs ou des données de cache.
- `mysql.analysis` : contient une classe servant à récupérer les données de calcul stockées sous MariaDB/MySQL.
- `mysql.cache` : contient les classes gérant les données du cache, principalement pour le widget.

Tous ces DAOs sont des implémentations de l'interface `IGenericDAO`, qui contient les prototypes de fonction du CRUD.

Package « model »

Ce package contient le cœur de l'application, à savoir le modèle métier utilisé dans la base MariaDB/MySQL. On retrouve le même découpage que dans le package précédent :

- `mongo` : contient le format temporaire des données brutes provenant directement de la base MongoDB.
- `mysql` : contient toute la structure de tables utilisée par Hibernate pour la persistance de données. C'est ici que le modèle de données a été construit.
- `mysql.analysis` : contient des classes concernant les tables supplémentaires utilisées pour stocker des résultats de calculs statistiques. Celles-ci sont principalement utilisées pour la partie « exploration des données » des clients web et mobile.
- `mysql.cache` : contient les classes reliées au cache.

Toutes ces classes utilisent Hibernate pour la persistance, exclusivement au travers des annotations fournies par l'ORM. Ceci nous permet de mieux visualiser les interactions entre les différentes entités et de ne pas alourdir inutilement le projet, contrairement à l'utilisation de fichiers XML.

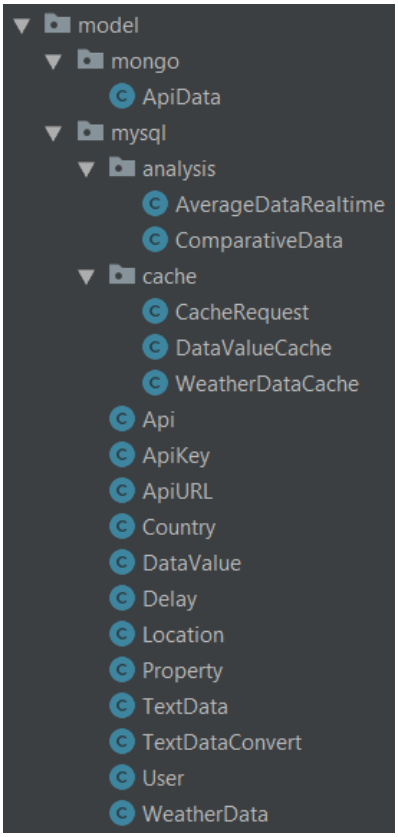


Figure 5 - Package "model"

Package « services »

Ce package contient les classes faisant le lien entre les contrôleurs et le modèle de données. On peut observer le découpage suivant :

- `auth` : contient le service d'authentification. Celui-ci utilise l'algorithme de chiffrement et de salage jBCrypt et les utilitaires de chiffrement Java intégrés pour fonctionner.
- `rawdata` : contient plusieurs classes-modules à transformer les données normalisées en un format JSON lisible pour la partie client. Une classe abstraite dicte le format de ces modules.
- `rawdata.modules` : contient les modules cités ci-dessus.

L'un des modules est isolé (Isolated_WidgetDataModule), car il ne respecte pas exactement le format prédéfini dans la classe abstraite, et a donc besoin d'un traitement spécifique.

