P

<?xml version="1.0"?><DocumentBlank xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"> <GraphicCharterDefinitionId>0</GraphicCharterDefinitionId> <TemplateBaseTypeId>0</TemplateBaseTypeId> <CompanyId>1</CompanyId> <ConfidentialId>0</ConfidentialId> <ConfidentialDescription /> <CountryId>0</CountryId> <PageSizeId>1</PageSizeId> <PageOrientationId>1</PageOrientationId> <PrePrintedStationary>false</PrePrintedStationary> <Project>Parking Connecté</Project> <Reference>20170727-100709-a</Reference> <TemplateType>1</TemplateType> <CultureId>fr-FR</CultureId> <LanguageId>2</LanguageId> <Customer>Sopra Steria Montpellier</Customer> <DocumentDate>2017-08-02T16:47:49.8111898+02:00</DocumentDate> <FirstPageHeader /> <FirstPageSubHeader /> <Saved>false</Saved> <IsValid>true</IsValid> <FirstPageCover>false</FirstPageCover> <IsNew>true</IsNew> <Title>Présentation des technos Digiroom</Title></DocumentBlank>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | |
|  |  |  |  | |
|  |  | Projet Parking Connecté | | |
|  |  |  | | |
|  |  | Tout ce qu’il faut savoir sur les technologies utilisées et la mise en place de l’environnement de travail | | |
|  |  |  | |  |

# Architecture globale prévisionnelle

## Schéma global

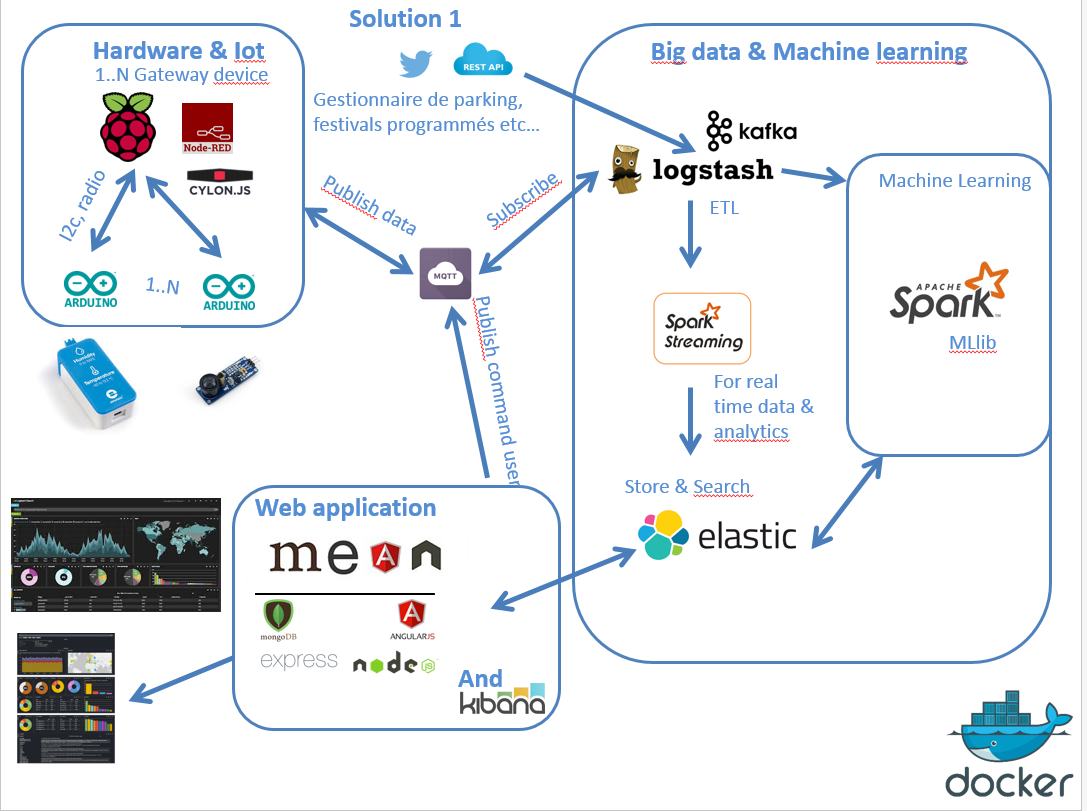
 Ce schéma regroupe toutes les technologies susceptibles d’être utilisées au cours du projet. Elles sont regroupées en trois groupes.

