Lucas BCHINI
Fayçal BENAZIZ
Florent JAMET
Kévin JEAN
Matthieu LANDOS
Nathan LOISEL
Alister MACHADO DOS REIS
Marcely ZANON BOITO

Document de conception

Systèmes distribués pour le traitement de données
-
Distributed systems for data processing

Sommaire

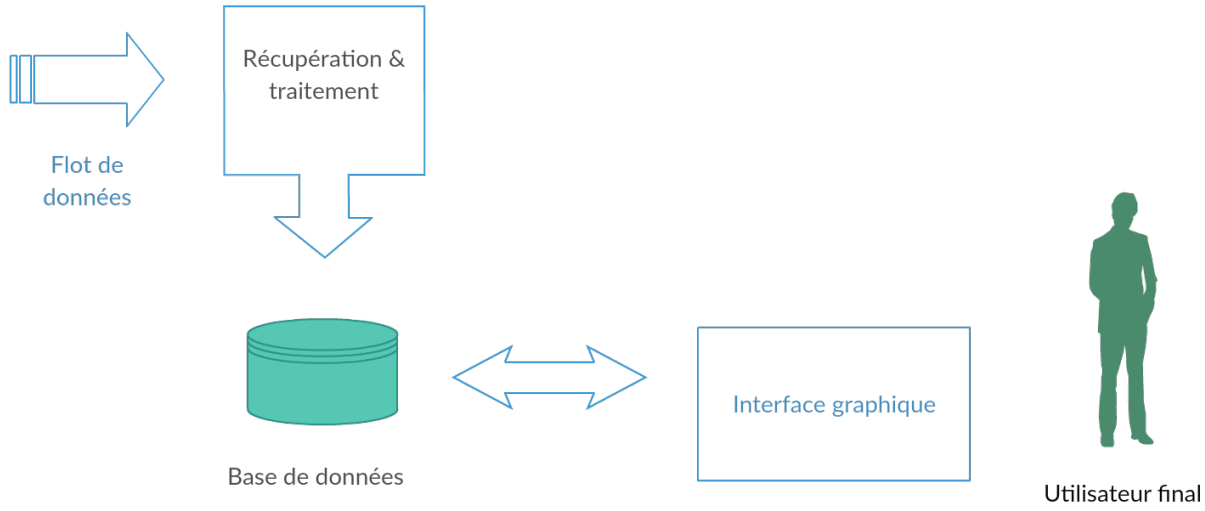
Introduction

Ce document rassemble les informations nécessaires ou utiles pour le déploiement, la compréhension et la prise en main de notre application. Vous trouverez ci dessous une description détaillée du fonctionnement derrière l'interface, des justifications pour les choix que nous avons fait, une notice pour le déploiement à grande échelle et enfin un manuel avec des captures d'écran de l'interface web.

Description, architecture et comportement général

Pour ce projet, nous avons décidé de créer une application simple d'utilisation pour obtenir des informations météorologiques diverses (pluviométrie, ensoleillement, humidité,…). Les données dont nous disposons ne permettent cependant que la consultation historique. L'utilisation d'une application web s'est imposée de par les consignes que nous avons eu autant que par sa simplicité d'accès.

L'architecture et les technologies que nous avons choisies le sont autant pour leur résistance aux pannes que pour leur distribuabilité.



OpenWeatherMap

OpenWeatherMap est une des plus grandes base de données météorologique auxquelles nous pouvions avoir accès. Elle permet d'obtenir le temps actuel pour un total de 209.579 villes. La version gratuite de leur API (*Application Programming Interface*) offre des

données instantanées uniquement et est ouverte à tous, cependant après les avoir contactés, ils nous ont offert l'autorisation d'accéder à des données historiques sur 5 jours (avec un pas de 3 heures entre chaque relevé). Ce service permet d'obtenir les données météorologiques pour 5 jours, à partir du dernier relevé effectué.

