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

IUT Robert Schuman
Institut universitaire de technologie
Université de Strasbourg

DOCUMENTATION DU PROJET

DOSSIER DE CONCEPTION

GROUPE « OVERKILL LEGACY »

COUTY Robin
CROCHEMORE Thomas
KOEHL Dylan
PÉQUIGNOT Ysaline
WEIBEL Lucas

Table des matières

Rappel du contexte du projet

L'objectif de ce projet est d'avoir une application donnant pour un lieu donné et une date donnée des prévisions météo les plus justes possibles. Le calcul des propriétés prédites est basé sur la comparaison des prévisions de différentes APIs : plus elles sont fiables, plus leur coefficient de confiance est élevé. Plus ce coefficient est élevé, plus il pondère l'estimation.

Pour ce faire, nous récupérons les informations de différentes API. Le serveur repère et insère ces caractéristiques dans un modèle générique, qui est utilisé directement par les interfaces client. Dans le cadre de notre projet, le côté serveur est réalisé en langage Java, le côté front web avec la librairie JavaScript Angular et l'application mobile avec le framework Ionic, utilisant Angular et Cordova.

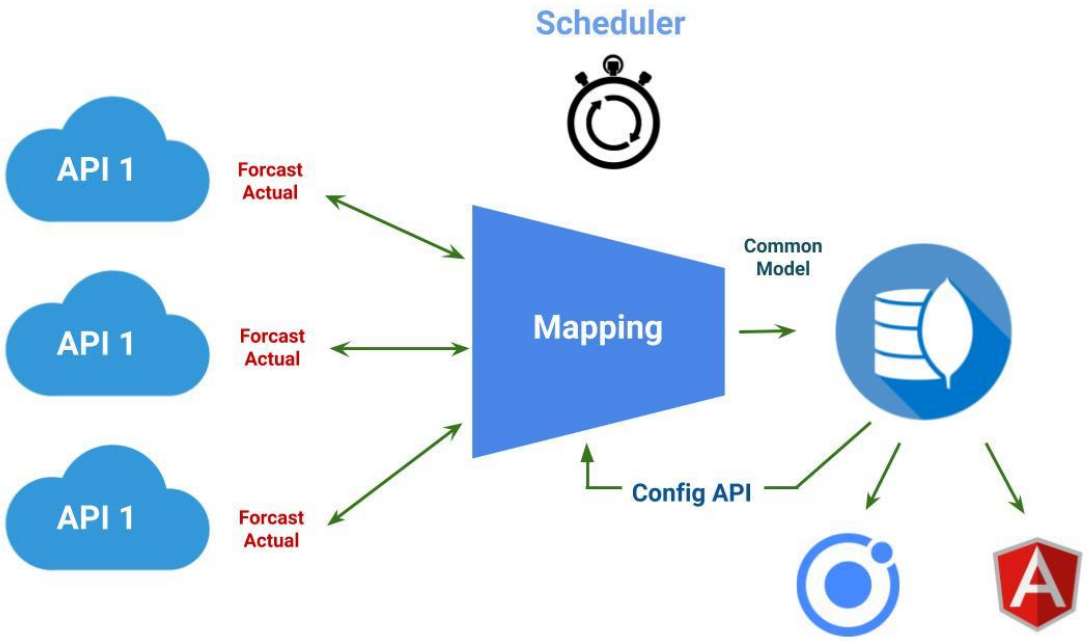

Figure 1 - Première version du schéma de l'application

Modélisation des données dynamiques côté client

Les données dynamiques côté client sont issus des providers qui appellent les routes du back-end. La première partie du projet étant lié à l'exploration de données, nous sommes partis sur une structure permettant de ramener un tableau à deux dimensions, que nous avons fait varier selon les usages. La structure de base est donc la suivante :

```
{
    "column_headers": [ ... ],
    "series_type": [ ... ],
    "data" : [
        { "label" : "...", "values" : [ ... ] },
        ...
    ]
}
```

Ainsi, on peut utiliser ce modèle pour afficher tout type de données à deux dimensions pour l'exploiter au travers de tableaux ou graphiques.