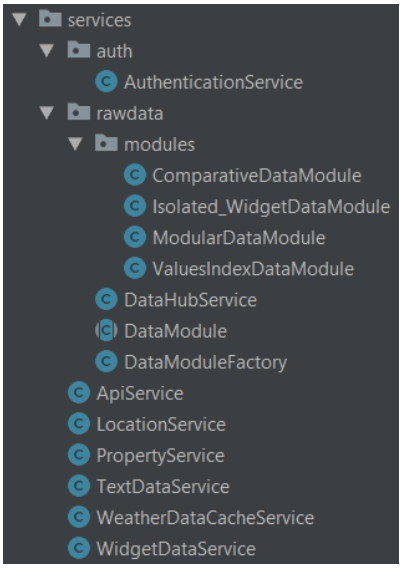
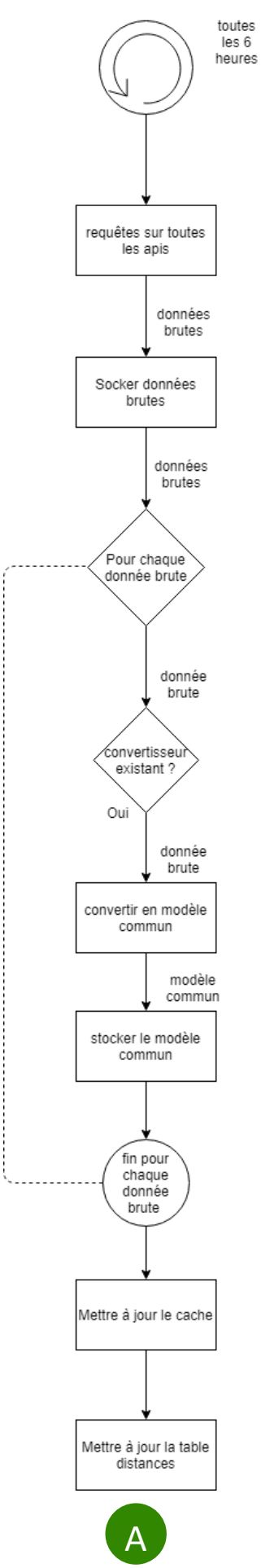
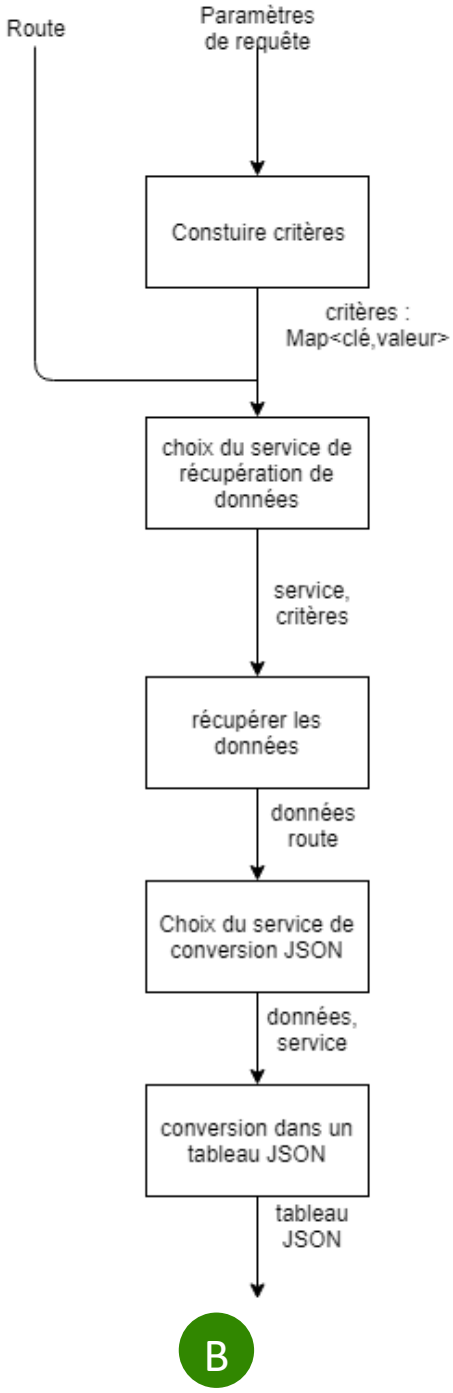
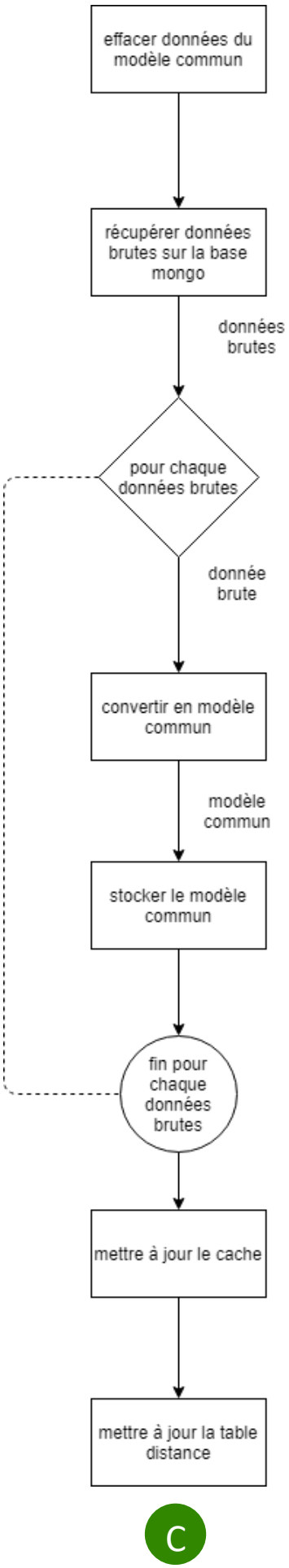