Le retour de ce service est une réponse JSON contenant 40 objets, contenant chacun 8 relevés par jour pendant 5 jours, pour chaque ville. Vous trouverez plus d'informations sur le format directement sur leur site web.

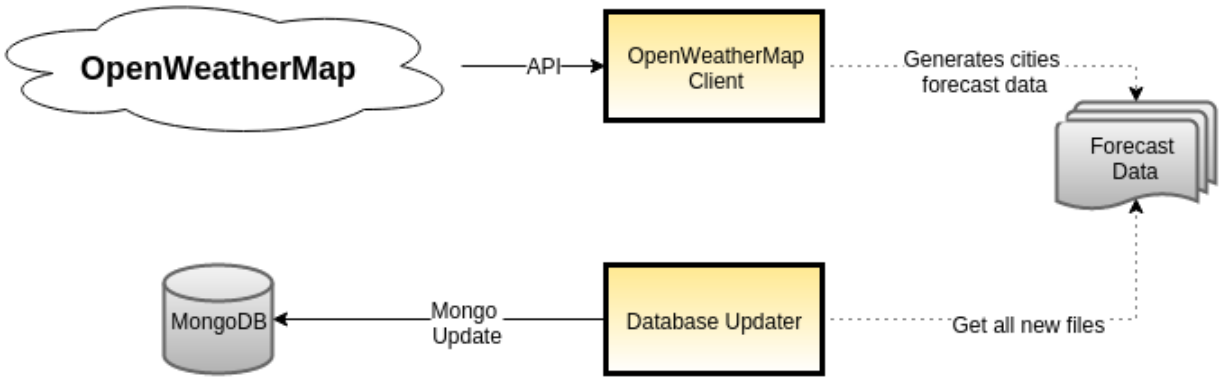
Génération des données et stockage en base de données

La porte d'entrée de notre système est la génération de données. Nous n'utilisons que des villes françaises à cause d'un manque d'infrastructure (espace disque, puissance de calcul…) mais tous nos scripts sont écrits de façon à pouvoir générer en temps réel (bien que dans l'application finale, ce choix n'ait pas été retenu) des données de villes du monde entier si nous disposions de l'infrastructure suffisante. Une fois le stockage fait dans la base de données, l'utilisation est indépendante de la partie gérant la récupération des données.

Nous avons aussi implémenté la structure nécessaire pour effectuer des requêtes en temps réel comme indiqué dans la suite de ce rapport.

Génération de données

Globalement, nous avons effectué tout le traitement en utilisant Python et des scripts Bash. Étant un langage de script par nature, la gestion des appels à l'API a été grandement simplifiée. Le schéma suivant décrit le fonctionnement de nos scripts pour la récupération de données.



- **OpenWeatherMap Client** :

Le client est suffisamment générique pour que chaque utilisateur puisse configurer le type de requête qu'il souhaite effectuer. En clair, il dispose de la clé de l'API et de deux types de requêtes : Temps instantané ou historique sur 5 jours, avec un pas de 3 heures.

Nous avons implémenté l'utilisation des deux mais nous n'utilisons que le deuxième dans la version livrée de notre projet (au début du projet, nous n'étions pas certains de la direction à suivre, nous avons donc travaillé le plus largement possible).

- **Getter et Scheduler** :

Afin de récupérer des données de la base tous les jours sans avoir à lancer un script manuellement, nous avons écrit un *getter* et l'avons intégré dans un *crontab*. Il fonctionne comme suit pour récupérer les données météorologiques mondiales :

- Nous récupérons tous les identifiants des villes du système OpenWeatherMap et les séparons dans plusieurs fichiers contenant moins de 50 000 identifiants par fichier (5 en tout). Cette étape est nécessaire pour respecter le nombre maximum de requêtes journalières.

- Nous avons créé une fonction qui choisit un des fichiers contenant les identifiants différent chaque jour et essaye de mettre à jour les historiques sur chaque ville pour les 5 jours qui viennent de passer.

