Correction : Redis

L'objectif de cet exercice est d'implémenter une API avec le framework FastAPI. A partir d'un `id_vehicule`, cette API renvoit l'ensemble des informations associées au véhicule. Les données proviennent de la table `vehicules`. Il faudra implémenter un système de cache si les données renvoyées sont identiques.

Exercice 1 : Installer et tester Redis via la CLI

1. Installer `redis`. Cette base de données ne fait pas partie du catalogue Onyxia. Pour l'installer, lancer un terminal puis lancer la commande :

```
kubectl run redis --image redis && kubectl expose pod/redis --port 6379
```

2. Installer la `cli` avec `sudo apt update && sudo apt install redis`
3. Se connecter à `redis` avec la commande `redis-cli -h redis` et créer une clé de type hash `AA-000-AA` avec comme sous clé `marque Mercedes`, `modele Classe A` et `annee 2020` pour vérifier que Redis fonctionne.

Lancer la commande suivante :

```
HSET AA-000-AA marque Mercedes modele "Classe A" annee 2020
```

> HSET permet de créer une entrée de type Hash où `AA-000-AA` est la clé et ensuite on y ajoute un couple de sous-clé et valeurs

Exercice 2 : lire les données de la base de données

1. Faire une fonction python qui lit les données de la table du précédent exercice avec la méthode `pd.read_sql` et qui renvoit le DataFrame. Appliquer avant de retourner le DataFrame la transformation suivante pour retirer les caractères spéciaux :

```
df.replace(r"\xa0", "", regex=True)
```

```
import sqlalchemy
import pandas as pd

con =
sqlalchemy.engine.create_engine("postgresql+psycopg2://user:mdp@postgresql
-xxx/defaultdb")
```

```
def read_data(con):
    df = pd.read_sql("vehicules", con=con)
    df = df.replace(r"\xa0", '', regex=True)
    return df

df = read_data(con=con)
```

2. Pourquoi cette méthode de lecture de la base n'est pas recommandée dans le cadre d'une utilisation via une API ?

> Cette méthode lit l'entiereté de la base de données, il est préférable d'utiliser un connecteur SQL qui lance la requête directement

Exercice 3 : filtrer les données

1. Faire une fonction qui prend pour entrée un DataFrame et un `id_vehicule` et renvoyer la première ligne correspondante sous la forme d'un dictionnaire python avec la méthode `to_dict`.
2. Tester la fonction avec les `id_vehicule` suivants : `813953`, `8000000`. Que se passe t'il ?
3. Modifier cette fonction afin de renvoyer un dictionnaire vide si aucune ligne n'est trouvée

```
def get_vehicule(df, vehicule_id: str):
    row = df[df["id_vehicule"] == vehicule_id]
    if len(row):
        dict_row = row.iloc[0].to_dict()
        return dict_row
    return {}


df = read_data(con=con)
get_vehicule(df=df, vehicule_id='813953')
get_vehicule(df=df, vehicule_id='8000000')
```

Exercice 4 : créer une API

1. A partir de la documentation de FastAPI créer l'api d'exemple dans un fichier `app.py` et la lancer avec la commande suivante dans un nouveau terminal linux :

Dans un fichier `app.py`

```
from typing import Union
from fastapi import FastAPI

app = FastAPI()


@app.get("/")
```

```
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: Union[str, None] = None):
    return {"item_id": item_id, "q": q}
```

```
fastapi dev app.py
```

> Il faudra au préalable installer fastapi

2. Tester un appel à cette api avec la commande `curl http://localhost:8000`

> curl est un outil pour lancer des requêtes HTTP

3. Modifier cette api pour prendre en paramètre le `id_vehicule` sous la forme `/vehicule/{id_vehicule}`

```
from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/vehicule/{vehicule_id}")
def read_item(vehicule_id: str):
    return {"vehicule_id": vehicule_id}
```

4. Ajouter les fonctions précédentes de recherche de données pour que l'API renvoit les informations du véhicule recherché. Ensuite, dans un autre terminal, tester le bon fonctionnement avec la commande :

```
from fastapi import FastAPI
import sqlalchemy
import pandas as pd

app = FastAPI()

con =
sqlalchemy.engine.create_engine("postgresql+psycopg2://user:mdp@postgresql
-xxx/defaultdb")
```

```
def get_vehicule(df, vehicule_id: str):
    row = df[df["id_vehicule"] == vehicule_id]
    if len(row):
        dict_row = row.iloc[0].to_dict()
        return dict_row
    return {}

def read_data(con):
    df = pd.read_sql("vehicules", con=con)
    df = df.replace(r"\xa0", '', regex=True)
    return df

@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/vehicule/{vehicule_id}")
def read_item(vehicule_id: str):
    df = read_data(con=con)
    data = get_vehicule(df=df, vehicule_id=vehicule_id)
    return data
```

```
curl http://localhost:8000/vehicule/813952
```

Exercice 5 : utiliser Redis avec Python

1. Installer le client python Redis

```
pip install redis
```

2. Dans un notebook, créer une connexion python à Redis et ajouter une clé `vehicule:813952` de type hash avec les informations correspondant à l'`id_vehicule` suivant : `813952`

```
r = redis.Redis(host='redis', port=6379, decode_responses=True)

info = get_vehicule(df=df, vehicule_id='813952')
r.hset("vehicule:813952", mapping=info)
```