Exemple : Tableau des prévisions pour les différentes APIs pour une ville donnée

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

Ainsi nous avons ajouté pour certains contextes des paramètres supplémentaires comme l'unité ou le type de données en x ou en y. Pour l'exploration de données nous avons trois routes, exploration des distances, des scores et les évaluations.

Exemple de l'exploration des distances pour une période donné, une ville et une API. Nous avons donc la valeur de la prévision, la valeur observée et la distance avec la valeur mesurée.

```
{
  "column_headers": [ "Prévision","Observé","Distance" ],
  "series_type": [ "Date/heure","Distance" ],
  "datas": [
    { "data": [6.81,17.34,10.53], "label": "2018-05-01 02:00:00.0" },
    { "data": [12.48,11.46,1.02], "label": "2018-05-01 08:00:00.0" },
    { "data": [12.63,8,4.63],     "label": "2018-05-01 14:00:00.0" },
    { "data": [6.64,10.91,4.26],  "label": "2018-05-01 20:00:00.0" }
  ]
}
```

Ainsi on peut l'exploiter pour l'affichage d'un tableau ou d'un graphique :

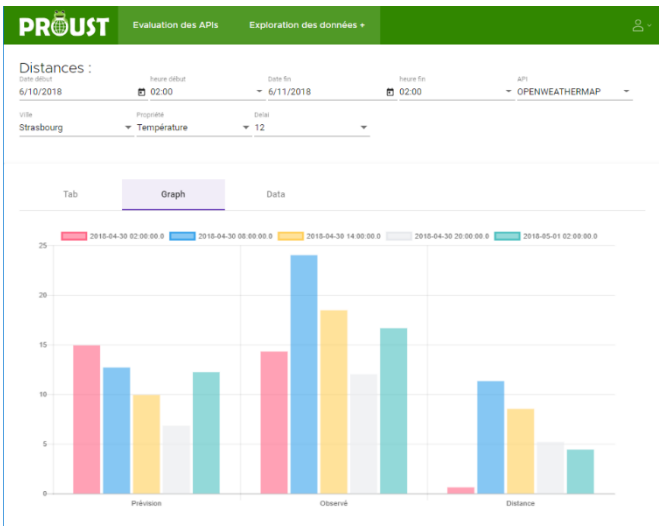

Figure 3 - Composant sous forme de graphique

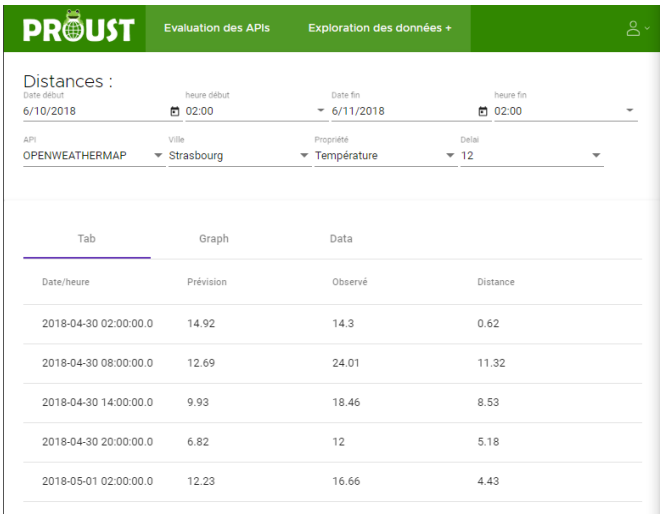

Figure 2 - Composant sous forme de tableau

Modélisation des données dynamiques côté serveur

Les données dynamiques du serveur sont classées en quatre catégories qui ont chacune un certain nombre d'utilisations. Vous trouverez ci-dessous un descriptif de chacune d'entre elles.