- Nous avons ensuite mis le script dans le *crontab* pour qu'il s'exécute tous les jours à l'heure demandée.

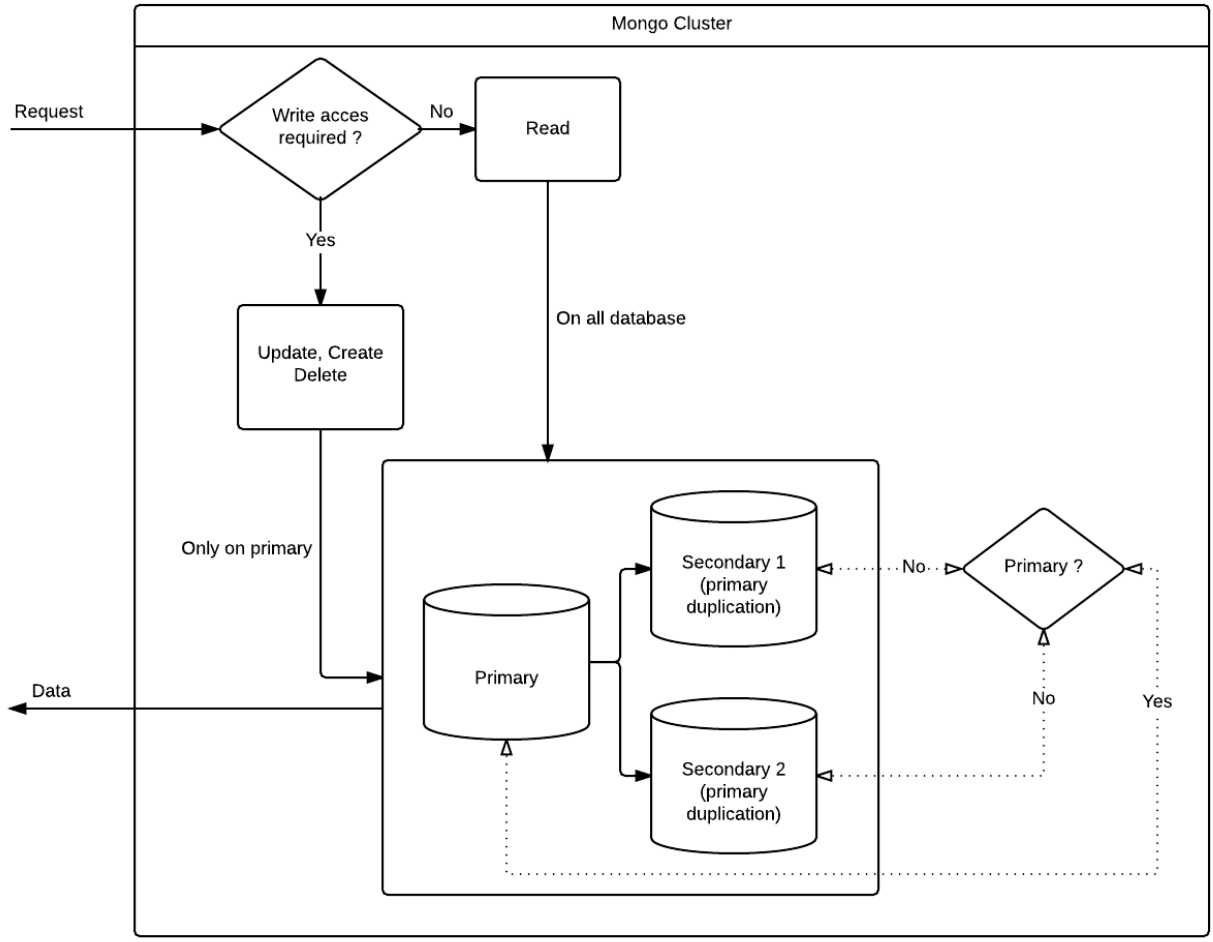
Néanmoins, cette méthode n'est pas applicable car nous ne disposons pas de l'infrastructure matérielle suffisante pour stocker une quantité aussi importante de données. Nous avons donc finalement choisi de n'utiliser que des villes françaises et avons désactivé le *crontab* afin de limiter la quantité de données mémorisée.

