Rapport - Fil Rouge

Génération message automatique dans bon virtual host rabbitmq
TP1
Redescente d'information à demander à Pierre Courbin.
TP1
Index TP2

> Kilian Weydert
> 12 janvier 2020
> *Développement d'applications et webservices pour l'IoT*

Introduction

Ce document est un rapport sur le travail fourni lors des différents TPs.
Le contexte est le suivant :

- Nous sommes une entreprise qui propose de rassembler les données issues d'objets domotiques/énergétiques afin de proposer des dashboards de suivi à nos clients. Nous pensons que plus on connait et on visualise ses propres données, plus on est en enclin à faire attention à sa consommation notamment. Nous voulons ainsi apporter notre solution pour tenter d'aider les gens à être plus sobres énergétiquement ;
- Nous avons déjà 2 clients (Annexe A.1, Annexe A.2) ;
- Nous avons déjà bien avancé sur une première structuration des différentes informations récupérable (Annexe B) ;
- Nous ne produisons pas de capteurs ou d'actionneurs. Nous nous appuyons sur un réseau de partenaires qui nous permettent de récupérer les données de nos clients sans avoir à installer quoi que ce soit chez lui.

Table des matières

Ingestion des données

Technologie utilisée : **RabbitMQ**

Définition de l'architecture

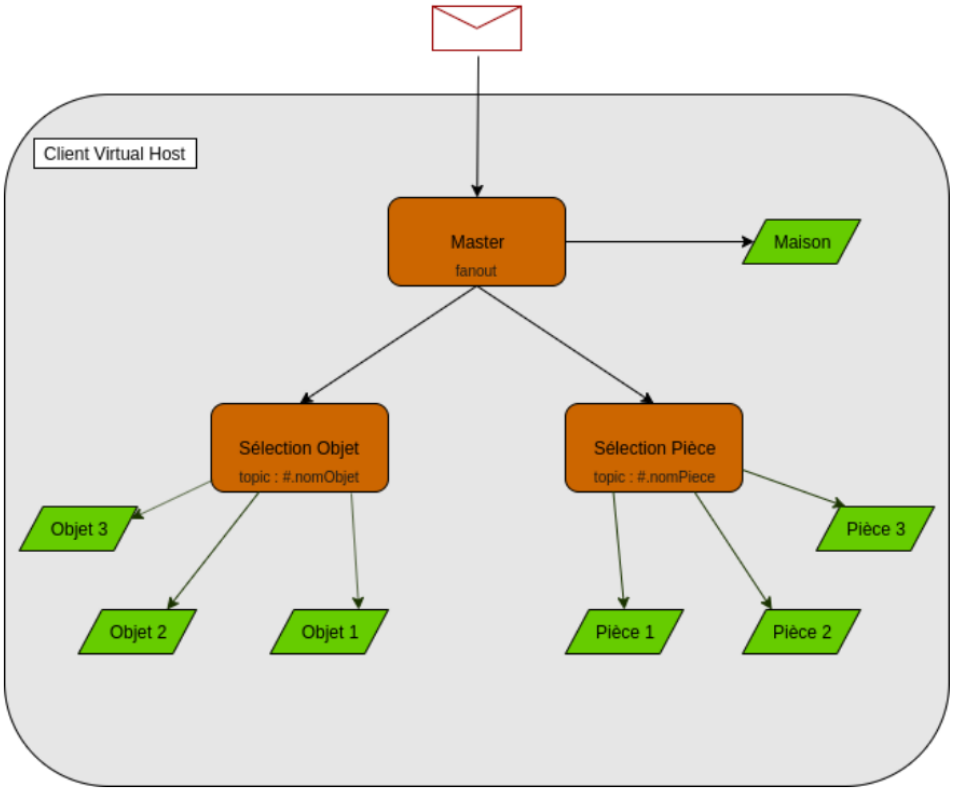
L'ingestion des données consiste en la récupération des données envoyées par l'ensemble des objets connectés. Nous allons trier les données dès l'ingestion afin d'obtenir des *queues* pour chaque :

- Client
- Pièce du client

- Objet du client

  Une telle architecture nous permet de faciliter le futur *stockage* et *traitements* des données.

