Créer Votre première application KAFKA

Maintenant qu'on a vu les principaux concepts de Kafka, on peut commencer à construire une application qui tire réellement parti de ses capacités. Nous allons nous intéresser aux solutions de locations de vélo à la demande, comme Vélib à Paris ou Vélô à Toulouse. Il existe une API qui permet de contrôler l'état des stations de vélos dans un grand nombre de villes, en France, en Europe et dans le monde : https://developer.jcdecaux.com. Nous allons utiliser cette API pour observer en temps réel les locations à chaque station.

Obtenir une clé d'API

Pour commencer, vous allez devoir récupérer une clé d'API en créant un compte sur https://developer.jcdecaux.com/#/signup. Une fois que vous aurez créé votre compte, vous disposerez d'une clé d'API affichée dans votre compte utilisateur. Si votre clé d'API est "XXX", vous pouvez vérifier qu'elle fonctionne correctement en récupérant la liste de toutes les stations à l'aide de la commande suivante :

```
$ curl https://api.jcdecaux.com/vls/v1/stations?apiKey=XXX
```

Vous devriez alors obtenir en réponse un gros morceau de JSON assez indigeste... Pour le rendre plus lisible, vous pouvez rediriger l'output de la commande précédente vers un "prettifier" de JSON :

```
$ curl https://api.jcdecaux.com/vls/v1/stations?apiKey=XXX | python -m json.tool

[

    {

        "address": "MAZARGUES - ROND POINT DE MAZARGUES (OBELISQUE)",

        "available_bike_stands": 20,

        "available_bikes": 1,

        "banking": true,

        "bike_stands": 21,

        "bonus": false,

        "contract_name": "Marseille",

        "last_update": 1493734764000,

        "name": "9087-MAZARGUES",
```

```
        "number": 9087,
        "position": {
            "lat": 43.250903869637334,
            "lng": 5.403244616491982
        },
        "status": "OPEN"
    },
    ...
]
```

Comme on peut le voir, l'API nous fournit le nombre d'emplacements libres (`"available_bike_stands"`) dans chaque station. Si ce nombre augmente (respectivement : diminue) entre deux appels à l'API, c'est que des vélos ont été loués (resp. : déposés) dans la station. Nous allons mettre en place une application qui va afficher l'évolution de ce nombre d'emplacements disponibles, sous la forme suivante :

```
+1 MAZARGUES - ROND POINT DE MAZARGUES (OBELISQUE) (Marseille)
+14 Lower River Tce / Ellis St (Brisbane)
+2 2 RUE GATIEN ARNOULT (Toulouse)
+20 ANGLE ALEE ANDRE MURE ET QUAI ANTOINE RIBOUD (Lyon)
+14 Smithfield North (Dublin)
+28 52 RUE D'ENGHIEN / ANGLE RUE DU FAUBOURG POISSONIERE - 75010 PARIS (Paris)
+6 RUE DES LILAS ANGLE BOULEVARD DU PORT - 95000 CERGY (Cergy-Pontoise)
+6 San Juan Bosco - Santiago Rusiñol (Valence)
+21 AVENIDA REINA MERCEDES - Aprox. Facultad de Informática (Seville)
+6 Savska cesta 1 (Ljubljana)
+31 DE BROUCKERE - PLACE DE BROUCKERE/DE BROUCKEREPLEIN (Bruxelles-Capitale)
+7 BRICHERHAFF - AVENUE JF KENNEDY / RUE ALPHONSE WEICKER (Luxembourg)
```

```
...
```

Pour obtenir le résultat ci-dessus, on pourrait évidemment créer une simple application qui récolterait les données en provenance de l'API et afficherait les différences entre deux appels. Mais une telle application nécessiterait une quantité de mémoire proportionnelle au nombre de stations. Par ailleurs, le traitement des données provoquerait des délais dans les appels à l'API. Enfin, si une des étapes du traitement de données venait à échouer, la collecte des informations serait interrompue.

Bref, il est plus intéressant de passer par une plateforme de gestion de messages pour traiter les données de manière asynchrone.

Du producer...

Nous allons stocker les données relatives à chaque station de vélos dans des messages Kafka : chacun des éléments de la liste renvoyée par l'appel à l'API ci-dessus va être stocké dans Kafka sous la forme d'une chaîne de caractères au format JSON. Pour cela, nous créons le script`velib-get-stations.py`qui contient un producer Kafka.