- **Format de données** :

L'API retourne un fichier JSON et chaque fois que nous exécutons notre script de traitement de données, nous obtenons deux fichiers pour chaque ville. Le fichier historique et le fichier "courant". Dans le fichier historique, nous avons toutes les données dont nous disposons depuis la première fois que le script a été exécuté pour une ville. Dans le fichier "courant", nous avons les données pour les 5 prochains jours.

Stockage en base de données

VUE LOGIQUE DE LA PARTIE STOCKAGE



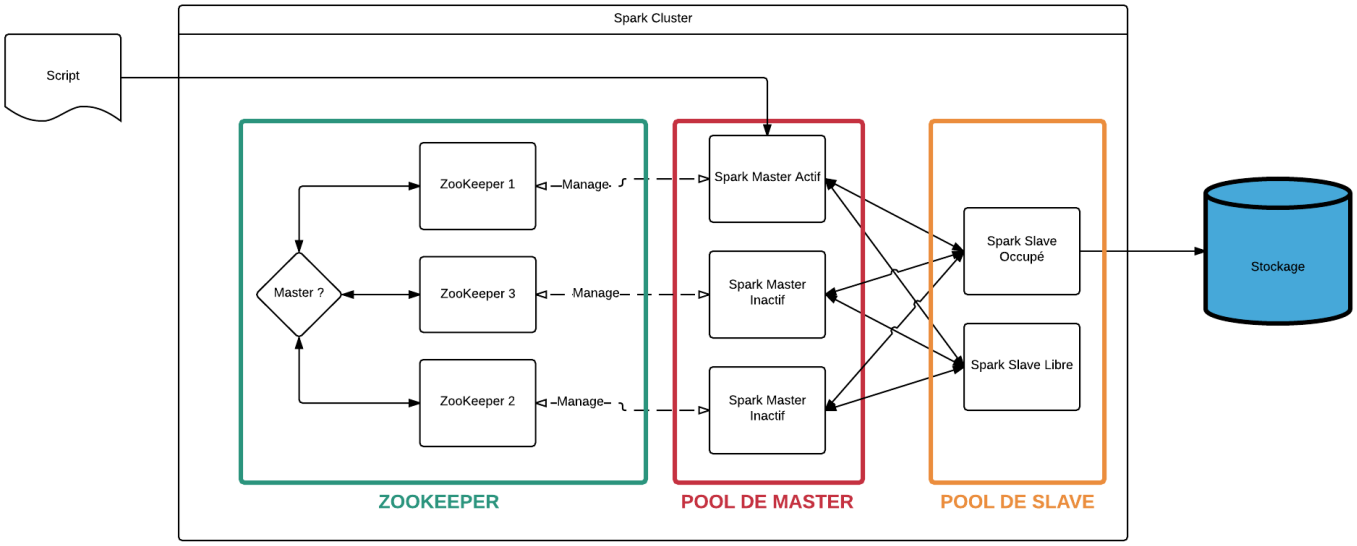
Par sécurité, nous avons choisi de tripler la base de données. Quand une requête est envoyée à MongoDB, si aucune écriture n'est nécessaire, la lecture se fera sur une des bases (sans distinction). Si une écriture est nécessaire, la base primaire est interrogée, une fois la transaction validée, les données sont dupliquées sur les deux bases redondantes.

En cas de panne d'une base secondaire, le consensus est toujours possible entre les deux restantes, l'application reste fonctionnelle. Si la base primaire tombe, une élection est faite (encore une fois par consensus) entre les deux bases restantes, la base élue deviendra alors primaire et l'application fonctionnera encore. Si deux bases sont en panne, le consensus n'est plus possible et l'application ne répondra plus.