Chaque client aura son propre serveur virtuel pour une meilleure séparation des données.



Pour chaque chaque client, on crée :

- Un host virtuel ;
- Un exchange *Master* connecté à une queue *Maison* ainsi qu'à un exchange *selectionPiece* et *selectionObjet* ;
- Un exchange *Selection Piece* chargé de distribuer le message dans la queue correspondant à celle de la pièce ;
- Un exchange *Selection Objet* chargé de distribuer le message dans la queue correspondant à celle de l'objet.
- Les objets utiliseront la routing key : *nomPiece.nomObjet*

Déploiement de l'architecture

docker-compose.yml permettant de déployer un serveur RabbitMQ :

```
version: '3.7'
services:

    rabbitmq:
        image: "rabbitmq:3-management"
        hostname: "rabbitmq"
        environment:
            RABBITMQ_ERLANG_COOKIE: "SWQOKODSQALRPCLNMEQG"
            RABBITMQ_DEFAULT_USER: "rabbitmq"
            RABBITMQ_DEFAULT_PASS: "rabbitmq"
            RABBITMQ_DEFAULT_VHOST: "/"
        ports:
            - "15672:15672"
            - "5672:5672"
        networks:
            - iot-labs
        labels:
```

```
            NAME: "rabbitmq"

networks:
    iot-labs:
        external: true
```

Nous allons implémenter l'architecture nécessaire pour notre Client 2.
C'est à dire que nous allons créer :

- Les exchanges : *Master*, *Selection Piece*, *Selection Objet*
- Les queues :
  - *Maison*
  - *Piece Principale* ; *Salle de Bain*
  - *ampoule_principale* ; *detecteurPresence_principale* ; *capteurTemperature_principale* ; *radiateur_principale* ; *capteurPuissanceBallonEau* ; *radiateur_salleDeBain* ; *ampoule_salleDeBain*.
- Les bindings (*comme sur le schéma ici*)

Virtual host	Name	Type	Features	Message rate in	Message rate out
client2	**(AMQP default)**	direct	D		
client2	**Master**	fanout	D		
client2	**Selection Objet**	topic	D		
client2	**Selection Piece**	topic	D		
client2	**amq.direct**	direct	D		
client2	**amq.fanout**	fanout	D		
client2	**amq.headers**	headers	D		
client2	**amq.match**	headers	D		
client2	**amq.rabbitmq.trace**	topic	D I		
client2	**amq.topic**	topic	D		

> Liste Exchanges

Overview					Messages			Message rates		
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
client2	**Maison**	classic	D Args	idle	0	0	0			
client2	**Piece Principale**	classic	D Args	idle	0	0	0			
client2	**Salle de Bain**	classic	D Args	idle	0	0	0			
client2	**ampoule_principale**	classic	D Args	idle	0	0	0			
client2	**ampoule_salleDeBain**	classic	D Args	idle	0	0	0			
client2	**capteurPuissanceBallonEau**	classic	D Args	idle	0	0	0			
client2	**capteurTemperature_principale**	classic	D Args	idle	0	0	0			
client2	**detecteurPresence_principale**	classic	D Args	idle	0	0	0			
client2	**radiateur_principale**	classic	D Args	idle	0	0	0			
client2	**radiateur_salleDeBain**	classic	D Args	idle	0	0	0			

> Liste Queues

This exchange

⇓

To	Routing key	Arguments	
Maison			Unbind
Selection Objet			Unbind
Selection Piece			Unbind

> Binding Master

From	Routing key	Arguments	
Master			Unbind

⇓

This exchange

⇓

To	Routing key	Arguments	
ampoule_principale	*.ampoule_principale		Unbind
ampoule_salleDeBain	*.ampoule_salleDeBain		Unbind
capteurPuissanceBallonEau	*.capteurPuissanceBallonEau		Unbind
capteurTemperature_principale	*.capteurTemperature_principale		Unbind
detecteurPresence_principale	*.detecteurPresence_principale		Unbind
radiateur_principale	*.radiateur_principale		Unbind
radiateur_salleDeBain	*.radiateur_salleDeBain		Unbind