`velib-get-stations.py`:

```
import json

import time

import urllib.request

from kafka import KafkaProducer

API_KEY = "XXX" # FIXME Set your own API key here

url = "https://api.jcdecaux.com/vls/v1/stations?apiKey={}".format(API_KEY)

producer = KafkaProducer(bootstrap_servers="localhost:9092")

while True:

    response = urllib.request.urlopen(url)

    stations = json.loads(response.read().decode())

    for station in stations:

        producer.send("velib-stations", json.dumps(station).encode())

    print("{} Produced {} station records".format(time.time(), len(stations)))

    time.sleep(1)
```

Dans ce script, on crée un producer qui va réaliser un appel à l'API toutes les secondes (`time.sleep(1)`). Chacune des stations contenues dans la réponse de

l'API sera redirigée vers le topic "`velib-stations`" de Kafka (`producer.send("velib-stations", ...)`).
Notez que pour exécuter ce script vous aurez besoin du package kafka-python que vous pouvez installer en exécutant :

```
$ pip install kafka-python
```

Nous aurons également besoin d'un cluster Kafka minimal, ainsi que d'un topic "velib-stations". Nous lançons un cluster et créons un topic à l'aide des commandes suivantes (comme expliqué dans le chapitre précédent) :

```
$ ./bin/zookeeper-server-start.sh ./config/zookeeper.properties


$ ./bin/kafka-server-start.sh ./config/server.properties


$ ./bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --
partitions 1 --topic velib-stations
```

Ceci fait, nous pouvons lancer notre producer qui va envoyer des messages à Kafka en continu :

```
python ./velib-get-stations.py
```

Notre topic Kafka se remplit progressivement et il nous reste à créer un consumer qui va lire les données de notre topic.

... au consumer

Nous allons également utiliser le package`kafka-python`pour développer un consumer. Le rôle de ce consumer est de stocker l'état des différentes stations et d'afficher un message lorsqu'une station change d'état.
`velib-monitor-stations.py`:

```
import json

from kafka import KafkaConsumer




stations = {}

consumer = KafkaConsumer("velib-stations", bootstrap_servers='localhost:9092',
group_id="velib-monitor-stations")

for message in consumer:

    station = json.loads(message.value.decode())

    station_number = station["number"]

    contract = station["contract_name"]
```

```
    available_bike_stands = station["available_bike_stands"]



    if contract not in stations:

        stations[contract] = {}

    city_stations = stations[contract]

    if station_number not in city_stations:

        city_stations[station_number] = available_bike_stands



    count_diff = available_bike_stands - city_stations[station_number]

    if count_diff != 0:

        city_stations[station_number] = available_bike_stands

        print("{}{} {} ({})".format(

            "+" if count_diff > 0 else "",

            count_diff, station["address"], contract

        ))
```

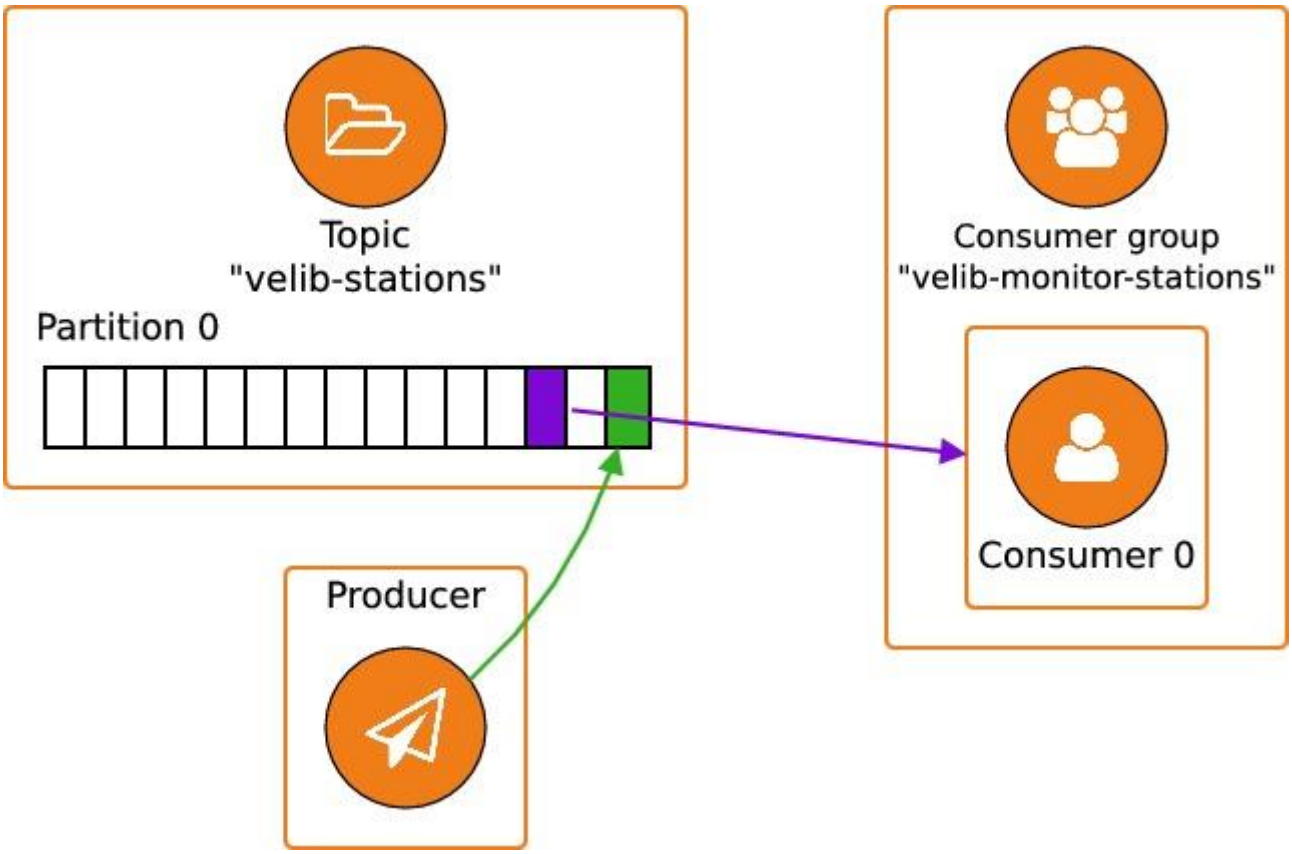
Dans ce script, nous créons un consumer Kafka pour le topic "`velib-stations`". Ce consumer fait partie du groupe "`velib-monitor-stations`" (`consumer = KafkaConsumer("velib-stations", ..., group_id="velib-monitor-stations")`). Il suffit de le lancer pour visualiser les fluctuations du nombre d'emplacements libres pour chaque station :

```
$ python ./velib-monitor-stations.py
```