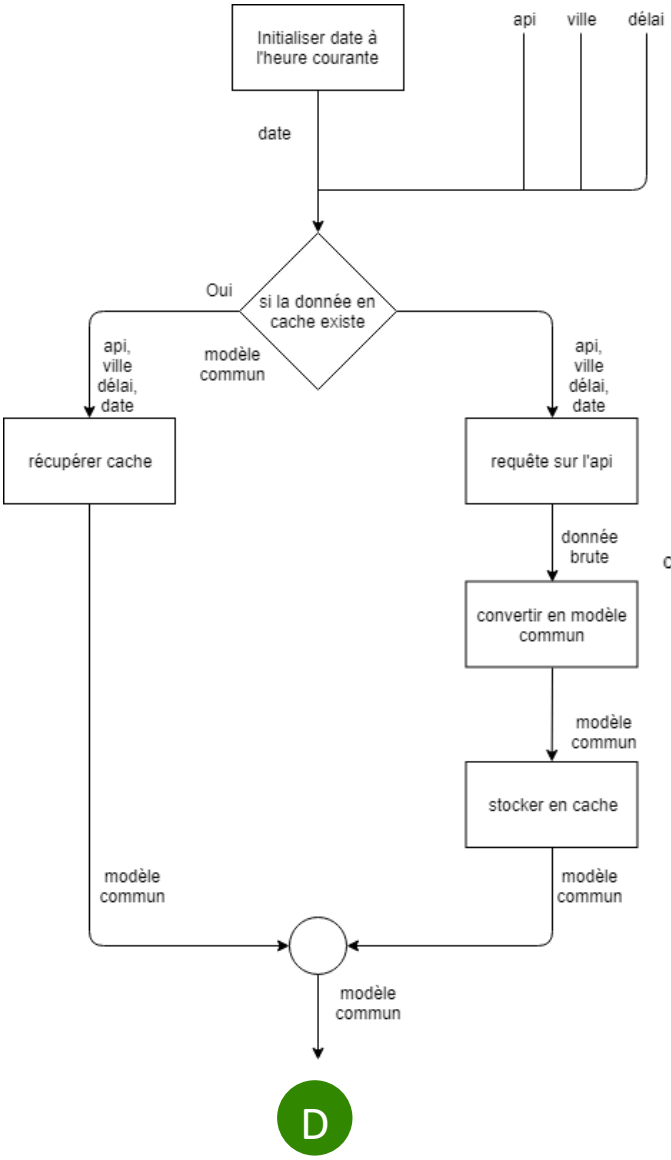
Figure 6 - Package "services"