Catégorie 1 : Payload JSON orienté données

```
{
    "column_headers": [ … ],
    "series_type": [ … ],
    "data" : [
        { "label" : "...", "values" : [ … ], "unit": "..." }, ...
    ]
}
```

Catégorie 2 : Payload JSON spécifique pour le widget

```
{
    "city": …
    "dateRequest": …
    "delay" : [ ... ],
    "properties" : [
        { "label" : "...", "rawdata" : [ … ], "unit": "..." }, ...
    ]
}
```

Catégorie 3 : Payload JSON pour les données externes au système central

Comprend les données comme "API", "Location", "Delay" ou "Property".

```
{
    "id": …,
    //la suite des données dépend de l'objet persistant retourné
    //il peut y avoir des sous-objets avec une structure similaire
}
```

Catégorie 4 : BeanParams (semi-DTOs)

Catégorie spéciale puisqu'il ne s'agit pas exactement de DTOs mais d'objets injectables dans une classe de ressources Jersey/JAX-RS. Ce type d'objets permet l'utilisation des Query Strings de manière pratique et transparente côté Java.

```
public class … {

    @QueryParam("...") public … … ;
    …
}
```

Une variante plus sécurisée a aussi été implémentée, changeant tous les QueryParams en private et ajoutant des getters à la classe.

Modélisation des données persistantes

Modèle de données MongoDB

La première étape a été de sauvegarder les données dans une base MongoDB. Il s'agit d'une base NoSQL, et ce sont donc des objets JSON hétérogènes qui sont stockés dans des collections.

Pour le premier sprint, nous avons créé un script Python qui récupère les données des prévisionnistes, et les stocke dans ladite base.

Le script était lancé toutes les 6h et se basait sur la collection « configScript » dont voici un exemple :

```
{
    "_id" : ObjectId("..."),
    "nameAPI" : "APIXU",
    "active" : true,
    "url-realTime" : {
        "strasbourg" : "http://api.apixu.com/v1/current.json?q=strasbourg",
        "gaborone" : "http://api.apixu.com/v1/current.json?q=gaborone",
        "helsinki" : "http://api.apixu.com/v1/current.json?q=helsinki"
    },
    "url-6h" : {
        "strasbourg" : "http://api.apixu.com/v1/forecast.json?q=strasbourg&hour=6",
        "gaborone" : "http://api.apixu.com/v1/forecast.json?q=gaborone&hour=6",
        "helsinki" : "http://api.apixu.com/v1/forecast.json?q=helsinki&hour=6"
    },
    "url-12h" : {
        "strasbourg" : "http://api.apixu.com/v1/forecast.json?q=strasbourg&hour=12",
        "gaborone" : "http://api.apixu.com/v1/forecast.json?q=gaborone&hour=12",
        "helsinki" : "http://api.apixu.com/v1/forecast.json?q=helsinki&hour=12"
    },
    "url-1d" : {
        "strasbourg" : "http://api.apixu.com/v1/forecast.json?q=strasbourg&days=1",
        "gaborone" : "http://api.apixu.com/v1/forecast.json?q=gaborone&days=1",
        "helsinki" : "http://api.apixu.com/v1/forecast.json?q=helsinki&days=1"
    },
    "url-3d" : {
        "strasbourg" : "http://api.apixu.com/v1/forecast.json?q=strasbourg&days=3",
        "gaborone" : "http://api.apixu.com/v1/forecast.json?q=gaborone&days=3",
        "helsinki" : "http://api.apixu.com/v1/forecast.json?q=helsinki&days=3"
    },
    "keyparam" : "key",
    "keys" : [
        "...",
        "...",
        "..."
    ],
    "lastUsedKey" : 0
}
```

Nous avons ici la configuration d'une API, avec une liste de clés à, puis une liste d'url pour chaque délai. La propriété `lastUsedKey` nous permet de savoir quelle clé a été utilisée en dernier et nous permet de faire une rotation de clés pour chaque requête.