> Binding SelectionObjet

From	Routing key	Arguments	
Master			Unbind

⇓

This exchange

⇓

To	Routing key	Arguments	
Piece Principale	Piece Principale.*		Unbind
Salle de Bain	Salle de Bain.*		Unbind

> Binding SelectionPiece

Découvrir la réception et l'envoi de messages

Test à la main

Essayons de générer un message à partir de l'exchange *master* ayant une *routing_key* = Salle de Bain.ampoule_salleDeBain. Si notre architecture fonctionne, il devrait aller dans les queues *ampoule_salleDeBain*, *Salle de Bain* et *Maison*.

Bingo !

Overview					Messages			Message rates		
Virtual host	**Name**	**Type**	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**
client2	**Maison**	classic	D Args	idle	1	0	1	0.20/s	0.00/s	0.00/s
client2	**Piece Principale**	classic	D Args	idle	0	0	0			
client2	**Salle de Bain**	classic	D Args	idle	1	0	1	0.20/s	0.00/s	0.00/s
client2	**ampoule_principale**	classic	D Args	idle	0	0	0			
client2	**ampoule_salleDeBain**	classic	D Args	idle	1	0	1	0.20/s	0.00/s	0.00/s
client2	**capteurPuissanceBallonEau**	classic	D Args	idle	0	0	0			
client2	**capteurTemperature_principale**	classic	D Args	idle	0	0	0			
client2	**detecteurPresence_principale**	classic	D Args	idle	0	0	0			
client2	**radiateur_principale**	classic	D Args	idle	0	0	0			
client2	**radiateur_salleDeBain**	classic	D Args	idle	0	0	0			

Script d'envoi

```
#!/usr/bin/env python
import pika

credentials = pika.PlainCredentials('rabbitmq', 'rabbitmq')
exchange='Master'
routing_key='Salle de Bain.ampoule_salleDeBain'
body="1"

# Connexion | ConnextionParameters(host, port, virtualHost, credentials)
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost', 5672, 'client2', credentials))
channel = connection.channel()

# Envoi
channel.basic_publish(exchange, routing_key, body)

print(" [x] Sent '" + body + "' !")

connection.close()
```

Script de réception

```
#!/usr/bin/env python
import pika

credentials = credentials = pika.PlainCredentials('rabbitmq', 'rabbitmq')

def callback(ch, method, properties, body):
    print(" [x] Received %s" % body)

# Connexion [BlockingConnection(host, port, virtualHost, credentials)]
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost', 5672, 'client2', credentials))
channel = connection.channel()

# Réception
channel.basic_consume(queue = 'ampoule_salleDeBain',
                      auto_ack = True,
                      on_message_callback = callback)
```

```
print(' [*] Waiting for messages. To exit press CTRL+C')

# Écoute
channel.start_consuming()
```