figure 1 : schéma global

### Hardware et IOT :

Elle regroupe tous les objets connectés et les technologies qui leur permettent d’émettre des données. On utilise des Raspberry Pi 3 et des Arduino avec Cylon.js ou Node RED pour communiquer avec ces derniers. Le protocole de communication avec le reste de l’architecture est MQTT qui est le standard le plus utilisé. On utilise Mosquitto pour envoyer les données.

* Raspberry Pi : c’est un nano-ordinateur qui fait la taille d’une carte de crédit et qui permet d’exécuter des variantes des systèmes d’exploitation GNU/Linux.
* Arduino : C’est un microcontrôleur qui permet de faire fonctionner des objets tels que des capteurs.
* Node RED est un outil logiciel permettant de faire communiquer les objets connectés (i.e. les Arduino et Raspberry). Il est basé sur la technologie node.js.
* Cylon.js : c’est une librairie JavaScript permettant également de faire communiquer les Arduino et les Raspberry.
* MQTT (MQ Telemetry Transport) est un protocole de messagerie basé sur la publication de messages et à la souscription à des flux via le protocole TCP/IP. On publie des messages sur un flux et les souscripteurs de ce flux reçoivent ces messages.
* Mosquitto est un « Broker » MQTT, c’est-à-dire qu’il permet d’envoyer des messages MQTT et de souscrire à des flux.

### Big Data et Machine learning :

Son rôle est de récupérer les informations depuis les objets connectés ainsi que dans les banques de données  existantes (météo, trafic routier, événements, etc.), de les adapter et de les stocker afin de les réutiliser. Kafka, logstash, et Spark Streaming sont des outils permettant de le faire. Le stockage est assuré par Elasticsearch. L’aspect « Machine learning » est assuré par Spark et la librairie MLlib.

* Kafka : cette technologie permet de manipuler des flux de données en temps réel. Par exemple il peut souscrire à un flux de données MQTT (topic) et faire des traitements de données pour les transformer et les rendre disponibles via un nouveau flux.
* Logstash : tout comme kafka logstash permet de gérer des données en temps réel. En plus de cela il est très facile d’intégrer ces données à une base Elasticsearch car ils font partie de la même suite d’outils (voir plus bas)
* Spark Streaming : c’est également un outil d’acquisition de données depuis les flux de données.
* Elastic Search : c’est une base de données no SQL qui permet de faire des recherches efficaces et qui fournit une API permettant d’interagir avec la base via des requêtes http.
* Spark & MLlib : Spark et MLlib sont des outils puissants qui permettent d’exploiter les données en grande quantité et de faire de l’apprentissage pour créer des modèles de prédiction.

### Restitution

Elle prendra la forme d’une application WEB qui utilisera les technologies node js, angular js, mongoDB, et Kibana.

* Node js : Une technologie web basée sur l’évènement et permettant d’avoir une application web robuste pouvant accepter une montée en charge (augmentation du nombre d’utilisations simultanées).
* Angular js : c’est un framework Javascript permettant de gérer dynamiquement la mise en page d’un site web.
* mongoDB : Un Système de Gestion des Bases de Données orientées document et pouvant être réparti sur plusieurs machines.
* Kibana : c’est un outil associé à elasticsearch qui permet de visualiser les données de la base et de réaliser notamment des tableaux de bord.

# Mise en place de la machine virtuelle

* Il faut installer **virtualbox** et créer une machine virtuelle Redhat et y installer centos 7 (iso fournie)
* On lui alloue 30Gio de stockage
* Bien choisir le mot de passe
* Démarrer la VM
* Il faut ensuite installer **putty** qui permet de se connecter en ssh à la machine virtuelle.
* Dans les paramètres de la machine virtuelle :
  + Onglet réseau
  + Avancé
  + Redirection de ports
  + Cliquer sur le + en haut à droite
  + Indiquer Nom : ssh Protocole : TCP Port Hôte : 2022 Port invité : 22
* Récupérer l’IP de la VM :
  + Sous windows : ouvrir le centre réseau et partage
  + A gauche modifier les paramètres de la carte
  + Clic droit sur « VirtualBox Host-Only Network » puis Propriétés
  + Chercher dans la liste : « Protocole Internet Version 4 » et cliquer sur propriétés
  + Récupérer l’adresse IP
* Ouvrir Putty et créer une nouvelle connexion avec l’adresse IP et le port 22
  + Se connecter avec les identifiants **root**
* **Yum install docker**
* **service docker start**