Le script permet de stocker un objet JSON contenant ayant une propriété `data` contenant le résultat brut de l'API requêtée :

```
{
    "_id" : ObjectId("5a80cb336e95526c30343a1a"),
    "city" : "helsinki",
    "storedDate" : "2018-02-12T00:01:07.188346",
    "delay" : 6.0,
    "api" : "APIXU",
    "data" : { "..." }
}
```

Les propriétés `api`, `city` et `delay` sont assez explicites, la propriété `storedDate` nous informe de la date à laquelle la requête a été faite, la propriété `data` nous donne le corps de réponse de la requête sur l'API en question.

Modèle de données MySQL

Configuration de l'API

Par la suite, nous avons remplacé le script Python par des requêtes directement depuis notre application Java. La configuration se fait en MySQL et est un peu plus généralisée, dans le sens où l'on ne retrouve plus de redondance dans les entrées. Nous laissons tout de même le script Python en fonctionnement dans une table de cache en cas de panne de serveur Tomcat, ce qui nous évite d'avoir des données perdues.

Nous avons ici une API contenant une liste de clés ainsi que deux schémas d'URL pour les requêtes observées et prévisionnelles. Un schéma d'url type ressemblera à cela :

```
http://api.openweathermap.org/data/2.5/
forecast?appid={key}&q={city},{country.code}
```

Chaque balise encadrée par des accolades est remplacée par les informations de requête nécessaires pour récupérer des informations.

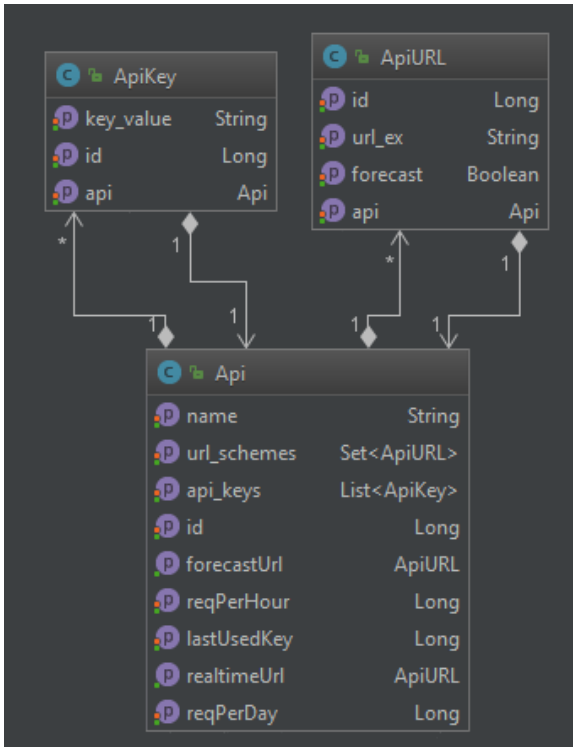

Figure 4 - Morceau du modèle commun concernant la configuration des APIs météo

Modèle commun

Pour pouvoir exploiter les données, chaque donnée brute est transformée dans un modèle commun. Voici le modèle mis en place :