Automatisation du déploiement

Le script suivant nous permet de créer une architecture fonctionnelle pour l'ajout de nouveaux clients. Pas de panique, si le monde s'écroule, ce script se suffit à lui-même, aucune configuration préalable n'est nécessaire :).

```
#!/usr/bin/env python
import pika
from os import system

credentials = credentials = pika.PlainCredentials('rabbitmq', 'rabbitmq')
nomClient = ""
nomPiece = ""
nomObjet = ""
listeObjets = []
listePieces = []

def ajouterClient(nomClient, listeObjets, listePieces):

    # Creation Virtual Host client avec API REST
    system('curl -u rabbitmq:rabbitmq -X PUT http://localhost:15672/api/vhosts/' + nomClient)

    connection = pika.BlockingConnection(pika.ConnectionParameters('localhost', 5672, nomClient, credentials)
    channel = connection.channel()

    # Creation des 3 exchanges
    channel.exchange_declare(exchange = "Master", exchange_type="fanout")
    channel.exchange_declare(exchange = "Selection Piece", exchange_type="topic")
    channel.exchange_declare(exchange = "Selection Objet", exchange_type="topic")

    # Creation de la queue "Maison"
    channel.queue_declare(queue = "Maison")

    # Binding des 3 exchanges et de la queue "Maison"
    channel.exchange_bind(destination = "Selection Piece", source = "Master")
    channel.exchange_bind(destination = "Selection Objet", source = "Master")
    channel.queue_bind(exchange = "Master", queue = "Maison")

    # Creation et Binding des queues pour chaque piece
    for piece in listePieces:
        channel.queue_declare(queue = piece)
        channel.queue_bind(exchange = "Selection Piece", queue = piece, routing_key = piece+".*")

    # Creation et Binding des queues pour chaque objet
    for objet in listeObjets:
        channel.queue_declare(queue = objet)
        channel.queue_bind(exchange = "Selection Objet", queue = objet, routing_key = "*."+objet)

def ajouterUnObjet():
    flag = True

    while flag:
        print("Nom de l'objet : ")
        nomObjet = input()
        listeObjets.append(nomObjet)
        response = ""

        while response != "y" and response != "n":
            print(nomClient + " a-t-il d'autres objets ? [y/n]")
            response = input()
            if response == "y":
                flag = True
            elif response == "n":
                flag = False
```

```
def ajouterUnePiece():
    for objet in listeObjets:
        print("Dans quelle piece se trouve " + objet + " ? : ")
        nomPiece = input()
        listePieces.append(nomPiece)

# -----------------------------------------------------------------------------------------

# PENSEZ A METTRE DES "" DANS LES INPUTS

print("")
print("Bienvenue dans l'utilitaire d'ajout client dans votre serveur Rabbitmq (localhost).")
print("-----------------------------------------------------------------------------------")
print("")

print("Renseignez le nom du client : ")
nomClient = input()

ajouterUnObjet()
ajouterUnePiece()

print("-----------------------------------------------------------------------------------")
print("Architecture OK")
print("")

ajouterClient(nomClient, listeObjets, listePieces)
```

Génération automatique de messages

Nous allons automatiser la génération automatiques de données de 3 capteurs à l'aide de 3 *docker-compose.yml* :

capteurTemperature :

```
version: '3.7'
services:
    moke1:
        image: "pcourbin/mock-data-generator:latest"
        hostname: "moke1"
        environment:
            SENZING_SUBCOMMAND: random-to-rabbitmq
            SENZING_RANDOM_SEED: 0
            SENZING_RECORD_MIN: 1
            SENZING_RECORD_MAX: 100
            SENZING_RECORDS_PER_SECOND: 1
            SENZING_RABBITMQ_HOST: rabbitmq
            SENZING_RABBITMQ_PASSWORD: rabbitmq
            SENZING_RABBITMQ_USERNAME: rabbitmq
            SENZING_RABBITMQ_QUEUE: capteurTemperature
            MIN_VALUE: 15
            MAX_VALUE: 28
            SENZING_DATA_TEMPLATE: '{"nom":"capteurTemperature","date":"date_now", "value":"float"}'
        tty: true
        labels:
            NAME: "moke1"
        networks:
            - iot-labs
networks:
    iot-labs:
        external: true
```

Stockage des données

Technologie utilisé : **MongoDB**

1. Définition de l'architecture
2. Déploiement de l'architecture
3. Découvrir l'envoi et la réception de messages
4. Automatisation du déploiement
5. Génération automatique de messages
6. Trouver la bonne clé de sharding

VH aussi dans mongoDB ?

Définition de l'architecture

Chaque client dispose de sa propre base de données. Afin d'offrir un traitement efficace, nous stockons et organisons les données par maison, pièce et objet dans des collections. Les données étant d'ores et déjà organisé de cette manière sur nos serveurs RabbitMQ, il suffit de créer une collection *MongoDB* par queue *RabbitMQ*.