# Virtualisation avec Docker

Chacune de ces technologies sera installée indépendamment grâce à Docker. Docker est un outil de virtualisation qui permet d’installer des logiciels dans des images qui contiennent un système d’exploitation allégé. Les containers sont une sorte de machine virtuelle qui permet d’exécuter les images. Elles démarrent en quelques secondes. On peut trouver des images déjà prêtes pour certaines technologies. Il faut ensuite les lancers pour effectuer les modifications que l’on souhaite, puis enregistrer son état dans une nouvelle image (Commit).

Voici une liste des commandes que j’ai pu utiliser :

* **docker run –it --name kafkacontainer kafkadocker bash**
* docker run : commande qui permet de lancer une image dans un container
* --it : options
* Option i : se connecter aux entrées et sorties standard du container
* Option t : se connecter au docker via un pseudo terminal pour exécuter des commandes
* Option --name kafkacontainer : Option pour donner un nom au container. En l’absence de cette option un nom aléatoire est attribué.
* Kafkadocker : nom de l’image
* Bash : commande à exécuter
* Pour résumer on démarre un container qui s’appellera kafkacontainer et qui exécutera un shell bash sur l’image nommée kafkadocker pour pouvoir par exemple se connecter à un topic MQTT et créer un topic kafka.
* **docker exec -it kafkacontainer bash**
* Docker exec : permet d’executer une commande sur un container en cours d’exécution et sur lequel on n’a pas la main. On peut avoir un aperçu des containers démarrés, de leurs noms et identifiants (hashcode) grâce à la commande **docker ps**
* –it : options
* Option i : se connecter aux entrées et sorties standard du container
* Option t : se connecter au docker via un pseudo terminal pour exécuter des commandes
* Kafkacontainer : nom du container à utiliser
* Bash : commande à exécuter
* Pour résumer on reprend la main sur le container nommé kafkacontainer et on dispose d’un invité de commandes pour pouvoir manipuler le container.
* **docker commit kafkacontainer kafkadocker**
* docker commit : commande qui permet d’enregistrer le container dans une nouvelle image.
* Ici on enregistre l’état du container kafkacontainer, dans l’image kafkadocker. Attention, si l’image existe elle sera remplacée, sinon une nouvelle image sera créée. Le container doit être en fonctionnement pour pouvoir effectuer l’opération « commit »
* **docker ps** : permet d’avoir un aperçu des containers en exécution, leur nom, identifiant temps écoulé depuis le démarrage, depuis le début d’exécution,…
* avec l’option –a on a aussi les containers qui ne sont pas en cours d’exécution, pour pouvoir les supprimer et gagner de la place sur le disque.
* **docker rmi kafkadocker** :
* Permet de supprimer une image
* **docker rm kafkacontainer** :
* Permet de supprimer un container (très utile car les containers peuvent s’accumuler et prendre beaucoup d’espace sur le disque)
* **docker images** :
* Permet de voir toutes les images
* Remarque : on peut utiliser le pipe « | » pour faire une recherche grâce à la commande « grep » : **docker images | grep kafka**
* **docker pull** docker.elastic.co/elasticsearch/elasticsearch:5.5.0
* permet de récupérer une image depuis une url : ici c’est une image contenant elasticsearch (le SGBD)

Afin de simplifier le lancement des containers et la communication entre eux, j’ai utilisé docker-compose. C’est un outil permettant de lancer une application qui repose sur plusieurs containers, grâce à la commande **docker-compose up**. Il utilise un fichier de configuration au format YAML. Ce fichier s’appelle **docker-compose.yml**. On y définit plusieurs informations pour chaque container que l’on veut démarrer :

* **image** : l’image à utiliser
* **container\_name :** le nom du container
* **command :** la commande à exécuter au démarrage
* **ports :** les ports réseaux à exposer aux autres containers
* **links :** les containers accessibles par leur nom (plutôt que par l’adresse ip, qui peut changer d’une exécution à l’autre).