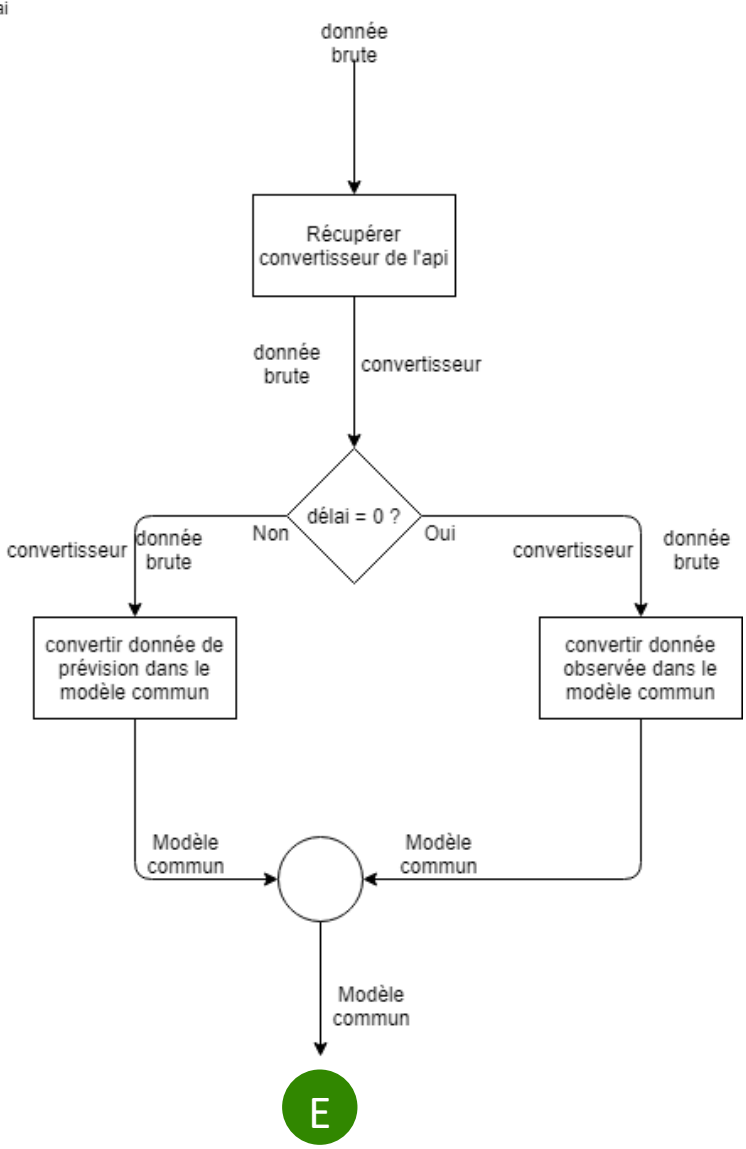
Processus


toutes les 6 heures
requêtes sur toutes les apis
données brutes
Socker données brutes
données brutes
Pour chaque donnée brute
donnée brute
convertisseur existant ?
Oui
donnée brute
convertir en modèle commun
modèle commun
stocker le modèle commun
fin pour chaque donnée brute
Mettre à jour le cache
Mettre à jour la table distances
A


Route
Paramètres de requête
Constuire critères
critères : Map<clé,valeur>
choix du service de récupération de données
service, critères
récupérer les données
données route
Choix du service de conversion JSON
données, service
conversion dans un tableau JSON
tableau JSON
B


effacer données du modèle commun
récupérer données brutes sur la base mongo
données brutes
pour chaque données brutes
donnée brute
convertir en modèle commun
modèle commun
stocker le modèle commun
fin pour chaque données brutes
mettre à jour le cache
mettre à jour la table distance
C





A	Processus de chargement immédiat des données (« compute »)
B	Processus de chargement et de transformation des données en JSON (« mapping »)
C	Processus de chargement des données MongoDB vers MySQL (« backup »)
D	Processus de chargement et de mise à jour du cache (« cache »)
E	Processus de transformation des données MongoDB vers MySQL (« parsers »)

Description des échanges clients / serveur

Liste des échanges client/serveur principaux

Préfixe de route `/weatherdata`

Échanges bruts

GET `/rawdata` : renvoie des données selon les critères passés en paramètre. Prend en compte des prévisions "brutes" uniquement, hors calculs supplémentaires.

- Paramètres de requête : `datetime`, `delay` (+ 1 au choix entre `api`, `city` et `property` pour le dernier paramètre)
- Exemple de réponse :

```
{
    "column_headers": ["APIXU", "OPENWEATHERMAP", "WUNDERGROUND"],
    "data": [
        { "label": "Température", "unit": "°C", "values": [12,11,9] },
        { "label": "Pression", "unit": "millibar", "values": [1025,1015,1020] },
        { "label": "Humidité", "unit": "%", "values": [45,null,62] }
    ]
}
```

Échanges statistiques

GET `/distance` : renvoie, sur une période donnée, l'écart entre les valeurs prévues et les valeurs observées par les différents prévisionnistes.