En résumé

- 1 database par client
- 1 collection par queue *RabbitMQ*
- Les données envoyées dans la collection auront le template prédéfini dans l'Annexe B

Déploiement de l'architecture

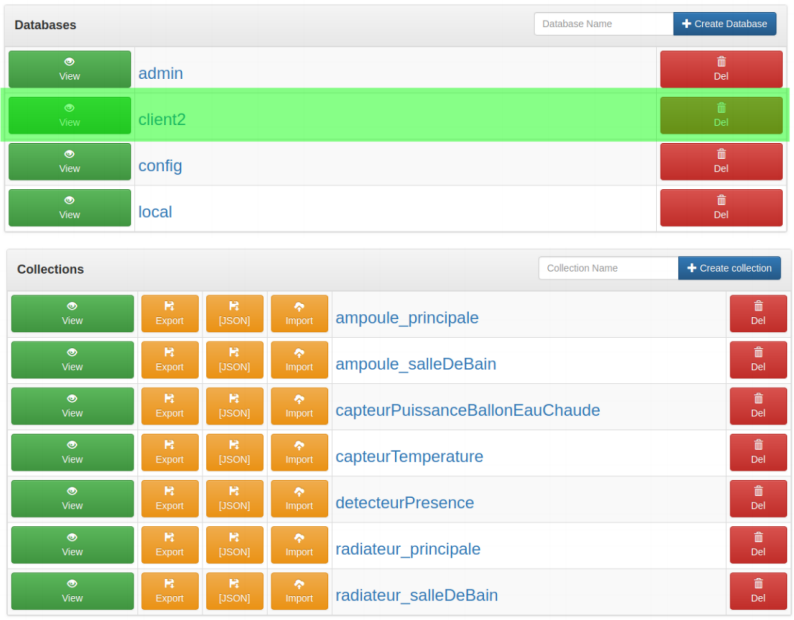
docker-compose.yml permettant de déployer un serveur MongoDB :

```
version: '3.1'
services:
    mongo:
        image: mongo:4.2.0-bionic
        hostname: "mongo"
        restart: always
        labels:
            NAME: "mongo"
        networks:
            - iot-labs
        ports:
            - 27017:27017

    mongo-express:
        image: mongo-express:0.49.0
        hostname: "mongo_express"
        restart: always
        ports:
            - 8081:8081
        environment:
             ME_CONFIG_MONGODB_SERVER: mongo
        labels:
            NAME: "mongo_express"
        networks:
            - iot-labs

networks:
    iot-labs:
        external: true
```

Implémentation de l'architecture de Client 2 :



Découvrir l'envoi et la réception de messages

Script d'envoi

```
#!/usr/bin/env python
import pymongo
import datetime

from pymongo import MongoClient

# Connexion
client = MongoClient()

# Creation de la database "client2"
db = client.client2

# Creation de la collection "ampoule_salleDeBain" dans notre database
collection = db.ampoule_salleDeBain

# Creation du document JSON
post = {
"nom": "ampoule_salleDeBain",
"date": datetime.datetime.utcnow(),
"valeur": 0
}

# Envoi du document JSON
post_id = collection.insert_one(post).inserted_id

# Affichage de l'id du document cree par MongoDB
print(post_id)
```

Script de lecture de données

```
#!/usr/bin/env python

import pymongo
import pprint

from pymongo import MongoClient

# Connexion
```

```
client = MongoClient()

# Acces a notre database "client2"
db = client.client2

# Acces a notre collection "ampoule_salleDeBain" au sein de notre database
collection = db.ampoule_salleDeBain

# Affichage de la requete
pprint.pprint(collection.find_one())
```

Scripts avancés

Récupérer la dernière valeur connu d'un capteur

```
#!/usr/bin/env python
import pymongo
import pprint

from pymongo import MongoClient

# Connexion
client = MongoClient()

# Acces a notre database "client2"
db = client.client2

# Acces a notre collection "ampoule_salleDeBain" au sein de notre database
collection = db.ampoule_salleDeBain

# Requete
results = collection.aggregate([{"$group": {"_id": "1", "lastRegistered": {"$last": "$date"}}}
])

# Affichage de la requete
for result in results:
     print(result)
```