Dans la suite je tâcherai d’expliquer comment mettre en place les images pour chaque technologie.

## Mosquitto

Pour le docker mosquitto, il suffit de récupérer sur le docker hub (équivalent de github et de l’appstore) grâce à la commande **pull** : ***docker pull toke/mosquitto***

Une autre méthode est l’utilisation d’un dockerfile. Il contient toutes les informations permettant de créer l’image à partir d’une autre image (souvent linux, debian, centos etc.). Le dockerfile est disponible sous le nom ***mosquitto.dockerfile***. Pour créer l’image on exécute la commande suivante : ***docker build + fichier.dockerfile***.

Ce fichier contient les commandes qui sont automatiquement exécutées pour installer tout le nécessaire (ici mosquitto client et server).

On pourra publier des messages avec la raspberry via ce serveur (broker), ou directement depuis la VM hôte (Centos) si on n’a pas installé la raspberry.

Publier un message :

* Récupérer l’adresse ip du serveur (l’image toke/mqtt doit être lancée sur un container) : docker inspect mqtt (mqtt est le nom du container à inspecter, voir plus bas docker-compose.yml)
* mosquitto\_pub -h 172.18.0.2 -t hello-mqtt -m "[{""room"" : ""digiroom"", ""temp"" : ""28""}]"
  + –h pour l’ip du broker
  + –t le nom du topic à alimenter
  + –m le message à envoyer

## Kafka

La mise en place de l’image pour kafka est sans doute la plus laborieuse. Il faut en effet récupérer l’image qui contient kafka, puis les plugins pour kafka depuis le web, les extraire (donc installer **unzip**), créer un fichier de configuration pour le plugin (il y aura peut-être un dockerfile pour automatiser le tout).

Instructions d’installation :

* Récupérer l’image de base : **docker pull spotify/kafka**
* Lancer l’image avec un nom de container simple :

**docker run -it --name kafka spotify/kafka bash**

* Depuis le shell récupérer un éditeur de texte et unzip :
  + **apt-get update && apt-get install unzip**
  + **apt-get install vim**
* récupérer et décompresser les plugins nécessaires (mqtt connect et sa dépendance eclipse paho)
  + cd **/opt/kafka\_2.11-0.10.1.0/libs/**
  + télécharger : **wget https://howtoprogram.xyz/wp-content/uploads/2016/07/kafka-mqtt-bin.zip**
  + décompresser : **unzip kafka-mqtt-bin.zip** => résultat : 2 archives .jar : **kafka-connect-mqtt-1.0-SNAPSHOT.jar** et **org.eclipse.paho.client.mqttv3-1.0.2.jar**
  + facultatif : supprimer l’archive : **rm kafka-mqtt-bin.zip**
* fichier de configuration pour le connecteur MQTT :
  + créer et ouvrir le fichier config **mqtt.properties** : **touch mqtt.properties** (toujours dans le répertoire libs)
  + copier le contenu suivant et enregistrer :

name=mqtt

connector.class=com.evokly.kafka.connect.mqtt.MqttSourceConnector

tasks.max=1

kafka.topic=word-count

mqtt.client\_id=mqtt-kafka-123456789

mqtt.clean\_session=true

mqtt.connection\_timeout=30

mqtt.keep\_alive\_intervall=60

mqtt.server\_uris=tcp://mqtt:1883

mqtt.topic=hello-mqtt

N’oubliez surtout pas de commit ! Pour cela il faut ouvrir un nouveau terminal en ssh sur la VM et faire la commande suivante : **docker commit kafka kafkadock** (cela crée une nouvelle image avec les modifs que l’on a faites, elle s’appellera kafkadock, c’est plus simple que **spotify/kafka.**