- Paramètres de requête : datetime, datetime_to, delay, city, api, property
- Exemple de réponse :

```
{
    "column_headers": ["Prévision", "Observé", "Distance"],
    "series_type": ["Date/heure", "Distance"],
    "data": [
        { "label": "25-02-2018_12:00", "values": [12, 11, 1] },
        { "label": "25-02-2018_18:00", "values": [10, 7.5, 2.5] },
        { "label": "26-02-2018_00:00", "values": [9, 6, 3] }
    ]
}
```

GET `/score` : renvoie un score calculé en fonction des écarts prévisionnels. La formule utilisée est **score = 1 / moyenne des distances.**

- Paramètres de requête : datetime, datetime_to, delay, city, property
- Exemple de réponse :

```
{
    "column_headers": ["Score"],
    "series_type": ["API", "Score"],
    "data": [
        { "label": "APIXU", "values": [0.5208] },
        { "label": "OPENWEATHERMAP", "values": [0.1715] },
        { "label": "WUNDERGROUND", "values": [0.2524] },
        { "label": "DARKSKY", "values": [0.4855] },
    ]
}
```

GET `/eval` : renvoie des indices de confiance calculés sur les 30 derniers jours selon les critères passés à la fonction. Ces indices sont classés par ordre descendant et calculés comme suit : **indice = score / somme des scores**

- Paramètres de requête: exactement 3 dans: delay, city, property, api
- Exemple de réponse :


```
{
    "column_headers": ["Indice de confiance"],
    "series_type": ["API", "Indice de confiance"],
    "data": [
        { "label": "APIXU", "values": [0.65] },
        { "label": "OPENWEATHERMAP", "values": [0.23] },
        { "label": "WUNDERGROUND", "values": [0.12] }
    ]
}
```


GET `/evalDetailed` : même fonctionnement que /eval, mais avec des données supplémentaires en retour (score et somme des scores)

- Paramètres de requête : exactement 3 dans: delay, city, property, api
- Exemple de réponse :


```
{
    "column_headers": ["Score", "Total des scores", "Indice de confiance"]
    "series_type": ["API", "Indice de confiance"],
    "data": [
        { "label": "APIXU", "values": [0.55, 1.19, 0.65] },
        { "label": "OPENWEATHERMAP", "values": [0.36, 1.19, 0.23] },
        { "label": "WUNDERGROUND", "values": [0.28, 1.19, 0.12] }
    ]
}
```


Échanges prévisionnels

GET `/widget` : renvoie des prévisions optimisées utilisant les résultats de /eval comme poids pour les calculs. Les données pris en compte sont celles datant des trente derniers jours.

- Paramètres de requête : city, api (+ un/tous/aucun dans property, delay)
- On peut renseigner plusieurs « property ». Exemple : `&property=Température&property=Humidité`
- Exemple de réponse :


```
{
    "city" : "Illkirch",
    "dateRequest" : "27/04/2018",
    "delay" : [6,12,24,72],
    "series_type": ["Date/heure", "Prévisions optimisées"],
    "properties" : [
        {
            "label" : "Température",
            "rawdata" : [22,25,23,19],
            "unit": "°C"
        },
        {
            "label" : "Humidité",
            "rawdata" : [14,17,15,3],
            "unit": "%"
        },
        {
            "label" : "Couverture nuageuse",
            "rawdata" : [0,1,2,1],
            "unit": "(1 - 3)" },
        {
            "label" : "Pression",
            "rawdata" : [1000,970,1110,1987],
            "unit": "millibar"
        }
    ]
}
```

`GET /values-index` : renvoie un tableau selon les critères renseignés, avec pour chaque api, sa prévision à l'heure courante et son indice de confiance calculé sur les 30 derniers jours

- Paramètres de requête : city , property, delay
- Exemple de réponse :

```
{
    "series_type": [ "API", "Valeur"],
    "column_headers": ["API", "Température(°C)", "Indice de confiance"],
    "datas": [
        { "data": [ 22.1,0.47101226], "label": "APIXU" },
        { "data": [20.4,0.29676393],  "label": "DARKSKY" }
    ]
}
```

Préfixe de route /

Échanges utilisateurs (Fonctionnel côté api, non implémenté en front end)

`POST /signup` : créé un compte avec le nom de compte et le mot de passe fournis, puis authentifie la personne avec `/signin`