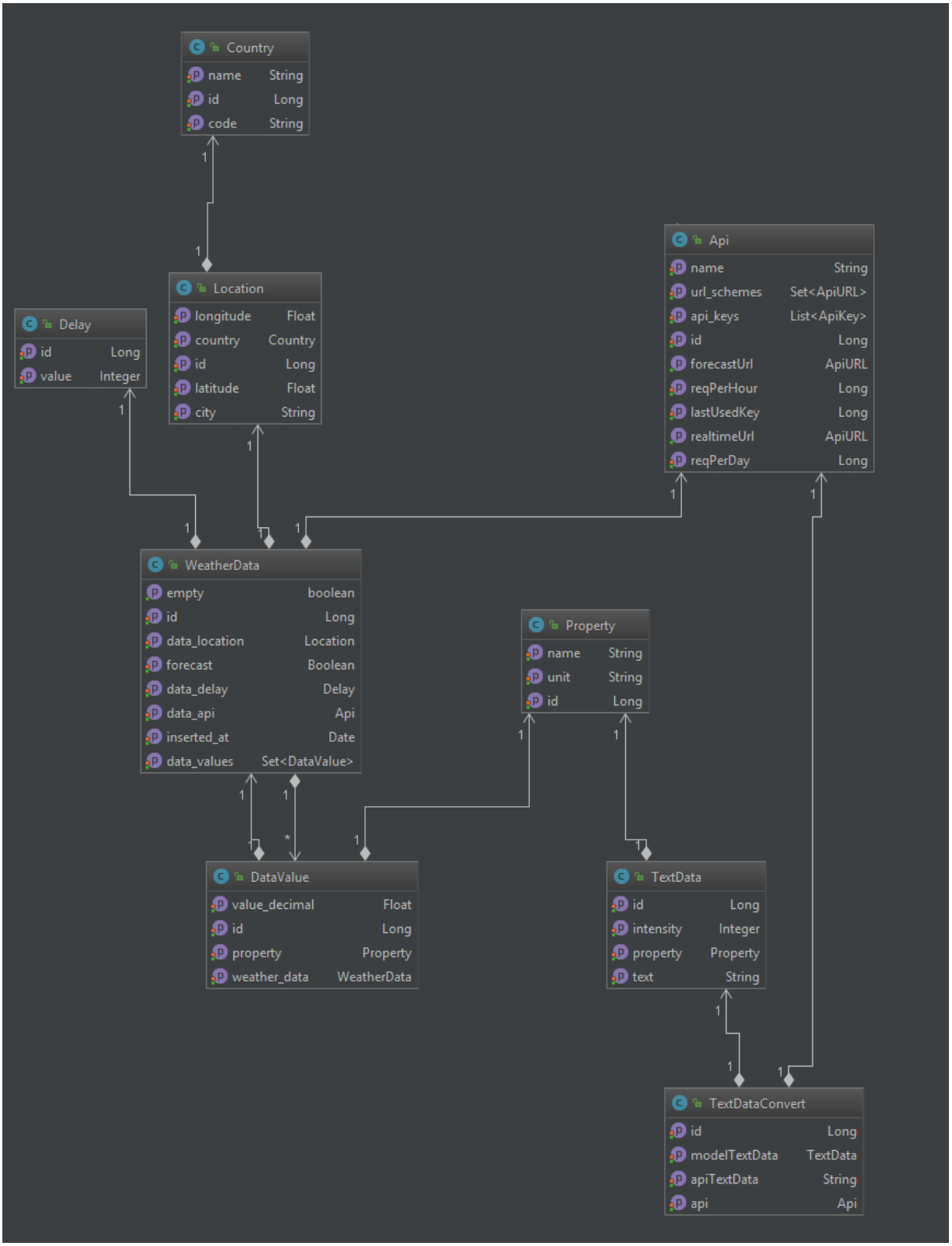

Figure 5 - Modèle commun MySQL

L'entité `Weatherdata` représente une donnée normalisée d'une réponse d'API. Elle a une date de requête (`inserted_at`) et est liée à un délai, un lieu, et une API. Un lieu est une ville, liée à un pays. De plus, elle contient une liste de valeurs décimales (`DataValue`) qui sont chacune liées à une propriété. (Par exemple, pour la propriété Température, on aura par une valeur de 37 pour 37°C).

Pour pouvoir évaluer les données textuelles récupérées par les APIs (pluie…), on utilise une table `TextData` qui aura une valeur, une intensité, et qui sera liée à `TextDataConverter`, qui fera le lien entre la valeur de l'API et la propriété. Chaque propriété textuelle a une unité d'un nombre entre x et n, correspondant. Par exemple, pour la propriété "Peu de nuages", on aura une intensité allant de 1 à 4.