Calculer la moyenne des valeurs d'un capteur entre deux dates fixées

```
#!/usr/bin/env python
import pymongo
import pprint
import datetime

from pymongo import MongoClient

# Connexion
client = MongoClient()

# Acces a notre database "client2"
db = client.client2

# Acces a notre collection "capteurTemperature" au sein de notre database
collection = db.capteurTemperature

sensor_name = "capteurTemperature"
start_date = datetime.datetime(2019, 1, 1, 1, 1, 1, 0)
end_date = datetime.datetime.utcnow()

# Requete
results = collection.aggregate(
                            [
                            {"$match": {"nom": sensor_name}},
                            {"$match": {"date": {"$gte": start_date}}},
                            {"$match": {"date": {"$lte": end_date}}},
                            {"$group": {"_id": "$nom", "moyenne": {"$avg": "$temperature"}}}
                           ])
```

```
# Affichage de la requete
for result in results:
    print(result)
```

Récupérer la valeur minimale d'un capteur entre deux dates fixées

```
#!/usr/bin/env python
import pymongo
import pprint
import datetime

from pymongo import MongoClient

# Connexion
client = MongoClient()

# Acces a notre database "client2"
db = client.client2

# Acces a notre collection "capteurTemperature" au sein de notre database
collection = db.capteurTemperature

sensor_name = "capteurTemperature"
start_date = datetime.datetime(2019, 1, 1, 1, 1, 1, 0)
end_date = datetime.datetime.utcnow()

# Requete
results = collection.aggregate([{"$match": {"date": {"$gt" : start_date}}},
                                {"$match": {"date": {"$lt" : end_date}}},
                                {"$group": {"_id": "1", "ValeurMinimale": {"$last": "$temperature"}}}
                               ])

# Affichage de la requete
for result in results:
    print(result)
```

Automatisation du déploiement

Le script suivant nous permet de créer une architecture fonctionnelle pour l'ajout de nouveaux clients. Pas de panique, si le monde s'écroule, ce script se suffit à lui-même, aucune configuration préalable n'est nécessaire (une fois de plus) :).

Génération automatique de messages

Nous allons utiliser une image docker pour connecter une *queue* de messages à une *collection* :

Attention à la propriété **AMQPHOST**, il faut préciser l'utilisateur utilisé dans RabbitMQ.

```
#https://github.com/marcelmaatkamp/docker-rabbitmq-mongodb
version: '3.7'
services:
    amqp2mongo1:
        image: "marcelmaatkamp/rabbitmq-mongodb"
        hostname: "amqp2mongo1"
        environment:
            AMQPHOST: 'amqp://guest:guest@rabbitmq'
            MONGODB: 'mongodb://mongo/client2'
            MONGOCOLLECTION: 'Maison'
            TRANSLATECONTENT: 'true'
        command: 'Maison'
        tty: true
        labels:
            NAME: "amqp2mongo1"
        networks:
            - iot-labs
```

```
        restart: always
networks:
    iot-labs:
        external: true
```

Trouver la bonne clé de sharding

Data Routing

1. Déploiement Nifi
2. Récupération des données d'une API
3. Récupération données CSV sur serveur FTP

Déploiement Nifi

```
version: '3.7'
services:
    nifi:
        image: "apache/nifi:latest"
        hostname: "nifi"
        ports:
            - "8083:8080"
        networks:
            - iot-labs
        labels:
            NAME: "nifi"
networks:
    iot-labs:
        external: true
```

Récupération des données d'une API

Récupération données CSV sur serveur FTP

Annexe

Annexe A : Descriptif client

Lorsqu'un objet du même type est présent dans différentes pièces, nous ajoutons le nom de la pièce en suffixe au nom de l'objet pour mieux les distinguer.