- Exemple de corps de requête :

```
{
    "username": "couty",
    "password": "youhou"
}
```

`POST /signin` : authentifie la personne avec ses identifiants, et renvoie un token JWT à fournir au serveur

- Exemple de corps de requête :

```
{
    "username": "couty",
    "password": "youhou"
}
```

- Exemple de réponse :

```
{
    "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c"
}
```

Préfixe de route `/apis`

`GET /withcfg` : renvoie toutes les apis avec leur configuration

- Exemple de réponse :


```
[
    {
        "id": 3,
        "lastUsedKey": 5,
        "name": "WUNDERGROUND",
        "reqPerHour": 10,
        "reqPerDay": 500,
        "forecastUrl": {
            "id": 6,
            "url_ex":
"http://api.wunderground.com/api/{key}/geolookup/conditions/forecast/q/{country.name}/{city}.json",
            "forecast": true
        },
        "realtimeUrl": {
            "id": 5,
            "url_ex":
"http://api.wunderground.com/api/{key}/conditions/q/{country.name}/{city}.json",
            "forecast": false
        }
    },
    {...}
]
```


`GET /list` : renvoie la liste des noms de chaque api

- Exemple de réponse :


```
[
    "DARKSKY",
    "WUNDERGROUND",
    "OPENWEATHERMAP",
    "APIXU",
    "WEATHERBIT"
]
```

Préfixe de route `/properties`

`GET /` : renvoie toutes les propriétés

- Exemple de réponse :

```
[
    { "id": 10, "name": "Poussière",   "unit": "(0-8)" },
    { "id": 9,  "name": "Brouillard",  "unit": "(0-3)" },
    { "id": 3,  "name": "Humidité",    "unit": "%" },
    { "id": 2,  "name": "Pression",    "unit": "millibar"},
    { "id": 1,  "name": "Température", "unit": "°C"}
]
```

`GET /{id}` : renvoie la propriété d'identifiant {id}

- Exemple de réponse :

```
{
    "id": 10,
    "name": "Poussière",
    "unit": "(0-8)"
}
```

`GET /list` : renvoie la liste des noms de propriété

- Exemple de réponse :

```
[
    "Poussière",
    "Brouillard",
    "Orage",
    "Neige",
    "Pluie",
    "Nuages",
    "Vitesse du vent",
    "Humidité",
    "Pression",
    "Température",
    "Vent",
    "Divers"
]
```

Préfixe de route `/textdata`

`GET /` : renvoie toutes les données textuelles des APIs, avec leur intensité et leur propriété associée

- Exemple de réponse :


```
[
    {
        "id": 54,
        "text": "Vent modéré",
        "intensity": 2,
        "property": {"id": 11,"name": "Vent","unit": "(0-6)" }
    },
    {
        "id": 55,
        "text": "Vent fort",
        "intensity": 3,
        "property": {"id": 11, "name": "Vent","unit": "(0-6)" }
    },
    {
        "id": 56,
        "text": "Bourrasques",
        "intensity": 4,
        "property": {"id": 11,"name": "Vent","unit": "(0-6)"}
    },
    { … }
]
```


`GET /{id}` : renvoie la donnée textuelle d'identifiant id, avec son intensité et sa propriété associée

- Exemple de réponse :


```
{
    "id": 1,
    "text": "Ensoleillé",
    "intensity": 0,
    "property": { "id": 5, "name": "Nuages", "unit": "(0-3)"}
}
```


`GET /list` : renvoie la liste des noms des données textuelles

- Exemple de réponse :


```
[
    "Fort brouillard",
    "Forte brume",
    "Brouillard modéré",
    "Brume modérée",
    "Léger brouillard",
    "Légère brume",
    "Pas de brouillard",
    "Forts orages et grêlons",
    "Forts orages et neige",
    "Forts orages et pluie",
    "Forts orages"
]
```

Préfixe de route `/locations`

`GET /` : renvoie toutes les villes avec leurs pays :

- Exemple de réponse :

```
[
    {
        "id": 4,
        "city": "Moscow",
        "latitude": 55.7522,
        "longitude": 37.6156,
        "country": {"id": 4,"name": "Russia", "code": "ru" }
    },{...}
]
```