Schéma architectural & résistance aux pannes

Ce diagramme résume le fonctionnement de la partie de récupération de données. Nous allons expliquer la fonction pour chaque zone (*Zookeeper*, *Masters*, *Slaves*), et dans le cas d'une panne, la réaction attendue.

VUE LOGIQUE DE LA PARTIE TRAITEMENT



Zookeeper

Le rôle du *Zookeeper* est transverse. Ce sont des petites instances qui servent à maintenir une instance de *master* active en permanence. Ils sont dupliqués afin de ne pas créer un SPOF (*single point of failure*). Ils surveillent leur fonctionnement mutuel grâce à un *heartbeat* (signalement régulier de chaque noeud afin de s'assurer de leur bon fonctionnement).

En cas de panne d'un *Zookeeper*, s'il il gérait le *master* actif, les deux restants se contactent, font un consensus pour élire un nouveau *master* et permettent à l'application de continuer à traiter des données. Si le *Zookeeper* en panne gérait un *master* inactif, cela signifie qu'il était déjà redondant, aucune réaction n'est nécessaire. Si deux *Zookeepers* tombent en panne, le consensus n'est plus possible donc aucun nouveau *master* actif n'est réélu et le traitement de données s'arrête.

Master Actif

Le *master* actif est le point le plus important, il maintient une liste de *slaves* actifs et attribue une tâche à chaque *slave* se signalant disponible. Quand un *master* tombe en panne,

son *Zookeeper* l'attend (si une remise en ligne est possible) et signale aux deux autres *Zookeepers* de choisir un autre *master*. Si le *master* en question revient en ligne, il sera un *master* inactif jusqu'à ce qu'un nouveau doive être élu.

Slaves

Chaque *slave* disponible se signale à la tâche *master* active. Un script de traitement lui est attribué pour une certaine collection de données, le *slave* exécute la tâche puis re-stocke les données dans la base *ad hoc*. En cas de panne, un *slave* ne se signalera pas, on considère donc que tant qu'un *slave* est fonctionnel, l'application "peut" fonctionner à un rythme moins soutenu.

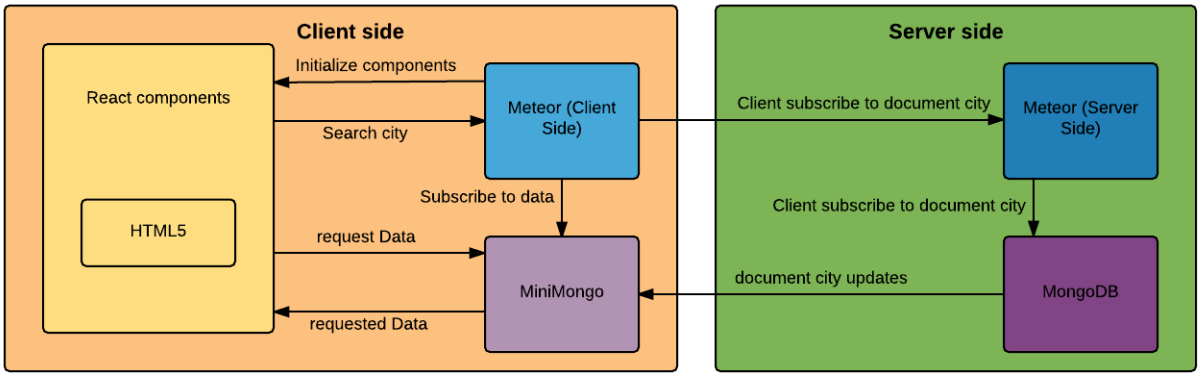
De la base de données à l'affichage

Une fois les données dans la base, il faut pouvoir les transférer à l'utilisateur dans l'interface désignée. Nous utilisons donc Meteor et MongoDB pour cela, l'architecture de cette partie est expliquée ci-dessous, le détail des architectures déployées de Meteor et MongoDB peuvent être trouvés plus bas pour Meteor et plus haut pour MongoDB.