## Apache Spark streaming

* Récupérer l’image pour spark streaming : **docker pull antlypls/spark:1.5.1** (patienter !)
* Lancer l’image : **docker run -it --name spark antlypls/spark bash**
* Créer le dossier qui contient l’application wordcount (elle sert à tester la partie spark streaming) : **mkdir app**
* ouvrir un deuxième terminal sur la vm hôte (centos) pour récupérer l’appli : **git clone https://github.com/antlypls/spark-kafka-docker-demo.git** && **cd spark-kafka-docker**
* **sbt assembly** pour compiler l’appli. Il faudra sûrement installer **sbt** (simple build tool)
  + si l’assembly ne marche pas :
    - **cd ..**
    - **git clone https://github.com/AyoubNas/spark**
    - **docker cp spark/direct\_kafka\_word\_count.jar spark:/app**
  + si l’assembly marche : **docker cp target/scala-2.10/ spark:/app**

Commit en ouvrant un nouveau terminal sur la VM hôte puis **docker commit spark sparkdock.**

## Elasticsearch

Cette étape est très simple :

* Récupérer l’image : **docker pull docker.elastic.co/elasticsearch/elasticsearch:5.5.**1
* Récupérer le hashcode de l’image (le nom ne fonctionne pas) : **docker images | grep elastic**
  + Le hashcode est dans la colonne « image ID »
* Lancer l’image : **docker run –it –name elasticsearch hashcode bash** (remplacer hashcode)
* Désactiver la sécurité (on est en local) :
  + **vi config/elasticsearch.yml**
  + Rajouter la ligne : **xpack.security.enabled: false**
  + enregistrer

Toujours commit : **docker commit elasticsearch elastic**

## Kibana

Pour kibana il suffit de récupérer l’image et elle est fonctionnelle : **docker pull docker.elastic.co/kibana/kibana:5.5.1**

On va la renommer. On la lance et on commit (je n’ai pas plus simple, voir docker tag aussi) :

* Récupérer le hashcode avec **docker images | grep kibana**
* **Docker run –it –name kibana hashcode bash**
* Depuis l’hôte : **docker commit kibana kibana**

## Logstash

Pour logstash on récupère l’image et on crée le fichier pour le premier pipeline (input – (filter) – output) :

* Récupérer l’image : **docker pull docker.elastic.co/logstash/logstash:5.5.1**
* lancer : **docker run –it –name logstash hashcode bash**
* installer le plugin logstash-input-kafka :
  + **cd /bin && logstash-plugin install logstash-input-kafka**
* créer le pipeline grâce à un fichier config
  + **touch opt/logstash/config.conf**
  + **vi opt/logstash/config.conf**
  + contenu du fichier :

input {

kafka {

bootstrap\_servers => "**172.18.0.5**:9092"

topics => ["word-count"]

}

}

filter {

}

output {

elasticsearch {

hosts => "elasticsearch:9200"

index => "wordcount"

}

stdout {}

}

* enregistrer

Commit depuis l’hôte : **docker commit logstash logstashdock**

# Lancement des containers

## Docker-compose

Pour lancer l’ensemble des containers on utilise docker-compose. Grâce à un fichier de configuration on va lancer d’une seule commande tous les containers et ils communiqueront entre eux grâce à leur nom et non pas leur adresse ip.

Contenu du fichier **docker-compose.yml**:

version: '2'

services:

mqtt:

image: toke/mosquitto

container\_name: mqtt

ports:

- "3306"

kafka:

image: kafkadock

container\_name: kafka

command: /bin/bash

ports:

- "2181"

- "9092"

links:

- mqtt

tty: true

spark:

image: sparkdock

container\_name: spark

command: /bin/bash

links:

- kafka

- elasticsearch

tty: true

elasticsearch:

image: elastic

container\_name: elasticsearch

command: /bin/bash

tty: true

kibana:

image: kibana

container\_name: kibana

command: /bin/bash

tty: true

links:

- elasticsearch

ports:

- "5601"

logstash:

image: logstashdock

container\_name: logstash

command: bin/bash

tty: true

links:

- elasticsearch

- kafka

Attention : Pour chaque service, il faut que le champ **image** corresponde au nom de l’image que l’on a mis en place. Pour renommer une image, on la lance en spécifiant un nom de container et on commit le container avec le nouveau nom de l’image. Exemple :

**docker run –it –name spark antlypls/spark bash**

Depuis l’hôte : **docker commit spark sparkdock**

## Lancement des tâches dans les containers

Tout d’abord on se place dans le dossier contenant le fichier **docker-compose.yml** et on fait :

**docker-compose up**

Pour faire fonctionner l’architecture il va falloir lancer sur chaque container les tâches correspondantes.