`GET /coord` : renvoie la ville aux coordonnées indiquées :

- Paramètres de requête : latitude, longitude
- Exemple de réponse :

```
{
    "id": 2,
    "city": "Gaborone",
    "latitude": -24.6545,
    "longitude": 25.9086,
    "country": {"id": 2, "name": "Botswana", "code": "bw" }
}
```

`GET /{id}` : renvoie la ville d'identifiant id

- Exemple de réponse :

```
{
    "id": 2,
    "city": "Gaborone",
    "latitude": -24.6545,
    "longitude": 25.9086,
    "country": {"id": 2, "name": "Botswana", "code": "bw" }
}
```

`POST /` : crée une nouvelle ville dans la base

- Exemple de corps de requête :

```
{
    "city": "Gaborone",
    "latitude": -24.6545,
    "longitude": 25.9086,
    "country": "Botswana"
}
```

- Exemple de réponse :

```
{
    "id": 2,
    "city": "Gaborone",
    "latitude": -24.6545,
    "longitude": 25.9086,
    "country": {"id": 2, "name": "Botswana", "code": "bw" }
}
```

`GET /list` : renvoie tous les noms de ville

- Exemple de réponse :
- `["Moscow", "Helsinki", "Strasbourg"]`

Responsabilités des différentes entités

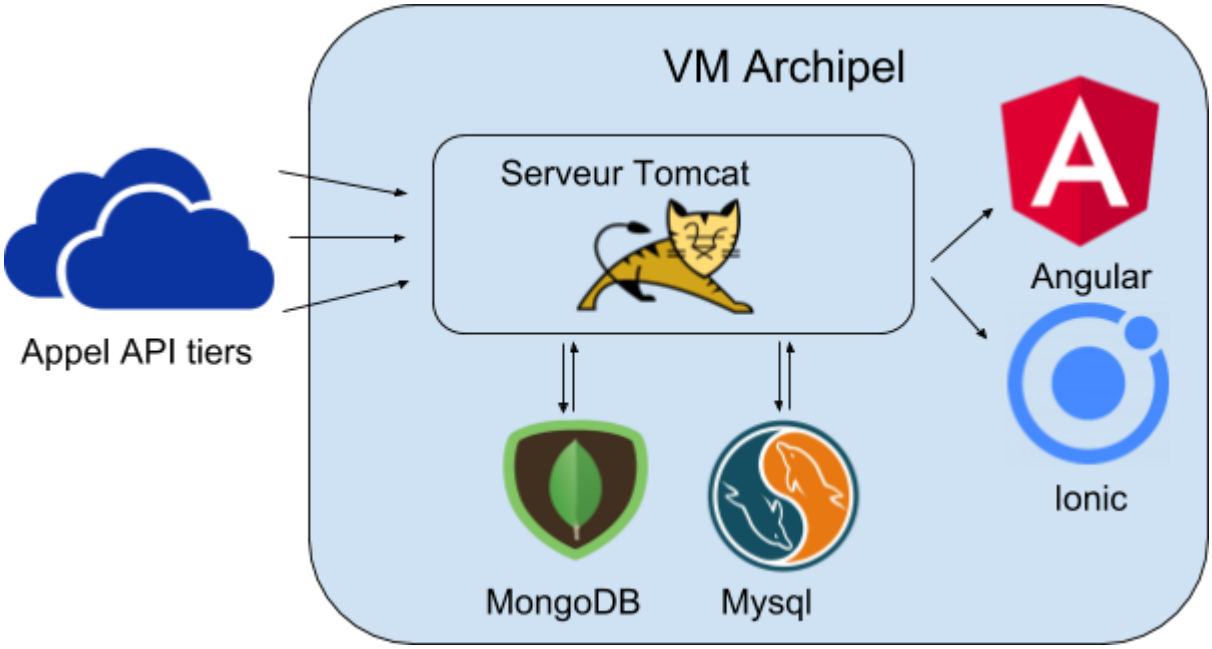


Figure 7 - Schéma général des entités impliquées dans le projet

VM Archipel : serveur virtuel hébergeant sous Debian les serveurs applicatifs nécessaires au projet.