Nous avons utilisé plusieurs technologies différentes pour l'affichage : MaterializeCSS, et Chartist en nous inspirant du « *flat design* » afin d'obtenir un rendu agréable.

Architecture et fonctionnement détaillé

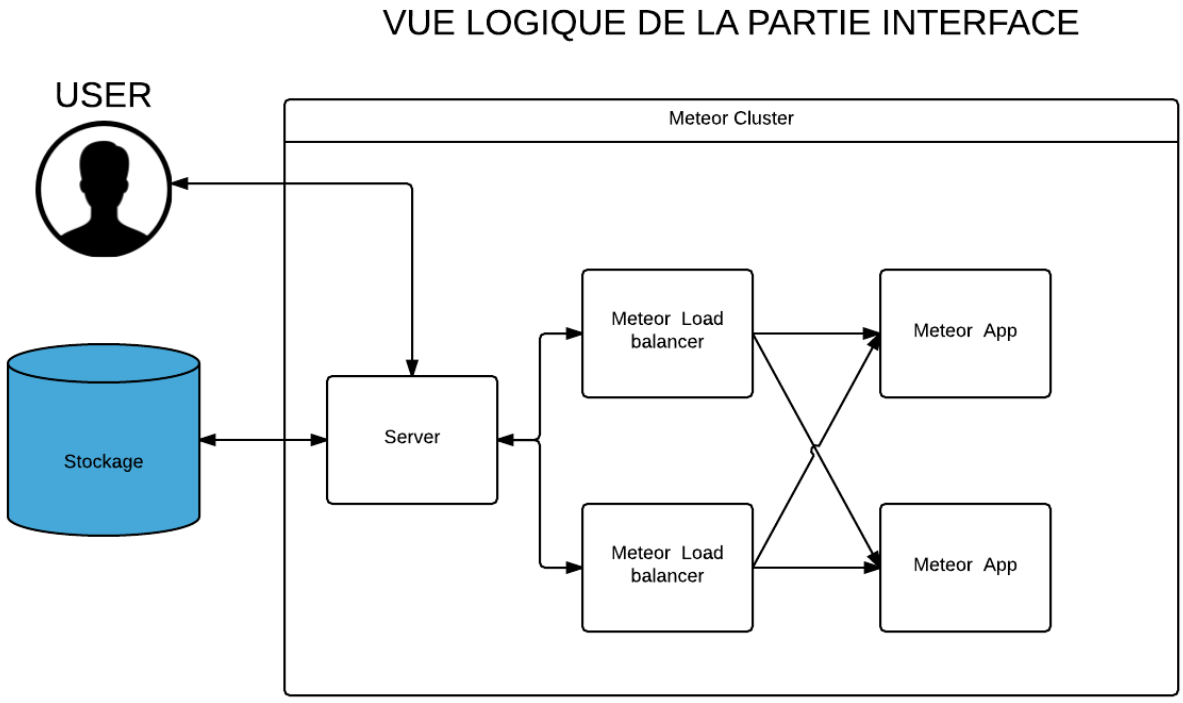
Vue logique de la partie Application



Quand un client se connecte à l'application, les instances de Meteor coté client et coté serveur sont liées. Meteor crée la page html en assemblant les composant *react* ensemble. Pour se connecter à la base de stockage locale et à sa partie placée sur le serveur, Meteor utilise un modèle *subscriber-publisher*.

Quand l'utilisateur saisit une recherche, une première requête est faite par Meteor client dans la base MiniMongo. Si elle n'est pas conclusive, Meteor client envoie une requête au coté serveur et est répercutée dans la base distante (grâce au *subscribing*, les données sont placées dans le MiniMongo) pour être accessibles à l'interface utilisateur. MiniMongo contient au plus une ville pour ne pas surcharger le coté client.

Distribution & fonctionnement de Meteor



Quand un utilisateur lance une requête, le serveur interroge les *load balancers* qui transmettent la requête à la bonne application Meteor. Meteor va ensuite chercher les données dans la base de données, puis les informations sont retournées à l'utilisateur pour être affichées.