Notez qu'on peut facilement ajouter un producer dans le code de notre consumer. Par exemple, si nous voulons être avertis par e-mail dès que la station la plus proche de chez nous devenait vide (`city_stations[station_number] == 0`), il vaudrait mieux ne *pas* ajouter un envoi d'e-mail au code de`velib-monitor-stations.py`. Si l'envoi d'e-mail prenait beaucoup de temps, cela ralentirait l'exécution de notre script de supervision des stations. Pour gérer ce cas de figure, il vaudrait mieux produire un message dans un nouveau topic ("`velib-empty-stations`") et créer un second consumer ("`velib-monitor-empty-stations.py`") qui serait en charge de lire les messages et d'envoyer les e-mails correspondants.

We're gonna need a bigger boat

Pour l'instant, le schéma de fonctionnement de notre application est le suivant :



Un producer ajoute des message à un topic doté d'une seule partition et les messages sont récupérés par un unique consumer.

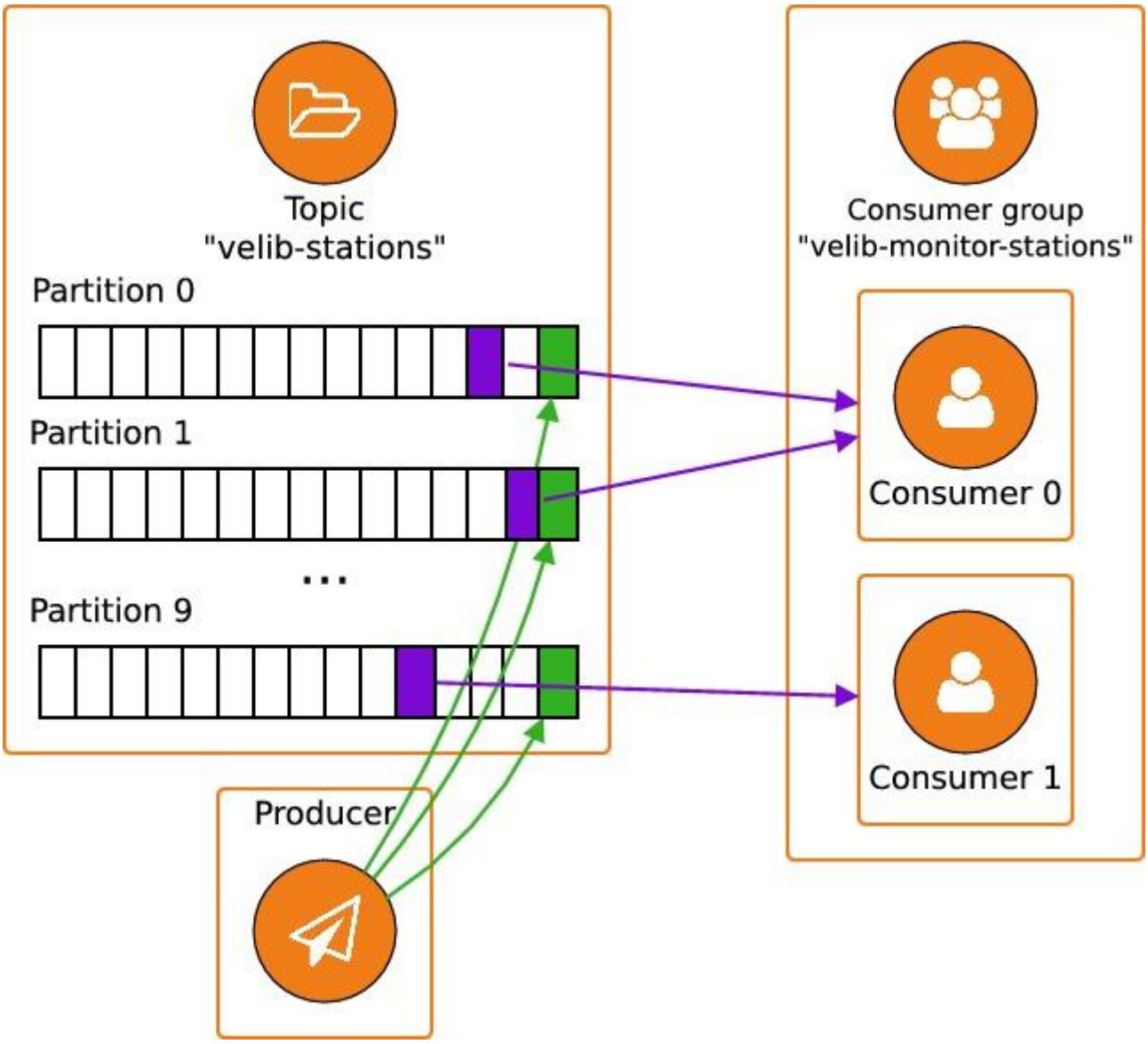
Pour pouvoir passer à l'échelle, comme on l'a décrit dans le chapitre précédent, on va vouloir augmenter le nombre de consumers, ce qui signifie mathématiquement qu'il va falloir augmenter le nombre de partitions de notre topic. On passe à 10 partitions à l'aide de la commande suivante :

```
$ ./bin/kafka-topics.sh --alter --zookeeper localhost:2181 --topic velib-stations --
partitions 10
```

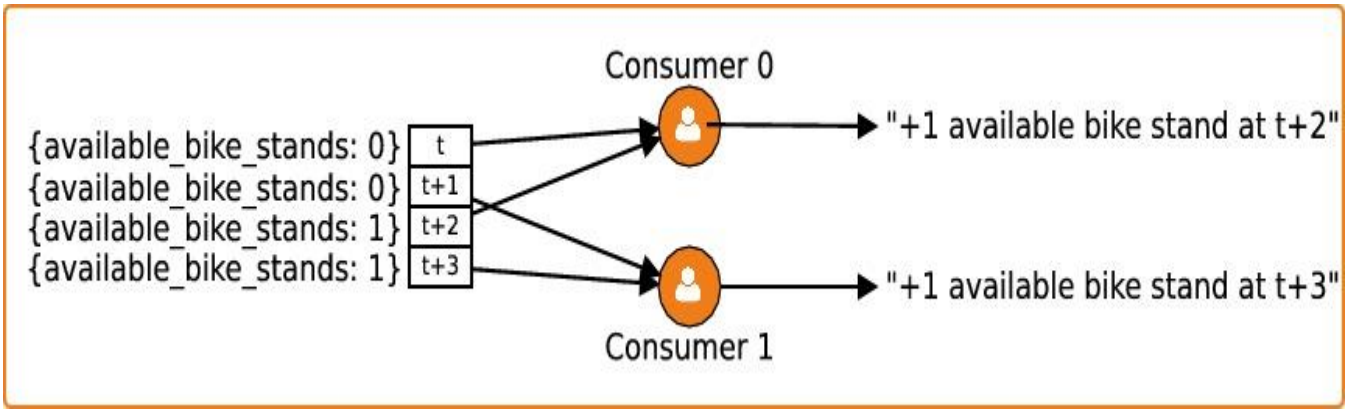
On peut alors lancer une seconde instance de consumer :

```
$ python velib-monitor-stations.py
```