A.1 : Client 1

Type	Nom Objet	Pièce
1 détecteur d'ouverture de porte	detecteurPorte_entree	Entrée
1 capteur d'activation de lumière	ampoule_entree	Entrée
2 détecteurs d'ouverture de porte	detecteurPorte1_salon detecteurPorte2_salon	Salon
2 capteurs d'activation de lumière	ampoule1_salon ampoule2_salon	Salon

Type	Nom Objet	Pièce
1 détecteur de présence	detecteurPresence_salon	Salon
1 capteur de température	capteurTemperature_salon	Salon
2 capteurs d'activation de chauffage	radiateur1_salon radiateur2_salon	Salon
2 capteurs de puissance énergétique	capteurPuissanceRadiateur1_salon capteurPuissanceRadiateur2_salon	Salon
2 détecteurs d'ouverture de porte	detecteurPorte1_chambre detecteurPorte2_chambre	Chambre
1 capteur d'activation de lumière	ampoule_chambre	Chambre
1 détecteur de présence	detecteurPresence_chambre	Chambre
1 capteur de température	capteurTemperature_chambre	Chambre
1 capteur d'activation de chauffage	radiateur_chambre	Chambre
1 détecteur de présence	detecteurPresence_cuisine	Cuisine
1 capteur de température	capteurTemperature_cuisine	Cuisine
1 capteur d'activation de chauffage	radiateur_cuisine	Cuisine
1 capteur d'activation de prise électrique	grillePain	Cuisine
1 capteur de consommation énergétique	capteurConsommationGrillePain	Cuisine
2 capteurs de puissance énergétique	capteurPuissanceFour capteurPuissanceMachineLaver	Cuisine
1 capteur de puissance énergétique	capteurPuissanceBallonEau	Salle de Bain
1 capteur d'activation de chauffage	radiateur_salleDeBain	Salle de Bain
1 capteur d'activation de lumière	ampoule_salleDeBain	Salle de Bain

A.2 : Client 2

Type		Pièce
1 détecteur de présence	detecteurPresence	Principale
1 capteur d'activation de lumière	ampoule_principale	Principale
1 capteur de température	capteurTemperature	Principale
1 capteur d'activation de chauffage	radiateur_principale	Principale
1 capteur d'activation de lumière	ampoule_salleDeBain	Salle de Bain
1 capteur de puissance énergétique	capteurPuissanceBallonEau	Salle de Bain
1 capteur d'activation de chauffage	radiateur_salleDeBain	Salle de Bain

Annexe B : Définition des données

1. Détecteur d'ouverture de porte

- nom : *string*
- date : *string*
- valeur : *integer (0 : fermé, 1 : ouvert)*

2. Détecteur de présence

- nom : *string*
- date : *string*
- valeur : *integer (0 : présence non détectée, 1 : présence détectée)*

3. Capteur d'activation de lumière

- nom : *string*
- date : *string*
- valeur : *integer (0 : fermé, 1 : ouvert)*

4. Capteur de luminosité

- nom : *string*
- date : *string*
- luminosite : *double (Lux)*

5. Capteur d'activation de chauffage

- nom : *string*
- date : *string*
- valeur :
  - *0 : Arrêt*
  - *1 : Confort*
  - *2 : Confort -1°C*
  - *3 : Confort -2°C*
  - *4 : Eco*
  - *5 : Hors gel*

6. Capteur d'activation de climatisation

- nom : *string*
- date : *string*
- temperatureCible : *double (°C)*

7. Capteur de température

- nom : *string*
- date : *string*
- temperature : *double (°C)*

8. Capteur d'activation de prise électrique

- nom :*string*
- date : *string*
- valeur : *integer (0 : fermé, 1 : ouvert)*

9. Capteur de consommation énergétique

- nom : *string*
- date : *string*

- consommation : *double (kWh)*

10. Capteur de puissance énergétique

- nom : *string*
- date : *string*
- puissance : *integer (kWh)*

11. Capteur de position

- nom : *string*
- date : *string*
- longitude : *string*
- latitude : *string*

12. Capteur position volet butée

- nom : *string*
- date : *string*
- valeur : *integer (0 : fermé, 1 : ouvert)*

13. Capteur ordre ouverture volet

- nom : *string*
- date : *string*
- ouverture : *integer (0% : fermé, 100% : ouvert)*