Pour la gestion de pannes dans Meteor, ce que le schéma propose permet de gérer une panne (le *load balancer* et l'application Meteor sont sur la même machine). En pratique, l'absence de serveur par manque de moyens ne nous permet pas de gérer de pannes sur Meteor : nous avons besoin d'utiliser un numéro de port pour contacter les *load balancers*, nous pourrions nous en passer si nous disposions d'un nom de domaine.

Changement d'échelle

Dans la mesure où ce projet implique de l'informatique distribuée, la question du passage à la l'échelle supérieure se pose immédiatement. Pour les trois principaux aspects (calcul via Spark, stockage via MongoDB et affichage via Meteor), voici les piste les plus évidentes pour augmenter la capacité de notre application :

- Concernant le calcul, pour améliorer la résistance aux pannes franches, il faut augmenter le nombre de *master* / *Zookeeper*. Pour améliorer la disponibilité et la puissance de calcul, il faut augmenter le nombre de *slaves*.

- Concernant le stockage, il faut simplement augmenter l'espace de stockage physique. Pour augmenter la robustesse du système, il faut augmenter le nombre d'instances de MongoDB en parallèle et sur des noeuds différents est l'idée centrale.

- Pour l'affichage enfin, pour améliorer la résistance et la disponibilité, il faut ajouter un serveur qui redirige la connexion. Une fois cela fait, il ne reste plus qu'à augmenter le nombre d'instances parallèles de Meteor. Les capacités de calcul peuvent être considérées comme minimales pour l'instant, à terme, dédier des serveurs au calcul et à l'affichage semble être une idée pertinente.

Choix faits

Spark

Nous avons décidé d'utiliser Spark (solution d'Apache) plutôt que Hadoop, principalement pour sa simplicité d'installation et d'utilisation. Spark est aussi potentiellement plus rapide car il utilise la mémoire vive plutôt qu'une gestion sur disque. Il est également plus adapté aux petites instances comme celle que nous avons créé.

Meteor

Nous avons choisi Meteor par confort, notre équipe disposant d'une compétence conséquente avant le début de ce projet. Sa base de donnée intégrée par défaut étant MongoDB, la passerelle entre les deux est quasiment automatique. Meteor a donc également impliqué le choix de MongoDB qui était déjà une voie désignée.

MongoDB

De par le choix de Meteor, MongoDB s'est imposé. Sa structure distribuée et la simplicité d'utilisation ont également été des choix décisifs.

Déploiement distribué

Pour déployer notre application sur plusieurs machines, la première étape est d'installer sur tous les logiciels nécessaires au fonctionnement normal (Vagrant, Python, Scala, Meteor, Spark et Zookeeper).

Il faut ensuite mettre en place les fichiers de configuration pour chacun d'entre eux. Ils peuvent être trouvés dans le code source fourni avec ce document.

Enfin, il faudra lancer les programme dans l'ordre suivant : Zookeeper, Spark, Vagrant (avec l'image du MongoDB) et Meteor. Si les fichiers de configuration sont correctement ajustés, le site sera accessible.

Difficultés rencontrées

Durant tout le projet, la principale difficulté provenait de l'apprentissage des différents langages et technologies. Certains d'entre nous avaient utilisé Meteor avant le projet, mais MongoDB, Spark ou Vagrant étaient inconnus. Naturellement, l'apprentissage d'une nouvelle technologie prends toujours un certain temps mais l'utilisation conjointe de plusieurs d'entre elles a demandé un effort supplémentaire.

Le langage Scala est utilisé par Spark pour créer des scripts de calcul. Malgré une documentation et une communauté importantes, il manquait d'exemples concrets pour notre cas. Par la nature du projet, il a également fallu distribuer MongoDB et Meteor, ces deux points ont à nouveau demandé une soirée chargée.