MongoDB est un système de gestion de bases de données non relationnels. Il nous permet de stocker les données bruts et hétérogènes issus des apis tiers afin de pouvoir réimporter l'ensemble des données par les parseurs vers la base relationnelle MySQL.

MySQL est un système de gestion de bases de données relationnel. Il nous permet de stocker les données transformées dans un modèle de données commun, c'est-à-dire normalisées.

Le serveur **Tomcat** héberge et implémente le fonctionnement de l'API en tant que tel. Il procède aux requêtes vers les APIs tierces, stock les données brutes et les importe dans le modèle commun. Il fournit des routes sous les conventions de l'architecture REST pour requêter les données du modèle commun.

Angular est un framework Javascript de type Single Page Application qui interroge l'API Java pour afficher les données sur les pages web.

Ionic s'appuie sur Angular pour la partie web et sur Cordova pour la partie native. L'application gère la consultation des données météo.

Technologies / bibliothèques utiliées côté client

Frameworks

Les deux technologies utilisées coté client sont Angular 5 et Ionic 3. Il s'agit de deux frameworks Javascript.

Angular permet de développer des sites de type « Single Page Application ». Il a l'avantage d'être déjà éprouvé, maintenu par une large communauté et d'être pratiqué par plusieurs membres de notre équipe en entreprise.

Ionic est une implémentation d'Angular et de Cordova adaptée pour développer des applications mobiles multi-plateformes. L'avantage de coupler ces deux technologies est de pouvoir mutualiser certains composants et de les transférer d'un sous-projet à l'autre.

Librairies / plugins

La première librairie implémentée dans notre projet est Angular Material, une implémentation de Material Design, une librairie de composants graphiques suivant certaines règles et bonnes pratiques en matière de design et permettant d'accélérer le développement sur ce type de projets. Elle fournit de nombreux composants comme différents types d'entrée, des tableaux, des arborescences ou des menus.

Lors de notre phase d'exploration de données, nous avons implémenté un composant permettant de représenter un jeu de données à deux dimensions, utilisable en tant que tableau ou graphique. Pour cela, nous avons ajouté la librairie Charts.js sous la forme de son package NPM « ng2 Charts » compatible Angular et Ionic.

Pour l'implémentation d'une carte, nous avons utilisé la librairie Frugal Map qui permet d'implémenter facilement des cartes de la librairie Leaflet sur Angular et d'y ajouter des points comme les villes prises en compte par notre API.

Pour certains composants nous avons aussi utilisé la librairie Clarity qui est une alternative à Material avec un design plus adapté à certaines contraintes.

Technologies / bibliothèques utilisées côté serveur

Hébergement

Tomcat 9.0.0M26 : serveur d'hébergement d'applications. L'application est exportée sous le format Web application Archive (.war) et compilée depuis l'IDE ou directement depuis la ligne de commande avec Maven.

Stockage

MySQL/MariaDB 10.1.3 : fork de MySQL rétro-compatible utilisé pour stocker les données normalisées

MongoDB 6.2 : base NoSQL utilisée pour les données brutes provenant directement des APIs des différents prévisionnistes

Code

Java 1.8 (JavaEE 6) avec comme bibliothèques :

- Hibernate 5 : ORM servant de pont entre le code et la base MySQL/MariaDB.
- Jersey 2 (JAX-RS) : bibliothèque REST orientée WebServices
- Maven : gestionnaire de dépendances et packages
- MongoDB Java Driver : driver MongoDB pour Java
- Mongojack : wrapper pour le driver MongoDB
- JUnit 5 "Jupiter" : bibliothèque de tests
- Log4J 2 : service de logging interne
- Java JWT (jjwt) : implémentation Java des JsonWebTokens
- jBCrypt : algorithme de chiffrement unidirectionnel par salage BCrypt adapté pour Java

Analyse

SonarCloud : utilitaire d'analyse de code en ligne (SonarQube)

SonarLint : analyse statique de code directement depuis l'IDE (plugin IntelliJ IDEA)

JaCoCo (Java Code Coverage) : outil de reporting pour tests unitaires et d'intégration.

Maintenance

StatusCake (branché à Slack) : outil servant à vérifier l'état d'un serveur distant (en ligne ou hors-ligne)

Annexe - Table des illustrations

Annexe – Index