Nous avons fait évoluer notre application pour obtenir le schéma de fonctionnement suivant :



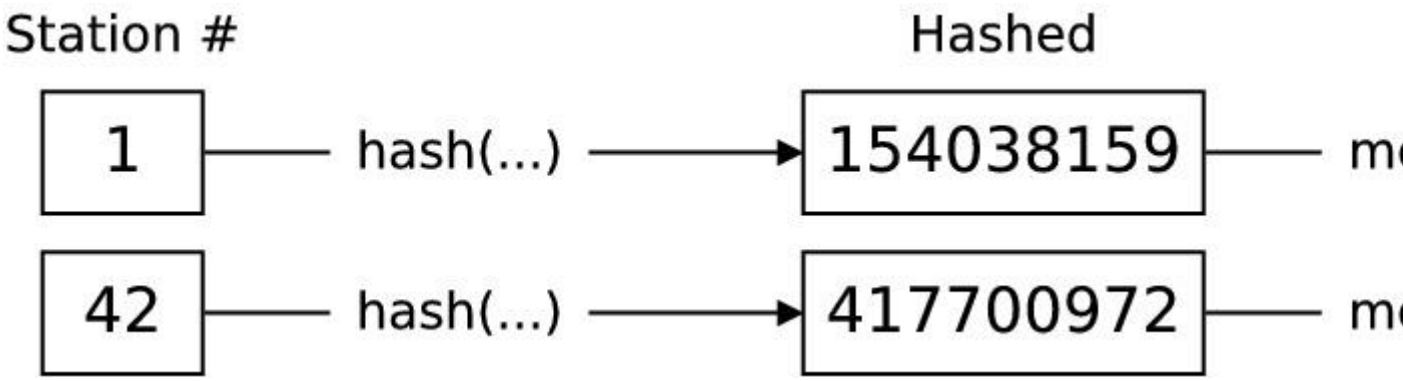
Une moitié des partitions est traitée par le premier consumer, et l'autre moitié est traitée par le second consumer. Le problème qu'on rencontre dans cette nouvelle architecture, est que les deux consumers sont également susceptibles de recevoir les messages qui concernent une même station. Par exemple, imaginons qu'une station dispose de 0 emplacement libre à l'instant t et qu'elle passe à 1 emplacement libre entre les instants $t+1$ et $t+2$. Les messages sont envoyés alternativement aux deux consumers :



Si l'on regarde les logs des deux consumers, on aura l'impression que le nombre d'emplacements libres a changé deux fois, puisque chacun des consumers aura émis une ligne de log (à $t+2$ et $t+3$ respectivement). Pour résoudre ce problème, il faut envoyer tous les messages qui concernent la même station au même consumer, donc à la même partition. Pour cela, on va modifier l'appel à `send(...)` du producer. Pour chaque message, la partition auquel il sera envoyé sera une fonction de l'identifiant de la station :

```
producer.send(..., key=str(station["number"]).encode())
```

Pour chaque station, le producer va réaliser un hash de l'identifiant de la station, modulo le nombre de partitions disponibles, et envoyer le message à la partition dont l'index est égal au résultat :



Ceci nous garantit que tous les messages qui concernent une même station seront traités par le même consumer.

Optimisation de la rétention

L'augmentation du nombre de consumers nous permet de faire passer à l'échelle le traitement des messages. Mais on ne va pas toujours avoir les ressources nécessaires pour augmenter le nombre de consumers. Surtout que dans notre application, il y a un grand nombre de messages qui sont redondants : pour être précis, les messages sont rendus obsolètes à chaque appel à l'API. Il est donc raisonnable d'envisager de supprimer les messages trop vieux. Il est possible de modifier la configuration de notre topic pour forcer la suppression des messages au bout de, par exemple, quatre secondes :

```
$ ./bin/kafka-configs.sh --zookeeper localhost:2181 --entity-type topics --entity-name
velib-stations --alter --add-config retention.ms=4000
```

En modifiant le paramètre `retention.ms`, on demande à Kafka d'effacer un segment de données toutes les quatre secondes. Un segment est une succession de messages dans une partition. Par défaut, un nouveau segment est créé chaque semaine, et dès que la quantité de messages dépasse 1 Go. Pour que la nouvelle valeur du paramètre `retention.ms` soit effective, il faut donc diminuer la longueur maximale d'un segment :

```
$ ./bin/kafka-configs.sh --zookeeper localhost:2181 --entity-type topics --entity-name
velib-stations --alter --add-config segment.ms=2000
```

Après avoir modifié les paramètres`retention.ms`et`segment.ms`de notre topic, les messages seront bien effacés toutes les quatre secondes.