Une fois passé tous les soucis liés à l'apprentissage, Meteor a été la cause principale de ralentissements, le lien entre Meteor et MongoDB (l'instance externe de Mongo) n'est pas forcément facile à utiliser, en particulier à cause de Meteor qui est à la fois client et serveur (le transfert de données entre les deux n'étant pas évident).

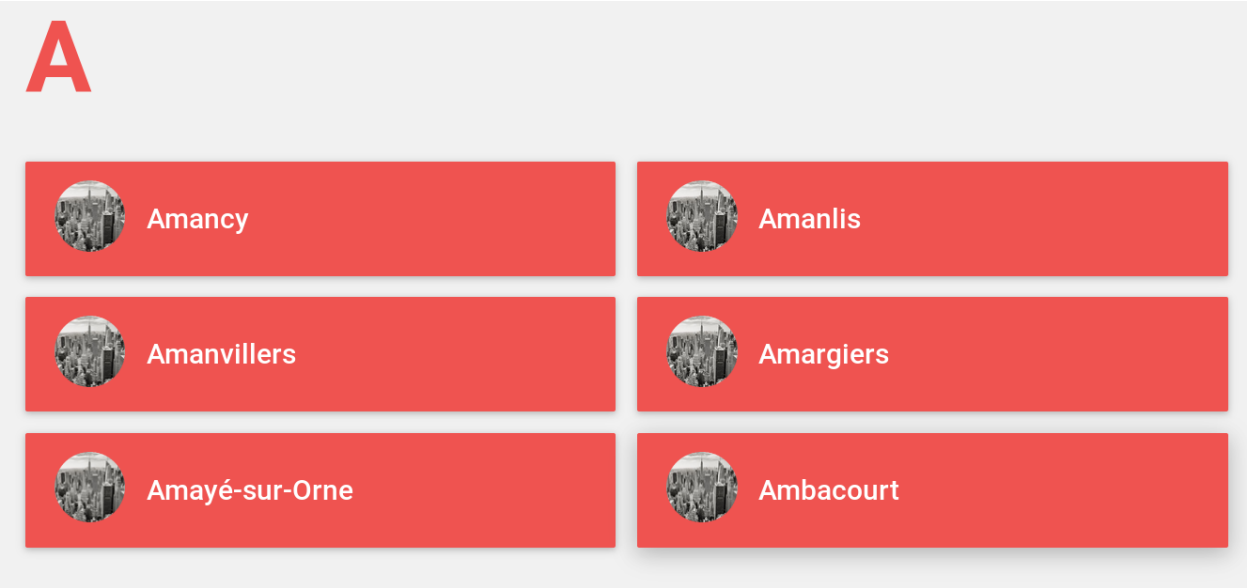
Nous avons effectué nos tests sur les ordinateurs de l'Ensimag qui ne nous accorde pas les droits d'administration, nous avons donc eu des soucis d'espace disque et d'installation de Meteor, Vagrant (MongoDB), Spark,...

Manuel utilisateur de l’interface web

Comme expliqué plus haut, nous avons voulu cette interface aussi simple et épurée que possible.

Interface de recherche

Voici l’interface de recherche. Il suffit de taper le nom de la ville dont vous voulez connaître la météo. Si l’orthographe vous échappe, ne saisissez que les premières lettres et notre application vous proposera la fin du nom (si la ville est connue, bien évidemment).



Interface de consultation

Dans l'interface de visualisation de la météo, le bouton "*Today*" permet d'afficher les 8 relevés du jour (relevé actuel + 7 précédents), le bouton "*5 days*" affiche le temps lissé sur 4 jours. Sur le relevé au centre de l'écran, les trois boutons (rouge, vert et bleu, respectivement) permettent de voir la température, le vent et la pluviométrie pour les points affichés.

Pour accéder aux statistiques (moyenne ou record), il suffit de cliquer sur le titre de l'onglet voulu. L'icone de loupe dans le coin haut-droit permet de rechercher une autre ville.