### Mosquitto

Pour le container mosquitto, il n’y a rien à faire si ce n’est récupérer son adresse ip pour ensuite publier des messages depuis l’hôte ou la raspberry : **docker inspect mqtt** (depuis l’hôte).

### Kafka

Ouvrez un shell sur le container kafka : **docker exec –it kafka bash** et rendez-vous dans **/opt/kafka\_2.11(...)** (commande **cd**).

Pour démarrer le serveur kafka il y a trois commandes à effectuer :

* **./bin/zookeeper-server-start.sh config/zookeeper.properties &**
* **./bin/kafka-server-start.sh config/server.properties &**
* **./bin/connect-standalone.sh config/connect-standalone.properties libs/mqtt.properties**

Le serveur kafka est abonné au topic mqtt et a créé le topic kafka. Il faut maintenant démarrer logstash qui va se charger de récupérer les messages et les stocker sur elasticsearch.

### Elasticsearch

**sudo sysctl -w vm.max\_map\_count=262144**

Démarrer le serveur elasticsearch :

* **Docker exec –it elasticsearch bash**
* **elasticsearch**

### kibana

Avant de démarrer le serveur kibana il faut lui indiquer où est le serveur logstash, car il cherche sur le localhost par défaut :

* docker exec –it kibana bash
* vi config/kibana.yml
* chercher la ligne qui commence par **elasticsearch.url:** et remplacer par **elasticsearch.url: http://elasticsearch:9200**
* lancer le serveur : **kibana**
* pour utiliser kibana sur le navigateur (donc sous windows)
  + depuis la vm : **docker ports kibana** noter le port
  + dans les configurations de la VM : rediriger ce port vers un autre port de votre choix. Par exemple, si le résultat de la commande est 22076, rediriger le port 22076 de la VM vers un autre port (on peut mettre le même).
  + Depuis le navigateur : http://ip\_de\_la\_vm:port

### Logstash

Pour logstash il suffit de démarrer le pipeline que l’on a créé grâce au fichier de **config.conf**:

* **docker exec –it logstash bash**
* **logstash –f opt/logstash/config.conf**

### spark

L’image spark est prête mais je ne me suis pas penché sur le fonctionnement. Lorsqu’on démarre les containers, sur celui de spark il y a un serveur qui se met en route automatiquement. On peut lancer l’application word-count qui compte les mots dans les messages kafka :

* **docker exec –it spark bash**
* **~/Development/spark-1.6.3-bin-hadoop2.6/bin/spark-submit --class "com.supergloo.WeatherDataStream" --master spark://todd-mcgraths-macbook-pro.local:7077 target/scala-2.10/kafka-streaming-assembly-1.0.jar**
* Publier sur mosquitto pour voir le résultat (les mots sont comptés périodiquement)

# Conclusion

Normalement vous avez pu vous familiariser avec chaque technologie en les mettant en place. Si quelque chose ne fonctionne pas vous pouvez récupérer toutes les images depuis le docker hub :

* Mosquitto : **docker pull *toke/mosquitto***
* Kafka : **docker pull digiroom/kafka**
* Spark : **docker pull digiroom/spark**
* Logstash : **docker pull digiroom/logstash**
* Elasticsearch : **docker pull digiroom/elasticsearch**
* Kibana : **docker pull digiroom/kibana**

Pensez à adapter les **‘image\_name’** dans le fichier **docker-compose.yml**

Adresse du docker hub : <https://hub.docker.com/r/digiroom/>

Pour push une nouvelle image :

* Docker login
  + Username : digiroom
  + Pswd : Sopra123\*
* Docker tag #image\_name digiroom/#image\_name
  + Pour qu’elle soit poussée au bon endroit
* Docker push digiroom/#image\_name
* Exemple : pour pousser une image qui s’appelle kafka :
  + Docker login
  + Docker tag kafka digiroom/kafka
  + Docker push digiroom/kafka
  + L’image est dispo sous <https://hub.docker.com/r/digiroom/kafka>