3. Lire ensuite le hash précédent dans sa totalité et vérifier que les données sont correctes

```
r.hgetall("vehicule:813952")
```

4. Ajouter une expiration à la clé précédente de 60 secondes

```
r.expire("vehicule:813952", 60)
```

Exercice 6 : intégrer Redis à une API

1. Pour chaque nouvel appel à l'API, ajouter les données dans Redis, seulement si elles n'existent pas
2. Pour chaque nouvel appel, ajouter une vérification de l'id dans Redis avant de lire la données dans la base de données
3. Ajouter un TTL de 60 secondes lors de l'ajout des données dans le cache

```
import sqlalchemy
import pandas as pd
from fastapi import FastAPI
import redis

app = FastAPI()

con =
sqlalchemy.engine.create_engine("postgresql+psycopg2://user:mdp@postgresql
-xxx/defaultdb")
r = redis.Redis(host='redis', port=6379, decode_responses=True)

@app.get("/vehicule/{vehicule_id}")
def read_vehicule(vehicule_id: str):
    info = r.hgetall(vehicule_id)
    if "error" in info:
        return {}

    if "id_vehicule" in info:
        return info

    df_temp = pd.read_sql("vehicules", con=con)
    df_temp = df_temp.replace(r"\xa0", '', regex=True)
    row = df_temp[df_temp["id_vehicule"] == vehicule_id]

    if len(row):
        dict_row = row.iloc[0].to_dict()
        r.hset(vehicule_id, mapping=dict_row)
        r.expire(vehicule_id, 60)
        return dict_row

    r.hset(vehicule_id, mapping={"error": "introuvable"})
    r.expire(vehicule_id, 60)
    return {}
```

Ce script permet de lancer une API qui écoute sur la route `/vehicule/{vehicule_id}` afin de renvoyer les informations du véhicule demandé.

- Vérifie que le véhicule n'est pas dans le cache, sinon renvoit les données depuis ce cache
  - Si la clé est `error` renvoit un dictionnaire vide (aucun véhicule trouvé pour cet id)
  - Si `id_vehicule` est dans les informations, renvoit le dictionnaire complet
- Lit la table `sql`
- Retourne la ligne correspondant à l'id recherché
  - Si un id est trouvé, ajoute les données dans le cache et lui applique un cache de 60 secondes
  - Si aucun id n'est trouvé, ajoute dans le cache la clé du véhicule recherché et comme sous clé `error`

Quelques notes :

- la méthode `hset`ne permet pas d'écrire un dictionnaire vide. Un contournement possible était donc de créer une clé spécifique si le véhicule n'existait pas et vérifier si cette clé est dans le cache
- Une autre méthode serait d'utiliser `set` et d'écrire le dictionnaire au format `string` dans redis, puis de le sérialisé en `json` lors de la lecture.

1. Dans un terminal lancer `fastapi dev app.py` pour lancer l'API
2. Dans un autre terminal, tester un appel avec la commande `curl http://localhost:8000/vehicule/813952`. Si on rappelle ensuite l'API sur le même id, les données seront renvoyés en moins de temps

Autre méthode possible avec une librairie tierce

1. Installer les différentes dépendances

```
pip install "fastapi[standard]" sqlalchemy fastapi_redis_cache
```

2. Créer un fichier `app2.py` avec le contenu suivant (remplacer les `xxx` par les informations de votre environnement) :

```
from fastapi import FastAPI, Request, Response
from fastapi_redis_cache import FastApiRedisCache, cache
from sqlalchemy.orm import Session
import sqlalchemy
import pandas as pd

con =
sqlalchemy.engine.create_engine("postgresql+psycopg2://user:mdp@postgresql
-xxx/defaultdb")
app = FastAPI(title="FastAPI")

@app.on_event("startup")
def startup():
    redis_cache = FastApiRedisCache()
    redis_cache.init(
```

```
        host_url="redis://redis:6379",
        prefix="myapi-cache",
        response_header="X-API-Cache",
        ignore_arg_types=[Request, Response, Session]
    )

@app.get("/vehicule/{vehicule_id}")
@cache(expire=60)
def read_vehicule(vehicule_id: str, request: Request, response: Response):
    df_temp = pd.read_sql("vehicules", con=con)
    df_temp = df_temp.replace(r"\xa0", '', regex=True)
    row = df_temp[df_temp["id_vehicule"] == vehicule_id]
    if len(row):
        dict_row = row.iloc[0].to_dict()
        return dict_row
    return {}
```

Ce script permer de lancer une API qui écoute sur la route `/vehicule/{vehicule_id}` afin de renvoyer les informations du véhicule demandé. Il utilise un `middleware` utilisable par un système de décorateur pour ajouter dans le cache. La documentation est disponible ici

3. Dans un terminal lancer `fastapi dev app2.py` pour lancer l'API
4. Dans un autre terminal, tester un appel avec la commande `curl http://localhost:8000/vehicule/813952`. Si on rappelle ensuite l'API sur le même id, les données seront renvoyés en moins de temps