Système utilisateur

Nous avons implémenté un système utilisateur dans notre API, qui n'a malheureusement pas été utilisé. Le mot de passe est chiffré puis salé au moyen de jBCrypt et de l'algorithme de chiffrement Java PBKDF2.

À des fins d'information, le mécanisme repose sur des JsonWebToken pour authentifier l'utilisateur de bout en bout.

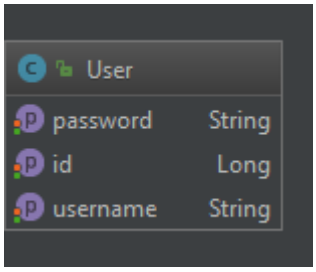

Figure 6 - Entité utilisateur

Système de cache

Lorsque l'utilisateur demande la prévision optimale, nous avons un système de cache. Si le cache n'existe pas ou n'est pas à jour, nous faisons des requêtes dans les APIs correspondantes et le stockons en cache, sinon, on retourne ce qui est déjà stocké.

Il est très semblable au modèle de données précédent, à la différence prête de l'entité CacheRequest. Cette entité nous permet de savoir si l'on a bien fait une requête à l'heure et aux critères précisés. En effet, il se peut qu'une requête ne correspond pas à un délai précis, et n'est donc pas stockée dans WeatherDataCache. Pour éviter le doublon de requête pour une donnée invalide, on préfère stocker l'appel à la requête dans une table, puis stocker le résultat de la requête dans une autre.

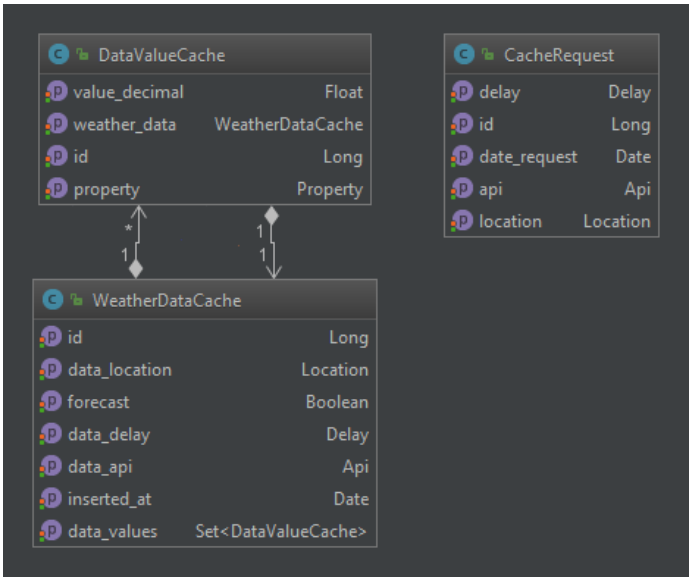

Figure 7 - Modèle de cache

Optimisation des données

Pour optimiser nos requêtes SQL dans le calcul de prévision optimale, nous avons une table se mettant à jour toutes les 6h stockant les distances entre la valeur observée et la prévision, ce qui nous évite un traitement long.

La table contient pour un lieu, une propriété, une API et un délai, la valeur de la prévision, la valeur observée pour cette prévision et la distance entre ces deux valeurs.

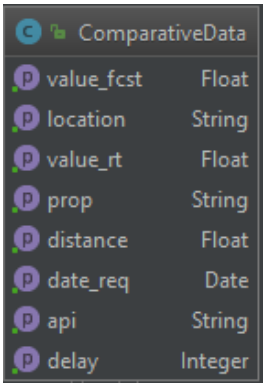

Figure 8 - Entité pour les calculs

Annexe – Table des illustrations

Annexe - Index